Rational. Tau

IBM

ユーザー ガイド

著作権

本書は、IBM Rational Tau バージョン 4.3 および新しい版で明記されていない限り、以降のすべてのリリースおよびモディフィケーションに適用されます。

本書は米国 IBM が提供する製品およびサービスについて作成したものであり、本書に記載の製品、サービス、または機能が日本においては提供されていない場合があります。日本で利用可能な製品、サービス、および機能については、日本 IBM の営業担当員にお尋ねください。本書で IBM 製品、プログラム、またはサービスに言及していても、その IBM 製品、プログラム、または サービスのみが使用可能であることを意味するものではありません。これらに代えて、IBM の知的所有権を侵害することのない、機能的に同等の 製品、プログラム、またはサービスを使用することができます。ただし、IBM 以外の製品とプログラムの操作またはサービスの 評価および検証は、お客様の責任で行っていただきます。

Copyright © 2000, 2009 by IBM Corporation.

IBM は、本書に記載されている内容に関して特許権 (特許出願中のものを含む) を保有している場合があります。本書の提供は、お客様にこれらの特許権について 実施権を許諾することを意味するものではありません。実施権についてのお問い合わせは、書面にて下記宛先にお送りください。

〒 106-8711
東京都港区六本木 3-2-12
日本アイ・ビー・エム株式会社
法務・知的財産
知的財産権ライセンス渉外

以下の保証は、国または地域の法律に沿わない場合は、適用されません。IBM およびその直接または間接の子会社は、本書を特定物として現存するままの状態で提供し、商品性の保証、特定目的適合性の保証および法律上の瑕疵担保責任を含むすべての明示 もしくは黙示の保証責任を負わないものとします。国または地域によっては、法律の強行規定により、保証責任の制限が禁じられる場合、強行規定の制限を受けるものとします。

この情報には、技術的に不適切な記述や誤植を含む場合があります。本書は定期的に見直され、必要な変更は本書の次版に組み込まれます。IBM は予告なしに、随時、この文書に記載されている製品またはプログラムに対して、改良または変更を行うことがあります。

本書において IBM 以外の Web サイトに言及している場合がありますが、便宜のため記載しただけであり、決してそれらの Web サイトを推奨するものではありません。それらの Web サイトにある資料は、この IBM 製品の資料の一部ではありません。それらの Web サイトは、お客様の責任でご使用ください。

IBM は、お客様が提供するいかなる情報も、お客様に対してなんら義務も負うことのない、自ら適切と信ずる方法で、使用もしくは配布することができるものとします。

本プログラムのライセンス保持者で、(i) 独自に作成したプログラムと その他のプログラム (本プログラムを含む) との間での情報交換、および (ii) 交換された情報の相互利用を可能にすることを目的として、本プログラムに関する情報を必要とする方は、製造元に連絡してください。

Intellectual Property Dept. for Rational Software|
IBM Corporation
1 Rogers Street
Cambridge, Massachusetts 02142
U.S.A.

本プログラムに関する上記の情報は、適切な使用条件の下で使用することができますが、有償の場合もあります。

本書で説明されているライセンス・プログラムまたはその他のライセンス資料は、IBM 所定のプログラム契約の契約条項、IBM プログラムのご使用条件、またはそれと同等の条項に基づいて、IBM より提供されます。

この文書に含まれるいかなるパフォーマンス・データも、管理環境下で決定されたものです。そのため、他の操作環境で得られた結果は、異なる可能性があります。一部の測定が、開発レベルのシステムで行われた可能性がありますが、その測定値が、一般に利用可能なシステムのものと同じである保証はありません。さらに、一部の測定値が、推定値である可能性があります。実際の結果は、異なる可能性があります。お客様は、お客様の特定の環境に適したデータを確かめる必要があります。

IBM 以外の製品に関する情報は、その製品の供給者、出版物、もしくはその他の公に利用可能なソースから入手したものです。IBM は、それらの製品のテストは行っておりません。したがって、他社製品に関する実行性、互換性、またはその他の要求については確証できません。IBM 以外の製品の性能に関する質問は、それらの製品の供給者にお願いします。

本書には、日常の業務処理で用いられるデータや報告書の例が含まれています。より具体性を与えるために、それらの例には、個人、企業、ブランド、あるいは製品などの名前が含まれている場合があります。これらの名称はすべて架空のものであり、名称や住所が類似する企業が実在しているとしても、それは偶然にすぎません。

この情報をソフトコピーでご覧になっている場合は、写真やカラーの図表は表示されない場合があります。

サンプルコードの著作権

本書には、様々なオペレーティング・プラットフォームでのプログラミング手法を例示するサンプル・アプリケーション・プログラムがソース言語で掲載されています。お客様は、サンプル・プログラムが書かれているオペレーティング・プラットフォー

ムのアプリケーション・プログラミング・インターフェースに 準拠したアプリケーション・プログラムの開発、使用、販売、配布を目的として、いかなる形式においても、IBM に対価を支払うことなくこれを複製し、改変し、配布することができます。このサンプル・プログラムは、あらゆる条件下における完全なテストを経ていません。従って IBM は、これらのサンプル・プログラムについて信頼性、利便性もしくは機能性が あることをほのめかしたり、保証することはできません。

それぞれの複製物、サンプル・プログラムのいかなる部分、またはすべての派生的創作物にも、次のように、著作権表示を入れていただく必要があります。

© (お客様の会社名) (西暦年). このコードの一部は、IBM Corp. のサンプル・プログラムから取られています。© Copyright IBM Corp._(西暦年)

商標

IBM、IBM ロゴ、ibm.com は、International Business Machines Corporation の米国およびその他の国における商標または登録商標です。これらおよびその他の IBM 商標に、この情報の最初に現れる個所で商標表示（® または ™）が付されている場合、これらの表示は、この情報が公開された時点で、米国において、IBM が所有する登録商標またはコモン・ロー上の商標であることを示します。このような商標は、その他の国においても登録商標またはコモン・ロー上の商標である可能性があります。現時点の IBM の商標リストについては、**www.ibm.com/legal/copytrade.shtml** をご覧ください。

Adobe、Adobe ロゴ、PostScript は、Adobe Systems Incorporated の米国およびその他の国における登録商標または商標です。

Intel、Intel（ロゴ）、Intel Inside、Intel Inside（ロゴ）、Intel Centrino、Intel Centrino（ロゴ）、Celeron、Intel Xeon、Intel SpeedStep、Itanium、Pentium は Intel Corporation または子会社の米国およびその他の国における商標または登録商標です。

Linux は、Linus Torvalds の米国およびその他の国における商標です。

Microsoft、Windows、Windows 2003、Windows XP、Windows Vista および / またはその他の Microsoft 製品は、Microsoft Corporation の米国およびその他の国における商標または登録商標です。

ITIL は英国 Office of Government Commerce の登録商標および共同体登録商標であって、米国特許庁に登録されています。

Pentium は、Intel Corporation の商標です。

UNIX は、The Open Group の米国およびその他の国における登録商標です。

Cell Broadband Engine, Cell/B.E は、米国およびその他の国における Sony Computer Entertainment, Inc. の商標であり、同社の許諾を受けて使用しています。

Java およびすべての Java 関連の商標およびロゴは、Sun Microsystems, Inc. の米国およびその他の国における商標です。

他の会社名、製品名およびサービス名等はそれぞれ各社の商標です。

ツールの紹介

Tau 4.3 は、サービス指向アーキテクチャなどの最先端ソフトウェア システムの開発を目的としたツールです。Tau 4.3 には、UML を使用してアプリケーションのモデリングを行うための豊富な機能が備わっています。UML については UML 言語ガイド を参照してください。

Tau の機能を最大限に活用しつつ迅速に作業を立ち上げるには、製品とともにインストールされるチュートリアルマニュアル（『UML チュートリアル』、『Java チュートリアル』など）が役立ちます。

Tau 4.3 にはサービス指向アーキテクチャアプリケーション実現のためのさまざまな機能が用意されています。以下の章を参照してください。

- Web サービスサポート Tau 4.3 がサポートする Web サービスのモデリング機能を紹介します。
- XML スキーマモデリング Web サービスの使用する XML データのモデルリング方法を説明します。
- Tau と System Architect の協調 Tau 4.3 を System Architect とともに使用してエンタープライズアーキテクチャ分析とサービス指向アーキテクチャの統合する方法について説明します。
- WSDL/XSD インポータリファレンス既存のサービス記述を WSDL や XSD にインポートする方法と UML モデルから WSDL や XSD を生成する方法について説明します。

さらに、以下の章にも便利な情報があります。

- 便利なショートカット キー ショートカット キーのリストです。Tau の機能に習熟するとともにショートカットキーを使うようにすると、より速く効率的な操作が可能になります。
- ツール環境の設定 Tau を構成管理ツールおよび要件管理ツールと統合した形でセットアップする方法について説明します。

1

Tau 4.3 の紹介

UML

Tau には、UML 1.x と下位互換性のある UML 2.0 を基にしたモデル駆動型のツール セットがあります。以下のダイアグラム タイプをサポートします。

- ユース ケース図
- シーケンス図
- 状態機械図
- アクティビティ図
- 相互作用概観図
- クラス図
- パッケージ図
- コンポーネント図
- 配置図
- 合成構造図
- テキスト図（UML 構文）

モデル ベリファイヤ（Model Verifier）を使用してリアルタイムの振る舞いのシミュレーションを行い、自分の実装を簡単にベリファイできます。

UML ツール セットは、Java コードにも対応しています。UML と Java はよく似た言語なので、テキスト図とテキスト シンボル内で 2 つを切り替えて使用できます。Tau での Java サポートと Eclipse インテグレーションは、Windows OS でのみサポートされます。

C コードまたは C++ コードの生成を完全にサポートします。また、最も一般的な OS とリアルタイム OS のためにターゲットを統合するためのコード生成もサポートします。

UML ですぐに作業を開始できるようになるために、以下の情報が役立ちます。

- ワークフローの説明
  プロジェクトのさまざまな段階で UML ツールを使用するためのロードマップを提供します。
- モデルの操作
  モデル ベースの開発の基礎について説明します。手順と概要情報を提供します。
- UML 言語ガイド
  UML 言語について説明します。
- UML チュートリアル
  サポートされるダイアグラムを使用して作業を行うためのチュートリアルです。作成したモデルをベリファイする方法についても学びます。
- UML クイック リファレンス ガイド
  グラフィックとテキストの UML で一般的に使用されている構成要素の例を示します。

Tau ユーザー インターフェイスの概要

全体の Tau ユーザー インターフェイスには、**デスクトップ**、**ワークスペース** ウィンドウ、**ショートカット** ウィンドウ、および出力ウィンドウがあり、必要に応じて表示／非表示を切り替えられます。さらに、フレームの下部にはステータス バー、上部にはメニュー バーとツール バーがあります。ウィンドウはすべて必要に応じてドッキングできます。よく使用するツールバーをショートカット ウィンドウにドラッグアンドドロップすることもできます。

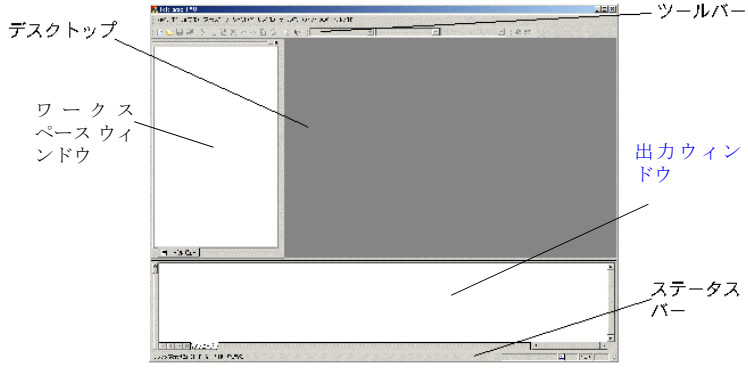


図 1: Tau のデスクトップ

デスクトップ

デスクトップは編集領域ともいい、作業中のドキュメントが表示される場所です。ここで実際の開発を行います。編集または表示のために開かれたダイアグラム、ドキュメント、ソース ファイルは、デスクトップに表示されます。表示されるエディタまたはビューアのタイプは、プロジェクトに含まれるファイルのタイプによって異なります。

デスクトップに 2 つ以上のドキュメントが開かれている場合、［ウィンドウ］メニューのコマンドを使用するか、Ctrl + Tab キー（前面へ）または Ctrl + Shift + Tab キー（背面へ）を押してウィンドウ間を移動できます。

ヒント

エディタを全画面表示にするには、［表示］メニューから［全画面表示］を選択します。元の画面表示に戻すには、Esc キーまたは Alt + 1 キーを押します。

参照

ウィンドウの操作

ワークスペース ウィンドウ

ワークスペース ウィンドウは、ワークスペース情報の構造を複数のビューとして表示し、管理するためのグラフィック ツールです。

ワークスペース ウィンドウには、展開可能なノードを含んだ情報構造が表示されます。これらのノードを展開／折りたたんだり、または、他のビューを使用することにより、情報を必要な分に絞り込んだ形でワークスペースに表示できます。

ワークスペース ウィンドウを移動して、フローティング パレットとして使用できます。フローティング パレットとして使用しない場合は、左端の位置にドッキングします。ドッキング状態では、ウィンドウは水平方向にのみサイズ変更できます。上下の境界は、上はツール バー、下は出力ウィンドウによって決定されます。

編集マーカー

グレー バー

［ファイル ビュー］と［モデル ビュー］では、編集できない要素の要素シンボルと要素名の間にグレー バーが表示されます。

レッド バー

［モデル ビュー］でモデルを編集すると、要素シンボルと要素名の間に、現行の作業セッションで変更されたことを示すレッド バーが表示されます。

アスタリスク

テキスト ファイルの編集を行ってまだ保存していない場合、ウィンドウ タイトルバーのファイル名の隣にアスタリスクが表示されます。

ビュー

ワークスペースにはさまざまなビューを表示できます。各ビューへは対応するタブからアクセスできます。1 つのビューはモデルの 1 つの局面を表示しています。

ファイル ビュー

［ファイル ビュー］では、ファイルとして表現されるすべての要素がワークスペースに表示されます。これらのファイルは、プロジェクト ファイル、UML モデル、テキスト ファイル、およびプロジェクト内に保管されているファイルです。ダイアグラムやクラスなどはファイルとしては表現されないため、ここには表示されません。［ファイル ビュー］に表示されるファイルには、モデルのテキスト表現が含まれています。

［ファイル ビュー］を表示するには、**ワークスペース ウィンドウ**の［ファイル ビュー］タブをクリックします。このビューで、すべてのファイルを開き、編集し、保存できます。ただし、ここでファイルを削除しても、ファイルは［ファイル ビュー］から削除されるだけで、OS のファイル システムからは削除されません。

フォルダを作成して複数のファイルを容易に管理できます。ファイルをフォルダに入れるには、そのファイルを［ファイルビュー］でドラッグ アンド ドロップします。ファイルのプロパティを表示するには、右クリックし、表示されるショートカット メニューから［プロパティ］を選択します。

モデル ビュー

［モデル ビュー］には、作業対象のすべてのデータが「抽象的な構造」として表示されます。すべての UML 要素は、このビューに表示されます。モデルに要素を追加してダイアグラムを作成するには、このビューを使用します。

このビューに表示される要素は、モデルの図形表現と見なされます。設計プロセスでは、ダイアグラム エディタを使ってもかまいませんが、［モデル ビュー］のノードについて作業するだけでも簡単にシステム全体を設計できます。

モデル ビューを表示するには、**ワークスペース** ウィンドウの［モデル ビュー］タブをクリックします。

プロジェクト ノードまたはモデル ノードのショートカット メニューには、［モデル ビュー フィルタ］というサブメニューがあります。このサブメニューをチェックすると、定義済みフィルタを［モデル ビュー］に適用できます。

メタモデルに**Resource**を基底とする 1 つのメタクラスが含まれている場合があります。このモデル要素は、実行時にロードされた要素内のリソース要素に対応付けられ、［ファイルの表示］モデル ビュー フィルタが有効になっている場合のみ［モデル

ビュー］内で表示されます。［モデル ビュー］にファイルとリソース要素を表示するには、［モデル ビュー フィルタ］ショートカット サブメニューから［ファイルの表示］を選択します。

メタモデルに Diagram のサブクラスであるオブジェクト モデル クラスを基底とするメタクラスが含まれている場合があります。このモデル要素は、ダイアグラムに対応付けられ、［ダイアグラムの表示］モデル ビュー フィルタが有効になっている場合のみ［モデル ビュー］内で表示されます。

便宜上、メタクラスは「構造」エンティティと「詳細」エンティティに分類されます。これは、［詳細の表示］フィルタを適用したときの［モデル ビュー］表示内容に影響があります。

メタモデルに Implementaion のサブクラスであるオブジェクト モデル クラスを基底とするメタクラスが含まれている場合があります。このモデル要素は、実装指向の要素に対応付けられ、［実装の表示］モデル ビュー フィルタが有効になっている場合のみ［モデル ビュー］内で表示されます。

［定義のソート］フィルタを有効にすると、そのモデル ビュー ノードの要素は辞書順でソートされて表示されます。ソート操作は、ダイアグラム ノードそのものには影響を及ぼしません。

［表示］メニューから［モデル ビューの再構成］コマンドを使用して、［モデル ビュー］に表示する情報を定義済みメタモデルに基づいて絞り込めます。この操作は、選択した要素が属するプロジェクトに影響を及ぼします。

［Standard View］以外のフィルタを選択した場合に［モデル ビュー］内のノードが「消える」ことがあります。これは、Tau の操作に、UML 情報の格納に使用する基本的なメタモデルに依存する操作（ドラッグ アンド ドロップなど）と、現在選択されているメタモデルに依存する操作（［モデル ビュー］でツリー内の要素を表示するなど）があることに由来します。表示を［Standard View］に切り替えると、「消える」ことはありません。このモデルが基本的なメタモデルと同じだからです。一方［ダイアグラム ビュー］は、ダイアグラムを所有できるモデル要素を基本に情報を表示し、ダイアグラム ノードを所有者要素の直下に配置します。

以下の操作を行うと、この状態が発生することがあります。

- ドラッグ アンド ドロップ
- 切り取りと貼り付け
- モデル ナビゲータの［ダイアグラム］タブによるダイアグラムの作成

上のように「消えた」ノードを復元するには、以下の 2 通りの方法があります。

- ［元に戻す］機能を使用してノードを元の場所に戻す。
- 表示を［Standard View］に切り替える。この表示ではすべてのノードが表示されます。［モデル ビュー］でドラッグ アンド ドロップで移動した場所に表示されなかったノードが、新しい場所に表示されるようになります。

インスタンス

［インスタンス］ビューはワークスペース ウィンドウのタブで、モデル ベリファイヤ（Model Verifier）が起動している場合のみ選択できます。［インスタンス］タブには以下のものが表示されます。

- 実行中のエージェント。継承とインスタンスがフラット化されて表示されます。
- 現在有効なエージェント。インスタンスとそのインスタンス番号および属性とともに表示されます。
- カーネル依存オブジェクト、キューおよびシステム環境エージェント。

［インスタンス］ビューを表示するには、**ワークスペース** ウィンドウの［インスタンス］タブをクリックします。

ファイル

ファイルは、ワークスペース ウィンドウの［ファイル ビュー］にアイコンで表示されます。

任意のファイルがプロジェクトに挿入できます。挿入されたファイルは、ワークスペース ウィンドウの［ファイル ビュー］にアイコンとして表示されます。このファイル アイコンをダブルクリックすると、そのファイルの表示用または編集用に関連付けられたプログラムが起動します。

他の UML ファイルを自分のモデルのフォルダに挿入できます。この方法でモデル間で情報の共有と移動ができます。

ショートカット ウィンドウ

ショートカット ウィンドウにツール バーを表示するように設定できます。ショートカット ウィンドウにツール バーを表示するには、ツール バーを右クリックし、ショートカット メニューから［ショートカット］を選択します。ツール バーを元の位置に戻すときは、ショートカット ウィンドウにあるツール バーを右クリックし、ショートカット メニューから［ツール バー］を選択します。

ショートカット ウィンドウの表示と非表示は、［表示］メニューの［ショートカット］コマンドで切り替えられます。

注記

一部のツール バーはショートカット ウィンドウに表示できません。

出力ウィンドウ

出力ウィンドウは、各種ツールの情報を記録、表示するための複数のタブから構成されます。たとえば、エラー メッセージ、警告、アクションの実行結果、イベントのログなどです。ツールごとに個別のタブがあります。

タブによっては、サブジェクト カラム内の要素からダイアグラム上の要素までナビゲートできるものがあります。

出力ウィンドウの表示と非表示は、［表示］メニューの［出力］コマンドで切り替えられます。

一般的なタブ

メッセージ

［メッセージ］タブには、プロジェクトの読み込みやその他の実行されたアクションに関する情報が表示されます。

このタブからツールの他の部分へのナビゲートはできません。

検索結果

このタブには、検索操作の結果が表示されます。

プレゼンテーション

このタブには、プレゼンテーションの一覧表示操作の結果が表示されます。

参照

このタブには、参照の一覧表示操作の結果が表示されます。

スクリプト

［スクリプト］タブには、Tcl スクリプトなどのスクリプトの実行結果が表示されます。

UML ツール用のタブ

チェック

モデルの完全なチェックを開始してエラーと警告を検出できます。このタブには、チェックの結果が表示されます。エラーは修正後もリストに残ります。もう一度チェック手順を実行するとこのリストが変更されます。

オートチェック

現在作業中のスコープでエラーを自動チェックする機能があります。スコープは、デスクトップに何のダイアグラムが開かれているかで決まります。このタブには、検出されたエラーが表示されます。修正されたエラーはリストから削除されます。

ビルド

［ビルド］タブには、セマンティック チェックと構文チェック、コード生成など、ビルド プロセスの結果が表示されます。アプリケーションをビルドする際、警告メッセージとエラー メッセージが、ここに表示されます。エラー メッセージまたは警告から、適切なエディタのソースに直接ナビゲートできます。

ナビゲート

このタブには、表形式のモデル ナビゲーション ツールであるモデル ナビゲータがあります。このツールは、モデル内のナビゲートに使用します。

Model Verifier

このタブには、モデルの振る舞いを検証した際のテキスト トレースが表示されます。また、Model Verifier コンソールに入力したコマンドのログも表示されます。

ウィンドウの操作

ウィンドウの配置

すべてのドキュメントウィンドウを並べて表示するには

- ［ウィンドウ］メニューから［水平方向に並べて表示］または［垂直方向に並べて表示］をクリックします。

すべてのドキュメントウィンドウを重ねて表示するには

- ［ウィンドウ］メニューから［重ねて表示］をクリックします。

ドキュメント ウィンドウを移動するには

注記

タブ付きドキュメントのあるウィンドウはドッキング状態を変更できません。

1. ドキュメント ウィンドウのタイトル バーを右クリックします。
2. メニューから以下のコマンドを選択します。
   - **ドッキング済み**：ウィンドウをアプリケーション ウィンドウ内にドッキングします。ウィンドウをドッキングする場所を選択できます。
   - **フローティング**：アプリケーションの外側でウィンドウを移動できます。
   - **MDI 子ウィンドウ**：編集領域内でのみウィンドウを移動できます。ウィンドウを最大化、最小化、縮小（元に戻す）できます。

アクティブ ドキュメントを全画面表示するには

- ［表示］メニューから［全画面表示］をクリックします。
  または
- **Alt + 1** キーを押します。

アクティブ ドキュメントを標準サイズで表示するには

アクティブ ドキュメントを全画面表示から標準サイズに戻すには、以下のいずれかを行います。

- 画面の上部にカーソルを移動します。メニュー バーが表示されたら、［表示］メニューから［全画面表示］をクリックします。
  または
- **Alt + 1** キーを押します。

ウィンドウの表示と非表示

ワークスペース ウィンドウを表示／非表示するには

- ［表示］メニューから［ワークスペース］をクリックします。
  または
- **Alt +0** キーを押します。

出力ウィンドウを表示／非表示するには

- ［表示］メニューから［出力］をクリックします。
  または
- **Alt + 2** キーを押します。

ウィンドウを閉じる

ドキュメント ウィンドウを閉じるには

- ［ウィンドウ］メニューから［閉じる］をクリックします。

すべてのドキュメント ウィンドウを閉じるには

- ［ウィンドウ］メニューから［すべて閉じる］をクリックします。

新規ウィンドウの作成

新規ドキュメント ウィンドウを作成するには

- ［ウィンドウ］メニューから［新しいウィンドウ］をクリックします。

タブ付きドキュメント

［一般］オプション ページで［タブ付きドキュメント］オプションを選択すると、1 つのウィンドウに複数のドキュメントがタブ付きで表示されます。

タブを右クリックして［切り離す］をクリックすれば、タブ付きのウィンドウからドキュメントを独立できます。独立した状態では、通常の MDI 子ウィンドウと同じように機能し、ドッキング状態を変更できます。

ウィンドウのドッキング

Tau フレームワークには、エディタ ウィンドウに 3 種類のモードがあります。これらのモードは、ダイアグラム ウィンドウのタイトル バーを右クリックし、表示されたショートカット メニューで個々に設定できます。

注記

タブ付きドキュメントのあるウィンドウはドッキング状態を変更できません。

ドッキング済み

ドッキングされたエディタ ウィンドウは、ワークスペース ウィンドウや出力ウィンドウのように、Tau フレームワークに沿って組み込まれます。ウィンドウを移動して適切な表示にできます。

フローティング

フローティング ウィンドウは、Tau フレームワークの上側に配置されます。フレームワークのフレーム内に移動すると、ドッキング ウィンドウになります。

MDI 子ウィンドウ

MDI 子ウィンドウはデスクトップ領域に配置されます。この領域内で手作業で調整したり、［ウィンドウ］メニューのコマンドを使用して調整できます。

ドッキング ウィンドウの自動非表示（**Windows**）

グリッパ バーにピンが表示されているウィンドウは自動非表示モードに設定できます。ピンをクリックするウィンドウは非表示になり、そのウィンドウを表すラベルが表示されます。マウス カーソルをラベルに合わせると、隠れていたウィンドウが表示されます。ウィンドウをドッキングするには、ピンをもう一度クリックします。

ドッキング ウィンドウの拡大と縮小

2 つのドッキング ウィンドウがメイン ウィンドウの同じ辺を共有している場合、ウィンドウのグリッパ バー（利用可能な場合）の矢印をクリックすると、メイン ウィンドウの全面までウィンドウを拡大できます。これによって、同じ辺を共有していたもう 1 つのウィンドウは最小化されます。ウィンドウを元のサイズに戻すには、矢印をもう一度クリックします。

保存済みワークスペース ウィンドウ

セッション中に開かれていたすべてのウィンドウは、ワークスペースを再ロードすると、再び開かれます。情報は、ワークスペースと同じパスと名前で .ttx ファイルに保存されます。

参照

ビューの体系化

メニュー バーとツール バー

初めて Tau を起動すると、ツール バーはメニュー バーのすぐ下に表示されます。

ワークスペースの環境設定と画面サイズにより、必要なツール バーを表示したり、まったく表示しないように設定できます。必要に応じてツール バーにコマンド ボタンを追加したり、ボタンのサイズを変更したり、別の場所に移動できます。

メニュー バー

メニュー バーには［ファイル］、［編集］、［プロジェクト］などのなじみ深いメニューがあります。実行しているタスクによって、メニューの数が変わります。

ほとんどのメニュー コマンドにはショート カットが割り当てられています。全タイプ共通のリファレンス ガイドに便利なショートカット キーのリストがあります。

IBM Rational 以外のツールを簡単に使用できるようにするため、［ツール］メニューにコマンドを追加できます。この設定は、［ツール］タブを使用して行います。

たとえば、［ツール］メニューに Windows のメモ帳を追加する方法を以下に示します。

［ツール］メニューにコマンドを追加するには

1. ［ツール］メニューから［カスタマイズ］ダイアログを選択し、次に［ツール］タブをクリックします。
2. ［新規（挿入）］ボタンをクリックします。
3. ［ツール］メニューに表示したいツール名を入力し、Enter キーを押します。

   たとえば、Windows のメモ帳のコマンドを追加したい場合、「メモ帳」と入力します。
4. ［コマンド］ボックスにプログラムのパス（例：`C:¥Windows¥notepad.exe`）を参照または手動で入力します。
5. ［引数］テキストボックスに、プログラムに渡す引数を参照または手動で入力します。メモ帳アクセサリの場合、このフィールドは空のままにします。

注記

［引数］テキストボックスの隣のドロップダウン ボタンをクリックして、使用できる引数のリストを表示できます。リストから引数を選択し、［引数］テキストボックスに引数構文を挿入します。

6. ［初期ディレクトリ］ボックスに、コマンドの実行形式ファイルがあるファイル ディレクトリを指定します。メモ帳アクセサリの場合、このフィールドは空のままにします。

［ツール］メニューにコマンドが表示されたら、それをクリックしてプログラムを実行できます。

プログラムに渡す引数を［引数］テキストボックスに入力するか、プログラムの初期ディレクトリを［初期ディレクトリ］テキスト ボックスに入力して追加できます。

［ツール］メニューに追加するプログラムに `.pif` ファイルがある場合、［初期ディレクトリ］テキスト ボックスで指定されたディレクトリは、`.pif` ファイルで指定される起動ディレクトリに置き換わります。

ツール バー

ツール バーにより、頻繁に使用するツールをすばやく使用できるように、パレットに設定できます。ツール バーを変更すると、その変更は保存されて次の作業セッション時に反映されます。

標準のツール バーは、メニュー バーから使用可能な操作に対応しています。標準のツール バーは、[表示] メニューの [標準] コマンド、またはツール バー領域のショートカットメニューによって表示／非表示を切り替えられます。標準以外のツール バーはショートカットメニューによってのみ切り替えられます。

注記

一部のツール バーとコマンドは修正できません。この機能は、Tau フレームワークに属する機能であり、エディタに関するツール バーには対応していません。

ツール バーのボタンを追加するには

1. 変更しようとするツール バーが表示されていることを確認します。
2. [ツール] メニューから [カスタマイズ] ダイアログを選択し、次に [コマンド] タブをクリックします。
3. [カテゴリ] ボックスでカテゴリ名をクリックし、[ボタン] 領域のボタンまたは項目を表示されているツール バーにドラッグします。

ツール バーのボタンを削除するには

1. 変更しようとするツール バーが表示されていることを確認します。
2. [ツール] メニューから [カスタマイズ] ダイアログを選択し、次に [コマンド] タブをクリックします。
3. ボタンを削除するには、ボタンをツール バーの外側にドラッグします。

デフォルトのボタンをツール バーから削除しても、[カスタマイズ] ダイアログにはそのボタンが残ります。表示をカスタマイズしたツール バーボタンを削除すると、その表示は完全になくなります。ただし、コマンドは [カスタマイズ] ダイアログの [コマンド] タブから使用可能です。

ヒント

表示をカスタマイズしたツール バー ボタンを再利用のため保存するには、未使用のボタンを格納するツール バーを作成してこのボタンを移動し、格納したツール バーを非表示にします。

ツール バーを表示／非表示するには

1. [ツール] メニューから [カスタマイズ] ダイアログを選択し、次に [ツール バー] タブをクリックします。
2. 表示／非表示したいツール バーを、[ツール バー] リストから選択／選択解除します。
3. [閉じる] をクリックします。

または、

1. ユーザー インターフェイスのツール バー領域の任意の場所を右クリックします。
2. 表示または非表示にしたいツール バーをクリックします。メニューは自動的に閉じます。

ツール バー ボタンの表示を変更するには

1. ［ツール］メニューから［カスタマイズ］ダイアログを選択し、次に［ツール バー］タブをクリックします。
2. 以下のオプションを選択します。
   - **ツールチップを表示**：ツール バーのボタンまたはフィールドにカーソルを移動するとツールチップが表示されます。
   - **大きいボタン**：ツール バーのボタンのサイズを大きくします。
3. ［閉じる］をクリックします。

ステータス バー

ステータス バーには、いくつかのタスクの状態に関する有意義な情報が表示されます。たとえば、エラーやツールチップなどが表示されます。また、進捗状況や現在のアクションなども表示されます。

テキスト ファイルについては、ステータス バーの右端に現在のライン番号とカラム位置が表示されます。

ラインの移動

テキスト ファイル内の指定行に移動するには、Ctrl + Shift + G キーを押し、表示されたダイアログで、移動先のライン番号を入力します。

プログレス バー

ワークスペースを開くとき、ステータスバーの右側に全進捗状況を示すプログレス バーが表示されます。

メッセージ フィールドにも、個々のロード プロセスの進捗状況を示すプログレスバーが表示されます。このとき、進行中の現在のアクションを示すメッセージも表示されます。

オプション

ツールのオプションは、現在のプロジェクトまたはワークスペースだけではなく、Tau 全体にも影響を及ぼします。これらのオプションは、以下の方法で変更できます。

［オプション］ダイアログには、変更するオプションごとに別のタブがあります。タブの数は現在アクティブなプロジェクトのタイプによって異なります。［オプション］ダイアログのオプションの説明を表示するには、ダイアログのタイトル バーにあるクエスチョン マークをクリックし、次に目的のオプションをクリックします。

［詳細］タブでは、［オプション］ダイアログで使用できるすべてのオプションをツリー表示で確認できます。［詳細］タブは［一般］タブのチェック ボックスを使用して表示するよう設定できます。［詳細］タブのオプションの値を変更するには、値を選択して F2 キーを押します。

オプション ファイル

オプション設定は、オプション ファイル `.tot` に保存できます。このファイルは後で編集できます。

プロジェクトにオプション ファイルを追加した場合、［ファイル ビュー］でそのファイルをダブルクリックしてオプション エディタで開くことができます。複数のオプション ファイルをプロジェクトに保存し、使用するファイルを選択できます。これは、頻繁にオプションを切り替える場合に便利な機能です。オプションを直接変更せず、オプション ファイルの優先順位を変更して使用できます。

インストール時、内部フレームワークの設定とオプションを含む拡張子「`.tot`」の付いたファイルが多数作成されます。通常これらのファイルはユーザーが編集すべきではありません。拡張子「`.tot`」の付いたファイルを編集すると、データが失われたり、ツールセットが正しく使用できなくなる場合があります。オプションを変更する必要がある場合は、［ツール］メニューの［オプション］ダイアログを使用して行います。

オプションの変更

オプションを変更するには

1. ［ツール］メニューから［オプション］をクリックします。
2. ［オプション］ダイアログのタブを使用して、オプションの選択または選択解除を行います。［詳細］タブで F2 キーを押してオプションの値を指定します。
3. ［OK］をクリックします。

オプション ファイルの操作

現在のオプションを新しいオプション ファイルに保存するには

1. ［プロジェクト］メニューから［オプション］をクリックし、次に［名前を付けて保存］をクリックします。
2. ［名前を付けて保存］ダイアログで、オプション ファイル（.tot）の名前と保存場所を指定します。
3. ［保存］をクリックします。
4. アクティブなプロジェクトにオプション ファイルを含めるかの確認を求められるので、［はい］をクリックします。

   このファイルをプロジェクトに含めると、［ファイル ビュー］から開いて編集できるようになります。

オプション ファイルを編集するには

1. オプション ファイルがプロジェクトに含まれている（[ファイル ビュー] に表示されている）ことを確認します。
2. オプション ファイルをダブルクリックします。ファイルが開き、オプション エディタに表示されます。
3. オプション エディタで、目的のオプションが表示されるようにオプション ツリーを展開します。
4. オプションを選択します。
5. 選択したオプションで、[値] フィールドをクリックして F2 キーを押します。これで、フィールドを編集できるようになります。
6. 新しい値を入力します。
7. 終了したらファイルを閉じます。変更を保存するかの確認を求められます。

使用するオプション ファイルを選択するには

1. 2 つ以上のオプションファイルがプロジェクトに含まれて（[ファイル ビュー] に表示されて）いないと、この操作を行うことはできません。
2. [プロジェクト] メニューから [オプション] をクリックし、次に [ファイル] をクリックします。
3. [オプション ファイル] ダイアログで、リスト内のオプション ファイルを選択し、矢印ボタンを使用してファイルを上または下に移動します。リストの一番上にあるオプション ファイルが使用されます。
4. [OK] をクリックします。

カスタマイズ

ユーザー インターフェイスの表示をカスタマイズできます。[カスタマイズ] ダイアログ（[ツール] メニューから表示）には、ツール バー、[ツール] メニュー、ウィンドウのレイアウト、およびアドイン モジュールのカスタマイズを行うオプションがあります。詳細については、第 55 章「Tau のカスタマイズ」を参照してください。

ローカル セットアップ（**UNIX**）

Windows ディレクトリ

ホーム ディレクトリに **windows** という名前の新しいディレクトリが作成されます。このディレクトリには Windows と UNIX の間で Tau のプロパティを調整するために使用される一連のファイルがあります。これらのファイルに保管される情報は、ユーザーが編集すべきではありません。

コピーと貼り付け

選択したテキストをマウスの中ボタンを使用して他のターミナル ウィンドウで直接貼り付けるという操作は、出力ウィンドウのタブからのみ行えます。

Solaris ネイティブ端末で見られる切り取り、コピー、貼り付けコマンド専用のボタンは、サポートされません。

File ダイアログ

［File］ダイアログに表示するファイルを絞り込むことができます。たとえば、「*.u2」を指定するとこのタイプのファイルのみ表示されます。

サポートリクエストの作成

サポートに情報を送信するツールは、［ヘルプ］メニューから［サポート リクエスト］を選択して起動します。

このサポートツールから、画面ショットの取得や状況を録画したビデオクリップなどを作成して、Telelogic サポートに送信できます。

ワークスペースの操作

ワークスペースの概要

ワークスペースは、個人の作業領域であり、自分のプロジェクトを配置する場所です。さらに、ワークスペースは、自分のプロジェクトに論理的な構造をもたせるものともいえます。ワークスペースは複数定義できますが、一時点で作業できるのは、1 つのワークスペースのみです。ワークスペースに追加するプロジェクトの数に制限はありません。

他のユーザーとワークスペースを共有することはできません。あるユーザーのワークスペースの内容は他のユーザーのニーズとは合致しないのが通常だからです。

プロジェクトは、ワークスペースからの相対パス名で保管されます。したがって、ワークスペースとその全内容を、情報を損なうことなく別の場所に移動できます。

ワークスペースの情報は、拡張子「`.ttw`」の付いたテキスト ファイルに保管されます。

新規ワークスペースの作成

Tau の起動時には、空のアクティブ ウィンドウが表示されます。以前にワークスペースを使用していない場合は、通常は、新規ワークスペースを作成するところから作業を始めます。

1. ［ファイル］メニューから［新規］をクリックします。
2. ［ワークスペース］タブをクリックします。
3. ワークスペースに付ける名前を入力します。

4. ワークスペースの保管場所を選択します。デフォルトで、ワークスペースと同名のフォルダが作成されます。
5. ［OK］をクリックします。

ワークスペースを開く

他の標準的なアプリケーションと同様に、ワークスペースについて、開く、閉じる、保存する、といった操作ができます。最近使用したワークスペースは、ショートカット リストから開くことができます。リストには 8 個までのワークスペース名が表示されます。

ワークスペースを開くには

1. ［ファイル］メニューから［ワークスペースを開く］をクリックします。
2. ［開く］ダイアログで、開くファイルを選択します。表示されていないときは参照して選択します。［開く］をクリックします。

最近使ったワークスペースを開くには

1. ［ファイル］メニューを開き［最近使ったワークスペース］にカーソルを合わせます。最近使ったワークスペース名が 8 個まで表示されます。
2. ワークスペースを選択します。

ワークスペースを保存して閉じる

- ワークスペースを保存するには［ファイル］メニューから［ワークスペースを保存］をクリックします。
- ワークスペースを閉じるには［ファイル］メニューから［ワークスペースを閉じる］をクリックします。

ワークスペースへのプロジェクトの追加

ワークスペースにプロジェクトを追加するには、次の 2 通りの方法があります。

［ファイル ビュー］から

1. ［ファイル ビュー］でワークスペースを右クリックし、ショートカットメニューから［プロジェクトの挿入］をクリックします。
2. ［プロジェクトの挿入］ダイアログで、追加するプロジェクトを選択します。
3. ［開く］をクリックします。

［プロジェクト］メニューから

1. ［プロジェクト］メニューから［プロジェクトをワークスペースに挿入］をクリックします。［ファイルの挿入］ダイアログが表示されます。
2. ［プロジェクトの挿入］ダイアログで、追加するプロジェクトを選択します。
3. ［開く］をクリックします。

参照

新規ワークスペースでのプロジェクトの作成

プロジェクトの操作

プロジェクトの概要

プロジェクトには、プログラムや最終バイナリ ファイルを作成するためのソース ファイルへのさまざまな参照が含まれています。

プロジェクトは、ワークスペースに登録されます。1 つのワークスペース内の複数のプロジェクトでダイアグラムやドキュメントを移動しながら作業を行うことができます。すでにあるワークスペースに入っているプロジェクトを別のユーザーのワークスペースに入れることもできます。このワークスペースに追加したプロジェクトは、別のパス、別のドライブ、ルート ディレクトリ内のディレクトリに配置できます。したがって、チーム内のメンバーは同じプロジェクトの作業を行うことができます。

プロジェクトの作成時にワークスペースが開かれていない場合、ワークスペース用のディレクトリとワークスペース ファイルも作成されます。あるいは、既存の開いているワークスペースへのプロジェクトの追加も可能です。

プロジェクト内で参照するファイルはどのタイプのファイルでもよく、ユーザーが作成したフォルダに入れて理解しやすいように整理できます。

プロジェクトは個人的なものではないので、ユーザー間で共有できます。一部の設定はグローバルです。たとえば、あるユーザーがファイルを追加すると、他のユーザーのプロジェクトにも同じように追加されます。使用するフォントの設定など、グローバルではない設定もあります。

プロジェクトの情報は、拡張子「`.ttp`」の付いたテキスト ファイルに保管されます。プロジェクト内のファイルは、相対パスでプロジェクト ファイルに保管されます。

アクティブ プロジェクト

ワークスペースにプロジェクトを追加すると、アクティブ プロジェクトになります。［プロジェクト］メニューのすべてのコマンドが、アクティブ プロジェクトに適用されます。開いているワークスペースに 2 つ以上のプロジェクトがある場合、ツールバーのアクティブ プロジェクト リストでアクティブにするプロジェクトを選択する必要があります。

Windows ユーザーへの推奨事項

一般的に、プロジェクトとそれに含まれるすべてのファイルを論理ドライブ（P:、Q: など）に保管することを推奨します。こうすることで、チーム メンバー間でモデルとプロジェクトをやり取りしやすくなります。たとえば、ハイパーリンクまたはトレーサビリティ リンクを使用する場合、リンクにはプロジェクトの絶対パスが論理ドライブを使った形で保存されます。また、DOORS インテグレーションや他のサードパーティ ツールのインテグレーションを使用する場合にも、この方法を推奨します。こう

しておけば、ファイル構造を正確に複製しなくても DOORS からモデルにナビゲートできます。詳細情報については、`subst` DOS コマンドを参照するか、システム管理者にお問い合わせください。一貫した論理モデル リポジトリを実現するため、常にこの論理ドライブの場所からツールを開いてください。

プロジェクトでの作業の開始

初めてツールを使用する際、まずファイルを作成してから後でこのファイルをプロジェクトに追加する、という方法を取ることができます。しかし、推奨されるのは、ワークスペース内でプロジェクトを作成し、その後ファイルをプロジェクトに追加するという作業フローです。この作業フローにした方が自分のプロジェクトを制御しやすくなります。

プロジェクトごとに新しいワークスペースを作成するか、または、既存のワークスペースにプロジェクトを追加します。プロジェクト内のファイルは、ローカルに配置することも、または他の場所に配置することもあります。いずれにせよ、自分または自分のプロジェクトに最適の作業環境を整えることが重要です。

プロジェクトの作成または修正を行う際、ワークスペースのビュー内にさまざまなコンポーネントが表示されます。

ツール レベル、または現行のプロジェクトのために、環境設定を変更、設定する複数の選択肢があります。たとえば、カスタム ツール バー、メニュー コマンド、ボタンなどを作成できます。

新規ワークスペースでのプロジェクトの作成

1. ［ファイル］メニューから［新規］をクリックします。
2. ［プロジェクト］タブをクリックします。
3. 作成するプロジェクトのタイプを指定します。
4. プロジェクトに付ける名前を入力します。参照ボタンを使用して、プロジェクトの場所を変更できます。デフォルトで［新しいワークスペースを作成する］オプションが選択されています。
5. ［OK］をクリックします。
6. 必要に応じて、プロジェクトの設定を変更できます。ヘルプボタンをクリックすると、設定に関する詳細情報を得ることができます。
7. ［次へ］をクリックし、次に［終了］をクリックします。

既存ワークスペースでのプロジェクトの作成

1. 新規プロジェクトを追加するワークスペースを開きます。
2. ［ ファイル］メニューから［新規］をクリックします。
3. ［プロジェクト］タブをクリックします。
4. 作成するプロジェクトのタイプを指定します。

5. プロジェクトに付ける名前を入力し、［現在のワークスペースに追加する］をクリックします。
6. ［OK］をクリックします。
7. 必要に応じて、プロジェクトの設定を変更できます。ヘルプボタンをクリックすると、設定に関する詳細情報を得ることができます。
8. ［次へ］をクリックし、次に［終了］をクリックします。

既存プロジェクトのワークスペースへの挿入

プロジェクトは配置されているワークスペース内で操作されます。ワークスペースには、他の標準的なアプリケーションと同じように、開く、閉じる、保存するといった操作を行うことができます。プロジェクト ファイル（.ttp）はどのワークスペースにも挿入できます。また、最初に作成されたワークスペース以外に挿入することもできます。

既存ワークスペースにプロジェクトを挿入するには

1. ワークスペース ウィンドウの［ファイル ビュー］タブを開きます。
2. ワークスペース アイコンを右クリックし、ショートカット メニューから［ファイルの挿入］を選択します。［ファイルの挿入］ダイアログが表示されます。
3. 挿入するプロジェクトを選択し、［ 開く］をクリックします。

注記

プロジェクト ファイルを表示するため、［ファイルの挿入］ダイアログのフィルタの変更が必要な場合があります。

参照

メニュー バーとツール バー

ワークスペースへのプロジェクトの追加

プロジェクトへのファイルとフォルダの追加

プロジェクトへのファイルの追加

プロジェクト ファイル、ワークスペース ファイルなどのほか、どのタイプのファイルでもプロジェクトに追加できます。プロジェクトに追加されたワークスペースは、特に機能を持たない通常のテキスト ファイルと同じように扱われます。ドラッグ アンド ドロップにより、フォルダ間でファイルを移動できます。

プロジェクトにファイルを追加するには、次の 2 通りの方法があります。

［ファイル ビュー］から

1. ［ファイル ビュー］でプロジェクトまたはフォルダを右クリックし、ショートカット メニューから［ファイルの挿入］をクリックします。
2. ［ファイルの挿入］ダイアログで、追加するファイルを選択します。

3. ［開く］をクリックします。

［プロジェクト］メニューから

1. ［プロジェクト］メニューから［プロジェクトに追加］をクリックし、次に［ファイル］をクリックします。
2. ［プロジェクトへのファイルの挿入］ダイアログで、挿入するファイルを選択します。
3. ［開く］をクリックします。

注記

プロジェクト メニューからファイルを追加する場合、追加先フォルダを指定することはできません。すべてのファイルはプロジェクトのルート レベルに追加されます。

最近使ったファイルを開くには

1. ［ファイル］メニューを開き［最近使ったファイル］にカーソルを合わせます。最近使ったファイル名が 8 個まで表示されます。
2. ファイルを選択します。

プロジェクトへのフォルダの追加

プロジェクトにフォルダを追加するとファイルを論理的に整理できます。追加されたフォルダは、プロジェクト ファイル内でのみ定義されます。ファイル システムには作成されません。OS 内のファイル パスは変更されません。

フォルダの追加時にファイル拡張子のリストを定義できます。これによって、フォルダに入れるファイルのタイプを示すことができます。フォルダにファイルを追加する際、リストがフィルタの役目を果たします。ファイルを追加するダイアログには、デフォルトでリスト内にある拡張子を持つファイルのみ表示されます。

プロジェクトにフォルダを追加するには、次の 2 通りの方法があります。

［ファイル ビュー］から

1. ［ファイル ビュー］でプロジェクトを右クリックし、ショートカット メニューから［新しいフォルダ］をクリックします。
2. ［新しいフォルダ］ダイアログの［フォルダ名］ボックスに、フォルダに付ける名前を入力します。
3. ［ファイルの拡張子］ボックスに、*.< 拡張子 > の形式で 1 つ以上のオプションの拡張子を入力します。2 つ以上の拡張子を入力する場合、それぞれをセミコロン（;）で区切ります。
4. ［OK］をクリックします。

［プロジェクト］メニューから

1. ［プロジェクト］メニューから［プロジェクトに追加］をクリックし、次に［新しいフォルダ］をクリックします。
2. ［新しいフォルダ］ダイアログの［フォルダ名］ボックスに、フォルダに付ける名前を入力します。
3. ［ファイルの拡張子］ボックスに、***.<** 拡張子 **>** の形式で 1 つ以上のオプションの拡張子を入力します。2 つ以上の拡張子を入力する場合、それぞれをセミコロン（;）で区切ります。
4. ［OK］をクリックします。

プロジェクトのアクティブ化

プロジェクトの機能を有効にするには、プロジェクトをアクティブにする必要があります。ワークスペース内で一時点でアクティブにできるプロジェクトは 1 つだけです。プロジェクトをアクティブにするには、以下のいずれかを行います。

- ［ファイル ビュー］でプロジェクトを右クリックします。ショートカット メニューから［アクティブ プロジェクトに設定］をクリックします。
- プロジェクト ツール バーで、［アクティブ プロジェクト］リストから目的のプロジェクトを選択します。
- ［プロジェクト］メニューから［環境設定］ダイアログを開き、［環境設定］ダイアログの［アクティブに設定］ボタンをクリックします。

注記

ワークスペースにプロジェクトが 1 つしかない場合、そのプロジェクトが自動的にアクティブになります。

参照

プロジェクトの設定と構成

ファイルとフォルダのプロパティ

選択したワークスペース項目を表示したり、必要に応じて編集できます。プロパティを表示できるのは、［ファイル ビュー］内で 1 つの項目を選択した場合のみです。

ファイル、プロジェクト、およびワークスペースのプロパティは、表示専用です。フォルダのプロパティは完全に編集可能です。

ファイルとフォルダのプロパティを表示するには

- ［ファイル ビュー］で項目を右クリックし、ショートカット メニューから［プロパティ］をクリックします。Alt + Enter キーを押してもプロパティを表示できます。

フォルダ プロパティを設定または変更するには

1. ［ファイル ビュー］でフォルダを右クリックし、ショートカット メニューから［プロパティ］をクリックします。
2. プロパティ エディタボックスで、［フォルダ名］と［ファイルの拡張子］の内容を変更します。
3. オプション：Enter キーを押して変更の適用とベリファイを行います。
4. 他のワークスペース項目の環境設定を表示または変更しないときは、［閉じる］ボタンをクリックしてダイアログを閉じます。

変更を保存せずにダイアログを閉じるには

- Esc キーを押します。

ファイルを作成する、開く、閉じる

新規ファイルを作成するには

1. ［ファイル］メニューから［新規］をクリックします。
2. 作成するファイルのタイプを指定します。
3. ファイルに付ける名前と保管場所を指定します。
4. プロジェクトが開いている場合、ファイルをプロジェクトに入れるかどうかを決められます。
5. ［OK］をクリックします。

ファイルを開くには

1. ［ファイル］メニューから［開く］をクリックします。
2. ［開く］ダイアログで、開くファイルを選択します。表示されていないときは参照して選択します。
3. ［開く］をクリックします。

または

1. ［ファイル］メニューから［最近使ったファイル］をクリックします。最近使ったファイル名が 8 個まで表示されます。
2. ファイルを選択します。

ファイルを閉じるには

- ［ファイル］メニューから［閉じる］をクリックします。

ファイルの移動

プロジェクト間でファイルを移動した場合を除いて、ドラッグ アンド ドロップでファイル（およびプロジェクトとフォルダ ノード）を移動しても、読み込み済みの UML モデル エンティティの現状には影響しません。プロジェクト間でファイルを移動する

と、移動したファイル中に含まれるモデル要素が移動元のプロジェクトから削除され、移動先のプロジェクトに追加されます。この移動操作を行うと、操作の取り消し用のキューは無効になります。

プロジェクトの設定と構成

プロジェクトの設定は、［設定］ダイアログで行います。分析、コード生成、ビルド、実行、ログ記録などのオプションを設定できます。

［設定］ダイアログのオプションを変更すると、現在アクティブな構成に保存されます。プロジェクトにはいくつかの構成を含むことができます。プロジェクトのアクティブ化を行い、［環境設定］ダイアログか対応するツール バーのリストにより、アクティブにする構成を設定できます。

プロジェクトの設定

プロジェクトの設定を変更するには、次の 2 通りの方法があります。

［ファイル ビュー］から

1. ［ファイル ビュー］でプロジェクトを右クリックし、ショートカット メニューから［設定］をクリックします。
2. ［設定］ダイアログが表示されます。必要に応じて設定を変更します。
3. ［OK］をクリックします。

［プロジェクト］メニューから

1. プロジェクト ツール バーで、［アクティブ プロジェクト］リストから目的のプロジェクトを選択します。
2. ［プロジェクト］メニューから［設定］をクリックします。
3. ［設定］ダイアログが表示されます。必要に応じて設定を変更します。
4. ［OK］をクリックします。

プロジェクトの構成

新規構成を追加するには

1. ［プロジェクト］メニューから［環境設定］をクリックします。
2. ［環境設定］ダイアログで［追加］をクリックします。
3. ［環境設定の追加］ダイアログで、構成に付ける名前を入力し、設定をコピーする既存の構成を選択し、新規構成を関連付けるプロジェクトを選択します。
4. ［OK］をクリックします。

構成を削除するには

1. ［プロジェクト］メニューから［環境設定］をクリックします。
2. ［環境設定］ダイアログで、削除する構成を選択します。
3. ［削除］をクリックします。
4. ［OK］をクリックします。

発見ベースのストレージ

概要

発見ベースのストレージは、モデルに含まれるすべてのファイルの所在を知るための新しい方法を提供します。

検索パス

検索パスはプロジェクトのプロパティで、Tau がファイルを検索する場所のリストを含みます。検索は繰り返し行われます。

このプロパティは、［発見］プロパティ ページの［発見位置］フィールドにテキストを入力して変更できます。

検索場所は、URN リファレンス、あるいは u2 モデルへの相対パスまたは絶対パスとなります。

- パスを手動で（参照ボタンを使用せずに）入力した場合、Tau でパスが変更されることはありません。
- 参照ボタンを使用してパスを指定した場合は、［Tau］->［オプション］->［一般］->［URN マップ］に対応してパスが調整されます。つまり、ファイルが URN ロケーションの下にある場合、URN リファレンスが計算されて保存されます。
- 参照ボタンを使用してパスを指定した場合、パスが URN ロケーションの下ではなくプロジェクトの相対パスであれば、この相対パスが計算されて保存されます。
- パスがプロジェクトと同じファイルシステムにない場合、パスは調整されずそのまま保存されます。

検索場所に Shallow 修飾子が付与され、`.u2shallow` ファイルと同じセマンティックを持ちます（以下を参照）。

検索場所の名前に、ワイルドカードは使用できません。

リソース マッチ ルール

リソース マッチ ルールは、検索ベースのストレージの対象とするファイルを指定する、正規表現です。

マッチ ルールは、包含的または排他的のどちらでもかまいません。

ルールは、［フォルダ］プロパティ ページの［ファイル フィルタ］フィールドにテキストを入力して変更できます。

ルールはセミコロンで区切りられたファイル マッチ パターンのリストです。

リストが空の場合（または存在しない場合）、すべてが包含されます。

エントリはリストに表示される順番に適用されます。最初に一致したエントリが優先されます。

ディレクトリ マッチ ルール

ディレクトリ マッチ ルールは、検索ベースのストレージの対象とするディレクトリを指定する、正規表現です。

マッチ ルールは、包含的または排他的のどちらでもかまいません。

ルールは、［フォルダ］プロパティ ページの［ディレクトリ フィルタ］フィールドにテキストを入力して変更できます。

ルールはセミコロンで区切りられたディレクトリ マッチ パターンのリストです。

リストが空の場合（または存在しない場合）、すべてが包含されます。

エントリはリストに表示される順番に適用されます。最初に一致したエントリが優先されます。

ディレクティブ

- Ignore ディレクティブ

  検索するディレクトリに「`.u2ignore`」という名前のファイルがある場合、そのディレクトリとディレクトリ内のすべてのファイルが無視されます。

  「`File.u2.u2ignore`」という名前のファイルがある場合は、「`File.u2`」という名前のファイルが無視されます。

  「directory.u2ignore」という名前のファイルがある場合は、「directory」ディレクトリが無視されます。

- Shallow ディレクティブ

  検索するディレクトリに「`.u2shallow`」という名前のファイルがある場合、そのディレクトリ内のすべてのサブディレクトリが無視されます。

  検索するディレクトリに「`directory.u2shallow`」という名前のファイルがある場合、その「`directory`」ディレクトリ内のすべてのサブディレクトリが無視されます。

- Discover ディレクティブ

  「.u2discover」という拡張子が付いたファイルを使用して、検索場所のほか、リソースの包含、除外のリストを提供できます。

注記

「*.u2x」という拡張子が付いたファイルは常に無視されます。

- u2discover ファイルのフォーマット

```
#This is a .u2discover file
DiscoveryPath:
#Do not recurse into subdirectories of the following location
Loca*tion1(Shallow)
```

```
Loca*tion2
ResourceInclusion(inclusive):
*.foo
*.boo
DirectoryInclusion(exclusive):
#Skip CVS subdirectories
CVS
```

検索場所に Shallow 修正子を追加できます。`.u2shallow` ファイル拡張子の場合と同じように解釈されます（上記を参照）。

`ResourceInclusion` セクションを “inclusive” または “exclusive”（ない場合はデフォルト “inclusive”）で修飾できます。'#' で始まる行は無視されます。すべての空白行は無視されます。セクションの順番は関係ありません。セクションは最初のカラムから始めます。すべてのセクションはオプションです。

`DirectoryInclusion` セクションを “inclusive” または “exclusive”（ない場合はデフォルト “inclusive”）で修飾できます。

キーワードとディレクティブでは大文字・小文字は区別されません。

I18N

`.u2discover` ファイルは、有効な UTF-8 ストリームの `UNICODE` 文字列を含むものとします。BOM やマジック ナンバーなどは無視されます。

ディレクティブの解釈

ディレクティブ／リソースには Directory/Resource Inclusion ルールが適用されます。プロジェクト内にある包含ルールは、まずマッチ ルールに追加され、各ディレクトリに移動すると、.u2discover ファイル（ある場合）にあるマッチルールのリストが付加されます。ディレクトリ／リソースがマッチ ルールに適用されると、最初のマッチが優先されます。追加の `.u2ignore/.u2shallow` ファイルが使用され、以前に名前が付けられたマッチが置き換えられます。

Generic SCC/Synergy

Generic SCC/Synergy インテグレーションは、通常ファイルと同じ方法で、検索されたファイルの操作を行います。

検索されたファイルをプロジェクト内で明示化するには

検索されたファイルをプロジェクト内で明示化するには、そのファイルを検索フォルダ（[ファイル ビュー] 内）からドラッグしてプロジェクト ノードにドロップします。

明示ファイルを検索対象とするには

[ファイル ビュー] からのファイル参照を削除します（ポップアップ メニューから [remove elements] を選択）。ファイルが含まれるディレクトリが `DirectoryPath` プロパティにあることを確認します。プロジェクトを保存して再ロードします。

フィルタの構文

サポートされるファイル マッチ用のワイルドカードは「*」のみです。

手動による再検索

プロジェクトをロードした後でも検索を実行できます。プロジェクトのコンテキスト メニューに［Discover Files］というメニュー項目があります。この方法では新しいファイルのみ追加できます。既に存在しないファイルを削除することはできません。

モデルとダイアグラム

モデル

モデルは、作業対象のシステムを書き表したすべてのダイアグラムから構成されます。各ダイアグラムは、それぞれ、アプリケーションの特定の側面を表します。UML を使ってシステムのモデリングを行う場合、クラス図はエンティティとそのエンティティ間の関係を表現しています。

ユース ケース図およびシーケンス図で表されたユース ケースを使うと、外部との相互作用とシステムの振る舞いの概要を記述できます。

アクティビティ図と相互作用概観図を使うと、モデルにおける同時並行的な振る舞いを記述できます。

状態機械図では、アクティブなクラスの振る舞いを記述し、合成構造図では、エンティティの外部への振る舞いと他のエンティティとの相互作用を記述します。

アプリケーションはモデルを元にコンパイルされます。タイプの異なるダイアグラムには、モデルの異なるビューが表示されます。つまり、ダイアグラムで使用可能なエンティティはモデルに存在しますが、モデルに存在するエンティティが必ずしもダイアグラムで使用可能とは限らない、ということです。

モデル要素

シンボル（プレゼンテーション要素）をダイアグラムから削除しても、モデル内の対応するエンティティが削除されることはありません。同じエンティティが他のダイアグラムで使われている可能性があるからです。

しかし、モデル内のエンティティを削除すると、ダイアグラム内の対応するシンボルが削除されます。これは、アプリケーションを表現するのはモデルであり、ダイアグラムはモデルのある側面を示しているに過ぎないからです。

モデル要素と表示要素が 1 対 1 の関係にあれば、表示要素を削除するとモデル要素も削除されます。モデル要素が編集可能な表示要素を 1 つだけ持つ場合、1 対 1 の関係になります。これは、たとえば状態機械図のシンボルの場合に当てはまります。

モデル要素はワークスペース ウィンドウの［モデル ビュー］に表示されます。

DOORS Analyst モデルのインポート

DOORS Analyst で作成したモデルは、以下の手順によって Tau プロジェクトの UML パッケージにインポートできます。

1. ［モデル ビュー］で UML パッケージを選択します。
2. 右クリックをして表示されたショートカット メニューから［Analyst モデルのインポート ...］を選択します。

   DOORS がまだ実行されていなければ、ここで DOORS が起動されます。ログイン情報を入力すると、ユーザーは DOORS に接続され、DOORS Analyst モデルをインポートできる状態になります。
3. ［モジュールの選択］ダイアログで、インポートしたい DOORS Analyst モデルを含むフォーマル モジュールを選択し、［OK］をクリックします。

新しいパッケージで DOORS Analyst モデルが使用できるようになります。DOORS Analyst モジュールから UML 要素を正しくインポートするためには、インポート前にフォーマルモジュール上で［Edit in Analyst］を実行して DOORS Analyst で編集したモジュールを保存しておく必要があります。

参照

第 2 章「モデルの操作」

第 4 章「UML 言語ガイド」

ビュー

ファイル

ダイアグラム

ダイアグラムは、UML を使ってモデルを表現したものです。ダイアグラムのタイプにしたがって、異なるプロパティとアクションを定義できます。

ダイアグラムは、通常、1 つのモデルのさまざまな見方 - ビューを表します。ダイアグラムにはさまざまなタイプがあります。ダイアグラムの名前は、UML の概念から導き出されたものです。サポートされるダイアグラムのタイプは、以下のとおりです。

- アクティビティ図
- クラス図
- コンポーネント図
- 合成構造図
- 配置図
- 相互作用概観図
- パッケージ図
- シーケンス図
- 状態機械図
- テキスト図

- ユース ケース図

ダイアグラムの使用

モデルを表現するために、いくつかの異なるタイプのダイアグラムを使用できます。ここでは、ダイアグラムの使用方法について説明します。

UML モデルを構築する際、どのような作業手順にするかは決まっていません。32 ページの図 2 は、作業手順の 1 つの例です。

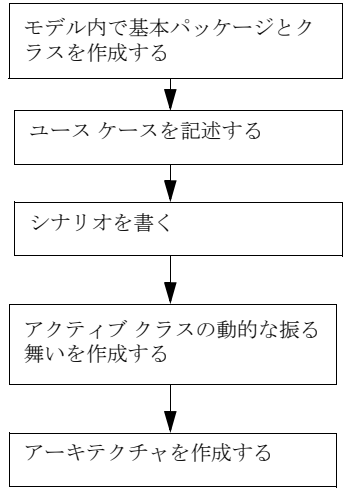


図 2: ダイアグラム作成のワークフロー

モデル内で基本パッケージとクラスを作成する

新規プロジェクトを作成すると、パッケージが自動的に挿入されます。ただし、プロジェクト作成時にこの機能を無効にした場合は、挿入されません。[モデルビュー] 内で新規パッケージを直接作成できます。これには、パッケージを右クリックし、ショートカット メニューから [新規モデル要素] を選択し、サブメニューから目的の要素を選択します。クラス図を追加するには、[モデル ビュー] でパッケージを右クリックし、ショートカット メニューから [新規ダイアグラム] を選択し、[クラス図] を選択します。

ユース ケースを作成する

ユース ケース図はパッケージ下に直接配置するか、クラス下に配置するか、コラボレーション下にグループ化できます。ユース ケースはパッケージ、クラスまたはコラボレーションに直接挿入できます。

シナリオを書く

ユース ケースを説明するシナリオは、シーケンス図として表現すると理解しやすくなります。シーケンス図は、構文が単純かつ直感的なので、動的な振る舞いの設計の基礎にも適しています。

クラスの動的な振る舞いを作成する

次のステップは、「アクティブ」として設定されたクラスの振る舞いの定義です。この作業には、状態機械図を使用します。アクティブ クラスごとにモデルに状態機械図を追加します。その状態機械図を 開くと、アクティブツール バーで使用可能なシンボルを使用したクラスの振る舞いを定義する内部状態機械が作成されます。

アーキテクチャを作成する

次のステップは、オブジェクトのやりとりの方法を定義することです。オブジェクトのインスタンス（パート）化とパート間のやりとりは、合成構造図で記述できます。合成構造図は、クラスの内部構造、クラスの属性、およびクラスのインスタンス化を表します。

次のステップへ

さらに作業を続けてゆくには、テスト プロジェクトを作成し、Tau を使用してさまざまなダイアグラム、要素、シンボルについての作業を行う方法について理解してください。

参照

第 4 章「UML 言語ガイド」
第 2 章「モデルの操作」

ワークフローの説明

このワークフローの説明は、プロジェクトのさまざまな作業場面でどのようにツールを効果的に利用できるかについての簡潔なロードマップを提示することを目的としています。実際のプロジェクトでは、ここで説明する内容をプロジェクト構成とアプリケーションドメインに照らして、調整する必要があります。

たとえば、反復設計アプローチを適用する、システムの保守を考慮する、などに状況によって、より簡略にもより複雑にも調整される可能性があるでしょう。

ワークフロー

ワークフローとは、成果（通常は製品）を得るために、プロジェクトに参加するメンバーがあるプロセスに従って実行する一連の作業のことです。

ワークフローは、より一般的な開発プロセスを特定のプロジェクトのニーズを満足するように調整した結果として生まれてきます。つまり、ワークフローは、一般的な開発プロセスとプロジェクト固有の調整の両方に大きく依存します。この章では、ワークフローよりも個々の作業に焦点を当てて説明します。

通常、プロジェクトに参加するメンバーのロールは、アーキテクト、デザインなど、1 つまたはいくつかの特定の作業に特化したものになります。

ワークフローを以下のように分けて説明します。

- 要求分析作業
- システム分析作業
- システム設計作業
- 詳細設計作業
- 実装作業

あるフェーズから別のフェーズに移るときには、そのフェーズのモデルを凍結して新規プロジェクトで新規モデルの作業を続けるために、旧モデルの情報をすべてコピーしたベースラインを作成することを推奨します。

作業をフェーズとして明確に分割することは、作業と情報を現実的に管理可能なサイズとして組織立てるという点で重要です。フェーズの数とフェーズとフェーズの差異は、プロジェクトと組織によってさまざまです。フェーズ間の差異を減らすことができる要因として、UML などの優れたモデリング言語と優れたツールサポートの利用があげられます。

強力な UML モデルのシミュレーション機能を使うことによって、要求モデルを実行可能なプログラムに落とし込むことが可能になり、要求と実装の間に生じるギャップを効果的に減らすことができます。

フェーズ	通常使用する ダイアグラム	新規プロジェクト ウィザード
要求分析作業	ユース ケース図 シーケンス図 相互作用概観図 アクティビティ図 クラス図	UML（モデリング用） または UML（モデル ベリファイ用）
システム分析作業	ユース ケース図 シーケンス図 相互作用概観図 アクティビティ図 パッケージ図 クラス図 合成構造図 状態機械図	UML（モデル ベリファイ用）
システム設計作業	ユース ケース図 シーケンス図 相互作用概観図 アクティビティ図 パッケージ図 コンポーネント図 クラス図 合成構造図 状態機械図	UML（モデル ベリファイ用）
詳細設計作業	パッケージ図 コンポーネント図 クラス図 合成構造図 状態機械図 テキスト図	UML（モデル ベリファイ用）
実装作業	配置図	UML（C コード生成用） UML（C++ コード生成用）
システム テスト作業	シーケンス図 合成構造図	UML（モデル ベリファイ用） UML（C コード生成用）

参照

第 72 章「ダイアログ ヘルプ」の「[プロジェクト] タブ」

要求分析作業

概要

要求分析の目的は、アプリケーション ドメインを理解し、構築するシステムに対するユーザーの要求を把握し、形式化、分析、妥当性検査することです。

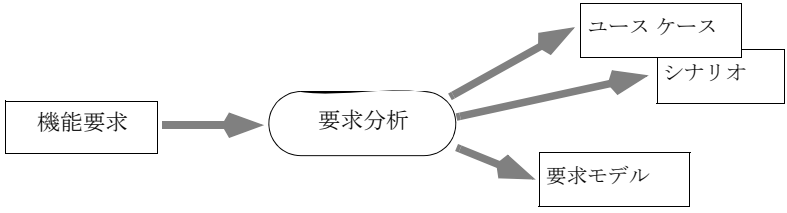


図 3: 要求分析作業の概要

要求仕様から作成される機能要求（機能要件）は、通常は、ドメイン エキスパートの知識と経験とともに、要求分析の重要なインプットとなります。

ユース ケースの特定

アプリケーション ドメイン内でアプリケーションの機能要求を実現するの十分であると考えられる一連のユース ケースを特定します。

要求モデルの作成

要求モデルの目的は、要求分析で見出した概念を特定して文書化し、相互に関連付けることです。

- アプリケーション ドメイン内で機能要求を実現するために十分と考えられる一連のクラスを特定します。
- 必要に応じて、特定した一連のクラスを適切な個別ドメインにグループ化し、クラス間の関係を記述します。
- 特定したそれぞれのドメインについて、実行可能な要求モデルを実装します（オプション）。

シナリオの作成

特定されたクラス インスタンスがユース ケースによってカバーされる要求事項を実現するためにどのように相互作用するかを表現した一連の相互作用（シーケンス図や相互作用概観図）を、ユース ケースごとに作成します。シナリオとは、あるユース ケースの実装と見なすことができます。

ベリフィケーションとバリデーション作業

- 要求モデルを実行して要求を検証します（オプション）。
- テスト結果を相互作用（シーケンス図）として保存します（オプション）。

ユース ケース

機能要求から、一連の重要なユース ケースとユース ケースに関わるアクターを特定します。これらをユース ケース図で記述します。

要求モデル

要求モデルは、以下のようなドメイン コンテキストの主なクラスが含まれるクラス図で構成されます。

- ユース ケースのアクターなどの重要なアクティブ エンティティ
- シグナルなどの重要なメッセージ
- クラスなどの重要なデータ

オプション：それぞれのアクティブエンティティの状態（ステート）指向の状態機械図に簡単な振る舞い記述を追加し、要求モデルを実行形式にします。モデル ベリファイヤ（Model Verifier）を使用してシミュレートします。

シナリオ

シーケンス図に簡単な相互作用で記述されている各ユースケースを詳細化します。

参照

ユース ケース モデリング

シナリオ モデリング

クラス モデリング

システム分析作業

概要

要求分析作業の目的は、解決すべき課題とその課題がシステムに課す要求を理解することでした。一方、システム分析作業の目的は、システムそのもののアーキテクチャを理解することです。

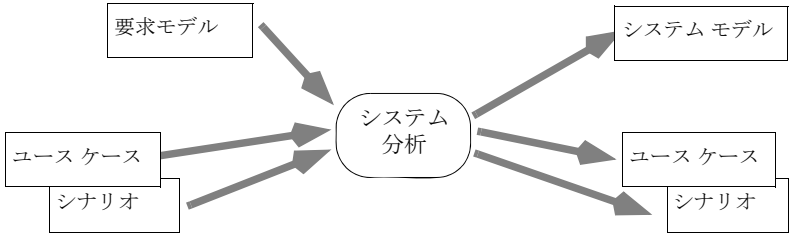


図 4: システム分析作業の概要

システム モデルの作成

システム モデルは、シンプルな論理アーキテクチャ、つまりシステムの完成時点で実装されている主要なオブジェクト、を記述するための 1 つの方法です。

- 要求モデルから、システム モデルで再利用できるクラスを決定します。
- 新しく一連のシステム分析クラスを確立します。このクラスは、アプリケーション アーキテクチャと概要レベルのプラットフォームの技術アーキテクチャを実現するためのものです。
- システム分析クラスと要求モデルから再利用したクラスを組み合わせて、1 つまたは複数のシステム モデルを作成します。
- 概要 レベル アーキテクチャごとに、1 つまたは複数の実行可能なシステム モデルを実装します（オプション）。

ユース ケースとシナリオの洗練

システム モデルのユース ケースとシナリオの目的は、論理アーキテクチャによって要求をどのように実装できるかを文書化することです。

- 概要レベル技術アーキテクチャとアプリケーション アーキテクチャを導入するために修正が必要な要求分析ユース ケースを特定します。
- 再利用されたユース ケースに加えて、プラットフォーム内のシステム分析要求を含めるために追加すべき一連の新規ユース ケースを作成します。
- 特定されたクラスがユース ケースによってカバーされる要求を実現するためにどのように作用し合うかを示す、一連の相互作用（シーケンス図、相互作用概観図、アクティビティ図）をユース ケースごとに作成します。

検証と妥当性検査作業

- システム モデルを実行することによってユース ケースとシナリオを検証します。
- テスト結果を相互作用（シーケンス図）として保存します（オプション）。
- システム モデルの妥当性を検査します。

システム モデル

システム モデルは、通常以下の要素で構成されます。

- パッケージ図 - パッケージ構造とパッケージ間の依存でモデルの編成を可視化
- クラス図 - あるシステム コンテキストにおいて最も重要なアクティブ エンティティ（アクティブ クラス）用。この時点でシステム アーキテクチャを考慮する必要あり。
- クラス図 - メッセージング関連用。最重要メッセージおよび重要なメッセージ データ（パッシブ クラスとデータ型）のインターフェイスとシグナル定義。
- クラス図 - 重要な操作と属性を含んだ最重要パッシブ データ モデリング用。
- 合成構造図 - アクティブ クラスの簡単なアーキテクチャと通信構造（パートとコネクタ）を可視化。
- 状態機械図 - アクティブクラス用。状態（ステート）中心型であり、遷移の詳細は定義しないが各状態機械の概要を提供。
  状態機械を完成させる、つまり実行可能にできると、モデル ベリファイヤ（Model Verifier）を使用してシミュレーションができるという利点あり。

クラス図を上で述べたように厳密に分類する必要はありません。アクティブ クラスとパッシブ クラスの関係を示す場合のように、各ダイアグラムで表示する内容に意味があることの方が重要です。1 つの定義を複数のコンテキストで可視化する手法は、UML でよく使われる手法であり、UML ツールの持つ強力なモデル駆動型アプローチによって、そのように表現されたモデルを簡単に維持管理できます。

ユース ケースとシナリオ

ユースケースを、要求モデルのドメイン コンテキストではなく、システム モデルのシステム コンテキストにしたがってアップデートします。すべての機能要求がユース ケースで網羅されるように、必要なユース ケースを追加してゆきます。

各ユース ケースにシナリオ（シーケンス図）を添付する必要があります。このシナリオは、ユース ケースの目標を達成するためにすべてのアクティブ エンティティ（アクティブ クラス）がどのように相互作用するかを記述しています。アーキテクチャが複雑な場合、システム レベル（システムと外部アクターとの通信を表現）と詳細レベル（システム内のアクティブ エンティティが、どのように相互作用し、どのように外部アクターと作用するかを表現）の両方のシナリオを作成することを推奨します。

参照

シナリオ モデリング
パッケージ モデリング
クラス モデリング
振る舞いモデリング

システム設計作業

概要

システム設計作業の主な作業は、システムの正確なアーキテクチャを定義することです。

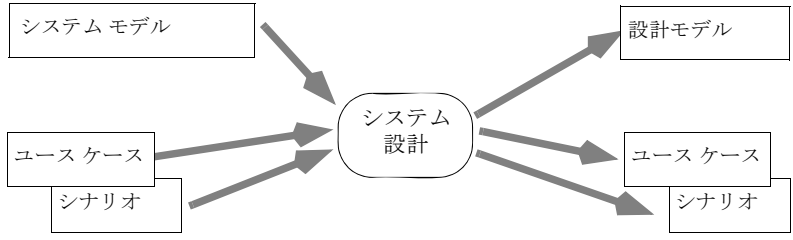


図 5: システム設計作業の概要

設計モデルの作成

システム設計作業において設計モデルを使用する意図は、正確なアーキテクチャを記述することです。

- システム モデルから、設計モデルで再利用できるクラスを決定します。
- 新しく一連のシステム設計クラスを確立します。このクラスは、技術ドメインでの詳細なアーキテクチャ要求と技術要求を実現するためのものです。
- システム設計 クラスとシステム モデルから再利用したクラスを組み合わせて、1 つまたは複数の設計 モデルを作成します。
- 1 つまたは複数の実行可能な設計モデルを実装します。
- 必要に応じて、パフォーマンス最適化のために 1 つまたは複数のターゲット モデルを作成します。

ユース ケースとシナリオの洗練

設計モデルのユース ケースとシナリオの目的は、正確なアーキテクチャが要求をどのように実装できるかをシステム モデルよりも詳細な方法で指定することです。

- 詳細な技術要求、技術アーキテクチャ、アプリケーション アーキテクチャを導入するために修正が必要なシステム分析ユース ケースを特定します。
- 再利用されたユース ケースに加えて、プラットフォーム内のシステム設計要求を含めるために追加すべき一連の新規ユース ケースを作成します。
- 特定されたクラスがユース ケースによってカバーされる要求を実現するためにどのように作用し合うかを示す、一連の相互作用（シーケンス図、相互作用概観図、アクティビティ図）をユース ケースごとに作成します。

検証と妥当性検査作業

- 設計 モデルを実行することによってユース ケースとシナリオを検証します。
- テスト結果を相互作用（シーケンス図）として保存します（オプション）。
- 設計モデルの妥当性を検査します。

設計モデル

設計モデルは、通常の以下の要素で構成されます。

- パッケージ図 - パッケージ構造とパッケージ間の依存でモデルの編成を可視化。
- コンポーネント図 - アクティブ エンティティ（アクティブ クラス）用。この時点ですべてのアクティブ エンティティがモデル化されている必要あり。
- クラス図 - メッセージング関連用。最重要メッセージおよび重要なメッセージ データ（パッシブ クラスとデータ型）のインターフェイスとシグナル定義。
- クラス図 - 重要な操作と属性を含んだ最重要パッシブ データ モデリング用。
- 合成構造図 - アクティブ クラスの簡単なアーキテクチャと通信構造（パートとコネクタ）を可視化。
- 状態機械図 - アクティブクラス用。状態（ステート）中心型であり、遷移の詳細は定義しないが各状態機械の概要を提供。状態機械を完成させる、つまり実行可能にできると、モデル ベリファイヤ（Model Verifier）を使用してシミュレーションができるという利点あり。

ユース ケースとシナリオ

完成したアプリケーション アーキテクチャにしたがってユース ケースをアップデートします。

完成したアプリケーション アーキテクチャにしたがってシナリオ（シーケンス図）をアップデートします。アーキテクチャが複雑な場合、システム レベル（システムと外部アクターとの通信を表現）と詳細レベル（システム内のアクティブ エンティティが、どのように相互作用し、どのように外部アクターと作用するかを表現）の両方のシナリオを作成することを推奨します。

参照

パッケージ モデリング

クラス モデリング

アーキテクチャ モデリング

振る舞いモデリング

詳細設計作業

概要

詳細設計作業は、システム設計の一環として行われることがよくあります。

しかし、システム設計と詳細設計を分かつポイントは存在します。つまり、詳細設計を開始する前に、モデル ベリファイヤ（Model Verifier）を使用して多くの機能要求は検証可能です。

詳細設計の目的は、詳細な振る舞いモデリングと詳細なデータ モデリングを追加して設計モデルを完成させることです。

設計モデルの洗練

この時点で設計モデルの仕様は以下のように完了している必要があります。

- パッシブ データが完成している。
- パッシブ クラスのすべての操作が完成している。
- インターフェイス、メッセージ、メッセージ データが完成している。
- 振る舞い設計が遷移中心となり遷移の詳細が完成している。

システム設計で完成した詳細なシナリオは、振る舞いの設計を完成するための非常に有効な情報源になります。

検証と妥当性検査作業

- 詳細設計モデルを検証します。
- すべての機能要求（ユース ケース）に詳細設計があることを確認します。

詳細設計モデル

詳細設計モデルは、通常以下の要素で構成されます。

- パッケージ図 - パッケージ構造とパッケージ間の依存でモデルの編成を可視化。
- コンポーネント図 - アクティブ エンティティ（アクティブ クラス）用。この時点ですべてのアクティブ エンティティがモデル化されている必要あり。
- クラス図 - メッセージング関連用。すべてのメッセージ データ（パッシブ クラスとデータ型）のインターフェイスとシグナル定義。
- クラス図 - すべての操作と属性を含むパッシブ クラス用。
- 合成構造図 - アクティブ クラスの詳細なアーキテクチャと通信構造（パートとコネクタ）を可視化。
- 状態機械図 - アクティブ クラス用。遷移の振る舞いを完全に記述。多くの場合、遷移中心の状態機械表現。
- 操作本体または状態機械図 - 全操作用。
- テキスト図 - 操作用。必要に応じて。

参照

クラス モデリング

振る舞いモデリング

実装作業

概要

このワークフローの説明では、実装作業を詳細設計の後に置いていますが、実際には、実装作業をできるだけ早い段階で開始することを推奨します。実装からのフィードバックが、選択したアプリケーション アーキテクチャと詳細設計の両方に影響を及ぼす可能性があるからです。

段階的アプローチを推奨します。実装のフィードバックを早期に得ることができるからです。

プロジェクトにおける実装作業は、ほとんどが自動化されています。それでも、以下のようにいくつかのステップが存在します。

コード アーキテクチャの計画

- 配置図を使用して、ハードウェアや基本的なソフトウェア層と関係性を表現します。
- 外部 C/C++ API をモデルにインポートする。
- 生成コードのファイル構造を決定する（makefile）。
- ターゲット環境とアプリケーションドメインを選択する（例：Win32、スレッド アプリケーション）。
- コード ジェネレータの設定を調整する。

実装

- 環境に対するシグナリング インターフェイスを実装する（「環境関数」）。
- ターゲット インテグレーションを調整する。

検証と妥当性検査作業

- シグナリング インターフェイスと他のすべての手書きコードのデバッグを実施する。
- テストを実施する。

参照

第 19 章「ビルドとコード生成の概要と例」
第 20 章「アプリケーション ビルド リファレンス」
第 27 章「C コード ジェネレータ リファレンス」
第 9 章「C/C++ のインポート」
第 41 章「C++ アプリケーション ジェネレータリファレンス」

システム テスト作業

概要

システム テスト作業では、これまで作業で完成した UML システムが要求を満たしているかどうか、また、システムが期待どおりに動作するかどうかを、UML テスト プロファイルを使用して検証します。

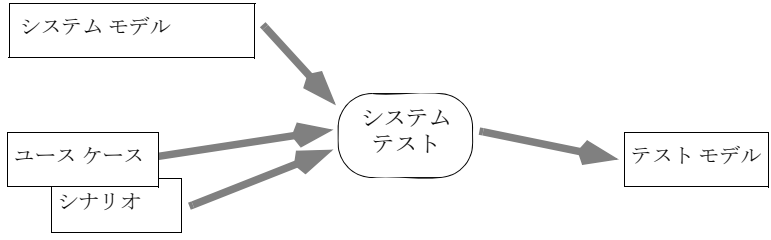


図 6: システム テスト作業

テスト モデルの作成

テスト モデルは、通常の以下の要素で構成されます。

- 以下の要素を含んだテスト コンテキスト クラス
  - 合成構造図 - SUT（System Under Test）とテスト コンポーネントのインスタンスを記述するパート用。
  - 一連のテスト ケース
  - テスト コンポーネント
- これまで作業で作成された UML モデルへの依存関係

テスト ケースの作成

これまで作業のテスト結果（シーケンス図に記述された相互作用）を保存していた場合は、この相互作用がテスト ケースの候補となります。また、これまで作業のシナリオもテスト ケースの候補となります。こういった相互作用は、テストケースに変換して、実行可能となるよう調整する必要があります。

テスト モデル

テスト コンテキスト クラスは、<<Test Context>> ステレオタイプが適用されたクラスです。テスト コンポーネントは、<<Test Component>> ステレオタイプが適用されたクラスです。合成構造図に記述されたパートは、テスト コンポーネントのインスタンスと <<SUT>> ステレオタイプが適用されたパートです。SUT は、通常、詳細設計作業で

作成された最上位クラス（システム）のインスタンスです。シグナルとデータ型は、通常、これまで作成された UML システムから再利用（インポートまたはアクセス）されます。

テスト ケース

テストケースは、テスト コンテキスト クラス上の <<Test Case>> ステレオタイプが適用された操作です。通常はシーケンス図に実装され、SUT が環境と通信する方法を指定します。

テスト

C コード ジェネレータを使用してテスト コンテキスト クラスを作成し、一括でテストを実行します。テストで不合格となったケースは、モデル ベリファイヤ（Model Verifier）を使用してデバッグします。

参照

第 48 章「UML テスト プロファイル」の「テスト モデルの作成」

ヘルプの使い方

ヘルプ ファイルには、対応する機能についての基本的なトピックと詳細なトピックがあります。

他に利用できるドキュメントについては、第 73 章「その他のリソース」を参照してください。チュートリアル、言語解説、インストール ガイド、Tau サポートサイトなどの外部サイトへのリンクがあります

インストール CD には、Adobe PDF 形式のドキュメントがあります。

ヘルプ ファイル内での移動

ヘルプ ファイルには、以下のように、情報を簡単に見つけるための機能があります。

- 46 ページの「検索」
- 47 ページの「検索ハイライト」
- 47 ページの「キーワード」
- 47 ページの「検索またはキーワード検索の同期」
- 47 ページの「ヘルプ ファイルのトピックのお気に入り登録」
- 48 ページの「ヘルプ トピックの印刷」

検索

完全なテキスト検索を実行するには

1. ヘルプ ビューアで［検索］タブをクリックします。
2. ［探したい語句を入力してください］フィールドに探す文字列を入力します。検索文字列として、通常表現、演算子、およびネストされた表現を使用できます。
3. また、［以前の結果から検索］、［類似する文字に合致］、および［タイトルのみ検索］のオプション（複数可）をチェックすることもできます。
4. ［検索開始］をクリックします。
5. 検索結果を表示するには、［トピックの選択］リスト内のトピックをダブルクリックするか、トピックを選択して［表示］をクリックします。

例 **1:** ——————————————————————————————

「link」で始まる単語を検索するには、［探したい語句を入力してください］フィールドに以下のように入力します。

```
link*
```

——————————————————————————————

検索ハイライト

検索した単語は、検索結果の全ページでハイライト表示されます。この機能は必要に応じてオフにできます。

検索ハイライトをオフにするには

1. ヘルプ ビューアで［オプション］ボタンをクリックし、メニューの［検索ハイライト オフ］をクリックします。
2. すでに検索を行っている場合、ヘルプ ビューアで［表示］ボタンをクリックすると、検索した単語がハイライト表示されなくなります。

再度オンにするまで、検索ハイライト機能はオフのままです。

検索ハイライトをオンにするには

1. ヘルプ ビューアで［オプション］ボタンをクリックし、メニューの［検索ハイライト オン］をクリックします。
2. すでに検索を行っている場合、ヘルプ ビューアで［表示］ボタンをクリックすると、検索した単語がハイライト表示されます。

再度オフにするまで、検索ハイライト機能はオンのままです。

キーワード

索引のリストを表示するには、［キーワード］タブを選択します。探している項目を見つけるには、単語の先頭文字を入力するか、リストをスクロールして検索します。項目を表示するには、項目をダブルクリックするか、項目を選択して［表示］をクリックします。

検索またはキーワード検索の同期

検索またはキーワード検索機能を使用している場合、検索結果は右側のウィンドウに表示されます。表示されているトピックが目次のどこにあるか確認するには、［同期］ボタンをクリックします。この方法で、関連トピックを見つけたり、次に検索するときのためにこのトピックが目次のどこにあるかがわかります。

ヘルプ ファイルのトピックのお気に入り登録

頻繁に参照するトピックや作業にとって重要なトピックがある場合、ウェブ ブラウザでよく行うように、「お気に入り」に登録できます。

トピックを「お気に入り」に登録するには

1. ［目次］、［キーワード］、［検索］などのタブを使用してトピックを検索します。
2. ［お気に入り］タブをクリックします。表示中のトピックの名前が［現在のトピック］フィールドに表示されます。
3. ［追加］をクリックします。追加したトピックの名前が［トピック］リストに表示されます。

ヘルプ トピックの印刷

1 つのトピック、または同じ章内の複数のトピックを選択して印刷できます。

現在表示されているトピックを印刷するには

- トピック ウィンドウを右クリックし、メニューから［印刷］を選択します。印刷ダイアログが表示されます。

目次から 1 つのトピックを印刷するには

1. トピック ウィンドウを右クリックし、メニューから［印刷］を選択します。印刷ダイアログが表示されます。
2. ［トピックの印刷］ダイアログで［選択されたトピックの印刷］をクリックし、［OK］をクリックします。印刷ダイアログが表示されます。

複数のトピックを印刷するには

1. 目次のブック アイコンを右クリックし、メニューから［印刷］を選択します。印刷ダイアログが表示されます。
2. ［トピックの印刷］ダイアログで［選択された見出しおよびすべてのサブトピックを印刷］をクリックし、［OK］をクリックします。印刷ダイアログが表示されます。

ヘルプでの検索構文

ヘルプビューアでは完全なテキスト検索を実行できます。また、文字（a-z）と数字（0-9）の組み合わせで検索文字を指定できます。「the」や「a」、「and」、「but」などの単語は制限されているので検索対象となりません。さらに、コロン（:）、セミコロン（;）、ハイフン（-）およびピリオド（.）などの句読点も検索対象となりません。

引用符やかっこによって検索要素をグループ化できます。

類似する文字に合致

ヘルプ ビューアの［検索］タブには［類似する文字に合致］オプションがあります。このオプションを選択すると、一般的な接尾辞を持つ単語をすべて検索できます。たとえば、このオプションを有効にして「run」を検索すると、「run」、「running」、「runner」が検索されます。「runtime」は検索されません。

正規表現

ヘルプ検索には、以下の正規表現を使用できます。

- * ：0 以上の文字の合致
- ? ：1 文字の合致
- 引用符内の文字列（"ab cd"）：逐語的に合致

検索対象	フィールドに入力する文字列
「analyze」、「analysis」、「analyses」、「analyzed」、「analyzing」を含むトピック	analy*
「analyzer」、「analyzed」を含むが「analyze」または「analyzers」は含まないトピック	analyze?
「analyze and generate」というフレーズを含むトピック	"analyze and generate"

演算子

ヘルプでの検索を絞り込むため、演算子（AND、OR、NOT、NEAR）を使用できます。検索文字列は左から右へ解釈されます。次の表に例を示します。

検索対象	フィールドに入力する文字列
「workspace」と「file」の両方を含むトピック	workspace AND file **または** workspace & file **または** workspace file
「workspace」と「file」のいずれかを含むトピック	workspace OR file **または** workspace | file
「workspace」を含むが「file」は含まないトピック	workspace NOT file **または** workspace | file
「workspace」の近くに「file」があるトピック（「workspace」と「file」の間は 8 文字以内）	workspace NEAR file
「workspace」を含むが「file」は含まないトピック、または「workspace」を含むが「directory」は含まないトピック	workspace NOT file OR directory

ネストされた表現

括弧を使用して表現をネストし、ヘルプで複雑な検索を行うことができます。かっこ内の表現が他の部分より先に解釈されます。表現のネスト可能なレベルは 5 レベルまでです。

検索対象	フィールドに入力する文字列
「workspace」を含むが「file」と「directory」のいずれかは含まないトピック	workspace NOT (file OR directory)
「workspace」を含み「file」と「project」が近くにあるトピック、または「workspace」を含み「directory」を含むが「project」が近くにあるトピック	workspace AND ((file OR directory) NEAR project)

UML モデリング

「UML モデリング」セクションの各章では、UML プロジェクトに固有の機能について説明しています。

2

モデルの操作

この章ではモデル ベースの開発について紹介します。ここでは、モデル バインディングの維持の方法について説明します。また、テキスト情報の構文カラー スキームについても触れます。

参照

第 1 章「Tau 4.3 の紹介」の 33 ページ、「ワークフローの説明」
第 3 章「ダイアグラムの操作」
第 4 章「UML 言語ガイド」

モデルとモデル要素

モデル ベースの開発

モデル ベースの特徴をもつ UML ツール セットは、複雑なモデルの作成と維持管理のための便利な機能を提供します。

操作方法は 2 通りあります。

- **ダイアグラム中心**の方法。モデルのダイアグラムの作成、編集をする際にモデルを作成します。
- **モデル中心**の方法。まず［モデル ビュー］ブラウザでモデルを作成してから、ダイアグラム ビューを定義します。

もちろん、2 つのパラダイムを組み合わせることもできます。

ダイアグラム中心のワークフロー

ダイアグラム中心のワークフローは、グラフィック言語を使用したことのあるユーザーにはよく知られた方法です。操作の進め方の例を以下に示します。

- ダイアグラムを作成する。
- ダイアグラムのエンティティを作成する。
- 定義したエンティティの詳細を記述するための、新しいダイアグラムを作成する。

この方法の利点は、新しいエンティティを作成すると、グラフィック コンテキストを使用できるようになるので、簡単かつ正確にモデルを作成できる点です。

モデル中心のワークフロー

モデル中心のワークフローは、モデル定義に存在するかどうかが確実ではないグラフィック表示に依存しません。ワークフローの例を以下に示します。

- モデル ブラウザでモデル要素を定義する。
- 新しいモデル要素をこのモデル構造内に配置する。
- モデルの関連パートを可視化する必要があるときに、ダイアグラムを作成する。
- モデル要素を［モデル ビュー］ブラウザからダイアグラムにドラッグすると、簡単にエンティティを可視化できる。
- モデル要素は、何度でも可視化できる。ダイアグラム ビューが異なる場合でも可能。

モデル ベース開発の結果、ダイアグラムにエンティティを記述するかどうかは、場合によって選択可能です。ダイアグラムのモデル表示の完全性を重視する場合、グラフィック表示されないエンティティをチェックするようにツールを設定できます。

モデル要素とプレゼンテーション要素

モデル要素

新しいオブジェクトまたは既存のオブジェクトに対して、新しい名前をつけて新しい定義を作成すると、ツールはモデルにオブジェクトが存在していないと認識します。これで新しいモデル要素が作成されます。このモデル要素はワークスペース ウィンドウの［モデル ビュー］に表示されます。

プレゼンテーション要素

ダイアグラムのシンボルは、モデル要素に基づいたプレゼンテーション要素です。通常、1 つのモデル要素に、プレゼンテーション要素をいくつでも配置できます。

要素プロパティ

クラス シンボルの属性名のように、あるプレゼンテーション要素のプロパティを変更すると、その変更はクラス内の他のプレゼンテーション要素にも反映されます。モデル要素が変更されると、この変更でプロパティに影響を受けたプレゼンテーション要素のすべてが更新されます。

（モデル ブラウザ、または、表示されたクラス シンボルのいずれかの）クラスに新しい属性を追加しても、この属性が、自動で、すべてのプレゼンテーション要素に表示されるわけではありません。もちろん、この属性をモデル内に配置して、プロパティを可視化するクラス シンボルに簡単に追加することもできます。

削除

クラス シンボルの属性を削除しても、シンボルの属性のプレゼンテーションが削除されるだけです。

モデルからの削除

［モデル ビュー］の属性を削除すると属性のモデル要素が削除され、これに伴い、その属性のすべてのプレゼンテーション要素も消失します（ダイアグラム内のプレゼンテーション要素を右クリックするかショートカット メニュー コマンドから［モデルの削除］を選択して削除することもできます）。

他のクラスで参照される属性タイプのように、他の場所から参照されるクラスを削除した場合、これらの参照のバインドが解除されます。この操作はダイアグラムに即座に反映されます（赤い波線で示されます）。

モデル要素

バインディング

新しい名前を入力して新しい定義を作成すると、ツールはそのオブジェクトがモデル内に存在していないと認識します。この場合、テキストはグレー表示になります。新しいモデル要素の定義にこの名前を使用する場合、名前を**コミット**しなければなりません（Enter キーを押すだけです）。

既存の定義を参照すると、入力した名前がモデル要素にバインドされます。名前がモデル要素にバインドされると、テキストの色が変わります。名前が解決できない（名前のスペルミス、または可視性ルールに従って定義が可視化されていない）場合、シンボルの下に赤い波線が表示されます。

GUID

UML エンティティにはデフォルトでランダムに生成されたグローバル一意識別子、GUID があります。GUID は、エンティティのライフタイムを通じて変更されません。

新しい要素の自動名前付け

モデル要素を定義する新しいシンボルを追加すると、現在のスコープで一意の名前となるようにシンボルにデフォルト名が作成されます。（テキストを選択せず）入力するだけで、デフォルトの名前を目的の名前に変更できます。

モデル要素のコピーと移動

モデル要素はコピー／貼り付けできます。

モデル要素を参照するシンボルをコピーしてこのシンボルを貼り付けると、既存のモデル要素のプレゼンテーションのみ新たに作成されます。これら 2 つのシンボルの一方を名前変更すると、他方のシンボルの名前も変更されます。2 つとも同じ要素のプレゼンテーションだからです。

ブラウザでモデル要素をコピーした場合、これを他の場所に貼り付けられます（ここでは、モデル要素自体のコピーとなります）。同じスコープに貼り付けると、同じ名前の定義が 2 つあることから競合が生じるため、チェッカーから何らかのレポートを受けます。どちらか一方のモデル要素の名前を変更すると、この競合は解消されます。

また、プロジェクト間でモデル要素をコピーすることもできます。プロジェクト間の定義を構造化された方法で再利用するには、同じモデル（.u2）ファイルで定義された複数のパッケージに定義を挿入します。

モデル要素がコピーできるのと同様に、定義もあるスコープから他のスコープへ、またはあるプロジェクトから他のプロジェクトへドラッグして移動できます。要素を別のファイルに保存することも可能です。第 3 章「ダイアグラムの操作」の 139 ページ、「新しいファイルで保存」を参照してください。

複数ファイルへのモデルの分割保存

また、モデルをいくつかの ファイルに自動的に分割保存することも可能です。このためには、ブラウザでパッケージまたはクラスを選択し、ショートカット メニューから［ファイルの自動作成］を選択します。モデルは、最小単位ではクラス別に個々のファイルに分割されます。

この機能を有効にするには、［ツール］メニューから［オプション］を選択します。［UML 基本編集］タブで、［デフォルトのファイル作成モード］を ［パッケージとクラス］に設定します。

［ファイルの自動作成］は、新しいファイルに保存できる要素用にアクティブになります。［ファイルの自動作成］を選択すると、選択された要素の下のすべてのクラスとパッケージが個別のファイルに格納されます。ファイルにはルート要素と同じ名前が指定されます。この名前のファイルがすでにプロジェクト内に存在する場合は、番号がファイル名に追加されます。

［ファイルの自動作成］は、プロジェクト ノード（.ttp）とモデル ノード用にもアクティブになります。［ファイルの自動作成］は、ライブラリと定義済みパッケージ用にはアクティブになりません。

テキストの強調表示

構文の強調表示

以下の表に、構文カラー テキスト トークンの主なカテゴリを示します。

カテゴリ名	**デフォルト カラー パレット**
名前	黒、赤褐色、グレー
リテラル	茶
キーワード	青（紫がかった）
コメント	緑
エラー	赤
角かっこ、大かっこ、丸かっこ	黒

これらのカテゴリには、テキストの実際のコンテンツや、後でパスに追加された情報に基づくサブカテゴリがあります（以下で詳述します）。マウスのカーソルをトークンに合わせると、トークン タイプやトークンで示されるオブジェクトのタイプなどのセマンティック情報を示すツールチップが表示されます。たとえば「‘`IfAction`’, class X, 2 references are currently bound to this object, CTRL + Click to navigate.」（‘`IfAction`’ とクラス X は、現在、このオブジェクトにバインドされた 2 つの参照です。Ctrl キーを押しながらクリックして、ナビゲートできます。）などと表示されます。

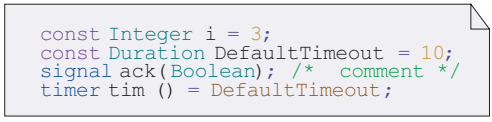


図 7: 構文テキストのカラー表示の例

構文エラー マーカー

テキストに構文エラーがあると、テキストのエラー箇所にマーカーが表示されます。マーカーの位置にカーソルを合わせると、ツールチップに構文のエラーメッセージが表示されます。例：
`'Syntax Error, found token?, expected Name'`

マーカーは上向きの三角形で、高さはテキストの高さの 2 分の 1 以下、テキストのベース ラインのほぼ中央に表示されます。マーカーの色はエラーの重要度によって異なります。

重要度	**色**
致命的なエラー	オレンジ
エラー	赤
警告	黄色
情報	水色

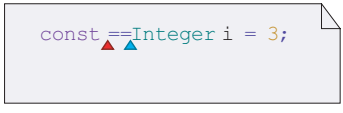


図 8: 構文エラー マーカー

セマンティックの強調表示

テキストの名前には、別の強調表示のルールを適用し、構文の強調表示の枠を広げます。

- 名前テキストのカラー表示のルール

カラー ルールの説明	**強調表示**
定義を示す名前	名前は黒で表示します。
タイプにバインドされた識別子を示す名前	名前は深緑で表示されます。

イベント クラス（シグナル、タイマー、操作）にバインドされた識別子を示す名前	名前は紺で表示されます。
属性、変数、パラメータにバインドされた識別子を示す名前	名前は赤褐色で表示されます。
何を示すか特定できない名前（構文エラーなど）	名前はダーク グレーで表示されます。

- 名前の下線に関するルール

下線のルールの説明	**強調表示**
バインドを解除された識別子を示す名前	赤の波線
使われなくなった識別子を示す名前	緑の波線

強調表示された名前にマウスのカーソルを合わせると、テキストの現在のステータスがツールチップに表示されます。特に、いずれかの下線のある名前にカーソルを合わせると、ある種の状況診断を行うことができます。例：This reference is currently not bound, see AutoCheck log for details.（この参照は、現在バインドされていません。オートチェックのログで詳細を確認してください。）

オブジェクトの場所

UML モデルの複数の場所で、たとえば［モデル ビュー］または出力ペインのオブジェクトをダブルクリックするか、出力ペインのオブジェクトに対して［検索］コマンドを選択して、定義を配置できます。上記の操作を実行すると、黄色のテキスト背景で強調表示された、正確なダイアグラムが表示されます。また、定義のプレゼンテーションが複数ある（あるいは、皆無の）可能性がある場合、プレゼンテーションの作成が起動します。

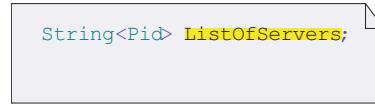


図 9: ロケーション マーカー

名前のナビゲーション

Ctrl キーを押しながらグレー表示されていない、下線のない名前をクリックすると、名前をナビゲートできます。これはツールチップにも表示されます。

プロパティ

必ずしも、モデル要素のプロパティのすべてを、ダイアグラムのプレゼンテーション要素から編集できるわけではありません。このため、モデル要素のプロパティを表示して、編集するためのプロパティ エディタが用意されています。プロパティ エディタはショートカット メニュー（要素を右クリックして、［プロパティ］にカーソルを合わせます）、または、Alt + Enter キーを押して開くことができます。

モデルのチェック

オートチェック

モデルを定義する際、ツールによって継続的にチェックされます。モデルを修正するたびに、出力ウィンドウの［オートチェック］タブが更新されます。また、タブにはエラーと警告も表示されます。

チェック機能の設計により、大きく複雑なモデルの分析にも時間が掛かりません。

エディタのフィードバック

ダイアグラムを編集すると即座にフィードバックが返ってきます。誤った構文を入力すると、エラー箇所に構文エラー マーカーが表示されます。また、ツールチップにエラーの特性が表示されます（たとえば、構文エラー）。

入力した名前が認識されないか（たとえば、スペルミス）、可視性ルールに違反しているため表示されない場合、名前の下に赤い波線が表示されます。

名前が既存の定義に**バインド**されると、通常の振る舞いでは、セマンティックの強調表示が適用され、結果としてテキストの色が変わります。

構文解析

ダイアグラム シンボル内のテキストを編集する際、テキストが正確かどうか解析され、モデルに追加されます。その後、このモデルをもとに、テキストはダイアグラム シンボルに書き戻されます。

このテキスト解析は、完全なモデル ベースのアプローチの結果です。（複数ある中から）1 つの特定の選択構文にテキストを書き込んだ場合、指定した書式が保存されないことがあります。

モデルの復元（**F8**）

構文解析により、変更箇所に構文エラーが見つかった場合、モデルを元の状態に復元できます。テキスト編集で構文エラーが選択された常態で、F8 キーを押して復元を実行します。この操作で編集したテキストがモデル情報から削除されます。

このコマンドを使用するときは以下の注意が必要です。コメントのような、モデルにバインドされていないテキストはすべて消去されます。モデルを初めて作成し、正確なモデル解析が行われていない場合、このコマンドですべてが消去されます。

名前のサポート

定義を参照したい場合、次のいくつかの方法を利用できます。

- プレゼンテーションの作成を使用して、完全なモデル間で素早くブラウズ、ナビゲートできます。
- **名前の完成**

  名前の最初の文字を入力してから（たとえば ca）、Ctrl + スペースバーを押すと、ツールが既存の名前にあてはめて、名前を完成しようとします（たとえば、card）。複数が合致する場合は、名前の完成スクロール メニューが開きます。特殊なケースをいくつか説明します。
  - ピリオド（「.」）の後の入力。名前の完成によって、左側の式の型に対してローカルまたは継承メンバー（構造的機能またはイベント クラス）である文字に一致する候補がリストされます。
  - スコープ修飾子（「::」）の後の入力。名前の完成によって、左側の式の名前空間にある候補がリストされます。
- **既存要素を参照**

  ダイアグラム要素作成ツールバーを使用して（モデル要素を定義または参照できる）新しいシンボルを作成し、ダイアグラム内でマウスの右ボタンをクリックすると、［既存要素を参照］というサブメニューのあるショートカット メニューが表示されます。このサブメニューにはシンボルの種類の定義がリストされ、ここから識別子を選択できます。

モデル全体のチェック

オートチェックの他、手動でモデルをチェックできます。モデル全体をチェックする場合、［すべてをチェック］クイック ボタン（［分析］ツールバー）を使用します。

モデルの部分のチェック

［モデル ビュー］でチェックするモデルの部分を選択します。［選択部分のチェック］クイック ボタン（［分析］ツールバー）を使用します。

エラーと警告

チェッカーがモデルについて何らかの問題を検知すると、出力ウィンドウの［チェック］タブにレポートが表示されます。通常、ダイアグラムまたは、［モデル ビュー］ブラウザで、問題（警告とエラー）ごとに、その原因までさかのぼって追跡されます。この操作は、メッセージをダブルクリックするか、メッセージを選択して右クリックし、ショートカット メニューから［検索］を選択して実行します。

モデルとダイアグラム

ダイアグラム

モデルのさまざまなビュー

ダイアグラムは、モデルのある特定の側面やパートに焦点を合わせた、モデルのプレゼンテーションです。UML の機能の 1 つに、モデルのさまざまなビューを表示する機能があります。これは、モデル要素が複数の場所で参照されることを意味します。通常、モデルを維持する際にこれは問題となりますが、強力なモデル ベースのツールを活用することで、すべての参照が自動更新されるようになります。つまり、モデル要素のプロパティが変更されると、そのモデル要素を参照しているすべての場所でこの変更が反映されます。

プレゼンテーション要素

シンボル

シンボルは、モデル要素とは異なる**プレゼンテーション要素**です。シンボルが削除されても、モデル要素はモデル内に残ります。以下のいずれかの手順を実行すると、モデル要素が削除されます。

- ［モデル ビュー］ブラウザでモデル要素を削除する。
- シンボル上で［モデルからの削除］コマンドを実行する。

クラス シンボルで属性の名前を変更した場合、この変更は、クラス内の他のプレゼンテーションにも反映されます。

（モデル ブラウザ、または、表示されたクラス シンボルのいずれかの）クラスに新しい属性を追加しても、この属性が自動で、すべてのプレゼンテーション要素に表示されるわけではありません。もちろん、この属性をモデル内に配置して、プロパティを可視化するクラス シンボルに簡単に追加することもできます。

クラス シンボルの属性を削除しても、シンボルの属性のプレゼンテーションが削除されるだけです。［モデル ビュー］ブラウザで属性を削除すると、他のクラス シンボルの属性の表示もすべて消失します。

［モデル ビュー］ブラウザでクラス自体を削除すると、このクラスを参照するすべてのシンボルがダイアグラムから消失します。たとえば他のクラスでの属性タイプとして、他の場所でクラスが参照されている場合、クラスを削除するとこれらの参照のバインドが解除されます。この操作はダイアグラムに即座に反映されます（赤い波線で示されます）。また、オートチェックにも反映されます。

プロパティ エディタ

プロパティ エディタを開く

プロパティ エディタは、[モデル ビュー] またはダイアグラムの要素を選択し、ショートカット メニューから [プロパティ] を選択して開きます。プロパティ エディタは、ドッキング ウィンドウとして開きます。他のエディタと同じように、タイトル バーを右クリックして、ドッキングを解除したり他の場所にドッキングできます。プロパティ エディタは、ユーザーが閉じるまで開いたままとなります。

複数のウィンドウ

プロパティ エディタは複数開くことができます。これは、いくつかの要素のプロパティの設定を比較する場合などに便利です。複数のウィンドウを開けるようにするには、1 つのプロパティ エディタ ウィンドウで [選択部分をトラック] を無効にします。

プロパティ エディタ ウィンドウ

プロパティ エディタのビューは、上から順に以下の領域で構成されます（64 ページの図 10 を参照）。

- ウィンドウの左上。選択している要素の名前とアイコンが表示されます。
- [オプション] ボタン。現在のプロパティ エディタのオプションを設定するために使用します。
- フィルタ選択メニュー。表示する要素のプロパティを絞り込みます。
- [ステレオタイプ] ボタン。要素に適用するステレオタイプを選択するために使用します。このボタンをクリックして開くダイアログは、要素のショートカット メニューから [ステレオタイプ] メニュー項目を選択して開くダイアログと同じです。
- 要素のプロパティの表示と編集に使用するコントロール。この領域は、編集する要素と選択したフィルタによって動的に変化します。

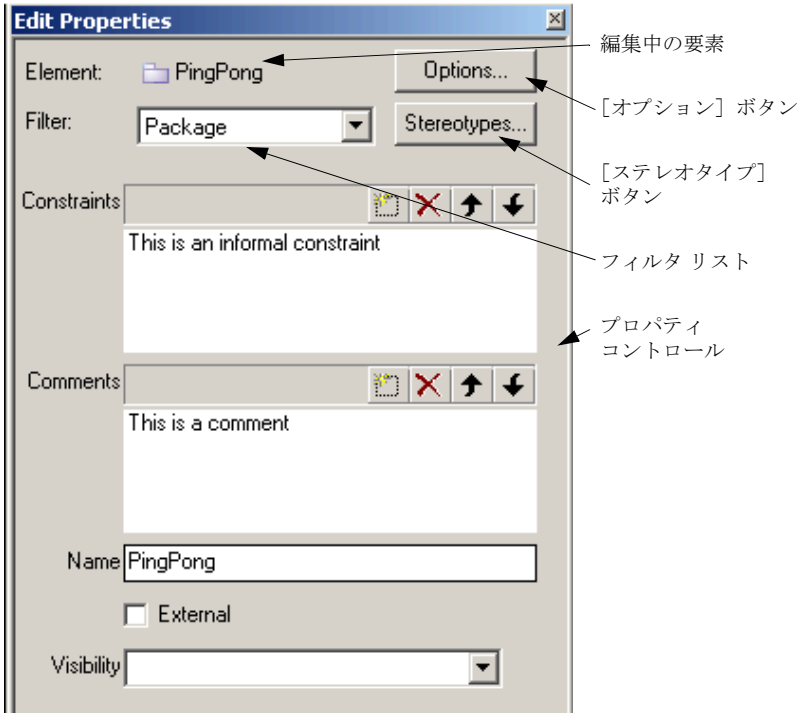


図 10: プロパティ エディタ

フィルタ リストには以下の項目があります（このとおりの順番とは限りません）。

- 編集中の要素のメタクラスの名前。
  この項目を選択すると、プロパティ エディタには要素のメタ特性値が表示されます（65 ページの「複数種類のプロパティ」を参照）。
- 編集中の要素に適用されている各ステレオタイプの名前。
  オプションの拡張（`0..1`）と必須拡張（`1`）を持つステレオタイプがメタクラスに適用されます。ステロタイプの拡張の詳細については、320 ページの「拡張性」を参照してください。ただし、非表示ステレオタイプ（<<hidden>> ステレオタイプが適用されたステレオタイプ）は表示されません。
- コメント
  この項目を選択すると、プロパティ エディタに編集中の要素に適用されたコメントが表示されます。コメントがない場合、要素にコメントを作成するボタンが表示されます。要素に複数のコメントがある場合、最初のコメントが表示されます。

すべてのプロパティ
この項目を選択すると、プロパティ エディタに編集中の要素のすべてのプロパティが表示されます。プロパティ コントロールの順序は、フィルタ リストの対応する項目順序と同じです。

インスタンスを選択している場合のプロパティエディ表示

インスタンスを選択している場合に、上で説明したプロパティエディタのコントロールの一部が意味を成さなくなります。その場合には該当コントロールは表示されません。

- ［Stereotypes］ボタンは表示されなくなります。インスタンスに対しては、ステレオタイプを適用できないからです。
- ［Options］ボタンは表示されなくなります。オプションの一部はインスタンスに当てはまらないからです。
- ［Filter］リストはインスタンスのシグニチャの情報に置き換えられます。

典型的な例としては、モデルビューであるインスタンス（たとえばステレオタイプインスタンス）を選択している場合にプロパティエディタの表示が上記のように修正されます。ただし、クラスのような構造型で型付けされた属性用にタグ付き値を編集している場合にも、インスタンスが選択されています。

複数種類のプロパティ

選択した要素には、原則的にメタ特性値とタグ付き値の 2 種類のプロパティを関連付けられます。どちらのプロパティもプロパティ エディタで編集できます。

メタ特性値

メタ特性値は、要素のメタクラスのメタ特性の値です。
要素の一連のメタ特性は固定されているので（また、UML 標準によってある程度限定されるので）新しいメタ特性を追加することはできません。既存の一連のメタ特性から、あるメタ特性の値だけを表示するように絞り込むことはできます。
このメタ特性値の例には、クラスの「Active」プロパティがあります。

タグ付き値

タグ付き値は、要素に適用されているステレオタイプの属性の値です。メタ特性値と違って、タグ付き値はかなり増える可能性があります。これは、要素に任意数のステレオタイプを適用でき、それぞれに任意数の属性を持たせることができるからです。たとえば、シンボルに表示するアイコンを指定する「Icon File」プロパティなどがあります。他の例を 71 ページの図 14 に示します。

プロパティ エディタのオプション

プロパティ エディタの［オプション］ボタン（66 ページの図 11 を参照）をクリックすると、［プロパティのオプションの編集］ダイアログが表示されます。このダイアログで設定するオプションは、現在のプロパティ エディタが開かれている間のみ有効です。つまり、2 つのプロパティ エディタを開いて、それぞれに別のオプションを設定できます。

注記

オプションの一部は、一般的な［オプション］ダイアログでも設定できます。そこで、すべてのプロパティ エディタに適用できるオプションを設定して保存できます。オプション値の一部はプロパティエディタの一般的なショートカット メニューでも変更できます。

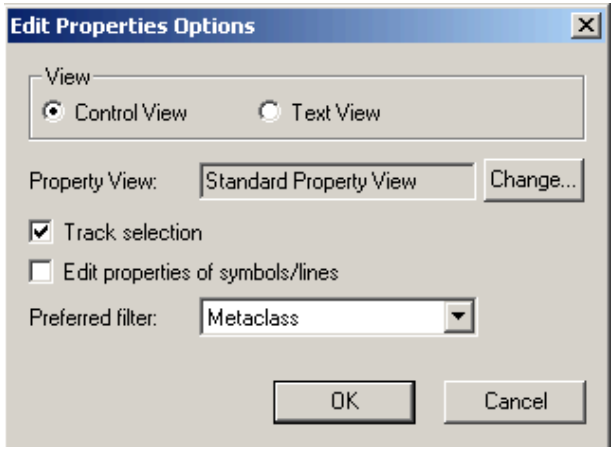


図 11: プロパティ エディタの［プロパティのオプションの編集］ダイアログ

View

プロパティ エディタでは、プロパティ値の編集にデフォルトで［コントロール ビュー］が使用されます。このビューには、要素プロパティの編集に使用するチェックボックスやプルダウン メニューなどがあります。タグ付き値については、UML 構文でのテキスト編集が可能です。テキスト編集は、［テキスト ビュー］で行うことができます。

テキスト編集の利点は、タグ付き値の簡単な説明を入れられることです。さらに、［テキスト ビュー］では、別のプロパティ エディタ、テキスト図、テキスト シンボルなどからプロパティ エディタにタグ付き値を簡単にコピーできます。［テキスト ビュー］を使用する代わりに、ダイアグラム内のステレオタイプ インスタンス シンボルを使用してタグ付き値を編集することもできます。

［コントロール ビュー］で入力したテキスト フィールドの値は、フィールドのエディット モード終了時にモデルにコミットされます。

プロパティ ビュー

プロパティ エディタは、メタモデルを使用してカスタマイズできます。メタモデルによりメタクラスで使用できるメタ特性を指定すると、プロパティ エディタで要素に表示するプロパティを決定してこの情報できるようになります。メタモデルの使用方法の詳細については、73 ページの「プロパティ エディタのカスタマイズ」を参照してください。

選択部分をトラック

プロパティ エディタには、デフォルトで［モデル ビュー］またはダイアグラムで選択した要素のプロパティが表示されます。2 つの要素プロパティの比較を可能にするなど、選択部分のトラック機能を無効にしたほうがよい場合があります。このためには、設定を変更する要素のプロパティ エディタを開きます。プロパティ エディタの［オプション］ボタンをクリックし、［選択部分をトラック］の選択を解除します。これで、他の要素のプロパティ エディタを開き、この新しいプロパティ ウィンドウでモデル内の選択部分をトラックできます。

シンボル / ラインのプロパティを編集

シンボルまたはラインを選択している場合、プロパティ エディタにはデフォルトでシンボルまたはラインに対応するモデル要素のプロパティが表示されます。選択したシンボルまたはラインのプロパティを表示するよう設定するには、このオプションを選択します。

たとえば、クラス シンボルを選択している場合、通常は、プロパティエディタには対応するクラスのプロパティが表示されます。［シンボル / ラインのプロパティを編集］オプションを選択している場合は、対応するクラスではなくクラス シンボルのプロパティが表示されます。

使用するフィルタ

このオプションにより、新しい要素を選択した最優先されるフィルタを指定します。［メタクラス］、［ステレオタイプ］、［コメント］、［すべてのプロパティ］の 4 つの選択肢があります。次回編集項目を変更する際、このオプションが有効になります。

一般的なショートカット メニュー

プロパティ エディタには、コントロール以外の場所でマウスを右クリックしたときに表示される、ショートカット メニューがあります。68 ページの図 12 にショートカット メニューを示します。

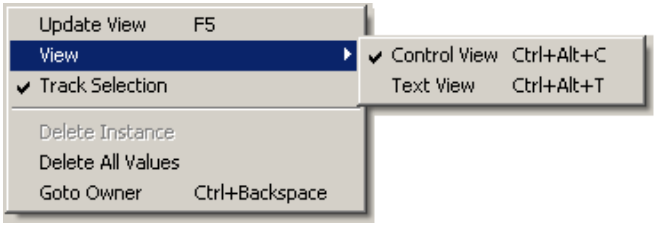


図 12: プロパティ エディタのショートカット メニュー

表示の更新

プロパティ エディタの表示を更新します。プロパティ エディタ以外で値が変更された場合など、プロパティ エディタの内容が自動更新されます。ただし、現在属性値が表示されているステレオタイプに属性が追加された場合や、プロパティ エディタを開いている間にアクティブ プロパティのメタモデルが変更された場合など、手動で表示の更新を行う必要があります。

表示

このメニュー項目は、［オプション］ダイアログ（66 ページの「プロパティ エディタのオプション」を参照）の対応するオプションのショートカットです。

選択部分をトラック

このメニュー項目は、［オプション］ダイアログ（66 ページの「プロパティ エディタのオプション」を参照）の対応するオプションのショートカットです。

インスタンスの削除

このメニュー項目は、適用された 1 つのステレオタイプのタグ付き値を編集する場合に使用できます。インスタンスに含まれるすべてのタグ付き値を削除し、ステレオタイプ インスタンス全体を削除できます。［ステレオタイプ］ダイアログを開き、適用ステレオタイプのリストから編集中のステレオタイプを削除するためのショートカットです。

すべての値の削除

このメニュー項目は、表示されているすべてのプロパティの値を削除するために使用します。デフォルトがあるプロパティはデフォルト値に戻され、ない場合は指定なしとなります。タグ付き値の編集の場合、このメニュー項目により、適用ステレオタイプ インスタンスを残し、すべてのタグ付き値を削除できます。

所有者へ移動

このメニュー項目は、編集中の要素の所有者のプロパティ ページへの移動に使用する便利なショートカットです。たとえば、クラス属性のプロパティを編集すると、[所有者へ移動] によって属性の所有者（クラスなど）を表示できます。

注記

編集のためにインスタンスを選択している場合は、プロパティエディタのショートカットメニュー項目の一部が使用できません。

コントロール ショートカット メニュー

各プロパティ コントロールのショートカット メニューも用意されています。このメニューの内容は、プロパティ コントロールの種類によって異なります。たとえば編集コントロールには、切り取り／コピー／貼り付けなどのメニュー項目があります。すべてのプロパティ コントロールに共通するメニュー項目を、69 ページの図 13 に示します。

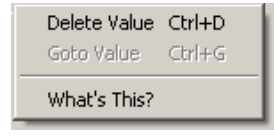


図 13: プロパティ コントロール、ショートカット メニューの例

値の削除

このメニュー項目は、プロパティ コントロールの値を削除するために使用します。プロパティにデフォルトがある場合はデフォルト値に戻ります。デフォルト値がない場合、指定値なしとなります。

値へ移動

リスト コントロールには、モデル内の他の要素のリストの値が表示されます。このコントロールでは、[値へ移動] メニュー項目を使用してリスト内で選択した要素に移動できます。

たとえば、ほとんどの要素には、編集中の要素に添付されたすべてのコメントを表示するコメント リストがあります。これらのコメントの 1 つを選択すると [値へ移動] メニュー項目が使用できるようになります。プロパティ エディタには選択したコメントのプロパティ（通常は 1 つのプロパティ、コメント テキスト）が表示されます。

これは何？

プロパティ コントロールに対応する属性（ステレオタイプまたはメタクラスの属性など）にコメントが添付されている場合、このメニュー項目が使用できます。このメニュー項目を選択すると、コメントにツール チップが表示されるようになります。ス

テレオタイプとメタモデルの設計者は、ステレオタイプとメタクラスのユーザーがプロパティ コントロールに入力すべき値がわかるように、ステレオタイプとメタクラスの属性にコメントを添付できます。

属性にコメントが添付されていない場合でも、「これは何？」テキストが表示されるコントロール（たとえばメタ特性値の表示など）もあります。これは、コントロールに入力する値がモデル要素に翻訳されるテキストである場合です。この場合は、コントロールに入力すべきテキストの種類がツールチップに表示されます。たとえば、コントロールに UML 式を入力する必要がある場合は、ツールチップに「式」と表示されます。

カラー コード

タグ付き値（メタ特性値以外の値など）の編集時、プロパティ エディタでタグ付き値の状態を示すためのカラー コード スキームを使用できます。

適用ステレオタイプ インスタンスで明示的に指定されたタグ付き値は、白いプロパティ コントロールで示されます。

ステレオタイプ インスタンスで指定されていないタグ付き値は、対応するステレオタイプ属性にデフォルト値がある場合、緑のプロパティ コントロールで示されます。

ステレオタイプ インスタンスで指定されていないタグ付き値は、対応するステレオタイプ属性にデフォルト値がない場合、黄色のプロパティ コントロールで示されます。

ステレオタイプの設計者はこのカラー コードを使用して、ステレオタイプ属性の意図をユーザーに知らせることができます。緑の値は、適切なデフォルト値があるので、値の指定は任意であることを表します。黄色の値は、その属性に適切なデフォルト値がないので、値の指定は必須であることを表します。

例 **2:** 色分けされた属性フィールドがあるステレオタイプ ——————————

3 つの属性を持つステレオタイプを考えます。71 ページの図 14 の例では、ステレオタイプ `MyStereo` がクラス x に適用されます。ユーザーが 2 つ目の属性に値を指定すると、このフィールドの色が黄色から白に変わります。

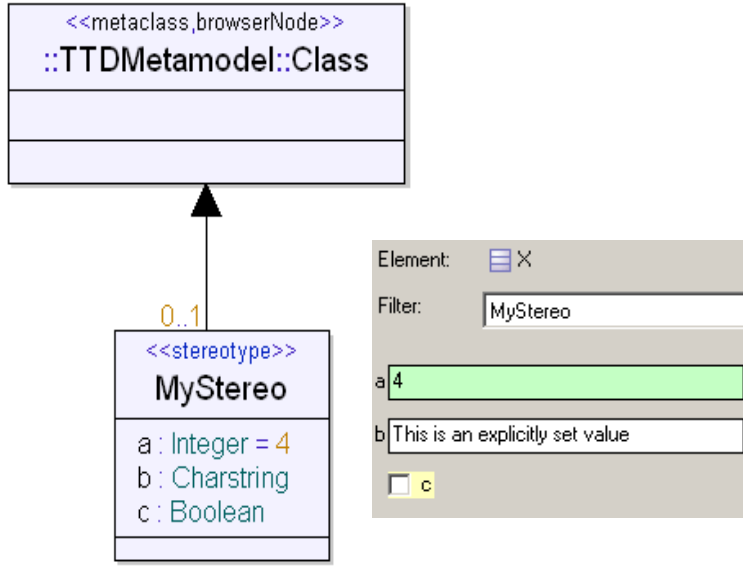


図 14 属性を持つステレオタイプ

あるコントロールのテキストに構文エラーがあるかどうかを示すための色分けもあります。構文チェックは、すべてのコントロールについて、そのテキストが U2 テキスト構文の文法にしたがっているかどうかを確認します。構文エラーのあるテキストは赤で示され、構文エラーがないテキストは黒で示されます。テキストが赤で示された状態のまま編集を終了すると、メタ特性値は編集前の正しい値に戻ります。このようなカラー コードによって、編集中に誤って正しい情報を失うのを防ぐとこができます。

例 **3:** プロパティ エディタでの構文エラーの色分け

ポートの 'Realizes' メタ特性は ID リストを要求します。'signal' は UML のキーワードなので、メタ特性の現在のテキスト（72 ページの図 15 参照）には構文エラーがあります。したがって、この状態で編集モードを終了すると、値は編集前の正しい値（いかなる値でも）に戻ります。

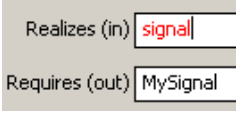


図 15: メタ特性値のエラーと正しい値

プロパティ エディタのカスタマイズ

要素に適用するステレオタイプを設計する際、2 つのユーザー ロールが考えられます。それは、ステレオタイプの属性を決定するステレオタイプの設計者と、ステレオタイプを要素に適用してステレオタイプ属性にタグ付き値を指定する、ステレオタイプのユーザーです。同一人物がこれら両方のロールを持つ場合もありますが、通常はステレオタイプの設計者とユーザーは別人です。

このセクションでは、設計者ロールに焦点を当て、新しいステレオタイプあるいはメタクラスの設計方法について説明します。また、設計者が望む方法でステレオタイプやメタクラスのインスタンスを編集できるように、プロパティ エディタをカスタマイズする方法についても説明します。

プロパティ エディタでは、カスタマイズのために通常 TTDExtensionManagement プロファイルというプロファイルを使用します。これはモデルのライブラリ フォルダにあります。

ステレオタイプの設計

プロパティ エディタで使用するステレオタイプの設計は、以下の手順によって行います。

- 新しいステレオタイプの定義を保管する場所を決定します。ステレオタイプをローカルの現行プロジェクトでのみ使用する場合、適用する要素と同じファイルに追加します。通常は、複数プロジェクトでステレオタイプを使用する場合が多いので、独自のファイルに保存されるパッケージに保管します。ステレオタイプを持つ再利用可能なパッケージを、プロファイル パッケージといいます。そのようなパッケージをツールのライブラリにロードする方法については、1732 ページの「アドイン」を参照してください。
- ステレオタイプに適切な名前を付けます。ステレオタイプの名前は、［ステレオタイプ］ダイアログ、プロパティ エディタのフィルタ リスト、ダイアグラムのシンボルなどに表示されます。
  `TTDExtensionManagement::instancePresentation` ステレオタイプを使用して、ステレオタイプによりわかりやすい表示名を付けると便利な場合があります。指定した表示名は、［ステレオタイプ］ダイアログとプロパティ エディタのフィルタ リストに表示されます。詳細については、76 ページの「TTDExtensionManagement プロファイル」を参照してください。
- ステレオタイプにコメントを付けます。コメントは、ステレオタイプの目的、適用できる要素に関する制約などを表します。コメントは、ステレオタイプを選択した際、［ステレオタイプ］ダイアログの下部に表示されます。また、ステレオタイプにツールチップとして表示されます。
- ステレオタイプに適切なタイプと多重度を持つ属性を追加します。ステレオタイプの属性はどのようなタイプと多重度でもかまいませんが、プロパティ エディタで［コントロール ビュー］を使用して編集する際にサポートされる一連のタイプと多重度を考慮する必要があります。サポートされないタイプまたは多重度のある属性を使用していると、［コントロール ビュー］で属性を変更できません。この場合、［テキスト ビュー］で編集する必要があります。

以下の表に、サポートされるタイプと多重度の組み合わせ、またそれぞれに使用されるグラフィカル コントロールを示します。メタクラスの属性のみに適用可能なタイプと多重度の組み合わせについては、75 ページの「メタクラスの設計」の表を参照してください。

属性のタイプと多重度	プロパティ コントロール
Boolean 単一の多重度	CheckBox
Charstring 単一の多重度	EditControl
Charstring 複数の多重度	EditList
Integer, Natural, Real 単一の多重度	EditControl
Enumeration 単一の多重度	DropDownMenu （各リテラルに 1 項目）
Enumeration 複数の多重度	CheckBoxList （各リテラルに 1 チェックボックス）
Structured type（クラスなど） 必須、単一の多重度（1）	Group （構造型の各属性に 1 つのサブ コントロール）
メタクラス タイプ 単一の多重度 参照	DropDownMenu （メタクラスの可視定義ごとに 1 項目）
メタクラス タイプ 複数の多重度 参照	EditControl

上記タイプのシンタイプもサポートされます。

- 属性のデフォルト コントロールが適切ではない場合、属性に `TTDExtensionManagement::extensionPresentation` ステレオタイプを適用し、タグ付き値としてカスタム コントロールを指定できます。詳細については、76 ページの「TTDExtensionManagement プロファイル」を参照してください。
- ステレオタイプのプロパティ ページに「値以外」のコントロールを追加できます。たとえば、プロパティ ページに静的テキストまたはボタンを追加できます。このためには、ステレオタイプに `TTDExtensionManagement::instancePresentation` ステレオタイプを適用し、`nonValueControls` 属性のタグ付き値として追加のコントロールを指定します。

- 各ステレオタイプ属性にコメントを添付できます。コメント テキストは、属性に対応するコントロールの［これは何？］ショートカット メニュー項目を選択すると表示されます。
- ステレオタイプ間で継承を利用できます。派生したステレオタイプのプロパティ ページには、派生ステレオタイプの属性の後に、ベースのステレオタイプ属性が表示されます。
- ステレオタイプ適用可能な要素の種類を指定します。このためには、ステレオタイプとメタクラスの間に拡張を設定します。これは、指定したメタクラスにステレオタイプを適用できることを示します。UML セマンティックでは、ステレオタイプに拡張がない場合、いかなる要素にも適用できません。
  拡張メタクラスのすべてのインスタンスにステレオタイプを自動的に使用可能とする場合、拡張を必須に設定（拡張ラインに「1」と入力）します。これで、編集中の要素に適用しなくても、プロパティ エディタのフィルタ リストにこのステレオタイプが表示されるようになります。
  ステレオタイプを手動で適用する場合、拡張を任意に設定（拡張ラインに「`0..1`」と入力）します。
  複数の拡張を使用することもできます。指定したメタクラスのどの要素にも、ステレオタイプを適用できます。

これで、新しいステレオタイプをテストする準備が整いました。適切な種類の要素（ステレオタイプによって拡張されたメタクラスの要素など）を作成します。作成した要素の場所からステレオタイプが見えるようにしてください。作成した要素のプロパティ エディタを開き、新しいステレオタイプのプロパティ ページを確認します。拡張を任意と指定した場合、［ステレオタイプ］ボタンを使用してまずステレオタイプを適用します。

メタクラスの設計

メタクラスの設計スコープは、ステレオタイプの設計の場合とほとんど同じです。大きな違いは、プロパティ エディタでメタクラスを使用できる要素の指定方法です。メタクラスでは、これはメタクラスを記述するクラスに `<<metaclass>>` ステレオタイプを適用することで実行されます。通常のクラスではなくメタクラスにするのが、このステップです。`base` 属性のタグ付き値によって、新しいメタクラスのベースとする組み込み UML メタクラスの名前を指定します。

注記

メタクラスの設計方法を覚える手始めとして、すべてのモデルでライブラリとして使用可能な TTDMetamodel プロファイルを学習すると良いでしょう。次のセクションで、独自のメタクラスとその属性のベースとして使用する、内蔵メタクラスとメタ特性の名前に関する情報を示します。また、指定メタクラスに対応するようプロパティ エディタをカスタマイズするための TTDExtensionManagement プロファイルについても説明します。
これは、プロパティ エディタの［オプション］ダイアログで［Standard Property View］と呼ばれる、TTDMetamodel です。

ステレオタイプとは異なり、メタクラスにはまったく新しい属性は指定できません。メタクラスのすべての属性は、基底メタクラスの既存メタ特性をベースにする必要があります。このためには、メタクラス属性に `metafeature` ステレオタイプを適用します。メタクラス属性の名前が対応するメタ特性と同じ名前の場合、タグ付き値 `base` は省略できます。その他の場合、指定は必須です。

注記

注意深いユーザーは、TTDMetamodel 内のメタモデル属性の一部が基底メタクラスのメタ特性に対応していないことに気づくでしょう。クエリ機能があり、`<<queryFeature>>` ステレオタイプを使用して、モデルからエンティティを算出するクエリエージェントを指定します。クエリ機能はプロパティエディタには表示されません。モデルビューにのみ表示されます。

以下の表に、メタクラスの属性にのみ適用可能なサポートされるタイプと多重度の組み合わせ、またそれぞれに使用されるグラフィカル コントロールを示します。ステレオタイプ属性に有効な組み合わせを示す表（ステレオタイプの設計）と比較してみてください。

属性のタイプと多重度	プロパティ コントロール
Metaclass type 単一の多重度 合成	EditControl
Metaclass type 複数の多重度 合成	EntityList

新しいメタクラスが設定できたら、パッケージに配置し、そのパッケージを独自のファイルに保管します。定義済みステレオタイプ `<<propertyModel>>` をパッケージに適用しておく必要があります。そして、プロファイルをロードするためのアドインの通常の作成手順を行います。プロファイルをロードしたら、プロパティ エディタの［オプション］ボタンを使用して、プロパティ エディタで使用するプロパティ ビューとしてプロファイル パッケージを指定します。

TTDExtensionManagement プロファイル

TTDExtensionManagement プロファイルには、独自のステレオタイプとメタクラスのプロパティ ページのカスタマイズを可能にする、ステレオタイプとクラスが含まれています。ここでは、このプロファイルの詳細について、その使用例を挙げて説明します。

ステレオタイプ

プロファイルには、プロパティ エディタと関係のある 3 つのステレオタイプ `instancePresentation`、`extensionPresentation`、`filterStereotypes` が含まれています。

instancePresentation

`instancePresentation` ステレオタイプは、ステレオタイプまたはメタクラスに適用して、ステレオタイプまたはメタクラスのインスタンスのプロパティ エディタでの表示方法をカスタマイズできます。

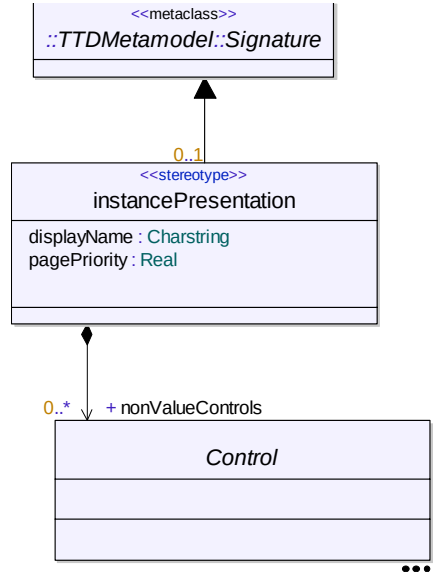


図 16: <<instancePresentation>> ステレオタイプ

displayName: Charstring

この属性は、ステレオタイプまたはメタクラスのインスタンスの表示名を指定します。表示名は、プロパティ エディタのフィルタ リストと［ステレオタイプ］ダイアログに表示されます。また、ツールの他の場所、たとえばツールチップや［モデル ビュー］などにも表示されます。

この属性にタグ付き値を何も指定しないと、表示名としてステレオタイプまたはメタクラスの名前が使用されます。

pagePriority:Real

この属性は、プロパティ エディタのフィルタ リストとプロパティ ページ（フィルタを使用する場合）での表示順序を制御します。ページ優先順位が高いステレオタイプは、低いページ優先順位番号を持つステレオタイプのインスタンスより前に配置されます。どのようなページ優先順位を指定しても、指定なしより優先順位が高いと見なされます。

注記

ページ優先順を指定する場合は、単一の数値を使用してください。それ以上複雑な式は評価されません。

nonValueControls:Control[*]

この属性は、特定の属性に対応しないプロパティ ページ内のコントロールなど、「値以外」のコントロールを指定します。例として静的テキストなどの「付属品」が挙げられますが、ボタンなどのように振る舞いを伴うコントロールの場合もあります。

extensionPresentation

`extensionPresentation` ステレオタイプ（78 ページの図 17）は、ステレオタイプまたはメタクラスの属性に提供し、属性に対応するコントロールのプロパティ エディタでの表示をカスタマイズできます。

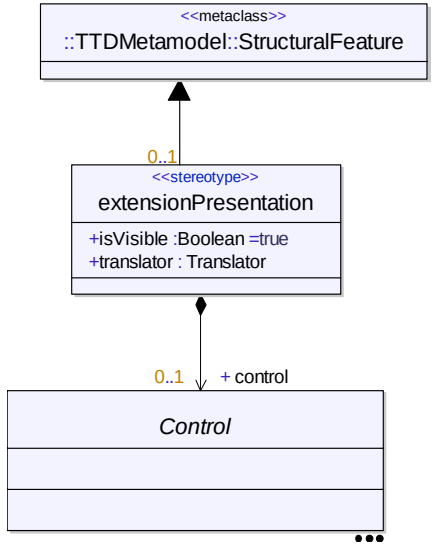


図 17: <<extensionPresentation>> ステレオタイプ

isVisible:Boolean

この属性は、属性のコントロールのプロパティ エディタでの表示／非表示を制御します。属性のコントロールを完全に非表示にする場合は、値を偽に設定します。

translator: Translator

この属性は、メタクラスのタイプを持つパートに排他的に使用します。75 ページの「メタクラスの設計」で説明したように、このような属性には、プロパティ エディタでは EditControl（単一の多重度の場合）または EntityList（複数の多重度）が使用されます。この場合、コントロールに入力されたテキストは UML テキスト構文なので、テキストの解釈のためトランスレータが必要です。Translator 列挙には、UML 文法の使用可能なエントリ ポイントごとに 1 つのリテラルが含まれます。Translator 列挙は非表示（内部）プロファイルにありますが、そのリテラルの名前は次のようにして表示できます。

- 右クリックして［既存要素を参照］を選択し、クラス図で列挙シンボルを作成する。
- 表示されたリストから、`U2ParserProfile::Translator` を選択する。
- 列挙シンボルを右クリックし、［表示／非表示］サブメニューから［Show Literals］を選択する。

注記

［参照の一覧表示］コマンド（ショートカット メニュー）を使用して、TTDMetamodel プロファイルでの Translator 列挙の使用方法を確認できます。たとえば、リテラル PEP_Multiplicity の参照のリストから、StructuralFeature::Multiplicity 属性のトランスレータとして使用されていることがわかります。このように、このトランスレータは、UML の多重度構文を解析するために使用されます。

control:Control[0..1]

73 ページの「ステレオタイプの設計」で説明したように、プロパティ エディタは属性のタイプと多重度、ときには集約の種類に応じて、デフォルトのコントロールを使用します。`control` 属性は、属性のデフォルト以外のコントロールの使用、あるいはデフォルト コントロールのプロパティの変更を可能にします。

例 **4:** ［テキスト ビュー］を使用したカスタム コントロールの指定

```
extensionPresentation(.
  control = EditControl(.
    text = "My Control",
    autoLayout = GrowRight
  .)
.)
```

Control クラスは抽象クラスです。このクラスはプロパティ エディタでサポートされるコントロールごとに 1 つの派生クラスを持ちます。

filterStereotypes

`filterStereotypes` ステレオタイプは、パッケージに適用して、そのパッケージ内の要素を選択したときにプロパティ エディタに表示されるステレオタイプの数を減らせます。

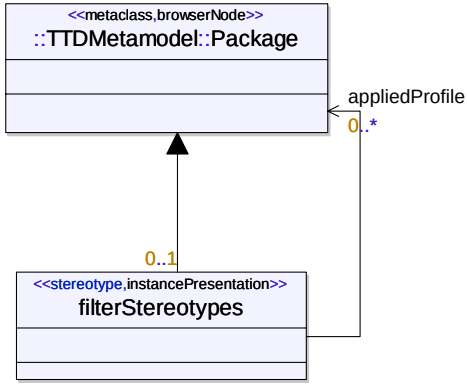


図 18: <<filterStereotypes>> ステレオタイプ

appliedProfile:Package[*]

このリストのプロファイル パッケージを指定すると、プロパティ エディタと［ステレオタイプ］ダイアログには、`filterStereotypes` ステレオタイプが適用されたパッケージで選択した要素に対して、これらのパッケージで定義されたステレオタイプのみ表示されます。

コントロール モデル

`TTDExtensionManagement` プロファイルには、プロパティ エディタで使用されるグラフィカル コントロールを表すさまざまなコントロール クラスがあります。利用可能なすべてのクラスを確認するには、クラス図 `Controls` を参照してください。

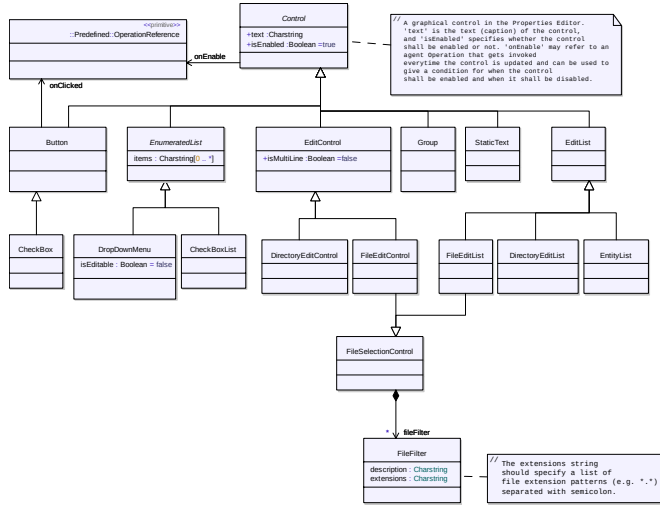


図 19: コントロール クラス

Control

`Control` クラスは、すべてのコントロール クラスの基底クラスです。

text: Charstring

この属性は、コントロールに使用するキャプションを指定します。この属性に何も指定しないと、キャプションは編集中のステレオタイプまたはメタクラスの属性の名前になります。

isEnabled:Boolean

デフォルトで、コントロールは有効になっています。つまり、表示された値の編集に使用できます。この属性を無効に設定すると、コントロールは無効になります。状況によっては、この属性の設定を無視して、プロパティ エディタによってコントロールを無効にすることがあります。編集中の要素に含まれるファイルが読み取り専用の場合、また、派生したメタクラス属性の場合などがこれに当てはまります。

onEnable:Operation

この属性は、コントロールを有効にすべき時期を動的に制御します。この属性にエージェント操作が指定された場合、プロパティ エディタによってコントロールを有効にするかどうかを決定する際、必ず呼び出されます。エージェント呼び出しのモデル コンテキストは編集中の要素です。呼び出しには以下のパラメータがあります。

- `[out] enable :Boolean`

  コントロールを無効にするには、エージェントによってこの out パラメータを無効にします。デフォルトで、コントロールは有効になっています。
- `stereotypeInstance :Entity`

  プロパティ ページで編集中の、コントロールを含むステレオタイプ インスタンス。このパラメータは、編集中のインスタンスがステレオタイプ インスタンスの場合のみ渡されます。

注記

`isEnabled` を無効にすると、`onEnable` エージェントは起動しません。

参照

第 57 章「エージェント」

Button

`Button` クラスは、クリック可能なボタンを表します。これは、値の編集用ではなく、プロパティ ページの値以外のコントロールに使用されます。CheckBox はトグル ボックス コントロールのある特殊なボタンで、ブール値の編集に使用できます。

onClicked:Operation

この属性は、ボタンをクリックしたときに実行される振る舞いを指定します。この属性により、ボタンをクリックしたときに実行されるエージェント操作を指定できます。エージェント呼び出しのモデル コンテキストは編集中の要素です。エージェント呼び出しにパラメータはありません。

EditControl

`EditControl` は、文字列値の編集に使用できます。編集中の文字列がディレクトリまたはファイルの名前の場合、特化された 2 つのバージョンのクラスが使用できます。これらのクラスにより、ディレクトリまたはファイルの選択ダイアログを開く参照ボタン［...］を追加できます。これで、コントロールに名前を手動で入力する必要がなくなります。

InstanceEditControl と呼ばれる特殊な EditControl があります。このコントロールはインスタンス（クラスのインスタンスなど）の編集のために使用されます。インスタンスはテキスト構文を使用してコントロール内に表示されますが、編集のためには参照ボタン［...］を使用します。このボタンを押すと、もう 1 つのプロパティエディタが開き、選択したインスタンスを編集できます。

isMultiLine:Boolean

デフォルトで、編集コントロールにはテキストが 1 行だけ表示されます。この属性を有効にすると、コントロールで複数行の編集が可能になります。同時に 2 行以上のテキストを表示するときは、コントロールの上下サイズを拡大する必要があります。この方法については、PositionedControl を参照してください。

EditList

EditList コントロールは、文字列のリストの編集に使用できます。このコントロールには、リストに新しい文字列を作成するボタンとリストから選択した文字列を削除するボタンが含まれます。さらに、選択した文字列をリスト内で上下に移動するための 2 つのボタンもあります。文字列の移動には、直接文字列を選択してドラッグアンドドロップする方法もあります。

図 20: 文字列の作成、削除、移動用のボタンをもつ EditList コントロール

編集中の文字列がディレクトリまたはファイルの名前の場合、EditList の特化バージョンである 2 つのクラスを使用できます。このクラスは、DirectoryEditList と FileEditList であり、ディレクトリオープン用またはファイル選択用のダイアログを開く参照ボタン [...] を追加します。これによって直接名前を入力する手間を省けます。

また、EntityList と呼ばれる特別な EditList があります。これは、他のメタクラスで型付けされる合成であるメタクラス属性（メタ特性など）のコントロールとして使用できます。EntityList 内の各編集項目は、モデル内の要素です。このような要素についてコントロールに表示される文字列は、その要素の UML テキスト構文です。

もう 1 つの特別な EditList は InstanceEditList です。これは、インスタンス（クラスのインスタンスなど）のリストを編集するために使用されます。インスタンスはテキスト構文を使用してコントロール内に表示されます（各行に 1 インスタンス）。インスタンスを編集するには、そのインスタンスをダブルクリックし、表示される参照ボタン [...] をクリックします。この操作でもう 1 つのプロパティエディタが開き選択したインスタンスの編集ができます。

StaticText

StaticText は、プロパティ ページの付属品として使用できる、値以外のコントロールです。これは、プロパティ エディタのコントロールに対する値の指定方法を示す静的テキストを追加する場合などに使用できます。

EnumeratedList

EnumeratedList は、列挙型要素のリストを編集するコントロールの共通ベースとなる抽象クラスです。このクラスの 2 つの具体的な特化として、DropDownMenu と CheckBoxList があります。

items: Charstring[*]

列挙型リストを列挙タイプの属性のコントロールとして使用される場合、列挙のリテラルごとに 1 項目が含まれます。各項目の名前は、デフォルトで対応するリテラルの名前です。`items` 属性の値として文字列のリストを指定すると、リストの項目名をカスタマイズできます。

DropDownMenu

`DropDownMenu` は、ドロップダウンメニューで編集を行う項目のリストです。

isEnabled:Boolean

デフォルトで、ドロップダウン メニューにある項目は 1 つしか選択できません。この属性を有効にすると、ドロップダウン メニューの編集が可能になり、項目名の手入力ができるようになります。

CheckBoxList

`CheckBoxList` は、チェックボックスのリストで編集を行う項目のリストです。したがって、このコントロールは、複数のリスト項目の選択が可能です。

Group

`Group` は、他のコントロールを包含するコンテナ コントロールです。通常は、構造型で多重度 1 のパート型属性用のコントロールとして使用します。構造型の属性ごとに 1 つのサブコントロールが含まれます。

ColorControl

`ColorControl` 整数型の属性に使用できます。値は、RGB の 3 つのコンポーネントで現される色参照として解釈されます。

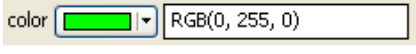


図 21: 緑色を値として指定した ColorControl

色値は矢印ボタンをクリックして表示される標準の色設定リストボックスか、RGB に対応する数値の直接入力で編集できます。

QueryControl

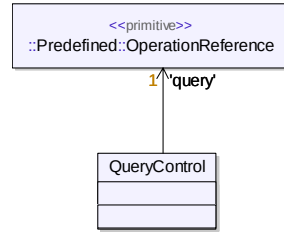


図 22: QueryControl の定義

`QueryControl` の外観は DropDownMenu と似ています。ただし、エンティティ数の固定されたリストではなく、クエリを通して動的にエンティティ数が変動するリストになっています。コントロールの値はクエリの結果から選択されたエンティティへの参照です。

query: Operation

この属性は、リストにデータを入れるために実行されるクエリエージェントへの参照です。

NavigationButton

`NavigationButton` はメタクラスによって型付けされる、単一多重度のメタ特性向けのコントロールとして使用できます。つまり、コントロールの値はモデル内の他のエンティティへの参照です。ボタンが押されると、そのエンティティを表示するプロパティページが開きます。

Navigation ボタンはモデル内の 2 つのエンティティに関係性があるときに使用でき、一方のエンティティのプロパティページからもう一方のページへのナビゲーションを容易にします。

GotoOwnerButton

`GotoOwnerButton` は、編集している要素の合成の親要素にナビゲートするための NavigationButton の特化形です。

ValueControl

コントロール クラスには、値の表示と編集が可能な `ValueControl` クラスを継承するものがあります。

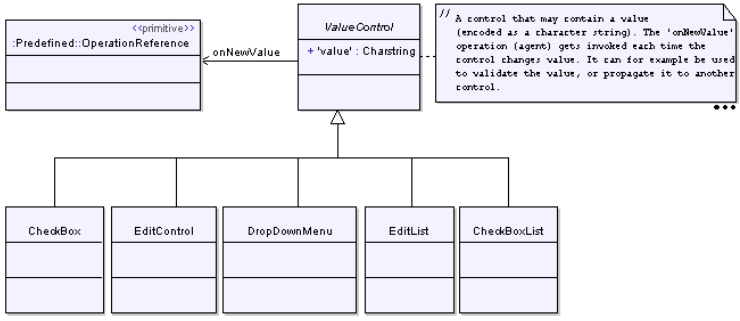


図 23: ValueControl クラス

value: Charstring

この属性は、プロパティ エディタの内部でコントロールの値の表現を保持するために使用されます。しかし、これを明示的に使用してコントロールに常に特定の値を表示させることもできます。

onNewValue:Operation

この属性は、コントロールに新しい値が入力されたときに実行されるエージェント操作を指定します。これは、コントロールに入力された値の有効性の確認や、他のコントロールへの値の移動のために使用できます。新しい値が設定される直前に呼び出されます。編集中の要素をモデル コンテキストとし、以下のパラメータを持ちます。

- `attribute :Entity`

  編集中の属性（ステレオタイプまたはメタクラス属性）。
- `newValue :Entity`

  コントロールに設定される新しい値。
- `stereotypeInstance :Entity`

  編集中の属性がステレオタイプ属性の場合、このパラメータは変更しようとするステレオタイプ インスタンスです。それ以外の場合、このパラメータは渡されません。

PositionedControl

`PositionedControl` クラスは、図形としての位置とサイズに関連するコントロールのプロパティを表します。デフォルトで、コントロールの配置を決定するため、プロパティ エディタは単純なオートレイアウトを適用します。属性は左揃えで上から下に

配置されます。コントロールのオートレイアウト ポジションは前のコントロールとの関連で計算されます。`PositionedControl` クラスの属性により、このレイアウトをある程度カスタマイズできます。

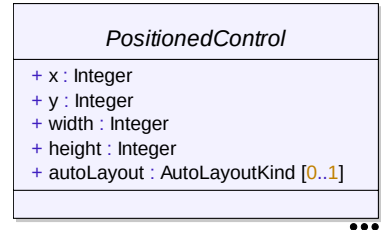


図 24: PositionedControl

x:Integer

この属性に値を指定して、コントロールのデフォルトの水平ポジションを置き換えます。

y:Integer

この属性に値を指定して、コントロールのデフォルトの垂直ポジションを置き換えます。

注記

コントロールのデフォルト位置を上書きするには、x 軸と y 軸の両方の値を指定する必要があります。コントロールに指定した位置は、デフォルトを使用する以降のコントロールにも影響を及ぼします。

width:Integer

この属性に値を指定して、コントロールのデフォルトの幅を置き換えます。

height:Integer

この属性に値を指定して、コントロールのデフォルトの高さを置き換えます。

autoLayout: AutoLayoutKind

この属性は、プロパティ エディタ ウィンドウのサイズ変更がコントロールに与える影響を決定する、オートレイアウト アルゴリズムのオプションを指定します。この属性には、以下の値を使用できます。

- `GrowRight`

  プロパティ エディタ ウィンドウのサイズを拡大すると、コントロールが右方向に拡大します。この振舞い方はほとんどのコントロールのデフォルトです。

- `GrowBottom`

  プロパティ エディタ ウィンドウのサイズを拡大すると、コントロールが下向に拡大します。
- `GrowRightAndBottom`

  プロパティ エディタ ウィンドウのサイズを拡大すると、コントロールが右下方向に拡大します。

プレゼンテーションの作成

［プレゼンテーションの作成］ダイアログは、［モデル ビュー］から［新規］コマンドを使用してダイアグラムの作成を行う代わりに、モデルへの適切なエントリ ポイントを提供します。このダイアログは、要素のショートカット メニューから開くことができます。

［プレゼンテーションの作成］ダイアログ

［プレゼンテーションの作成］ダイアログにはタイトルと一連のタブが表示されます。ダイアログのタイトルには、［プレゼンテーションの作成］の対象となる現在のエンティティのタイプと名前が表示されます。個々のタブには、タブの説明と選択肢が表示されます。

タブの選択肢をクリックすると、ダイアログが閉じ、必要に応じてモデル要素、シンボル、ラインまたはダイアグラムが作成され、それらにナビゲートします。

新しいシンボル

［新しいシンボル］タブを使用して、既存ダイアグラム内で現在のエンティティのシンボルを作成するか、またはエンティティのプレゼンテーション要素を含む新規ダイアグラムを作成して、シンボルを作成できます。

ダイアグラムの作成

［ダイアグラムの作成］タブでは、モデル ビューの作成ルールに従って新しいダイアグラムを作成します。このタブで、現在のエンティティの下にダイアグラムを作成できます。これは、［モデル ビュー］のショートカット メニューから［新規］を選択して、ダイアグラムを作成するのと同じです。

［場所］カラム

モデル内の選択肢の場所です。

［ダイアグラム名］カラム

選択肢の名前です。

［アイテムの種類］カラム、［ダイアグラムの種類］カラム

記述されたエンティティの種類です。たとえば、クラス、シンボル、クラス図などがあります。

参照

モデルのナビゲートと作成

第 3 章「ダイアグラムの操作」の 144 ページ、「シンボルの追加」

モデル ナビゲータ

モデル ナビゲータは、出力ウィンドウの［ナビゲート］というタブのことです。このタブによって、モデルのエンティティのあらゆる側面をブラウズ、またはナビゲートできます。

モデル ナビゲータは、モデルのナビゲーションに適切かつ強力なツールを提供します。［モデル ビュー］には階層構造のスコープ ビューでモデルが表示されますが、モデル ナビゲータにはさまざまなビューがあり、モデルの内部関係に基づいてモデルを詳しく調べられます。

また、モデル ナビゲータにより、以下を実行できます。

- ダイアグラムを選択して表示する。
- 現在のエンティティを示すシンボルまたはラインへナビゲートする。
- 現在のエンティティに関連したエンティティへのナビゲーション ショートカットをとる。

［モデル ビュー］のショートカット メニューまたはエディタのショートカット メニューから［モデル ナビゲータ］を選択すると、モデル ナビゲータが開きます。

モデルのナビゲートと作成

ダイアグラムまたはその要素をダブルクリックすると、モデルのナビゲートまたは作成が可能になります。

- ダブルクリックした要素を表すプレゼンテーション要素がある場合、この要素のダイアグラムの［ナビゲート］タブが表示される。
- ダブルクリックした要素を表すプレゼンテーション要素がない場合、［プレゼンテーションの作成］ダイアログが表示される。

モデル ナビゲータのタブ

［モデル ナビゲータ］タブ自体にも一連のタブがあります。これらのタブには、タブの説明と選択肢が表示されます。表示されるタブは現在のエンティティによって異なります。ウィンドウが表示されたときに選択されているタブは、以下の基準に従って決まります。

- 最後に使用したタブ
- 適切なタブで最優先されるもの

各カラム ヘッダーの右側の垂直バーをドラッグして、カラム幅を変更できます。

ソート

タブの選択肢は、名前カラムを基準にして昇順にソートされています。タブに名前カラムがない場合、タイプまたはインデックス番号カラムが基準になります。

カラム ヘッダーをクリックして手動でソートすることもできます。もう一度クリックするとソートの順番が逆になります。

タブのカテゴリ

モデル ナビゲータのタブは、以下の 2 つのグループに分類できます。

- ［モデル ビュー］またはダイアグラムに選択肢が表示されるタイプのタブ。このグループには**プレゼンテーション タブ**と**リンク タブ**があります。
- モデル ナビゲータのフォーカスを新しいモデル要素に移動する（CTRL+ マウスクリック）タイプのタブ。これらのタブは**エンティティ タブ**と呼ばれます。

それぞれのタブ グループの詳細を以下に示します。

- **プレゼンテーション タブ**

  プレゼンテーション タブの選択肢をクリックすると、ダイアグラムのシンボルやライン（［シンボル］タブ）、または、ダイアグラム自体（［ダイアグラム］タブ）にナビゲートします。
- **リンク タブ**

  ［リンク］タブの選択肢をクリックすると、ダイアログが閉じて、他のリンクの終端にナビゲートします。
- **エンティティ タブ**

  エンティティ タブの選択肢を CTRL+ クリックすると、クリックした選択肢がモデル ナビゲータにフォーカスが移ります。つまり、クリックした選択肢が現在のエンティティとなります。現在のエンティティは［モデル ビュー］で選択します（可能な場合）。このカテゴリには、［パッケージ］、［特性］、［お気に入り］、［定義］、［ショートカット］、［参照］、［モデル インデックス］、および［最近］タブがあります。

モデル ナビゲータのタブは以下の表に示す順に配置されています。

優先順位	タブ名	カテゴリ
1	シンボル	プレゼンテーション
2	ダイアグラム	プレゼンテーション
3	リンク	リンク
4	パッケージ	エンティティ
5	特性	エンティティ
6	お気に入り	エンティティ
7	定義	エンティティ
8	ショートカット	エンティティ
9	参照	エンティティ
10	モデル インデックス	エンティティ
11	最近	エンティティ

ナビゲーション

選択肢をダブルクリックすると、［モデル ビュー］とダイアグラムの両方に選択肢が表示されます（可能な場合）。

CTRL キーを押しながらクリックまたはダブルクリックすると、再びモデル ナビゲータの焦点がクリックした選択肢になります。また、選択肢が［モデル ビュー］とダイアグラムの両方に表示されます（可能な場合）。

Shift キーを押しながらクリックまたはダブルクリックすると、選択肢は［モデル ビュー］のみに表示されます。ダイアグラムには表示されません。

モデル ナビゲータのタブとショートカット メニューに、最近使用したモデル ナビゲータのエンティティのリストが表示されます。このリストで、最近、現在のモデル ナビゲータ エンティティとして使用したエンティティを、再度モデル ナビゲータの焦点にできます。

プレゼンテーション タブ

シンボル

［シンボル］タブには、現在のエンティティに関連したシンボルとラインが表示されます。

ダイアグラム

［ダイアグラム］タブには、現在のエンティティと密接に関連したダイアグラムが表示されます。

リンク

［リンク］タブには、現在のエンティティの外部から、および、外部へのハイパーリンクのリストがあります。リンクをクリックして、そのリンクに関連付けられたリンクのエンド ポイントにナビゲートします。

エンティティ タブ

パッケージ

［パッケージ］タブには、現在のエンティティを含むパッケージで表示される定義がすべて一覧表示されます。

特性

現在のエンティティがクラスまたは類似したものである場合（正確には、現在のエンティティが分類子であるか、または分類子に含まれている場合）、［特性］タブにクラスの定義と継承された定義のリストが表示されます。

定義

［定義］タブでは、現在のエンティティのスコープのローカル定義と継承された定義がすべて一覧表示されます。

参照

［参照］タブには、定義が使用されている場所に素早くナビゲートできるよう、現在の定義タブへのモデル参照のリストが表示されます。このタブに含まれる情報は［モデル ビュー］のショートカット メニューの［参照の一覧表示］と類似しています。

ショートカット

［ショートカット］タブで、モデルの一般に利用される関係を素早くナビゲートします。最も一般的なショートカットについては、［ショートカット］カラムに関するテキストで説明します。

お気に入り

モデル内で再度ナビゲートしたい場所を選択するため、［お気に入り］タブでお気に入りの設定方法やナビゲートの方法を確認します。このタブの内容は、現在のツール セッションのみで維持されます。リストの項目を追加または削除するには、リストで［…の追加］（または ［…の削除］）、または、［すべてのアイテムの削除］行をクリックします。

［モデル ビュー］内のショートカット メニューから［お気に入り］を選んで、選択されている要素をこのリストに追加することもできます。

モデル インデックス

［モデル インデックス］タブには、名称未設定パラメータ（戻りパラメータ）以外のモデル定義がすべてアルファベット順にリストされます。［検索］ダイアログの説明も参照してください。

最近

［最近］タブで、モデル ナビゲータが焦点を合わせたエンティティをトラックして、最近使用したエンティティを再度モデル ナビゲータの焦点にできます。このタブの代りに、最近使用したモデル ナビゲータのエンティティを最大 5 つ表示するショートカット メニューを利用することもできます。

カラム

以下はモデル ナビゲータに表示されるカラムのリストと、表示される情報の簡単な説明です。

［インデックス］カラム

この列は、［参照］タブと［お気に入り］タブにあります。このカラムにはエンティティがアクセスされた順番を示す数字が表示されます。数字が小さいほど、最近アクセスされたことを意味します。

［リンク］カラム

現在のエンティティへの外部からのリンクと外部へのリンクの数。

［場所］カラム

モデル内の選択肢の場所

［名前］カラム、［ダイアグラム名］カラム

選択肢の名前

［ページ］カラム

ダイアグラムのページ番号。このカラムは、［ダイアグラム］タブにあります。

［ロール］カラム

参照リストにおける現在のエンティティのロールを示すリスト。このカラムは、［参照］タブにあります。

［ショートカット］カラム

このカラムには、現在のエンティティからさまざまな関連エンティティへのショートカットがリストされています。このカラムは、［ショートカット］タブにあります。以下に、［ショートカット］カラムに表示されるショートカットの例をいくつか挙げます。

- ［スコープ］ショートカット：現在のエンティティを含むスコープ エンティティをモデル ナビゲータの焦点とします。
- ［コンテナ］ショートカット：現在のエンティティを所有するエンティティをモデル ナビゲータの焦点とします。
- ［モデル ルート］ショートカット：現在のエンティティのモデル ルートをモデル ナビゲータの焦点とします。特に、2 つ以上のモデルを持つワークスペースがある場合、このショートカットは便利です。
- ［定義済みのパッケージ］ショートカット：定義済みタイプの内部ライブラリをモデル ナビゲータの焦点とします。

［種類］カラム、［アイテムの種類］カラム、［ダイアグラムの種類］カラム

記述されたエンティティのタイプ。たとえばクラス、シンボル、クラス図などがあります。

［ビュー］カラム

現在の定義を示すシンボルとラインの数。

ダイアグラムの生成

Tau は、既存のモデル要素を可視化する目的でダイアグラムの自動生成をサポートします。一般的によく使用されるダイアグラム、たとえば継承図、合成図、依存関係図などを生成する、組込み済みのダイアグラムジェネレータが多数用意されています。特定のニーズのためにカスタムダイアグラムジェネレータを追加することもできます。

ダイアグラムを生成するには、以下の手順を実行します：

1. モデルビューで要素を選択します。選択した要素にしたがって、生成されるダイアグラムが決まってきます。たとえば、特定のクラスの上位クラス、下位クラスを可視化したい場合は、そのクラスを選択する必要があります。
1. コンテキストメニューから［ダイアグラムの生成］を選択して、サブメニューで使いたいダイアグラムジェネレータを選択します。たとえば、継承図を生成したい場合は、［Generate inheritance view］を選択します。

生成されたダイアグラムは通常は選択した要素の下に配置されます。ただし、一部のダイアグラムジェネレータはモデル内の別の場所に配置します。たとえば、最上位のダイアグラムとして配置したり、別パッケージ内に配置します。配置された後で、希望の場所に移動できます。

注記

あるエンティティを選択した際に、そのエンティティの下には配置できないダイアグラムを生成するダイアグラムジェネレータが表示されることがあります。たとえば、クラス図は操作の下には配置できませんが、操作を選択したときに、一部のダイアグラムジェネレータが表示されます。この場合、そのダイアグラムジェネレータを起動しようとすると［メッセージ］タブにエラー・メッセージが表示されます。

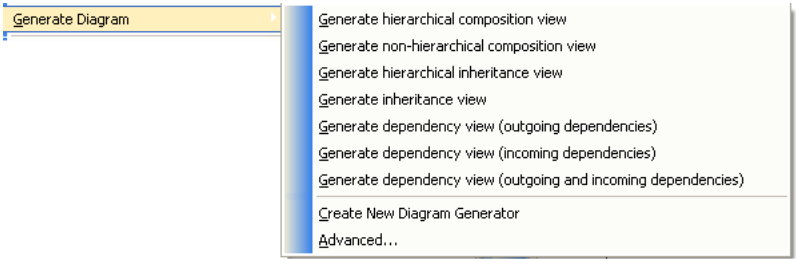


図 25: ダイアグラムの生成 コンテキストメニュー

ダイアグラム生成パラメータ

ダイアグラムジェネレータに実パラメータを与えてダイアグラムの生成を制御できます。たとえば、あるクラスの継承ビューを生成したい場合は、パラメータを使用して、直近の上位下位クラスのみを表示するのか、すべての上位下位クラスを表示するのかを制御できます。

ダイアグラムジェネレータをコンテキストメニューの［ダイアグラムの生成］から実行する場合、パラメータにはデフォルト値がセットされます。このパラメータを変更するには、生成されたダイアグラムで［ダイアグラム生成パラメータの編集］コンテキストメニューを使用します。生成されたダイアグラムからプロパティエディタを開いて、<<generated>> ステレオタイプを選択し、パラメータを編集することもできます。［Parameters］フィールドでテキストとして編集することも、［Edit Parameters］ボタンを押してダイアログから編集することもできます。

ダイアグラムの再生成

生成されたダイアグラムは、モデルの新しい情報に基づいて再生成できます。たとえば、新しい上位下位クラスが追加されたときに継承図を再生成したい場合などに有用です。ダイアグラムコンテキストメニューの［再生成］を使用して、再生成できます。ダイアグラム生成パラメータを修正した場合にもダイアグラムを再生成する必要が生じます。

生成したダイアグラムを再生成するには、ダイアグラムのコンテキストメニューから利用可能な［再生成］コマンドを使用します。［ツール］メニューから［すべての大グラムの再生成］を選択して、モデル内のすべてのダイアグラムを再生成することもできます。このコマンドで再生成されるのは自動生成されたダイアグラムだけです。

重要！

ダイアグラムの再生成を実行すると、ダイアグラムに含まれていたすべてのものが削除されて再生成されます。もし自動生成したダイアグラムに手動で変更を加えていた場合、たとえば、レイアウトや色の変更を手動で行っていた場合は、それらの変更はすべて失われます。

生成されたダイアグラムを通常のダイアグラムに変更する

修正されたダイアグラムに対して誤って再生成を実行することを回避するために、手動で管理してゆくダイアグラムについては、自動生成ダイアグラムから通常のダイアグラムへと変換することを推奨します。このためには、以下の手順を実行します。

1. モデルビューで生成したダイアグラムを選択します。
2. コンテキストメニューから［ステレオタイプ ...］を選択します。
3. ‘generated’ チェックボックスのチェックを解除して［OK］を押します。
4. この操作を行うとこのダイアグラムは再生成できなくなります。

既存のダイアグラムでダイアグラムジェネレータを使う

ダイアグラムジェネレータは必ずしも新規のダイアグラムを生成するわけではありません。既存のダイアグラムに情報を追加するためにダイアグラムジェネレータを使うこともできます。このためには、以下の手順を実行します。

1. 右マウスボタンを使用して、モデルビューから対象の要素をダイアグラムにドラッグします。
2. 要素をダイアグラムにドロップして、表示されるコンテキストメニューから［ダイアグラムの可視化］を選択します。
3. サブメニューで使用したいダイアグラムジェネレータを選択します。

要素をドロップした場所に、選択した ダイアグラムジェネレータで生成されたシンボルとラインが挿入されます。

高度なオプション

［ダイアグラムの生成］コンテキストメニューで表示されるダイアグラムジェネレータの他に、より高度なダイアグラムジェネレータも用意されています。これらのダイアグラムジェネレータを使うには、［ダイアグラムの生成］コンテキストメニューから［詳細 ...］コマンドを選択します。この操作で［ダイアグラム生成］ダイアログが開きます。

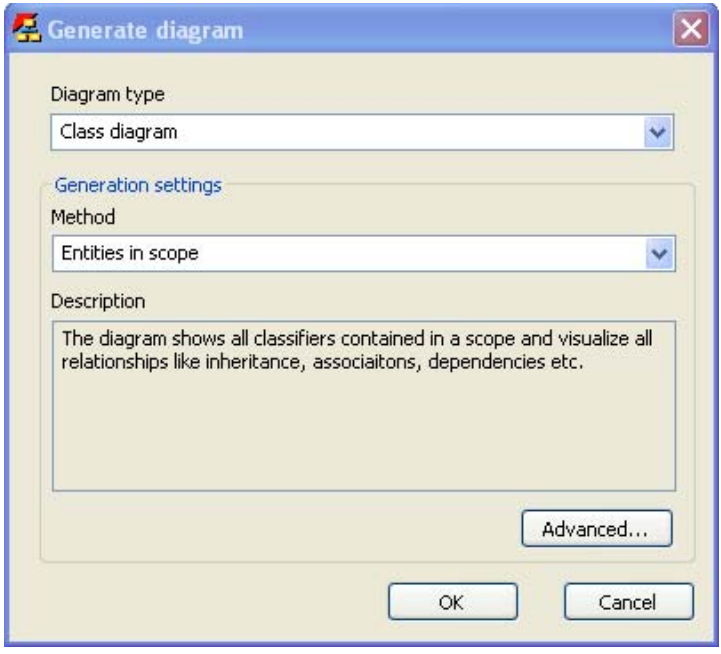


図 26: ダイアグラム生成ダイアログ

このダイアログでは、ダイアグラムジェネレータの種類は限定されますが、よりカスタマイズ可能なレイアウトオプションを指定できます。

ダイアグラムタイプ

［ダイアグラム生成］ダイアログでの最初の操作は、［ダイアグラム タイプ］を選択することです。ダイアグラムを生成するメソッドのあるダイアログタイプのみが表示されます。

生成の設定

次の操作は、生成方法である［メソッド］を選択することです。選択された［ダイアグラムタイプ］に適用できるメソッドのみが表示されます。

選択された生成メソッドの説明は、利用可能な生成方法のリストの下の［説明］欄に表示されます。

［詳細］ボタンを押すと、ダイアログが表示され、選択された生成メソッドについての詳しい設定ができます。これらの設定は生成されたダイアグラムに関連付けられ、生成後にプロパティエディタで編集できるようになります。

カスタマイズ

カスタムダイアグラムを生成する目的で、独自のダイアグラムジェネレータを作成できます。詳細については、ダイアグラムジェネレータの追加を参照してください。

プログラムからダイアグラムジェネレータを起動することもできます。これは、アドインを作成する場合などに有用です。詳細についてはダイアグラムジェネレータのプログラムからの起動を参照してください。

クエリ

このセクションでは、一定の条件を満たすエンティティを探すための UML モデルのクエリの実行方法について説明します。

クエリは、［検索］ダイアログの基本的な検索機能では探せないモデル内のエンティティを検索するときに便利な機能です。クエリは、必要な情報を探すための、標準 API の代替手法です。クエリでは、使用可能な多くの API 関数（COM、C++、Tcl）の呼び出しを行うので、ある 1 つのクエリの式の機能は該当する API セットを使用する場合と同等です。

概念

クエリは、モデルからのエンティティのコレクションを返す操作です。

「述語」は、ブール値の true または false を返す操作です。

クエリと述語はいずれも任意数の入力引数を取ることができます。また、常に暗黙的に存在する入力引数の 1 つとして、「モデル コンテキスト」があります。このモデル コンテキストというエンティティ上で、クエリまたは述語が呼び出されます。

UML モデル内で、クエリと述語操作を定義できるように、TTDQuery というライブラリが用意されています。これは 101 ページの図 27 のステレオタイプを定義します。104 ページの「［クエリ］ダイアログ」も参照してください。

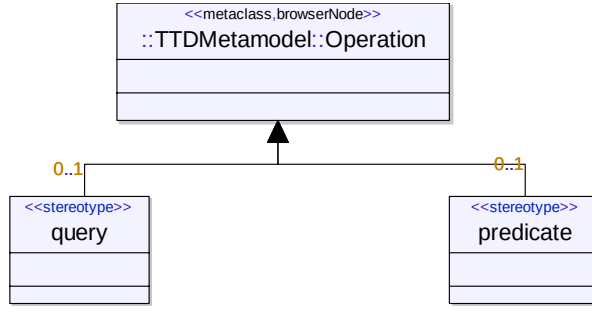


図 27: ステレオタイプを持つ TTDQuery ライブラリ

これらのステレオタイプに加えて、TTDQuery プロファイルには、すぐに使用できる多くのクエリと述語が用意されています。

クエリと述語の呼び出しは、クエリ式 にまとめられます。これは、Tau によって解釈可能な式で、クエリ操作の実行時と同じように、解釈の結果としてモデルからのエンティティのコレクションが返されます。クエリ式では、ブール演算子とリテラルのほか、OCL にあるコレクション演算子の小サブセットも使用して、クエリと述語の呼び出しによって取得した結果を修正できます。

注記

公開 API の多くの操作は、クエリまたは述語として動作します。これらの操作は、クエリ式にも使用できます。これらの API 操作の UML 定義は、u2 と呼ばれるライブラリにあります。

クエリ式

クエリ式は、UML 式のテキスト構文で表します。クエリ式の型は、エンティティのコレクションです。つまり、クエリ式が解釈されると、結果はエンティティのコレクションとなります。

クエリ式に含まれるすべてのサブ式は、ブール型またはエンティティのコレクション型でなければなりません。ブール型の式の場合は、通常のブール演算子を使用できます。クエリ式では、以下のブール演算子とリテラルがサポートされます。

```
and (&&)
or (||)
not (!)
true
false
```

かっこ内の表記を使用することもできます。

エンティティのコレクション型の式の場合は、定義済みコレクション演算子を使用できます。

コレクション演算子

クエリ式内の式の実行から得られたエンティティのコレクションで、いくつかの定義済み演算子を使用できます。これらの演算子の名前と意味は、OCL（オブジェクト制約言語）に基づいています。実際、定義済みコレクション演算子を呼び出す場合に矢印表記（->）ではなくピリオド（.）を使用すること以外は、クエリ式は正式な OCL 式です。ただし Tau でサポートされるのは OCL のサブセットのみです。このサブセットを使用して、強力なクエリを実行できます。

select

構文：

```
select(<boolean expr>)
```

型：エンティティのコレクション

`select` は、エンティティの 1 つのコレクションをエンティティの別のコレクションへと変換します。結果のコレクションには、入力コレクションのエンティティのうちブール式が true と評価するものが含まれます。つまり、`select` は、述語を通してコレクションにフィルタをかけるために使用できます。

exists

構文：

```
exists(<boolean expr>)
```

型：boolean

`Exists` は、ブール式です。入力コレクション内に、少なくとも 1 つのブール評価が `true` になるエンティティが存在する場合は `true` を返し、それ以外の場合は、`false` を返します。

isEmpty

構文：

```
isEmpty()
```

型：boolean

この演算子は、入力コレクションが空の場合 `true` を返します。それ以外の場合は、`false` を返します。

例

使用可能な内蔵クエリと述語 を定義済みブール演算子とコレクション演算子と組み合わせて使用するクエリ式の例を以下に示します。

例 **5**

パッケージで定義されたすべてのアクティブ クラスを検索します。

[モデル コンテキスト = パッケージ]

```
GetAllEntities().select(IsKindOf("Class") and
HasPropertyWithValue("isActive", "true"))
```

例 **6**

クラスによって直接所有されているモデル内のすべての属性を検索します。

[モデル コンテキスト = モデル、つまりセッション]

```
GetAllEntities().select(IsKindOf("Attribute") &&
GetOwner().exists(IsKindOf("Class")))
```

例 **7**

モデル内のすべての <<access>> 依存を検索します。

[モデル コンテキスト = モデル、つまりセッション]

```
GetAllEntities().select(not
GetTaggedValue("access(..)").isEmpty())
```

このクエリによって必要な結果が取得されますが、かなり非効率的です。これは、モデル内のすべてのエンティティ上で、適用された <<access>> ステレオタイプのチェックが行われるためです。エンティティが「依存」であるというチェックを追加するだけでパフォーマンスが著しく向上します。依存以外のエンティティについて、GetTaggedValue クエリを呼び出す必要がなくなるからです。

```
GetAllEntities().select(IsKindOf("Dependency") and not
GetTaggedValue("access(..)").isEmpty())
```

HasAppliedStereotype 述語を使用して、式を書き直せます。これは、ステレオタイプが要素上で適用されているかどうかをチェックするときに推奨される方法です。

```
GetAllEntities().select(IsKindOf("Dependency") and
HasAppliedStereotype("access"))
```

<<access>> 依存を効率よく最短で検索するためには、クエリ式に、次のように GetStereotypedEntities クエリを使用します。

[モデル コンテキスト = TTDPredefinedStereotypes ライブラリ内の <<access>> ステレオタイプ]

```
GetStereotypedEntities()
```

この例のように、同じ結果を取得するために使用できるいくつかのクエリ式が存在する場合があります。同じ意味を持つクエリ間で実行パフォーマンスが大きく異なる可能性があるので、クエリ式を書く前に、それぞれの選択肢を検討することが重要です。

［クエリ］ダイアログ

［クエリ］ダイアログを使用して、実行するクエリ式を組み立てられます。このダイアログを開くには、［モデル ビュー］またはダイアグラムでエンティティを選択して、メニュー項目［編集］-> ［クエリ ］を選択します。選択したエンティティがクエリ式のモデル コンテキストになります。

注記

クエリ式のモデル コンテキストとして、プレゼンテーション要素（ダイアグラム内のシンボルや行など）を使用できます。ダイアグラム内の選択されたエンティティから［クエリ］ダイアログを開くと、選択したプレゼンテーション要素がモデル コンテキストになります。モデル要素のクエリを実行する場合は、ショートカット メニュー［モデル ビューで表示］を使用して、対応する要素を［モデル ビュー］で検索します。

［クエリ］ダイアログには、現在のモデルで検索されたすべての使用可能なクエリと述語がリストされます。このリストには、定義 済み TTDQuery と u2 ライブラリで提供されるすべての「内蔵」クエリと述語が含まれます。また、他の場所で定義されているすべてのクエリと述語（ユーザー定義のクエリと述語など）も含まれます。

クエリ式を実行するには、［実行］ボタンを押します。デフォルトでは、結果が［検索結果］タブに出力されます。出力先は、ドロップ ダウン コントロールで別のタブ名を指定して変更できます。

編集コントロールで直接式を書くか、または使用可能なクエリと述語のリストでエントリをダブルクリックして、クエリ式を組み立てられます。選択した操作（クエリまたは述語）に入力仮パラメータがない場合、操作の呼び出しが直接クエリ式のテキス

ト内のカーソルの位置に追加されます。ただし、操作に 1 つ以上の入力仮パラメータがある場合は、ポップアップ ダイアログ（105 ページの図 28 を参照）が表示されます。このダイアログで、操作呼び出し用の対応する実パラメータを指定できます。

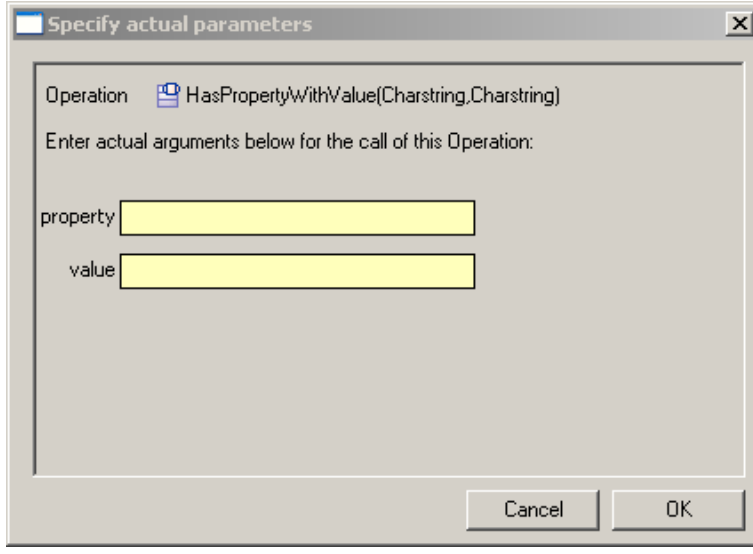


図 28 実パラメータの指定

このダイアログは、実際はプロパティ エディタ（パラメータは操作のプロパティと見なされる）であり、編集された値はプロパティ エディタと同じカラー コードに従います。パラメータの意味について「これは何？」ヘルプを表示するなど、プロパティ エディタの他の機能も利用できます。

クエリ式を新規クエリとして保存

［クエリ］ダイアログの［保存］ボタンを使用して、クエリ式をモデルの新規クエリとして保存できます。作成したクエリ式を今後も使用するために保存したい場合、この機能を使用します。新しいクエリの名前と説明、およびそのクエリを保存するモデル内の場所を指定するように指示されます。すべてのクエリを、別の .u2 ファイルに保存されるプロファイル パッケージなど、共通の場所に格納するとよいでしょう。これで、保存したクエリを複数のプロジェクトに組み込んで使用できます。

クエリ式を新規クエリとして保存すると、新しいクエリ式で使用可能なクエリと述語のリストに入り、ただちに利用できるようになります。

内蔵クエリと述語

クエリ式には、あらかじめツールに内蔵されたさまざまなクエリと述語を使用できます。これは、プロファイル ライブラリ TTDQuery と u2 で定義されて文書化されています。

また、ユーザー定義クエリと述語で説明したように、ユーザー定義クエリと述語を追加することもできます。

ユーザー定義クエリと述語

あらかじめツールに内蔵されているクエリと述語のほかに、新たなクエリと述語を定義できます。これを行うには、<<query>> または <<predicate>> ステレオタイプが適用されているエージェントを定義します。そのようなエージェントの実装では、クエリまたは述語のシグニチャの要件を満たす必要があります。したがって、クエリ エージェントは一連のエンティティを返し、述語 エージェントはブール値を返す必要があります。この必須出力パラメータは、最初のパラメータとしてエージェントに渡されます。また、エージェントは任意の数の入力パラメータを取ることができます。これらのパラメータには、エージェントによってサポートされている任意の型を使用できます。

API からのクエリ式の実行

106 ページの図 29 のエージェントを使用して、公開 API からクエリ式をプログラムにとして実行できます。

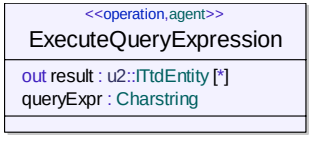


図 29 クエリ式のエージェント

このエージェントは（他のエージェントと同じように）、InvokeAgent 操作を使用して起動します。

例 **8: Tcl API** からのクエリ式の実行 **----------**

以下の例は、Tcl スクリプトからクエリ式 “GetAllEntities()” を実行する方法を示しています。スクリプトによって結果のエンティティの Tcl ID が単に出力されます。

```
set s [std::GetSelection]
set a [u2::FindByGuid $U2 "@TTDQuery@ExecuteQueryExpression"]
set p [lappend p {} "GetAllEntities()"]
u2::InvokeAgent $U2 $a $s p
output [lindex $p 0]
```

ドラッグ アンド ドロップ

ここでは、ドラッグ アンド ドロップによるモデルの操作方法について説明します。

ドラッグ アンド ドロップ操作は、ドラッグ ソースとドロップ ターゲットとして以下の 3 つの組み合わせがあります。

- モデル ビュー内
- モデル ビューからダイアグラムへ
- ダイアグラム内とダイアグラム間

ドラッグ アンド ドロップ操作は、マウスの左ボタンまたは右ボタンを使用して実行できます。マウスの右ボタンを使用してドラッグ アンド ドロップ操作を行う場合はショートカット メニューが開き、ソース要素からターゲット要素へドラッグした結果として、実行可能な操作が表示されます。ショートカット メニューには必ず強調表示で示される選択肢があります。これは、マウスの左ボタンを使用してドラッグ アンド ドロップを行った場合に実行される操作を示しています。操作の隣にかっこ付きでモディファイア キーが表示されている場合もあります。この操作は、モディファイア キーを押した状態で、マウスの左ボタンを使用してドラッグ アンド ドロップを行うことで、実行が可能です。

次のセクションでは、ドラッグ アンド ドロップによって実行可能な操作について説明します。

モデル ビュー内

移動

モデル ビュー内の要素を移動します。

これは、モデル ビュー内でのドラッグ アンド ドロップのデフォルトの操作です。マウスの左ボタンを使用してドラッグ アンド ドロップを行った場合に、実行されます。

コピー

モデル ビュー内の要素をコピーします。

この操作は、Ctrl キーを押しながらマウスの左ボタンを使用してドラッグ アンド ドロップ操作を行った場合に、実行されます。

リンク

ドラッグ ソース要素とドロップ ターゲット要素との間にリンクを作成します。現在アクティブなリンク タイプが使用されます。

参照

リンクを使った作業

トレービリティを含むコピー

モデルビュー内の要素（サブ要素を含む）をコピーして、コピーからオリジナルへの <<trace>> 依存を作成します。

この動作を行わせるには、右マウスボタンを使ってドラッグアンドドロップし、[トレーサビリティを含むコピー] コマンドをポップアップメニューから選択します。

依存は、パッケージ、クラス、属性、操作などすべての定義について作成されます。

モデル ビューからダイアグラムへ

プレゼンテーションの作成

ドラッグ ターゲット要素との関連で、ドラッグ ソース要素を表すシンボルを作成します。

これは、モデル ビューからダイアグラムへのドラッグ アンド ドロップのデフォルトの操作です。マウスの左ボタンを使用してドラッグ アンド ドロップを行った場合に、実行されます。

プレゼンテーションの作成 (ラインを含む)

[プレゼンテーションの作成] と同じ操作ですが、ドロップ ターゲット ダイアグラムの他の要素とドラッグ ソース要素との接続を表すラインが作成されます。

ダイアグラムの可視化

ドラッグ ソース要素とドロップ ターゲット要素のために用意されているダイアグラム生成メソッドを含むサブメニューです。ドラッグソース要素は、ダイアグラム内の既存の要素に影響を与えずに、ダイアグラム内で可視化されます。

参照

ダイアグラムの生成

ダイアグラム内とダイアグラム間

ダイアグラム内とダイアグラム間でのドラッグ アンド ドロップによって実行される操作は、モデル ビュー内の場合と同じです。

バージョンの比較とマージ

同じモデルから派生したバージョンを比較できます。この比較はモデル ファイル（.u2 拡張子）間またはプロジェクト ファイル（.ttp 拡張子）間で行います。自動あるいは手動で相違点を受け入れることができます。同じモデルから派生した 2 つ以上のバージョンの比較とマージの場合は、確実にバージョン管理を実行するために、構成管理システムを使用することを推奨します。

注記

モデル要素には、必ず GUID があります。比較とマージの機能を使用するには、まったく同じモデルから派生したバージョンでなければなりません。同じ名前で並行作成された 2 つのモデルでも、同じように見えるモデル要素にも異なる GUID があります。

マージのバリエーション

比較やマージの操作には 3 通りの基本バリエーションがあります。これらのバリエーションは、2 種類、3 種類、4 種類の比較（マージ）と呼ばれます。同じモデルから派生した 2 つのバージョンを比較またはマージする場合、2 種類で実行します。バージョン管理下の異なる構成ブランチで開発された同じモデルから派生したバージョンで作業する際、3 種類と 4 種類のマージを実行します。

- 2 種類：同じモデルから派生した 2 つのバージョンを比較、または 1 つにマージします。
- 3 種類：（最も近い）共通の世代を考慮しながら、同じモデルから派生した 2 つのバージョンを比較または 1 つにマージします。
- 4 種類：同じモデルから派生した 2 つのバージョンを比較、またはマージします。前回のマージを考慮に入れる場合、4 種類の比較（マージ）を実行します。この方法では、前回のマージからのマージ分岐が結果のモデルに自動的に反映されます。

構成

109 ページの図 30 の構成を考えます。メインの構成ブランチに、最初にバージョン A1 があります。

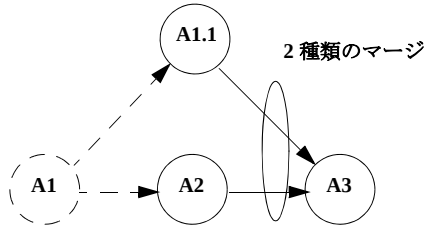


図 30: 2 種類のマージのバージョン ツリー

バージョン A1 が変更されます。新しいバージョン（バージョン管理下）はバージョン A2 となります。

バージョン A1 はバージョン A1.1 に分岐し、後にバージョン A2 にマージされてバージョン A3 となります。このマージは、最も近い共通の世代である A1 との関係をまったく考慮せずに実行するか（**2 種類のマージ、**109 ページの図 30）、A1 からのバージョン情報をもとに実行できます（**3 種類のマージ、**110 ページの図 31）。

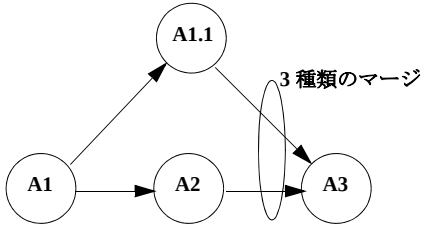


図 31: 3 種類のマージのバージョン ツリー

この例では、2 種類のマージと 3 種類のマージのどちらを実行しても結果はバージョン A3 となります。構成管理の観点からは、3 種類のマージの方法をとり、常に共通の世代を考慮に入れることを推奨します。

3 種類のマージより、2 種類のマージのほうがよい場合があります。これにはいくつかの理由があります。以下に例を示します。

- 2 種類のマージの方が簡単に実行できる場合。
- 共通の世代のバージョンに対するアクセス権がないか、共通の世代の名前を手動で指定する必要があり、誤って指定すると、エラーのリスクが高い場合。
- すべての相違点を手動で確認する必要があり、相違点の解決に自動サポートが不要な場合。
- すべての修正が比較機能でチェックされ、2 種類のマージが単なる更新作業の場合。開発が分岐されていない、または分岐された一方が変更されていない場合がこれに当てはまります。

バージョン A1.1 と A3 で作業を継続し、その結果、バージョンがそれぞれ A1.2 と A4 となります。次に、これらのバージョンは、**4 種類のマージで**バージョン A5 にマージできます。111 ページの図 32 を参照してください。4 種類のマージでは、前回のマージ（結果 A3 となった）を考慮し、このマージでハンドリングされたプロパティは表示されません。このため、A1.1 から A1.2 への変更、A3 から A4 への変更のマージに焦点を当てることができます。

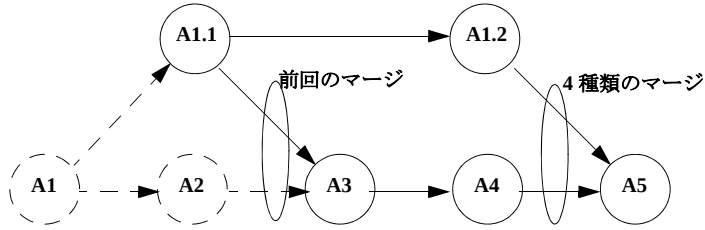


図 32: 4 種類のマージのバージョン ツリー

名前付け規則

上記の例では、名前付け規則をもとに、以下に示すそれぞれの異なる状況でマージされたバージョンに名前を付けます。2 種類のマージと 3 種類のマージで同じ結果になります。どちらの方法をとるかは、状況によります。

- 2 種類のマージ：現在のバージョンは**バージョン 1** です。109 ページの図 30 では A2 です。A1.1 は**バージョン 2** となります。マージが完了すると、結果は A3 となります。
- 3 種類：メイン構成ブランチへのマージを考慮した場合、A2 は**バージョン 1**、A1.1 は**バージョン 2**、A1 は**共通の世代**となります。マージが完了すると、結果は A3 となります。
- 4 種類：メインの構成ブランチへのマージを考慮した場合、A4 は**バージョン 1**、A3 は**バージョン 1 の前の世代**となります。A1.2 は、**バージョン 2**、A1.1 は**バージョン 2 の前の世代**となります。マージが完了すると、結果は A5 となります。

注記

共通の世代という用語を使用する場合、上記の例のようにどのファイルのペアにとっても共通の世代である A1 の前に幾つかバージョンが存在する際に、常に**最も近い**共通の世代を意味します。

プロジェクトのマージ

複数のファイルを含むプロジェクト間のマージは、version 2 と ancestor 1 と ancestor 2 に、モデル ファイル（.u2）ではなくプロジェクト ファイル（*.ttp）を指定することで、実行できます。モデルには通常かなりの階層がりますが、プロジェクトのマージは正しい分岐を行うための自動マージの手助けとなります。プロジェクトのマージは、マージ中にすべてのモデル要素がロードされるため、個々のモデル要素のマージより有用です。また、モデル ファイルごとにマージを繰り返さなくても、すべてのモデル ファイル (.u2) を一度にマージできます。

比較／マージの考慮点

比較／マージに関して考慮に入れるべき問題がいくつかあります。

- **世代**は実際の世代である必要があります（できるだけバージョン 1 と同じファイルは使わないようにします）。
- **プロジェクト全体**をツールにロードする必要があります（複数ファイルのうちの 1 つだけではありません。1 つのファイルの変更が他のファイルにも影響を与える可能性があるからです。また、この変更は他のファイルに反映されなければなりません。反映されないと、一貫性が失われます）。
- すべてのファイルをマージの実行前に**保存**することを推奨します。こうしておくことで、マージ操作をキャンセルした場合に、マージ操作開始前の状態に復元できます。

バージョンの比較

［ツール］メニューの［バージョンの比較］を選択して、バージョン比較機能のダイアログ ウィンドウを表示します。このウィンドウには、比較操作のオプションがあります。

バージョン **2** ― ファイルから読み取り済み

このフィールドには、比較対象となる（同じモデルから派生した）バージョンを含むファイルの名前（.u2 または .ttp）が表示されます。2 種類の比較では、これはファイルの前バージョンです。3 種類の比較では、これは構成管理対象ブランチに属するバージョンです。4 種類の比較では、これは前回のマージを考慮に入れる構成管理対象ブランチに属するバージョンです。

共通の世代（**3** 種類）またはバージョン **1** の前の世代（**4** 種類の比較）

3 種類の比較では、このフィールドを使って（最も近い）**共通の世代**を含むファイルの名前（.u2 または .ttp）を選択します。このバージョンは、バージョン 2 と現在ロードされているバージョンの共通の派生元でなければなりません。

4 種類の比較では、このフィールドを使って、前回のマージから現在ロードされているバージョンに対する、直前の世代を含むファイルの名前（.u2 または .ttp）を選択します。これは、**バージョン 2 の前の世代**とその直前のマージの結果、作成されたバージョンでなければなりません。

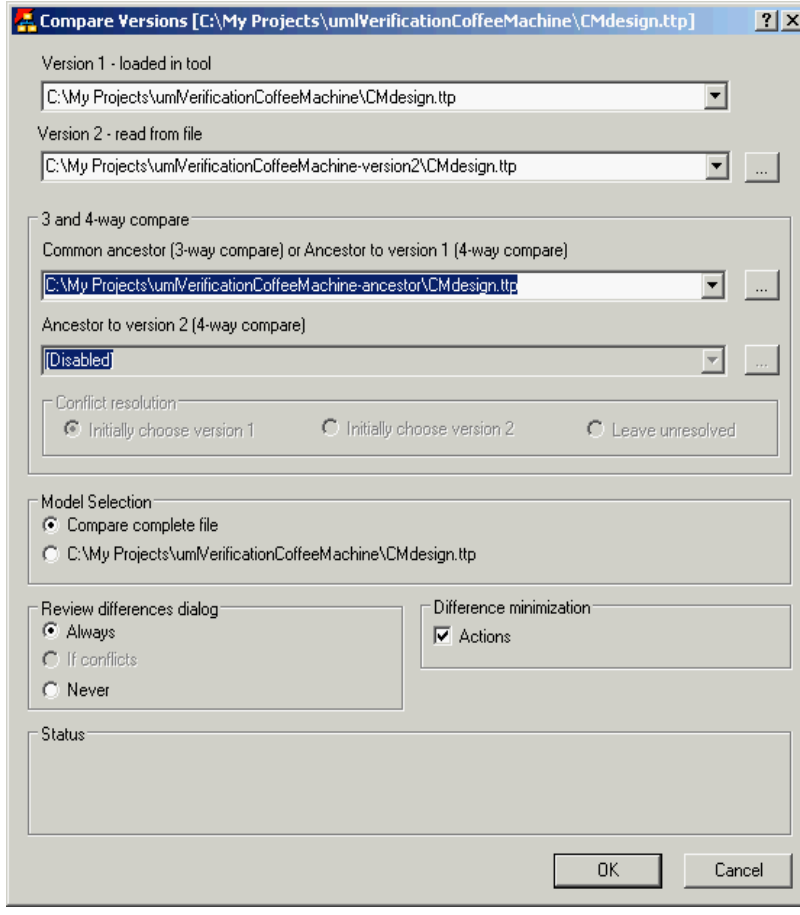


図 33: バージョン比較ダイアログ

バージョン **2** の前の世代（**4** 種類）

4 種類の比較は前回のマージを参照しながら実行されます。4 種類の比較では、フィールド［バージョン 2 の前の世代］を使って、前回のマージから［バージョン 2 ― ファイルから読み取り済み］フィールドで選択されたバージョンに対する、直前の世代を含むファイルの名前（.u2 または .ttp）を選択します。通常、これは前回のマージで使用されたバージョンの 1 つで、**バージョン 1 の前の世代**です。

モデルの選択

- ファイル全体を比較（デフォルト）：モデル全体に対して比較操作を行います。ほとんどの場合このケースが適用されます。
- ファイルの選択された部分だけを比較：ダイアグラムまたはワークスペース ウィンドウの選択されたモデルに対してのみ比較操作が適用されます。

相違点ダイアログでのレビュー

- 常に：このオプションを選択すると、［相違点のレビュー］ダイアログに必ず操作結果が表示されます。
- 競合した場合：このオプションを選択すると、変更または競合する定義があった場合のみ、［相違点のレビュー］ダイアログに操作結果が表示されます。たとえば、相違点が追加されたモデル要素のみで構成されていると、ダイアログは表示されません。結果ダイアログには操作の概要のみ表示されます。
- しない：このオプションを選択すると、結果ダイアログには操作の概要のみ表示されます。

相違点の最小化

比較とマージ機能を実行する場合、大部分のケースにおいて［アクション］オプションを選択することを推奨します。［アクション］オプションを設定した場合、比較操作は、中間モデルのアンバインドから生じる GUID の相違を無視します。この相違は、フローのパートを切り離して新しいシンボルを挿入する場合などに発生します。切り離されたフローはしばらくの間アンバインド状態になり、再結合されたシンボルは新しい GUID の値を受け取るためです。

バージョンのマージ

［ツール］メニューから［バージョンのマージ］を選択して、この機能のダイアログ ウィンドウを呼び出すことができます。

バージョン **2** ― ファイルから読み取り済み

このフィールドには、マージ対象となる（同じモデルから派生した）バージョンを含むファイルの名前（.u2 または .ttp）が表示されます。2 種類のマージでは、これはファイルの前バージョンです。3 種類のマージでは、これは構成管理対象ブランチに属するバージョンです。4 種類のマージでは、これは前回のマージを考慮に入れる構成管理対象ブランチに属するバージョンです。

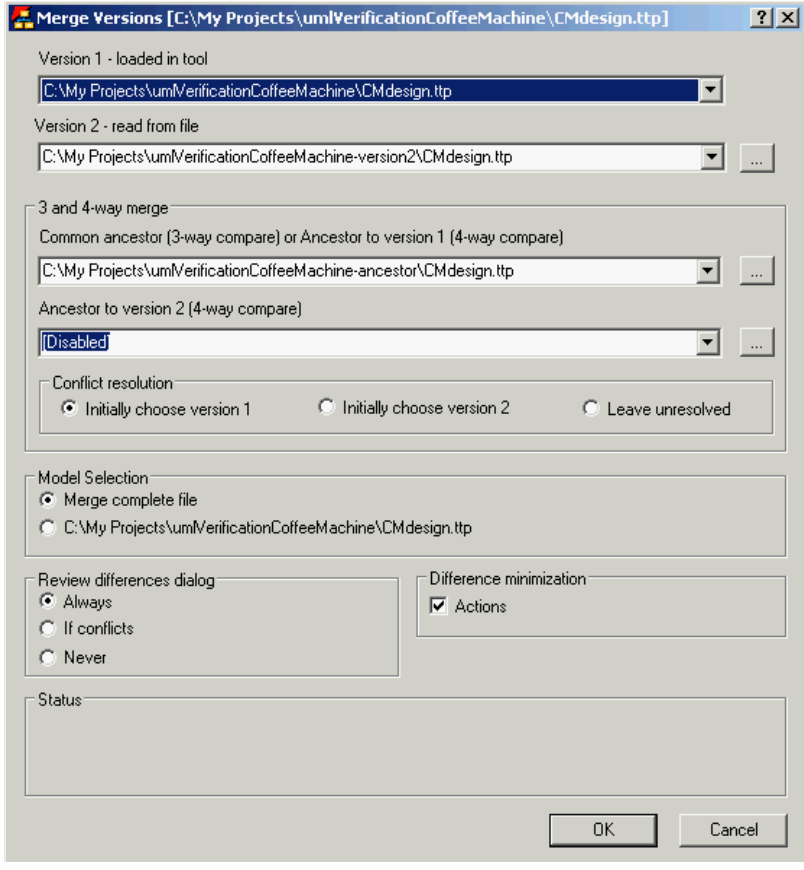


図 34: バージョンマージダイアログ

共通の世代（3 種類）またはバージョン 1 の前の世代（4 種類の比較）

3 種類のマージでは、このフィールドを使って（最も近い）**共通の世代**を含むファイルの名前（.u2 または .ttp）を選択します。このバージョンは、バージョン 2 と現在ロードされているバージョンの共通の派生元でなければなりません。

4 種類のマージでは、このフィールドを使って、現在ロードされているバージョンの直前の世代を含むファイルの名前（.u2 または .ttp）を選択します。これは、**バージョン 2 の前の世代**との直前のマージの結果、作成されたバージョンでなければなりません。

バージョン 2 の前の世代（4 種類）

4 種類のマージは前回のマージを参照しながら実行されます。4 種類のマージでは、フィールド［バージョン 2 の前の世代］を使って、前回のマージから［バージョン 2 ― ファイルから読み取り済み］フィールドで選択されたバージョンに対する、直前の世代を含むファイルの名前（.u2 または .ttp）を選択します。通常、これは前回のマージで使用されたバージョンの 1 つで、**バージョン 1 の前の世代**です。

注記

4 種類のマージ：**バージョン 2 の前の世代**で示されるバージョンに戻すには、プロパティで［V2 を承諾］、次に［V2 を拒否］を実行します。これでプロパティを**バージョン 2 の前の世代**の状態に戻すことができます。

モデルの選択

- ファイル全体をマージ比較 (デフォルト) : モデル全体に対してマージ操作を行います。ほとんどの場合このケースが適用されます。
- ファイルの選択された部分だけをマージ：ダイアグラムまたはワークスペースウィンドウの選択されたモデルに対してのみマージ操作が適用されます。

相違点ダイアログでのレビュー

これらのオプションについては、112 ページの「バージョンの比較」を参照してください。

競合の解決

このオプションは、3 種類または 4 種類のマージを実行する際に有効です。マージ操作によって自動的に競合がマージされなかった場合にどうするのかを決定します。

- 最初はバージョン 1 を選択：このオプションを選択すると、マージ操作で、現在ロードされているバージョン 1 のプロパティが使用され、マージ後のバージョンが作成されます。
- 最初はバージョン 2 を選択：このオプションを選択すると、マージ操作で、［バージョン 2 ― ファイルから読み取り済み］フィールドの引数であるバージョン 2 のプロパティが使用され、マージ後のバージョンが作成されます。
- 未解決のままにする：このオプションを選択すると、マージ操作は競合を未解決のままにします。どちらのバージョンを選ぶかを指定してすべての競合を明示的に解決するまで、マージ操作は完了しません。マージ操作は、一時的なバージョンを作成するために、バージョン 1 の前の世代（このバージョンは共通の世代（3 種類）またはバージョン 1 の前の世代（4 種類の比較）の指定値です）のプロパティを使用します。

望ましくない相違があった場合は、マージの実行後に拒否されます。

注記

このオプションは非競合の相違には影響を与えません。つまり、バージョン 1 とバージョン 2 の非競合の相違はマージ結果のバージョンに必ず含まれます。

コマンド ラインの使用

［比較］または［マージ］ダイアログをコマンドラインから開くことができます。このためには、Tau インストールディレクトリ内のディレクトリにある Tcl スクリプトの `u2compare.tcl` と `u2merge.tcl` を使用します。`u2compare.tcl` と `u2merge.tcl` の 2 つのスクリプトは、以下の 2 つの操作モードをサポートしています。

- 単一ファイルモード：モデル ファイル（.u2）での操作に使用
- プロジェクト モード：プロジェクト ファイル（.ttp）での操作に使用

注記

Tau の呼び出し（`Windows` では VCS.EXE、UNIX では tau）では、tcl ファイルはフルパスで指定する必要があります。たとえば、Windows での呼び出しは以下のようになります。

```
VCS.EXE -script "C:¥Program
Files¥IBM¥Rational¥TAU¥4.3¥etc¥u2compare.tcl" ...
```

コマンドは以下のとおりです。

単一ファイル モード

```
<Tau> -script {u2compare.tcl | u2merge.tcl}
        {forceVersion1 | forceVersion2 | leaveUnresolved}
        {reviewDifferencesNever | reviewDifferencesAlways |
reviewDifferencesIfConflicts}
        {suppressSetupNever | suppressSetup}
        {true | false}
        [<version1>.ttp | <version1>.ttw]
        <version1>.u2 <version2>.u2
        [<ancestor1>.u2 [<ancestor2>.u2]]
```

プロジェクトモード

```
<Tau> -script {u2compare.tcl | u2merge.tcl}
        {forceVersion1 | forceVersion2| leaveUnresolved}
        {reviewDifferencesNever | reviewDifferencesAlways |
reviewDifferencesIfConflicts}
        {suppressSetupNever | suppressSetup}
        {true | false}
        {<version1>.ttp | <version1>.ttw}
        <version1>.u2 <version2>.u2
        [<ancestor1>.u2 [<ancestor2>.u2]]
```

下表でコマンドに使用される属性を説明します。

ForceVersion	**説明**
`forceVersion1`	競合時はまずバージョン `1` を選択します。
`forceVersion2`	競合時はまずバージョン `2` を選択します。
`leaveUnresolved`	競合を解決しません。

ReviewDifferences	説明
reviewDifferencesNever	［相違点のレビュー］ダイアログを表示しない。
reviewDifferencesAlways	［相違点のレビュー］ダイアログを常に表示する。
reviewDifferencesIfConflicts	競合時のみ［相違点のレビュー］ダイアログを表示する。

SuppressSetup	説明
suppressSetupNever	比較 / マージの設定ダイアログを表示する。
suppressSetup	比較 / マージの設定ダイアログを表示しない。

ExitOnSuccess	説明
true	操作完了後に Tau を終了する。
false	捜査後に Tau を終了しない。

例 **9:** 前の世代の単一ファイルとの単一ファイルマージ、**Windows** ————————

```
VCS.EXE -script "C:¥Program
Files¥IBM¥Rational¥TAU¥4.3¥etc¥u2merge.tcl" forceVersion1
reviewDifferencesAlways suppressSetupNever false
C:/work/version1/project.ttp C:/work/version1/file.u2
C:/work/version2/file.u2 C:/work/ancestor/file.u2
```

————————————————————————————————————

例 **10: 2** つの前の世代と相対パスとのプロジェクト マージ、**Windows** ————————

```
VCS.EXE -script "C:¥Program
Files¥IBM¥Rational¥TAU¥4.3¥etc¥u2merge.tcl" forceVersion1
reviewDifferencesAlways suppressSetupNever false
version1/project.ttp version1/project.ttp version2/project.ttp
ancestor1/project.ttp ancestor2/project.ttp
```

————————————————————————————————————

例 **11:** 相対パスとの単一ファイルの比較、**UNIX** ————————————————

```
tau -script ${TAU_HOME}/etc/u2compare.tcl forceVersion1
reviewDifferencesAlways suppressSetupNever false
version1/file.u2 version2/file.u2
```

————————————————————————————————————

相違点ダイアログでのレビュー

比較／マージ操作からの出力

操作の結果は出力ウィンドウ、または、［相違点のレビュー］ダイアログに表示されます。結果が比較情報の概要のみである場合、操作結果の概要のみが表示され、出力ウィンドウにメッセージが表示されます。［競合の解決］オプションでの設定とバージョン間の相違により、［相違点のレビュー］ダイアログに表示される内容が異なります。

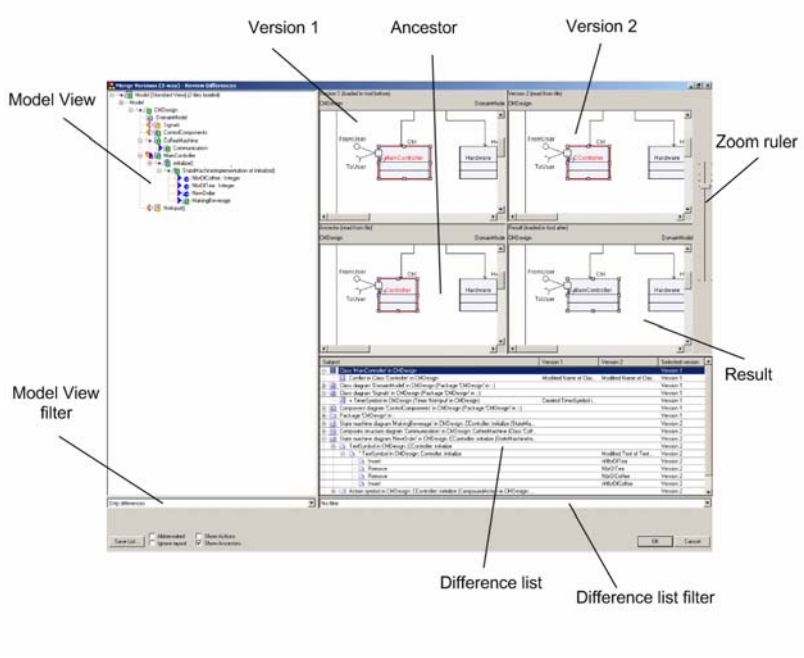


図 35: 相違点レビューダイアログ

モデルビュー

［相違点のレビュー］ダイアログの左側のモデルビューでは、モデルのすべての相違点が表示されます。Special icons provide additional information such as version where changes have been done, found conflicts and a current merge state for those changes. The meanings of the icons are listed below.

Icon	Icon Meaning
	Model element has been added in Version 1
	Model Element has been added in Version 2
	Model element has been deleted in Version 1
	Model element has been deleted in Version 2
	Model element has been modified in Version 1
	Model element has been modified in Version 2
	Model element has been modified in both version, no conflict.
	Conflict, model element has been deleted in Version 1 and modified in Version 2
	Conflict, model element has been modified in Version 1 and deleted in Version 2
	Conflict, model element has modified in both version
	Model element does not contain direct differences, but has differences in children

Icon	Icon Meaning
	Unresolved conflict
	Version 1 is accepted
	Version 2 is accepted
	Modifications in Version 1 and/or Version 2 are rejected
	Modifications in both versions are accepted

モデルビューフィルタ

モデルビューにフィルタが適用できます。以下のフィルタが使用できます：

- **フィルタなし：**すべてが表示されます。
- **相違点のみ：**相違点のあるエンティティが表示されます。
- **競合のみ：**競合のあるエンティティが表示されます。
- **未解決のみ：**未解決の競合のあるエンティティが表示されます。

バージョン **1**

このウィンドウには、現在選択されている［サブジェクト］アイテム、または、全相違点のプレゼンテーション要素が、比較（マージ）の実行前にツールにロードされたバージョンのプロパティとともに、表示されます。

結果

このウィンドウには、比較（マージ）の実行後にツールにロードされたバージョンのプロパティが表示されます。

バージョン **2**

このウィンドウには、現在選択されている［サブジェクト］アイテム、または、全相違点のプレゼンテーション要素が、［バージョン 2 － ファイルから読み取り済み］フィールドの引数であるバージョンのプロパティとともに、表示されます。

ズーム ルーラ

グラフィック ウィンドウの右側にあるズーム ルーラで、3 つのプレゼンテーション要素を同時に拡大表示できます。

相違点リスト

相違点リストはモデルビューで選択されたエンティティとその子についてすべての相違点を表示します。

モデルビューで可能な操作は、相違点リストでの相違点の選択の際にコンテキストメニューから利用可能です。See section Difference Grouping for a description of the nodes visible in the difference list.

相違点リストのカラム

- **サブジェクト**：このカラムには、2 つの比較バージョンの全相違点があったアイテム、または、アイテム グループのアイコンと識別子が表示されます。
- **バージョン 1**：このカラムには、バージョン 1 に属する全相違点（グループの相違点の数）、または、アイテム（アイテム固有のプロパティ）のバージョン情報が含まれます。
- **バージョン 2**：このカラムには、バージョン 2 に属する全相違点（グループの相違点の数）、または、アイテム（アイテム固有のプロパティ）のバージョン情報が含まれます。
- **選択されたバージョン**（マージの場合のみ）：このカラムには、マージ結果に、現在、含まれている相違点のバージョンが表示されます。

相違点リストフィルタ

相違点リストにフィルタが適用できます。以下のフィルタが使用できます：

- **フィルタなし :** すべての相違点が表示されます。
- **競合のみ :** 競合のみが表示されます。
- **未解決のみ :** 未解決の競合のみが表示されます。

コンテキストメニュー

［モデルビュー］でエンティティを右クリックすると、または相違点リストで項目を右クリックすると、コンテキストメニューが開き、以下の操作を実行できます （122 ページの図 36）。

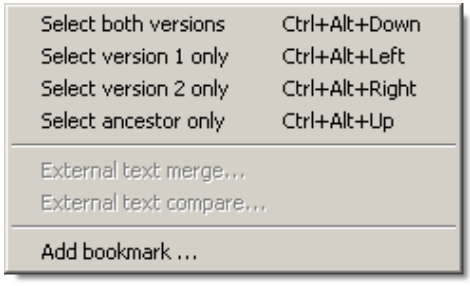


図 36: コンテキストメニュー

Select both versions

Differences from both versions are merged into the model. If a conflict can be consolidated the merged result will be entered into the model. If the conflict cannot be consolidated then it will be resolved according to the option “競合の解決” and a corresponding difference(s) will be merged into the model. This operation is done hierarchically on the selected entity and all its children. If the changes have been done in one version only or the changes from both versions have been already accepted, then this menu item is disabled.

Select version 1 only

Only differences from version 1 are merged into the model. If there are differences from version 2 currently in the model the ancestor version will be entered instead. This operation is done hierarchically on the selected entity and all its children. If there is no changes in the version 1 or the changes have been already accepted, then this menu item is disabled.

Select version 2 only

Only differences from version 2 are merged into the model. If there are differences from version 1 currently in the model the ancestor version will be entered instead. This operation is done hierarchically on the selected entity and all its children. If there is no changes in the version 2 or the changes have been already accepted, then this menu item is disabled.

Select ancestor only

Differences from version 1 and version 2 will be rejected from the model. This operation is done hierarchically on the selected entity and all its children. If the changes have been already rejected, then this menu item is disabled.

External text compare/External text merge

An external textual compare and merge tools can be used for comparing and/or merging comments, text symbols, task symbols and instance expressions. The “External text compare...” and “External text merge...” operations are available where applicable.

If an external textual merge is done, the result will be checked if it can be reentered into the model. If it cannot be entered into the model, the result file from the external tool is saved and the path is reported together with an error message box.

Path and command line switches for the external text compare/merge tool are available via the Tools menu, Options dialog, under the ［比較 / マージ］タブ tab.

Add bookmark

A bookmark can be added on a selected entity. A comment can be added to the bookmark. The bookmarks can later be listed in the model navigator.

一覧を保存

相違点リストを XML 形式の外部ファイルに保存します。このファイルの内容をより読みやすい形式で表示するには、XML ファイルとともに XSL スタイルシートファイルが必要になります。Tau ではインストール時にサンプルのスタイルシートが提供されています。導入ディレクトリの `etc` ディレクトリにある u2compare.xsl (差分中心のスタイル) と u2compare_diagram.xsl (ダイアグラム中心のスタイル) という名前のファイルです。

XML ファイルを表示するには、XSL ファイルを XML ファイルと同じディレクトリにコピーして、拡張子は `xsl` のままでファイル名だけを XML ファイルと同じに変更します。この操作で、XML ファイルを WEB ブラウザで表示できるようになります。

生成されるイメージのサイズ、および未解析のテキストを保存するかどうかは、レビュー情報の保存オプションで指定できます。指定箇所は、［ツール］メニューから、［オプション］ダイアログを開き、［比較 / マージ］タブです。

省略形

［省略形］チェックボックスは、相違点を説明するテキストの概要を表示するために使用できます。

レイアウトを無視

［レイアウトを無視］チェックボックスを使用して、レイアウトのみに関する相違点（位置、サイズ、または線分の接点などの変更）を一時的に非表示にできます。レイアウトに関する相違点はリストから削除されません。非表示にされるだけであり情報は維持されます。

アクションの表示

このオプションを設定すると、テキストの相違だけではなくモデルの相違も表示されます。これは、アクション シンボル内のステートメント、テキスト シンボル、およびテキスト ダイアグラムに関係します。

世代を表示

このオプションを設定すると、前の世代のバージョンも表示されます。

キャンセル

変更を取り消して、Version 1 を元の状態に戻します。

Difference Grouping

The model centric information view is used to group changes into sets of differences. Each model element has own set of properties. All differences related to those properties are added into the same group even if modifications have been done in different versions. The separate groups are created for differences in semantic and presentation models, see 124 ページの図 37. Each group has a representative element. The representative element for top-level groups is one of those elements which are visible in the model view, i.e. “Definition”, “Implementation” or “Diagram” in terms of UML meta-model. The following rules are applied while doing the grouping:

- Difference related to created or deleted entity is added into the group where representative element is an owner of created or deleted entity, see 125 ページの図 38.
- Differences related to moved entity are added into two groups. The “Moved (from)” difference is added into the group where representative element is an owner in ancestor version, i.e. the “old” owner. The “Moved (to)” difference is added into the group where representative element is an owner in modified version, i.e. the “new” owner, see 125 ページの図 39.
- Differences related to modified attributes are added into the group where representative element is the corresponding modified model element, see 125 ページの図 40.

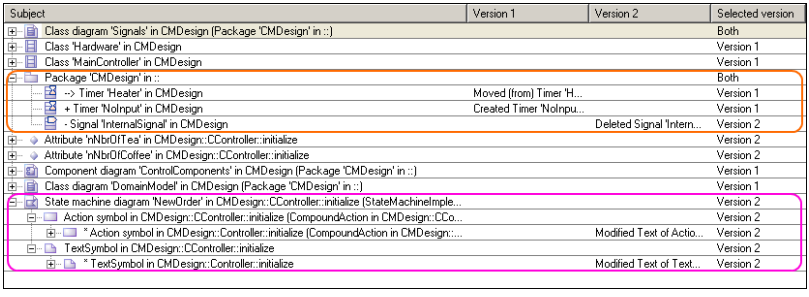


図 37: Semantic and presentation model differences grouping

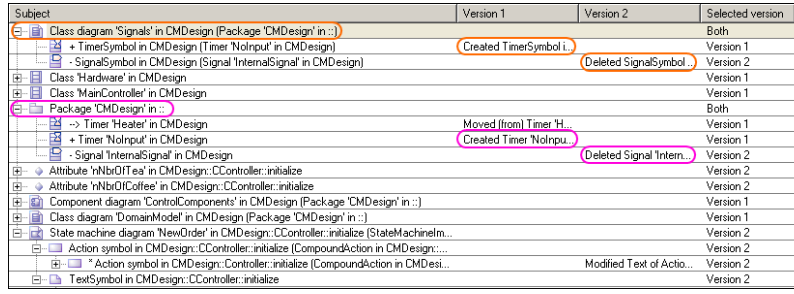


図 38: Grouping of “created entity” and “deleted entity” differences

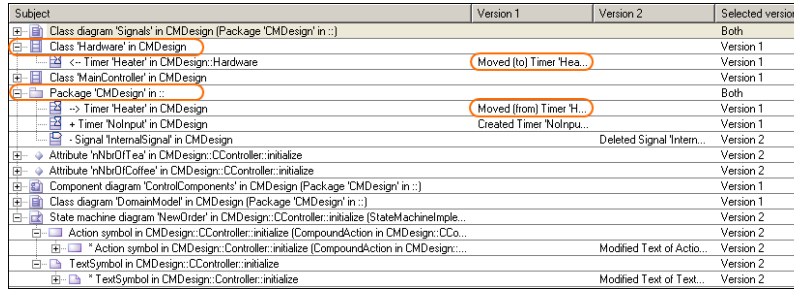


図 39: Grouping of “moved entity” differences

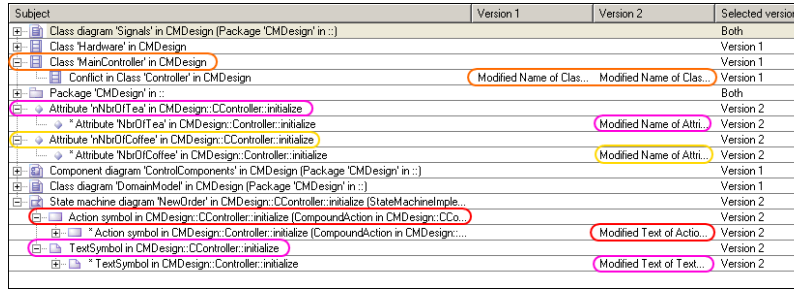


図 40: Grouping of “modified attributes” differences

In the 相違点リスト each group is represented by a set of sub-nodes. There are five main kinds of nodes (see 126 ページの図 41):

Composite Group Node

This composite node contains the set of differences directly or indirectly owned by the representative element. This group can contain all other kinds of nodes.

Composite Conflict Group Node

This composite node contains conflicting differences which are owned by different representative elements. For example, if entity has been moved in Version 1 and in Version 2 and the new owners of that entity are different in Version 1 and Version 2, then the Composite Conflict Group will be created. This group can contain Difference Nodes only.

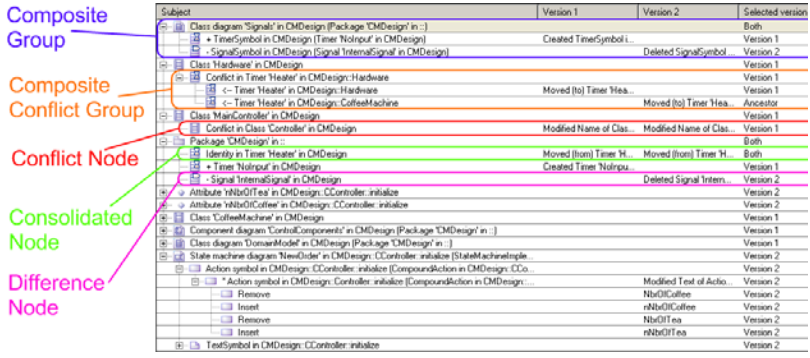


図 41: Nodes in Difference list

Conflict Node

This node corresponds to conflicting differences which are related to the same representative element.

Consolidated Node

This node corresponds to consolidated differences which are related to the same representative element.

Difference Node

This node describes the simple change that has been made in Version 1 or in Version 2.

By using the コンテキストメニュー it is possible to accept or reject any individual difference as well as a group of differences. The merge tool sets up additional relations between differences. Thus the accepting of some individual difference may automatically lead to accepting or rejecting another difference(s). Such relations are used in order to preserve semantic consistency of the merged models.

テキスト マージ

マージツールは、モデルベースのマージとテキスト マージの 2 種類のマージ手法を使用します。テキスト マージの目的は、モデルベースのマージでは充分には対応できないテキスト シンボル内のマージを可能にすることです。モデルベースのマージとテキスト マージは、ともにマージ ツールに組み込まれており、共通のグラフィカル インターフェイスから使用できます。したがって、相違点ダイアログでのレビューの相違点のリストには、モデルベースの相違とテキストの相違の両方が含まれる可能性があります。

動的相違

アクション シンボルやテキストシンボルなどのシンボルは、モデル要素（クラスのテキストによる説明など）を含むことがあります。したがってシンボル内のテキストは、モデル内の選択された他の相違点のバージョンによっては、マージ操作の最中に変更される可能性があります。このため、テキストのマージはモデルマージの最中に動的に行う必要があります。テキスト マージの相違点を含む全相違点には、最低でも一つの相違点（これ自体も全相違点）があります。113 ページの図 33 を参照してください。全相違点「Comment」はコメントシンボル内のテキストの相違を示しています。

Composite Textual Difference Node

This group node corresponds to a group of primitive textual differences. This group can contain Textual Difference Nodes only.

Textual Difference Node

This node corresponds to a primitive textual difference. There are two operations that represent modifications in the text: **Remove** and **Insert**. **Remove** means that a part of the text has been deleted (in comparison with the ancestor version). **Insert** means that a new text has been added.

テキストマージを開始するには、［V1 と結果］または［V2 と結果］リーフを選択します。実行されると今度はリーフが展開され、一連の相違点が表示されます。115 ページの図 34 を参照してください。

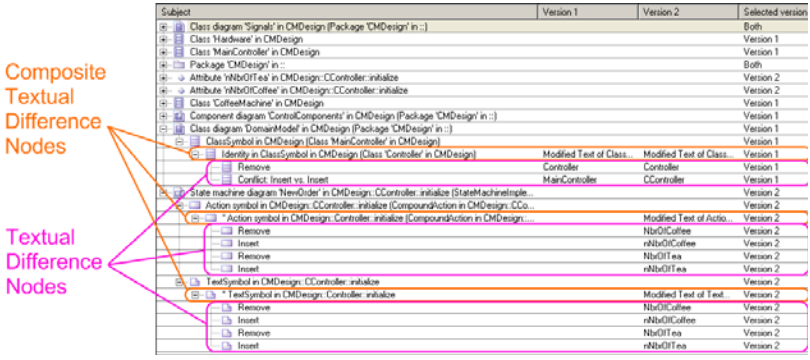


図 42: Textual difference nodes

バージョンの選択

As well as for the semantic and presentation model differences it is possible to accept or reject the group of textual differences or an individual textual difference. The merge tool sets up relations between textual differences and semantic model differences. As soon as textual difference is accepted/rejected the corresponding semantic difference is accepted/rejected. And vice versa, as soon as semantic difference is accepted/rejected, the corresponding textual difference is accepted/rejected.

When Textual Difference Node is selected in 相違点リスト , the modified part of code is highlighted in the one or several windows which represent Ancestor, バージョン 1, バージョン 2 and 結果 models, see 図 43 on page 129.

If the selected node corresponds to **Remove** operation, then the removed part of the text is selected in Ancestor window (and in 結果 window, if that operation is rejected). If the selected node corresponds to **Insert** operation, then the inserted part of the text is selected in バージョン 1 (and/or バージョン 2) window (and in the 結果 window, if that operation is accepted).

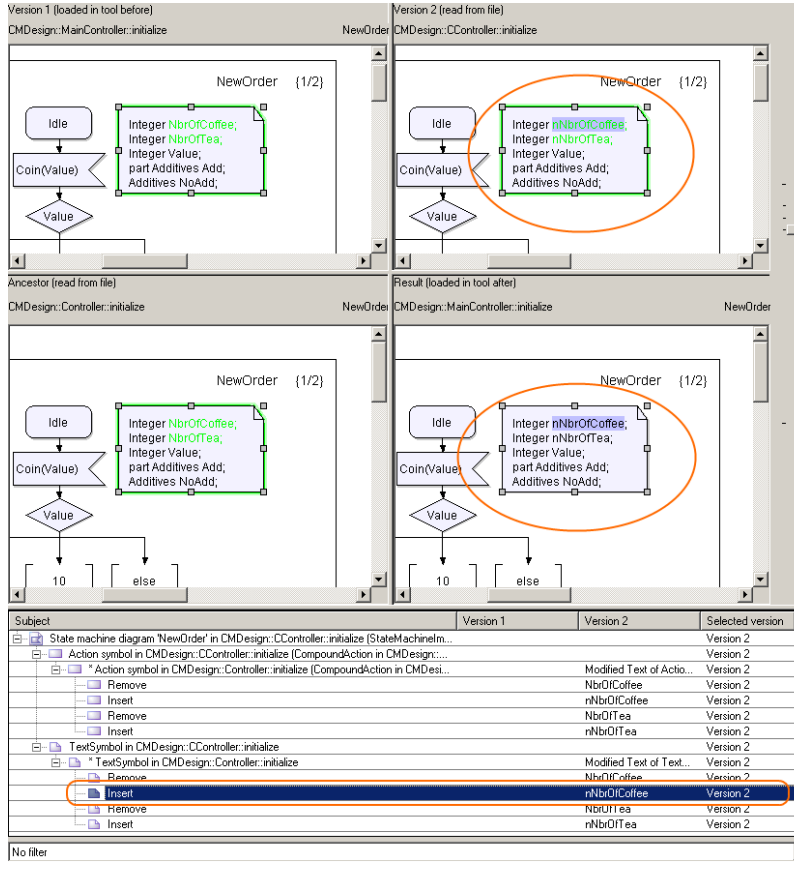


図 43: Textual difference highlighting

カラーリング

The colors are used in the 相違点ダイアログでのレビュー in order to simplify the understanding of changes that have been done in both versions. The **blue** color is used to mark presentation elements that correspond to model elements modified in version 1 only. The **green** color is used to mark presentation elements corresponding to model elements that have been modified in version 2 only. And presentation elements that correspond to model elements modified in both versions are marked by **red** color. The same coloring is applied to modified presentation model elements, i.e. Symbols and Lines as well as to parts of the text.

バージョンの選択

通常は、あるバージョンの全相違点を 1 つの単位として承諾または拒否し、1 つの全相違点内の個別の相違の単位では選択できません。しかしテキストの相違の場合は、個別の相違の選択が可能です。テキストに追加／削除されたモデル要素ごとに、表示／非表示を設定できます。このためには、［V1 と結果］と［V2 と結果］内の対応するモデル要素を承諾または拒否します。相違を示しているテキスト行を選択して、この相違を承諾するか拒否するかを選択できます。個々のテキスト行は、テキストシンボルでの対応する行番号とともに［挿入］または［削除］として表示されます。［挿入］は、相違の挿入を承諾するとそのテキストが結果ウィンドウに**挿入**されることを意味します。［削除］は、相違の**削除**を承諾するとそのテキストが結果ウィンドウから取り除かれることを意味します。

テキスト相違のコミット

テキストの挿入と削除を実行した後、結果ウィンドウのテキストはモデルにコミットできます。これは相違の［挿入］または［削除］の選択を外す（たとえば、最上位レベルの全相違点を選択する）ことで実行されます。

外部テキスト比較 / 外部テキストマージ

外部テキストの比較とマージ ツールを使用して、コメント、テキストシンボル、タスクシンボル、およびインスタンス式の比較とマージを行うことができます。相違点または競合を右クリックすると、状況に応じて［外部テキスト比較］と［外部テキストマージ］のオプションが利用できるようになります。

外部テキストのマージを実行すると、マージ結果をモデルに再入力できるかどうかチェックされます。モデルに入力できない場合、マージ結果が外部ツールから保存され、エラー メッセージ ボックスにそのパスが一緒に表示されます。

コマンドラインの設定

外部テキストの比較 / マージ ツールのパスとコマンドライン スイッチは、［ツール］メニューから開く［オプション］ダイアログの［比較 / マージ］タブから利用できます。

作業開始に使用する基本モデル

モデルの操作を行う場合、モデル ベリファイヤ（Model Verifier）で実行できるように、一定レベルまで完成していなければなりません。この完成レベルは要件によって異なります。また、チェック機能ですべてをチェックするわけではありません。チェック機能では、モデルの観点から構文とセマンティックをチェックしますが、コード生成の観点からのチェックではありません。モデルはチェックに合格しなければなりませんが、コード生成を妨げる制限がまだ存在している可能性があります。

このセクションでは、モデル ベリファイヤ（Model Verifier）、または C コード アプリケーションのコード生成を完成させるための基本的要件について説明します。どちらも似ており、UML とアプリケーション コード間の同じマッピング テーブルでビルドされます。

参照

プロジェクトで新しいワークスペースを作成する方法については、第 66 章「ツール環境の設定」を参照してください。

第 4 章「UML 言語ガイド」の 161 ページ、「UML 言語ガイド」

初期設計

アクティブ クラスと振る舞い

通常、UML の設計は、パッケージをコンテナとして定義するところから始めます。モデル ベリファイヤ（Model Verifier）またはアプリケーションとして実行するシステムには、アクティブ クラスが必要です。動的振る舞いは、通常、状態機械 モデル要素に属する、状態機械実装でモデル化されます（一般に、**initialize** と名付けられるか、属するアクティブ クラスに因んで名づけられます）。このようなカプセル化された要素は、新しい状態機械図の作成時に作成されます。ワークフローの基本的なステージの 1 つに、システムの最上位レベルのアクティブ クラスを決定する作業があります。

モデル例

モデル ベリファイヤ（Model Verifier）を作成するためには、ビルド アーティファクトが必要です。アーティファクトがない場合、ビルド タイプとビルド ルートを指定するように指示されます。実行形式モデル（132 ページの図 44）は以下の要素で構成されます。

- パッケージ（オプション）
- クラス図（オプション）
- 少なくとも 1 つのアクティブ クラス
- 状態機械実装のある少なくとも 1 つの状態機械（オプション、暗黙的も可能なので）
- 少なくとも 1 つのビルド アーティファクト

ショートカット メニューの［新規］とそのサブメニューでモデル要素を作成し、モデル内のワークスペース ウィンドウで直接モデル内でモデルを設計できます。ただし、この方法では、作業をパッケージ内にクラス図の作成から始める必要があります。

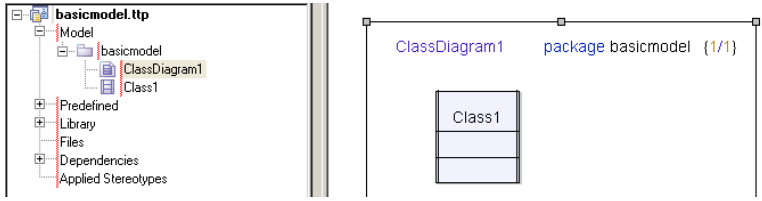


図 44: アクティブ クラスを持つモデル

上記の例でモデル ベリファイヤ（Model Verifier）を実行すると、実際の振る舞いを持たない暗黙的な状態機械が生成されます。

シグナリングの例

UML シグナルとインターフェイスに応じた形態の通信を行うためには、アクティブ クラスにはポートが必要です。ポートは、送信可能なシグナルまたはインターフェイスで定義されます。（内部または外部）通信を行うモデル（133 ページの図 45）には以下の要素が必要です。

- 少なくとも 1 つのシグナル（動的振る舞いを定義する状態機械で使用）
- 少なくとも 1 つのポート
- インターフェイス（オプション）

インターフェイスを使用してシグナルをカプセル化する作業も、UML モデルを設計するワークフローにおいて基本的なステージです。

状態機械

状態機械でコードを生成できますが、中にはシミュレート可能な振る舞いをもたない暗黙的な状態機械があります。

モデルに単純状態機械を追加すると、振る舞いが見やすくなります（133 ページの図 46）。

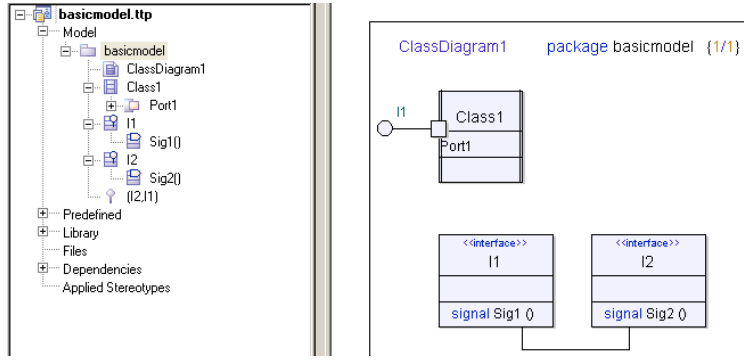


図 45: インターフェイスとシグナルを持つモデル

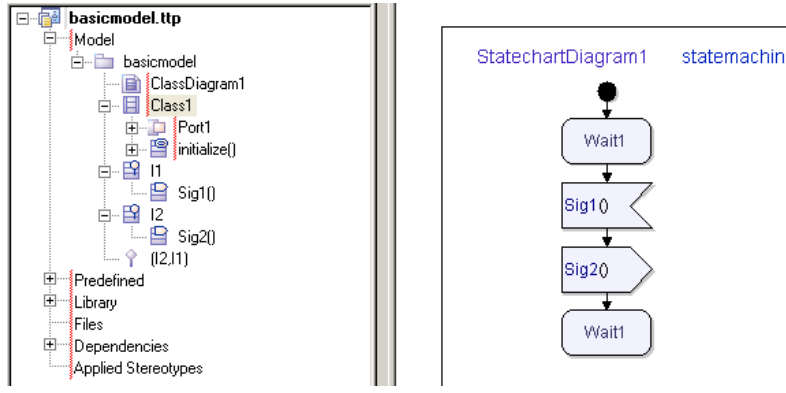


図 46: シグナルを持つ状態機械

内部通信

オープン システムとクローズド システム

133 ページの図 45 のモデルはオープン システムを記述します。ここで言うオープン システムとは、環境と相互作用するシステムを指します。クローズド システムは相互作用がシステムに組み込まれたオープン システムから設計されることがあります。

以下に、Class 1 をパートとしてもつ、もう 1 つのアクティブ クラス（Class 2）を加えて拡張した例を示します。この例では、新しいインターフェイス（I3）とシグナル（Sig3）が追加されています（134 ページの図 47）。

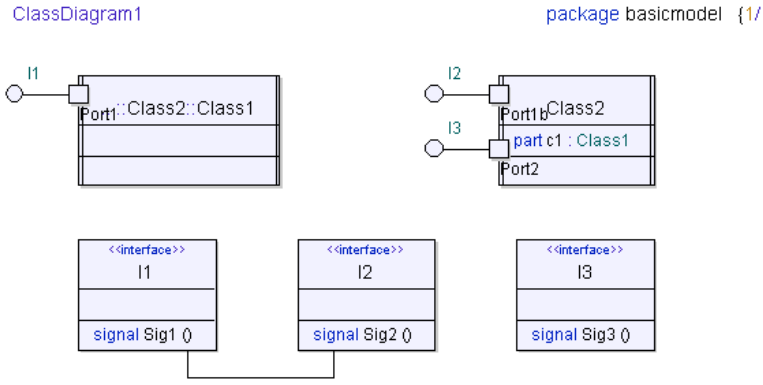


図 47: Class 2 と新しいインターフェイス

ポートと振る舞いポート

以下の例では、環境との通信にポートが使用されています。ポートは、同じモデルの他のパートとの通信にも活用されます。より規模の大きい設計では、アクティブ クラスを内包するアクティブ クラスがあります。振る舞いポートを利用することで、パートとパートを持つクラス（のインスタンス）内の状態機械間のコミュニケーションを表示するように設計できます。

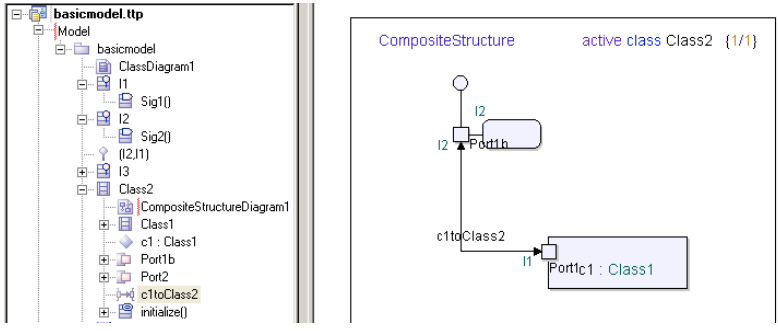


図 48: 合成構造図の振る舞いポート

アーキテクチャ

合成構造図は、モデル内のシグナルの送信方法が不明瞭な場合にのみ必要です。合成構造図は、複雑なアクティブ クラスを持つモデルがどのように構造化されているかを明示的に示す際、非常に便利です。最上位クラスにパートとの通信を必要とする状態機械がある場合、このクラスは振る舞いポートを使用する合成構造図でモデル化できます（134 ページの図 48）。

Class 2 の状態機械を 135 ページの図 49 に示します。

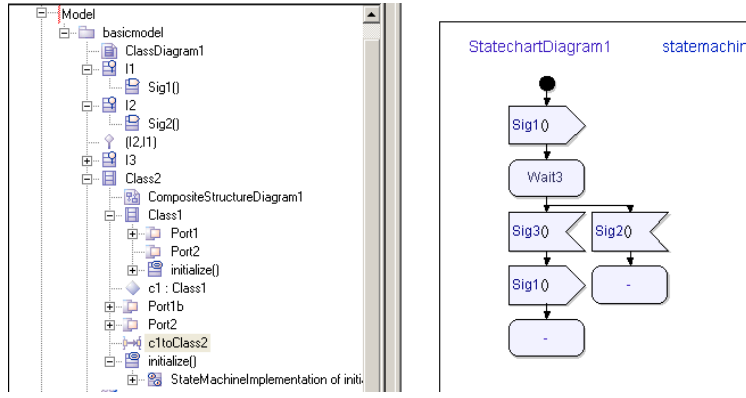


図 49: Class 2 の状態機械

3

ダイアグラムの操作

プロジェクトを開いていれば、もうモデル編集の準備が整っています。

モデルを編集する際、ワークスペース ウィンドウで［モデル ビュー］タブをアクティブにすると、画面が大変見やすくなります。このビューから、ダイアグラムで行った変更が、モデルにどのような影響を与えるかをすぐに確認できます。

ダイアグラム エディタをモデル情報と組み合わせて使う場合、使い方は作成するアプリケーションによって異なります。以下は、ツールに慣れて、必要に応じてワークフローを自由に採用したり、変更できるようにするための推奨事項です。

ダイアグラムの一般的な操作方法

ダイアグラムの一般的な操作方法：

- ダイアグラムの作成
- ダイアグラムを開く、保存、印刷
- ダイアグラムの移動
- ダイアグラムのサイズ変更
- 新しいファイルで保存
- 検索
- テキスト解析
- ダイアグラムの自動レイアウト
- ビューの体系化

グリッド

ダイアグラム描画エリアにグリッドを表示します。グリッドが表示されると**グリッドに吸着**する機能が**オン**になります。グリッドの間隔は 2 ミリに設定され、変更できません。ショートカット メニュー、または、［オプション］の設定変更で、グリッドを表示または非表示にできます。デフォルトでは非表示です。シンボル、ライン、テキスト フィールド（シンボルに固定されたものを除く）は、すべてグリッドに吸着します。シンボルのサイズ変更もグリッドのメモリ単位に行われます。自動サイズ調整の場合も同様です。

フレーム

すべてのダイアグラムには、すべてのシンボルを内包するフレームがあります（ただしポート シンボルはフレーム上に配置）。キャンバスの左上隅から（x=10, y=10）ミリのところに、左上隅のフレーム シンボルが配置されます。フレームのサイズはダイアグラムのサイズとレイアウトに合わせて設定されます。フレームの上または外側にシンボルを置くため間隔を広げる必要があれば、サイズを変更できます。

フレームは、キャンバス スペースの許す限り、どの方向にもサイズ変更または移動できます。

ヘッダー

ダイアグラムのヘッダーは、ダイアグラムの左上隅に配置されます。テキストは右揃えです。テキストの位置は定義するエンティティのプロパティから算出されます。

ダイアグラム名

ダイアグラム名は、ダイアグラムの右上隅に配置されます。テキストは右揃えです。ダイアグラム名はモデル情報から算出されるため、［モデル ビュー］からのみ変更できます。

ダイアグラムの作成

ダイアグラムには、モデル要素を表示する一連のプレゼンテーション要素が含まれます。ダイアグラムはワークスペース ウィンドウから管理します。

1. **ワークスペース** ウィンドウで、[モデル ビュー] タブをクリックする。
2. 適切なパッケージを選択または作成する。
3. これで、ショートカット メニューから [新規] を選択して、適切なダイアグラム（または、モデル要素）を作成できるようになります。

参照

第 2 章「モデルの操作」の 89 ページ、「プレゼンテーションの作成」

ダイアグラムを開く、保存、印刷

プロジェクトを保存する際、ダイアグラムの情報がファイルに保存されます。特にファイルの指定がない場合、この情報はデータファイル（.u2 拡張子）に保存されます。このファイルは、通常、現在使用中のプロジェクト ファイル（`.ttp`）とワークスペース ファイル（`.ttw`）と同じ場所に保管されます。

- ダイアグラムを開くには：[モデル ビュー] のダイアグラム アイコンをダブルクリックする。
- 開いたダイアグラムを印刷するには：[ファイル] メニューから [印刷] をクリックする。

新しいダイアグラムの作成時、ダイアグラムのサイズはプリンタ設定から導き出されます。たとえば、プリンタの印刷の向きが横に設定されていれば、ダイアグラムも横に設定されます。

ダイアグラムイメージの保存

ダイアグラムは、JPEG、GIF、BMP、SVG などさまざまな形式の画像としてエクスポートできます。

ダイアグラムを画像として保存するには、ダイアグラムを開き、[ファイル] メニューの [名前を付けて保存] を選択して、表示されるダイアログで画像のファイル形式を指定します。

新しいファイルで保存

UML 要素を別ファイルに保存する操作が用意されています。この操作は、ワークスペース ウィンドウで [モデル ビュー] のショートカット メニュー [新しいファイルで保存] を選択して行います。このショートカット メニューを選択すると、保存するファイル名と要素を指定するよう指示されます。

以下のモデル要素を新規ファイルに保存できます。

- アーティファクト

- クラス
- 実装
- パッケージ

参照

第 69 章「印刷」の 2130 ページ、「印刷するダイアグラムの選択」
第 72 章「ダイアログ ヘルプ」の 2155 ページ、「保存」

ダイアグラムの移動

ダイアグラムは同じプロジェクト内、または、同じワークスペース内のプロジェクト間で移動できます。

- ［モデル ビュー］でダイアグラムに対応するアイコンをクリックし、このダイアグラムを目的の場所にドラッグする。

ダイアグラムのサイズ変更

ダイアグラムを作成すると、デフォルトで、自動サイズ調整モードに設定されます。ダイアグラムのショートカット メニュー、またはダイアグラム要素プロパティツールバーで、このモードのオン／オフを切り替えられます。ダイアグラムが自動サイズ調整モードに設定されていると、ダイアグラムの要素をエディタ キャンバスの好きな場所に、ドロップ、挿入、貼り付け、移動ができます。要素をフレームの外側に配置すると、ダイアグラムのサイズはその要素に合わせて自動的に変更されます。自動サイズ調整モードでは、常に、ページ全体が全ダイアグラム要素を収容できる最小値に設定されます。

ダイアグラムのサイズとレイアウトの初期設定は［印刷設定］（サイズと印刷の向き）で決定されます。プリンタがインストールされていない場合、サイズは現在の［オプション］設定で決定されます。ダイアグラムのサイズ変更は、以下の 2 つのステップで行います。ダイアグラムを拡大する場合、以下のいずれかの方法によって行います。

- ショートカット メニューでダイアグラムのサイズを変更する。［モデル ビュー］のダイアグラムを右クリックし、［ダイアグラム サイズ］を選択します。表示されたダイアログで手動によるサイズ変更を選択したり、自動サイズ調整モードに戻せます。
- 自動サイズ調整がオフの場合、Ctrl キーを押しながらフレーム シンボルのドラッグ ハンドルをクリックすると、ダイアグラムの用紙サイズ全体が段階的に拡大される。Ctrl+ Shift キーを押しながらダイアグラムのドラッグ ハンドルをクリックすると、サイズが縮小されます。押したキーに応じてマウス カーソルの形が変わります。

ダイアグラムのサイズに合わせて、フレーム シンボルのサイズが変わります。ダイアグラムのサイズを縮小する場合、フレーム シンボルが、ダイアグラム内のすべてのシンボル（またはライン）より小さくなることはありません。フレームは、内側のシンボル（ライン）から、1 グリッド ポイント以上離します。ダイアグラムのサイズを変更すると、ダイアグラム名シンボルは自動的に移動します。フレームに置かれたポー

トとラインはフレームの動きに連動して、移動します。キャンバスのサイズは、フレームと余白を合わせた大きさよりも小さくなることはありません。フレームのサイズ変更はグリッドの間隔で行われます。

検索

［編集］メニューから［検索］を選択して、［ダイアグラムと定義の検索］ダイアログを開きます。［ダイアグラムと定義の検索］ダイアログで定義を検索できます。定義が使用されている場所を検索するには、［ダイアグラム中のテキストも検索する］オプションを指定します。結果は出力ウィンドウの［検索結果］タブに表示されます。エンティティが使用されている場所を一覧表示するには、［モデル ビュー］のエンティティを右クリックし、ショートカット メニュー［参照の一覧表示］をクリックします。

参照

モデル インデックス

テキスト解析

一般に、テキスト シンボル（および外部ファイル）には C++ スタイル構文が使用されますが、他のすべてのシンボルには UML スタイル構文が使用されます。パーサは UML からのわずかな逸脱（「+」ではなく「public」、「^」ではなく「output」など）であれば受け入れますが、すべてを UML スタイルに逆構文解析します。テキスト シンボルでは、UML スタイルの逸脱は受け入れられますが、C++ スタイルに変換されます（可視性演算子の「+」は「public」に変換）。

これらの変換は以下のカテゴリに分類されます。

- 可視性：+ を `public` に変換
- 可視性：- を `protected` に変換
- 可視性：# を `private` に変換
- シグナル送信：^ を `output` に変換
- 分岐選択肢：`else` を `default` に変換

逆構文解析フェーズの他のプロパティもあります。

- パラメータ方向 “in”、逆構文解析されません。
- 引用符を必要としない名前であれば、引用符が付いた名前から引用符がなくなります。（逆構文解析後 `'Name1'` は `Name1` になります。
- パラメータ方向 `"in / out"` は、逆構文解析後、`"inout"` になります。
- リテラルを含むデータ型のみ列挙データ型になります。つまり、`datatype colors { literals red, green; }` は逆構文解析されて `enum colors { red, green }` となります。

逆構文解析でショートカットの表記を展開できます。一度に定義された複数の属性、リモート変数、シグナル、タイマー、例外または同義語（たとえば、`Integer i, j, k;`）は、逆構文解析後、複数の個別の定義（`Integer i; Integer j; Integer k;`）に展開されます。

アンパーサは、シグナルやタイマーの定義で省略されていた丸かっこを追加します（"timer T" は "timer T()" になります）。

範囲開始値は（可能な場合）、次の UML スタイルに変換されます：">= n" は "n..*" に変換され、">=0" は "0..*" に変換されます。

自動引用符付け

自動引用符付けの目的は、引用符 の入力を支援することです。名前に空白を入れる場合は、その両側に引用符を追加する必要があります。自動引用符付けが適用されるのはごく一部のシンボル（ラベル）のみです。たとえば、名前が含まれているラベルなどは、自動引用符付けされます。

ワード ラップ

ワード ラップでは、語を複数の行に分割できます。この機能は、複数行ラベルと自動サイズ調整されないシンボルに適用されます。語の分割箇所を決定するため、次の文字や符号が検索されます。「:」、「::」、空白、大文字、カンマ（「,」）、ピリオド（「.」）、アンダスコア。

ダイアグラムの自動レイアウト

ダイアグラム（キャンバス背景）のショートカット メニューには、自動レイアウト アルゴリズムが関連付けられている、ダイアグラム タイプ用の、［自動レイアウト］メニュー項目があります。メニュー項目を選択すると、ダイアグラム要素は、特定のダイアグラム タイプに適したレイアウトに配置されます。たとえば、クラス図と状態機械図には階層的なレイアウトがあります。

［要素の表示］ダイアグラムを使用してダイアグラム要素を配置する際、自動レイアウト アルゴリズムが使用されます。

自動レイアウトを使用する際、以下のことを考慮する必要があります。

- クラス図のレイアウト アルゴリズムに含まれるのは汎化ラインだけ。
- 状態機械図のレイアウト アルゴリズムにはフロー ラインと遷移ラインが含まれる。

ビューの体系化

エディタには、ビューを体系化するいくつかの機能があります。これらの機能にはスクロール、ズームイン／アウトなどのショートカットが含まれます。

ダイアグラムを閉じるときに、現在のスクロールとズームの設定を個別のファイルに保存できます。ファイルの拡張子は .u2x、名前は <project>_DiagramSettings です。このファイルはプロジェクト（ttp ファイル）に追加されません。プロジェクトのロード時に、拡張子 .u2s と対応する名前を持つファイルをロードするステップがあります。この機能は、［スクロールとズームの設定を記憶する］オプションで設定できます。

スクロール

ダイアグラムが、デスクトップに全体表示できないサイズに設定されている場合、ビューをスクロールできます。スクロールにはウィンドウ スクロール バーを使用します。

（**Windows** の場合）インテリマウス ポインティング デバイスでスクロールすることもできます。

- スクロール ホイールを使用して、垂直方向にスクロールする。
- Ctrl キーを押しながらスクロール ホイールを使用して、水平方向にスクロールする。

ズーム

［表示］メニューの［ズーム］コマンドを使用して、固定倍率で段階的に拡大／縮小できます。

ショートカット メニューで連続ズームを実行できます。ダイアグラムを右クリックし、［ズーム］にカーソルを合わせて、目的の拡大レベルを選択します。テキスト編集モードではない場合、マイナス記号（-）記号を使用してズームアウト、プラス記号（+）記号を使用してズームインできます。

（**Windows** の場合）インテリマウス ポインティング デバイスでスクロールできます。

- Shift キーを押しながらマウスの中央 ボタンを使用して、ダイアグラムをズームする。
- スクロール ホイールをダブルクリックして、等倍表示にする。
- Shift キーを押しながらスクロール ホイールをダブルクリックする。現在開かれているダイアグラム（カレント ダイアグラム）全体がデスクトップに表示されるようサイズが調整されます。

参照

ウィンドウのドッキング
ワークスペースの操作

共通のシンボルの操作

- シンボル情報
- シンボルの追加
- 要素の表示
- シンボルの選択
- シンボルの移動
- シンボルのサイズ変更
- シンボルの接続
- シンボルのテキスト フィールドの編集
- ダイアグラム要素のプロパティ

- コメントの処理
- シンボルのコピー、切り取り、削除、貼り付け
- アイコン
- イメージ セレクタ
- 元に戻す
- モデル参照
- モデルの更新
- ネストされたシンボル

シンボル情報

ダイアグラムのシンボルやラインにカーソルを合わせてしばらく置くと、ツールチップにコンテキスト依存情報が表示されます。この機能は［ツール］メニューの［オプション］コマンドから開くダイアログの［UML 詳細編集］タブで制御できます。

［シンボルとラインのツールチップを表示］を選択すると、ステレオタイプやバインド情報などのコンテキスト モデル情報を表示できます。

［表示モードのツールチップを表示］を選択すると、テキスト編集モードで構文解析情報を表示できます。

モデル要素の詳細の表示 / 隠すツールバー

［モデル要素の詳細の表示 / 隠す］ツールバーを使用して、ダイアグラム内のシンボルの特定機能の表示 / 非表示を切り替えられます。これらの設定はダイアグラムごとに保存され、ダイアグラム内のすべての要素に同じ設定が適用されます。

- Show/Hide qualifiers

  ラベル テキストの修飾子部分を切り替えます。たとえば、次のようになります。Package1::Package2::Class1 will be toggled to Class1.
- Show/Hide stereotypes

  ステレオタイプ ラベルの表示 / 非表示を切り替えます。ラベルを非表示にすると、ラベルが占有するスペースが最小と見なされ、結果的にシンボル サイズに影響する場合があります。
- Show/Hide quotation marks

  自動引用符の表示 / 非表示を切り替えます。これによって、一部のシンボルが自動引用符付けされ、このボタンの影響を受けます。引用符を非表示にしても、テキストはそのまま引用符付きと見なされます。自動引用符付けされるテキストは、通常の場合空白が含まれている名前です。

シンボルの追加

シンボルを追加するには、［ダイアグラム要素の作成］ツールバーの対応するアイコンをクリックしてから、ダイアグラム内でクリックまたは右クリックしてシンボルを配置します。

［モデル ビュー］からシンボルを生成することもできます。この場合、モデル要素を目的のダイアグラムにドラッグします。

状態機械図は、Ctrl キーを押しながら、新規シンボルの［ダイアグラム要素の作成］ツールバーをクリックしてフローへのシンボルの挿入を行えます。このシンボルは、現在選択されているシンボルの後に挿入されます。

既存要素を参照

ほとんどのシンボルについて、シンボルを右クリックするとショートカット メニューが表示されます。このメニューを持たないシンボルを以下に示します。

- 遷移ライン（状態（ステート）指向ビューでデザインされた場合、状態機械図で使用）
- ステート、シグナルおよび操作に関連付けられていない状態機械遷移シンボル

ショートカット メニューには以下のように、［新規 < モデル要素 > の作成］、［未接続を維持］、［既存要素を参照］の 3 つの選択肢があります。

- **新規 < モデル要素 > の作成**：新しいシンボルが作成され、シンボルに対応するモデル要素がモデルに作成されます。
- **未接続を維持**：新しいシンボルが作成されますが、これに対応するモデル要素は作成されません。
- **既存要素を参照**：タイプとスコープに合致する既存のモデル要素がドロップダウン ボックスに表示されます。

自動配置

前回のシンボルと接続して、シンボルを配置する場合が多くあります（たとえばクラスのポートなど）。このような配置を行うために、シンボルの自動配置機能があります。

- Shift キーを押したまま、シンボル ツールバーをクリックする。クリックしたシンボルは、現在選択しているシンボルに接続されます。
- Ctrl キーを押したままシンボル ツールバーをクリックする。クリックしたシンボルは、現在選択しているシンボルと次のシンボルの間に挿入されます。

シンボルが現在選択されているシンボルに対して、構文上正しいフローで接続されない場合、これらのシンボルはツールバーでグレー表示されます。

Shift + スペースキーおよび Ctrl + スペースキーを使用して、自動配置可能なシンボルのリストを表示することもできます。

参照

第 2 章「モデルの操作」の 61 ページ、「名前のサポート」
第 2 章「モデルの操作」の 89 ページ、「プレゼンテーションの作成」
第 2 章「モデルの操作」の 90 ページ、「モデルのナビゲートと作成」
ダイアグラムの自動レイアウト

要素の表示

［要素の表示］ダイアログで、現在のダイアグラムに表示するモデル要素を選択します。

［要素の表示］は以下のメニューから選択できます。

- ［ツール］メニュー
- ダイアグラムのショートカット メニュー

［要素の表示］を選択すると、現在のダイアグラムのシンボルとして表示可能なモデル要素のリストが表示されます。モデル要素にチェックマークを付けるか解除して、これに対応するシンボルをダイアグラムに追加あるいは削除できます。

［要素の表示］ダイアログには以下の機能があります。

- ［すべて］ボタンを 1 回クリックして、リスト内のすべてのモデル要素にチェックマークを付ける。
- ［なし］ボタンを 1 回クリックして、チェックマークをすべて解除する。
- ［短く表示］チェック ボックスで、モデル要素の概要リストと詳細リストを切り替える。
  - **概要リスト**には、現在のダイアグラムで一般的に使われるモデル要素が含まれます。（たとえば、クラス図のクラスなど）一般的なシンボルでなくても、すでに現在のダイアグラムにシンボルとして表示されているモデル要素は概要リストに含まれます。
  - **詳細リスト**には、選択されたスコープにある、現在のダイアグラムのシンボルとして表示可能なモデル要素がすべて表示されます。このリストには、現在のダイアグラムで一般的に使われるモデル要素のほか、特殊な変換も含まれます。（たとえば、ユース ケース図にアクターとして表示されるクラスの選択肢も詳細リストに含まれます。）
- ［スコープの選択］ボタンをクリックして、ダイアログを表示する。このダイアログからスコープを選択してメイン ダイアログに表示するモデル要素を選択します。デフォルトで、ダイアグラムが属するローカル スコープのモデル要素だけがこのリストに含まれます。

シンボルの選択

Ctrl キーを押しながらテキスト フィールドの外側（シンボル枠の内側）をダブルクリックすると、シンボル、外向きラインと接続されたすべてのシンボルを選択します。

シンボルまたはライン以外の場所でクリックしてドラッグすると、選択矩形が作成されます。この矩形の中にあるものがすべて選択されます。

Ctrl キーを押しながら、シンボルまたはライン以外の場所でクリックしてドラッグしても、選択矩形が作成されます。この場合、矩形に接触したものがすべて選択されます。

状態機械 フローでは、Ctrl キーを押しながら、フローのシンボルをダブルクリックすると、該当シンボルと後続のシンボルすべてが選択されます。フローが分岐している場合も、この機能を利用できます。

シンボルの移動

シンボルを移動するには、シンボルをクリックしてダイアグラム内の希望する場所にドラッグします。シンボルを他のダイアグラムにドラッグすることもできます。

テキスト フィールドのあるシンボルの選択では、テキスト編集モードにならないようにします。テキスト編集モードではカーソルの形状が変わります。

テキスト フィールドの移動

いくつかのテキスト フィールド（ラベル）を移動できます。具体的には、ラインに属すラベルと、そのラベルがシンボル境界の外にあるシンボル（ポート、ピンなど）に属すラベルを移動できます。

この操作を実行するには、最初にラベルを選択します。その後にラベルをいずれかのハンドルでドラッグできます。新しい位置は、デフォルト位置のオフセットとして保存されます。ラベルが属すラインまたはシンボルを移動すると、ラベルも移動してオフセットが保持されます。

オフセットを解除するには、ショートカット コマンド［すべてのラベル位置のリセット］をクリックします。現在選択されているシンボルに属するすべてのラベルがそのデフォルト位置にリセットされます。

ラベルはデスクトップ上の任意の位置にドラッグできます。フレーム シンボルやダイアグラム領域の外の場合でも有効です。ダイアグラムの外のラベルは出力されません。

シンボルのサイズ変更

シンボルのサイズを手動で変更するには

1. 該当するシンボルを選択します。
2. 8 つあるグレーの正方形のうち 1 つにマウスのカーソルを合わせます。
3. マウスで、シンボルを目的の大きさまでドラッグします。

自動サイズ変更

すべてのシンボルで、［自動サイズ変更］を選択して中に入力したテキストのサイズにあわせてシンボルのサイズを自動変更できます。シンボルを右クリックして、ショートカット メニューから［自動サイズ変更］を選択します。

シンボルを折りたたむ

コンパートメント（クラス シンボルなど）のあるシンボルは、そのシンボルのショートカット メニューから［折りたたむ］メニュー項目をチェックして折りたたむことができます。コンパートメントとその中のラベル類は折りたたんだ状態では表示されません。

サイズ変更後のシンボル表示

シンボルの右下隅の外側に 3 つの点が表示された場合、シンボルのサイズが小さすぎてシンボルのテキスト フィールドにあるテキストをすべて表示できないことを意味しています。テキストに合わせてシンボルをサイズ変更するには、シンボルを選択して 3 つの点をダブルクリックします。

シンボルの接続

シンボルを手動で接続するには

1. シンボルをクリックして、ライン ハンドルを見つけます。
2. ラインを他のシンボルにドラッグします。
3. 移動先のシンボルに到達すると、ラインの先端に十文字の付いた丸が表示されます。シンボルの内側のラインを接続する境界に近い位置でクリックして、接続を完了します。

接続の結果、モデル要素となる場合もあります。たとえば、汎化ハンドルをクラス図の他のクラスにドラッグすると、2 つのクラス間のラインが作成され、同時に［モデル ビュー］に新しいアイコンが作成され、汎化が追加されたことが示されます。

状態機械図のシンボルは、自動配置でフローに自動的に接続されます。

リンクツールバーから 依存リンクの追加を行えます。

参照

ラインの描画

シンボル フローの編集

フローまたはフロー ブランチの選択

アクティビティ図のフローで、Ctrl キーを押しながらフローのシンボルをダブルクリックすると、該当シンボルと後続のシンボルがすべて選択されます。フローが分岐している場合も、この機能を利用できます。

フローへのシンボルの追加

アクティビティ図にシンボルを追加する際、接続されたシンボルのフローを作成できます。Shift キーを押しながらツールバーをクリックします。クリックしたシンボルは、現在選択しているシンボルに接続されます。シンボルが現在選択されているシンボルに対して、構文上正しいフローで接続されない場合、これらのシンボルはツールバーでグレー表示されます。

参照

自動配置

フローへのシンボルの挿入

フロー ラインまたは遷移ラインが選択された状態でＣｔｒｌ キーを押すと、フローに挿入できるシンボルのみ、シンボル／ライン作成ツールバーで選択できる状態になります。

以下のいずれかを選択した状態で、Ctrl キーを押しながらボタンをクリックすると操作を挿入できます。

- シンボルを 1 つ選択した場合（このシンボルの後に操作が挿入されます)。
- ラインを 1 本選択した場合（このライン上に操作が挿入されます)。
- 1 つのラインで結ばれた 2 つのシンボルを選択した場合 (シンボル間に操作が挿入されます)

注記

- Ctrl キーを押しながら分岐シンボルを選択することはできません。分岐シンボルからは複数の出力フローが可能であるためです。

参照

自動配置

フローからのシンボルの削除

シンボルがフローから切り取り、または、削除された場合、削除されたシンボルとその接続ラインが自動作成ラインに置き換わります。

シンボルのテキスト フィールドの編集

シンボルのテキストフィールドを編集するには、まずシンボルを選択する必要があります。

- テキスト フィールドを編集するには、シンボルを選択して、そのテキスト フィールドの追加または変更したい場所をクリックします。これで、テキストを変更できます。ステレオタイプ情報など << >>（ギルメット）で囲まれたテキストは編集できません。
- シンボルを選択してテキストフィールド内でダブルクリックすると、最も近くにあるテキストが選択されます。

- シンボルを選択してテキスト フィールドをクリック アンド ドラッグすると、編集モードに入って、テキストを選択できます。シンボルが選択されていないと、この操作でシンボルは移動します。
- シンボルを選択して F2 キーを押すと、シンボル内のメイン テキストを編集できます。テキストが 1 行の場合すべてのテキストが選択されますが、テキストが複数行に渡る場合は、テキストは選択されず、テキストの末尾にテキスト カーソルが表示されます。

シンボルに誤った構文のテキストを入力すると、最初のエラー箇所に赤いエラー マークが表示されます。入力中、テキストは継続的にチェックされます。

注記

テキスト フィールドの外側（シンボル外枠の内側）をダブルクリックすると、シンボルのダブルクリック操作が実行されます（通常はナビゲーション）。

ダイアグラム要素のプロパティ

［ダイアグラム要素のプロパティ］と呼ばれるツールバーがあります。このツールバーには、選択したシンボル／ラインのさまざまなプロパティを制御するためのドロップダウン メニュー ボックスがあります。

- フォント
- フォント サイズ
- シンボル／ラインの背景色

このツールバーには、プロパティの設定を削除して、デフォルト設定に戻すボタンがあります。

シンボルを選択していない場合、ツールバー コマンドは現在のダイアグラムのすべてのシンボルに適用されます（個別にプロパティが設定されているシンボル以外）。

コメントの処理

コメント シンボルはすべてのシンボルに追加できます。

1. ツールバーのコメント シンボルをクリックします。
2. ダイアグラムにシンボルを配置します。
3. コメント シンボルの注釈ラインを、コメント添付先のシンボルに接続します。

コメントと制約

シグニチャシンボルのショートカット コマンド［コメントを表示］（［表示／非表示］のサブメニュー）を使用して、現在のダイアグラム内にコメント シンボルがないシグニチャ シンボルが所有するコメント モデル要素ごとに、1 つのコメント シンボルを作成して追加します。

シグニチャシンボルのショートカット コマンド［制約をシンボルで表示］を使用して、現在のダイアグラム内にシンボルがないシグニチャ シンボルが所有する制約モデル要素ごとに、1 つの制約シンボルを作成して追加します。

備考カラム

2 つ以上の コメント シンボルが近接して垂直またはほぼ垂直に配置されている場合、備考カラム が形成されます。151 ページの図 50 を参照してください。備考カラムが検出されると、垂直位置が自動調整されて、左揃えカラムになります。

Shift キーを押しながら最上部のコメント シンボルを垂直に少し（カラム幅全体を超えない範囲）移動すると、カラムを水平に移動できます。カラム内の別のコメント シンボルを少し移動すると（Shift キーを押しながら）、カラム内の所定の位置に戻されます。コメント シンボルを大きく移動すると、カラムから削除されます。

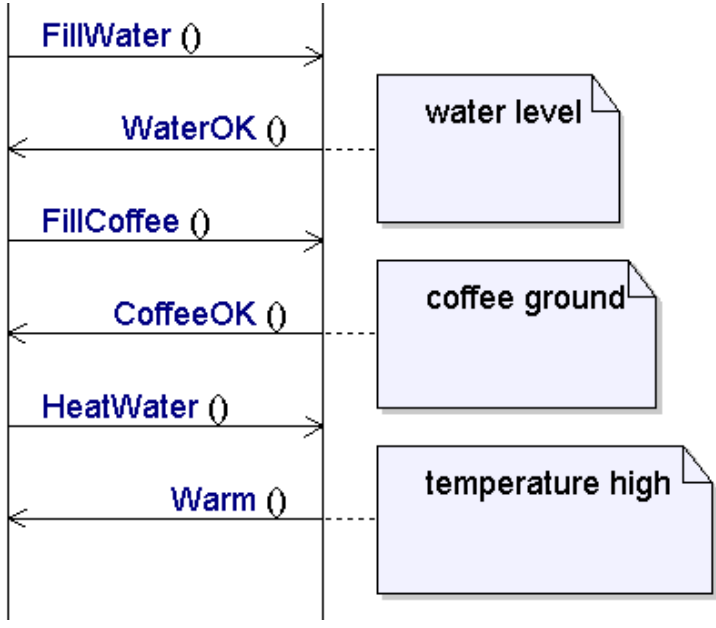


図 50: 備考カラム

注記

備考カラム内の最上部のコメント シンボルは、カラムにインクルードするライフライン ヘッダーの下に配置する必要があります。

シンボルのコピー、切り取り、削除、貼り付け

すべてのシンボルにはショートカット メニューがあります。このメニューは、シンボルを右クリックして表示できます。必要に応じて、このメニューから［切り取り］、［コピー］または［貼り付け］を選択します。

MS Word などの他のツールにシンボルを貼り付けることもできます。

シンボルを削除するには、削除するシンボルを選択して、Delete キーを押します。

注記

シンボルのタイプとモデルとの関係によって、シンボルの削除操作がモデルに影響を与える場合と、与えない場合があります。通常、ダイアグラムにシンボルを追加すると、モデルに情報を追加できます。シンボルとモデルに 1 対 1 の関係がある場合、ダイアグラムのシンボルを削除すると、モデル内の情報だけ削除できます。状態機械のフロー シンボルなどがこれに当てはまります。シンボルとこれに対応するモデル要素を削除するには、[モデルからの削除] を実行します。

アイコン

ユーザー指定アイコン

選択したシンボル アイコンを、画像ファイルを使用してユーザー指定アイコンに置き換えられます。アイコンは以下のレベルで指定できます。

- 特定シンボル
- 特定セマンティック モデル要素。モデル要素に関連づけられるすべてのシンボルが、このアイコンを使用します。
- 特定ステレオタイプ。このシンボルでステレオタイプ化されたモデル要素に関連づけられるすべてのシンボルが、このアイコンを使用します。
- 特定のタイプ（クラスやデータ型など）。このタイプのインスタンスに関連付けられるすべてのシンボルが、このアイコンを使用します。

ステレオタイプの追加

この機能はステレオタイプによって制御されます。この機能を使用するには、アイコンを持つモデル要素を右クリックし、ショートカット メニューから、[ステレオタイプ] を選択します。ダイアログで、ステレオタイプ [`TTDStereotypeDetails::Icon`] を選択します。プロパティ エディタから [ステレオタイプ] ボタンを使用して、ダイアログを開くこともできます。

このステレオタイプが適用できるエンティティは、メタモデル プロパティによって制御されます。この情報を参照するには、[モデルビュー] の [ライブラリ] セクションから、パッケージ [`TTDStereotypeDetails`] を開きます。そのクラス図で、サポートされるエンティティ（メタクラス）とアイコン ステレオタイプの関連を参照できます。

順序付け

上記の選択肢で、複数のユーザー指定アイコンを指定した場合、順序は上記のリストに従います。このため、特定シンボルに指定したアイコンがある場合、まずこのアイコンが使用されます。指定されたアイコンがない場合、モデル要素のアイコンが使用されます。

アイコン モード

ユーザー指定アイコンで識別されるシンボルには、[アイコン モード] と呼ばれるショートカット メニューがあります。このメニューを選択すると、シンボルが通常のシンボルではなく、可視化されたシンボルになります。

画像ファイル

アイコンは、シンボル、モデル要素、または、適用されたステレオタイプのプロパティで定義されており、そのエンティティの [プロパティの編集] ダイアログを使用して変更できます。このダイアログの [フィルタ] ドロップダウン メニューで [アイコン] を選択して、[Icon File] テキスト フィールドを表示します。このフィールドに入力するテキストは、モデル ファイル（.u2）から画像ファイルのある場所への相対パスです。

アイコンに指定できる画像ファイルのフォーマットは以下のとおりです。

- ビットマップ（ファイル拡張子 .bmp）
- JPEG 圧縮画像（ファイル拡張子 .jpeg または .jpg）
- 拡張メタファイル（ファイル拡張子 .emf）
- GIF（ファイル拡張子 .gif）
- TIFF（ファイル拡張子 .tif、.tiff）
- Targa（ファイル拡張子 .tga、.targa）
- PCX（ファイル拡張子 “.pcx）

注記

アイコン画像に白または透明の背景を使用するとダイアグラムの印刷時に背景が黒くなることがあります。

イメージ セレクタ

ダイアグラム内のシンボルは、イメージ セレクタを使用して ユーザー定義の画像として表示できます。この機能は [アドイン] によって提供されます。[ImageSelector] アドインをアクティブにすると、[イメージのロード] と [イメージの削除] コマンドが [ツール] メニューに追加され、使用可能になります。

元に戻す

複数レベルの元に戻す操作とやり直し操作を行うことができます。ツール全体（ワークスペース ウィンドウとエディタ）で一般的な元に戻すスタック機能を利用できます。操作が元に戻されると、この操作はまずやり直しスタックに置かれ、元に戻された操作をやり直すことができるようにします。

注記

元に戻す操作は、現在表示されていないダイアグラムにも実行できます。

テキスト編集モードで元に戻す操作とやり直し操作を行う場合、いくつか考慮しなければならない点があります。元に戻す操作は更新ごとに行えます。更新の際、以下のスキームに従います。一連の文字を追加しても、Back Space キー、Delete キー、矢印キーやマウスを使って選択しない限り、更新されません。同様に Delete キーを連続して押しても、他の操作を行わなければ更新されません。

プロジェクトのファイル／リソースを明示的にアンロード（復帰を含む）すると、元に戻すスタックが空になります。

ファイル システムの操作を元に戻すことはできません。

保存を実行しても、元に戻すスタックが空になりません。

モデル参照

モデル定義とその使用の参照を検索するには、類似の機能を持つ一連のショートカット コマンドで行います。これらのコマンドには、コマンドが適用される要素に依存する、コンテキスト依存があります。

参照の一覧表示

［参照の一覧表示］は、［モデル ビュー］のショートカット コマンドで、すべてのモデル要素に適用されます。このコマンドはダイアログを呼び出して、出力ウィンドウの［参照］タブに参照リストを返します。ここでは以下の設定ができます。

- **... へ行われた参照**
  モデル要素のすべての参照先リストです。たとえば、特定クラスを属性タイプとして使用する方法を確認する場合はクラスを参照します。
- **... から行われた参照**
  セクションからの参照リストです。たとえば、属性からタイプとして利用されているクラスを参照します。
- **内包する階層をレポートに含める**
  このオプションを選択すると、選択した要素に含まれる要素へまたは要素からのあらゆる参照が再帰的に検索されます。たとえば、パッケージ外部の定義、パッケージで使用された定義、およびパッケージに含まれた定義がすべて検索されます。
- **内部参照をレポートに含める**
  このオプションを選択すると、オブジェクトまたは内包する自身への階層からの参照、または内包する階層への参照がレポートされます。たとえば、パッケージ内で行われた参照は検索せずに、パッケージとパッケージのコンテンツの利用法が検索されます。

プレゼンテーションの一覧表示

［プレゼンテーションの一覧表示］は、［モデル ビュー］のショートカット コマンドで、すべてのモデル要素に適用されます。すべてのプレゼンテーション要素のリストを出力ウィンドウの［プレゼンテーション］タブに表示します。

既存要素を参照

これは新しいシンボルを配置するショートカット コマンドです。

ナビゲート

［モデル ビュー］のショートカット コマンドで［モデル ナビゲータ］を表示します。、既存のプレゼンテーションがない場合は、［プレゼンテーションの作成］ダイアログを表示します。

［モデル ビュー］で複数のノードを選択した場合、［参照の一覧表示］コマンドと［プレゼンテーションの一覧表示］コマンドは選択したすべての要素に適用されます。

参照

第 3 章「ダイアグラムの操作」の 144 ページ、「シンボルの追加」

モデルの更新

［ActiveModeler］アドインを有効にすると、ショートカット メニューに新しいメニュー［モデルの更新］が追加されます。このコマンドをバインドされていないエンティティに使用し、現在のモデルに呼び出せます。

この機能はコンポジッド ストラクチャ図、ユース ケース図とシーケンス図で利用できます。

利用法

プレゼンテーション要素からモデルを更新するには、プレゼンテーション要素を選択して、ショートカット メニューから［モデルの更新］を選択します。このコマンドは、現在選択されている要素にモデルの更新手順が登録されている場合のみ利用できます。モデルの［モデルの更新］は［ツール］メニューにもあります。

出力ウィンドウの［ActiveModeler］タブにはコマンド実行中の変更に関するさまざまな情報が表示されます。これらの変更をナビゲートするには、タブの列をダブルクリックするか、F4 キーと Shift + F4 キーを押してリスト内を移動します。

他のコマンド同様に、［モデルの更新］コマンドの実行結果を元に戻すことができます。モデルに複数の変更が加えられ、すべての変更がそのコマンドで実行される場合も、コマンド全体がひとつのアクションとして見なされ、一度に元に戻されます。

ネストされたシンボル

他のシンボル内に配置できるシンボルもあります。他のシンボル内にシンボルを作成すると、作成のコンテキストとして親シンボルのモデル要素が使用されます。

親シンボルに自動サイズ調整が設定されている場合、作成したシンボルに合わせて親シンボルのサイズが調整されます。設定されていない場合は、親シンボルの境界線内に納まるように新しいシンボルのサイズが調整されます。ネストされたシンボルは、親シンボルの境界線の外にドラッグすることはできません。

区画をもつシンボル

区画をもつシンボルには、その区画と区画に含まれるテキストフィールドに関連する特殊な機能があります。

区画にはテキストフィールドが含まれます。そのテキストフィールドはモデル要素と関連付けられます。区画のテキストフィールドは左揃えです。

クラスシンボルのような特定のシンボルは、デフォルトの区画セットを使って作成できます。たとえば、クラスシンボルに、属性と操作の区画を持たせられます。

区画は選択可能です。また区画上で実行できる一連の操作があります。

マウスでカーソルを区画上に移動すると、ツールチップが表示されて区画のタイプを示します。

サイズ変更

区画付きのシンボルのサイズを変更すると、シンボルの大きさに適合しない区画は表示されなくなります。サイズによっては区画全体ではなく区画の一部が非表示になります。

シンボルを区画をすべて表示できる大きさ以上に大きくすると、余白分は均等に各区画に割り当てられます。

区画の作成

区画を持つことができるシンボルには、シンボルのショートカットメニューに［区画］メニューがあります。このメニューのサブメニューには、区画を作成するための一連の操作があります。これらの操作の 1 つを実行すると、区画が作成されます。新しく作成した区画はシンボルの下部に追加されます。

区画の削除

区画は、区画を選択して通常の［削除］コマンドで削除できます。一部の区画はその区画が表示しているモデル要素と直接的に関連付けられます。この場合は、［モデルの削除］コマンドを使って削除することもできます。

区画の移動

区画の順番は、［移動］ツールバーの［上に移動］、［下に移動］コマンドを使って変更できます。

区画上での表示 / 非表示

特定の区画を選択した場合、ショートカットメニューに区画の用途である要素タイプの表示、非表示を選択するメニューが現れます。表示、非表示の操作はそれぞれ適用できる場合にのみ表示されます。

シンボルを選択すると、任意の既存の区画について要素の表示、非表示を選択するメニュー、または特定のタイプのモデル要素用の新しい区画を作成するメニューが表示されます。これらの操作は適用可能な場合にのみ表示されます。複数の区画が同じタイプのモデル要素を表示している場合、表示、非表示の操作は初めの要素に対してのみ実行できます。特定の区画で要素を表示、非表示するには、その区画でショートカットメニューを表示してください。

作成済みの要素をダイアグラムにドラッグした場合、デフォルトではその要素が所有するモデル要素は表示されないことに注意してください。

参照

デフォルトのクラスシンボルの外観

要素を可視状態にするには、その要素を区画やシンボルにドラッグアンドドロップするか、手動でタイプインします。

ヒント

名前の完成を使用すると、新たなフィーチャを誤って作成することを回避できます。名前をタイプインして、Ctrl+ スペースキー、または Shift+ スペースキーを押すと、複数の候補がある場合はリスト表示されます。

区画のテキストフィールド

要素の削除

区画内のテキストフィールドは、あるモデル要素と関連付けられた別個の表現要素です。このため、モデル要素と結びついたテキストフィールドについては、区画内のテキストを削除できません。ラベルと関連付けられている要素を削除するには、テキストモードに入り、ショートカットメニューから［削除 < 要素 >］コマンドを使用します。

注記

モデル要素と結びついたテキストフィールドを、区画内のテキストの削除によって削除することはできません。削除されるのはその場所にある文字だけです。その行はモデル要素と結びついたままです。

燃える要素と結びついていないテキストフィールドはテキストを削除キーや後退キーで削除することで削除できます。空のテキストフィールドは、カーソルを先頭において後退キーを押すか、末尾において削除キーを押すと削除できます。

要素を非表示

区画のテキストフィールドに表示されている特定の要素を非表示にするには、テキスト編集モードに入って、ショートカットメニューから［非表示 < 要素 >］操作を実行します。

テキストフィールドの移動

テキストフィールドは、［移動］ツールバーの［上に移動］、［下に移動］コマンドを使って上下に移動できます。

共通のライン操作

ラインのスタイル

ラインに適用できるスタイルには次の 4 種類があります。ラインの作成時に、コンテキストメニューから選択できます。

自動経路選択

ラインは障害物を避けるように自動的に経路を選択します。そのための可能な経路がある場合は直角に曲がります。通常は直線です。

直交

ラインは常に直角で曲がり、頂点とラインセグメントは移動できます。頂点をラインに追加することやラインから削除できます。

非直交

ラインの頂点の移動、追加、削除が自由にできます。

ベジェ

カーブしたラインを使用できます。ラインが選択されると、2 つの制御点が表示されて、カーブを調節できます。

参照

2161 ページの「［UML ラインスタイル編集］タブ」

ラインの描画

ツールバーボタンを使用するか、ラインを表現するラインハンドルを使用して、ラインを作成できます。

ラインハンドルを使用してラインを作成する

1. ソースシンボルを選択します。
2. ラインハンドルをクリックします。
3. 頂点を追加します。オプションで終端点をロックできます。
4. ターゲットシンボルまたはラインをクリックします。

ツールバーボタンを使用してラインを作成する

1. ツールバーボタンをクリックします。
2. ソースシンボルをクリックします。
3. 頂点を追加します。オプションで終端点をロックできます。
4. ターゲットシンボルまたはラインをクリックします。

自動経路ラインを除いて、ラインの作成中に頂点を追加できます。頂点の追加ができるときは、カーソルの形が＋マークになります。

ラインの開始点が選択可能な場合は、シンボルエッジ上の開始点の位置を固定できます。ラインをで自動経路選択で作成した場合は、カーソルが「錠前」の形になります。この状態でクリックすると、ラインの開始点は現在の位置に固定されます。自動経路選択以外のラインスタイルで作成した場合は、Shift キーを押しながらクリックすることで開始店を固定できます。

頂点の編集

既存のラインに頂点を追加するには、頂点を作成したいセグメントの上で Ctrl キーを押しながらクリックします。

頂点を削除するには、削除したい頂点上で Ctrl キーを押しながらクリックします。

この操作は直交 または 非直交 のスタイルのラインでのみ可能です。操作が可能な場合は、マウスカーソルの形が変化します。

参照

シンボルの接続

ラインの移動

ラインを移動するには、ラインのいずれかの端点をクリックして、目的の位置までドラッグします。

ラインの削除

ラインは、通常、ダイアグラムから削除してもモデル内に残るモデル要素を表示します。ライン（たとえば関連ライン）を完全に削除したい場合、必ず［モデルからの削除］を実行します。

ラインの方向変更と双方向化

- 描画したラインの方向を変更する場合 （可能な場合)、ラインをクリックして、ショートカット メニューから［逆向き］を選択します。
- ラインを双方向にする場合（可能な場合)、方向やシグナル リストが表示されない端点に近いライン上でラインを右クリックします。ショートカット メニューから［この方向を有効にする］を選択します。
- 双方向のラインでどちらか一方向のみ有効にしたい場合、無効にする端点に近いライン上にカーソルを合わせます。ショートカット メニューから［この方向を有効にする］の選択を解除します。

  この操作の前後で、ラインの方向を目的の方向に変更することもできます。

4

UML 言語ガイド

この章では、Tau 4.3 で実装、サポートしている UML 言語について説明します。

- サポートする UML のバージョンについては、UML のバージョンを参照してください。

参照

モデルの操作

ダイアグラムの操作

ワークフローの説明

概要

UML は、ソフトウェアやシステムの仕様決定、可視化、文書化、ビルドなどに使用できるモデリング言語です。以下のセクションでは、さまざまな抽象化レベルでのシステムの構造や振る舞いを表現するために使用できる、各種ダイアグラムや構成要素について説明します。構成要素には、要求や分析など開発の初期フェーズで有用なものと、設計、実装、テストなど開発の後期フェーズで役立つものがあります。このように、さまざまな開発フェーズを連結してシステムを記述できる能力は、UML の大きな長所の 1 つです。

UML のバージョン

Tau で使用されている言語は、最新の OMG UML 2.1 Superstructure サブミッションです。Tau の実装が言語仕様と異なっている場合もあります。これは、主としてツールの最適化、もしくは、サブミッションの旧バージョンに基づいた設計があるためです。

また、Tau では、UML に対して定義されたグラフィック表記とテキスト構文を併用できる点など、言語を一部拡張しています。

ダイアグラム

UML は、システムに対するさまざまな視点を表現するために使う一連のダイアグラムで構成されています。システムの構造に重点を置いたダイアグラムもあれば、エンティティ間の相互作用や、特定条件下で実行される一連のアクションなど、システムの振る舞いの表現のみに使用されるダイアグラムもあります。通常これらのダイアグラムは、システムの仕様を定める主要な手段となります。

Tau でサポートされるダイアグラムを以下に示します。

ダイアグラム	目的
ユース ケース図	一連のアクターがユース ケースの観点から、どのように相互作用するのかを表します。通常はサブジェクト、つまり記述されているシステムの文脈で考えます。
シーケンス図	ユース ケースまたは操作のイベント シーケンスを表します。
パッケージ図	パッケージとパッケージ間の依存関係を表します。
クラス図	クラスとクラス間の関係を、通常はパッケージその他の（コンテナ）クラスのスコープで宣言するダイアグラムです。
合成構造図	（コンテナ）クラスの各パート間の接続関係を表し、コンテナの内部構造を示すダイアグラムです。

ダイアグラム	目的
アクティビティ図	並列と相互に関わり合う振る舞いを表現します。複雑な構造を単純化して表示でき、制御の特定のフローに焦点を当てることができます。
相互作用概観図	並列な振る舞いを記述します。ユース ケースを記述するためによく使用されます。
コンポーネント図	コンポーネントの設計に焦点を当て、コンポーネント間の関係と構造を表現します。
配置図	物理的な実装の構造とソフトウェアとハードウェアの関係を表現します。
状態機械図	クラス、状態機械、操作の振る舞いを記述します。
テキスト図	図形でなくテキストでエンティティを定義するダイアグラムです。

モデルとダイアグラム

モデルとは物理システムを表現したもので、通常は、1 つまたは複数のパッケージに含まれたエンティティによって定義されます。

Tau では、1 つのプロジェクト内にあるものはすべて 1 つの同じモデルに属します。

モデルはシステムの表現であるため、その表現を詳細化することが必要になる場合もあります。たとえば、自動アプリケーション生成の情報源として使用する場合は、要求の可視化として使用する場合よりも、アルゴリズム レベルで詳細に表現する必要があります。

モデルは、ダイアグラムやモデル要素など、システムを表すのに必要なすべてのエンティティを含みます。モデル、厳密に言えばモデルに含まれるモデル要素は、通常、さまざまなダイアグラムでシンボル（モデル要素に対し、プレゼンテーション要素と呼ぶこともあります）を使用して表示されます。

モデル要素

モデルの主な内容は、クラス、属性、操作、アクション、制約などのモデル要素です。モデル要素は、エンティティのすべての特性を格納するために使用します。この仕組みで、ダイアグラムのモデル要素はさまざまな側面を示すことができます。たとえば、1 つのクラス図でクラスの属性や操作を示し、別のクラス図で、定義されているクラス階層を示すことができます。これらのダイアグラムでは、同じモデル要素の異なる側面を表示していることになります。

シンボル

シンボルは、モデル要素（のパート）を描画された図形として可視化するものです。各シンボルは、ダイアグラムに示される二次元オブジェクトです。シンボルは、サイズと、ダイアグラムの座標系における位置を示すようになっています。

クラス シンボルなど、ほとんどのシンボルは、対応するモデル要素を直接的に可視化したものですが、テキスト シンボルなどのように、基礎となるモデル要素がないものもあります。シンボルは、特定のダイアグラムのみに関連付けられます。

モデル要素とシンボルの区別は重要ですが、日常の言葉では、2 つの区別は曖昧になる場合が多いと思います。クラス モデル要素やクラス シンボルは、多くの場合、単にクラスと呼ばれます。

モデル要素のさまざまなビュー

164 ページの図 51 は、3 つの異なるビューを使用して表した、モデル要素 a の例です。1 つ目のビューでは、このモデル要素はクラス C の属性として表されています。2 つ目では、クラス C とクラス D の関連の端として示されています。3 つ目では、クラス C の内部構造の一部として示されています。

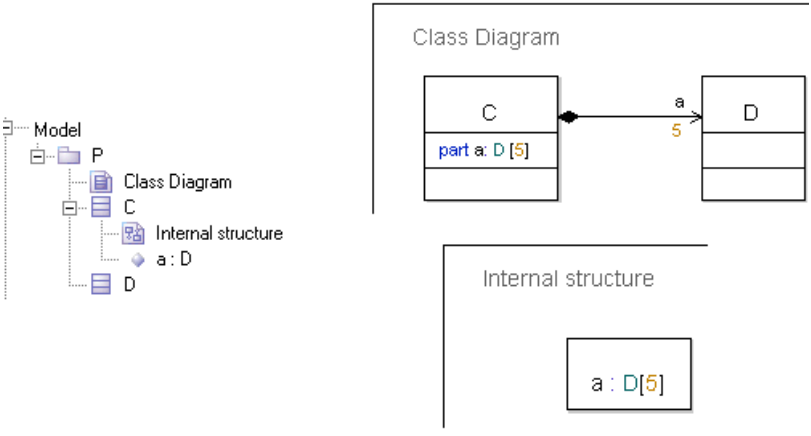


図 51: 属性のさまざまなビューの例

また、「モデルの削除」（モデルからの削除）と「ダイアグラムからの削除」の違いも重要です。左側のブラウザ ビューにはモデルが示され、ブラウザ ビューから要素を削除すると、要素は、モデルからだけでなく、モデルが表示されているダイアグラムからも削除されます。右側の 2 つのダイアグラム ビューでは、同じ属性 a が 3 通りの方法で表示されています。1 つ目はクラス C の属性入力領域に、2 つ目はクラス C とクラス D の関連の端として、3 つ目はクラス C の内部構造の一部として表示されています。

シンボルとモデル要素の削除

要素をダイアグラムから削除する方法は 2 つあります。通常の削除では、シンボルは削除されますが、モデル要素はモデルに残ります。モデルからの削除では、要素はモデルからだけでなく、モデルが表示されている他のすべてのダイアグラムからも削除されます。

一部のダイアグラムでは、モデル要素とシンボルが密接に接続されています。このようなダイアグラムには、テキスト図、シーケンス図、状態機械図があります。これらのダイアグラムでは、シンボルとモデル要素は 1 対 1 のマッピング関係にあり、一方が削除されると、もう一方も削除されます。（つまり、これらのダイアグラムでの削除はモデルからの削除と同じです。）これは、たとえば、アクションや遷移に当てはまりますが、ステートには当てはまりません。

参照

シンボルの追加

シンボルの移動

シンボルのサイズ変更

シンボルの接続

シンボルのテキスト フィールドの編集

シンボルのコピー、切り取り、削除、貼り付け

言語構成要素一覧

次の表に、すべての具象モデル要素、その他 UML の重要な言語構成要素を示します。

UML モデル要素
イベント受信 , タイム イベント受信 , アクセス , アクション（操作本体、状態機械、状態機械図）, アクション （相互作用図とシーケンス図）, アクション ノード（アクティビティ図）, アクティブ クラス , アクティビティ , アクティビティ終了 , アクター , 集約 , 任意値（any）式 , アーティファクト , 代入 , 関連 , 属性
振る舞いポート
選択 , クラス , 分類子 , コメント , コンポーネント , 合成状態 , 合成 , 複合文 , 条件式 , 定数 , コネクタ（合成構造図）, コネクタ（アクティビティ図）, 継続 , 共通リージョン , 生成
データ型 , 分岐（状態機械図）, 分岐（アクティビティ図）, 依存 , デプロイメント , デプロイメント スペシフィケーション , ダイアグラム , 消滅
エントリ接続ポイント , 実行環境 , 終了接続ポイント , 式 , 拡張
フィールド式 , フロー終了 , フォーク
汎化 , ガード
履歴の次のステート

UML モデル要素
命令式 , 実現化 , インポート , インデックス式 , 開始ノード , インライン フレーム , 相互作用 , 相互作用参照 , インターフェイス , インターナル
ジョイン , ジャンクション
ライフライン（生存線）リテラル
表現 , マージ , メッセージ , メソッド , メソッド呼び出し
New, 次のステート , ノード , now 式
オブジェクト ノード , offspring, 操作 , 操作本体 , シグナル送信アクション（出力）
パッケージ , Parent, パート , アクティビティ区画 , Pid 式 , ピン , ポート , 定義済み , プロファイル
範囲チェック式 , 実現化インターフェイス , 要求インターフェイス , リターン
保存 , Self, シグナル送信 , Sender, シグナル , シグナル リスト , シグニチャ , 開始遷移 , ステート , 状態機械 , 状態機械実装 , ステート式 , ステレオタイプ , 停止 , サブジェクト , シンタイプ
タグ定義 , タグ付き値 , ターゲット コード式 , アクション（タスク） , this 式 , タイマー, タイマー アクティブ式 , タイマー リセット , タイマー リセット アクション , タイマー設定 , タイマー設定アクション , タイマー タイムアウト , 遷移
ユース ケース

スコープ、モデル要素、ダイアグラム

パッケージやクラスなど、一部のモデル要素は名前スコープを表現できます。つまり、これらのモデル要素には、他のモデル要素の定義を含めることができます。名前スコープ内のすべての定義には、一意の名前を付ける必要があります。一意の名前を付けないと、セマンティック チェッカーから注意を促されます。スコープは、グループを構成するモデル要素の「コンテナ」または「グルーピング」と考えるとよいでしょう。

ほとんどのスコープには、モデル要素だけでなく、モデル要素が表示されているダイアグラムも含まれます。次の表に、各スコープに使用できるダイアグラムを示します。

スコープ単位	使用できるモデル要素	ダイアグラム
パッケージ	パッケージ, クラス, ユース ケース, アーティファクト, ステレオタイプ, 関連, データ型, インターフェイス, シンタイプ, 選択, 操作, 属性, シグナル, シグナル リスト, タイマー, 状態機械	クラス図 シーケンス図 テキスト図 ユース ケース図
クラス	クラス, アーティファクト, ステレオタイプ, データ型, インターフェイス, シンタイプ, 選択, シグナル, シグナル リスト, タイマー, 属性, 操作, ユース ケース, 状態機械,	クラス図 合成構造図 テキスト図
ユース ケース	相互作用, 状態機械実装, アーティファクト, 操作本体	シーケンス図 状態機械図 テキスト図
相互作用	ライフライン（生存線）	シーケンス図 ユース ケース図
ステレオタイプ	属性	
データ型	リテラル, 操作	
選択	属性, 操作,	
インターフェイス	シグナル, タイマー, 属性, 操作	
操作	操作本体, 状態機械実装, 相互作用	
操作本体	状態機械, クラス, アーティファクト, ステレオタイプ, データ型, インターフェイス, シンタイプ, シグナル, シグナル リスト, タイマー, 操作, 属性	状態機械図 テキスト図
状態機械実装	クラス, アーティファクト, ステレオタイプ, データ型, インターフェイス, シンタイプ, シグナル, シグナル リスト, タイマー, 操作, ステート, アクション, 属性	状態機械図 クラス図 ユース ケース図 テキスト図
複合文	アクション, 属性	

オーバーロードされた定義

特定の種類の定義は、スコープ内で同一の名前を複数持つことができます。これは、操作、シグナル、タイマー および 状態機械などの振る舞い特性にあてはまります。これらの定義は、名前だけではなくパラメータの型も使って識別されます。操作の名称とパラメータ型の一覧を、振る舞い特性の「シグニチャ」と呼びます。同じスコープ

内のすべての振る舞い特性は、一意のシグニチャをもつ必要があります。同一スコープ内で同じ名前を持ち、シグニチャが異なる 2 つの振る舞い特性は、「オーバーロードされている」といいます。

一般的な言語構成要素

UML には、複数のダイアグラムに共通する言語構成要素がいくつかあります。

名前

すべての UML モデル要素には、名前、すなわち識別子があります。名前には、一定のルールがあります。

命名ルール

名前には、文字、数字、_（アンダースコア）が使用できます。

名前の先頭は数字でなく、文字またはアンダースコアでなければなりません。また、特殊な場合として、先頭が常に ~（チルダ）になるデストラクタ名もあります。

識別子でのスペースと特殊文字の使用

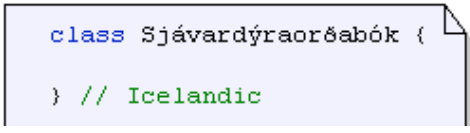


図 52: 識別子での特殊文字の使用

名前を単一引用符で囲めば、上記の制限を取り除くことができるので、名前の一部として（ほとんどの）任意の文字を使用できます（168 ページの図 52 を参照）。たとえば、名前を単一引用符で囲めば、名前の一部としてスペースを使用できます（169 ページの例 12 を参照）。

文字列の処理に使用できる、いくつかのエスケープ文字があります。¥n、¥t、¥b、¥r、¥f は、charstring 内（“” 内）に入れるか、文字（'¥n' など）として使用できます。

「¥"」は charstring 内で使用でき、「¥'」は文字として使用できます。「¥¥」は charstring 内または文字列として使用でき、円記号を表します。単一引用符で囲まれたその他のエスケープ文字（'¥+'、'¥s'）は、識別子（+、s）と解釈されます。二重引用符で囲まれている文字列内で円記号に続く文字は、その文字自体を表します（“a¥qa” は” aqa” を表す）。

```
¥n: new line
¥t: tab
¥b: backspace
¥r: carriage return
¥f: form feed
¥": quotation mark, e.g. "my ¥"quoted¥" word"
```

```
¥': apostrophe character, '¥''
¥¥: yen mark
```

例 **12:** 識別子でのスペースの使用

```
Boolean 'has finished'=false;
```

大文字と小文字の区別

識別子では、大文字と小文字が区別されます。つまり、違いが大文字か小文字かだけである名前も、異なるものとして認識されます。

例 **13:** 大文字と小文字の区別

```
Integer MyInt, myint; // Two distinct attributes
```

参照

名前付きの定義は、モデル内の他の場所から参照されることがあります。単純なケースとして、参照が定義の名前から構成される場合があります (必要に応じて、単一引用符で囲みます)。通常は、参照はさらに複雑になります。

- 参照は修飾子を含むことがあります。

スコープをまたがって同じ名前を持つ定義を区別するためには、名前に修飾子をつけます。修飾子とは、スコープパスと特殊なスコープ解決演算子 「::」から成る識別子の接頭辞です。グローバル名はパスがありませんので、「::」が先頭に置かれます。「::」で始まる修飾子を絶対修飾子と呼び、その他のものを相対修飾子と呼びます。

- 参照は実テンプレート引数を含むことがあります。

参照定義がテンプレートである場合 (つまり、参照定義が テンプレート パラメータ をもつ場合) は、参照は、テンプレートパラメータの実の値を含む必要があります。実テンプレート引数は、「<」「‘>」ブラケット内の名前に続くカンマ区切りのリストで与えます。

- 参照はパラメータ型のリストを含むことがあります。

振る舞い特性を参照する場合は、パラメータ型の名前を追加する必要があります。これは、名前とともにパラメータの型が振る舞い特性のシグニチャの一部となるからです。パラメータ型の名前は、名前の後の括弧で囲んで与えます。

例 **14:** 異なる種類の参照

この例では、2 つの属性が修飾名を使って型を参照しています。

```
::Predefined::Integer i;

UtilityTypes::Sorts::ClientIdx j;
```

型がテンプレートクラスの場合は、実テンプレート引数を指定する必要があります。

```
MyClass<Integer, 4> k;
```

操作のような振る舞い特性を参照する際は、パラメータ型を指定する必要があります。このような参照の場合は、通常の操作の呼び出しと区別して文法的な曖昧さを排除するために、キーワード「operation」を使っていることにも注意してください。

```
OperationReference r = operation foo(Integer, Boolean);
```

予約語

Tau に予約されており、モデル要素の名前として直接使用できない名前があります。名前の一覧は「UML 2 テキスト構文」を参照してください。

これらの単語を単一引用符で囲んで定義の名前に使用できますが、混乱の原因になるため、どうしても必要な場合以外は、使用しないでください。

例 **15:** 予約語を単一引用符で囲んだ使用方法

```
Integer ‘class’; // confusing attribute name, but valid
```

参照

アプリケーションまたはモデル ベリファイヤ（Model Verifier）の生成時に使用できる予約語の詳細な一覧については、予約語を参照してください。

代替構文

UML で定義されているグラフィック表記に加えて、通常の「テキスト」でモデルを記述するための補足テキスト構文が定義されています。この記法は、グラフィック シンボルの代わりに使用することも、またはグラフィック シンボルと併用することもできます。

テキストは、テキスト図またはダイアグラムのテキスト シンボル中に表示されます。

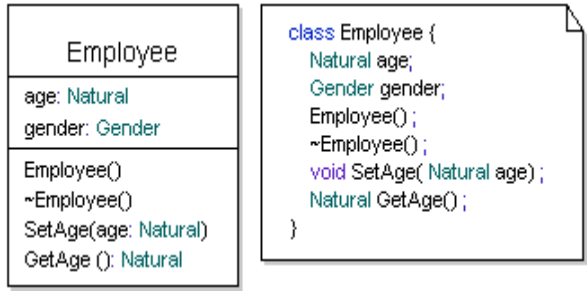


図 53: クラス シンボルの構文とテキスト構文の違いの例

171 ページの図 53 では、同じモデル要素が、1 つのダイアグラム中で 2 度表示されています。左側ではグラフィック表記が使用され、右側ではテキスト シンボル中にテキスト構文が使用されています。これらのビューのいずれかに変更を加えると、自動的にもう一方に反映されます。

共通の要素プロパティ

以下のプロパティは、さまざまなモデル要素に使用されます。これらのプロパティは、プロパティ エディタで表示して制御できます。

Visibility（可視性）

多くのモデル要素には可視性があります。可視性を使用して、要素に対する、要素が定義されているスコープ外からのアクセス権限を決定します。スコープ内では、可視性に関わらず、すべての要素にアクセスできます。可視性には、以下に示す複数のレベルがあります。

- **Public**
  要素が public 可視性を持っている場合、そのコンテナを見ること（アクセス）が可能なすべての要素が、その要素にもアクセスできます。
- **Protected**
  要素が protected 可視性を持っている場合、そのコンテナのサブクラス、および要素と同じスコープ内のすべての要素が、その要素にアクセスできます。
- **Private**
  要素が private 可視性を持っている場合、同じスコープ内の要素のみが、その要素にアクセスできます。
- **Package**
  要素が package 可視性を持っている場合、同じパッケージ内のすべての要素がその要素にアクセスできます。

- **なし**
  可視性が指定されていない場合は、要素は以下の表に従ってデフォルトの可視性が設定されます。

定義のデフォルトの可視性は、スコープとタイプによって異なります。

スコープ	可視性
Class, Choice, Stereotype, Collaboration, Artifact	Private
Package	Public
Interface	Public
DataType	Public

注記

リテラルは必ず public 可視性を持ちます。
データ型のすべてのリテラルと public 静的メンバーは、修飾子のないデータ型の外側でも見えます。修飾子は曖昧性を解決するためにのみ必要です。たとえば、同一スコープ内の 2 つのデータ型が同じ名前のリテラルを持つ場合などです。

Virtuality（仮想性）

仮想性は、クラスなど分類子を汎化し、特化クラスの内部モデル要素が再定義可能かどうかを決定する場合に必要になります。

仮想性は、タイプ内の要素（特化可能な分類子）にのみ適用されます。コンテナが特化されている場合、各内部要素の個々の仮想性によって、その要素が変更可能かどうかが決まります。

- **Virtual**
  内部要素が仮想（Virtual）の場合、コンテナを特化して要素を再定義（変更）できます。
- **Redefined**
  特化コンテナ内の要素の仮想性が再定義（Redefined）の場合、基底コンテナの元の要素の定義は変更されています。基底コンテナの元の要素は仮想でなければなりません。

  再定義された要素もまだ仮想です。すなわち、コンテナが再度特化されれば、要素をさらに再定義（変更）できます。
- **Finalized**
  特化コンテナ内の要素の仮想性が最終（Finalized）の場合、基底コンテナの元の要素の定義は変更されています。基底コンテナの元の要素は仮想でなければなりません。最終であるということは、コンテナが再度特化されても、この要素をさらに再定義することは不可能であるということです。つまり、仮想性が最終であるということは、再定義されているがもはや仮想はでないという意味になります。

- **なし**
  内部要素に仮想性がない場合、コンテナが特化されると、要素を再定義（変更）できません。

Derived（導出）

要素が導出されている場合は、他の要素を使用することでその値を計算できます。計算法の指定方法はコンテキストによって決まります。

導出要素の典型的な使用法は、導出属性です。属性にアクセスする際に使用する導出規則を、アクセス演算子「get」や「set」を使用して指定できます。

例 **16: S** 導出属性のための導出規則の指定

```
Integer y;
Integer / x
  get { return 5; }
  set { y = value; };
```

他のプロパティ

- **External**
  定義が外部であるとは、定義がこのモデルの外部にあることを意味します。付属のコード ジェネレータは、外部要素に対してはコードを生成しません。したがって、外部要素は、外部で利用可能な定義のモデル表現と見なすことができます。
- **Abstract**
  分類子が抽象である場合、この分類子を直接インスタンス化できません。通常は、この抽象分類子は別の分類子によって特化されています。その場合は、特化分類子をインスタンス化できます。
- **Static**
  定義が静的であれば、包含する分類子のすべてのインスタンスがこの要素の実装を共有します。つまり、同じデータを使用します。したがって、静的な定義は、定義されている分類子のインスタンスがなくても使用できます。

パラメータ

操作、シグナル、状態機械などの振る舞い特性を表す定義は、パラメータを持つことができます。一般的な形式（分類子シンボルとプロパティ エディタで使用される）は次のとおりです。

```
name:type, name2: type2
```

パラメータにより、呼び出しから振る舞いへとデータが流れる（フローする）方向を指定できます。

- **In**（デフォルト）
  データは、呼び出し側から呼び出される振る舞いに渡されます。

- **In/Out**
  データは、呼び出し側から呼び出される振る舞いに渡され、さらに、呼び出された振る舞いから呼び出し側へ戻されます。
- **Out**
  データは、呼び出された振る舞いから呼び出し側へ戻されます。
- **Return**
  データは、呼び出した振る舞いから呼び出し側に、呼び出しの結果の戻り値として渡されます。戻り値パラメータとしては 1 つのみが許されます。

テンプレート パラメータ

テンプレート パラメータとは、使用する文脈に依存しない定義を行って、より動的で柔軟な方法で分類子を使用するための概念です。テンプレート パラメータは、**コンテキスト パラメータ**とも呼ばれます。

クラスや操作など、特化やインスタンス化が可能な要素には、テンプレート パラメータを持たせることができます。

テンプレート パラメータは、インスタンス化、または、そのテンプレートパラメータをもつ分類子が特化や再定義される際に、実際のパラメータ「値」と結び付けられます。特化の際には、テンプレート パラメータのサブセットを値と結び付けることもできます。インスタンス化の際は、すべてのテンプレート パラメータを値と結び付ける必要があります。

原則として、テンプレート定義が参照される場合にはテンプレートパラメータについて実際の値が指定されている必要があります。ただし、以下の 2 つの例外があります。

1. テンプレートパラメータがデフォルト値をもつ場合、実際の値を与える必要はありません。この場合はデフォルト値が使用されます。
2. テンプレートパラメータを使用した振る舞い特性の呼び出しにおいて、呼び出しで使用されている実際の呼び出し引数でそのテンプレートパラメータを引数としていない場合は、実際の値を指定する必要はありません。

演算子 `reinterpret_cast<T>` と `cast<T>` は、実際のテンプレート パラメータとしては使用できません。`reinterpret_cast<T>` または `cast<T>` 演算子を含むインスタンス化テンプレートは、名前解決では解決できません。

例 **17:** キャスト演算子を含むインスタンス化テンプレート ——————————

この制限を以下の例に示します。

```
template<const Integer x>
class MyTemplate { }
enum E { L }

/* These template instantations cannot be resolved */
MyTemplate<cast<Integer>(L)> myVar1;
MyTemplate<reinterpret_cast<Integer>(L)> myVar1;
```

——————————————————————————————————————

定義済みの名前

付属のユーティリティ パッケージ `Predefined` には、役に立つデータ型、リテラル値、演算があらかじめ定義されています。このエンティティの名前は特に予約されてはいませんが、誤解による予期せぬ障害を招く可能性があるため、この名前を他のエンティティには使用しないことを推奨します。

参照

定義済み

ユース ケース モデリング

ユース ケース モデリングでは、主として、システムやそのシステムの一部分の「使われ方」を、各要素の振る舞いに対する要求に基づき、また、相互作用対象のアクターに関連付けて決定することに焦点を当てます。

ユース ケース図

ユース ケース図は、ユース ケースとアクターの関係を示すことにより、システムの使用の状況を説明するものです。ユース ケース図により、システムの動的側面が静的に表示されます。

例

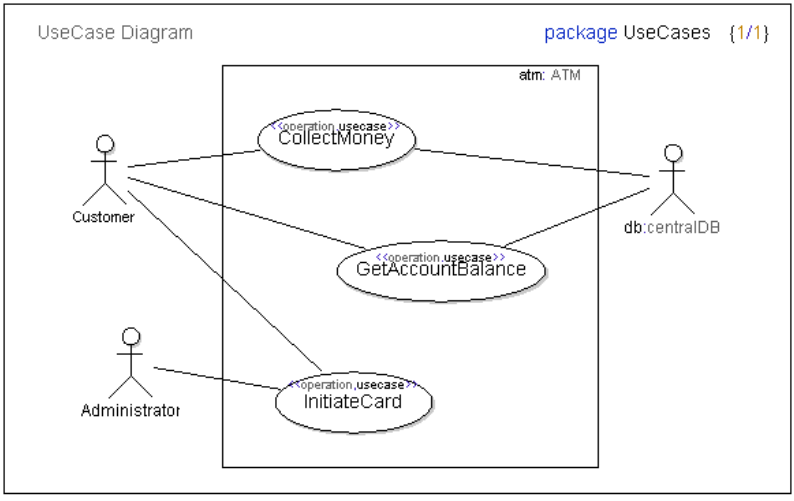


図 54: アクター、ユース ケース、サブジェクト、アクターとユース ケースの関連を示したユース ケース図

ユース ケース図のモデル要素

ユース ケース図には、以下の要素が使用されます。

- ユース ケース
- アクター
- サブジェクト
- 依存
- インクルード
- 拡張
- 汎化
- 関連

ユース ケース図の作成

ユース ケース図は、パッケージ、クラス、コラボレーションに含めることができます。

1. ［モデル ビュー］でパッケージ（クラス、コラボレーション）を選択します。
2. ショートカット メニューから［新規］、次に、［ユースケース図］を選択します。

ツールバーを使ってユース ケース図を描画したり、モデルからユース ケースをドラッグしてユース ケース図に入れることもできます。

- ツールバーを使用するには、まずユース ケース シンボルをクリックしてから、ダイアグラム内のユース ケース シンボルの配置場所をクリックします。

ユース ケース

ユース ケースは、システムまたはシステムのパートの機能の、一貫した単位を表します。通常、システムはクラスで表します。機能は、多くの場合、システムの振る舞いなど、システムと 1 つまたは複数の外のアクターとのやりとりに関連して表現されます。

ユース ケースは多くの点で操作と類似しており、ステレオタイプが <<use case>> の操作としてモデリングされます。

シンボル

図 55: ユース ケース シンボル

ユース ケース図では、ユース ケースは、ユース ケース シンボルを使用して可視化されます。ユース ケースは、以下のスコープ内で指定できます。

- パッケージ
- クラス
- コラボレーション

ユース ケースの記述

ユース ケースの振る舞いは、以下を使用して定義します。

- 相互作用
- 状態機械
- 操作本体

テキストを使用してユース ケースを記述できます。この場合、通常は、テキストに一定の構造をもたせることになります。ユース ケース名、その目的、事前条件、事後条件、例外的な場合、ユース ケースが実行するアクションの形で記述された実際の機能などを、整理して記述するためです。

例 **18:** テキストで記述されたユース ケース

```
Use case:CloseAccount
Goal:Close a user account and make sure the balance of the
account is settled
Preconditions:Customer has an open account
Postconditions:Customer has closed the account and has paid
outstanding dues
Description:
1. Check balance of account
2.a If balance is positive, pay customer
2.b If balance is negative, collect payment from customer
3. Terminate card associated with account
4. Close account
```

ユース ケースの命名

ユース ケースの名前には、述語を使うのが一般的です。つまり通常は、「do something」などのように、「動詞と目的語」を含んだフレーズを使用します。引用符で囲んだ名前を使用すれば、名前に空白を含めることもできます。

例 **19:** 引用符で囲んだユース ケース名

```
<<usecase>> void 'Open Account' ();
```

動詞と名詞の間に空白を入れない記法もよく使われます。

アクター

アクターは、機能の起動やユース ケースの情報源といった形でユース ケースに関与するエンティティを表現します。

シンボル

図 56: アクター シンボル

アクターは、ユース ケース図で、棒線画を使用して表されます。アクターは、関連を使用してユース ケースと接続されます。

アクターのロール

ユース ケース図では、アクターとユース ケースの関係を示すことに重点が置かれます。アクターは、多くの場合 1 つまたは複数のサブジェクトのコンテキストでユース ケースに関与しているエンティティです。アクターは、ユース ケースの定義対象のサブジェクトの外部にある、ユーザー（人間）、外部ハードウェア デバイス、他のサブジェクトなどの場合があります。アクターは、1 つの物理エンティティとは限りません。たとえば、コンピュータ ネットワーク全体であってもかまいません。

複数の異なるユース ケースでは、同一の物理エンティティを表すために、別のロールをもつ複数の別のアクターを使用できます。また、1 つのアクターで、複数のユース ケースの異なる物理エンティティを示すこともできます。

アクターは、クラスのパートまたはインスタンスを参照します。

繰り返しますが、ユース ケース図では、アクターとユース ケースの関係を示すことに重点が置かれます。しかし、アクターのタイプに注目すると有用な場合もあります。たとえば、継承を使用してアクター間の関係を示したり、アクターのプロパティを示したりすることです。これらの情報は、アクターが <<actor>> ステレオタイプのクラス シンボルとして表示されているクラス図で表示されます。

アクター シンボルは、アクターに適用されたステレオタイプまたはアクターが参照するクラス（アクターにステレオタイプが適用されていない場合）を可視化します。

注記

アクターのステレオタイプ use case、actor、subject は表示されません。

サブジェクト

サブジェクトは、一連のユース ケースのシステム境界を定義します。サブジェクトはシステム、サブシステム、クラスを表します。サブジェクトは、クラスのパートまたはインスタンスを参照します。

サブジェクトは、UML 1.X のユース ケースのシステム境界に該当します。

シンボル

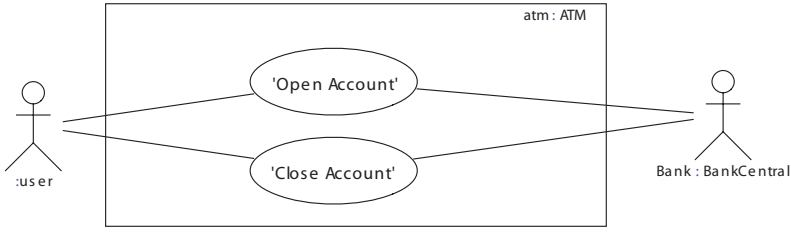


図 57: サブジェクト シンボル（ATM）

ユース ケースを、サブジェクト シンボルの内部に含むことができます。サブジェクト シンボルは、アクティブ クラスなどの振る舞いを表す一連のユース ケースを囲むように描画します。名前とクラス タイプは、サブジェクト シンボルの右上隅に入力できます。

関係

コラボレーションやユース ケース図で以下の関係が使用できます。

関連

関連は、アクターとユース ケースの間で使用され、アクターがそのユース ケースに関与していることを示します。逆に言うと、ユース ケースは、アクターによって実行されていることになります。1 つのアクターが複数のユース ケースに関与したり、1 つのユース ケースに複数のアクターが関与したりすることもできます。

インクルード

インクルード関係は、複数のユース ケース間で使用され、あるユース ケースが別のユース ケースの一部であることを示します。いわば、大きいユース ケースを小さいユース ケースに分割する仕組みです。インクルードする側のユース ケースは、通常それ自身はあまり意味はなく、インクルードされているユース ケースに依存することがほとんどです。

拡張

拡張関係は、複数のユース ケース間で使用され、ユース ケースをいつどのように拡張ユース ケースに挿入するかを示します。拡張ユース ケースは、それ自体で完成していなければなりません。拡張では、通常は、一定の条件下で使用する補足的な機能について記述します。

依存

依存は、ユース ケース間やアクター間で指定されます。依存はエンティティ間の関係の仕方については何も示しません。

2 つのユース ケース間に依存が作成されると、暗黙的にインクルード関係が成立します。

汎化

汎化はユース ケース間で指定できます。つまり、あるユース ケースが、より一般的なユース ケースを特化します。クラスに関連付けられたアクターでは、汎化を指定できます。汎化テキストは非形式的です。

参照

UML の関係

モデルの更新

［ActiveModeler ］アドインを有効にすると、ショートカット メニューに新しいメニュー［モデルの更新］が追加されます。このコマンドをモデルにバインドされていないエンティティに使用し、現在のモデルに追加できます。

ユース ケース図では、以下の機能がサポートされます。

ユース ケース図

ダイアグラムの各要素をすべて更新して、ダイアグラム全体を更新できます。

アクター シンボル

アクター シンボルがバインドされているが、アクターの型がバインドされていない場合、新しいクラスが作成されます。クラス名は、「`actor`」+「`Class`」です。アクター シンボルがバインドされていない（グレー ライン）の場合、アクターに対応する属性が、追加作成されます。

サブジェクト シンボル

サブジェクトのタイプがバインドされていない場合、新しいクラスが作成されます。クラス名は、「`subject`」+ 「`Class`」です。

シナリオ モデリング

シナリオ モデリングでは、主としてシステムやサブシステムの用途、用法についてのシナリオを記述することに焦点を当てます。このシナリオは、ライフライン上で生起するイベントのシーケンスとして記述されます。

モデリング作業を通じて、メッセージのやりとりを詳細化してゆくことで、システム内のコンポーネント間での責任分割、さらにシステムと、そのシステムと相互作用する外部アクターの間の境界線がより明確になります。

シナリオ モデリング作業は、多くの場合、分析作業の早期に行われますが、設計作業でも、より厳密な形で行われることがあります。作成されるシナリオは、システムとシステム コンポーネントの動的インターフェイスの仕様です。シナリオには通常、以下の 2 つの目的があります。

- コンポーネントの振る舞いモデリングの基盤となる
- テスト ケースの基盤となる

UML では、シナリオは相互作用を使用してモデリングされ、イベントは、このセクションで説明するシーケンス図に示されます。相互作用概観図は、個々の相互作用の制御と調整に使用されます。

シナリオ モデリングは、多くの場合、ユース ケース分析の一環として行われます。ユース ケースごとに、そのユース ケースに関連する振る舞いを記述する相互作用が生成され、シーケンス図を使用して相互作用が可視化されます。

シーケンス図

説明

シーケンス図は相互作用を記述するものです。シーケンス図によって、ライフラインやイベント間のメッセージのやりとりが可視化されます。

例

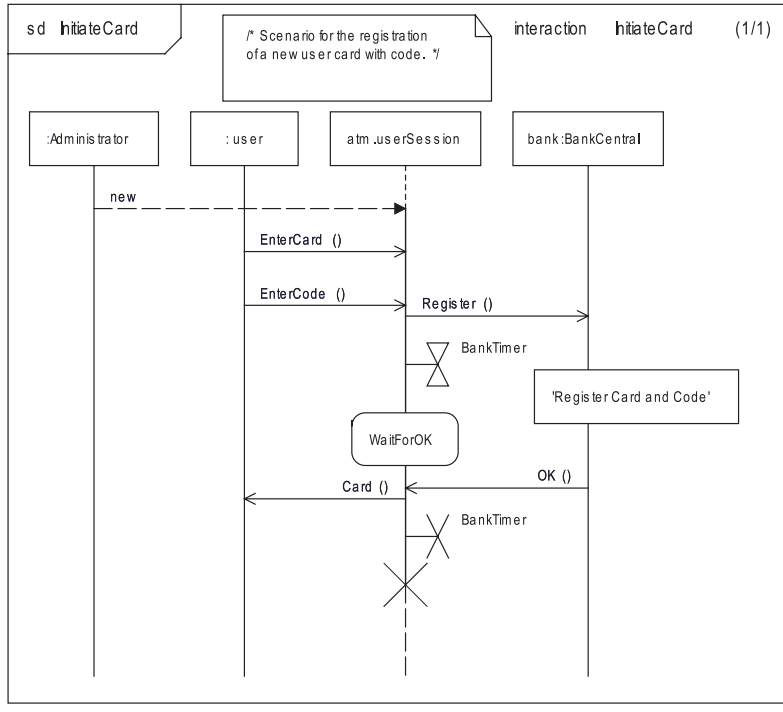


図 58: シーケンス図

シーケンス図のモデル要素

シーケンス図では、以下の要素を使用します。

- ライフライン（生存線）
- メッセージ
- アクション
- ステート
- 相互作用参照
- タイマー イベント
- 生成
- 消滅

シーケンス図の作成

シーケンス図は、相互作用の実装の図形記述です。たとえば、あるパッケージ内でシーケンス図を作成すると、シーケンス図は相互作用とその実装の下に自動的にカプセル化されます。

また、シーケンス図に、操作とユース ケースのような、他の振る舞いの実装を付与できます。この場合、シーケンス図はその振る舞いの内部に作成します。

相互作用

相互作用は、ユース ケース、操作、その他振る舞いを持つエンティティの振る舞いの記述です。相互作用では、パート間の情報交換に重点が置かれます。相互作用は通常シーケンス図によって記述されます。

相互作用の意味は、相互作用から導出される一連のトレースによって定義されます。トレースとは、イベント発生のシーケンスです。このシーケンスは、完全に整列されているとはかぎりません。トレースによるシナリオには、可能なものも不可能なものもあります。

相互作用は、他の相互作用から参照できるので、再利用が可能です。通常は、別のユース ケースを参照する相互作用参照シンボル、または振る舞いの定義として相互作用を含む操作によって行います。ライフライン分解によって相互作用を参照することもできます。

相互作用の通常の使用方法には、以下の 2 つがあります。

- システムとコンポーネントの、外部から見える振る舞いの指定
- システム実行のトレースの記述

参照

シーケンス図

ユース ケース

相互作用参照

相互作用参照は、シーケンス図で相互作用の参照を表すために使用します。参照先の相互作用は、通常、それ自体のシーケンス図で記述されます。相互作用参照で使用する名前は、相互作用自体の名前でなく、相互作用を含むユース ケースまたは操作の名前です。

相互作用参照は、以下の 2 つの点で便利です。

- 重要なメッセージのやりとりに焦点を当てながらも詳細な相互参照を隠す、カプセル化機構として使用できます。
- 相互作用の記述の再利用を可能にします。

シンボル

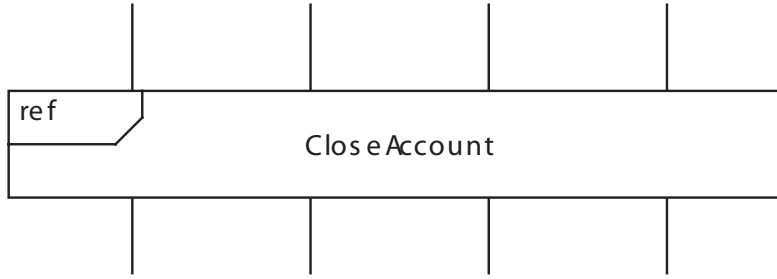


図 59: 相互作用参照

構文

相互作用参照シンボルには、ユース ケース、操作その他振る舞いを持つエンティティを参照する名前が含まれます。

参照

相互作用

ユース ケース

シーケンス図

ライフライン（生存線）

ライフラインは、1 つの相互作用に関与する個々の参加者を表します。パートとストラクチャフィーチャには 1 よりも大きい多重度を持たせることができますが、ライフラインは、1 つの相互作用エンティティのみを表します。ライフラインが、多重度が 1 よりも大きいパートを表す場合は、インデックスにより、特定のインスタンスを選択する必要があります。

シンボル

ライフライン シンボルは、「頭部」と生存線である「軸」から構成されます。ライフラインがまだ生成されていない場合は、軸は破線になります。ライフラインが消滅する（インスタンスが終了する）と、軸が再度破線になります。

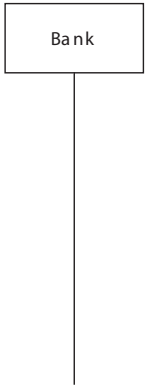


図 60: ライフライン シンボル

ライフラインの作成

ライフラインを生成するには以下の手順を行います。

- ［ダイアグラム要素の作成］ツールバーを使用して［ライフライン］シンボルを選択します。ライフライン シンボルをダイアグラムに配置します。**パート名**または**クラス名**などの適切な情報をヘッダーに入力します。
- クラス シンボルをモデルからシーケンス図にドラッグして、このクラスを表現するライフラインを作成します。ライフラインはクラスのどのインスタンス、ヘッダー内のテキスト**クラス名**で表現します。
- モデルからパート シンボルをドラッグして、このパートを表示するライフラインを作成します。ライフラインはパートのインスタンスを、ヘッダー内のテキスト**パート名**（スコープ修飾子が必要な場合は、**修飾子 :: パート名**）で表現します。

イベントの整列

ライフラインに沿ったイベント発生の順序は、イベントが実際に発生する順序を表す重要なものです。ただし、ライフライン上のイベント発生間の絶対距離は、特に意味をもちません。

イベントの順序は、1 つのライフライン上では厳密に指定されています。しかし、通常は、複数のライフライン上ではイベント間に特定の順序はありません。また、各非同期コンポーネントをそれ自体のライフラインで記述した相互作用やシーケンス図を使用して、分散システムを記述することも可能です。

一般に、複数のライフライン上のイベントの順序を確定するための唯一の仕組みは、メッセージ送信による同期です。この、シーケンス図の順序確定機構は、**部分的整列**（partial ordering）と呼ばれます。なぜならば、完全な順序の確定ではなく、また、まったく不規則な順序付けでもないからです。

非同期でも分散型でもないシステム（スレッドのない通常のプログラム）の場合は、当然ながら、非同期のケースよりも厳密に順序を解釈できます。

ライフライン分解

ライフラインで、合成状態、つまりパートのあるオブジェクトを参照できます。この方法によって、相互作用の複雑性は軽減され、最も重要なメッセージのやりとりに焦点を当てられます。

ただし、場合によっては、内部のやりとり、つまり合成 オブジェクトのパート間の詳細なメッセージのやりとりも確認するとよい場合があります。ライフラインの分解機構によって、同じ振る舞いに対して、ハイレベルの概要と詳細の 2 種類の情報を記述できます。詳細な相互作用は、ライフラインの頭部から参照されて、別のユース ケースまたは操作で定義されます。188 ページの図 61 の例を参照してください。

分解の例

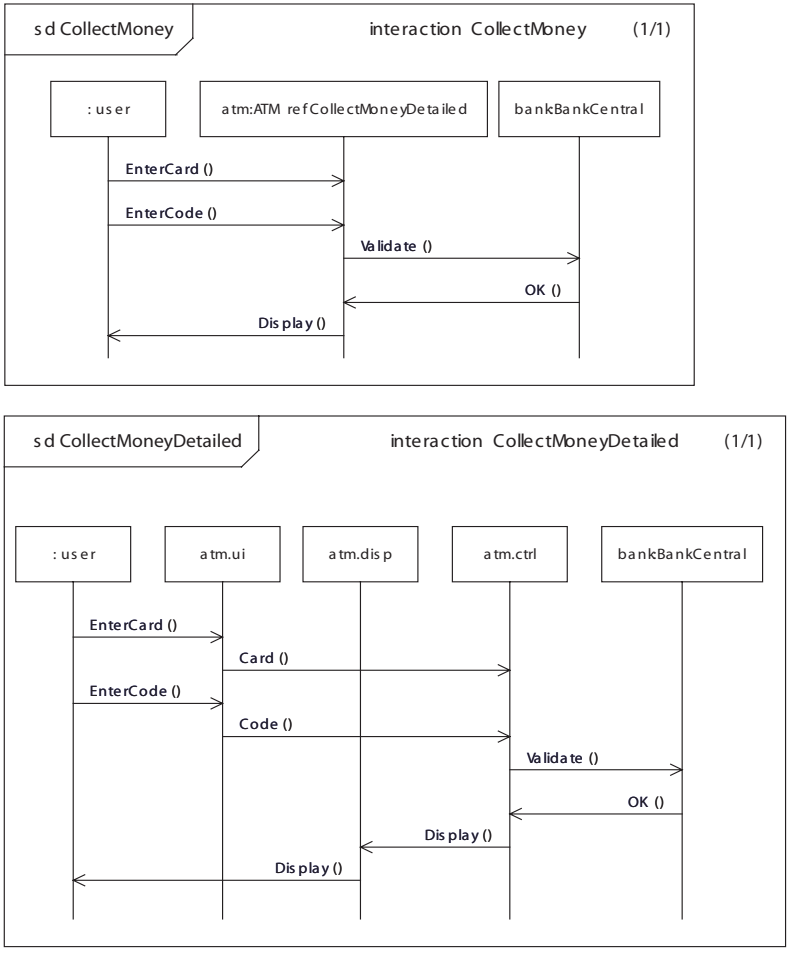


図 61: ライフライン分解の例

構文

ライフラインでは、次の構文が使用できます。

Bank

パート、ポート、属性、またはサブジェクトを参照するインスタンス名

`Bank:BankCentral`
インスタンス名と、クラスを示すタイプ名

`:BankCentral`
クラスを示すタイプ名

`atm[3]`
多重度を 1 インスタンスに減少させる セレクタ式を持つインスタンス名

`atm.Display`
パートを参照する属性を持つインスタンス名

`atm ref OpenAccountDetailed`
インスタンス名と、別の相互作用とシーケンス図で記述されるユース ケースまたは操作を参照するライフライン分解

`atm[2].Display:ATM ref CloseAccountDetailed`
セレクタ、パート、タイプ、ライフライン分解を持つインスタンス名

メッセージ

メッセージとは、シグナル、メソッド呼び出し、メソッド応答の発生を意味します。メッセージには、通常 2 つのイベントがあります。1 つは送信ライフラインの送信イベント（アウト）で、もう 1 つは受信ライフラインの受信イベント（イン）です。メッセージは、水平に描画される場合と傾斜をつけて描画される場合がありますが、ダイアグラム中で、受信イベントを送信イベントより上に表示するべきではありません。

シンボル

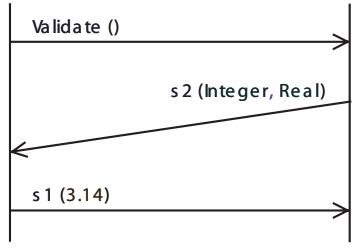


図 62: メッセージ

メッセージを送信し、受信側に渡すには時間がかかる場合がありますが、傾斜にはその意味は含まれません。同様に、水平メッセージは必ずしも直接受信側に届けられるというわけではありません。

シグナルとメッセージには関係があるため、メッセージ名はモデルに表されているシグナルを必ず参照しなければなりません。シグナルにパラメータがある場合は、メッセージには実パラメータ表現が必要です。

メッセージの作成

メッセージに関連するテキスト フィールドは 3 つあります。そのうち 1 つはシグナル名とパラメータで、あとの 2 つはゲート名です。

メッセージを配置する方法は 2 通りあります。

従来の方法 : [ダイアグラム要素の作成] ツールバーの [メッセージ ライン] をクリックします。
送信者ライフラインをクリックし、次に受信者ライフラインをクリックします。

シングル クリックによる方法 : [ダイアグラム要素の作成] ツールバーの [メッセージ ライン] をクリックします。
ライフラインの間で**クリックしたまま**にすると、メッセージが左側のライフラインに接続されます。ライフラインをクリックして**放す**と、右側のライフラインにメッセージが接続されます。これで、以下のような方法でメッセージを作成できます。

- クリックして放し、右のライフラインに受信ポイントを作成します。
- ドラッグしてライフラインと交差し、メッセージを受信するライフラインの左に近い位置でマウスボタンを放します。
- Shift キーを押しながらクリックして右から左にメッセージを送信します。

作成するライン タイプは以下のとおりです。

- **通常のメッセージ** : ツールバーでメッセージを選択します。ライフラインの間でクリックすると、左から右にメッセージが流れる形で作成されます。Shift キーを押しながらクリックすると、右から左にメッセージが流れる形で作成されます。クリックしたままライフラインを交差するようにドラッグし、放すと、メッセージ方向にある次のライフラインに接続します。
- **自身へのメッセージ** : 要素ツールバーで、[メッセージ ライン] シンボルを選択して、同じライフラインを 2 回クリックします。
- **現在のモデル内の既存シグナルを参照して、メッセージを描画 :**

1. [ダイアグラム要素の作成] ツールバーの [メッセージ ライン] をクリックします。
2. メッセージの送信元とするライフラインにカーソルを合わせてクリックします。
3. メッセージの送信先とするライフラインの近くにカーソルを合わせて、右クリックします。ショートカット メニューの [既存要素を参照] にカーソルを合わせ、リストからシグナルを選択します。

[既存要素を参照] を選択するとスコープ内のシグナルが表示されます。以下の条件に従ってシグナルが表示されます。

- 送信先ライフラインにタイプがある場合、実現化インターフェイスのシグナルを考慮に入れて、このタイプで受信可能なすべてのシグナル、およびクラス自体で定義されたシグナルなどがリストに表示されます。
- 送信元ライフラインにタイプがある場合、すべての必須インターフェイスを考慮に入れて、このタイプで送信可能なすべてのシグナルがリストに表示されます。

- 送信元と送信先のライフラインにタイプがなく、送信先ライフラインにセレクタがある場合、実現化インターフェイスのシグナルを考慮に入れて、セレクタ タイプで受信可能なすべてのシグナルと、クラス自体で定義されたシグナルなどがリストに表示されます。
- 送信元と送信先のライフラインにタイプがなく、送信元ライフラインにセレクタがある場合、既存のすべての必須インターフェイスを考慮に入れて、セレクタ タイプで送信可能なすべてのシグナルがリストに表示されます。
- 上記以外の場合、ライフライン自体から見えるすべてのシグナルがリストに表示されます。

場合によって、メッセージが 1 本のライフラインにのみ接続されるように描画することもできます。これは、特にシーケンス図を使ってトレースする場合に有用です。メッセージには 4 つのタイプがあります。

- **新しいメッセージ**。メッセージは送信済みですが、まだ受信されていません。このメッセージは送信側に接続されています。
- **消失メッセージ**。メッセージは送信済みですが受信されていません。メッセージは送信側に接続され、小さな丸がメッセージの矢印に描画されます。
- **古いメッセージ**。メッセージは受信されましたが、送信側がまだ特定されません。このメッセージは受信側に接続されています。
- **基底メッセージ**。メッセージは受信されましたが、送信側が不明です。このメッセージは受信側に接続され、メッセージが小さな丸から出てきます。

プロパティエディタを使用してメッセージを Lost または Found とマーク付けできます。

メッセージラインは以下の方法でも作成できます:

- 1 本のライフラインを選択した状態で、Shift キーを押しながらシンボル要素ツールバーの［メッセージ ライン］シンボルをクリックすると、新しいメッセージが作成されます。新しいメッセージはライフラインの最後、消滅ライフライン シンボルの前に配置されます。
- 2 本のライフラインを選択した状態で、Shift キーを押しながらシンボル要素ツールバーの［メッセージ ライン］シンボルをクリックすると、ライフラインの間に通常のメッセージが生成されます。通常のメッセージは、ライフラインの最後（消滅ライフライン シンボルの前）に水平に、左から右の方向に配置されます。
- 1 つのメッセージを選択を選択した状態で、Shift キーを押しながらシンボル要素ツールバーの［メッセージ ライン］シンボルをクリックすると、選択したメッセージのすぐ下に通常のメッセージが生成されます。通常のメッセージは選択したメッセージと同じライフラインに接続され、同じ方向性を持ちます。

注記

メッセージを編集する際、パラメータの表示／非表示に関わらず、メッセージのすべてのパラメータを見ることができます。編集モードを終了すると、メッセージ テキストがパラメータの表示／非表示は他のメッセージと同じ設定になります。

パラメータの切り替え

ダイアグラム中のメッセージパラメータの表示 / 非表示を切り換えます。デフォルトで、すべてのパラメータが表示されます。クイックボタンを押して、現在のシーケンス図のメッセージ パラメータの表示／非表示を切り替えられます。

不完全なメッセージ

メッセージは、イベントのうち 1 つしか指定されていないという意味で、不完全な場合があります。受信イベント（イン）がない場合は、消失メッセージです。送信イベント（アウト）がない場合は、拾得メッセージです。

消失メッセージ

消失メッセージとは、送信イベントが既知で、受信イベントがないメッセージです。これを使用して、メッセージが宛先に届かないケースを記述できます。

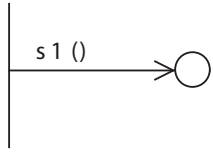


図 63: 消失メッセージ

拾得メッセージ

拾得メッセージとは、受信イベントが既知で、（既知の）送信イベントがないメッセージです。これを使用して、メッセージの送信元が記述の範囲外にあるケースをモデリングできます。また、複数のライフラインが送信側となり得、かつどのライフラインであるかがシナリオに関係しないような場合に、不要な指定を避けるためにも使用できます。

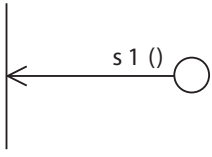


図 64: 拾得メッセージ

メッセージのコピー

メッセージをコピーするには 2 つの方法があります。1 番目の方法は常に送信者と受信者を維持します。

CTRL + ドラッグ : CTRL キーを押しながら、コピーしたいメッセージをクリックしてコピー先の場所へドラッグアンドドロップします。そして、CTRL キーをリリースします。

2 番目の方法は、送信者と受信者を変更します。

コピー / 貼り付けコマンド : コピーしたメッセージを右クリックして、メッセージのショートカットメニューを開きます。メニューから［コピー］を選択します。メッセージのコピー先の場所で右クリックして、メニューから［貼り付け］を選択します。コピー / 貼り付けのショートカット操作である、CTRL + C キーと CTRL + V キーも使用できます。貼り付けられる場所は貼り付けコマンドを発行する直前にクリックした場所です。

以下のオプションがあります。

- クリックした場所が 2 つのライフラインの間である場合、新しいメッセージはそのライフラインを結ぶように作成されます。
- クリックした場所が 2 つのライフラインの間でない場合、送信者と受信者は元のメッセージとおなじになります。

タイマー イベント

タイマーは通常、相互作用内の 2 つの異なるイベントで記述されます。1 つ目のイベントはタイマー設定で、2 つ目のイベントはタイムアウトまたはリセットです。

タイマーを使用するには、メッセージで対応するシグナルや操作を宣言する必要があるのと同様に、宣言が必要です。タイマーは、クラス図ではタイマーシンボルで宣言し、テキスト シンボルまたはテキスト図ではテキスト構文を使用して宣言します。

タイマーイベントシンボルには、名前とパラメータのための 1 つのテキストフィールドがあります。

タイマー設定

設定イベントにより、タイマー インスタンスが生成され、アクティブになります。タイマー設定イベントは、タイマー設定アクションにマッピングします。

タイマー リセット

リセット イベントは、アクティブなタイマーを中止します。タイマー リセット イベントは、タイマー リセット アクションにマッピングします。

タイマー タイムアウト

タイムアウト イベントは、タイマー継続時間が経過し、タイマー シグナルが受信され、状態機械に処理されると発生するものです。タイムアウト イベントは、タイマー シグナル処理にマッピングします。

シンボル

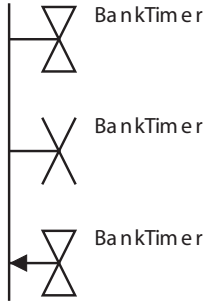


図 65: タイマー設定、リセット、タイムアウトのシンボル

参照

タイマー
タイマー設定アクション
タイマー リセット アクション

タイマー仕様ライン

タイマー仕様ラインを使用して、**絶対時間**ライン、**相対時間**ラインと**全体順序**ラインを生成します。

絶対時間ライン

絶対時間ラインをライフラインの左または右側に追加して、絶対時間、または時間範囲「`{<Time>}`」を指定できます。ラインはライフラインに沿って上下に移動できます。絶対時間ラインを作成するには、シンボル パレットの［タイマー仕様ライン］シンボルをクリックして、一端のみライフラインに接続したラインを描画します。

相対時間ライン

相対時間ラインを作成するには、シンボル パレットの［タイマー仕様ライン］シンボルをクリックして、両端がライフライン接続したラインを描画します。

特定の持続時間の観察「{<Duration>}」、または、持続時間の制約「{<Duration>..<Duration>}」を、テキスト フィールドで指定できます。

相対時間ラインには、上限、下限、持続時間を指定できます。ラインはライフラインの右側に描画されますが、左側に移動することもできます。相対時間ラインの上限と下限はライフラインに沿って上下に移動できます。

通常、相対時間ラインの開始イベントと停止イベントはライフラインの他のイベントに接続されます。たとえば：

- メッセージの到達
- メッセージの送信
- 参照シンボルの始点／上部
- 参照シンボルの終点／下部

相対時間ラインの始点または終点を、開始イベントと終了イベントが接続されていない他のイベントに配置できます。

全体順序ライン

全体順序ラインは 2 本のライフライン間を移動するタイマー仕様ラインです。全体順序ラインを作成するには、シンボル パレットの［タイマー仕様ライン］シンボルをクリックして、2 本のライフラインの間にラインを描画します。

全体順序ラインは、メッセージ ラインを使用しない複数の異なるライフラインでイベントを指定する際に、活用します。全体順序ラインは、中央に矢印が付いた破線で表示されます。通常、テキストはラインに関連付けられていませんが、特定の持続時間「<Duration>}」、または、時間範囲「<Duration>..<Duration>}」をラインに関連付けることができます。

ステート

ステート シンボルは、ライフラインで記述されたインスタンスが特定のステートにあることを示すのに使用します。

シンボル

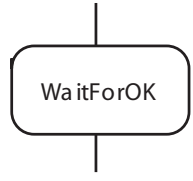


図 66: ステート

シナリオの作成時にステートを使用するのは、多くの場合、特定の状態を強調するためです。通常、ライフライン上に通過済みのステートをすべて表記することはありません。

ステートは、ライフラインが参照するアクティブ クラスの状態機械に同じ名前のステートがある場合、モデル要素に結び付けられます。

ただし、トレースの場合は、各ステート シンボルは状態機械遷移の特定の「次のステート」オカレンスにマッピングされます。これは、ライフライン オブジェクトに主状態機械が 1 つしかない場合です。パートのあるアクティブ オブジェクト、つまり複数の状態機械のあるアクティブ オブジェクトの場合は、単純なマッピングは実行不可能です。

アクション

アクション シンボルは、ライフラインで発生するイベントを表現するために使用します。状態機械のアクション シンボルに相当します。非形式的な文は、コメントとして記述する必要があります。

シンボル

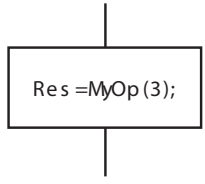


図 67: アクション

アクションで使用できるテキスト構文は、状態機械図のアクション（タスク）シンボルと同じです。

生成

生成イベントは、アクティブ クラスに適用される New 操作に相当します。

生成されたライフラインは、生成イベント受信前は破線で示され、まだ生成されていないことを示します。生成ライン上の名前はライフラインに対応するクラスの名前です。

シンボル

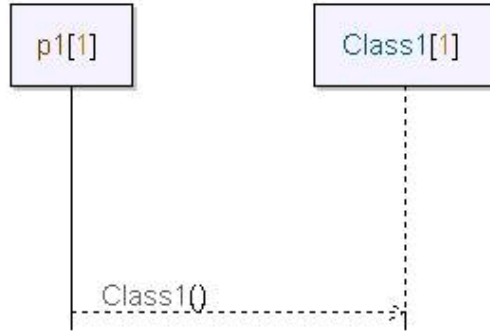


図 68: 生成メッセージ

作成ラインの作成

クラスの動的インスタンスを表現するライフラインを描画する際、作成イベントを描画できます。これは［ダイアグラム要素の作成］ツールバーの［生成ライン］ボタンを使用して行い、メッセージのように扱うことができます。生成ラインの名前は、ライフラインに対応するクラスの名前にします。生成ラインは、クラスのコンストラクタ操作を参照します。生成ラインに関連するテキスト フィールドが 3 つあります。そのうち 1 つはコンストラクタ操作名とパラメータで、あとの 2 つはゲート名です。仮パラメータは、操作パラメータと同様に、メソッド呼び出しラインに追加できます。

コンストラクタのバインディング

基底クラスのコンストラクタが「initialize」という名前になっていると、基底クラスのコンストラクタへのコンストラクタ初期化参照のバインディングが失敗します。コンストラクタ名はクラス名と同じにすることを推奨します。

例 **20:** バインドされないコンストラクタ **initialize**

```
class AutoDispatchableClass :tor::DispatchableClass {
  initialize(tor::DispatchableClass d) {
    d.addToCurrentDispatcher(this);
    init();
    'start'();
  }
}

class MyClass :AutoDispatchableClass {
  initialize(tor::DispatchableClass
d):AutoDispatchableClass(d) {  }
}
```

`AutoDispatchableClass` 参照はバインドされない。

消滅

消滅イベントは、インスタンスの終了を表します。状態機械の停止アクションに相当します。消滅イベント後は、ライフラインにイベントは発生しません。

シンボル

図 69: 消滅

インライン フレーム

インライン フレーム シンボルを使用すると、相互作用内で同様に処理すべきメッセージをグループ化できます。つまり、バリエーションごとにダイアグラムを作成するのでなく、複数の種類のバリエーションを 1 つのダイアグラムで表現できます。

シンボル

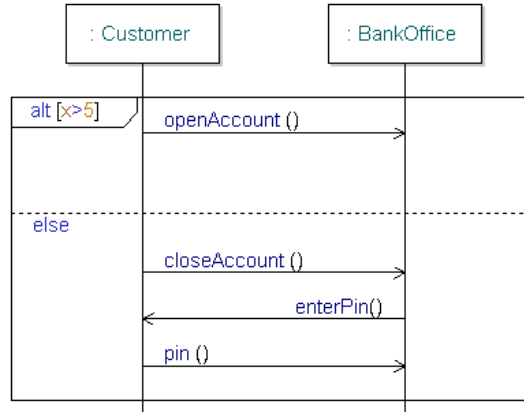


図 70: インライン フレーム

[インライン フレーム] シンボルを別の [インライン フレーム] シンボル内に保持することが可能です。既存の [インライン フレーム] シンボルと同じ高さに 2 つ目の新しい [インライン フレーム] シンボルを追加すると、既存の [インライン フレーム] シンボルの内部に入ります。

[インライン フレーム] シンボルには、1 つまたは複数のインライン フレーム セクションがあります。デフォルトの [インライン フレーム] シンボルにはインライン フレーム セクションが 1 つあります。インライン フレーム セパレータ ラインは、[インライン フレーム] シンボルを複数のインライン フレーム セクションに分割します。個々のインライン フレーム セパレータ ラインには制約テキストがあります。

インライン フレーム セパレータ ラインは、[インライン フレーム] シンボルが 1 つだけ選択されたときに表示されるライン ハンドルとともに、作成されます。

インライン フレーム セパレータ ラインはシンボル内で上下にドラッグできますが、同じインライン フレーム シンボルに接続された、他のセパレータ ラインを越えることはできません。インライン フレーム セパレータ ラインを削除するには、セパレータ ラインを選択して、Delete キーを押します。

セクションを削除すると、セクション内のオブジェクトも削除されたセパレータの一部として削除されます。

インライン フレームには、以下の組み合わせのテキストが 1 つあります。

- 演算子キーワード 例 : seq （デフォルト キーワード）alt、else、loop、assert
- 制約テキスト 例 : ”[a<3]”、”else”

バリエーション

バリエーションはいくつか考えられます。フレームはメッセージの代替グループを表現するために分割されることもあります。バリエーションを以下に示します。

- `alt` : 代替の 1 つのブランチ、つまり分岐を表します。フレームは複数のオペランドに分割でき、各オペランドを条件と関連付けることができます。条件の値が true の代替ブランチのみが選択されます。ブランチのうち 1 つのみを else ブランチにできます。
- `opt` : グループ化されたメッセージがオプションであることを表します。つまり、発生しなくてもかまいません。opt フレームは分割できません。条件と関連付けることはできます。この場合、2 つ目の選択肢が空であるような選択肢と同じような振る舞いをします。
- `loop` : 一連のメッセージが、何度か繰り返されることを表します。loop フレームは分割できません。繰り返し回数は、最小値と最大値を `loop(min, max)` という形式で指定します。「`max`」に「*」指定することもできます。これは、無限ループを表します。
- `par` : 複数のオペランドのメッセージを互いにインターリーブするか、並行して発生するようにすることはできるが、各オペランド内の整列制約を守る必要があることを表します。有効にするには、par フレームを分割する必要があります。
- `seq` : これは、シーケンス図の通常のセマンティックを表します。つまり、各ライフラインは他のライフラインと独立です。まず弱い順序付けが厳密な順序付けに優先して使用されます。
- `strict` : シーケンス図または組み合わせフラグメント内の該当メッセージに、厳密な順序付けをする必要があることを表します。つまり、ダイアグラム中の垂直方向の位置は、事象が発生する順序に相当します。これを、シーケンス図のデフォルトである、各ライフラインにそれぞれのタイムラインがある弱い順序付けと比較します。厳密な順序付けを使用すると、これを関連ライフラインの共通グローバル時間と考えることができます。
- `neg` : 該当の一連のメッセージが無効であることを表します。
- `critical` : 該当のメッセージが他のインライン フレームとインターリーブできないことを示します。これはたとえば、par フレーム内で、一連のメッセージの暗黙的インターリーブを無効にするために使用できます。
- `break` : シーケンス図の以降の部分に割り込み、代わりに break フレームで囲まれた一連のメッセージを実行するという例外的なオカレンスを表します。break フレームは分割できません。
- `assert` : assert フレームで表されたシーケンスのみが有効で、他のシーケンスは無効であることを表します。assert フレームは分割できません。
- `ignore` : 所定のメッセージが重要でなく、フレームに表示されないことを表します。これで、相互作用で最も重要なメッセージのみが表示されるようになります。形式は、ignore {<list_of_messages>} です。逆の操作は consider です。ignore フレームは分割できません。
- `consider` : フレーム内で所定のメッセージが重要であり、表示されないメッセージが重要でないことを表します。形式は、consider {<list_of_messages>} です。逆の操作は ignore です。consider フレームは分割できません。

共通リージョン

共通リージョンは、1 つのライフラインでメッセージが（送信または）受信された順序が重要でないことを示すのに使用します。

シンボル

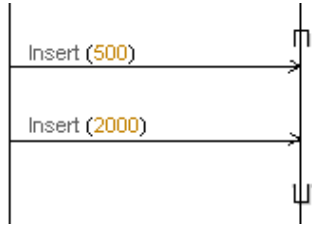


図 71: 共通リージョン

継続

継続は alt インライン フレームでのみ使用され、シーケンスのパートからパートへの継続のしかたを決定するラベルとして機能します。継続で終わる相互作用は、同じ継続で始まる相互作用のみ継続されます。

シンボル

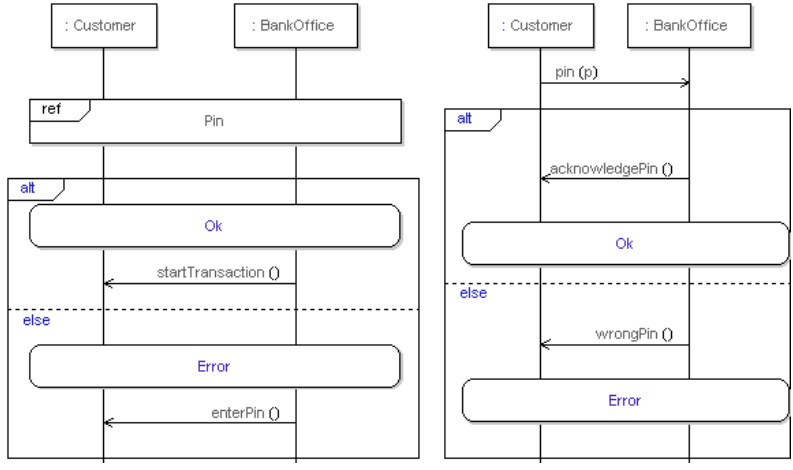


図 72: 継続

継続シンボルは、ステート シンボルと似ていますが、複数のライフラインにまたがることができます。

このシンボルの中央にはテキスト フィールドがあります。入力されたテキストは解析されず、そのままシンボルに保存されます。

［継続］シンボル内にシンボルとラインを配置することはできません。

メソッド呼び出し

メソッド呼び出しはメッセージに似ていますが、常に同期的です。つまり、必ずメソッド応答と関連付けられます。メソッド呼び出しは、たとえば、異なるクラス間での操作の起動方法などのモデリングに使用します。

シンボル

図 73: メソッド呼び出しとメソッド応答

メソッド呼び出しの結果は、実線の矢印で表される呼び出し、破線の矢印で表される応答、アクティベーション領域、サスペンション領域の 4 つの図形要素で示されます。サスペンション領域は呼び出すライフラインの破線の長方形で、アクティベーション領域は呼び出されるライフラインの実線の長方形です。

呼び出しメッセージと応答メッセージには、関連するテキスト フィールドがそれぞれに 3 つあります。そのうちの 1 つは操作名とパラメータで、あとの 2 つはゲート名です。

完全なメソッド呼び出しを描画するには以下の手順を行います。

1. ［ダイアグラム要素の作成］ツールバーの［メソッド呼び出し］シンボルをクリックします。
2. メソッド呼び出しの送信側のライフラインに**呼び出しメッセージ**開始イベントを置き、これを受信側にドラッグします。
3. 操作名とパラメータ情報を入力するか、操作をモデルからメッセージにドラッグします。
4. 呼び出しメッセージと応答メッセージの名前フィールドの操作パラメータ タイプ情報を編集します。

応答メッセージのメイン テキストは、通常、呼び出しメッセージと同じメソッドを参照します。メソッドが送信パラメータに値を割り当て、さらに戻り値があるような場合、パラメータは異なります。応答ラインのみに戻り値 < 値 > を与えることはできます。

メソッド呼び出しまたは応答ラインを削除すると、これに接続されたサスペンションとアクティベーション領域シンボルも削除されます。

サスペンションまたはアクティベーション領域シンボルを削除すると、シンボルのみ削除され、接続されたメソッド呼び出しと応答ラインは削除されません。

呼び出しメッセージをドラッグすると、メソッド呼び出しシンボル全体が対応する方向に移動します。

応答メッセージをとラッグすると、一致する方向に、メソッド呼び出し シンボルのサイズが縮小または拡大されます。

注記

メソッド呼び出しを切り取り／貼り付け、またはコピー／貼り付けによって編集すると、予期せぬ結果になることがあります。貼り付け操作時のエディタの振る舞いには、アクティベーションとサスペンション領域シンボルは含まれません。領域シンボルはオプションと考えることができます。領域シンボルを選択して Delete キーを押すと、メソッド呼び出し全体を削除せずに、これらのシンボルを削除できます。領域シンボルを元に戻すには、取り消しを実行するか、新しいメソッド呼び出しを生成します。

ゲート名

ショートカット メニューで［ゲート テキストの追加／削除］を選択して、メッセージ、メソッド呼び出し、または作成イベントにゲート名を追加できます。2 つのゲート名テキストはラインの下に配置されます。ゲート テキストがアクティブになると、ゲートがデフォルト名を取得します。デフォルト名は変更可能です。

アクティベーションとサスペンション

メソッド呼び出しを発信するライフラインは、受信側が実行でビジー状態の間は中断されます。つまり、応答を待つのみです。メソッド呼び出しを受信するライフラインは、起動されたメソッドの実行中はアクティブになります。呼び出し側に応答が返されると、アクティベーション領域とサスペンション領域の両方がクローズになります。

モデルの更新

［Active Modeler］アドインを有効にすると、ショートカット メニューに新しいメニュー［モデルの更新］が追加されます。このコマンドをバインドされていないエンティティに使用し、バインド可能な現在のモデルにモデル要素を作成できます。

シーケンス図では、以下がサポートされます。

シーケンス図

ダイアグラムの全要素を更新して、ダイアグラム全体を更新します。

相互作用オカレンス シンボル

参照を持つユース ケースと同じコンテキストで、シーケンス図で新しいユース ケースを作成します。新しいユース ケースの名前には、参照シンボルの名前を使用します。名前を指定しない場合、デフォルト名が使用されます。

修飾名は考慮されません。修飾子は新しいユース ケース名の一部になります。

ライフライン シンボル

ライフラインのタイプがバインドされていないか、またはタイプが指定されていない場合、クラスを作成してライフラインのタイプを更新します。バインドされていないタイプがある場合、このタイプを使用してクラス名を指定します。ライフラインに名前が指定されている場合、これに従ってクラス名、「`lifeline name`」+「`Class`」が命名されます。ライフラインの名前が指定されていない場合、デフォルト名が使用されます。

ライフラインに名前が指定されている場合、この名前で、ライフラインに対応する属性の有無を確認します。対応する属性がない場合、新規に作成します。

モデルの更新は、このライフラインを送信側または受信側に持つメッセージ ラインごとに実行されます。このライフラインが所有するすべてのシンボル（ステートやタイマーなど）で、モデルの更新が実行されます。

修飾名は考慮されません。

メッセージ ライン

メッセージ ラインのテキストに合致するシグナルを作成します。パラメータのタイプはメッセージ ラインの値から導き出されます。パラメータがバインドされていない識別子で表現されている場合、Any タイプと識別子と同じ名前を持つ新しい属性がシーケンス図に作成されます。

可能であれば、シグナルがインターフェイスに追加されます。ライフラインのタイプを使用して、適切なインターフェイスが検索されます。

- タイプに正しい方向で実現または要求されたインターフェイスを持つポートがあれば、このインターフェイスが再利用されます。
- 合致するインターフェイスが見つからないか、ライフラインのタイプが指定されていない場合、インターフェイスとして他のメッセージ ラインが検索されます。
- 1 つのインターフェイスに属すシグナルがバインドされた同じライフライン間に、他のメッセージラインが存在する場合、このインターフェイスが再利用されます。
- 上記の方法でインターフェイスを検索できない場合、新規作成されます。

さらに、送信側と受信側のライフラインのタイプが指定されている場合、シグナルの送受信を可能にするためこれらのタイプが更新されます。タイプがパッシブの場合、アクティブに変更されます。ポートがない場合、新規ポートが作成されます。シグナルを含むインターフェイスを要求または実現するようにポートが更新されます。

メソッド呼び出しライン

送信側ライフラインのタイプでクラスに操作を作成します。ライフラインのタイプが指定されていない、またはタイプがバインドされていない場合、エラーが発生します。操作のパラメータと戻り値は、メソッド呼び出しラインの値から算出されます。また、応答ラインから戻り型が算出されます。

操作がすでにバインドされており、所有者がインターフェイスの場合、この操作をインターフェイスからクラスにコピーできます。

メソッド応答ライン

操作に戻り値があるか確認し、ない場合新たに作成します。タイプは応答ラインの値から導き出されます。戻りパラメータと戻り型が供給された値と異なる場合、エラーが発生します。

更新は操作がバインドされている場合のみ行われます。

［ステート］シンボル

ライフラインのクラス タイプを指定する状態機械にステートを作成します。ステート名はシンボル内のテキストと同じです。ライフラインのタイプが指定されていないか、タイプがバインドされていない場合、エラーが報告されます。クラスに状態機械がない場合、新たに作成します。

［タイマー設定］シンボル

［タイマー設定］シンボルを持つライフラインのタイプを指定するタイプにタイマーを作成します。タイマー名はタイマー シンボルのテキストと同じです。名前を指定しない場合、デフォルト名が使用されます。ライフラインのタイプを指定していない場合、エラーが報告され、モデルは変更されません。

タイマー パラメータとデフォルト持続時間は考慮されません。

［タイマー タイムアウト］シンボル

［タイマー設定］シンボルを参照してください。

［タイマー リセット］シンボル

［タイマー設定］シンボルを参照してください。

表示と削除フィルタ

レイアウトの圧縮

［レイアウトの圧縮］ボタンは、［オプション］のシーケンス図の指定にしたがって、メッセージ間とライフライン間の間隔を圧縮します。

［レイアウトの圧縮］ボタンをクリックすると、ライフラインが圧縮されて、水平方向に移動します。

Shift キーを押しながら［レイアウトの圧縮］ボタンをクリックすると、ライフラインが上記のように圧縮され、さらにライフラインのオブジェクトもライフラインに沿って上／下に圧縮されます。

Ctrl キーを押しながら［レイアウトの圧縮］ボタンをクリックすると、ライフラインは最初のイベントを持つライフラインがダイアグラムの左になるように再配置されます。

Shift + Ctrl キーを押しながら、［レイアウトの圧縮］ボタンをクリックすると、ライフラインのライフラインとオブジェクトが上記のように圧縮され、さらにライフラインと最初のイベント（たとえばシグナル送信）が関連付けられ、ダイアグラムの左側に再配置されます。

選択したシグナルの削除

選択したメッセージを削除します。このコマンドは、選択されたメッセージと同じシグナルを使用して、メッセージを削除します。また、他のオブジェクトの削除にも使用できます。

<X> が選択された場合、このコマンドですべての <X> を削除します。

<X> には以下のメッセージが適用されます。

- 生成ライン
- ステート シンボル
- タイマー シンボル（設定、リセット、タイムアウト）
- タイマー仕様ライン（絶対時間、相対時間、全体順序ライン）
- メソッド呼び出し（呼び出しライン、アクティベーション シンボル、応答ライン、サスペンション シンボル）
- アクション シンボル
- 消滅シンボル
- 参照シンボル
- インライン フレーム シンボル
- 継続シンボル
- テキスト シンボル
- コメント シンボル

選択したシグナルの保持

Shift キーを押しながら［選択したシグナルの削除］をクリックすると、フィルタの効果が逆になります。選択したメッセージと、選択したメッセージと同じシグナルを使用するメッセージだけがシーケンス図に保持されます。他のオブジェクトについては、以下のルールに従います。

<X> を選択していない場合（<X> は選択したシグナルの削除で定義される）、このコマンドによりすべての <X> が削除されます。

スペースの確保

このコマンドを実行すると、選択したシンボルまたは行の下にスペースが作成されます。Shift キーを押しながらツールバーの［スペースの確保］をクリックすると、選択したシンボルまたは行の下のスペースが削除されます。

相互作用概観図

相互作用概観図は、相互作用間の制御フローに焦点を当てたアクティビティ図の一形態です。

相互作用概観図内の相互作用参照は、操作とアクティビティを定義、参照できます。相互作用参照は、アクション ノード ノードとオブジェクト ノードの代わりに使用されます。アクティビティ エッジと分岐ノード、フォーク ノード、アクティビティ終了ノードなどの制御構成要素は、アクティビティズと同じです。

次の表に、バリエーションに示された一般的な相互作用オペランドを相互作用概観図で表す方法を示します。

オペランド	相互作用概観図の構成要素
alt	対応するマージ ノードに一致する分岐ノード
par	対応するジョイン ノードに一致するフォーク ノード
loop	ダイアグラム中の分岐とグラフ サイクル

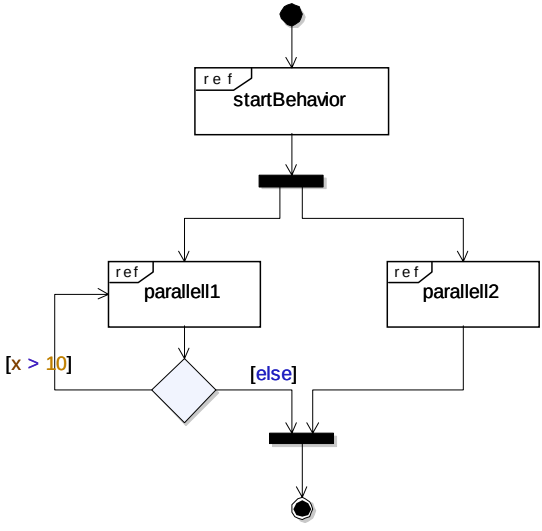


図 74: 相互作用概観図

相互作用概観図の作成

相互作用概観図はクラスとユース ケースに含めることができます。

1. ［モデル ビュー］のクラス（ユース ケース）を選択します。
2. ショートカット メニューから［新規］を選択し、［相互作用概観図］を選択します。

相互作用概観図のモデル要素

相互作用概観図には、以下の要素が使用されます。

- 分岐
- フロー終了
- フォーク
- 開始ノード
- ジョイン
- マージ
- 相互作用参照、アクションノードを参照。
- 関係

参照

シーケンス図
アクティビティ図

パッケージ モデリング

比較的大きなシステムのモデリングの場合は、すべての定義を論理的で管理できるグループとして体系付けるには、パッケージ構成要素が不可欠です。体系付けの原則として、同時に変更される可能性がある、意味的に近接した要素をグループ化するとよいでしょう。

パッケージ図

パッケージ図は、パッケージの集合とその相互の関係を可視化するために使用します。システムの内訳を論理パッケージとパッケージ間の依存関係にモデリングするために使用します。

パッケージ図には、パッケージと、インポート依存やアクセス依存のようなパッケージ間の依存が含まれます。

同じ目的でクラス図も使用できます。

例

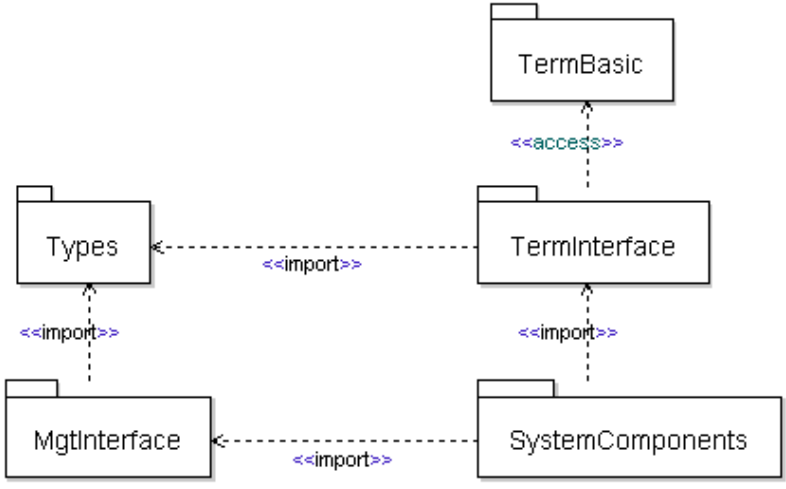


図 75: パッケージとその関係

パッケージ図のモデル要素

パッケージ図には、以下の要素が使用されます。

- パッケージ
- 関係

参照

クラス図

パッケージ

パッケージは、要素をグループに体系付ける仕組みです。パッケージは、グループ化された要素の名前空間となります。パッケージ内では、要素はそれぞれの名前を使用して直接参照できますが、パッケージ外からは、多くの場合、モデル要素の名前を修飾して参照する必要があります。

通常、モデルは互いに依存する複数のパッケージから構成されます。パッケージ相互の関係を理解することは、システムが複雑か複雑でないかにかかわらず、モデリングという点で重要です。パッケージの相互関係は多くの場合システム アーキテクチャを反映したものとなるため、システムの規模が大きくなるにつれ、この重要性は高くなります。

シンボル

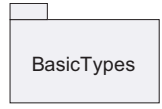


図 76: パッケージ

パッケージにより、パッケージで定義された個々の要素の可視性、要素へのアクセス権限も制御できます。

- クラスやその他のパッケージなどの定義は、パッケージに収集できます。
- パッケージはインポートされることも、別のパッケージからアクセスすることもできます。

他のシンボル階層をパッケージ シンボル内にネストできます。パッケージシンボル内に作成された要素の所有者は、そのパッケージになります

構文

パッケージ シンボルには、パッケージの名前の入るテキスト フィールドが含まれます。参照対象のパッケージが別の名前空間で定義されている場合は、パッケージ名の前には OuterPackage::MyPackage などのように修飾子が付きます。

参照

関係

関係

パッケージ図では以下の関係を使うことができます。詳細については UML の関係で説明しています。

- 依存
- 包含

依存はステレオタイプ化されて、より正確な意味を与えられます。このために使用するステレオタイプは、<<import>> と <<access>> です。

インポート

インポートは、特にパッケージ間や、クラスまたは状態機械などからパッケージへの場合で有効な特殊依存です。定義の名前をパッケージから現在の名前空間にインポートします。通常は、現在の名前空間もパッケージです。これで、修飾子を使用する必

要がなくなります。パッケージ Q によってインポートされたパッケージ P 内の定義の名前は、次にパッケージ Q をインポートするかパッケージ Q にアクセスするパッケージに自動的に含められます。

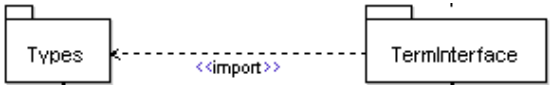


図 77: インポート

注記

インポート スコープ内で修飾子なしにアクセスできる名前の数が非常に大きくなるので、インポート依存関係の使用は制限してください。アクセス依存関係を使用した方がよいでしょう。定義の一部だけを使用する場合は、修飾子の使用も考慮する必要があります。修飾名を使用すると入力する文字数が多くなりますが、モデルに使用されている定義がわかりやすくなります。

アクセス

アクセスは、特にパッケージ間や、クラスまたは状態機械などからパッケージへの場合で有効な特殊依存です。定義の名前をパッケージから現在の名前空間にインポートします。通常は、現在の名前空間もパッケージです。これで、修飾子を使用する必要がなくなります。パッケージ Q がアクセスしたパッケージ P 内の定義の名前は、次にパッケージ Q をインポートするかパッケージ Q にアクセスするパッケージには含められません。

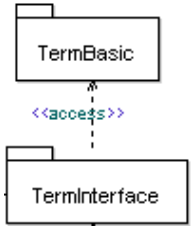


図 78: アクセス

インポートとアクセスは、密接な関係にあります。主な違いは、インポートは遷移的である点です。つまり、パッケージにアクセスしたりパッケージをインポートしたりすると、そのパッケージが次にインポートする定義の名前も自動的に取得されますが、アクセス先の定義の名前は取得されません。210 ページの図 75 では、パッケージ **TermBasic** はパッケージ **TermInterface** のアクセス先となっています。つまり、**TermBasic** の定義の名前を **TermInterface** で直接参照できます。ただし、これらの名前は、パッケージ **TermInterface** をインポートするパッケージ **SystemComponents** で

は直接には使用できません。したがって、**SystemComponents** では、明示的にパッケージ **TermBasic** をインポートまたはアクセスして名前を参照するか、明示的に名前を修飾する必要があります。

アーキテクチャの観点では、インポートよりもアクセスのほうが好ましいと言えます。これは、必要なすべてのパッケージを検討することがアクセスによって強制され、誤って余分なものの取り込むことがなくなるからです。

注記

パッケージをインポートしたりパッケージにアクセスしたりしなければ、パッケージ内の定義を参照できないわけではありません。定義が public であれば、修飾を使用して参照できます。たとえば、`TermBasic::Xterm` を使用してパッケージ `TermBasic` の要素 `Xterm` を参照できます。しかし、明快さという点で、通常はパッケージの相互依存関係の記述を作成するほうがよいでしょう。

参照

UML の関係

<<noScope>> パッケージ

<<noScope>> パッケージは、通常、パッケージの要素を複数のファイルに分割する必要がある場合に使用します。ただし、パッケージの内容を複数のパートに構造化する必要があるが、UML の名前スコープ ポイントのパッケージは 1 つのエンティティとして見なければならない場合にも使用できます。

セマンティック上は、<<noScope>> ステレオタイプのパッケージは、[モデル ビュー] の他のパッケージと同じ可視性を持ち、別のファイルに格納するという点でも他のパッケージと同じように機能します。セマンティックの観点では、<<noScope>> パッケージのすべての要素は包含するパッケージの一部と見なされます。修飾子を使用して <<noScope>> パッケージ内の要素を参照する場合、<<noScope>> パッケージの名前は修飾子の一部として使用できません。<<noScope>> ステレオタイプは、修飾子なしで、すべての定義をパッケージ外部でも見えるようにします。また、明示的な修飾子を使用して曖昧なケースを解決することもできます。

例 **21:** <<noScope>> パッケージ

```
package A {
    <<noScope>> package B {
        class C {

        }
    }
    C c; // <<noScope>> makes C visible
}

package A {
    <<noScope>> package B {
        class C {

        }
    }
    class C { }
```

```
    C c; // class A::C hides class B::C
}

package A {
    <<noScope>> package B1 {
        class C {

        }
    }
    <<noScope>> package B2 {
        class C {

        }
    }
    B1::C c;
/* 'C' is an ambiguous name. B1::C or B2::C must be used. If
C is used without qualifier there will be name resolution
errors. */
}
```

注記

«noScope» ステレオタイプは、通常はコード生成向けの形式モデルではサポートされません。C コードジェネレータは、以下の制約の下にこのステレオタイプを部分的にサポートします。
- «noScope» パッケージは、外向きまたは内向きの依存をもてません。このパッケージは直近の自分を含むパッケージに関係付けられる必要があります。
- «noScope» パッケージの名前は修飾子として使用できません。
- «noScope» パッケージは最上位のパッケージにはなり得ません。
セマンティックチェッカはこれらの制約違反を検知しません。

<<openNamespace>> パッケージ

複数のパッケージをまとめて 1 つのパッケージとして定義し、なおかつ含まれるパッケージの数を段階的に増やすことができると便利な場合があります。この場合、あるセッションでどのサブパッケージをロードしているかによって、グループ化しているパッケージの内容は変化します。

これを実現するには、Tau では <<openNamespace>> パッケージを使用します。動作シナリオは以下のとおりです。同じスコープ内で 2 つのパッケージを、たとえばモデルルートとして、定義します。パッケージに同じ名前を付け、両方とも <<openNamespace>> ステレオタイプにします。セマンティック上は、これでパッケージの内容はマージされます。この指定で、一方のパッケージの要素からもう一方のパッケージの要素を修飾子なしで直接使用でき、名前はマージされるすべてのパッケージ内で一意となる必要が生じます。

ネストされた <<openNamespace>> パッケージの階層を持つことができます。したがって、たとえば 1 つのファイルに格納された、<<openNamespace>> Sub を含む <<openNamespace>> Top がある場合、<<openNamespace>> Sub を持つ <<openNamespace>> Top を含む別のファイルを持つことができます。これらのファイルを両方とも同じプロジェクトにロードすると、Top の内容と Sub の内容がマージされます。

<<openNamespace>> パッケージを使用する最も重要なシナリオは、別に管理されているパッケージ階層のベース バージョンがあり、それを特定のアプリケーションで使用する際にサブパッケージなどで拡張したい場合です。

クラス モデリング

クラス モデリングは、設計中のシステムを構成するオブジェクトの種類を特定するプロセスです。この作業は、多くの場合、設計フェーズの初期や分析フェーズで行いますが、通常は、ユース ケースおよび／またはシナリオ モデリングを通じて、設計対象システムを構成するオブジェクトが特定された後に行います。他のオブジェクトと同じプロパティ、振る舞い、関係を持つと考えられるオブジェクトは、1 つのグループにまとめられ、オブジェクトの「クラス」としてモデリングされます。

クラス モデリング作業には、クラスを同定する作業の他に、これらのクラスを定義する作業も含まれます。この作業には、クラス図を使用します。同定されたクラスごとに、以下の質問に対する回答を検討します。

クラスに構造があるか。
クラスのインスタンスにはどのパートが含まれているか。

クラスの構造は、属性や、汎化や関連などの関係を使用してクラス図に記述されます。クラスがどのように構成されているかを示すためには、合成構造図も使用できます。

クラスに振る舞いがあるか。
どの操作が可能か。

クラスの振る舞いはクラスに対する操作と見なされ、これらの操作のシグニチャをクラス図に記述します。シグナル、タイマー、状態機械など、クラスの他の振る舞い特性に関しても同じことが当てはまります。

クラスと他の要素の間にどのような関係があるか。

クラスには、他のクラスとの関係に加え、インターフェイス、データ型、選択などとの関係もあります。クラス モデリングでの使用方法の詳細は、312 ページの「UML の関係」セクションを参照してください。

クラスはアクティブかパッシブか。

単純に言うと、アクティブ クラスは動的なイベント起動の振る舞いを定義し、パッシブ クラスは情報を処理します。アクティブクラスのインスタンスは、イベントをディスパッチできます。

クラスが環境に露出している通信ポートはどれか。

クラスのポートは、クラス図で可視化されます。

クラス図

クラス図はモデルを静的に表示するもので、モデル内のオブジェクトのタイプを記述するために使用します。これらのタイプは通常はクラスですが、基本、列挙、インターフェイス、選択、シンタイプなど他の分類子の場合もあります。クラス図により、タイプ間の関係と、その構造特性や振る舞い特性も示されます。

クラス図に表示される定義は、デフォルトでは、ダイアグラムを持つスコープ（クラスやパッケージなど）に含まれますが、別のスコープの定義を表示することもできます。

パッケージ図を、システムのパッケージやその相互依存関係の記述手段として使用する方法については 209 ページの「パッケージ モデリング」を参照してください。ただし、同じ情報をクラス図で記述することもできます。

クラス図の例

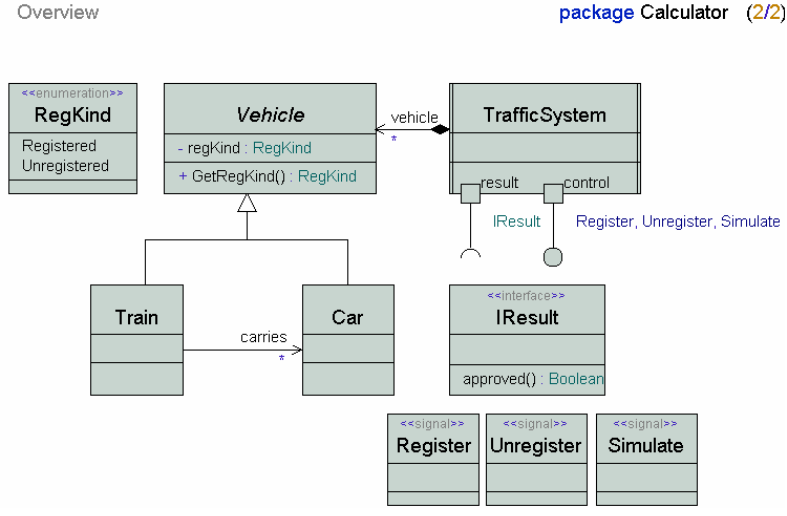


図 79: クラス図

クラス図のモデル要素

クラス図には、以下のモデル要素が使用されます。

- アーティファクト
- コラボレーション
- クラス
  - アクティブ クラス

- 属性
- 操作
- ポート
- インターフェイス
  - 実現化インターフェイス
  - 要求インターフェイス
- シグナル
- シグナル リスト
- タイマー
- データ型
- 選択
- シンタイプ
- 状態機械
- 関係

参照

パッケージ図

クラス

クラスは、同じプロパティ（属性）、振る舞い（操作）、構造、関係を持つオブジェクトのグループの抽象化です。クラスは、（抽象と定義されていなければ）複数のインスタンスにインスタンス化できます。それらはすべて同じプロパティを持ちます。

シンボル

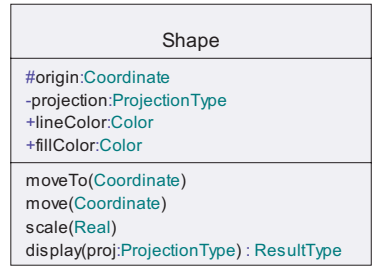


図 80: 属性と操作を持つクラス

クラスのインスタンスがそれ自体の実行スレッドを維持する（他のインスタンスと並行して実行される）場合、クラスはアクティブ クラスであると言います。そうでなければ、インスタンスは別のアクティブ インスタンスのスレッドで実行され、この場合、クラスはパッシブであると言います。

クラスをアクティブにするには、以下のいずれかを行います。

- ダイアグラム（または［モデル ビュー］）で、アクティブにしたいクラスを右クリックして、ショートカット メニューの［アクティブ］をクリックします。
- クラスのプロパティ エディタを表示して、［Active］を選択します。

アクティブ クラスは、ダイアグラム内で、外枠の縦が二重線で表示されます。

クラスには、内部通信から見た構造を記述する合成構造図（旧アーキテクチャ図）にパートとコネクタで可視化された内部構造を持たせることもできます。また、ランタイム実行の観点で記述する状態機械（initialize またはクラスと同じ名前で呼ばれる）を使用することもできます。この状態機械は、アクティブ クラスのインスタンスの生成時に実行される予定の「メイン」の振る舞いです。

また、クラスには一連のポートも使用できます。ポートは、クラスのアーキテクチャ記述で、クラスのインスタンスがどのように他のインスタンスと接続できるかを指定します。ポートは、複数の関係者に露出されている一連のインターフェイスのグループ化にも使用されます。

モデルにクラスを追加するには、複数の方法があります。

- ワークスペース ウィンドウの［モデル ビュー］にクラスを直接追加します。クラスが常駐するスコープを選択して、ショートカット メニューから［新規］、［クラス図］を選択します。
- クラス図にクラスを描画します。クラス図を作成して開き、ツールバーから［クラス］シンボルを選択してダイアグラム内に配置します。
- クラスのテキスト定義はテキスト シンボル、または、テキスト図に挿入します。
- 合成構造図で、タイプ名のないバインドされていないパートをダブルクリックします。この操作で、該当するパートを表示するダイアグラムを新たに作成できます。このダイアグラムは、パートに対して作成されたインライン クラスに属します。作成可能なダイアグラムは、クラス図、合成構造図、状態機械図、ユースケース図です。

アクティブ クラスの複数状態機械

アクティブ クラスに任意数の状態機械を挿入できます。ただし、以下の点を考慮に入れる必要があります。

- 状態機械の 1 つに **initialize** の名前が付いている場合、またはクラスと同じ名前が付いている場合、この状態機械はクラスのメイン 状態機械と見なされます。クラスのインスタンスの作成時、この状態機械が実行されます。この状態機械を省略すると、ビルド時にスタートとストップ シンボルが状態機械図に自動挿入されます。
- アクティブ クラスの他の状態機械を実行する場合は、明示的に呼び出す必要があります。

構文

クラスシンボルには、以下に示すように、編集可能なテキスト フィールドのある入力領域があります。

- クラス ヘッダー（必須）
- 属性（オプション）
- 操作（オプション）
- 制約区画（オプション）
- ステレオタイプ インスタンス区画（オプション）

クラス ヘッダー

以下の例に、いくつかのタイプのクラス ヘッダーを示します。

例 **22:** クラス ヘッダー

単純なクラス：

```
myClass
```

仮想性を含むクラス：

```
redefined myC
```

テンプレート パラメータを使用するクラス：

```
MyParamClass < type T, Integer c >
```

属性

例 **23:** クラスと属性

属性を持つクラス：

```
public A :Integer = 4
```

多重度を持つ属性：

```
A:Integer [10]
B:Integer [3,>15]
C:Integer [*]
```

操作

例 **24:** シグナル

```
signal s (Integer, Real)
```

例 **25:** メソッドの例

```
private m( x:Integer) :Integer
```

抽象クラス

クラスは「抽象」である場合があります。抽象クラスのインスタンスは生成できません。したがって、このクラスをインスタンス化するには特化する必要があります。

クラスが抽象であれば、クラスの名前は、クラス シンボルで*イタリック*表記されます。

クラスを抽象にするには、以下のいずれかを行います。

- ダイアグラム（または［モデル ビュー］）で、抽象にしたいクラスを右クリックして、ショートカット メニューの［抽象］をクリックします。
- クラスのプロパティ エディタを表示して、［Abstract］を選択します。

仮想性

仮想性は、クラスが再定義可能かどうかを決めるものです。これは、クラスが別のクラスに包含されている場合のみ有効です。

可視性

属性や操作といったクラスの特性の可視性は、定義されているクラスの外からアクセスできるかどうかを定義するものです。

- **なし**

  特性に定義されている可視性はありません。
- **Public**

  特性は、包含されているクラスが見える場所であればとこからでも参照できます。
- **Protected**

  特性は、特性を定義するクラスのどの子孫（特化による）からでも参照できます。
- **Private**

  private 特性を定義するクラスのみが使用できる特性です。
- **Package**

  特性は、包含されているクラスが見える、包含する最も近いパッケージ内の場所であればとこからでも参照できます。

可視性の詳細については、可視性を参照してください。

外部クラス

クラスを外部として定義するには、以下の手順を行います。

- クラスのプロパティ エディタを表示して、[External] を選択します。外部プロパティは、プロパティ エディタにのみ表示されます。

クラスとコンポーネント

コンポーネントとして抽象を表現する特定の概念はありませんが、抽象は他の方法でモデリングできます。

UML では、クラスとコンポーネントはよく似ています。コンポーネントは、メタモデルのクラスのサブクラスです。クラスにもコンポーネントにも、属性、操作、合成構造（合成構造図に表されるもの）、ポート、インターフェイスなどがあります。コンポーネントの主な目的は、あるエンティティを表現する用語を提供すること、そして、コンポーネント ベースのモデリングで最も重要な機能性を強調することです。これには、コンポーネントの実現化を表す能力や、コンポーネントの要求インターフェイスや提供インターフェイスを指定する能力も含まれます。通常、提供インターフェイスは実現化分類子のいずれかによって実現化されます。

制約区画

[制約区画の追加] ショートカット メニュー項目を使用して、1 つまたは複数の制約区画をクラス シンボルに追加できます。制約区画はインターフェイスシンボルやステレオタイプシンボルなど、他のクラス的な シンボルにも追加できます。

制約区画は、クラス シンボルの最後の通常の表示可能入力領域の下に置かれます。

制約区画は、制約シンボルに似ています。1 つの読み取り専用 “{}” テキスト ラベルと、編集可能なメイン テキスト ラベルがあります。

ショートカットコマンド [制約を区画として表示] を使用して、クラス シンボルの下に制約区画がないクラス シンボルに関連付けられたモデル要素に適用されている制約ごとに、1 つの制約区画を作成して追加します。ショートカットコマンド [制約をシンボルとして表示] を使用して、クラスシンボルに関連付けられたモデル要素に適用されている制約ごとに、1 つの制約シンボルを作成して追加します。

ステレオタイプ インスタンス区画

[ステレオタイプインスタンス区画の追加] ショートカット メニュー項目を使用して、1 つまたは複数のステレオタイプ インスタンス区画をクラス シンボルに追加できます。ステレオタイプ インスタンス区画は、インターフェイスシンボルやステレオタイプシンボルなど、他のクラス的な シンボルにも追加できます。

ステレオタイプ インスタンス区画は、クラス シンボルの最後の通常の表示可能区画の下に置かれます。

ステレオタイプ インスタンス区画は、ステレオタイプ インスタンス シンボルに似ています。1 つの読み取り専用 ”<<>>” テキスト ラベルと、編集可能なメイン テキスト ラベルがあります。

クラス シンボルのショートカット コマンド［ステレオタイプを区画として表示］を使用して、クラス シンボルの下にステレオタイプ インスタンス区画がないクラス シンボルに関連付けられたモデル要素に適用されているステレオタイプ インスタンスごとに、1 つのステレオタイプ インスタンス区画を作成して追加します。ショートカットコマンド［ステレオタイプをシンボルとして表示］を使用して、クラスシンボルに関連付けられたモデル要素に適用されているステレオタイプ インスタンスごとに、1 つのステレオタイプ インスタンス シンボルを作成して追加します。

参照

データ型

選択

コラボレーション

コラボレーション シンボルは、アイコン モードのサポートも含め、クラス シンボルと同じように振る舞いますが、コラボレーション シンボルには属性と操作が表示されません。

属性

属性は、実行時に 1 つまたは複数の値を持つ構造特性です。

属性は、UML 言語の、関連する複数の構成要素をモデリングするために使用します。

- **属性**

  構造化分類子の属性は、属性としてモデリングされます。このような属性のインスタンスは、フィールドと呼ばれ、フィールド式を使用して参照できます。また、クラス スコープ属性 (いわゆる「静的属性」) というものもあります。あるクラスのすべてのインスタンスは、この属性について同じ値を共有します。

  合成構造図では、合成属性はパートととして参照されることがあります。これは、合成構造図がクラスの階層構造を示すものであるためです。

  属性は、関連の端を表すのにも使用します。

- **ローカル変数**

  状態機械、操作、合成文のローカル変数は、属性としてモデリングされます。このような属性は、名前で直接、必要に応じてスコープ分類子で修飾して参照できます。

- **定数**

  定数は、読み取り専用属性としてモデリングされます。定数の値は、属性のデフォルト値です。通常、定数はパッケージ レベルで定義されますが、属性の定義できる場所であればどこでも定義できます。定数は、名前で直接参照されますが、必要に応じてスコープ分類子で修飾して参照することもできます。定数の値は、いったん設定すると変更できません。

1 つの属性は、必ず 1 つの静的なタイプをもちます。タイプは、属性の定義の時点で決定され、以下のいずれかになります :

- クラス

- インターフェイス
- 基本型または列挙型
- シンタイプ
- デリゲート
- 選択

属性は、関連と密接な関係があります。誘導可能な関連の端と属性は、実質的には同じものです。つまり、まず属性を定義し、次にこの属性を、誘導可能な関連の端の役割名としてクラス図に可視化できる、ということです。逆ももちろん可能です。まず、誘導可能な関連の端が 1 つある関連を定義します。次に、関連の端を、クラス シンボルの属性入力領域に属性として可視化します。

呼び出しをするために、特定の関連の端や関連付けられている属性を使用する場合は、誘導可能でなければなりません。

例 **26:** 誘導可能性 ——————————————————————————

A と B というクラスがあるとします。操作 B.op() をクラス A から起動しようとしているものとします。

関連の端の名前（役割名）が b で、向きが A から B の関連がある場合、関連が誘導可能である場合のみ、b.op(); を呼び出すことができます。

——————————————————————————————————

属性は、合成構造図でシンボルとして可視化できます。これは、包含クラスのすべての属性について可能ですが、通常はパートのためにのみ使用されています。

集約の種類

属性のタイプがクラスの場合、この属性の値はオブジェクト、つまりクラスのインスタンスであることになります。この場合、属性には、属性を包含するクラスのインスタンスと値のインスタンスの間のライフタイム関係を決定する集約の種類を複数持たせることができます。

- **なし**

  2 つのクラスのインスタンスの間にライフタイム依存はありません。つまり、属性には、値クラスのインスタンスの参照が 1 つまたは複数あることになります。
- **共有集約**

  2 つのクラスのインスタンスの間にライフタイム依存はありません。しかし、非形式的には、一方がもう一方に「所有されている」と見なされます。属性入力領域では、共有集約は、shared a:myclass などのように、属性名の前に shared というキーワードを付けることで示されます。コード ジェネレータには、共有集約に特定のセマンティックを追加するものがありますが、実際には、セマンティックが弱いためにほとんど使用されません。通常は、集約のない関連を代わりに使用するほうがよいでしょう。

- **合成**

  包含するクラスのインスタンスと値クラスのインスタンスの間には、強力な一部／全体の関係があります。つまり、実際には、2 つのインスタンスの間にはライフタイム依存があるということになります。包含するインスタンスが終了すると、その中に包含されているインスタンスも終了します。合成は、part a:myclass などのように、属性名の前に part というキーワードを付けることで示されます。

注記

非静的属性には、定義コンテキストがインスタンス化された場合のみ値を持たせることができます。上述の、属性の定義コンテキストは、異なった方法でインスタンス化されます。たとえば、パッケージは使用時にインスタンス化され、イベント クラスは起動時にインスタンス化されます。

デフォルト値

属性には、式として指定したデフォルト値を持たせることができます。属性にデフォルト値がない場合は、値は、明示的に割り当てないかぎり、定義コンテキストのインスタンス化時に定義されません。

ポート

タイプがクラスの属性には、コネクタを接続できる通信ポートを持たせることができます。これらのコネクタは、シグナルを属性との間でやりとりする、システム内の通信経路を記述します。これは、属性がパートを表す場合に主に使用されます。

多重度

属性には、範囲の集合としてモデリングした多重度を持たせることができます。多重度は、属性がランタイム時に保持できるインスタンスの数を制限します。

属性の多重度が >1 であるかどうかにより、属性の実際のタイプが異なります。多重度が >1 の場合は、属性は値のリストを保持できるコンテナ タイプになります。多重度がちょうど 1（または `0..1`）の場合はそうなりません。

使用できるデータ型 ライブラリによって、コンテナ タイプも異なります。通常、コード ジェネレータが異なると、ターゲット言語と適切に統合されるよう、コンテナ タイプも異なります。特定のデータ型 ライブラリがロードされていない場合、付属の定義済みパッケージに、多重度 >1 の属性のタイプとして文字列タイプが使用されます。(文字列タイプは、整列リストまたはシーケンスを表す定義済みの集合 タイプです。リストの値は、属性のタイプに従わなければなりません。)

多重度は、クラス シンボルの属性入力領域で、次のように、属性のタイプの後に角かっこで囲んで示されます。

```
a :myClass [*]
```

上の例では、多重度は未接続です（アスタリスクで示します)。

つまり、値はいくつでも持つことができます。多重度が指定されていない場合、デフォルトで 1 と見なされます。

開始基数

多重度が >1 の合成属性の場合、式を使用してインスタンスの最初の数字を指定する省略表現があります。その数字により、所有側クラスのインスタンス化時に自動的に生成されるインスタンスの数を指定できます。インスタンスの最初の数字を省略すると、インスタンスは 1 つだけ生成されます。

注記

基数を解釈するかどうか、またその解釈の方法は、コード ジェネレータによって異なります。コード ジェネレータによっては、属性の基数を無視するものがあります。

属性が合成構造図でパート シンボルを使用して示されている場合は、開始基数を指定できます。ただし、このようなシンボルでは、構文は次のようになります。

```
a :myClass [*] / 2
```

ここで、a のインスタンスの開始基数は 2 です。

可視性

属性に可視性を指定できます。可視性は、**public**、**private**、**protected**、**package** のいずれかになります。

導出

属性を `derived` と宣言できます。これは、属性の値が対応するオブジェクトに格納されるのではなく、他の属性の値などから計算されることを示します。導出属性の構文では、次に示すように、属性名の前に / を付けます。

```
/a:myClass
```

導出属性のための導出規則の指定方法については、Derived（導出）を参照してください。

静的

静的属性は、インスタンス スコープでなくクラス スコープに所有されている属性です。これは、特定クラスのすべてのインスタンスに共有されている属性インスタンスが 1 つだけであるということです。

定数

定数属性は、値を動的に変更できない属性です。定数の値は、属性のデフォルト値です。

外部定数属性とは、値がモデル外で定義されるか、後で（ビルド タイムなど）定義されることを示します。

例 **27:** テキストによる定数宣言 ——————————————————————————

```
const Integer a = 10;
```

```
const Integer extern ext_const;
```

操作

操作は、クラスのインスタンスが、操作のシグニチャと一致する呼び出しを処理できるという宣言です。操作は、操作本体または状態機械で実装できます。この実装（メソッドと呼ばれることが多い）は、操作が起動すると実行されます。これは、受信側がパッシブ インスタンスであれば、操作起動後直ちに実装が実行され、受信側がアクティブ インスタンスであれば、実装の実行は遅れ、後でインスタンスが操作コールを受け付けられる状態になったときに実行される場合があるということです。

操作はテキスト シンボルまたはテキスト図で、クラス シンボルの操作入力領域で、および特殊操作シンボルを使用して、テキストによって宣言できます。

Tau は、操作の `derived` プロパティを標準 UML の拡張機能としてサポートしています。このプロパティを使用して、操作は実装を伴わないが暗黙的に計算されることを示すことができます。このプロパティは、分析専用です。生成されたコードには影響しません。

シンボル

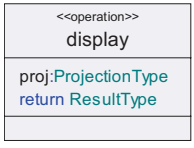


図 81: 操作

構文

シンボルには、操作ヘッダーとパラメータという 2 つの編集可能なテキスト フィールドがあります。下のフィールドは常に空白です。

アクティブ クラス

アクティブ クラスは、それ自体の制御スレッドがあるクラスです。アクティブというプロパティにより、通常のクラスと区別されます。描画的には、227 ページの図 82 に示すように、特殊なアクティブ クラス シンボルによって示されます。

シンボル

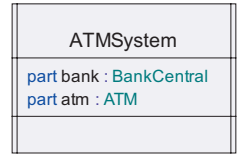


図 82: アクティブ クラス

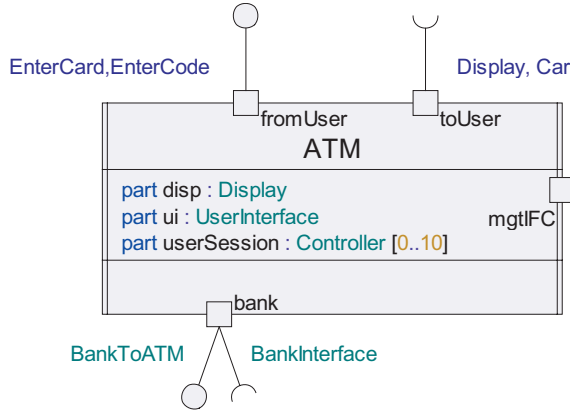


図 83: アクティブ クラスとポート、実現化インターフェイス、要求インターフェイス

クラスは、以下の方法によってアクティブにできます。

- クラスを選択し、ショートカット メニューの［アクティブ］を選択する
- クラスを選択し、プロパティ エディタで［アクティブ］を選択する

アクティブ クラスは、UML でリアルタイムの振る舞いをモデリングするための基本構成単位です。アクティブ クラスにより、モデルの構造（アーキテクチャ）と振る舞いの両方が定義されます。UML のアクティブ クラス 概念のこの二元性により、強力で柔軟な設計能力が得られます。

構造

アクティブ クラスの構造は、1 つまたは複数の合成構造図で定義され、アクティブ クラスは他のアクティブ クラスのインスタンスの集合として定義されます。これらのアクティブ クラスには構造を持たせることもでき、これで複雑なアーキテクチャの記述が可能になります。

振る舞い

アクティブ クラスの振る舞いは、1 つまたは複数の状態機械図の状態機械によって定義されます。状態機械には、`initialize()` と命名するか、クラスと同じ名前を付ける必要があります。

アクティブ クラスの仕様を完全に決定するには、構造定義か状態機械定義、またはその両方を持たせる必要があります。

アクティブ クラスにはそれ自体の制御フローがあり、振る舞いを開始することも、インターフェイスで見られる振る舞いにパッシブに反応することもできます。伝統主義者は、アクティブ クラスよりも *リアクティブ* クラスという名前を好みます。これは、通常このようなクラスはイベント駆動であるためです。振る舞いは、通常、タイマーを使用して開始されます。タイマーがタイムアウトすると、何らかの振る舞いが起動します。

アクティブ クラスに、合成構造図で定義された、包含されているパートがいくつかある場合、各パートは非同期にシステムの他のパートと並行して実行されます。このセマンティックにより、モデルが分散物理環境に配置でき、共有メモリ アクセスによる単一プロセッサでの実行に依存しないようにできます。

アクティブ クラスは、ポートによってインターフェイスの実現化と要求ができます。ポートは、要求インターフェイスや実現化インターフェイスとともに、アクティブ クラスとその環境の間の静的規約を定義します。

属性と操作

アクティブ クラス シンボルでは、クラスの属性をシンボルの第 2 入力領域で、操作を第 3 入力領域で指定または表示できます。

注記

public 属性と操作がクラス シンボルに示されていても、クラスの外からアクセスできるようにするには、クラスで実現化されたインターフェイスの一部として宣言する必要があります。このため、アクティブ クラスの属性と操作の入力領域の使用は実用上あまり便利でなく、主として、インターフェイスの仕様が完全に決定される前の分析フェーズで使用されます。

参照

属性

操作

ポート

ポートは、アクティブ クラスの相互作用点で名前のあるものです。ポートにより、実装されたインターフェイス（実現化インターフェイス）と、必要な別のクラスのインターフェイス（要求インターフェイス）が指定されます。

ポートは通常、アクティブ クラスのみで使用します。作成済みのポートをアクティブ クラス シンボルまたはパート シンボルで可視化するには、ショートカット メニューの［表示／非表示］から［ポートの表示］コマンドを選択します。

シンボル

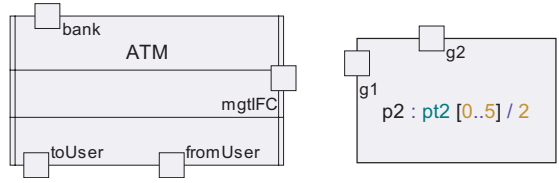


図 84: クラスのポートとパートのポート

シンボルには、名前の入るテキスト フィールドが 1 つあります。

ヒント

ポート シンボルを追加する最も簡単な方法としては、まずポート シンボルを置くシンボルのフレームを選択し、ツールバーのポート シンボルをクリックします。

ポートタイプ

シンボルには、名前の入るテキスト フィールドが 1 つあります。このフィールドには任意に **タイプ**を入れることができます。ポートのタイプは、主に分析フェーズで使用するために入力します。

注記

Tau のコード ジェネレータではポート タイプを考慮しません。その代わりに、ポートの実現化インターフェイスと要求インターフェイス用に与えられた情報に基づいてコードが生成されます。

振る舞いポート

ポートには、振る舞いポートと非振る舞いポートの 2 種類があります。これらの違いは、振る舞いポートがクラスの状態機械と直接関連付けられているのに対し、非振る舞いポートはコネクタを使用して接続する必要があり、通常はクラス外からの通信をクラスの内部パートの一部に中継するだけのものであるという点です。

振る舞いポートは、クラスの状態機械と直接接続されているポートです。このポートに送られるシグナルはすべてクラス自体の振る舞いによって処理されます。

ポートとインターフェイス

ポートごとに、実現化インターフェイスと要求インターフェイスを指定できます。ポートの実現化インターフェイスにより、ポートを通じて処理できる着信要求が定義されます。要求インターフェイスにより、1 つまたは複数のコネクタを通じ外部からポートに接続されたクラスが処理しなければならない発信要求が定義されます。227 ページの図 83 に、実現化インターフェイスと要求インターフェイスのあるポートの例を示します。

アクティブ クラスの構造または振る舞いを定義する際、ポートは、この目的で使用されるダイアグラム（合成構造図または状態機械図）の境界で宣言できます。ポートはパートからも参照できます。この場合、パート シンボルの境界に示されます。

また、アドレッシング メカニズムとして、（追加されたコネクタのもう一方の端にある受信側に関する知識なく）ポートを通じて、状態機械からメッセージを送ることもできます。

ポートの実現化インターフェイス（または要求インターフェイス）には、通常、インターフェイスの参照に加えて、シグナル リスト、シグナル、属性の参照が含まれています。

ポートの実現化インターフェイスと要求インターフェイスは、実現化インターフェイス シンボルと要求インターフェイス シンボルをポートに追加することによって可視化されます。これらのシンボルでは、サポートされているインターフェイスや必要なインターフェイスの名前（またはシグナル リスト、シグナル、属性）が指定できます。

実現化インターフェイスや要求インターフェイスを指定する方法として他には、［プロパティ］ダイアログによるものがあります。

ポートによって、以下のものが表されます。

- インターフェイスとクラスの接続点
- これらのクラスを別のインスタンス、または囲む側のフレーム シンボルと接続する、合成構造図の接続ラインの接続点

ポート シンボルは、以下の場所に使用できます。

- クラス シンボル
- パート シンボル
- 振る舞いシンボル
- アクティブ クラスが所有している状態機械のフレーム
- 合成構造図のフレーム
- アーキテクチャ図や状態機械図の中（ポートがこれらの図のフレームに置かれた場合と同じセマンティックを持つ）

ポートには、explicit コネクタも implicit コネクタも持たせることができます。各ポート シンボルに追加できるインターフェイス シンボルの数は、0、1、2 のいずれかです。

インターフェイス シンボルが 2 つある場合、一方を、ポートへの着信インターフェイス（またはシグナル）を指定する実現化インターフェイス シンボルとして定義し、もう一方を、ポートからの発信インターフェイスを指定する要求インターフェイス シンボルとして定義しなければなりません。

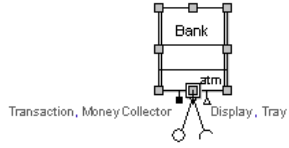


図 85: 実現化インターフェイスと要求インターフェイスのあるポート

インターフェイスのある、または、インターフェイスのないポートは、以下のいずれかの方法で、直接クラスに描画できます。

- クラスを選択して、Shift キーを押しながらツールバーの［ポート］シンボルをクリックします。新しいポート名を入力します。ポートはクラスの左枠の上左隅に近い位置に配置されます。
- ツールバーの［ポート］シンボルをクリックし、ポートの配置先のクラスをクリックします。名前テキスト フィールドを編集します。

継承

スーパータイプに所属するポートのあるクラス間の汎化の場合、これらのポートも継承されます。

ポートは、public または private と宣言することにより、ポートが外部に露出されるか、内部のみで使用されるかを区別できます。サブクラスのポートにシグナルを追加できます。

インターフェイス

インターフェイスは、インスタンス化できない構造化分類子です。インターフェイスを実装するクラスが実装する必要のある一連の属性、操作、シグナルをグループ化するために使用します。クラスがインターフェイスを実装することを、インターフェイスを**実現化**する、と言います。これで、インターフェイスで宣言された操作がサポートされます。クラスがインターフェイスを**要求**することもできます。この場合、クラスは、操作を実行するために他のアクティブ クラスに依存します。

シンボル

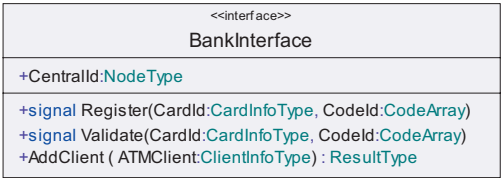


図 86: インターフェイス シンボル

インターフェイスの操作は、通常、インターフェイスを実現化するクラスによって提供されるサービスを記述します。当然、クラスは複数のインターフェイスを実現化できます。

操作の他に、インターフェイスにはシグナルや属性を持たせることもできます。

インターフェイスは特化が可能で、テンプレート パラメータを持たせることができます。インターフェイスの多重継承は、アクティブ クラスの通信インターフェイスの定義に便利な仕組みです。

インターフェイスは、関連インターフェイスを実現化するクラス間のプロトコルや規定を定義するために、互いに関連付けることもできます。MgmI インターフェイスと MgmReplyI インターフェイスを定義する例を 232 ページの図 87 に示します。2 つのインターフェイスを関連付けることにより、インターフェイス間に関係ができます。これで、一方のインターフェイスがポートなどで参照されたり、コネクタと関連付けられたりすると、もう一方のインターフェイスが自動的に反対方向に挿入されるようになります。したがって、あるポートを通じてクラスが MgmI インターフェイスを実現化した場合、MgmReplyI インターフェイスは自動的に同じポートの要求インターフェイスになります。

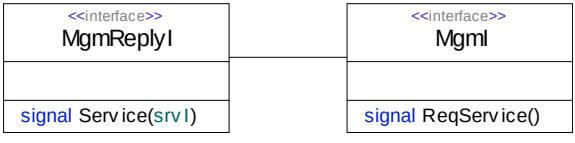


図 87: 2 つの関連インターフェイスを使用して定義された規定

構文

シンボルには、以下に示すように、編集可能なテキスト フィールドが 3 つあります。

- ヘッダー
- 属性

- 操作

ヘッダー フィールドは、インターフェイスの名前の定義に使用します。

属性フィールドには、インターフェイスを実現化するクラスが実装しなければならない属性の定義が入ります。通常、これは、実現化するクラスの protected 属性への getter 操作と setter 操作の省略表現です。

操作フィールドには、インターフェイスを実現化するクラスが処理しなければならない操作とシグナルの定義が入ります。

参照

実現化インターフェイス

要求インターフェイス

Pid

実現化インターフェイス

クラスのポートに追加された実現化インターフェイスは、クラスがそのポートによって実現化するインターフェイスを可視化します。インターフェイス、シグナル、シグナル リスト、属性をテキスト フィールドで指定できます。

シンボル

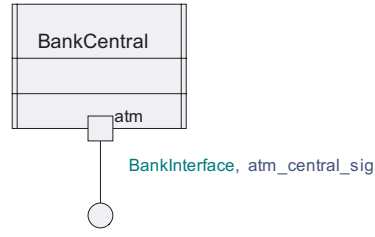


図 88: 実現化インターフェイス

構文

シンボルには、テキスト フィールドが 1 つあります。

例 **28:** 実現化インターフェイス

```
S, p, SigList
```

参照

インターフェイス

要求インターフェイス

Pid

要求インターフェイス

クラスのポートに追加された要求インターフェイスは、クラスがそのポートによって処理されるものと見なす要求を可視化します。インターフェイス、シグナル、シグナル リスト、属性をテキスト フィールドで指定できます。

シンボル

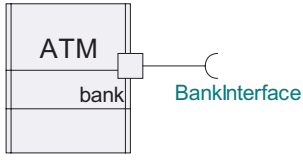


図 89: 要求インターフェイス

構文

シンボルには、テキスト フィールドが 1 つあります。

例 **29:** 要求インターフェイス

```
S, p, SigList
```

参照

インターフェイス

実現化インターフェイス

Pid

シグナル

シグナルは、UML における通信の主要手段の 1 つです。シグナルは、アクティブ クラス間で送信される非同期メッセージを表します。シグナルにより、データを伝送できます。データは、シグナルの宣言パラメータ型に一致しなければなりません。

シグナルの最も便利な宣言の方法として、インターフェイスを実現化するクラスの能力を表すインターフェイスで他のシグナル、操作、属性とともに宣言します。

ただし、235 ページの図 90 に示すように、クラス シンボルに類似した特殊シグナル シンボルの使用により、スタンドアロン シグナル宣言も可能です。

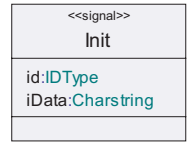


図 90 シグナル

異なるシグナルを多数使用する場合は、テキスト シンボルを使用してシグナルをテキストで宣言したほうが実用的なことが多いでしょう。

例 **30:** テキストによるシグナル宣言

```
signal Init (IDType id, Charstring iData);
signal SetupReq, SetupInd, AbortReq, AbortInd;
signal ForwardedMsg (IDType, MsgData);
```

構文

シグナル シンボルには、以下に示すように、編集可能なテキスト フィールドが 2 つあります。

- ヘッダー
- パラメータ

ヘッダー フィールドではシグナルの名前を宣言し、パラメータフィールドではシグナルのパラメータを宣言します。パラメータの名前は省略してもかまいませんが、パラメータ タイプは必須です。

多くのクラス的シンボルに設けられている 3 つ目の入力領域は、シグナル シンボルの場合は常に空白です。

参照

メッセージ

シグナル リスト

インターフェイス

タイマー

None

シグナル リスト

`signallist` キーワードは、記述をわかりやすくするために、関連するシグナルのグループを表すために使用されます。通常、ポートやコネクタに使用されます。

例 **31:** シグナル リスト宣言

```
signallist MgtSignals = MOGetStatus, MOSet, MOReset;
```

注記

インターフェイスを使用したシグナルのグループ化は、シグナル リストよりも構造化されたアプローチです。これは、インターフェイスがシグナル宣言をカプセル化するためです。

参照

シグナル

インターフェイス

タイマー

タイマーは、シグナルと同様に、遷移のトリガとなるイベントです。タイマーはアクティブクラスが実行している実装コードによって設定され、タイムアウト時に、タイマー イベントを同じアクティブクラスインスタンスの状態機械で受信できます。時間値はアクティブ タイマーに関連付けられ、タイムアウトの時間となります。

シンボル

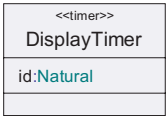


図 91: タイマー

シグナルと同様、タイマーにもパラメータを持たせることができます。これを利用して、すでにアクティブになっているタイマーをリセットせずに、同じ種類のタイマーを複数設定できるようにできます。つまり、パラメータの異なる複数のタイマーを同時にアクティブにできます。

構文

タイマーは、次に示すように、テキスト シンボルを使用してテキストで宣言することもできます。

例 **32:** テキストによるタイマー宣言

```
timer DisplayTimer (Natural id) = 2;
timer BankTimer () = BankTimeout;
timer UserTimer ();
```

タイマーをテキストで宣言する場合、タイマーにデフォルト持続時間、つまり、タイムアウトまでの時間を与えることもできます。これで、時間を指定せずにタイマーを設定できます。

参照

タイマー ハンドリングと時間

タイマー設定アクション

タイマー リセット アクション

タイマー設定

タイマー リセット

タイマー タイムアウト

データ型

データ型は、以下の 2 つ目的で使用されます。

- 使用できる基本タイプの記述
- ユーザー定義列挙タイプの記述

基本タイプは、多くの場合、特定の UML プロファイル（スタンドアロン プロファイルまたは特定のコード ジェネレータとともに使用するために定義されたプロファイル）に付随するモデル ライブラリで定義されます。後者の場合は、通常、データ型によってターゲット言語の基本タイプが定義され、UML モデルで使用できるようになります。

基本データ型をユーザー モデルで定義することもできますが、コード生成に問題が生じる場合があります。

列挙は、単にリテラルのリストとして値を列挙することによって一連の値を定義します。

いずれの場合も、データ型には、**操作**によって定義された振る舞いを任意で持たせることもできます。

シンボル

図 92: 列挙データ型

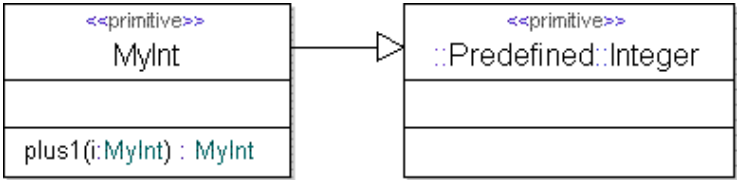


図 93: 演算子のあるデータ型

列挙データ型

列挙データ型は、リテラル値が論理名であるようなデータ型です。論理名は、オプションで、単純式で指定された複合的な値に追加できます。

使用できるデフォルト演算は以下のとおりです。

- 等価（==, !=）
- 関係演算（<, >, <=, >=）
- 代入（=）

例 **33:** 列挙データ型

```
enum UKColors { blue, red, white }

enum LinePrinterState {
  outOfService = 1,
  inServiceFree =2,
  inServiceBusy = 6
}

void op() {
  LinePrinterState e;
  Integer i;
  e = cast<LinePrinterState>(1);
  e = inServiceFree;
        i = cast<Integer>(e);
```

```
}
```

注記

238 ページの例 33 の操作 op のように cast 演算を使用して、整数と列挙タイプの間で変換を行うことができます。

基本データ型

基本データ型は通常、プロファイルのモデル ライブラリで定義されますが、ユーザー定義も可能です。ただし、ユーザー定義の基本タイプにはリテラル構文がありません。したがって、実際にはあまり有用ではありません。

データ型を別の既存データ型と関係付ける方法には以下の 2 つがあります。

- コピー コンストラクタを使用する
- 継承を使用する

いずれの場合も、既存データ型のリテラル構文が使用されます。UML に新規基本データ型を導入する方法として、コピー コンストラクタ メカニズムを推奨します。ほとんどのモデル ライブラリで使用されているのがこの仕組みです。

注記

基本データ型は通常、コード ジェネレータで特別な処理が必要です。ユーザー定義基本データ型は、コード ジェネレータのドキュメントで特に言及されていなければ、コード ジェネレータでは機能しません。

例 **34:** 演算子のあるデータ型

```
datatype simpleInt {
  simpleInt(Integer) {}
}
datatype myInt : Integer
{
  myInt plus1 ( myInt i) { return i+1;}
}
```

リテラル

リテラルは、列挙されたデータ型によって定義されたタイプの要素です。リテラルはそのデータ型に所有されます。リテラルの可視性は常に public です。

リテラルには、名前に加えて（名前はすべての定義にあります）、算術式での使用を可能にする整数値を持たせることができます。

選択

選択は、1 つの値を保持できるデータ型です。この値は、実行時はデータ型が複数でもかまいません。タイプの選択は、変数に値を割り当てる際に行われます。タイプ フィールド候補ごとに、フィールドの有無を確認する論理演算子 `IsPresent()` があります。

例 **35:** 選択

```
choice IntOrBool {
    Integer a;
    Boolean b;
}

IntorBool ib;
Integer i;
Boolean b=true;

ib.a=5;
i=ib.IsPresent("a")?ib.a:0; /* check if ib is Integer;
       if Integer, return ib,
       if not, return 0 */
ib.b=b;
```

例 **36:** 選択

```
choice IntOrBool {
  Integer a;
  Real r;
  Integer GetInt() {
    if (IsPresent("r")) {
      return 0;
    } else {
      return a;
    }
  }
}
```

`IsPresent()` 演算子の使用例

```
IntOrBool MyVar;
Real num_real;
Integer num_int;
MyVar.a=1;
if(IsPresent(MyVar,"a"))
  {
    num_int =MyVar.a;
    MyVar.r=3.14;
  }
  else
  {
    num_real=MyVar.r;
  }
if (MyVar.IsPresent("r")) {
switch (MyVar.r) {
  case 3.14 :
  {
    nextstate idle;
  }
  default :
  {
    nextstate idle;
  }
}
```

選択インスタンス値は、1 つの代入値（`choice_field = value`）のみを持つインスタンス式で指定できます。

例 **37:** 選択インスタンスの値

```
choice choice_type
{
  public Integer ifield;
  public Boolean bfield;
}
choice_type an_int = choice_type (. ifield = 1.);
```

シンタイプ

シンタイプは、別のデータ型（親 タイプ）に基づくデータ型です。2 つのタイプは、タイプ互換性とリテラルの点では同じです。シンタイプのリテラルは、親のリテラルと同一か、親のリテラルのサブセットです。シンタイプは、別のタイプのエイリアス（制約を付けることのできる）と見なすことができます。

例 **38:** シンタイプ

```
syntype myInt = Integer constants (> -10, != 0, <10);
syntype smallPrime = Natural constants (1,2,3,5,7);

Integer [1..10] myvar; /* inline syntype definition */
```

状態機械

状態機械の概念は、振る舞いモデリングセクションで詳しく説明しています。

ステレオタイプ

ステレオタイプの概念は、拡張性セクションで詳しく説明しています。

関係

クラス図では、以下の関係を使用できます。これらの詳細については、UML の関係セクションを参照してください。

- 関連
- 集約
- 合成
- 依存
- 拡張
- 汎化
- 実現化
- 表現

オブジェクトモデリング

クラスモデリングが設計しているアプリケーション中のオブジェクトの種類に対して焦点を当てているのに対して、オブジェクトモデリングはオブジェクトがどのように実行時に現れるのかについて関心を払うモデリングです。この分析作業で発せられる典型的な疑問は以下のようなものになるでしょう：

- **異なる時点でアプリケーションに存在するオブジェクトは何か？**
- **オブジェクトは属性値によってどのような見え方をするか？**
- **オブジェクト同士はどのように関連付けられるか？あるオブジェクトについての知識をもっているのはどのオブジェクトか？**

オブジェクトはインスタンスとも呼ばれます。したがって、この分析作業をインスタンスモデリングと呼ぶこともあります。

通常オブジェクトモデリングとクラスモデリングは並行に行われます。アプリケーションのオブジェクトが識別されると、それらはモデルの中で定義されます。この作業は、オブジェクトの種別を見つけ出す以前に行われることもあります。

実際のアプリケーションでは実行時に現れるオブジェクトの数はきわめて大きくなります。したがって、設計上注目すべきオブジェクトのみをモデルとして描いてゆく、というアプローチを取ること多くなります。たとえば、アプリケーションの初期化時の動作を理解するという目的のためには、そのアプリケーションの起動時に作成されるオブジェクトのみを検出してゆくというやり方になります。

オブジェクトモデリングでは、主として オブジェクト図を使用して、オブジェクトとその関係を定義してゆきます。同時にクラス図を使ってゆく場合もあります。

オブジェクト図

オブジェクト図は、特定の時点でアプリケーションに存在するオブジェクトのビュー（いわゆる「スナップショット」）を提供します。オブジェクト図に現れるオブジェクトは、名前が与えられ、場合によっては型（種別）が決まることもあります。「スロット（Slot）」と呼ばれるオブジェクトの属性値を与えることもできます。オブジェクト間の連結関係は、リンクラインで示されます。

名前の付けられたオブジェクトは、Tau では、**名前付きインスタンス**と呼ばれて、他の名前のないインスタンス、たとえば適用済みステレオタイプインスタンスなど、とは区別されます。名前付きインスタンス の定義は、デフォルトでオブジェクト図を含むスコープに配置されます。ただし、名前付きインスタンスを、［モデルビュー］からオブジェクト図にドラッグアンドドロップして、別のスコープで表示することも可能です。

オブジェクト図は、1 つの名前付きインスタンスを表している複数のインスタンスを含んでいる場合もあります。

オブジェクト図の例

下図のオブジェクト図は、212 ページの図 78 のクラス図で説明したアプリケーションで使用可能なオブジェクトの、スナップショットビューを示しています。

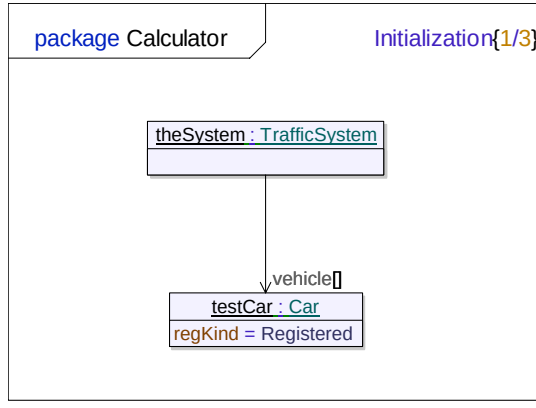


図 94: オブジェクト図

このオブジェクト図は、ある時点（ダイアグラム名から判断するにおそらく初期化時）でアプリケーションが TrafficSystem クラスのインスタンスを一つ含むということを、示しています。そのインスタンスは、vehicle リスト属性内の testCar と呼ばれる Car のインスタンスを保持します。testCar オブジェクトの regKind 属性は、値 Registered を持っています。

オブジェクト図のモデル要素

オブジェクト図では、以下のモデル要素を表現できます :

- 名前付きインスタンス
- スロット（Slot）
- 依存

参照

クラス図 .

名前付きインスタンス

名前付きインスタンスは、モデル化されたシステム内のオブジェクト（インスタンス）表し、そのオブジェクトについての完全または部分的な説明を付与します。オブジェクトは時とともに変化するので、名前付きインスタンスが提供するのは特定時点または特定の期間でのオブジェクトの情報です。UML オブジェクト図では、以下の点について形式的な表現ができないことに注意してください。

- オブジェクトと名前付きインスタンスとが合致する時点、期間

- 名前付きインスタンスがオブジェクトの完全な仕様を含むか、部分的な仕様しか含まないか

通常、名前付きインスタンスには名前があります。多くの場合子の名前は非形式的に解釈され、名前付きインスタンスで記述される実行時オブジェクトのどのプロパティにも対応しません。ただし、定義についての一般的な規則には従う必要があります。たとえば、同じスコープにある複数の名前付きインスタンスの名前は一意である必要があります。(スコープ、モデル要素、ダイアグラム参照)

通常、名前付きインスタンスには型があります。指定された型がクラスの場合は、名前付きインスタンスはそのクラスのオブジェクトを記述しています。指定された型がデータ型の場合は、名前付きインスタンスはそのデータ型の値を記述しています。操作、シグナルなどの振る舞い特性を型として指定することもできます。その場合、名前付きインスタンスはシステム内のイベントを記述することになります。たとえば、型が操作の場合は、名前付きインスタンスは操作呼び出しを記述し、型がシグナルの場合は、名前付きインスタンスそのシグナルのイベントを記述します。

名前付きインスタンスの型として、関連を使うこともできます。その場合、名前付きインスタンスは、リンクを表現します。

名前付きインスタンスについて抽象型を指定することもできます。これは記述される対象のオブジェクトが抽象オブジェクトであるということではなく、オブジェクトの表示されるプロパティがすべて抽象型のみであることを意味します。記述される実行時オブジェクトは、その抽象型の具体的なサブタイプになります。

名前付きインスタンス型が、クラス属性やシグナルパラメータのようなストラクチャフィーチャを持っている場合、名前付きインスタンスはそのストラクチャフィーチャに対して値を指定します。このような値の指定をスロット（Slot）と呼びます。

名前付きインスタンスは、オブジェクト図では **InstanceSymbol**. を使用して表示されます。

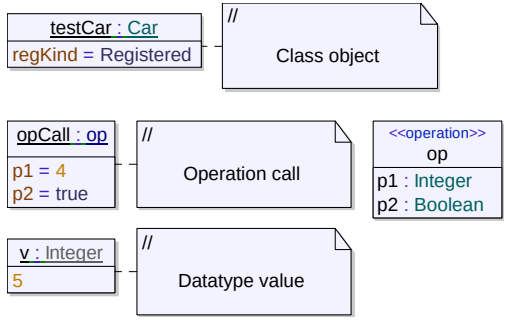


図 95: 名前付きインスタンスを定義しているインスタンスシンボル

上図のように、インスタンスシンボルは二つの基本区画を持っています。上の区画は、名前付きインスタンスの名前と型を保持します。下の区画はスロットを保持します。スロットの定義の構文は通常の割り当て文の構文と同じであることに注意してください（ストラクチャフィーチャは割り当てられた値です）。データ型値には、単純な値も使用できます。

注記

現在のセマンティックチェッカはデータ型値とデータ型の間の型互換性をチェックしません。したがって、オブジェクト図中のデータ型値は、非形式的なモデリング向けです。

リンク

リンクは、型が関連である名前付きインスタンスです。リンクは 2 つのオブジェクトの間の実行時の関係を図示します。プログラミング言語の言葉で言えば、リンクはポインタまたは参照に対応するといえるでしょう。

リンクはオブジェクト図では以下の 2 つの方法で表示されます :

1. 2 つのインスタンスシンボルを連結するリンクライン。
2. インスタンスシンボル内の通常のスロット。スロットの右手がターゲットの名前付きインスタンスを指します。

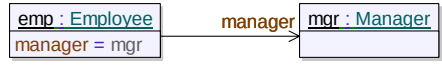


図 96: 2 通りのリンクの指定法

リンクラインのターゲット端に入力できるテキストは、式です。このテキストは、スロット（Slot）式の左辺です。

リンクの名前は、リンクラインの中央にあるラベルに入力して指定できます。

スロット（**Slot**）

スロットは、名前付きインスタンスの型に属するストラクチャフィーチャに対して値を指定する場所です。

スロットには、ある一般的なオブジェクトの値を示すという用途があります。名前付きインスタンスにスロットが定義されていないということは、必ずしも、対応するオブジェクトに何もストラクチャフィーチャがないということを意味しません。単に、その値がモデリング上興味の対象になっていない、ということにすぎません。

スロットは、ある特定の型のすべての種類のストラクチャフィーチャ（継承したフィーチャや public ではないフィーチャも含む）を参照する可能性があります。

スロットは、ストラクチャフィーチャ（左辺）に対する値（右辺）の割り当てになります。右辺は、単純な識別子であることが多いですが、より複雑な式を記述することもあります。以下のモデルを参照してください :

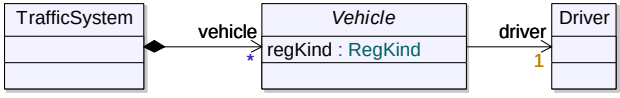


図 97: 関係のある 3 つのクラス

TrafficSystem の 1 つのインスタンスに定義されたスロットは、この例では、以下の表にあげた左辺をもつことができます。

スロットの左辺	意味
vehicle[]	`vehicle` コレクションの中の 1 つのインスタンス。コレクションのインデックス値は指定されていません。
vehicle[4]	`vehicle` コレクションの中の 1 つのインスタンスで、印でクス値 4 が指定されているもの。
vehicle[].driver	`vehicle` コレクションの中の 1 つのインスタンスの `driver` インスタンス。

最後の例がリンクラインとともに可視化されると、間接的なリンクはコンパクトな表記になります :

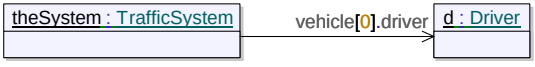


図 98: traffic system から最初の vehicle の driver へのリンクを可視化

自己参照

オブジェクトの自己参照の指定には、同等な 2 つの表記があります。スロットの右辺は、包含している名前付きインスタンスへの参照か、または、this キーワードです。

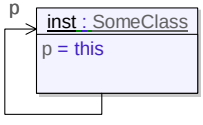


図 99: リンクラインのある自己参照のスロットラベル

アーキテクチャ モデリング

アーキテクチャ モデリングでは、アクティブ クラスの内部構造が、通信の観点から記述されます。これは、クラスの属性（この状況ではパートと呼ばれます）をコネクタと接続し、これらのコネクタで送ることのできるシグナルを指定することによって行います。このパートとコネクタの構造を、クラスのアーキテクチャまたはクラスの合成構造と呼びます。

アーキテクチャ モデリングは通常、設計フェーズでクラス モデリングと並行で、またはクラス モデリングの後に行います。

合成構造図

合成構造図（旧アーキテクチャ図）は、他のアクティブ クラスとの関連で、アクティブ クラスの内部ランタイム構造を定義します。これらの構成単位は、包含するクラスの合成パートである場合は、パートと呼ばれます。また、パートはアクティブ クラスのインスタンス化のみに限定されます。合成構造図で、パートの通信ポート間のコネクタを可視化することにより、アクティブ クラス内の通信を表現することもできます。

例

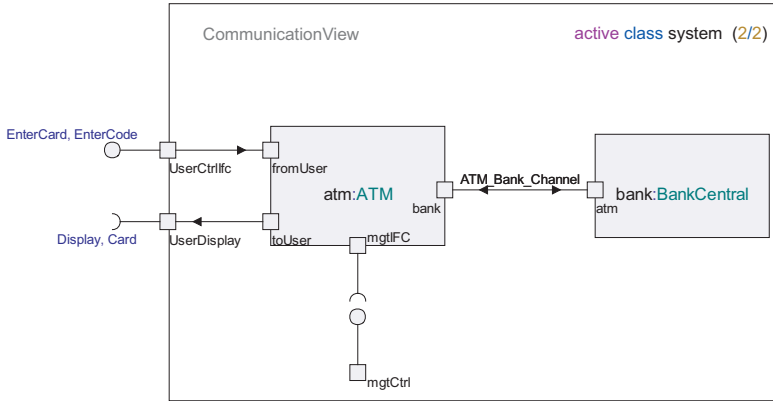


図 100: パート、ポート、コネクタを示す合成構造図

パート

パートは、包含するクラス インスタンスによって所有される 1 つまたは複数のインスタンスを表します。

すべての属性について、パートに多重度を持たせ、ランタイム インスタンスの数を制約できます。パートの多重度が >1 の場合、パートにはコンテナ タイプが仮定されます。コンテナ タイプは、ロードされているプロファイルやアドインによって異なりますが、デフォルトは文字列タイプが使用されます。

包含するクラスのインスタンスが生成されると、これらのパートに対応する一連のインスタンスを、直ちに、または後で、パートの開始基数と多重度で記述されたとおりに生成できます。

シンボル

図 101: パート

- パート シンボルに名前しかない場合、パート シンボルが生成されると、暗黙的クラスが自動的に構築されます。
- 同じ名前を持つ複数のパート シンボルを合成構造図に使用できます。

参照されているクラスを省略すると、インライン クラス定義のあるパート定義に相当します。このようにパートを指定すると、クラス定義がクラスの使用から切り離されず、記述がより簡潔になりますが、再利用にはあまり適さなくなります。

アクティブ クラスのパートは、アクティブ クラス シンボルの属性入力領域に表示できます。パートは包含するクラスの 1 つの属性となり得るからです。属性がパートである場合、その描画はコンテナ クラスとパート クラスの間の包含関係を記述しています。

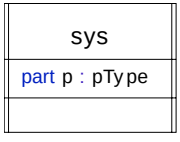


図 102: クラス シンボルの属性入力領域で可視化されたパート

また、249 ページの図 103 に示すように、クラス図で合成関係を使用してパートの階層の概要を表すこともできます。

開始基数によって、包含するエンティティの生成時に自動的に生成される開始インスタンスの数が決まります。開始基数を指定しない場合は、初期に生成されるインスタンスの数は、パートの多重度の下限と同じになります。多重度を指定しない場合は、自動的にインスタンスが 1 つ生成され、同時インスタンスの数に上限はなくなります。これらのインスタンスは、パートのタイプを決める分類子のインスタンスです。

パートは、ポートにコネクタを追加することにより結合できます。パートは、静的 / 動的に生成 / 終了されたアクティブ インスタンスの記述に使用します。

パートは、一連のインスタンスが存在し得ることを示します。この一連のインスタンスは、パートのタイプを決める分類子によって指定されたインスタンスのトータル セットのサブセットです。包含するクラスのインスタンスが終了すると、包含されているインスタンスも終了します。

パート シンボルは、モデルの属性を示します。合成構造図でのパート シンボルの外観は、対応する属性の集約の種類によって異なります。集約の種類が合成の場合、パート シンボルの輪郭は実線になります。集約の種類が参照または共有の場合、パート シンボルの輪郭は点線になります。

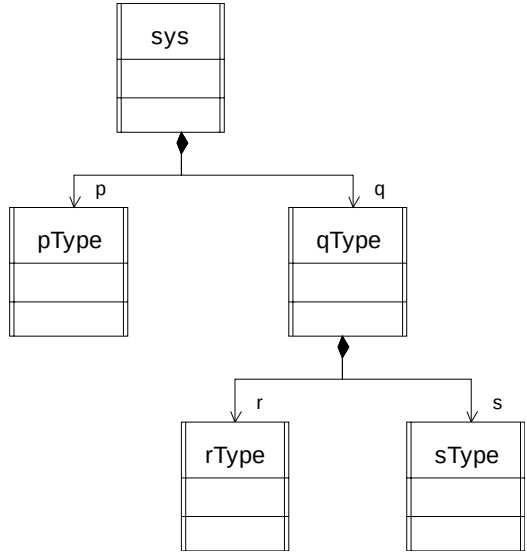


図 103: クラス図で合成を使用して可視化したパート階層

例 **39:** 単純なパート

```
myP
```

例 **40:** タイプ ベースのパート

```
myP :PT
```

例 **41:** 開始インスタンス数と最大インスタンス数を指定したパート

```
myP :PT [0..10] / 1
```

コネクタ

コネクタは、アクティブ クラスのパート間、またはアクティブ クラスの環境とそのパートの 1 つとの間の通信を可能にする手段です。コネクタにより、通信経路を直感的に可視化できます。

コネクタには一方向のものも双方向もあり、各方向で、許可される情報を指定します。コネクタによって送信または伝送可能な情報は、シグナル、属性、シグナル リスト、インターフェイスで記述できます。シグナルの数が多い場合は、コネクタの各方向に使用するインターフェイスやシグナル リストを定義したほうが便利です。

デフォルトで、コネクタに名前はなく、非遅延かつ双方向です。コネクタライン上のショートカット メニューから、コネクタ ラインのプロパティを制御できます。

シンボル

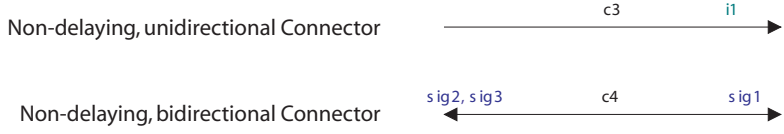


図 104: コネクタ タイプ

コネクタ ラインは、2 つのエンド ポイント、たとえばダイアグラムのパート シンボル、振る舞いシンボル、フレームなどに追加されたポート間の通信経路を指定します。

- 必要であれば、コネクタは省略してもかまいませんが、暗黙的に生成されます。
- コネクタは、ショートカット メニューから方向の変更や双方向化が可能です。
- 双方向コネクタの方向を変更すると、シグナル リスト領域が入れ替わります。
- コネクタの名前はオプションです。
- コネクタと関連付けられたインターフェイスやシグナルなどのリストはオプションです。

アクティブ クラスの構造には、explicit コネクタ ラインか implicit コネクタ ライン、またはその両方を入れることができます。explicit コネクタは可視ですが、implicit コネクタは不可視で、参照できません。

implicit コネクタは、以下のところにある実現化インターフェイスと要求インターフェイスで該当するすべてのものから計算されます。

- 包含するクラスに包含されたパートのポート
- 包含するクラスのポート
- 包含するクラスの振る舞いポート

注記

ポートに explicit コネクタがある場合は、そのポートには implicit コネクタは接続されません。

構文

ラインには、2 つ（一方向コネクタ）または 3 つ（双方向コネクタ）の編集可能テキストフィールドがあります。

中央のフィールドではコネクタの名前を指定し、ラインの終わりのフィールドではシグナル リスト領域を指定します。ラインの各矢印にシグナル リスト領域が 1 つあります。シグナル リスト領域は空白でもかまいません。

コネクタ ラインに適用されたステレオタイプは、名前フィールドの上にあるテキストフィールド（編集不可）に表示されます。

例 **42:** コネクタ シグナル リスト

```
i1,i2,sl1
```

シグナル リストとインターフェイス

シグナル リストからポートへのコネクタを描画できます。その場合、以下のようになります。

- シグナル リストにシグナルまたはインターフェイスがない場合、シグナルとインターフェイスの推定のため、接続したポートが使用されます。
- コネクタに関連付けられたシグナル リストにシグナルまたはインターフェイスもない場合、トランスポートされたすべてのシグナルとインターフェイスを指定する必要があります。

合成構造図のコネクタ ラインのショートカット メニューに［すべてのシグナルを表示］コマンドがあります。このコマンドで、シグナル リスト テキスト フィールドに、接続ポートから取得したシグナルとインターフェイスのリストを挿入できます。

- このコマンドの実行で変更されたリストから、既存のシグナルとインターフェイスを削除することはできません。
- 既存していないシグナルとインターフェイスのみ、シグナル リストに追加できます。
- 2 つの接続ポートにあるシグナルとインターフェイスを結合できます。これで、1 つのポートに表示されるシグナルまたは、インターフェイスを、シグナル リストに表示できます。

- シグナルが実現化または、要求された場合、どのシグナル リストにシグナルを挿入するか決定されます。

パート コミュニケーション

通常、パート間のコミュニケーションはポートとポート間のコネクタ ラインで明示的にモデル化されます。

パート間のコミュニケーションがあいまいでない場合、つまり、ダイアグラム内のパートのクラスに、唯一の手段で接続されるポートが定義されている場合、これを明示的にモデル化する必要はありません。

パート シンボルに直接、コネクタを接続できます。この場合のパート シンボルとコネクタの振る舞いは、名称未設定ポートが作成されてパートに接続され、コネクタがこのパートに接続されます。コネクタの作成時と、既存コネクタへの再接続時の両方で、この方法を活用できます。この名称未設定ポートは、コネクタ ラインが削除された場合に削除されません。このポートがモデルに必要ない場合は、手動で削除する必要があります。

振る舞いポート

アクティブ クラスに構造がある、つまりパートがある場合でも、それ自体の振る舞いを持たせることができ、これは状態機械として表現します。この振る舞いは、合成構造図で振る舞いポートを使用して参照できます。

振る舞いポートの主な目的は、アクティブ クラスのパートとアクティブ クラスの振る舞いの間のコネクタを定義する場合にあります。この場合、振る舞いポートが必要です。

状態機械の通信インターフェイスを定義するため、パートのポートの場合と同様に、コネクタを振る舞いポートに追加できます。1 つの図に複数の振る舞いポートを使用できます。この場合、ポートは同じ基礎振る舞いを参照します。

シンボル

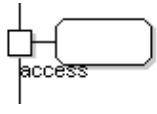


図 105: 振る舞いポート

振る舞いポート シンボルは、定義されたクラスの固有状態機械の参照を指定します。

- 1 つの図に複数の振る舞いポート シンボルを使用できます。
- このシンボルには、テキスト フィールドはありません。

振る舞いポートの外観は、クラス図の通常のポートと同じです。付加された振る舞い情報は、アーキテクチャ図と状態機械図にのみ表示されます。

ヒント

振る舞いシンボルを合成構造図に追加する方法は 2 つあります。ポート シンボルをダイアグラムに追加するか、［モデル ビュー］ブラウザから既存ポートを合成構造図にドラッグします。いずれの場合も、コマンド振る舞いポートをショートカット メニューから選択する必要があります。

関係

依存

合成構造図の依存関係は、パート間で使用され、あるパートが別のパートに依存していることを示します。一般的な使用方法として、パート間の生成依存、つまり、あるパートのインスタンスにより別のパートの新規インスタンスが生成できるということを示します。

モデルの更新

［ActiveModeler］アドインを有効にすると、ショートカット メニューに新しいメニュー［モデルの更新］が追加されます。このコマンドをバインドされていないエンティティに使用し、現在のモデルに呼び出すことができます。

合成構造図では、以下がサポートされます。

合成構造図

ダイアグラムの全要素を更新して、ダイアグラム全体を更新します。

コネクタ ライン

コネクタがバインドされており、ラベル内にバインドされていないシグナルがある場合、これらのシグナルは接続されたポートの［要求／実現化］リストに作成、追加されます。

コネクタがバインドされていない（グレー ライン）でない場合、コネクタが作成され、ラベルには接続されたポートの情報が挿入されます。また、有効な方向はポートから導き出されます。

パート シンボル

アクティブ クラスを作成し、パート シンボルのタイプを更新します。また、ポートをクラスとパート シンボルに追加します。アクティブ クラスには単純な状態機械が含まれ、このマシンから、直接ビルドできます。クラスの名前はパートの名前から導出されます（ある場合）。

コンポーネント モデリング

コンポーネント モデリングでは、システムの主要コンポーネントを特定し、そのインターフェイスと関係をモデリングします。

コンポーネント モデリング時の最重要点は、実装の詳細をコンポーネント内部に隠すことにより強力なカプセル化を実施し、明確に定義された少数のインターフェイスのみを露出することです。

コンポーネント間の弱い結合、つまり依存を最小限にすることも、コンポーネント モデリング時によく適用される設計原則です。

コンポーネント図

コンポーネント図では、一連のコンポーネント、その関係、および実現化インターフェイスと要求インターフェイスによってシステムの静的構造を記述します。クラスやアーティファクトなどの他のモデル要素もコンポーネント図に示して、コンポーネントとの関係を表すことができます。

例

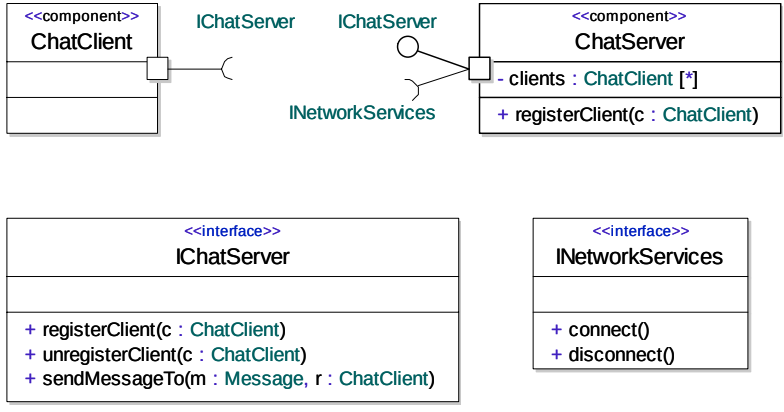


図 106: コンポーネント図

コンポーネント図のモデル要素

コンポーネント図には、以下の要素が使用されます。

- コンポーネント
- アーティファクト
- クラス
- インターフェイス

- ポート
- 実現化インターフェイス
- 要求インターフェイス
- 関係

参照

クラス図

コンポーネント

コンポーネントは、システムの小さいパートをカプセル化したもので、明確に指定されたサービスを提供します。

コンポーネントが提供するサービスは、その実現化インターフェイスによって指定されます。コンポーネントには、実現化インターフェイスによってのみアクセスします。コンポーネントは他のサービスに依存する場合もあります。これは、その要求インターフェイスによって指定されます。

コンポーネントの実装、すなわち振る舞いとアーキテクチャは、クライアントに露出されてはなりません。インターフェイスのみが露出された場合、クライアントに影響を及ぼさずに、コンポーネントを、まったく実装の異なる別のコンポーネントと簡単に置き換えることができます。

UML において、クラスとコンポーネントの違いはほとんどなく、相互に置き換えが可能です。クラスでできることはすべてコンポーネントでもできます。ただし、コンポーネントを使用する場合は、上述の設計原則に従う必要があります。

シンボル

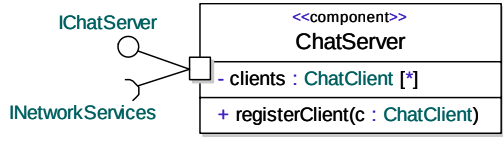


図 107: コンポーネントとポート、実現化インターフェイス、要求インターフェイス

コンポーネント シンボルは、クラス シンボルと同じです。ただし、`<<component>>` というキーワードを上部に追加します。

参照

クラス .

関係

コンポーネント図では、以下の関係を使用できます。

- 関連
- 集約
- 合成
- 依存
- 汎化
- 実現化
- 表現

アクティビティ モデリング

アクティビティ モデリングでは、アクティビティ図を使用して、振る舞いを小さい振る舞い単位に組織化することによりモデリングし、その単位間の制御とデータ フローを記述します。また、システムにおけるこれらの単位の分散も記述できます。

アクティビティ モデリングを抽象レベルでビジネス モデリングに使用したり、非常に低いレベルで使用してアクション コード レベルでの振る舞いをモデリングしたりすることもできます。非同期の分散システムのデザインに特に有効です。

この章では、UML 標準規格で規定されているアクティビティ モデルの実行セマンティックについて説明します。403 ページの「アクティビティ シミュレーション」で説明しているように、Tau ではこのセマンティクスはアクティビティをシミュレートするときに重要です。該当する場合は、この実装における UML 標準規格からの逸脱、または UML 標準規格規格の拡張については注記に記載しています。

参照

シナリオ モデリング

振る舞いモデリング

アクティビティ図

アクティビティ図では、振る舞いがどのように小さい振る舞い単位、アクション ノードに分割されるかを記述し、アクティビティ エッジ、および分岐ノード、フォーク ノード、アクティビティ終了 ノードなどの制御構成要素を使用して、単位間の実行シーケンスを制御します。

複数のアクション間でのオブジェクトとデータの受け渡しを記述するためにオブジェクト ノードとピンを使用します。

アクティビティ区画は、関連アクションを、たとえば機能や所有者ごとに関連グループにまとめるために使用します。

アクティビティ図はフローチャートに似ています。

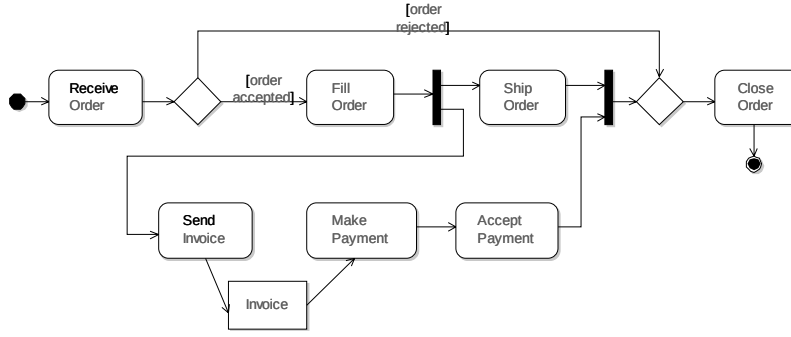


図 108: アクティビティ図

アクティビティ図の作成

アクティビティ図は、パッケージ、クラス、ユース ケース、操作、およびアクティビティに含めることができます。

1. ［モデル ビュー］でアクティビティ図を作成する場所となるエンティティを選択します。
2. ショートカット メニューから［新規］を選択し、［アクティビティ図］を選択します。

フロー方向

アクティビティ図のフローは任意の方向に作成できますが、水平または垂直に揃えて構造化されたフローを作成するための便利な設定があります。フロー方向の制御は［ツール］メニューから［オプション］ダイアログを表示して行います。［［UML 詳細編集］タブ］タブを選択して、［アクティビティ図］セクション、［折り返しの自動生成］を見ます。デフォルトでは［水平］に設定されています。

アクティビティ図で水平方向を選択すると、水平方向のアクティビティ フローを簡単に作成できます:

- ライン ハンドルはシンボルの外枠の右中央に配置されます。
- 新規の［フォーク／ジョイン］シンボルはデフォルトで垂直方向に設定されています。(デフォルトの方向が変更されても、既存の［フォーク／ジョイン］シンボルの設定は変更されません。)
- 新規の区画 シンボルのヘッダー サイズは、デフォルトで、高さが幅より大きく設定されています。(デフォルトの方向が変更されても、既存の区画 シンボルの設定は変更されません。)

垂直方向を選択した場合:

- ライン ハンドルはシンボルの外枠の下中央に配置されます。
- 新規の［フォーク／ジョイン］シンボルはデフォルトで水平方向に設定されています。
- 新規の区画 シンボルのヘッダー サイズは、デフォルトで、幅が高さより大きく設定されています。

Shift + Ctrl キーをしながらフローにシンボルを追加すると、デフォルトのフロー方向を変更できます。

モデル要素からのアクティビティ シンボル

ドラッグ アンド ドロップで、情報を［モデル ビュー］からアクティビティ図にコピーすることもできます。たとえば、操作ノードをドラッグ アンド ドロップしてこの操作を参照するアクティビティ シンボルを作成できます。同じ操作を相互作用ノード、状態機械 ノード、およびユースケース ノードにも適用できます。

注記

アクティビティ ノードをアクティビティ シンボルにドラッグする前に参照から可視にするためには、ショートカット メニューを使用して既存のアクティビティ シンボルの［アクション］を選択する必要があります。

アクティビティ図のモデル要素

アクティビティ図には、以下の要素が使用されます。

- 開始ノード
- アクション ノード
- オブジェクト ノード
- 分岐
- マージ
- フォーク
- ジョイン
- コネクタ
- イベント受信
- シグナル送信
- タイム イベント受信
- アクティビティ終了
- フロー終了
- アクティビティ区画
- ピン
- 関係

アクティビティ

アクティビティはユース ケース、操作その他振る舞いを持つエンティティの振る舞いを表すシグニチャです。アクティビティでは、振る舞いを小さい振る舞い単位、アクション ノードに分類し、トークン フロー モデルをベースにしてこれらの単位の実行を制御します。アクティビティの実装は通常アクティビティ図によって記述されます。

シンボル

図 109: アクティビティ

構文

アクティビティ シンボルは、操作シンボルを基にしています。アクティビティの名前用の編集可能フィールドと、アクティビティのパラメータのための入力領域があります。

アクティビティに適用されたステレオタイプは、名前フィールドの上にあるテキスト フィールド（編集不可）に表示されます。

アクティビティ実装

アクティビティ実装は、アクティビティシグニチャの実装です。アクティビティ実装には、アクティビティ図と、アクティビティ エッジに接続している一連のアクティビティ ノードが含まれています。アクティビティ実装は、通常、アクティビティ生成時に暗黙的に生成されます。

トークンフロー

アクティビティ実装の実行セマンティクスは、トークン フロー モデルをベースにしています。トークンは、あるアクティビティ ノードから別のアクティビティ ノードに向け、接続されたアクティビティ エッジを通じて流れてゆきます。トークンには次の 2 つの種類があります。

- 制御トークン
- データ トークン（またはオブジェクト トークン）

アクティビティ エッジは両方の種類のトークンを転送できます。制御トークンがエッジを越えて転送されるときは制御フローを表し、データ トークンがエッジを越えて転送されるときはデータ フローを表します。制御フローは、オブジェクト ノード以外の任意のアクティビティ ノードを末端に接続した、1 つのアクティビティ エッジです。データフローは、エッジの末端のいずれかの側、または両側にオブジェクト ノードを接続した、1 つのアクティビティ エッジです。

コントロールトークンは、モデル化されたシステムの論理制御の状況を構成します。一方、データトークンは、モデル化されたシステム内を流れてゆくデータ単位の状況を現すために必要です。

1 つのアクティビティエッジとは、アクションノード、制御ノード、オブジェクトノード、ピン、コネクタと連結した、方向付きのエッジです。エッジの方向は、フローの方向を現しています。アクティビティエッジのセマンティクスは、そのターゲットノードとソースノードに依存します。

アクティビティが呼び出されると、そのアクティビティに含まれる各開始ノードに制御トークンが置かれて、アクティビティ実装の実行が開始されます。次に、これらのトークンは発信アクティビティ エッジを横切って下流方向に流れ、これらのエッジがつながっているアクティビティ ノードの着信アクティビティ エッジ側に集まります。アクティビティ ノードは、その**入力条件**が満たされるとすぐに実行を開始できます。アクティビティ ノードの種類によって入力条件が異なります。ただし、標準的な条件として、実行を開始するためには、各着信アクティビティ エッジに使用できるトークンが存在している必要があります。アクティビティ ノードがその実行を完了すると、(ある種の) トークンをすべての発信アクティビティ エッジ エンドに送信します。最終的に、これらのトークンはほかのアクティビティ ノードに届き、その手順が繰り返されます。

注記

アクティビティの実装は、トークンが流入している間は継続します。アクティビティ実装のどのアクティビティ ノードも入力条件を満たしていない場合、トークンは流れず、アクティビティ実装は実行モードのままです。つまり、制御はアクティビティの呼び出し側には戻りません。特殊なアクティビティ ノードであるアクティビティ終了ノードが実行されたときのみ、全体のアクティビティ実装が実行を終了し、制御がアクティビティの呼び出し側に戻ります。

開始ノード

開始ノードは、アクティビティ実装の制御フローの開始点を指定します。アクティビティが呼び出され、実装の実行が開始すると、その実装の各開始ノードは制御トークンを受け取ります。

アクティビティ実装は開始ノードをいくつでも持つことができます。つまり、複数の制御フローを開始できます。また、開始ノードを持つことは必須ではありません。フローはピン、イベント受信、タイム イベント受信から開始することもできます。

開始ノードは着信アクティビティ エッジを持たない場合もあるので、入力条件はありません。開始ノードは、制御 トークンを受け取るとすぐに実行を開始し、このトークンを発信エッジに渡します。

シンボル

図 110: 開始ノード

アクションノード

アクション ノードは、アクティビティ内の実行可能な機能です。アクション ノードの振る舞いは、アクティビティ、操作、または状態機械を使用するなど、多くの方法で指定できます。また、振る舞いをアクション モードに関連付けないようにすることもできます。これは、開発の初期の段階など、振る舞いの詳細が分からないときに役に立ちます。

アクション ノードに振る舞いがある場合は、そのアクション ノード内でインラインに定義できるか、またはそのアクション ノードから参照できます。インラインで定義された振る舞いは、合成的な階層のアクティビティ実装を指定する場合に適しています。合成状態と比較してください。参照される振る舞いは、モデル内の複数のアクション ノードで同じ振る舞いを再利用する場合に適しています。参照される振る舞いを使用する場合、通常ではアクティビティとなりますが、一般的には操作を参照することもできます。参照される振る舞いも実装を持つことができます。たとえば、参照されるアクティビティはアクティビティ実装を持つことができます。

トークンがすべての着信アクティビティ エッジで利用可能なとき、アクション ノードの入力条件が満たされます。その結果、トークンは消費されて実行が開始します。実行が終了すると、制御トークンがすべての発信エッジに与えられます。

実行時のデッドロックの回避

アクションノードへの入力条件が満足されない限り、そのアクションノードは実行できません。実行時のデッドロックを回避するには、アクティビティ実装でのトークンフローのセマンティクスを理解することが非常に重要です。一般的な誤解を解くための例として、261 ページの図 111 に示したアクティビティ実装を考えます。

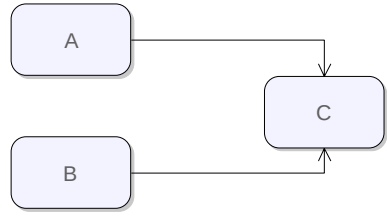


図 111: アクションノード間の制御フロー

この例には、3 つのアクションノード、A、B、C があり、2 つの制御フロー、A から C、B から C があります。ここで、C は両方のエッジにトークンがある場合のみ実行されます。A から C へのエッジにのみトークンがある場合は、C ノードは B から C へのエッジでトークンを待ちます。ノードはエッジ上に集められることを理解してください。

最低エッジの一方のみにトークンがあれば C ノードを実行できるようにしたい場合は、下図のようにノードの間に マージ ノードを挿入します。

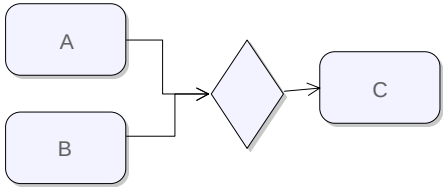


図 112: マージノード

ピン

アクション ノードは、振る舞いを持つ場合、その振る舞いへのパラメータを表すピンを持つことができます。トークンが着信アクティビティ エッジを通って直接アクション ノードに到達するか、接続されているピンを通って間接的に到達するかは重要です。直接到達する場合は、アクション ノードはその入力条件が満たされたときに実行されます。間接的に到達する場合は、アクション ノード自体は実行されません。代わりに、トークンが「ストリーミング」の形で振る舞いの実装に流入して、その結果、振る舞いの実装の実行は開始ノード上の制御トークンではなく、ピン上のデータ トークンを使って開始されます。これらの 2 つの仕組みを組み合わせることが、制御トークンをアクション ノードに流入させ、そしてデータ トークンをそのピンに流入させることによって可能になります。振る舞いがその実行のためのデータを必要とするとき、これはよく用いられる設計の方法です。したがって、振る舞いは入力ピン上で入力データを取得し、実行が開始するときに制御のための制御トークンを取得します。アクティビティ最終ノードを実行して実行を終了する前に、通常、出力ピンにデータ トークンとして配置される出力データが与えられます。

ピンの詳細については、ピンを参照してください。

シンボル

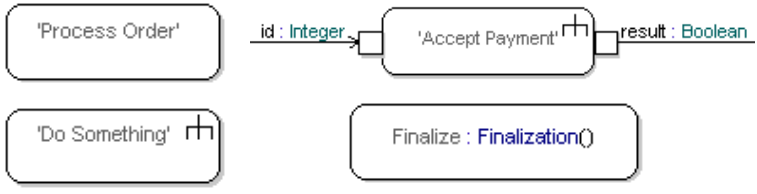


図 113: アクションノードーピンあり／なし、振る舞いあり／なし（左 : インライン、右 : 参照）

ショートカット メニュー選択項目として、[アクション] があります。チェックマークを付けると、テキストフィールドがアクション コードに追加されます。デフォルトでは、アクションテキストフィールドは表示しません。

ショートカット メニュー選択項目として、[区画参照] があります。このコマンドは、シンボル名フィールドの上に区画参照のテキスト フィールドを表示します。デフォルトでは、[区画参照] テキスト フィールドは表示されません。

ショートカット メニュー選択項目として、[すべてのパラメータの表示] があります。[すべてのパラメータの表示] コマンドで、現在、選択しているモデルに対してすべてのピン／パラメータ シンボルを表示できます。

構文

アクション ノード シンボルには非形式名を含むことができます。アクション ノードが振る舞いを参照する場合、振る舞いのシグニチャはコロンの後に表示されます。アクション ノードがインラインの振る舞いを含む場合は、シンボルの右上に「レーキ」のマークが表示されます。

アクション ノードを明示的に含むアクティビティ区画を、名前フィールドの上の別のテキスト フィールドに指定できます。この構文は、丸かっこで囲んだアクティビティ区画への参照をカンマで区切ったリストです。

アクション ノードに適用されたステレオタイプは、アクティビティ区画参照フィールドの上にあるテキスト フィールド（編集不可）に表示されます。

オブジェクト ノード

オブジェクト ノードは、フローに関与するクラスなどの分類子のインスタンスを表します。インスタンスとその値は、アクティビティで使用できます。

オブジェクト ノードの実行前に各着信アクティビティ エッジにトークンが存在しているとき、オブジェクト ノードの入力条件が満たされます。オブジェクト ノードの実行は、単純に、データ トークンを各発信エッジに配置することを意味します。データ トークンのタイプはオブジェクト ノードのタイプであり、すなわち分類子です。

オブジェクト ノードは出力データの取得方法は指定しません。取得方法を指定するには、出力ピンを持つアクションノード ノードを使用できます。このアクション ノードの振る舞いはデータの計算方法を指定します。

シンボル

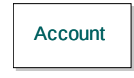


図 114: オブジェクト ノード

構文

オブジェクト ノード シンボルは、それが表す分類子の名前を含む 1 つのテキスト ラベルを持ちます。オブジェクト ノードに非形式名を付けることもできます。構文は <name> : <type> になります。

オブジェクト ノードに適用されたステレオタイプは、名前フィールドの上にあるテキスト フィールド（編集不可）に表示されます。

分岐

分岐ノードは、ガード条件に基づいて複数の外向きフローから 1 つを選択するためにフローで使用する制御ノードです。分岐ノードには、それぞれガードの付いた着信エッジが 1 つと発信エッジが複数あります。

トークンが分岐ノードの着信エッジに到達すると、発信エッジのガードが評価されます。「else」ガードが最後に評価されること以外、ガードを評価する順序は UML には定義されていません。従って、相互に排他的なガード条件を指定することを推奨します。最大で 1 つのガードは「else」ガードとなる可能性があります。このガード条件は、他のガード条件が満たされない場合でも満たされます。

入力トークンは、ガード条件を満たした最初のエッジに置かれます。そのようなエッジがない場合、トークンは分岐ノードによって破壊されます。通常、これは例外的な状況であり、「else」ガードをエッジの 1 つに設けて、この状況を避けることが望ましいです。

注記

アクティビティ シミュレータの活動実行セマンティクスでは、現在インフォーマル分岐と分岐回答のみをサポートします。このような分岐ノードを実行すると、どの発信エッジを選択するかをモデル ベリファイヤ（Model Verifier）が対話によって指示します。これによって、正確なガード条件が分かる前にアクティビティをシミュレーションできるので、これは開発の初期段階で有用な機能です。

シンボル

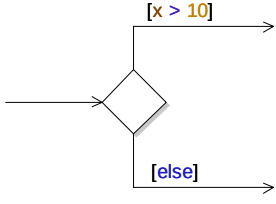


図 115: 分岐ノード

正式な定義では、ガード条件の値は論理表現になります。アクティビティ実装に可視変数、たとえばローカル変数などがある場合、それをガード条件に使用できます。キーワード `else` をガードに使用すると、他のガードの値がいずれも `true` にならない場合はこのエッジが選択されるよう指定されます。

複数の外向き分岐フローを 1 つのフローにマージし直すには、マージ ノードを使用します。

注記

分岐 ノードとマージ ノードには、アクティビティ図エディタのシンボル パレットの同じシンボルを使用します。

マージ

マージ ノードは、複数のフローを 1 つにまとめるのに使用する制御ノードです。トークンが着信エッジの 1 つに到達すると、そのトークンは発信エッジに中継されます。ジョインと異なり、内向きフローの同期化ではありません。

シンボル

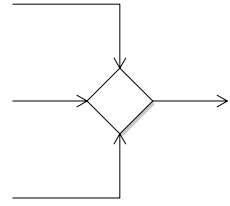


図 116: マージ ノード

注記

マージ ノードと分岐 ノードには、アクティビティ図エディタのシンボル パレット内の同じシンボルを使用します。

フォーク

フォーク ノードは、1 つのフローを複数の並行フローに分割する制御ノードです。トークンが入力エッジに到達すると、そのトークンのコピーが作成され、コピーが各発信エッジに配置されます。これにより、フォーク ノードはアクティビティ モデルに並列処理を導入するための手段となります。

複数の並行フローを 1 つのフローに結合し直すには、ジョインノードを使用します。

シンボル

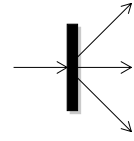


図 117: フォーク ノード

注記

フォーク ノードとジョイン ノードには、アクティビティ図エディタのシンボル パレット内の同じシンボルを使用します。

ジョイン

ジョイン ノードは、複数の並行フローを 1 つのフローにジョイン、つまり同期化し直す制御ノードです。

すべての着信エッジでトークンが利用可能なとき、ジョイン ノードの入力条件が満たされます。この条件が満たされると、次の規則に従ってトークンが出力エッジに置かれます。

- すべての入力トークンが制御トークンである場合、1 つの制御トークンが出力エッジに置かれます。
- 入力トークンのいくつかがデータ トークンである場合、これらのデータ トークンを除くすべてのトークンが出力エッジに置かれます。

注記

アクティビティ シミュレータ内でのアクティビティ実行セマンティクスの現在の実装は、この規則には従っていません。代わりに、ジョインに最後に到着するトークンによって、発信エッジに配置されるトークンの種類が決まります。

1 つのフローを複数の並行フローに分岐するには、フォーク ノードを使用します。

シンボル

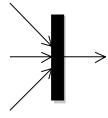


図 118: ジョイン

注記

ジョイン ノードとフォーク ノードには、アクティビティ図エディタのシンボル パレット内の同じシンボルを使用します。

コネクタ

コネクタ ノードは、複雑なフローの表記の図形的な省略表現です。アクティビティ エッジをコネクタ ノードで終了し、同じ名前の別のコネクタ ノードで継続させることができます。コネクタ ノードを使用して、アクティビティ実装仕様を複数のアクティビティ図に分割できます。

コネクタ ノードには複数の着信エッジを含むことができますが、含むことができる出力エッジは多くても 1 つです。セマンティック上、コネクタ ノードはマージノードと同じです。コネクタ ノードの着信エッジに到着するトークンは、その出力エッジに中継されます。コネクタ ノードに出力エッジを含まない場合、セマンティック上、コネクタ ノードはフロー終了 ノードと同じです。

シンボル

図 119: コネクタ ノード

構文

コネクタ ノード シンボルには、コネクタ ノードの名前が入るテキスト ラベルがあります。

イベント受信

イベント受信ノードは、特定のイベント、通常はシグナルを待っていることを示すために使用します。特定のイベントが受信されると、制御トークンをすべての発信エッジに配置することによって、フローが継続します。

セマンティック上、イベント受信ノードは、受信対象のイベントを待ち受ける振る舞いを持つアクション ノードノードと同じです。

このイベントで渡されるデータは、イベント受信ノードからの出力ピンを使用して、後でフロー内で使用できます。受信イベント ノードには入力ピンを含まないことも可能です。

イベント受信アクションは、状態機械のシグナル受信（入力）に似ています。

シンボル

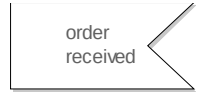


図 120: イベント受信ノード

シグナル送信

シグナル送信ノードは、シグナルのインスタンスを生成して送信するために使用します。これは状態機械のシグナル送信アクション（出力）に似ています。

セマンティック上、シグナル送信ノードは、シグナルを送信する振る舞いを含むアクションノードノードと同じです。

シグナル送信ノードには、送信対象の信号のフォーマル パラメータの実際の引数を与える入力ピンを含むことができます。シグナル送信ノードには出力ピンを含むことができません。

シンボル

図 121: シグナル送信シンボル

タイム イベント受信

タイム イベント受信はイベント受信ノードの特殊なバージョンです。タイム イベント受信は、特定のタイム イベント、通常はタイマーのタイムアウトまたは絶対時間値を待っていることを示すのに使用します。特定のタイム イベントが受信されると、制御トークンをすべての発信エッジに配置することによって、フローが継続します。

タイマーの詳細については、タイマー ハンドリングと時間を参照してください。

イベント受信ノードとは違って、タイム イベント受信ノードにはピンを含むことはできません。パラメータを持つタイマーを待つためには、代わりにイベント受信ノードを使用してください。

シンボル

図 122: タイム イベント受信

アクティビティ終了

アクティビティ終了ノードは、アクティビティの終わりを示します。トークンがアクティビティ終了ノードに到達すると、アクティビティのすべてのフローが終了し、アクティビティの実行が完了します。制御はアクティビティの呼び出し側に戻ります。

アクティビティ終了ノードには任意の数の入力エッジを含むことができますが、出力エッジを含むことはできません。

アクティビティ内の 1 つのフローを終了するには、フロー終了ノードを使用します。

シンボル

図 123: アクティビティ終了

フロー終了

フロー終了は、アクティビティ内の 1 つのフローの終わりを示します。アクティビティ全体ではなく、特定のフローのみが終了します。アクティビティ内に進行中の他のフローが存在する場合があります（フォークと比較してください）。

フロー終了ノードが受け取ったトークンはそのノード内で消費されます。1 つのフロー終了ノードは任意の数の入力エッジを持つことができますが、出力エッジを持つことはできません。

アクティビティ全体を終了するには、アクティビティ終了ノードを使用します。

シンボル

図 124: フロー終了

アクティビティ区画

アクティビティ区画は、スイムレーンとも呼ばれ、関連アクションノードを互いにグループにまとめる仕組みです。アクティビティ図を複数のセクションに分割でき、これで、特定のアクティビティを実行するセクションや、セクション間のデータ フローを確認しやすくなります。

たとえば、ビジネス モデリングでは、会社の様々な部門をそれぞれ区画で表すことができます。他の例としては、リアルタイム オペレーティング システムのスレッドを区画で表すことができます。これで、ダイアグラムには、システムのアクションがスレッド間でどのように分散されているかが示されます。

アクティビティ区画には、通常、クラスと呼ぶタイプを含むことができます。このことは、アクティビティ区画のアクションを実行するインスタンスはこのタイプのインスタンスでなければならないという制約を表しています。このアクティビティ区画は、後でアクションを実行する特定のインスタンスを指定することによって、実行するアクションをさらに制約できます。また、このアクティビティ区画は、アクションを実行するインスタンスを含む属性を指定できます。

シンボル

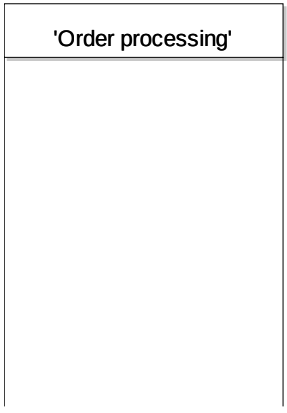


図 125: アクティビティ区画

アクティビティ区画のタイプ、インスタンスまたは属性に関する制約は、名前ラベルの真下にあるラベルに指定します。構文は、ライフライン（生存線）に使われる構文と同じです。

アクティビティ区画 シンボルに図形として含まれるアクションノードノード シンボルは、そのアクティビティ区画に所属するアクションを表します。1 つのアクション ノードを複数のアクティビティ区画に所属させることができます。これは、アクティビティ区画 シンボルを回転するときに起こり得ます。この結果、2 つのアクティビティ区画 シンボルの交差部分に同じアクション ノード シンボルが含まれます。ただし、アクティビティ図は 2 次元なので、この方法で、複数のアクティビティ区画への関与を実現することはできません。アクション ノードが複数のアクティビティ区画に所属するように指定するため、含まれる区画の明示的なリストをアクション ノードについて指定できます。アクション ノードにアクティビティ区画参照の明示的なリストを含む場合、このリストは図形としての位置から推定できる暗黙の参照に優先します。

例 **43:** 暗黙的または明示的アクティビティ区画参照

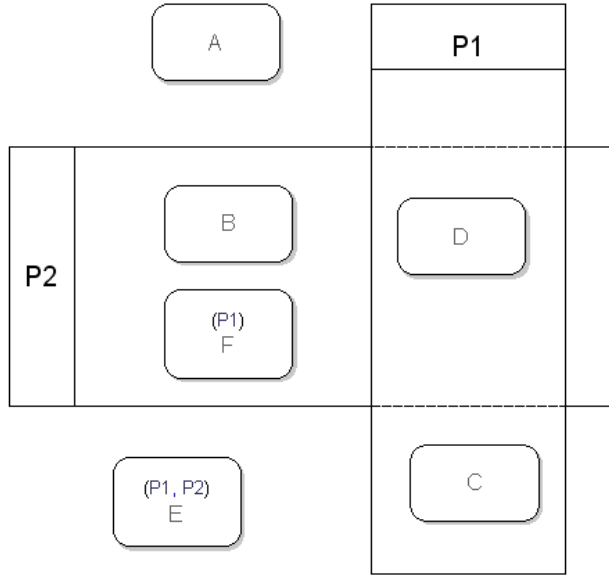


図 126: アクティビティ区画を参照するアクション ノード

上記のアクション ノードは、暗黙のアクティビティ区画参照（アクション ノード シンボルの図形的な位置から推定）と明示的な区画参照の両方を使用します。

A はどの区画にも所属しません。

B は区画 P2 に所属します（暗黙の参照）。

C は区画 P1 に所属します（暗黙の参照）。

D は区画 P1 と区画 P2 に所属します（暗黙の参照）。

E は区画 P1 と区画 P2 に所属します（明示的な参照）。

F は区画 P1 に所属します（明示的な参照）。

ディメンション指定シンボルとしての区画 シンボル

区画 シンボルに複数行がある場合、最上部にある区画 シンボルはディメンション指定シンボルとして使用される可能性があります。その区画 シンボルにディメンションを選択できるショートカット メニューがあります。ディメンションを選択した場合、区画 シンボルのメインのラベルがイタリック体になります。

水平ディメンションと垂直ディメンションの両方を同時に使用できます。

ピン

ピンは、第 4 章「UML 言語ガイド」の 261 ページ、「アクション ノード」ノードの振る舞いのパラメータを表し、振る舞いとの間のデータの受け渡しに使用します。ピンは、アクションとの間の入出力用のオブジェクト ノードと見なすことができます。

着信エッジを持つピンはデータを振る舞いに入力するので、入力ピンと呼ばれます。発信エッジを持つピンはデータを振る舞いから出力するので、出力ピンと呼ばれます。ピンが表すパラメータの方向は、エッジがピンに接続される方法と一致しなければなりません。たとえば、入力ピンは着信エッジだけを含み、対応するパラメータの方向は「in」である必要があります。

ピンの実行のセマンティクスはオブジェクト ノードの場合と同じです。従って、実行により各発信エッジにデータ トークンが配置され、これらのデータ トークンのタイプはピンが表すパラメータのタイプです。

ピンはストリーミングにも非ストリーミングにもなることができます。ストリーミングの場合、アクション ノードノードの振る舞いが実行しているときでも、ピンは出力データ トークンの作成を実行できます。実際に、ストリーミング入力ピン上のトークンの存在とアクション ノードノードの振る舞いが呼び出されるときの条件との間に関係はありません。ただし、非ストリーミングの場合、トークンがすべての入力ピンで利用できるようになるまで振る舞いは実行されません。

注記

アクティビティ シミュレータ内でのアクティビティ実行セマンティクスの現在の実装は、ストリーミング ピンをサポートするだけです。ただし、アクション ノードノードの振る舞いのアクティビティ実装で、ストリーミング ピンとジョイン ノードを結合することにより、非ストリーミング ピンをエミュレーションできます。次に、ジョイン ノードには 2 つの着信エッジがあります。1 つはデータ トークンが到着するピンからの着信エッジで、もう 1 つは、振る舞いを実行するときに制御トークンが到着する開始ノードからの着信エッジです。

シンボル

図 127: ピン シンボル

構文

ピン テキストの構文はパラメータと同じで、`name :Type` となります。

関係

アクティビティ エッジ

アクティビティ エッジは、アクティビティ実装内のノードを接続するために使用します。アクティビティ エッジは、接続された 2 つのノード間の制御トークンとデータ トークンのフローを可能にします。

アクティビティ エッジは必ず方向を持っています。すなわち、トークンはアクティビティ エッジ上を 1 方向にのみ流れることができます。エッジの方向は流れの方向を現します。アクティビティ エッジは両方の種類のトークンを転送できます。制御トークンがエッジを越えて転送されるときは制御フローを表し、データ トークンがエッジを越えて転送されるときはデータ フローを表します。

アクティビティ エッジには、アクティビティ エッジが表すフローを説明する非形式名を含むことができます。

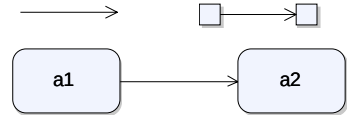


図 128: アクティビティ エッジ

振る舞いモデリング

実行可能なモデルを得るには、操作とアクティブ クラスの詳細な振る舞いを決定する必要があります。仕様決定は振る舞いモデリングで行います。このアクティビティは通常、設計フェーズの最後に行われます。

振る舞いの仕様にステートが含まれる（状態機械実装）ことも、ステートレス（操作本体）であることもあります。いずれの場合も、振る舞いの記述には、以下の 2 つの方法があります。

- 状態機械図の状態機械として記述する
- テキスト図のテキスト記述として記述する

ステートを含む実装の場合はグラフィック形式（状態機械図）のほうが好ましい場合が多く、操作の単純な実装であれば、操作本体を構成するアクションのテキスト記述で十分なことがあります。

状態機械図

状態機械図は、状態機械を可視化したものです。サポートされている状態機械図の作成スタイルは 2 つあります。スタイルについては以下で説明し、例を示します。2 つのスタイルを組み合わせることもできます。

状態（ステート）指向ビュー

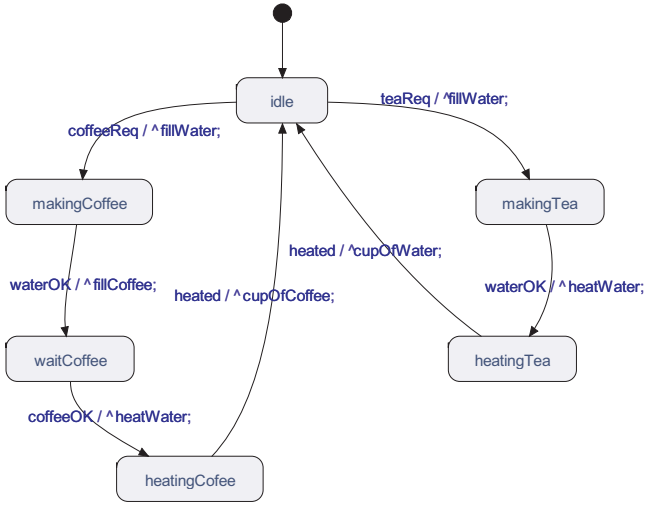


図 129: 状態機械の状態（ステート）指向ビュー

状態機械の状態（ステート）指向ビューにより、複雑な状態機械の概観ができますが、特定の遷移の制御フローや通信の側面に焦点を当てている場合はあまり実用的ではありません。そのため、状態機械を、遷移中に実行できるさまざまなアクションに明示的なシンボルを使用して、遷移指向で記述することも可能です。

遷移指向ビュー

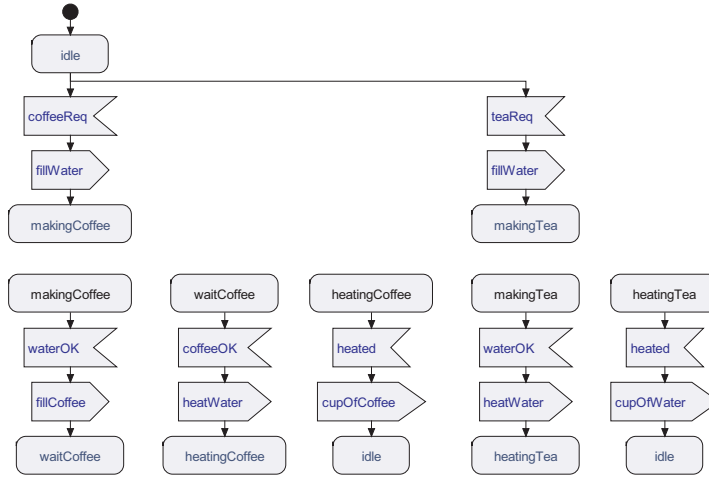


図 130: 状態機械の遷移指向ビュー

状態機械図の作成

状態機械図はクラスと操作（ユースケース含む）に含まれます。

1. ［モデル ビュー］から状態機械を作成する場所となるエンティティを選択します。
2. ショートカット メニューから［新規］を選択し、［状態機械図］を選択します。

状態機械

UML 状態機械は、データとシグナルのハンドリングで拡張された有限状態機械です。状態機械の基本要素は、ステートと遷移です。状態機械 パラダイムに基づくモデルでは、実行は開始点としての何らかのステートと、遷移を実行する起動イベントによって行われます。遷移では、アクションを実行できます。遷移の終わりに、新規ステートに入ります。状態機械は、遷移を開始する新たな起動イベントが発生するまで、このステートでアイドルの状態です。遷移を終了する他の方法として、状態機械（アクティブ クラス）全体を停止します。

ヒント

状態機械は、［モデル ビュー］でクラスを右クリックしてショートカット メニューで［新規］->［状態機械図］を選択するか、［プレゼンテーションの作成］ダイアログを開いて作成します。

シンボル

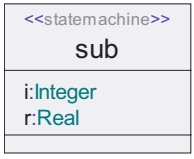


図 131: 状態機械

構文

シンボルには、以下に示すように、編集可能なテキスト フィールドが 2 つあります。

- クラス ヘッダー
- パラメータ

(操作フィールドは空白です。)

パラメータフィールドには、状態機械の仮パラメータが入ります。パラメータは以下の目的で使用されます。

- 生成時のアクティブ クラス インスタンスへの値の受け渡し
- 合成状態へ入った際の値の受け渡し

ステート

ステートは、包含するオブジェクトが、別のステートへの遷移のトリガとなるイベントを待っている状態機械の状況を表します。状況には、静的状態があることがあります (ステートにサブステートがない場合)。この場合、状態機械は、その状態にある間は非アクティブです。状況は、ステートのサブステートに状態機械の振る舞いが隠れているという意味で、動的であることも考えられます。

シンボル

図 132: ステート

ステート シンボルは 1 つまたは複数のステートを参照し、このステート (または一連のステート) からの遷移のターゲットや、ステートへの遷移のソースとして機能します。

構文

- 単純なステート

  `State1`
- リストのあるステート

  `St1,st2`
- 含まれていないステートのリストを含む、アスタリスク ステートのあるステート

  `*(st1,st2)`

アスタリスク ステートは、* シンボルに続くリストで挙げられたステートを除き、現在の状態機械で定義されたすべてのステートを参照するショートカットです。

状態機械は階層的であるため、ステートにはサブ状態機械を持たせることができます。これは、277 ページの図 133 に示すように、ステートの名前に続くコロンの後に状態機械の名前を指定することによって、示すことができます。

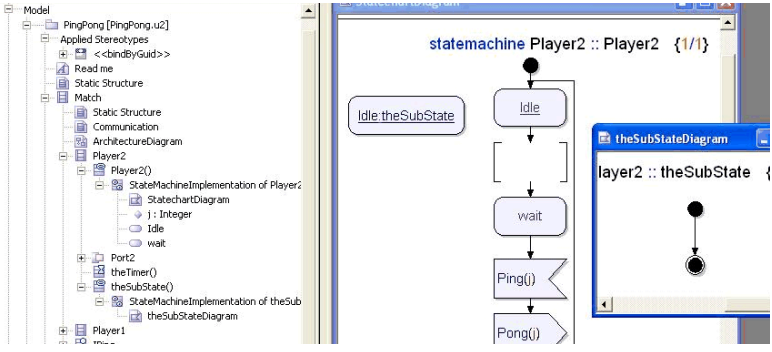


図 133: サブステートの参照

`<state>:<state machine>` 構文を使用できるのは、着信ラインのないステート シンボルの場合のみです（ステート シンボルが「nextstate」ではない）。ラベル `s:myStatemachine` 付きのステート シンボルに着信ラインがある場合は、構文エラーになります。

注記

ステート シンボルには、そのステート シンボルをターゲットとする遷移がある場合は、ステートのリストやアスタリスク ステート定義を含めることができません。ステート リストとアスタリスク ステートでは、遷移のソースのみを指定でき、遷移のターゲットは指定できません。

ステートにサブステート 状態機械があり、この状態機械にエントリ ポイントがある場合、そのエントリ ポイントはステート シンボルで表すことができます。これが可能なのは、ステート シンボルをターゲット ステートとする遷移が 1 つのみの場合だけです。

例 **44: via** 句を含むステート

サブステート 状態機械のエントリ ポイントを決める via 句を含むステート St1

```
St1 via entry1
```

状態機械がサブステートを持つ場合は、フローの分岐を表すマークとしてシンボルの右上に「レーキ」のマークが表示されます。278 ページの図 134 を参照してください。

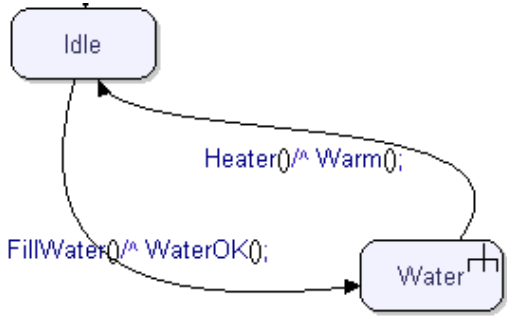


図 134: サブステートを持つステート

ステート シンボルはステートの定義とステートの参照（遷移のターゲット ステート）の両方に使用できるため、シンボルを利用して、遷移の終点と、新規遷移の開始点の両方にし、遷移をチェーンにできます。これは、状態機械の状態（ステート）指向レイアウトを使用している場合は便利です。ただし、遷移指向レイアウトを使用している場合は、読みやすくためにこれを避け、遷移を常に分離するほうがよいでしょう。上述の例を参照してください。

ステートが多数の遷移のソースになる場合、読みやすくするために、これらを複数のダイアグラムで指定してもかまいません。ステート シンボルはステートの部分定義です。

同じ遷移が複数のステートで有効な場合は、ステート シンボルから複数のステートを参照できます。

参照

合成状態

遷移

遷移は、状態機械がアクティブ ステートを変更する際に実行されるアクションのシーケンスです。

遷移に使用する構文は、状態（ステート）指向構文を使用するか遷移指向構文を使用するかにより、2 つのカテゴリに分類されます。状態（ステート）指向遷移構文については 296 ページの「シンプル遷移」で説明しています。遷移指向構文は、遷移開始のための一連のトリガ シンボル、および遷移の詳細を記述する一連のアクション シンボルで記述します。

複数のトリガ シンボルが、遷移を開始させるイベントに対応しています。これに基づき、さまざまな種類の遷移があります。

- トリガ付き遷移
- ガード付き遷移
- ラベル付き遷移
- 開始遷移

トリガ付き遷移には、遷移と関連付けられたトリガがあります。通常、このトリガは特定のシグナルによって定義されますが、タイマーや操作などによっても定義できます。トリガ付き遷移の詳細については、281 ページの「シグナル受信（入力）」セクションを参照してください。

ガード付き遷移の特徴は、特定のイベントにトリガ*されない*ということです。イベントの代わりに、true または false の条件（ガード）によってトリガされます。

ラベル付き遷移は、ステートごとの振る舞いの記述という意味では本当の遷移ではありません。ラベル付き遷移は、ダイアグラムで異なる 2 ページに記述できるよう、遷移を 2 つ（またはそれ以上）のパートに分解するのに使用します。ジャンクションも、フローを分割するのに使用する、ラベル付き遷移の関連構成要素です。

開始遷移（開始）は、状態機械生成時に直接実行される遷移です。

遷移は常に、停止、リターン、別の遷移への制御の移転により、状態機械がステートに入ると終了します。

ガード付き遷移

ガード付き遷移にはトリガがある場合とない場合があります。

ガード付き遷移にトリガがある場合、トリガ イベントが発生した後に式の値が求められます。式の値が true の場合、遷移が起こります。式の値が false の場合は、状態機械はステートにとどまり、トリガ イベントの原因となったシグナルがシグナル キューに置かれます。

参照

保存

履歴の次のステート

履歴の次のステートは、すぐ前のステートに戻るために、遷移の終わりに使用します。

シンボルは、単純な遷移とフロー ライン（詳細）遷移の両方を終了するために使用できます。

浅い履歴

デフォルトでは、履歴の次のステートは、**浅い**ものです。これは、History の付いた Nextstate が遷移の終わりで解釈されると、次のステートは、現在の遷移がアクティブ化されたステートになるということです。

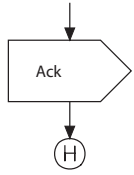


図 135: 浅い履歴の次のステート

履歴の次のステートは、シンボルで名前の代わりにハイフンを使用することにより、通常の Nextstate でも表現できます。

```
nextstate -;
```

詳細な履歴

履歴の次のステートを**詳細な**ものにすることもできます。浅い履歴と同様に、次のステートは、現在の遷移がアクティブ化されたステートになります。これは、入ったステートのサブステートのすべてのレベルまで繰り返し適用されます。

ヒント

履歴の次のステートは、選択し、ショートカット メニューから［詳細な履歴］コマンドを選択することによって深くできます。

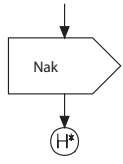


図 136: 詳細な履歴の次のステート

詳細な履歴の次のステートは、次の構文を使用して、通常の Nextstate でも表現できます。

```
nextstate ^-;
```

例

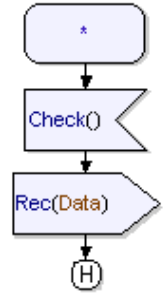


図 137: アスタリスク ステート遷移のある浅い履歴の次のステート

上述の例では、遷移は、遷移がトリガされた際にアクティブであったステートになります。

シグナル受信（入力）

シグナル受信シンボルは、特定の遷移をトリガするシグナルを定義します。

遷移は、ガード式でガードすることもできます。ガード式はシンボルに示されます。

シンボル

図 138: シグナル受信

シグナル受信シンボルはシグナルを受信し、常にステート シンボルが前になければなりません。両方合わせて遷移を定義します。

ヒント

ショートカット メニューから、シンボルを水平に反転させることもできます。
シグナル受信シンボルを削除すると、続くサブツリーも削除されます。

同じ遷移振る舞いを 1 つのステートで複数のトリガ用に呼び出す場合、シグナル受信シンボルに、識別子のリストを持たせることができます。この仕組みでは、各シグナルのパラメータのハンドリングはできず、すべてのシグナルが、1 つの Nextstate で終わる同じ遷移のトリガとなります。

シグナルを受信すると、パラメータは通常、ローカル変数に格納されます。また、パラメータを無視することもできます。

オプションのガード式はトリガの後に定義し、角かっこで囲みます。

シグナル キュー

状態機械は、状態機械に送られるシグナルを到着順に格納する**シグナル キュー**に関連付けられます。

各ステートの、考えられるすべてのトリガに遷移を指定する必要はありません。モデリングしようとしているアプリケーションやドメインの知識から、どのシグナルが到着するかを予測することが可能な場合がよくあります。シグナル キューで次に処理するシグナルが現在のステートで処理されない場合、シグナルは破棄されます。シグナルを一時的に保存することもできます。

構文

シグナル受信でトリガとして以下の種類を参照できます。

- シグナル
- タイマー
- 操作

例 **45:** 単純なシグナル受信

```
s1( i )
```

例 **46:** 複数のトリガのあるシグナル受信

```
s1(i), myTimer, s3
```

例 **47:** 仮想性のあるシグナル受信

```
redefined input s1( i )
```

例 **48:** アスタリスクシグナル受信

すべてのトリガが遷移を起動できるものと指定することもできます。これは、アスタリスクを使用してすべての可視トリガを指定することによって行います。

```
*
```

例 **49:** ガード付きシグナル受信

```
s1 [ x>10 ]
```

開始

開始シンボルは、状態機械の開始点または合成状態の開始点 1 つを定義します。したがって、開始シンボルにより、開始遷移が定義されます。

シンボル

図 139: 開始

構文

開始シンボルには、以下の目的で使用できるテキスト フィールドが 1 つあります。

- 合成状態のエントリ ポイントの参照

```
Entry1
```

- 遷移の仮想性の定義

```
virtual
virtual Entry2
```

アクション

アクションは、通常、テキスト構文を使用してアクション シンボルで行います。使用できるアクションは以下のとおりです。

- ローカル変数の定義文
- 空文
- 複合文
- 代入
- アクション
  - Signal Sending（output）
  - New
  - Set
  - Reset
- 式文
- If 文
- 分岐文
- ターゲット コード文
- While 文
- For 文

- Delete 文
- Try 文
- 終了文
  - Return
  - Break
  - Continue
  - Stop
  - Nextstate
  - Goto（join）
  - Throw

これらの文の一部には、グラフィック構文、つまり専用シンボルもあります。stop、return、分岐、シグナル送信の各文には、遷移に対する重要な操作を強調できるようにするための個別シンボルがあります。当然、これらの文にテキスト構文を使用することもできます。最重要アクションについて説明します。

シグナル送信アクション（出力）

遷移でのシグナル送信アクションにより、シグナルを別の状態機械、環境、または同じ状態機械内に送ることができます。シグナルにパラメータがある場合は、パラメータ タイプに一致する文を指定する必要があります。シグナル送信時、パラメータを無視してもかまいません。

1 つのシグナル送信に複数のパラメータを指定することもできます。これで、連続した別個のシグナルの送信として処理されます。

シンボル

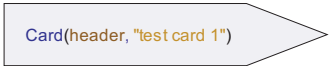


図 140: シグナル送信

シグナル送信シンボルは、遷移からシグナルを送ります。

ヒント

ショートカット メニューから、シンボルを水平に反転させることもできます。

シグナル アドレッシング

以下に示すように、シグナルを受信側にダイレクトしたり、ルーティングしたりする方法にはいくつかあります。

- アドレッシングを省略する
- シグナルを受信側に対するメソッド アプリケーションとしてダイレクトする
- ポートまたはインターフェイスを通じてシグナル送信する

これらのアドレッシング メカニズムそれぞれについて説明します。シグナルの直接アドレッシングは、受信側に対するメソッド アプリケーションでは、ピリオド（<receiver>.<signal>）を使用して表現されます。

シグナル送信

アドレスやパスは指定しません。シグナルは、考えられるパスのいずれか（ポート／コネクタ）で送信されます。

受信側が **this** の場合

コンテキストがアクティブ クラスの状態機械または操作の場合、**this** とは現在のアクティブ インスタンスの状態機械、つまり自分と同じことです。

コンテキストがパッシブ クラスの操作の場合は、代わりに **self** を使用して現在のインスタンスの状態機械を参照します。この場合、**this** はパッシブ クラスのインスタンスを指します。

ポートまたはインターフェイスを通じたシグナル送信

ポート識別子が指定されます。シグナルは、このポートを通じて送信されます。

インスタンスを 1 つだけ実現化する匿名ポートがクラスに定義されている場合は、識別子をインターフェイス名にすることもできます。この場合、その匿名ポートが参照されます。

受信側が属性の場合

変数または属性が宛先として指定されます。変数や属性のタイプはインターフェイス（これを通じてシグナル送信される）またはアクティブ クラス（または RTUtilities パッケージで定義された特殊タイプ Pid）でなければなりません。

属性により、暗黙的属性 **self**、**sender**、**parent**、**offspring** の 1 つを参照することもできます。

受信側が式の場合

式のタイプは、インターフェイスまたはアクティブ クラス（または RTUtilities パッケージで定義された特殊タイプ Pid）でなければなりません。これは受信側が属性の場合と似た状況です。違いは、フィールドや文字列抽出など、より複雑な式をかっこ内で指定できるという点です。

例

例 **50:** アドレッシング メカニズム

アドレスやパスの指定なし

```
SuspendInd
```

受信側が暗黙的属性

```
sender.Ack(id)
```

受信側が属性で、シグナルにパラメータあり

```
Bank.Card(carddata)
```

受信側が Pid 式（Pid 要素のあるインデックス付き配列）

```
(myList[10].addr).Sig1
```

インターフェイス（ポートを参照）

```
Ack(id) via myInterface
```

これらのすべてのアドレッシング メカニズムには以下の共通点があります。

- 状態機械の、生きているインスタンスが通信経路の終わりにない場合、シグナルは消失します。
- 宛先が、中止された状態機械 インスタンスを参照する場合、シグナルは消失します。
- 受信側状態機械が、シグナルが処理されないステートにある場合、シグナルは消失します。

分岐

分岐構成要素は、遷移で、式の値によって選択されたアクションを実行するために使用します。これはスイッチと似た仕組みです。分岐には**質問**パートが 1 つあり、ここに、分岐実行時に値が求められる動的な式が含まれます。また、分岐には複数の**回答**パートがあり、それぞれが範囲式（または値または定数を含む単純式）を持ち、複数の部分遷移となります。

シンボル

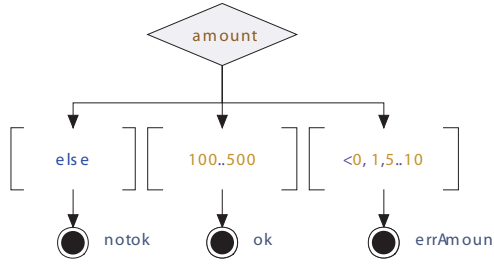


図 141: 分岐の使用

分岐シンボルは、遷移の振る舞いパートの選択パスを指定します。

- 式は以下のようになります。式を定義する必要があります . 各パスには、使用するパスの式に一致する回答でラベルを付けます。
- 分岐シンボルを削除すると、続くサブツリーも削除されます。

分岐回答

分岐回答シンボルは、遷移の振る舞いパートの選択パスのうち 1 つを指定し、分岐質問の回答となる範囲条件を含みます。

範囲条件は以下のいずれかで指定します。

- 特定の値（たとえば `10` や `true`）
- 一端が決まっている範囲（たとえば `>10`）
- 両端が決まっている範囲（たとえば `2..10`）
- 上述の選択のカンマ区切りリスト

インフォーマル分岐

モデルを早期にベリファイしやすくするため、インフォーマル分岐を指定できます。これらの分岐は、文字列と、文字列である回答のある式を持っています。

非決定性分岐

非決定性分岐を記述することもできます。これは、any（引用符なし）を使用し、分岐の回答を空白にしておくことによって行います。

構文

例 **51:** 分岐式テキストの例

```
v+4
```

例 **52:** 分岐選択肢テキストの例

単純な例

```
True
```

一端の決まった範囲

```
>10
```

両端の決まった範囲

```
0..3
```

複数の範囲

```
<-5, 0..2, >10
```

ガード

ガード シンボルは、以下の目的で使用できます。

- 一定の条件の値が true になった場合に遷移をトリガする
- 接続遷移、つまり終了ポイントによってサブステートから出る

シンボル

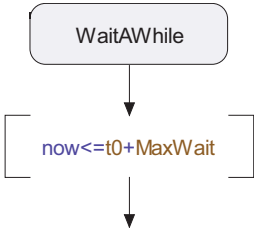


図 142: ガード付き遷移

条件に基づくガード付き遷移の場合、遷移は、条件を定義する式の値が true になると呼び出されます。この式は、単純な式で、副作用を発生させないものである必要があります。

遷移が、終了ポイントの参照によって定義される場合、遷移のソース ステートには、サブステート 状態機械がなければなりません。遷移は、このサブステート 状態機械が、指定された終了ポイントを通って終了するたびに実行されます。

構文

例 **53:** ガード付き遷移

```
[ x>10 ]
```

例 **54:** 接続遷移

名前のある終了ポイント a を通って合成状態が終了するとトリガされる遷移

```
[ a ]
```

例 **55:** 接続遷移

名前のない終了ポイントを通って合成状態が終了するとトリガされる遷移

```
[ ]
```

タイマー設定アクション

タイマー設定アクションにより、タイマー インスタンスが生成され、アクティブになります。アクティブなタイマー インスタンスに対して設定アクションを再度実行すると、初めのタイマー インスタンスが暗黙的にリセットされ、新規タイマー インスタンスが生成されます。

パラメータのあるタイマーの場合、個別のパラメータ値があれば、複数のタイマー インスタンスを同時にアクティブにできます。

構文

例 **56:** 絶対時間

```
set (MyTimer, aTime);
```

例 **57:** 相対時間

```
set (MyTimer, now+10);
```

例 **58:** デフォルト持続時間のあるタイマー

```
timer MyTimer () = 5;
...
set(MyTimer);
```

例 **59:** パラメータのあるタイマー

```
timer MyTimer (Integer id);
Integer i = 1;
...
set (MyTimer (i), now+5);
```

参照

タイマー アクティブ式

タイマー リセット アクション

タイマー リセット アクションにより、アクティブなタイマー インスタンスがあればリセットされます。

構文

例 **60:** 通常のタイマーのリセット

```
reset(MyTimer);
```

例 **61:** パラメータのあるタイマーのリセット

```
reset(MyTimer(i));
```

アクション（タスク）

アクション シンボルは、変数代入、for ループ、値を返すプロシージャのコールなど、遷移の振る舞いパートにテキスト コードを作成するために使用します。

シンボル

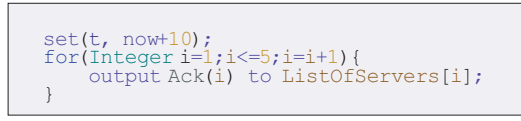


図 143: アクション シンボル

構文

例 **62:** 単純な例

```
Integer v1;
v1 = 4;
output s(v1);
```

代入

代入は、次に示す例の構文に従って行われます。代入の左側には変数識別子、インデックス付き変数の要素、構造体またはクラスの構造体フィールドを入れることができます。右側には、左側と同じタイプの式が入ります。

例 **63:** さまざまな代入

```
Integer i = 0;
myObject = new (theType);
person.age = person.age+1;
arrival[currentDate,person] = now;
```

代入は、それ自体を式として使用することもできます。代入が成功した場合、代入式で返された値が右側の式です。

例 **64:** 代入式

```
if ((a=10)==10) { output s; };
```

複合文

複合文には、中かっこ {} で囲まれた文が複数含まれます。また、名前空間も複合文によって定義されます。これで、複合文中でローカル変数を宣言できるようになります。

New

new 文は、アクティブ クラスとパッシブ クラス両方のインスタンスの生成に使用します。現在のクラスと同じクラスのインスタンスを生成するために、キーワード **this** を使用できます。この構成要素は、生成されたオブジェクトの参照を返します。

オブジェクトへの参照を使用して、作成されたインスタンスと交信することは常に可能です。new 文の実行結果を参照属性に割り当てることで、たとえばシグナルを作成されたインスタンスに直接送信したり、インスタンス上の操作を呼び出したりできます。

しかし、モデルに存在するポートやコネクタを使用してインスタンスと交信できるようにするには、アプリケーションに存在するアーキテクチャ（コネクタとポートの構成要素）に作成されたインスタンスを追加する必要があります。

注記

アーキテクチャへのインスタンスの追加方法は、使用したコード ジェネレータによって異なります。このセクションでは、モデル ベリファイヤ（Model Verifier）および C コード ジェネレータと関連付けてこの方法を説明します。第 41 章「C++ アプリケーション ジェネレータ リファレンス」の 1378 ページ、「属性」では、C++ コード ジェネレータを使用した場合の同様の機能について説明しています。

モデル ベリファイヤ（Model Verifier）または C コード ジェネレータを使用している場合は、コネクタ 構成要素を含むアクティブなオブジェクトの合成属性に追加するだけで、インスタンスをアーキテクチャに追加できます。この方法は、属性の多重度によって異なります。属性が多重度 [0..1] を持つ場合、単純な割り当てで十分です。属性が 1 より大きい多重度を持つ場合は、append 式を使用する必要があります。以下の例を参照してください。

インスタンス数に制限（多重度）を設けてクラスまたはインスタンス セットに new を実行すると、最大許容インスタンス数を超える場合は、実行されません。この場合、値 NULL が返されます。

パート間に依存関係を使用して、パートのインスタンスが別のパートの新規インスタンスを生成できることを可視化できます。

例 **65: new** 文

```
/* Type based creation based on the type ‘a’.
The created instance is not inserted into any attribute and
can thus not be accessed using connectors/ports. */
new a;

/* Creation based on the type of the creator */
new this;

/* Creation plus assignment to a non-composite attribute.
Note that the instance can not be reached using
ports/connectors */
a aRef;
aRef = new a;

/* Creation plus assignment to a composite attribute. The
instance can be reached using the connector structure of the
```

```
creating object that contains the aPart attribute (that is
assumed to be defined as ‘part a[0..1] aPart;’ */
aPart = new a;

/* Creation plus assignment to a composite attribute with
multiplicity >0. The instance can be reached using the
connector structure of the creating object. The aMultPart
attribute can for example be defined as ‘part a[*]
aMultPart;’*/
aMultPart.append(new a);
```

保存

着信するシグナルを、一定の順序で処理する必要があることはよくあります。しかし、外界から着信するシグナルは、予期した順序で着信するとは限りません。他のシグナルの処理を待つ間、シグナルをシグナル キューに一時的に保存するため、セーブ シンボルを使用します。

各ステートに複数のシグナルを保存できますが、保存されたシグナルが次のステートで処理されない場合、破棄されることがあります。

シンボル

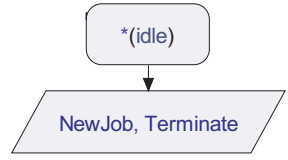


図 144 セーブ シンボルの使用

セーブ シンボルは、シグナルを処理しないステートで処理されそうな場合に、シグナルが破棄されないようにします。

- このシンボルの前には必ずステート シンボルが必要です。
- セーブ シンボルの後にはシンボルを挿入できません。

構文

例 **66:** 保存

単純な例

```
save s;
```

アスタリスク保存

```
save *;
```

停止

ストップ シンボルは、現在のインスタンスの実行を停止します。アクティブ クラスのインスタンスの削除は、クラスの状態機械内から、停止アクションを実行することによってのみ可能です。

シンボル

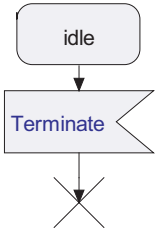


図 145: 停止

停止アクションは、次のように処理されます。

1. インスタンスがパートのない単純な状態機械の場合、状態機械は直ちに停止します。
2. インスタンスにパートがある場合、そのインスタンスに加え、各パート インスタンスが上述の 1 に従って処理されます。

リターン

リターン シンボルは、操作やサブステートの実行を終了し、呼び出しコンテキストに制御を移します。

シンボル

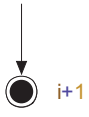


図 146: 操作のリターン

構文

例 **67:** リターンの単純な例

```
4+r
```

例 **68:** 合成状態の終了ポイント名のあるリターン

```
exP2
```

操作にリターン タイプがない場合や、合成状態終了がデフォルト終了ポイントで行われる場合、テキスト フィールドを空白にします。

ジャンクション

通常、複雑な状態機械を複数のダイアグラムに分割するにはステートと Nextstate で十分です。しかし、遷移が非常に長い場合、遷移の記述を複数のパートに分割する必要があることがあります。これは、ジャンクション シンボルによって行うことができます。ジャンクション シンボルは、ラベルと jump 文の両方として使用します。他に、複雑なフローで交差フロー ラインを避けるためにもジャンクションを使用できます。

シンボル

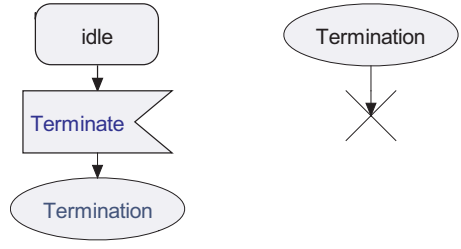


図 147: ラベルまたは goto としてジャンクションを使用

ジャンクション シンボルは、ラベルとジョイン シンボルに相当しますが、フロー ラインをマージする必要がある場合にも必ず使用します。

- ジャンクション シンボルには、複数の内向きフロー ラインを持たせることができます。

構文

シンボルには、テキスト フィールドが 1 つあります。

フロー

フロー ラインは、遷移の 2 つのシンボルを接続します。

- ドローイング エリアでシンボルが選択されており、<Shift> キーを押しながら別のシンボルをツールバーから追加した場合、シンボルの間に自動的にフロー ラインが生成されます。
- ライン ハンドルから次のシンボルにラインを接続することによってもラインを生成できます。
- シンボルを削除すると、シンボルに接続されたラインも削除されます。

シンプル遷移

シンプル遷移ラインは、ステート指向スタイルを使用している場合に遷移を定義するために使用します。

- シンプル遷移ラインは、ステート シンボルからのみ引くことができます。
- ライン ハンドルから次のシンボルにラインを接続するとラインを生成できます。
- シンボルを削除すると、シンボルに接続されたラインも削除されます。

構文

シンプル遷移ラインに関連付けられたテキスト フィールドが 1 つあります。このテキスト フィールドは、遷移のトリガと、遷移のガードとアクションを記述します。

トリガとガードは、シグナル受信（入力）シンボルで使用したものと同じ構文に従います。アクションは、アクション（タスク）シンボルと同じ構文に従います。ただし、ダイアグラムのスペースを節約するため、シグナル送信に省略表現を使用します。^s は output s と同じ意味です。

単純な例

```
s1(x) / ^s;
```

ガードのあるシンプル遷移

```
[ x>10 ] / myproc(x);
```

ガードとシグナル受信の両方

```
s1 [ x>10 ] / myproc(x);
```

式

UML の式は、他のほとんどのプログラミング言語の式と類似しています。式には、変数への参照（属性）、リテラル、定数、操作（呼び出し）が含まれます。

ほとんどの式は最後にセミコロンの付いたアクションとして私用されます。たとえば、以下のような式がアクションとして使用されます。

- 代入 式
- 呼び出し式
- new 式

- 条件式

特殊な変数アクセスや複素数値の生成のために使用する式がいくつかあります。

- フィールド式
- インデックス式
- インスタンス式
- this 式

また、変数アクセスと同様に、システムの基礎の動的ステートに依存する式のグループもあり、これらはよく命令式と呼ばれます。

- 任意値（any）式
- now 式
- Pid 式
  - Self
  - Sender
  - Parent
  - Offspring
- タイマー アクティブ式

他にも以下の式が使用できます。

- Assert 式
- 範囲チェック式
- ターゲット コード式

呼び出し式

呼び出し式は操作の呼び出しのために使用されます。操作呼び出しのための実パラメータを含む場合があります。

例 **69:** 呼び出し式

```
foo(3, true, “mmo”)
```

この呼び出し式の値は、操作呼び出し後の戻りパラメータの実際の値です。呼び出された操作が戻りパラメータを持たない場合は、呼び出し式は値を持たず、そのため、式アクション内のスタンドアローン式としてのみ使用できます。

操作呼び出しの前に、実引数として与えられる式は評価されます。ただし、UML は式の評価の順番を定義していないことに注意してください。実際にどの順序で評価が行われるかは、使用するコードジェネレータや生成コードのコンパイラに依存します。したがって、モデルを呼び出し式の実引数の評価の順序に依存しないようにすることを推奨します。

例 **70:** 引数評価の順序が未定義の場合

```
foo(f1(), f2())
```

この例での操作の呼び出しは、‘f2’、‘f1’、‘foo’（右から左）の順序で行われるか、または ‘f2’、‘f1’、‘foo’（左から右）の順序で行われます。

注記

C コードジェネレータ（AgileC を除くモデルベリファイヤ、モデルエクスプローラを含む）を使用している場合、ターゲットコンパイラの種類によらず、呼び出しの引数は常に左から右に評価されます。UML の標準には評価の順序が定義されていないため、この振る舞いについて独自の解釈をすることは推奨されません。

new 式

new 式には、292 ページの「New」に説明されている new() 構成要素が含まれます。

条件式

条件式は次のような形式です。

```
expr_1 ? expr_2 :expr_3
```

1 番目の式は論理タイプ、2 番目と 3 番目の式は同じタイプです。

まず式 `expr_1` の値が求められます。値が true であれば、`expr_2` の値が求められ、これが条件式の結果となります。値が true でなければ、`expr_3` の値が求められ、これが結果となります。

例 **71:** 条件式

```
imax = ( i > j ) ? i :j; /* imax = max (i, j) */
```

フィールド式

フィールド式は、構造データ型のフィールド、つまりクラスの属性にアクセスするために使用します。

例 **72:** フィールド式

```
a.b = true;
test = a.b;
```

インデックス式

インデックス式は、インデックス付きデータ型の要素、通常は配列や文字列にアクセスするために使用します。

例 **73:** インデックス式

```
iarr[i, j] = 1;
i = iarr[k,l];
```

インスタンス式

インスタンス式は、1 つの操作で複素数値を生成するために使用します。これにより、各フィールドを別個に初期化するのでなく、1 つの操作で構造型を初期化できます。ただし、構造型の初期化にはコンストラクタを使用することを推奨します。

例 **74:** インスタンス式

```
class sType {
    Integer Age;
    Charstring Name;
    Boolean MaleGender;
}
s = sType(. ‘John’, 44, true .);
```

インスタンス式は、タグ付き値を含むステレオタイプインスタンスを記述するときにも使用されます。

this 式

this は、現在のインスタンスを指します。this をパッシブ クラスの操作で使用する場合、this はパッシブ クラスのインスタンスを指します。this をアクティブ クラスの操作または状態機械で使用する場合、this はアクティブ クラスのインスタンスを指します。

命令式

命令式には以下の式があります。

- 任意値（any）式
- now 式
- Pid 式
- ステート式
- タイマー アクティブ式

任意値（**any**）式

any 式は、指定タイプの任意の値を返します。

例 **75: any** 式

```
anInt = any(Integer);
```

```
output resultSig(any(Boolean));
```

now 式

now 式は、現在の時間値を返します。

例 **76: now** 式

```
Time time_0 = now;
set(delayTimer, now + 10);
```

Pid 式

Pid 式は、データ型 Pid の式です。Pid 式は、302 ページの「Pid」に説明されている **self**、**parent**、**offspring**、**sender** のいずれかです。

例 **77: Pid** 式

```
currentClientId = sender;
new serverAgent;
if (offspring != NULL)
  output sender.serverId(offspring)
  else output sender.AllServersBusy;
```

ステート式

ステート式は、現在の状態機械で、最後にいたステートのチェックに使用できます。状態機械に合成状態が含まれる場合、式によって、包含する最も近いスコープの、最後にいたステートが返されます。返される式は、Charstring データ型です。どのステートにもいなかった場合は、空文字列が返されます。

例 **78:** ステート式

```
if (state  == "idle") return ;
```

タイマー アクティブ式

タイマー アクティブ式は、指定タイマーがアクティブかどうかをチェックするために使用します。論理値が返されます。タイマーは、タイマーがまだタイムアウトしていないか、タイマーがタイムアウトしてもタイマー シグナルがまだ処理（または破棄）されていなければ、アクティブです。

例 **79:** タイマー アクティブ式

```
if (active(userTimeout)) reset(userTimeout);
```

範囲チェック式

範囲チェック式は、式がランタイム時、値範囲条件に一致するかどうかをチェックするために使用します。形式は次のとおりです。

```
expr_1 in type type_ident
```

type_ident は、制約によってさらに制限が可能です。範囲チェック式は、指定タイプに式が一致するかどうかにより、論理値を返します。

例 **80:** 範囲チェック式

```
sender in type clientType;
intVar in type Integer constants (1..9, -9..-1);
age in type ageSyntype;
```

ターゲット コード式

ターゲット コード式は、選択された実装言語に依存し、UML パーサによって解析されず、生成されたコードに直接追加される実装言語コードを含みます。

ターゲット コードの形式は次のとおりです。

```
[[ target_code_details ]]
```

ターゲット コード（たとえばインライン C++）には、UML コンテキストで指定されているタイプに一致するものであれば、実装言語の任意の式を含めることができます。

ターゲット コードに次のテキストが含まれている場合

```
]]
```

次のように、# でエスケープする必要があります。

```
#]]
```

ターゲット コードに

```
#
```

が含まれている場合、次のように、# でエスケープする必要があります。

```
##
```

モデル エンティティをターゲット コードから参照する必要がある場合は、形式は `#(name)` となります。`name` はモデル中の識別子です。

例 **81:** ターゲット コード式

```
Real side_a, side_b;
...
```

```
Real hypotenuse = [[ sqrt( pow(#(side_a),2) + pow(#(side_b),2) )]];
```

参照

C Application

Pid

各アクティブ クラスは、4 つの異なる Pid 式にアクセスできます。これは、インスタンス自体に所属する暗黙的な属性にアクセスするものと見ることができます。

self

self 式は、インスタンスを指す Pid 値を返します。

sender

sender 式は、このインスタンスが最後に受信したシグナルのソースであるアクティブ オブジェクトを指す Pid 値を返します。受信されたシグナルがない場合、値 NULL が返されます。

parent

parent 式は、このインスタンスを生成したアクティブ オブジェクトを指す Pid 値を返します。インスタンスがシステム起動時に静的に生成された場合、値 NULL が返されます。

offspring

offspring 式は、最後に生成されたアクティブ オブジェクトを指す Pid 値を返します。（このインスタンスによって）生成されたオブジェクトがない場合、値 NULL が返されます。

注記

各アクティブ オブジェクトで、Pid 式に対応する暗黙的属性は、状態機械終了時にグローバルに更新されません。

Pid 値は、アクティブ オブジェクトの参照と見ることができます。値は、TTDRTTypes パッケージで定義された、定義済みデータ型 Pid です。

ヒント

Pid データ型が定義されている TTDRTTypes パッケージにアクセスするには、RTUtilities アドインをオンにします。これは、［ツール］メニューから［カスタマイズ］ダイアログを選択し、［アドイン］タブを開いて［RTUtilities］を選択することによって行います。これで、［モデル ビュー］の［ライブラリ］で TTDRTTypes パッケージが使用できるようになります。

アクティブ オブジェクトの参照

データ型 Pid の変数を宣言できます。ただし、そのような変数にはあらゆる種類のアクティブ オブジェクトの参照が含まれるため、まとまりがなくなる場合があります。アクティブ オブジェクトを参照する変数は、タイプを以下のいずれかにすることによって制限できます。

- インターフェイス
- アクティブ クラス

これで、代入を実行しようとする場合などに、値が一定の種類のインスタンスを参照することをベリファイする静的なタイプ チェックができます。

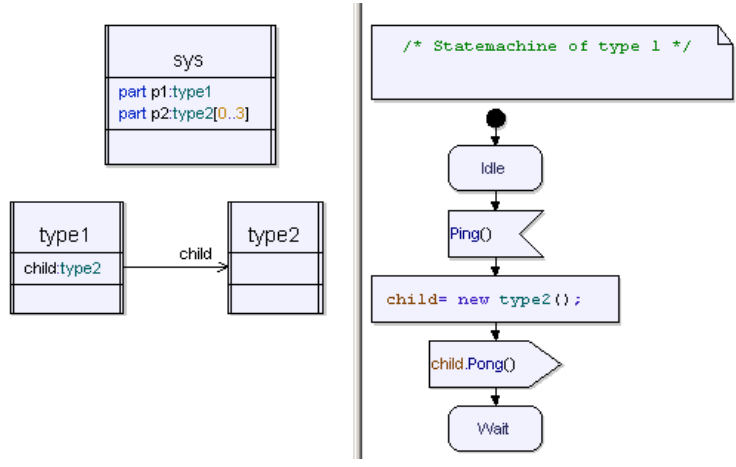


図 148: Pid 変数の使用

タイマー ハンドリングと時間

タイマーは、特殊なタイマー シンボルまたは対応するテキスト構文で定義します。タイマーの宣言では、デフォルトの持続時間、つまり、タイマー設定からタイムアウトまでの時間を指定できます。タイマーをキャンセルする必要がある場合は、リセット アクションを使用します。タイマーは、タイマーをハンドリングしている状態機械が定義するアクティブ クラスのスコープ内、通常はアクティブ クラスのクラス図で宣言する必要があります。

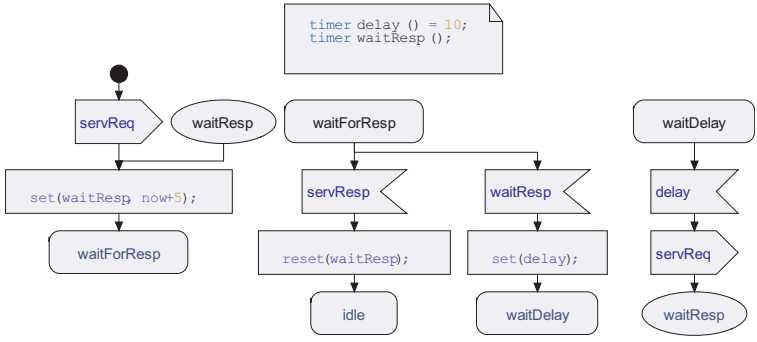


図 149: タイマー ハンドリング

タイマーは、すべてのアクティブ タイマーの状態を把握するランタイムのシステムによって自動的にモニターされます。タイムアウト時、タイマー シグナルは、タイマーを設定し、適切なステートで通常のシグナル受信によってタイマー シグナルを処理する必要のあるプロセスに送られます。通常のシグナルと同様、ステートで受信できないタイマー シグナルは破棄されることがあります。

ヒント

タイマーに関するデータ型が定義されている TTDRTTypes パッケージにアクセスするには、RTUtilities アドインをオンにします。これは、［ツール］メニューから［カスタマイズ］ダイアログを選択し、［アドイン］タブを開いて［RTUtilities］を選択することによって行います。これで、［モデル ビュー］の［ライブラリ］で TTDRTTypes パッケージが使用できるようになります。

合成状態

合成状態は、他のステートや遷移で構成されたステートです。合成状態のサブステートにいる場合、合成状態に定義された遷移のトリガにより、合成状態（およびサブステート）が終了して新規ステートに移ります。

合成状態は、2 つの方法で生成できます : インライン 状態機械定義による、または他の場所で定義された状態機械を参照による方法です。

合成状態は、あるステートについて状態機械図を作成すると暗黙的に作成されます。

合成状態では、ステートシンボルの右上角に「レーキ」シンボルが付きます。

合成状態には、ラベルの付いたエントリ ポイントと終了ポイントを複数持たせることができます。

サブステートの遷移は、外部ステートの遷移よりも優先順位が高くなります。これは、シグナルによってトリガされた遷移と、タイマーによってトリガされた遷移の両方に当てはまります。

これは、UML では、**遷移優先**と呼ばれます。

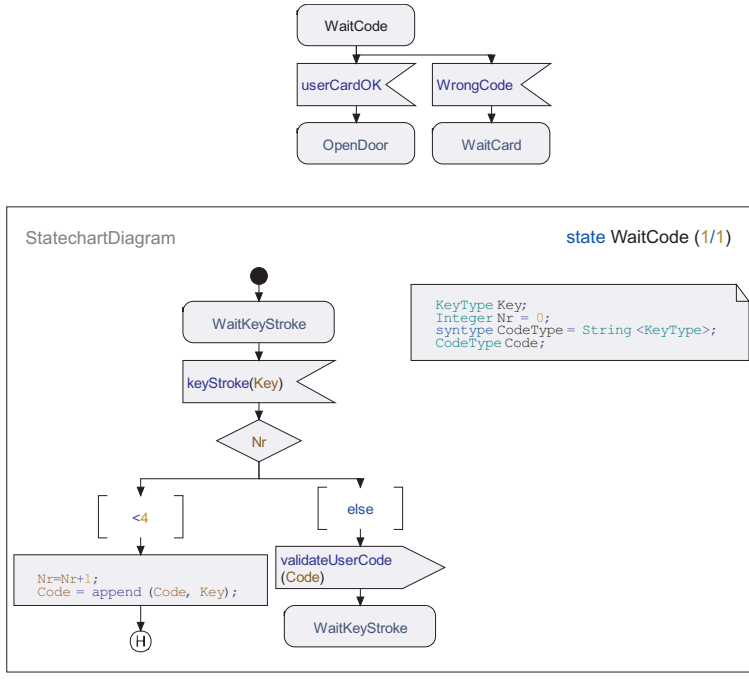


図 150: 合成状態の使用

エントリ接続ポイント

エントリ接続ポイントは、合成状態に入るための名前のある開始点です。エントリ接続ポイントは、合成状態の開始シンボルと、合成状態に入る際に nextstate シンボルで参照されます。

合成状態に、名前のある、または名前のない開始シンボルが少なくとも 1 つ必要です。合成状態では、名前のある開始シンボルは 1 つのみです。

ヒント

エントリ接続ポイントは、モデルビューで定義します。状態機械を選択し、ショートカット メニューから［新規］-［入口接続ポイント］コマンドを選択します。

終了接続ポイント

終了接続ポイントは、合成状態から出るための名前のある終了ポイントです。終了接続ポイントは、合成状態のリターン シンボルと、合成状態から出る接続遷移で参照されます。

合成状態から出る接続遷移が複数ある場合、これらの接続遷移のうち 1 つは名前がなくてもかまいません。

ヒント

終了接続ポイントは、モデルビューで定義します。状態機械を選択し、ショートカット メニューから［新規］-［出口接続ポイント］コマンドを選択します。

状態機械継承

状態機械は、状態機械間の継承によって直接特化するか、状態機械を所有するアクティブ クラスの特化により、特化できます。特化された状態機械では、元の状態機械に特性を追加したり、特性を変更したりできます。追加できる特性は、ステート、遷移、変数、その他状態機械で宣言できるエンティティです。特化によって特性が変更されるようにするには、元の状態機械で**仮想**と宣言されていなければなりません。仮想定義は、特化された状態機械で再定義できます。状態機械で、以下の概念を仮想（したがって再定義可能）にできます。

- 遷移
  - 開始
  - シグナル受信
  - ガード
  - 保存
- 操作

操作本体

操作本体は、ステートのないメソッドです。アクションは、多くの場合、他のアクションのリストを含んだ複合アクションになります。

ヒント

テキスト図は、操作本体を定義するのに便利です。定義するには、［モデル ビュー］で、操作のショートカット メニューから［新規］->［テキスト図］を選択します。次に、ダイアグラムに操作本体のテキスト定義を入力します。

つまり、実行方法の定義を UML 言語で形式的に表現するのではなく、他の言語を使用することが可能です。その場合、操作本体には、非形式的記述を含む非形式的表現が含まれます。

参照

状態機械実装

インターナル

実装

テキスト図

状態機械実装

状態機械実装は、ステートと、状態機械 シグニチャの実現化に必要な他のすべてのものを含むメソッドです。状態機械実装は、通常、状態機械定義時に暗黙的に定義されます。

参照

状態機械

インターナル

実装

インターナル

インターナルは、クラス定義をシグニチャ指向パート 1 つと実装指向パート 1 つに分割し、クラスのシグニチャを、クラスの実装とは別のファイルに格納できるようにするために使用します。この目的は、バージョン ハンドリングやデリバリーをシグニチャと実装で別個にし、コンポーネント ベースのモデリングを容易にすることです。

参照

状態機械実装

操作本体

実装

テキスト拡張シンボル

テキスト拡張シンボルをアクション シンボルに接続して、アクション シンボルの内容を表示できます。これは、遷移指向のフローで、大量のテキストをもつアクションによってダイアグラムの概要がわかりにくくなる場合などに特に役立ちます。アクション コードは、アクション シンボルまたはテキスト拡張シンボル内で編集可能です。

デプロイメント モデリング

デプロイメント モデリングでは、システムのランタイム アーキテクチャのモデリングを行います。配置可能なソフトウェア、アーティファクトが、物理情報処理リソースを表すノードにどのように配置されるかを記述します。デプロイメント スペシフィケーションは、アーティファクトがどのようにノードに配置されるかを記述するのに使用します。関連は、ノード間の接続のモデリングに使用します。

配置図

配置図は、相互接続された一連のアーティファクトに配置された一連のノードを指定します。デプロイメント スペシフィケーションは、アーティファクトをノードに配置する際に使用する実行パラメータの指定に使用します。実行環境を使用して、一連のサービスを、配置されているアーティファクトに提供するノードのモデリングができます。

例

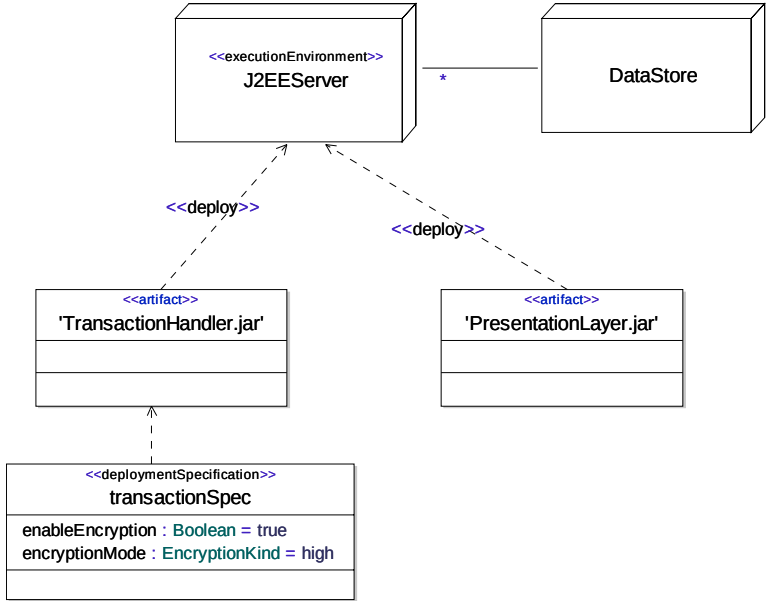


図 151: 配置図

配置図のモデル要素

配置図には、以下の要素が使用されます。

- アーティファクト
- ノード
- 実行環境
- デプロイメント スペシフィケーション
- アーティファクト
- クラス
- 関係

参照

クラス図

コンポーネント図

アーティファクト

アーティファクトは、ソフトウェア開発プロセスで使用または生成される物理情報を表します。ソース ファイル、スクリプト、ライブラリ、実行形式プログラムは、アーティファクトの例です。

アーティファクトは、表現関係によって複数の要素を表します。つまり、アーティファクトは、これらの要素から組み立てられます。たとえば、C++ のヘッダー ファイルを表すアーティファクトには、ヘッダー ファイルで宣言されたクラスとの表現関係を持たせることができます。この情報は、モデルから物理ヘッダー ファイルを生成する際、コード ジェネレータで使用できるようになります。

デプロイメント モデリング中、アーティファクトは、デプロイメント関係を使用してノードに配置されます。

アーティファクトはクラスと似ており、属性と操作を持たせることができます。アーティファクトは、(任意の要素の) 依存、(アーティファクト間の) 汎化、(通常は他のアーティファクトの) 合成の各関係に関与することもできます。また、アーティファクトは名前空間であり、他のモデル要素を所有することもできます。

シンボル

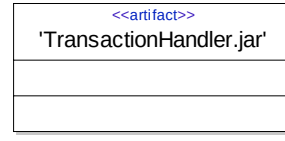


図 152: アーティファクト シンボル

アーティファクト シンボルは、クラス シンボルと同じです。ただし、`<<artifact>>` というキーワードを上部に追加します。

ノード

ノードは、名前のある情報処理リソース、通常は特定のコンピュータです。ノードは、関連を使用して、モデル ネットワーク トポロジーと接続できます。

シンボル

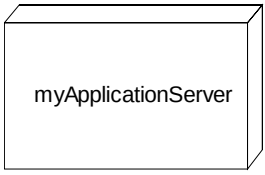


図 153: ノード シンボル

構文

ノードは、内部に名前のある三次元立方体として描かれます。

実行環境

アーティファクトの実行環境が配置された特殊な種類のノードです。実行環境は、通常、実行中にアーティファクトが必要とする一連のサービスで構成されます。

J2EE bean を配置するために準備された J2EE サーバがその典型的な例です。

シンボル

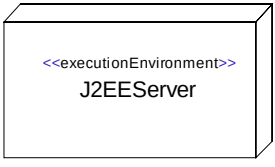


図 154: 実行環境シンボル

構文

ノードと同じです。ただし、`<<executionEnvironment>>` ステレオタイプが適用されます。

デプロイメント スペシフィケーション

デプロイメント スペシフィケーションは、ノードへの配置時、アーティファクトの実行パラメータとして機能する一連のプロパティを指定するために使用します。

ディプロイメント スペシフィケーションは、仕様からアーティファクトへの依存を示すことにより、アーティファクトに適用します。

シンボル

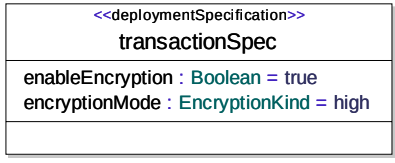


図 155: デプロイメント スペシフィケーション シンボル

構文

クラスと同じです。ただし、`<<deploymentSpecification>>` ステレオタイプが適用されます。

関係

配置図では、以下の関係を使用できます。

- デプロイメント
- 表現
- 関連
- 集約
- 合成
- 汎化
- 依存

デプロイメント

アーティファクトを、デプロイメント ターゲット、通常はノードに配置するために使用する特殊な種類の依存です。ノードに配置されたアーティファクトは、そのノードのコンテキストで実行されます。

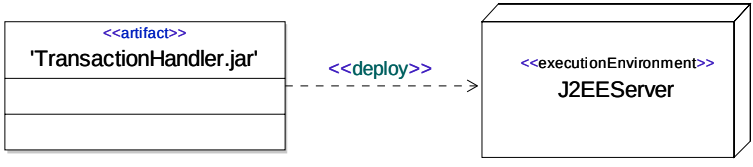


図 156: デプロイメント依存

表現

表現は、アーティファクトから他の一連の要素に使用して、アーティファクトがそれらの要素から組み立てられていることを記述する特殊な種類の依存です。

たとえば、C++ のヘッダー ファイルを表すアーティファクトには、ヘッダー ファイルで宣言されたクラスとの表現関係を持たせることができます。この情報は、モデルから物理ヘッダー ファイルを生成する際、コード ジェネレータで使用できるようになります。

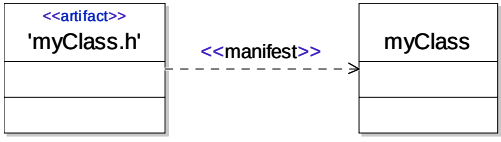


図 157: 表現依存

UML の関係

ラインの編集に関する一般的なヘルプについては、以下を参照してください。

158 ページの「ラインの描画」

159 ページの「ラインの移動」

159 ページの「ラインの削除」

160 ページの「ラインの方向変更と双方向化」

依存

依存は 2 つの定義間の関係で、何らかの理由で、一方の定義（*クライアント*）が、もう一方の定義（*サプライヤ*）に依存していることを示します。依存のセマンティックは比較的緩やかなので、他の関係クラスでは不適切になってしまい一定の関係のモデリングができない場合でも使用できます。

依存の特徴的な使用法が 1 つあります。アクティブ クラスのインスタンス間の生成関係、つまり、あるインスタンスが New 文を使用することによって、あるクラスの新規インスタンスを生成する関係を示す場合です。この場合、パート間、またはパートと振る舞いシンボル（包含するアクティブ クラスの状態機械を参照している）の間で依存を使用できます。

ステレオタイプを適用することによって、依存に対してより詳細な意味を与えるのが一般的です。インポートとアクセス依存を参照してください。

汎化

汎化は 2 つのシグニチャ（クラスや操作など）間の関係で、一方がより一般的なシグニチャで、もう一方がより特化されたシグニチャであることを表現します。特化されたシグニチャは、一般的なシグニチャのメンバー定義を継承し、さらに追加のメンバーを含む場合もあります。このため、汎化関係は「継承」とも呼ばれます。

2 つのタイプ（2 つのクラスなど）間で汎化が行われると、より特化されたタイプが、より一般的なタイプ（スーパータイプとも呼ばれる）のサブタイプを定義します。これは、より一般的なタイプのインスタンスを、より特化されたタイプのインスタンスと置き換えることができるということです。他の言い方をすると、特化された型は、割り当てにおいて、一般的な型を置き換えられます。

構文

汎化ラインには、弁別子を含むテキスト フィールドがあります。

実現化

実現化関係は、特殊な汎化関係です。実現化は、クラスとインターフェイスの間で使用され、実現化するクラスがインターフェイスに一致する (インターフェイスを実装する) ことを表します。

関連

関連は 2 つ以上の分類子のセマンティック関係で、これらの分類子のインスタンスが関係付けられることを示します。

シンボル

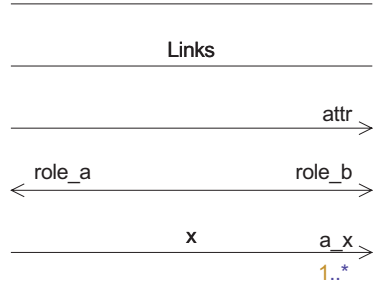


図 158: 関連

ラインには、名前フィールドが 1 つ、役割名フィールドが 2 つ、多重度フィールドが 2 つ含まれます。

関連には関連の端が 2 つあり、これらは属性として表されます。これらの属性は、両方とも関連が所有する（関連付けられている分類子のいずれも関連に影響を受けない状況を反映）か、関連が一方の属性を所有し、もう一方を 1 つの接続分類子 C が所有する（関連が C からの方向のみに誘導可能である状況を反映）か、各接続分類子 C が属性それぞれを所有する（関連が両方向に誘導可能である状況を反映）場合があります。関連が一方向の場合は、2 つ目の（リモート）属性は、必要な場合（役割名や多重度を持っている場合など）のみ存在します。

関連には、関連自体に所属し、特定の関連の端には所属しないプロパティを持たせることができます。

関連は、両方向に誘導可能です。

多重度

関連の端の多重度は、関連によって関係付けができるクラスのインスタンスの数を定義します。

集約の種類

関連は、通常の関連、つまり集約、または合成です。

ラインの端部パートをクリックすると表示されるショートカット メニューで、集約タイプを変更できます。選択肢は、[関連]、[集約]、[合成] です。集約タイプを選択するには、まず役割名を追加する必要があります。

- 集約ラインは、集約クラスのインスタンスが、非形式的に、コンポーネント クラスのインスタンスに所有されていると見なされることを指定します。
- 合成ラインは、コンポーネント クラスが存在する間だけ集約クラスのインスタンスが存在する、強力な集約形式を指定します。したがって、包含されるインスタンスのライフタイムは、それを包含するインスタンスのライフタイムと強く関連しています。

誘導可能端

誘導可能端は、もう一方の端のタイプである分類子の属性でもある関連の端です。

シンボル

ラインには、名前フィールドが 1 つ、役割名フィールドが 2 つ、多重度フィールドが 2 つ含まれます。

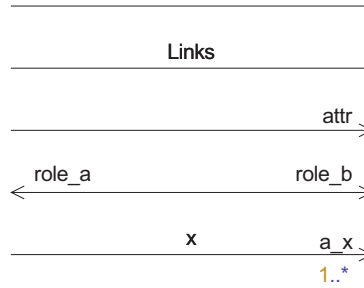


図 159: 関連

例

例 **82:** ロール テキスト

```
+ myrole
```

例 **83:** 多重度テキスト

無限範囲

```
*
```

範囲条件

```
0..3
```

複数の範囲条件

```
1..7,>10
```

参照

属性
集約
合成

集約

集約は、特殊な種類の関連です。集約関係（全体／部分関係）を指定する 2 項関連です。

集約には、集約端と部分端という 2 つの端があります。集約は、集約端の分類子のインスタンスが、部分端の分類子のインスタンスを集約することを指定します。集約インスタンスは、別の集約の一部になることもできます。

集約部分は、複数の集約の一部になることができます。

シンボル

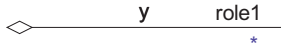


図 160: 集約

参照

属性

関連

合成

合成

合成は、特殊な種類の集約です。合成パートは、合成によって所有され、1 つの合成の一部にしかなることができません。

タイプがアクティブ クラスの合成パートは、合成構造図で記述されている、クラスの内部構造のパートとしても使用できます。

シンボル

図 161: 合成と対応属性

参照

属性

関連

集約

パート

合成構造図

包含

包含関係は、1 つの定義が別の定義を含んでいる状況を表現します。含まれる定義は、コンテナ定義のスコープに表示されます。この関係が名前空間の間で使用される場合には、名前空間のネストとも呼ばれます。

シンボル

包含ラインは「含む側」定義から「含まれる側」定義に向かって描画されます。「含む側」サイドにはプラスマークが表示されます。

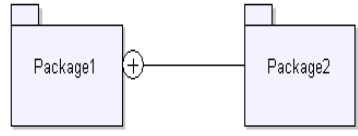


図 162: 包含

拡張

拡張は、ステレオタイプとメタモデル クラスの間で使用され、ステレオタイプがメタクラス (メタモデル クラス) を拡張することを示します。

関連

説明

関連の詳細については、ユース ケース モデリングセクションを参照してください。

テキスト図

テキスト図は、定義の内容のテキスト構文を表すために使用します。この方法で記述するのに適した定義用に使う場合も、また、通常のダイアグラムでグラフィック表現の代わりに使用する場合もあります。テキスト図を使用したほうが実用的な場合の代表例は、操作本体の定義です。

テキスト図の作成

テキスト図は、テキストシンボルをもつダイアグラムがあるモデルの任意の場所に含めることができます。テキスト図は、他のダイアグラム内のテキスト シンボルと同様に、モデル要素とプレゼンテーション要素のコンテナの役割を果たします。テキスト図を作成するには、モデルで適切な参照ノードを選択して、ショートカット メニューから［ 新規］、次に［テキスト図］を選択します。

テキスト図の要素

テキスト図に情報を追加する方法は大きく分けて 2 つあります。

- ダイアグラムに情報を入力する。入力に応じてモデルが更新されます。
- ワークスペース ウィンドウの［モデル ビュー］にあるモデルからテキスト図に、ドラッグ アンド ドロップで移動する。

テキスト図のエンティティは、作成方法とは関係なく、常にモデル要素です。つまり、すべての編集作業がモデルとプレゼンテーション要素に影響します。

Tab キーを使用して、選択したテキスト ブロックをインデントできます。Shift + Tab キーでインデントをナビゲートできます。

参照

第 3 章「ダイアグラムの操作」の 141 ページ、「テキスト解析」

共通シンボル

フレーム

ダイアグラムのシンボルは、キャンバスに置かれたフレーム シンボルで囲まれます。

- フレームにはすべての辺にマージンがあります。
- フレームは、マージンを含め、キャンバスのすべての方向にサイズ変更や移動ができます。

テキスト シンボル

テキスト シンボルは、変数、インターフェイス、データ型などの定義に使用します。

シンボルにラインを接続することはできません。

構文

例 **84:** インターフェイスとシンタイプの定義を含む ——————————

```
interface i {
```

```
  signal s;
}
syntype s = Integer;
```

例 **85:** ステレオタイプの決まったクラスの定義を含む

```
<<struct>> class X {
  private Integer I;
  void inc ( Integer incr ) {
    I = I + incr;
  }
}
```

コメント

コメント シンボルは、ダイアグラムのグラフィック シンボルに関するコメント テキストの定義に使用します。

コメントは、テキスト構文で作成することもできます。

コメント シンボル

コメントシンボルはテキスト シンボルと同様に描くことができますが、シンボルの左上端に読み取り専用を表すマークが表示されます。マークは「//」であり、これにより制約シンボルなどと区別されます。注釈ラインを使用して、シンボルを別のシンボルに接続できます。

コメント シンボルは左側で接続されますが、ショートカット メニューでシンボルを水平に反転させて、右側から接続することもできます。ダイアグラム内のコメント シンボルが他のシンボルに接続されない場合、コメント モデル要素はダイアグラムを所有している要素に属します。コメント シンボルがダイアグラム内の 2 つ以上のシンボルに接続される場合、コメント モデル要素はダイアグラムを所有している要素に属します。

構文

テキストは非形式的であり構文的にチェックされません。

参照

コメントの処理

制約

制約シンボルを使用して、ダイアグラム内のグラフィック シンボルに関連する制約テキストを定義できます。

制約はテキスト構文で作成することもできます。

制約シンボル

制約シンボルはコメントシンボルと同じように作成されますが、シンボルの左上隅には読み取り専用テキスト ラベルがあります。テキストは “{}” に設定されて、制約シンボルとコメント シンボルが区別されます。注釈ラインを使用して、シンボルを別のシンボルに接続できます。

構文

テキストは非形式的であり、構文的にチェックされません。

ステレオタイプ インスタンス

ステレオタイプ インスタンス シンボルを使用して、モデル要素に関連するステレオタイプ インスタンス テキストを定義できます。

ステレオタイプ インスタンス シンボルは、テキスト構文で作成することもできます。

ステレオタイプ インスタンス シンボル

ステレオタイプ インスタンス シンボルは コメントシンボルと同じように作成されますが、シンボルの左上隅には読み取り専用テキスト ラベルがあります。テキストは「<<>>」に設定されて、制約シンボルとコメント シンボルが区別されます。注釈ラインを使用して、シンボルを別のシンボルに接続できます。

構文

テキストは非形式的であり、構文的にチェックされません。

注釈ライン

注釈ラインは、コメント、制約、およびステレオタイプ インスタンス シンボルを別の要素に接続します。

注釈ラインはシンボルのライン ハンドルから引き、ダイアグラム フレーム内の他の任意のシンボルにつなぐことができます。ただし、コメント、制約、ステレオタイプ インスタンス シンボル、およびテキスト シンボルにはつなぐことはできません。

拡張性

UML は、一定の管理の下でカスタマイズ可能な言語です。UML 構成要素を拡張したり、特定の目的で使用するために特化するために、あらかじめ決められた仕組みがあります。

UML の拡張性は、プロファイル とメタモデルの概念に基づいています。

メタモデルは、ツールのリポジトリに格納する情報の記述に使用する、特殊な種類の UML パッケージ クラス モデルです。パッケージは、パッケージ名の前に <<metamodel>> というキーワードが付いていればメタモデルです。通常、メタモデルは、メタクラスを定義する <<metaclass>> というキーワードによってステレオタイプの決められた一連のクラスを含みます。

別のメタモデルを定義し、これらのメタモデルに基づき、内蔵リポジトリを使用してユーザーレベル モデルを格納することもできます。唯一の要件として、メタモデルは、ランタイム リポジトリとストレージの定義に使用されるオブジェクト モデルにマッピング可能でなければなりません。

プロファイルは、パッケージ名の前のヘッダーに <<profile>> というキーワードの付いた特殊な種類のパッケージです。プロファイルには、属性（*タグ付き値定義*と呼ばれる）を持ち、1 つまたは複数の*メタクラス*を拡張する一連の*ステレオタイプ*が含まれます。

ユーザー モデルでは、ステレオタイプを拡張メタクラスのインスタンスであるオブジェクトに適用できます。これで、値の追加が可能になります。

メタモデル

メタモデルは、モデル リポジトリに格納される情報の概念的ビューを定義する一連のメタクラス、メタ属性などです。メタモデルは主に、プロファイル定義の基礎を形成するために使用します。

ユーザー プロファイルにより、より多くの情報をモデル要素に関連付けるためにメタクラスを拡張するステレオタイプが定義できます。追加分の情報は、ユーザーから見て、プロパティ エディタを使用して編集可能で、モデル リポジトリに格納されます。

UML ツール セットにより、特定のモデルのさまざまなビューを提供する複数のメタモデルを表現できます。

ヒント

メタモデルの例が参照できるようになっています。［モデル ビュー］の［Library］フォルダの TTDMetamodel パッケージをチェックしてください。このパッケージは、格納されている情報を記述する単純なメタモデルです。TTDMetamodel の目的は、基礎リポジトリ構造によく似たビューを提供することです。このメタモデルの各クラスは、リポジトリ定義のコア クラスに直接対応しています。だだし、TTDMetamodel は、ステレオタイプに役立つクラスのみが含まれているという意味で、コア レポジトリを単純化したものです。コア レポジトリ モデルのほとんどすべての関連と属性が省略されているという点でも、単純化されています。

メタクラス

メタクラスは、UML リポジトリに格納されている一連の要素のカテゴライズに使用します。メタモデルで、クラス名のステレオタイプを <<metaclass>> としたクラス シンボルを使用して定義できます。

ステレオタイプ

ステレオタイプは、所定エンティティのモデルに格納される情報の拡張に使用します。追加の情報は、ステレオタイプの属性で記述します。

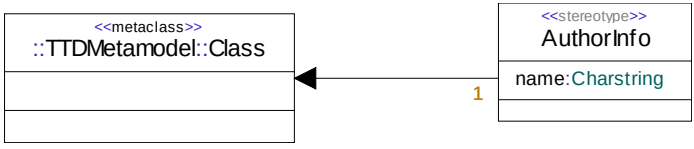


図 163: ステレオタイプの例

322 ページの図 163 は、すべてのクラスを、クラス定義の作者に関する情報で拡張する例です。これは、メタクラス *TTDMetamodel::Class* を拡張する属性 *name* を持つステレオタイプ *AuthorInfo* を定義することによって行うことができます。

タグ定義

タグ定義は、ステレオタイプの属性です。ステレオタイプを適用すると、特定の値を与えることにより、タグ定義が使用されます。

タグ付き値

タグ付き値は、タグ定義に使用できる値です。値は、プロパティ エディタを使用して設定します。

適用されたステレオタイプの表示

モデル要素に適用されたステレオタイプを［モデルビュー］内に表示できます。モデル要素に適用されたステレオタイプを表示するには、以下の操作を行います。

- ［ツール］メニューから［オプション］をクリックします。
- ［オプション］ダイアログ ボックスで、［UML 基本編集］タブを選択します。
- ［ステレオタイプ インスタンスを表示する］チェック ボックスをチェックします。
- ［OK］をクリックします。

参照

拡張

プロファイル

プロファイルは、ステレオタイプが <<`profile`>> の特殊な種類のパッケージです。プロファイルは、メタクラスを拡張するステレオタイプを定義することにより、UML リポジトリに格納できる情報を拡張するのに使用します。322 ページの図 163 は、単純なプロファイルの例です。

プロファイルは、パッケージ インポートまたはアクセス構成要素を使用して適用します。たとえば、モデルに何らかのプロファイルを適用する場合、モデルの上部パッケージには、該当のプロファイルを定義するパッケージを参照するインポートまたはアクセスが必要です。

拡張

拡張は、ステレオタイプとメタモデル クラスの間で使用され、ステレオタイプがメタモデル クラスを拡張することを示します。

拡張ラインに関連付けられたテキスト フィールドが 1 つあります。このフィールドに入るテキストは、1 または `0..1` です。テキストが 1 の場合、拡張メタクラスのインスタンスであるすべての要素に、ステレオタイプが自動的に適用されます。

テキストが `0..1` の場合は、ステレオタイプを手動で適用する必要があります。322 ページの図 163 は、拡張ラインの例です。

ステレオタイプを手動で適用すると、一部のシンボル（クラス シンボル、シグナル シンボルなど）には、適用されたステレオタイプが示されます。

定義済みデータ

UML のデータ モデリング構成要素は強力で、さまざまな方法によるモデリングやデータ定義に対応します。ただし、UML 内蔵のデータのタイプは多くありません。アプリケーションドメインによって UML をさまざまなデータのタイプで拡張できるようになっています。これは、モデル ライブラリ（定義済みパッケージとも呼ばれることも多い）でデータ型を定義することによって行います。

次に示すように、使用できる定義済みデータがいくつかあります。

定義済み

このパッケージには、常に使用できる操作を持つ汎用データ型が含まれています。

TTDRTTypes プロファイル

このパッケージの内容は、モデル ベリファイヤ（Model Verifier）と C コード ジェネレータのみでサポートされているデータ型と操作です。

TTDCppPredefined プロファイル

このパッケージには、C++ アプリケーション ジェネレータのみでサポートされているデータ型と操作が含まれます。

ヒント

定義済みパッケージの詳細は、ツールで簡単にチェックできます。定義済みパッケージにアクセスするには、通常、アドインをオンにします。これは、［ツール］メニューから［カスタマイズ］ダイアログを選択して行います。表示されたダイアログで、［アドイン］タブを選択します。このタブにより、適切なプロファイルをオンにできます。このセクションで述べたパッケージをチェックするには、RTUtilities アドインと CppTypes アドインをオンにします。

参照

データ型

定義済み

Predefined 定義済みパッケージは、独自の UML 拡張で、プロジェクトで常に使用できます。このパッケージは、プロジェクトで定義されたモデルによって自動的に使用されます。パッケージにより、いくつかのデータ型、その他ユーティリティが定義されます。

一部のデータ型は OMG UML にあります（Integer、Boolean など）が、この定義済みパッケージには、通常はないデータ型の操作があります。

データ型ごとに、タイプの表現に適用する一連の操作があります。

パッケージには、以下の定義が含まれます。

種類	定義
データ型	Boolean, Character, String, Charstring, Integer, Natural, Real, Array, Any
定数	PLUS_INFINITY, MINUS_INFINITY

Predefined パッケージは、［モデル ビュー］で直接確認したり参照したりできます。各プロジェクトに、定義済みパッケージのノードがあります。このノードを拡大すると、使用できるデータ型、演算子、その他の定義が参照できます。

PLUS_INFINITY

PLUS_INFINITY はデータ型 Real の定数です。ホスト、または特定のターゲットで使用できる最大 Real 数を参照するために使用できます。

MINUS_INFINITY

PLUS_INFINITY はデータ型 Real の定数です。ホスト、または特定のターゲットで使用できる最大の負の Real 数を参照するために使用できます。

TTDRTTypes プロファイル

None

パッケージには、定義済みシグナル none も含まれます。

このシグナルは、非決定性振る舞いのモデリングに使用するシステム内蔵シグナルです。シミュレーションのための抽象特化またはモデルでのみ使用し、アプリケーションの元になるモデルには使用しません。

none は、非決定性遷移（自発遷移）を定義するために、トリガとして使用します。このイベントの原因は制御できません。発生や発生頻度も制御できません。

シミュレーション中に、モデル ベリファイヤ（Model Verifier）によって制御することはできます。［メッセージ］ウィンドウで、シグナル `::NONE` を挿入できます。

メタモデル クラス

以下に、他の重要なメタモデル クラスについて説明します。

メタモデル プロファイル

TTDMetamodel は、［モデル ビュー］で直接確認したり参照したりできます。プロジェクトを追加する場合、プロファイルが適用されたモードが常にあります。TTDMetamodel はこれらのプロファイルの 1 つです。ライブラリ ノードと TTDMetamodel ライブラリ パッケージを拡大すると、言語モデル要素、抽象メタクラス、その間の関係が参照できます。

分類子

分類子は、UML 言語のメタクラスです。

分類子はデータの記述で、インスタンスの集合、つまりインスタンス セットのシグニチャです。分類子はタイプを定義します。たとえば、`StructuralFeature` タイプなどです。分類子は、関連により、他の分類子と関連付けることができます。

ほとんどのクラス的なモデル要素は分類子です。以下のようなものがあります。

- クラス
- データ型、シンタイプ、選択
- ステレオタイプ
- インターフェイス
- コラボレーション

シグニチャ

シグニチャは、UML 言語のメタクラスです。

シグニチャは、別のシグニチャの定義の基礎にできるエンティティです。これを可能にする主な仕組みは以下の 2 つです。

- 特化、または継承
- パラメタライゼーション

特化とは、スーパー シグニチャを一連のサブ シグニチャに特殊化できるということです。各サブ シグニチャは、スーパー シグニチャのすべてのプロパティを継承し、他にもプロパティを持つことができます。メタモデルでは、特化の仕組みは、シグニチャが所有する汎化クラスによってモデリングされます。

パラメタライゼーションとは、シグニチャに仮コンテキスト パラメータのリストを持たせることができるということです。このようなシグニチャは*テンプレート*と呼ばれます。テンプレートの仮コンテキスト パラメータは、テンプレートのインスタンス化時（`TemplateTypeInstantiation` などで）に実コンテキスト パラメータと置き換えることができます。パラメタライゼーションにより、シグニチャがより柔軟になり、さまざまなコンテキストで使用できます。メタモデルでは、パラメタライゼーションの仕組みは、シグニチャが所有する `ContextParameter` クラスによってモデリングされます。

シグニチャに基づいて新規シグニチャを定義するこれらの 2 つの仕組みに加え、シンタイプという、シグニチャ 1 つのみを持つ仕組みがあります。この仕組みでは、シグニチャの制約により、新規シグニチャを定義します。

シグニチャには、実装を持つものもあります。その場合、シグニチャは実装のファサードとして機能し、シグニチャのユーザーが知らなくてもよい詳細をすべて非表示にします。ファサードにより、実装から定義が分離され、システムのパートのコンパイルが個別にできるようになります。C プログラミングのヘッダー ファイルの使用などと比較してください。ファサードには、以下のことが当てはまります。

- ファサードは実装に依存しない
- ファサードは使用方法に依存しない（これはすべての定義について当てはまります。

実装がファサードに依存します。

以下のモデル要素はシグニチャです。

- 分類子
- 操作、シグナル、タイマー

実装

実装は、シグニチャのユーザーが知る必要はないが、実行には必要な、シグニチャに関する詳細を記述します。通常、シグニチャはエンティティの静的プロパティを記述します。対応する実装は、動的プロパティのほうに関係します。

実装には、インターナルとメソッドの 2 種類があります。インターナルは、クラスの構造を、物理的に、または通信の観点から記述します。メソッドは、操作、StateType、クラスをランタイム実行の観点から記述します。

実装はシグニチャ（ファサードとも呼ばれる）にのみ依存し、シグニチャの使用方法には依存しません。これは、システムのパートの個別分析ができるようにするために重要です。

メソッド

メソッドは、操作の実装です。ランタイム時にどのように実行されるかを記述します。メソッドには 3 種類あり、それぞれに実行セマンティックがあります。

- 操作本体：操作本体のアクションの実行により実行されるステートレス メソッドです。
- 状態機械実装：アクティブ ステートで開始できる遷移に関連するアクションを実行することによって実行される、ステートと遷移のあるメソッドです。
- 相互作用：一連の属性間の相互作用と情報のやりとりを記述するメソッドです。他のメソッドと異なり、相互作用は、実行方法を完全に指定するだけでなく、実際どのように実行されるかを記述したり（トレースの記述）、どのように実行しなければならないかを部分的に記述したり（これで他のメソッドにセマンティック要件を適用できます）するためにも使用できます。
- アクティビティ実装：小さい振る舞い単位の管理セットを実行するメソッドです。

シグニチャと実装

シグニチャと実装は、UML 言語のメタクラスです。シグニチャはエンティティを宣言し、実装は同じエンティティを定義します。すなわち、これらの概念により、シグニチャを実装から物理的に分離できます（C や C++ のヘッダーファイルと比較してください）。

これが可能な概念を以下に示します。

操作

操作シグニチャおよび操作本体、アクティビティ実装、状態機械実装または相互作用

アクティビティ

アクティビティ シグニチャとアクティビティ実装

状態機械

状態機械 シグニチャと状態機械実装

クラス

クラス シグニチャとインターナル

集合タイプと多重度

多重度の概念と集合タイプの間には深い関係があります。このセクションではこの関係の側面について説明します。

暗黙的な集合

UML で属性を定義する際、属性が多値であると定義するために多重度を使用できます。UML の観点から、多重度は属性の実装に関する制約を定義します。文字列集合タイプの暗黙的なインスタンス化は、デフォルトで実装として使用されます。このため、アクション コードで属性を使用する際、文字列集合タイプに定義された操作を使用できます。

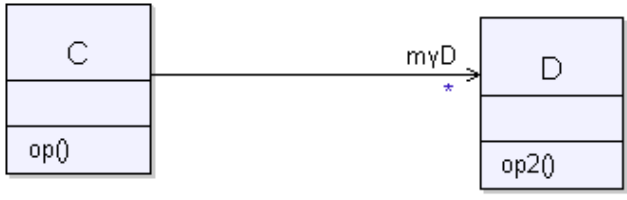


図 164: 多重度 * を持つクラス

例 **86: String** で使用可能な **append**、**length**、**indexing** 操作

一例として、328 ページの図 164 の状況、多重度 * を持つ myD 属性のあるクラス C を考えてみましょう。

この状況で、アクション コードの myD 属性を使用する際、文字列集合タイプで使用可能な操作を使用できます。たとえば、329 ページの図 165 の例のように、op 操作を定義することがあります。

```
void op() {
   Integer len;
   myD.append(new D());
   myD.append(new D());
   len = myD.length();
   for ( Integer i = 1; i < len; i = i + 1) {
       myD[i].op2();
   }
}
```

図 165: 演算子の定義

暗黙的集合タイプの変更

状況によっては、異なる一連の操作や異なる実装特性を使用するため、使用する暗黙的なタイプを変更できます。

これには 2 通りの方法があります。非形式的多重度を使う方法と、`<<containerType>>` ステレオタイプを使う方法です。

非形式的多重度（**Informal Multiplicity**）

非形式的多重度は、UML 多重度が集合のために特別なコンテナタイプを使うべきではないということを意味します。その代わりに、属性の型をコンテナタイプであると仮定します。

標準の String ではなく Bag 集合タイプを使用したいとします。

プロパティ ページを使用する場合、`myD` 属性のタイプを `Bag<D>` に変更して、`InformalMultiplicity` プロパティを選択します。このオプションを選択すると、UML のダイアグラムで記述される多重度は制約としてのみ表示され、集合タイプが暗黙的な String タイプではなく自ら指定したタイプとして表示されます。`InformalMultiplicity` プロパティを選択しないと、暗黙的な集合タイプがそのまま使用され、属性の完全な定義は、String<Bag<D>> となります。しかし、`InformalMultiplicity` プロパティを設定すると、暗黙的な集合タイプは使用されません。

テキスト シンボルで使用されるテキスト構文を変更した場合、属性の定義は D[*] myD; から Bag<D> {[*]} myD; に変更されます。多重度を表示するために使用した中かっこ構文は、非形式的に制約として解釈されるので注意が必要です。

属性入力領域で変更を行う場合、テキストを myD:D[*] から myD: Bag<D> {[*]} に変更します。ここでも、多重度が非形式的な方法で解釈されることを示す中かっこが使用されていますが、これは暗黙的な集合タイプを意味するものではありません。

<<containerType>> ステレオタイプ

多重度が 1 ではない属性が大量にある場合、そのすべての集合型を変更するために非形式的多重度を一つ一つに指定してゆくのは労力を要します。このような場合は、定義済みステレオタイプの `<<containerType>>` を使うと有効です。このステレオタイプはモデルレベル、任意のパッケージレベル、任意の分類子レベルで適用できます。タグ付き値 'Type' を含んでおり、これが、使用するスコープにあるすべての属性に対して、暗黙のコンテナ型を指定します。

指定したコンテナ型がテンプレート型の場合は、ただ 1 つの型のテンプレートパラメータを持つ必要があります。たとえば、以下のような形です。

```
<<containerType (. Type = MyContainerType<Any> .)>>
package P {}
```

この 'Any' は属性の実際の型を参照します。つまり、集合の要素の型です。

`<<containerType>>` をモデルの異なるスコープレベルに適用できます。属性から Model ノードまでのスコープパスに複数の `<<containerType>>` インスタンスがある場合、属性のスコープに最も近いものが使用されます。

コンテナ型を変更した場合は、変更を反映するために、[すべてをチェック] コマンドを起動してモデルを再バインドさせます。

注記

典型的な使用例は、Java や C# のように、UML String 型の表現を持たない言語をターゲットとした UML モデルを作成するときです。この場合には、`<<containerType>>` ステレオタイプを適用（通常は Model レベルで適用）して、そのターゲット言語に存在する適切なデフォルトコンテナ型を指定するのが有効です。

多重度と合成

一般に、集約の種別は、2 つの関連オブジェクト間の生存期間の依存関係を区別するために使用します。合成は、生存期間が依存し合うことを示していますが、参照（および、共有依存）は、生存期間が依存し合うということを意味しません。また、合成は、ある特定のインスタンスがあるコンテナ内のパート（Part）であることを示します。こういったことは、UML の観点からは単なる静的制約になりますが、実装コードの観点からは、合成と静的な多重度の組み合せによってさまざまな考慮が必要になることになります。

多重度と同様に、Tau では、「形式的」な集約と「非形式的」な集約を利用できます。形式的な集約を使うと、Tau が自動的に実装を判断します。非形式的な集約は、モデル上の非形式的な制約として表現されるだけです。

デフォルトでは、Tau は合成と静的多重度を形式的に解釈を使用し、あるプロパティを持つ実装を行います。

例 **87:** 多重度

```
part A[1] a;
```

'a' オブジェクトはコンテナとともに割り当て、終了されます。たとえば、A がパッシブ クラスであると仮定した場合、C コード ジェネレータは、属性 'a' が C とともにインライン記述されるコードを生成します。

```
part A[4] a;
```

4 つの A オブジェクトが、コンテナとともに割り当てられ、終了されます。たとえば、A がパッシブ クラスであると仮定した場合、C コード ジェネレータは、属性 'a' が C ストラクトとともにインライン記述されるコードを生成します。つまり、A のポインタの配列ではなく、A の配列が生成されます。

多重度と同様に、集約の正式解釈をしないようにできます。これも、形式／非形式の多重度と同じ仕組みで制御できます。非形式的多重度を定義した場合、自動的に非形式的な集約を意味します。逆もまた同様です。

Value<> テンプレート

非形式的多重度／集約の種類を使用する場合、必要に応じて属性タイプの定義に値または参照セマンティックを使用できます。値の定義には Value<> テンプレートを使用できます。

例 **88:** 値のセマンティック

```
Value<A> a;

String<Value<A>> {[4]} a;
```

例 **89:** 参照のセマンティック

```
A a;//

String<A> {[4]} a;
```

合成属性を使用する際、以下の例に示すように、いくつか考慮すべき点があります。特に、合成と通常の参照を組み合わせて使用する場合、また値のインスタンス化と UML レベルの合成を組み合わせて実行する場合に注意が必要です。

例 **90:** 合成とインスタンス化の組み合わせ

```
class C {}
class D {
Value<Value<C>> myC1; // Means the same as
                      // 'Value<C> myc1'
part Value<C> myC2; // Means the same as 'Value<C> myc2'
                    // and as 'part C myC2'
Value<C>[*] CList1; // Means the same as
                    // 'part C[*] CList1'
```

```
}
```

多重度と集合タイプのまとめ

集合タイプの暗黙的な生成と合成セマンティックの使用を要約すると、次のようになります。

- ある属性が多重度 >1 を持つ場合、暗黙的な文字列集合タイプとなります。
- 属性の多重度が固定で、かつ合成集約の種類がある場合、インスタンスは自動的にコンテナの一部として割り当てられます（コード ジェネレータ固有の方法で）。
- 多重度と集約の種類のデフォルト実装が希望するものでない場合、属性の非形式多重度プロパティで無効にできます。

注記

これらのルールは、操作やシグナル パラメータなど、多重度を持つことができるすべてのエンティティに適用されます。

更新されたモデルにおける **Value** テンプレート

2.2 以前のバージョンの Tau では、Value<> テンプレートはサポートされませんでした。古いレベルで作成されたモデルをアップグレードすると、ツールは Value 要素を必要な箇所に自動で追加します。332 ページの例 91 を参照してください。

例 **91:** モデルのアップグレード

2.2 以前のモデル：

```
class Class2 { }
syntype Class2String = String < Class2 > constants (0..10);
class Class1 {
    public part Class2String myPart;
}
```

アップグレードされたモデル：

```
class Class2 { }
syntype Class2String = String < Class2 > constants (0..10);
syntype Class2String_Value = String<Value<Class2> >;
class Class1 {
    public Class2String_Value myPart;
}
```

SysML

このセクションには Tau でサポートされる SysML のダイアグラムとシンボルのリストがあります。

SysML は、ハードウェアからソフトウェア、データ、手順、そしてファシリティまでにわたる仕様記述、分析、設計、システム ベリフィケーションとシステム バリデーションのためのビジュアル モデリング言語です。

SysML のゴール

- システム開発ライフサイクル全般にわたるコミュニケーションの改善
- 知識の取り込みを拡張
- 設計の再利用を促進
- 初期段階での設計の検証を可能にする

SysML はモデルベースの設計、つまり、一貫性があってあいまいさがない設計と仕様記述を可能にします。SysML は UML 2.0 に基づいており、SysML を採用することによって、システムのうちソフトウェアとして実装すべき部分の仕様と初期の作業開始点として作成したモデルを再使用できるようになります。システムとソフトウェアエンジニアリングの間のギャップを埋めることでエラーと保守コストを減らすことができます。

SysML は システムエンジニアリング（ハードウェアとソフトウェア）向けということで UML に採用されています。SysML を使うと以下のモデル化が可能です。

- 要求 については、要求のモデリングを参照してください。
- 割り当て（たとえば、振る舞いから構造への機能割付けなど）
- パラメトリック制約

Tau における SysML

SsyML はインストール時に提供されるアドインの 1 つです。SysML を使用してモデルを作成するには、このアドインを活動化する必要があります。

- ［ツール］>［カスタマイズ］を選択します。
- ［アドイン］タブから［SysML］をチェックします。
  これで［SysML］メニューが表示されます。

［モデルビュー］を SysML view と Standard view で切り替えられます。

- ［表示］メニューから［モデルビューの再構成］を選択して、希望するビューを選びます。

注記

SysML の要求部分については、要求プロファイルを使用しています。詳細については、要求のモデリングを参照してください。

SysML Diagram のタイプ

- Activity diagram

- Block definition diagram（クラス図に関連）
- Internal block diagram（合成構造図に関連）
- Parametric block diagram（合成構造図に関連）
- Requirement diagram（クラス図に関連）
- Sequence diagram
- State machine diagram
- Use case diagram

SysML diagram と symbol

Activity diagram

SysML で使用される activity diagram は、通常の UML アクテビティ図です。

Block definition diagram

クラス図に関連しており、以下のシンボルとラインが追加されています。

- Artifact symbol
- Block symbol
- Parametric definition symbol
- Collaboration symbol
- Flow Port
- Flow Specification
- Interface symbol
- Operation symbol
- Package symbol
- Port symbol
- Primitive/enumeration symbol
- Realized interface symbol
- Required interface symbol
- Signal symbol
- State machine symbol
- Stereotype symbol
- Timer symbol
- Association/aggregation/composition line
- Dependency line
- Extension line
- Generalization/realization line
- Text symbol
- Comment symbol
- Constraint symbol
- Stereotype Instance symbol
- Text symbol

- Comment symbol
- Constraint symbol
- Stereotype Instance symbol

Internal block diagram

合成構造図に関連しており、以下のシンボルとラインが追加されています。

- Flow port symbol
- Part symbol
- Port symbol
- Binding line
- Connector line
- Text symbol
- Comment symbol
- Constraint symbol
- Stereotype Instance symbol
- Text symbol
- Comment symbol
- Constraint symbol
- Stereotype Instance symbol

Parametric block diagram

合成構造図に関連しており、以下のシンボルとラインが追加されています。

- Constraint Parameter
- Flow port
- Parametric Use
- Part symbol
- Port symbol
- Binding connector line
- Text symbol
- Comment symbol
- Constraint symbol
- Stereotype Instance symbol

Requirement diagram

クラス図に関連しており、以下のシンボルとラインが追加されています。

- Block symbol
- Package symbol
- Requirement symbol
- Allocate dependency line
- Association/aggregation/composition line
- Dependency line

- Derive dependency line
- Generalization/realization line
- Satisfy dependency line
- Text symbol
- Comment symbol
- Constraint symbol
- Stereotype Instance symbol

Sequence diagram

SysML で使用される sequence diagram は、通常の UML シーケンス図です。

State machine diagram

SysML で使用される State machine diagram は、通常の UML シーケンス図です。

Use case diagram

SysML で使用される use case diagram は、通常の UML ユースケース図です。

SysML diagram タイプ上のステレオタイプ

自動的に適用されるもの

以下のステレオタイプは SysML diagram タイプに自動的に適用されます。

- <<internalBlockDiagram>> （合成構造図に関連）
- <<parametricDiagram>> （合成構造図に関連）
- <<requirementDiagram>> （クラス図に関連）
- <<blockDiagram>> （クラス図に関連）

Diagram に適用可能なステレオタイプ

<<diagramDescription>> Version、Description、Reference および Completeness タグ付き値

Class に適用可能なステレオタイプ

- <<constraint>> Equation タグ付き値
- <<block>>
- <<requirement>> Text、Id および Type タグ付き値

Comment に適用可能なステレオタイプ

- <<rationale>>

Dependency に適用可能なステレオタイプ

- <<allocate>>
- <<binding>>
- <<deriveReqt>>
- <<copy>>
- <<satisfy>>
- <<trace>>
- <<verify>>

ObjectNode と **ActivityEdge** に適用可能なステレオタイプ

- <<continuous>>
- <<discrete>>
- <<overwrite>>
- <<noBuffer>>

ActivityEdge に適用可能なステレオタイプ

- <<optional>>
- <<control>>
- <<stream>>
- <<probability>>、Value タグ付き値

Operation と **Activity** に適用可能なステレオタイプ

- <<controlOperator>>
- <<nullTransformation>>

InformalConstraint に適用可能なステレオタイプ

- <<precondition>>
- <<postcondition>>
- <<resourceConstraint>>
- A ControlValue enumeration: disable、enable

SysML レポート

以下の SysML 固有のレポートが使用できます。

- SysML Dependency matrix
- SysML Dependency Report
- SysML Requirements Report
- SysML Requirements Gap Report

SysML Dependency matrix

マトリクス内の依存についてソースとターゲットを表示します。異なる依存はそれぞれ別のレポートになります。以下のレポートを使用できます。

- All
- All - Reversed
- Allocate
- Allocate - Reversed
- Derive
- Derive - Reversed
- Satisfy
- Satisfy - Reversed
- Verify
- Verify - Reversed

ソース要素が縦、ターゲット要素が横に表示されます。リバースマトリクスを表示すると、その逆になります。

SysML Dependency Report

すべての依存を 1 つの表にリスト表示します。各依存について、以下のプロパティが表示されます。

- Trace kind
- Source name
- Source kind
- Target name
- Target kind

表は、CSV 形式で保存可能です。

SysML Requirements Report

すべての要求を 1 つの表にリスト表示します。各要求について、以下のプロパティが表示されます。

- Id
- Text

参照

要求レポート

SysML Requirements Gap Report

テーブル形式の要求ギャップレポートです。各要求について、以下のプロパティが表示されます。

- Id
- Text
- Incoming requirement dependencies
- Outgoing requirement dependencies
- Connected name
- Connected type

参照

要求レポート

将来サポートされなくなる概念

SysML の仕様は現在も進展中であり、すでに実装された機能でも変更を余儀なくされる可能性があります。実際すでにいくつかの機能はサポートされなくなる予定になっています。

サポートされなくなる対象は `SysMLDeprecated` という名前のパッケージに移動されました。これらの使用は推奨できません。このパッケージ内の要素 (およびステレオタイプインスタンスのような任意のインスタンス) は将来のリリースにおいて削除されます。

これらの概念のデータを喪失しないためには、手動または API を使用したプログラミングによってデータを移動してください。

変更または非推奨となった概念は以下のとおりです :

- <<requirement>>

  <<requirement>> ステレオタイプは 1460 ページの「要求 プロファイル」に置き換えられました。新しいステレオタイプは 2 つの属性、Text と Id のみをもっています。旧ステレオタイプのインスタンスは、新しい要求ステレオタイプのインスタンスと <<requirementDeprecated>> ステレオタイプのインスタンスの、2 つのインスタンスに置き換えられます。旧ステレオタイプのタグ付値は、新ステレオタイプではなくなりますが、<<requirementDeprecated>> ステレオタイプのインスタンスに保持されます。
- verifyMethodKind
- riskKind
- optimizationDirectionKind
- <<effectiveness>>

スケジューラビリティ、パフォーマンス、時刻のプロファイル

このセクションでは、UML Profile for Schedulability、Performance、および Time（別名 UML Real -time profile）のすべてのステレオタイプ、タグ付き値、列挙をリストアップします。

注記

タグ付き値の一部は、テキスト構文を使用した編集のみで可能です。このようなタグ付き値については、本文ではイタリックで表記しています。

RTresourceModeling

GRMacquire

GRMblocking : Boolean

GRMcode

GRMrealize

GRMmapping : GRMmappingString

GRMdeploys

GRMrelease

GRMrequires

RTtimeModeling

RTaction

RTstart : RTtimeValue

RTend : RTtimeValue

RTduration : RTtimeValue

RTclkInterrupt

RTstimulus

RTstart : RTtimeValue

RTend : RTtimeValue

RTclock

RTclockId : Charstring

RTdelay

RTevent

RTat : RTtimeValue

RTinterval

RTintState : RTtimeValue

RTintEnd : RTtimeValue

RTintDuration : RTtimeValue

RTnewClock

RTnewTimer

RTtimerPar : RTtimeValue

RTpause

RTreset

RTset

RTtimePar : RTtimeValue

RTstart

RTtime

RTkind : RTkindEnum

RTtimeout

RTtimer

RTduration : RTtimeValue

RTperiodic : Boolean

RTtimeService

RTtimingMechanism

RTstability : Real

RTdrift : Real

RTskew : Real

RTmaxValue : RTtimeValue

RTorigin : Charstring

RTresolution : RTtimeValue

RToffset : RTtimeValue

RTaccuracy : RTtimeValue

RTcurrentVal : RTtimeValue

RTkindEnum

リテラル :

- dense
- discrete

RTconcurrencyModeling

CRaction

CRatomic : Boolean

CRasynch

CRconcurrent

CRcontains

CRdeferred

CRimmediate

CRthreading : CRthreadingEnum

CRmsgQ

CRsynch

CRthreadingEnum

リテラル：

- local
- remote

SAprofile

SAaction

SApriority : Integer

SAblocking : RTtimeValue

SAdelay : RTtimeValue

SApreempted : RTtimeValue

SAready : RTtimeValue

SArelease : RTtimeValue

SAworstCase : RTtimeValue

SAabsDeadline : RTtimeValue

SAlaxity : SAlaxityEnum

SArelDeadline : RTtimeValue

SAengine

SAaccessPolicy : SAaccessControlPolicyEnum

SAcontextSwitch : TimeFunction

SAschedulable : Boolean

SApreemptible : Boolean

SApriorityRange : Range

SArate : Real

SAschedulingPolicy : SAschedulingPolicyEnum

SAutilization : Real

SAaccessPolParam : Real

SAowns

SAprecedes

SAresource

SAacquisition : RTtimeValue

SAcapacity : Integer

SAdeacquisition : RTtimeValue

SAconsumable : Boolean

SAaccessControl : SAaccessControlPolicyEnum

SAptyCeiling : Integer

SApreemptible : Boolean

SAaccessCtrlParam : Real

SAresponse

SAutilization : Real

SAspare : RTtimeValue

SAslack : RTtimeValue

SAoverlaps : Integer

SAschedRes

SAscheduler

SAschedulingPolicy : SAschedulingPolicyEnum

SAsituation

SAtrigger

SAschedulable : Boolean

SAoccurrence : RTarrivalPattern

SAendToEnd : Charstring

SAusedHost

SAuses

SAlaxityEnum

リテラル：

- hard
- soft

SAschedulingPolicyEnum

リテラル：

- rateMonotonic
- deadlineMonotonic
- HKL
- fixedPriority
- minimumLaxityFirst
- maximizeAccruedUtility
- MinimumSlackTime

SAaccessControlPolicyEnum

リテラル：

- FIFO
- priorityInheritance
- noPreemption

- highestLockers
- priorityCeiling

PAprofile

PAclosedLoad

PArespTime : PAperfValue

PApriority : Integer

PApopulation : Integer

PAextDelay : PAperfValue

PAcontext

PAhost

PAutilization : Real

PAschdPolicy : PAschdPolicyEnum

PArate : Real

PActxtSwT : PAperfValue

PAprioRange : Range

PApreemptable : Boolean

PAthroughput : Real

PAopenLoad

PArespTime : PAperfValue

PApriority : Integer

PAoccurrence : RTarrivalPattern

PAresource

PAutilization : Real

PAschdPolicy : PAschdPolicyEnum

PAcapacity : Integer

PAaxTime : PAperfValue

PArespTime : PAperfValue

PAwaitTime : PAperfValue

PAthroughput : Real

PAstep

PAdemand : PAperfValue

PArespTime : PAperfValue

PAprob : Real

PArep : Integer

PAdelay : PAperfValue

PAextOp : PAextOpValue

PAinterval : PAperfValue

PAschdPolicyEnum

リテラル :

- FIFO
- priority

RSAprofile

RSAclient

RSAtimeout : RTtimeValue

RSAclPrio : Integer

RSAprivate : Integer

RSAconnection

RSAshared : Boolean

RSAhiPrio : Integer

RSAloPrio : Integer

RSAmutex

RSAorb

RSAserver

RSAsrvPrio : Integer

RSAchannel

RSAschedulingPolicy : RSAschedulingPolicyEnum

RSAaverageLatency : RTtimeValue

RSAschedulingPolicyEnum

リテラル :

- FIFO
- RateMonotonic
- DeadlineMonotonic
- HKL
- FixedPriority
- MinimumLaxityFirst
- MaximizeAccruedUtility
- MinimumSlackTime

5

エラー メッセージと警告メッセージ

このドキュメントは、UML ツール セットから表示されるエラー メッセージと警告メッセージのリファレンス ガイドとしてご利用いただけます。

一般的なアプリケーションのエラーと警告

Tau の minidump ファイル （Windows）

Tau には、Windows プラットフォームの機能を取り込む デバッグ情報が組み込まれています。実行中に Tau が強制終了されて minidump ファイルが作成されたことを示すメッセージが表示されたら、最寄りの IBM Rational サポートにご連絡ください。minidump ファイルには、現在の呼び出しスタックが含まれており、どの呼び出しが実行されたか特定するのに役立ちます。内部のツール呼び出しが原因でエラーが発生したのかどうかを判断するため、まだ特定されていない問題を解決できることもあります。また OS 上の依存関係と第三者呼び出しを考慮するので、物理的な外部環境とのインテグレーションを改善し、Tau が依存している他社のソフトウェアの要件を明確にできます。

図 166: 強制終了時のメッセージボックス

Minidump ファイル の場所

minidump ファイルは、デフォルトでローカル設定ディレクトリに作成されますが、環境変数を使用して、格納場所を変更することもできます。

デフォルトの格納場所

```
C:¥Documents and Settings¥<user>¥Local Settings¥Temp
```

環境変数を使用して格納場所を変更する例

```
TAU_DUMP_PATH=c:¥DevTools¥IBM¥Rational¥minidumps¥
```

Minidump ファイルの内容

minidump ファイルには呼び出しスタックとレジスタのみが含まれ、メモリはありません。つまり、minidump ファイルが作成された元のモデルに関する情報は含まれません。

ビルドから発生するエラーと警告

UML 設計時には許可されてもコード生成時には構造上の制約がある場合があるので注意する必要があります。このような制約は、それぞれのビルド タイプ（コード ジェネレータ）によって異なります。

フェーズと識別子

UML モデルを別の言語または形式に変換するプロセスには複数のフェーズがあります。モデルの処理中に、問題が起こった場所を識別できるようフェーズごとにエラー メッセージと警告メッセージが表示される場合があります。ビルド アーティファクトのプロパティで［Verbose mode］をオンにすると、ビルド プロセスから可能な限り多く情報を取得できます。フェーズを識別する接頭辞は以下のとおりです。

- TSX：構文分析
- TSC: セマンティック チェック
- TNR: 名前解決
- TAB: アプリケーション ビルド
- TCI: C/C++ インポート
- TIL: 中間言語
- TCC: C コードの生成
- TCG: C++ 生成
- TSI, OGC: SDL のインポート
- TUI: UML インポート

TSX：構文分析

構文分析では、UML による構築を正しく行えるように言語要素がどのように構築されて集成されているかをチェックします。

TSC: セマンティック チェック

セマンティック チェックでは、UML モデルが完全で、言語構成要素間の関係に意味があることをベリファイします。

コードジェネレータステレオタイプごとのセマンティックチェックに関するメッセージのリストはこちらをご覧ください。

TNR: 名前解決

名前解決では、UML エンティティの名前を識別して、モデル内の正しい定義へのバインドを試行します。

TAB: アプリケーション ビルド

アプリケーション ビルダは、別の表現またはアプリケーションを UML モデルから生成するプロセス全体を管理します。

TCI: C/C++ インポート

C/C++ インポートでは、外部 C または C++ ヘッダー ファイルを UML パッケージにインクルードして、データ型宣言を UML データ モデルに変換します。

TIL: 中間言語

中間言語フェーズでは、効率の良い C コード生成に必須な UML モデルから中間表現への変換を行います。その後に、アプリケーションのビルドに必要な C コードと make ファイルを出力する C コード ジェネレータによって中間モデルが使用されます。

TCC: C コードの生成

中間モデルが C コード ジェネレータ用の入力に使用されて、C コードとアプリケーションをビルドするのに必要な make ファイルが出力されます。この段階からのほとんどの出力では、コンパイラ エラー メッセージまたは警告メッセージが表示されますが、ジェネレータ自体からメッセージが表示される場合もあります。

TCG: C++ 生成

C++ 生成フェーズでは、UML モデルに基づいて一連の C++ 宣言を作成します。

TSI, OGC: SDL のインポート

SDL インポート操作時に生成されるエラー メッセージは、出力ウィンドウの［スクリプト］タブに出力されます。

TUI: UML インポート

UML インポート操作時に生成されるエラー メッセージは、出力ウィンドウの［スクリプト］タブに出力されます。

TSX：構文分析

構文分析では、UML による構築を正しく行えるように言語要素がどのように構築されて集成されているかをチェックします。

構文エラーのほとんどの直接的要因は、UML モデルで特定できます。

このフェーズで発生するエラーと警告には、TSX という接頭辞が付きます。

```
Internal error: <string>
```

これらのエラーは発生してはならないエラーです。発生した場合は、Tau サポートにご連絡ください。

TSX0026: ポートに 2 つの in part または out part を含めることはできません

ツールを通常の方法で使用している場合、このエラーは発生しません。不正なカスタマイズまたはアドインが行われていると、そのようなモデルが作成される場合があります。その場合は、カスタマイズまたはアドインを修正する必要があります。

TSX0047: タグ付き値はここでは使用できません

たとえば、クラス シンボル内などでは、プロパティ（タグ付き値）を編集できない場合があります。ステレオタイプ自体のみを追加できます。

プロパティの編集には、プロパティ エディタを使用することを推奨します。

TSC: セマンティック チェック

セマンティック チェックについて

セマンティック チェックでは、UML モデルが完全で、言語構成要素間の関係に意味があることをベリファイします。

モデル内に不完全な構成要素が含まれていると、セマンティック エラーが発生します。サポートされている構成要素を識別するには、UML 言語ガイドが役立ちます。

このフェーズで発生するエラーと警告には、TSC という接頭辞が付きます。

コードジェネレータステレオタイプごとのセマンティックチェックに関するメッセージのリストはこちらをご覧ください。

TSC0123: 再帰的な依存が、%n の定義で見つかりました <string>%s 経由)

これは循環依存性エラーです。2 つのクラスを同時に他のクラスのコンテナにすることはできないので、これは不正と見なされます。

このタイプのエラーの例を以下に示します。

例 92

```
class X {
    part Y y;
}

class Y {
    part X x;
}
```

TSC0134: C コードを生成する場合、遷移は、stop、nextstate または join action で終了する必要があります

分岐では、「else」などのすべての答えの可能性を含める必要があります。

TSC0092: 対応する 'virtual(仮想)' または 'redefined(再定義)' 操作が親シグニチャで見つかりませんでした (または存在しません)。

このエラーは要因としてさまざまな状況が考えられます。以下の例は、起こり得る状況を示しています。

汎化がないアクティブ クラスでの再定義操作の使用

例 **93:** 汎化のないクラス

```
active class P {
  redefined void Op() { }
}
```

アクティブ クラスの汎化で再定義操作を使用すると、このエラーが発生する場合があります。

例 **94:** 親クラスでの操作の不一致

```
active class P {
}
active class C :P {
     redefined void Op() { }
}
```

親クラスでの操作（Op）に異なるシグニチャがあると、以下のような状況になる場合があります。

例 **95:** 仮想性は「**virtual**」または「**redefined**」でなければならない

非仮想操作を再定義することはできません。

```
active class P {
     void Op () {   }
}
active class C :P {
    redefined void Op() { }
}
```

例 **96:** 異なる戻り型

```
active class P {
    virtual Integer Op () { return 1; }
}
active class C :P {
    redefined void Op() { }
}
```

例 **97:** 異なる仮パラメータのカウント

```
active class P {
    virtual void Op (Integer x) {  }
}
active class C :P {
    redefined void Op() { }
}
```

例 **98:** 異なるタイプの仮パラメータ

```
active class P {
    virtual void Op (Integer x) {  }
}
active class C :P {
    redefined void Op(Real x) { }
}
```

TSC0196: ファイナライズされた操作を再定義することはできません。

親クラス内の操作はファイナライズされていますが、子と同じシグニチャになっています。

例 **99:** ファイナライズされた操作

```
active class P {
    finalized void Op () { }
}
active class C :P {
    redefined void Op() { }
}
```

TSC0236: 操作 **'<name>'** はポート上で **'Realized' (** 実現化 **)** として指定することはできません。

このチェックによって以下のケースが検出されます。

```
active class <class name>
{
    port <port name> in with <in_name>;
}
```

この場合の `<in_name>` は、同じ名前の操作にバインドされます。

例 **100**

```
active class a {
   void foo()  {}
   port p in with foo;
}
```

これはエラーとして報告されます。この問題を解決するには、アクティブ クラス a のインターフェイスで `foo()` を定義する必要があります。

TSC0237: 操作 **'<name>'** はポート上で **'Required' (** 要求 **)** として指定することはできません。

このチェックによって以下のケースが検出されます。

```
active class <class name>
{
    port <port name> out with <out_name>;
}
```

この場合の <out_name> は、同じ名前の操作にバインドされます。

例 **101**

```
active class a {
   void foo()  {}
   port p out with foo;
}
```

これはエラーとして報告されます。この問題を解決するには、アクティブ クラス a のインターフェイスで foo() を定義する必要があります。

TSC2300: 式 **'any (type)'** には **interface** 型を定義できません

このタイプのエラーの例を以下に示します。

例 **102**

```
interface I {
}

active class X {
    Integer Op () {
        switch (any (I)) {
            case 5  :{ return 1; }
            default :{ return 0; }
        }
    }
}
```

TSC2302: データ型からの関連は、誘導可能リモート関連を終端にすることはできません。

データ型に属性を設定できないので、データ型からの関連付けがあると不正と見なされます。誘導可能性は常にデータ型に従っている必要があります。

ツールを通常の方法で使用している場合、このエラーは発生しません。不正なカスタマイズまたはアドインが行われていると、そのようなモデルが作成される場合があります。その場合は、カスタマイズまたはアドインを修正する必要があります。

TSC2303: 関連の **1** つの終端のみを集約または合成にできます。

集約と合成は異なる種類の「part-of」構成要素なので、2 つのクラスを異なるクラスのコンテナにすることはできません。

例 **103**

この状況は 358 ページの図 167 のような状況で発生します。

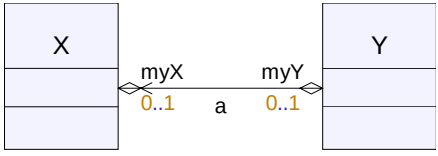


図 167: 循環参照のあるクラス

TSC2304: パートでない属性には、初期カウントを与えることはできません。

UML では、正規の属性に初期カウントを指定できません。パートのみに有効です。

このタイプのエラーの例を以下に示します。

例 **104**

```
class Z {
    Integer [1..*] a / 1;
}
```

TSC2305: パートにはデフォルト値を与えることはできません。

パートはアクティブ クラスのインスタンスなので、デフォルト値を指定できません。このタイプのエラーの例を以下に示します。

例 **105**

```
active class X {
    part Y a = 10;
}
```

TSC2306: 合成属性や関連の終端を、データ型により型指定することはできません。

UML でパートと見なされる合成属性にも、データ型のインスタンスを使用できません。

このタイプのエラーの例を以下に示します。

例 **106**

```
class X {
    part Integer d;
}
```

TSC2307: 合成属性にはこの属性を所有する型を **(** 直接または間接的に **)** 指定できません。

これは循環依存性エラーです。クラスはそれ自体のコンテナにできないので、これは不正と見なされます。

このタイプのエラーの例を以下に示します。

例 **107**

```
class X {
    part X y;
}
```

TSC2308: 呼び出し式の **'via'** は、ポートを参照する必要があります。

このタイプのエラーの例を以下に示します。

例 **108**

```
class Y {}
signal sig ();
active class X {
    port p out with sig;
    void Op () {
        output sig via Y;
    }
}
```

TSC0269: インターフェイス **I** とクラス **Y** の間の汎化は無効です。

このタイプのエラーの例を以下に示します。

例 **109**

```
class Y {
}
interface I :Y {
}
```

TSC2325: 継承の循環

このエラーは、シグニチャが直接または間接的にそれ自体に基づいている場合に発生します。

このタイプのエラーの例を以下に示します。

例 **110**

```
class X :Y {
}
class Y :X {
}
```

TSC4001: C コードを生成する場合、戻り値は代入式の左辺で処理する必要があります。

たとえば、操作を返す値からの戻り値は無視できません。そのような戻り値は、たとえば属性に保存する必要があります。

例 **111**

整数を返す以下のような操作 Op を考えます。

```
Op ():Integer
```

ここで Op を呼び出します。

```
...
Integer i;
...
i=Op(); // Correct way of calling Op
Op(); // Error is reported
...
```

このチェックが実行されるのは、意味チェッカーがいずれかの C コード ジェネレータとビルド タイプが関連するビルドのコンテキストで実行される場合のみです（モデル ベリファイヤ（Model Verifier）、C コード ジェネレータ、AgileC コード ジェネレータ）。

TNR: 名前解決

名前解決では、UML エンティティの名前を識別して、モデル内の正しい定義にバインドします。名前解決エラーは、モデル内の不一致によって発生します。たとえば、不明瞭になり確実には解決できないような名前の変更をエンティティ上で行うとこのエラーが発生します。

このフェーズで発生するエラーと警告には、TNR という接頭辞が付きます。

エラーの**主体**が UML エンティティではなくプロジェクト ファイル（`.ttp` ファイル）に関連している TNR エラーの場合は、必ず Tau サポートに報告してください。

TNR0023: 要素の参照 <name> を見つけることができませんでした

エンティティはその名前（名前バインド）またはその GUID（GUID バインド）によって探すことができます。

名前バインドでは、名前を使用して現在の範囲内のエンティティを参照します。GUID バインドとは、エンティティがその固有な ID（GUID）によって参照されるという意味です。つまり、何らかの理由でエンティティが削除され、モデル内のいずれかの場所でその GUID によって参照されるとエラーが発生します。

問題を解決するには、正しい GUID と一緒にエンティティをロードして、参照を削除するか名前バインドが使用されるように参照を変更します。

TAB: アプリケーション ビルド

アプリケーション ビルダーは、別の表現またはアプリケーションを UML モデルから生成するプロセス全体を管理する Tau アプリケーションです。

このプロセスの必須ツールセットから発生するエラーは、開発プラットフォームが正しく設定されていない場合に発生します。リリース ノートに記述されている制約と無関係なエラーが発生した場合は、必ず Tau サポートに報告してください。

このフェーズで発生するエラーと警告には、TAB という接頭辞が付きます。

TCI: C/C++ インポート

C/C++ インポートでは、外部 C または C++ ヘッダー ファイルを UML パッケージにインクルードして、データ型宣言を UML データ モデルに変換します。

このフェーズでは、C/C++ に適合している必要がある外部データ モデルを扱います。サポートされている構成要素と作成される UML モデルについては、第 9 章「C/C++ のインポート」の 462 ページ、「動作原理」で詳しく説明しています。

このフェーズで発生するエラーと警告には、TCI という接頭辞が付きます。

TIL: 中間言語

中間言語フェーズでは、効率の良い C コード生成に必須な UML モデルから中間表現への変換を行います。その後に C コード ジェネレータによって中間モデルが使用されます。

このフェーズでは、意味的に UML に適合することが判明しているモデルを操作します。リリース ノートに記述されている制約とは無関係なエラーが発生した場合は、必ず Tau サポートに報告してください。

状況を診断しやすいように、完全なモデルを Tau サポート宛てに送付してください。問題に迅速に対処いたします。

環境変数 TTD_INTERMEDIATE_U2_MODEL を 1 に設定してビルドを再実行すると、技術サポートは状況を判断しやすくなります。その手順を実行すると、コードの生成に必要な変換を表す中間モデルがファイルに保存されます。ファイルは `intermediate_model.u2` と命名されて、ターゲット ディレクトリとして定義されているフォルダに保存されます。これを UML モデルと一緒に提供すると、技術サポートは問題を迅速に解決できます。

このフェーズで発生するエラーと警告には、TIL という接頭辞が付きます。

TCC: C コードの生成

中間モデルが C コード ジェネレータ用の入力に使用されて、C コードとアプリケーションをビルドするのに必要な make ファイルが出力されます。この段階からのほとんどの出力では、コンパイラ エラー メッセージまたは警告メッセージが表示されますが、C コード ジェネレータ自体からメッセージが表示される場合もあります。

状況を診断しやすいように、完全なモデルを Tau サポート宛てに送付してください。問題に迅速に対処いたします。

このフェーズで発生するエラーと警告には、TCC という接頭辞が付きます。

TCG: C++ 生成

C++ 生成フェーズでは、UML モデルに基づいて一連の C++ 宣言を作成します。

このフェーズでは、意味的に UML に適合することが判明しているモデルを操作します。リリース ノートに記述されている制約とは無関係なエラーが発生した場合は、必ず Tau サポートに報告してください。

状況を診断しやすいように、完全なモデルを Tau サポート宛てに送付してください。問題に迅速に対処いたします。

このフェーズで発生するエラーと警告には、TCG という接頭辞が付きます。

UML によるモデルのベリファイ

「UML によるモデルのベリファイ」セクションの各章では、UML モデルでのシミュレーション機能とテスト機能の使用方法について説明しています。

6

アプリケーションのベリファイ

モデル ベリファイヤ（Model Verifier）により、UML モデルの振る舞いを検証して、実装が正しいことを確認できます。

モデル ベリファイヤ（Model Verifier）対応のアプリケーションをビルドするときは、C コード ジェネレータでアプリケーションをビルドする場合と同じような手順を実行します。このセクションでは、基本的なビルド機能とモデル ベリファイヤ（Model Verifier）の使用方法について説明します。

参照

第 7 章「モデル ベリファイヤ（Model Verifier）リファレンス」
第 44 章「C++ アプリケーションの デバッグ」

モデル ベリファイヤ（**Model Verifier**）の概要

モデル ベリファイヤ（Model Verifier）により、UML モデルの振る舞いを検証して、実装が正しいことを確認できます。アプリケーション ビルダを使用して、モデルから C コード実行形式プログラム、つまりモデルベリファイヤ対応のアプリケーションを生成し、シミュレーション用にカスタマイズされた定義済みランタイム ライブラリにリンクします。モデルをシミュレートする、とは、さまざまなコマンドとブレークポイントを使用して実行形式プログラムを実行するという意味です。シミュレーションは、自動的に実行するか、遷移を段階的に追跡したりシグナルを送信して手動で行うことができます。

ユーザー インターフェイスまたは Model Verifier コンソール コマンドを使用して、モデル ベリファイヤ（Model Verifier）を制御できます。シミュレーション セッションの実行中に、実装のさまざまな側面を表示できます。モデルの内部的な振る舞いのみを確認したり、環境との間のシグナルの送受信のみを検証したりできます。

シミュレーション セッションの実行は、状態機械図またはシーケンス図で図式的にトレースするか、出力ウィンドウまたはシーケンス図でテキスト形式によってトレースできます。シミュレーション セッションでは、テキスト ファイルとして保存できるログが生成されます。

アプリケーションで環境との通信が行われる場合は、その動作をシミュレートすることもできます。メッセージを送信して、環境からモデルに着信するシグナルをシミュレートします。

シミュレーション セッションの実行中は、シミュレーションの多様な観点を表示するさまざまなビューを使用できます。このようにして、値やステップなどを監視できます。

モデル ベリファイヤ（Model Verifier）を使用してアクティビティをシミュレートすることもできます。詳細については、アクティビティ シミュレーションを参照してください。

モデルベリファイヤ対応のアプリケーションの生成

モデルベリファイヤ対応のアプリケーションを生成するには、いくつかの方法を使用できます。必要に応じて使用する方法を決めてください。

重要！

実行形式のモデル ベリファイヤ（Model Verifier）アプリケーションを生成するには、C/C++ コンパイラをインストールしておく必要があります。

ビルド アーティファクトを使用したビルド

この方法では、デフォルトのビルド設定が不十分な場合に、ステレオタイプと属性を適用してビルド設定を指定し、アプリケーションのビルドをカスタマイズできます。

ビルド アーティファクトを使用してモデル ベリファイヤ（Model Verifier）をビルドするには、以下の手順を行います。

- 必要に応じて、ビルド アーティファクトを作成します。右クリックしてショートカット メニューから［Model Verifier］を選択し、サブメニューから［新しいアーティファクト］を使用します。
- 目的のモデル ベリファイヤ（Model Verifier）のビルド アーティファクトを右クリックします。ショートカット メニューから［ビルド (Model Verifier)］を選択して、サブメニュー［ビルド］を選択します。

ヒント

このビルド アーティファクトは次回以降のモデル ベリファイヤ（Model Verifier）セッションのビルドと起動に再利用できます。ビルドごとに新しいビルド アーティファクトを作成する必要はありません。

選択したモデル要素のビルド

ビルド アーティファクトはビルドを行うための必須要素です。したがって、最初にビルド アーティファクトを作成せずに UML モデル要素上でモデル ベリファイヤ（Model Verifier）のビルドを行うことはできません。

ただし、ツールでは以下のように必要な手順を簡単に実行できます。

1. ［モデル ビュー］で目的のビルド ルートを右クリックします。
2. メニュー項目の［Model Verifier］を選択して、［新しいアーティファクト］を選択します。
   - モデル ベリファイヤ（Model Verifier）のビルドに適したデフォルト設定付きのビルド アーティファクトが **ArtifactNNNN** というデフォルト名で作成されます。
3. ここでもう一度コマンド［ Model Verifier］を選択して、サブメニューから［起動］を選択します。

構成を使用したビルド

この方法では、複数のビルド アーティファクトを同時にビルドできます。ただし、それらの複数のビルド アーティファクトを同時にモデル ベリファイヤ（Model Verifier）ビルド アーティファクトとして使用することはできません。同時に実行できるモデル ベリファイヤ（Model Verifier）のインスタンスが 1 つに限られるためです。

注記

アクティビティ モデルをシミュレートするには、特定のアドイン（ADSim）を使用する必要があります。このアドインは、シミュレートするアクティビティからビルド アーティファクトを自動生成します。このビルド アーティファクトは、上記で説明しているように、通常どおり起動できます。アクティビティ シミュレーションの詳細については、アクティビティ シミュレーションを参照してください。

参照

第 20 章「アプリケーション ビルド リファレンス」

モデル ベリファイヤ（**Model Verifier**）の実行

モデル ベリファイヤ（Model Verifier）をエラーなしで正しくビルドできたら、モデル ベリファイヤ（Model Verifier）を起動できます。モデル ベリファイヤ（Model Verifier）を起動すると、ワークスペース ウィンドウに［インスタンス］タブが表示されます。出力ウィンドウに［Model Verifier］タブが表示されます。

モデル ベリファイヤ（**Model Verifier**）の起動

前のビルドで作成したモデル ベリファイヤ（Model Verifier）実行形式プログラムを起動するか、1 つのコマンドでビルドしてモデル ベリファイヤ（Model Verifier）を起動できます。

リビルドなしのモデルベリファイヤの起動

すでにモデル ベリファイヤ（Model Verifier）をビルドしている場合は、このコマンドを使用できます。リビルドをスキップするため起動プロセスの実行時間が短縮されますが、チェックは実行されません。したがって、モデル ベリファイヤ（Model Verifier）が、ワークスペースに現在ロードされているモデルに基づいているかどうかを確認することはできません。そのように関連付けられていない場合、またはビルド後にモデルが変更されている場合は、モデル ベリファイヤ（Model Verifier）によって間違った情報または不完全な情報が表示されることがあります。

1. ［ビルド］メニューから［Model Verifier の開始］を選択します。
2. 表示されるダイアログで、モデル ベリファイヤ（Model Verifier）実行形式プログラムが含まれるファイル（このファイルは前のビルドで作成されている必要があります）の名前と場所を指定します。
   - ［参照］ボタンは標準の［開く］ダイアログを使用してファイル システムを参照できる便利な機能です。

［OK］ボタンをクリックすると、Tau はファイルにモデル ベリファイヤ（Model Verifier）実行形式プログラムが含まれているかどうかを簡単にチェックしてから、指定された実行形式プログラム上でモデル ベリファイヤ（Model Verifier）の起動を試みます。

ビルド アーティファクトを使用したビルド後の起動

この方法はすでにモデル ベリファイヤ（Model Verifier）のビルド アーティファクトを作成している場合に便利です。

1. ［モデル ビュー］で、起動するモデル ベリファイヤ（Model Verifier）を表すビルド アーティファクトを右クリックします。
2. ショートカット メニューから［ビルド (Model Verifier)］を選択して、メニュー項目［起動］を選択します。

モデル ベリファイヤ（Model Verifier）がビルドされます。ビルドに成功した場合は、新規にビルドされたモデル ベリファイヤ（Model Verifier）が起動されます。

選択したモデル要素のビルド後の起動

モデル ベリファイヤ（Model Verifier）の適切なビルド アーティファクトを作成していない場合は、以下の手順を実行します。

1. ［モデル ビュー］で、ビルド ルートにするクラスを右クリックします。
2. ショートカット メニューから［Model Verifier］を選択して、［新しいアーティファクト］を選択します。
   - **ArtifactNNNN** という名前の新しいビルド アーティファクトが作成されます。
   - 必要に応じて、新規に作成されたビルド アーティファクトの名前を変更し、ビルド設定を調整します（オプション）。
3. ショートカット メニューからもう一度［Model Verifier］を選択し、新規に作成されたアーティファクトの名前（デフォルトで **Artifact0001**）を選択して、最後に［起動］を選択します。

構成を使用したビルド後の起動

この方法は、複数のビルド アーティファクトをビルドし、そのうちの 1 つをモデル ベリファイヤ（Model Verifier）とし、新規にビルドしたモデル ベリファイヤ（Model Verifier）を起動する場合に便利です。1 つの特殊な状況として、構成にモデル ベリファイヤ（Model Verifier）ビルド アーティファクトが 1 つのみ含まれている場合があります。

- ［プロジェクト］ツール バーの［構成の実行］ボタンをクリックします。アクティブな構成に含まれているすべてのビルド アーティファクトのビルドが行われ、新規にビルドされたモデル ベリファイヤ（Model Verifier）が起動されます。

注記

同時に複数のモデル ベリファイヤ（Model Verifier）は起動することはできません。複数のモデル ベリファイヤ（Model Verifier）ビルド アーティファクトが検出された場合、すべてがビルドされますが起動されるのは 1 つのみです。

注記

Windows Vista を使用している場合、生成アプリケーションがポートを使用できるようにするため、モデルベリファイヤの実行ユーザーには管理者権限が必要です。

モデル ベリファイヤ（**Model Verifier**）の終了

モデル ベリファイヤ（Model Verifier）を終了するには、以下のいずれかの手順を実行します。

- ［ベリファイ］メニューから［Model Verifier の停止］を選択する。
- ［ビルド］メニューから［停止］を選択する。

インスタンス

モデル ベリファイヤ（Model Verifier）を起動すると、［インスタンス］タブが表示されます。［インスタンス］タブには以下のものが表示されます。

- 実行中のアクティブ クラス インスタンスのツリー。継承関係とインスタンスがフラット化され、現在有効なアクティブ クラス インスタンスとそのインスタンス番号を表す新しいノードが表示されます。
- ready queue、タイマー 待ち行列、システム環境インスタンス、そのオブジェクトが存在する場合のアクティブ タイマー リストなどのランタイム依存オブジェクト。

属性と仮パラメータは、アクティブ クラスインスタンス ノードの子ノードとして表示されます。ただし、それぞれの値は［ウォッチ］ウィンドウに表示されます。

以下のものを表すために暗黙の属性が作成されます。

- 状態機械の制御状態、状態属性
- データ型 Pid の定義済み識別子（Sender、Parents、Offsprings など）
- アクティブ クラスのインスタンスの呼び出しスタック、CallStack 属性
- アクティブ クラスのインスタンスのメッセージ待ち行列、Queue 属性

ReadyQueue

ReadyQueue オブジェクトは識別子のリストとして表示されます。識別子の構文は次のとおりです：`instance_name[instance_number]`。

以下の情報が識別子と同じラインに表示されます。

- アクティブ クラス インスタンスの現在の状態：**<state_name>**。インスタンスが遷移の実行中、つまりその状態から移動している場合は、次のような状態が表示されます：**(state_name)**。
- 次に使用されるシグナル：**>signal_name**
- メッセージ待ち行列の長さ：**]number**

TimerQueue

TimerQueue オブジェクトの要素は、type Time の追加フィールドがあるメッセージ内の要素と似ています。

実行のトレース

モデル ベリファイヤ（Model Verifier）の実行時に、実行トレース情報を入手できます。これによって、アプリケーション内の各遷移とイベントを追跡できます。以下の 3 種類のトレース方法から選択できます。

- テキストのトレース
- UML モデルの追跡
- シーケンス図のトレース

上記の 3 つの方法を任意に組み合わせて使用できます。

テキスト トレース

テキスト トレースでは、シミュレーションで実行された各ステップが出力ウィンドウの［Model Verifier］タブに表示されます。出力のトレース レベルを設定して、表示される情報の範囲を選択できます。0 ～ 6 のトレース レベルを指定できます。デフォルトではトレース レベル 1 が設定されています。レベル 0 を指定すると、テキスト トレースが無効になります。

トレース レベルを適用するユニットを指定することもできます。1 つのユニットは 1 つのアクティブ クラス インスタンスです。ユニットを指定しないと、トレース設定はすべてのユニットに適用されます。

トレース レベルの変更は、Model Verifier コンソールの入力を使用してのみ変更できます。

- トレース レベルを変更するには、「`set-trace <` 任意のユニット名 `> <trace-level>`」と入力します。
  - ユニット名として「?」を入力すると、使用可能なすべてのユニットのリストが表示されます。
- テキスト トレースを無効にするには、「`set-trace 0`」と入力します。

参照

テキスト トレース レベル

Set-Trace

カスタム テキスト トレース

カスタム テキスト メッセージをモデル ベリファイヤ コンソールに出力して、トレースできます。アプリケーション固有のデバッグ情報を出力するのに利用でき、内蔵トレース情報を補完できます。

コンソール ウィンドウでテキスト メッセージをトレースするには、ユーティリティ関数 xPrintString() を使用できます。

この関数はモデル ベリファイヤ（Model Verifier）のライブラリのファイル `scttypes.h` で定義されています。

例 **112: xPrintString()** の使用例

```
#include <scttypes.h> /* For the definition */
...
xPrintString("Bugs Bunny¥n");
...
```

トレース前の文字列のフォーマッティング

`xPrintString()` は、トレースのためにフォーマットされた文字列を必要とします。文字列をフォーマットする必要がある場合、`xPrintString()` の呼び出し前にフォーマットするか（378 ページの例 113 を参照）、あるいは関数 `xWriteBuf_Fmt()` を使用します。この関数には ”printf” 形式の引数を使用できます。

例 **113:** 文字列のフォーマッティング

以下の C の文で表される結果を取得するには、

```
printf("x=%d¥n", 4);
```

以下の 2 つのステップを実行します。

```
char str[20]; /* array long enough to hold the result */
sprintf(str, "x=%d¥n", 4);
xPrintString(str);
```

あるいは :

```
xWriteBuf_Fmt("x=%d¥n", 4);
```

アプリケーションのテキスト トレース

ターゲット アプリケーション実行時のトレース メッセージを表示できる機能を持つモデル ベリファイヤ（Model Verifier）を実行している場合、テキスト トレース機能を組み合わせることもできます。トレース コードを管理してモデル ベリファイヤ（Model Verifier）セッションとターゲット アプリケーション セッション間でその使用を区別するには、以下の重要な C マクロ（XTRACE などの）に着目し、条件付きコンパイルによってその 2 つを組み合わせることができます。

例 **114:** モデル ベリファイヤ（**Model Verifier**）トレースとアプリケーション トレースの組み合わせ

```
#ifdef XTRACE
  /* XTRACE is defined for Model Verifier */
```

```
  #define myprintf(S) xPrintString(S)
#else
  /* If not Model Verifier, then assume application*/
  #define myprintf(S) printf(S)
#endif
```

UML モデルの追跡

この追跡メソッドにより、シミュレートされたモデルの定義に使用された UML ダイアグラムでの実行を追跡できます。たとえば、UML モデルのステート チャート図、クラス図、テキスト図内の状態機械遷移とステートメントの実行を追跡できます。ADSim アドインを使用して、アクティビティ図で実行を追跡することもできます（詳細についてはアクティビティ シミュレーションを参照）。

追跡を開始すると、選択されたステートメントが開いたダイアグラムが表示されます。実行を続行するに従って、ダイアグラム内の次のステートメントが順次にハイライト表示されます。

注記

UML モデルの追跡はデフォルトで有効になっています。

UML モデルの追跡の有効化

UML モデルの追跡を有効にするには、以下のいずれかのタスクを実行します。

- ［ベリファイ］メニューから［次のステートメントを表示］をクリックする。
- ［Model Verifier］ツール バーの［次のステートメントを表示］ボタンをクリックする。

ボタンを押すと、UML モデルの追跡が有効になります。

UML モデルの追跡の無効化

UML モデルの追跡を無効にするには、再度［次のステートメントを表示］ボタンをクリックします。

状態機械とアクティビティ追跡モード

デフォルトで、モデル ベリファイヤ（Model Verifier）は**状態機械 モード**で実行されます。このモードは、実行の現段階、つまり実行直前のステートメントを表示することを主な目的としています。実行直前のシンボルまたはシンボル内の文の横に、緑の三角形が挿入されます。

トークン フローをベースとした実行モデルを使用するアクティビティ モデルをシミュレートする場合は、**アクティビティ モード**のほうが適しています。このモードの主目的は、トークン フローの結果としてのアクティビティ ノードの実行の追跡です。アクティビティ ノードの実行は、実行中のアクティビティ ノード シンボルを選択して、テキスト形式とグラフィック形式で追跡できます。

モデル ベリファイヤ（Model Verifier）を起動すると、モデル ベリファイヤはモデルにとって適切な追跡モードを自動的に選択します。Model Verifier コンソールで `statemachine-mode` コマンドと `activity-mode` コマンドを使用して、必要に応じて状態機械 モードとアクティビティ追跡モードを切り替えられます。

注記

アクティビティ モデルのシミュレート時に `activity-mode` コマンドを使用しなかった場合は、ADSim アドインで生成された状態機械によって追跡が行われます。これは混乱を招くことがあるので、通常は推奨できません。状態機械とアクティビティの両方を同時に追跡することはできません。

参照

シーケンスのトレース レベルと UML モデルの追跡レベルの変更

実行追跡レベル

シーケンス図トレース

このトレース方法では、シーケンス図内のそれぞれの遷移を追跡し、遷移時に行われるシグナル送信を観察できます。

シーケンス図トレースの有効化

シーケンス図内の追跡を有効にするには、以下のいずれかのタスクを実行します。

- ［ベリファイ］メニューから［シーケンス図でトレース］をクリックする。
- ［Model Verifier］ツール バーの［シーケンス図でトレース］ボタンをクリックする。

［シーケンス図でトレース］ボタンを押すと、シーケンス図トレースが有効になります。

シーケンス図トレース ウィンドウは、通常のウィンドウまたはドッキング ウィンドウとして開くことができます。この動作は、［シーケンス図トレースウィンドウをドッキング］オプションで設定できます。

参照

シーケンス図

シーケンス図トレースの無効化

- シーケンス図トレースを無効にするには、再度［シーケンス図でトレース］ボタンをクリックします。

UML ソースへのナビゲート

シーケンス図でシンボルをダブルクリックすると、そのシンボルのソースにナビゲートします。

トレースの中断

トレースは、新規に作成したシーケンス図をデフォルトの場所から移動したりするなど、実行中にモデルを変更すると中断されます。ただし、新しいシーケンス図を開いてそのダイアグラム内でトレースを続行できます。

シーケンス図のトレース レベルの変更

シーケンス図のトレース実行時に表示される情報の精度は、任意に変更できます。以下のセクションを参照してください。

ヒント

UML モデルの追跡とシーケンス図のトレースを同時に行う場合は、Tau IDE の一方にシーケンス図ウィンドウをドッキングすると便利です。このためには、シーケンス図ウィンドウのタイトルバーを右クリックして［Docked to］を選択します。

ヒント

実サイズのアプリケーションでは、トレース結果のシーケンス図には大量のライフラインが含まれることがあります。アクティビティ シミュレーション セッションのシーケンス図トレースによって、大量のライフラインが含まれることもあります。このような場合、概要を見るためにダイアグラムの表示をズームアウトするとよいでしょう。たとえば、ダイアグラムのコンテキスト メニューから［Zoom］-> ［Zoom to fit］を選択します。この概要表示モードでは、シンボルとラインのテキストは小さくて読むことができませんが、ダイアグラム内のシンボルまたはラインにマウスカーソルを合わせると、ツールチップにテキストが表示されます。

参照

- 402 ページの「シーケンスのトレース レベルと UML モデルの追跡レベルの変更」
- 419 ページの「シーケンス図トレース レベル」

アプリケーションの実行

実行を開始するために使用できる複数のコマンドがあります。ニーズに合ったコマンドを選択してください。

実行の開始

以下のいずれかの方法を使用して、実行を開始できます。

- ［ベリファイ］メニューから、使用するコマンドをクリックする。
- ［Model Verifier］ツール バーからコマンドを選択する。
- Model Verifier コンソールでコマンドを入力する。

参照

ユーザー インターフェイス コマンドのリスト

Model Verifier コンソール

実行の停止

たとえば、Go コマンドを使用して実行を開始した場合は、以下のいずれかの方法を使用しないと実行を停止できません。

- ［ベリファイ］メニューから［実行の中断］をクリックする。
- ［Model Verifier］ツール バーから［実行の中断］を選択する。
- Model Verifier コンソールで、「`break`」と入力する。

注記

アプリケーションが一定時間使用されないか、動的エラーが発生したときは、それぞれのアクティブ ブレークポイントでも実行が停止されます。

実行の再開

以下のいずれかの方法を使用して、同じシミュレーションを何度でも再開できます。

- ［ベリファイ］メニューから［再実行］をクリックする。
- ［Model Verifier］ツール バーから［再実行］を選択する。

シミュレーションは常に最初から、つまりシミュレーション タイム ゼロから再開されます。シミュレーションを特定の位置から再開したい場合は、リプレイ モードで実行する必要があります。

参照

リプレイ モード

ランタイム プロンプト

インフォーマル分岐、ANY 分岐、非実装演算子などの一部の構成要素では、実行の続行時にユーザー入力が必要になります。

分岐プロンプト

実行中にインフォーマル分岐または ANY 分岐に到達すると、ダイアログが開いて可能な選択肢が表示されます。オプションを選択して［OK］をクリックすると実行が再開されます。

操作プロンプト

パッシブ クラスまたはアクティブ クラスの操作が宣言されて実装がないときは、その実行によってダイアログが開きます。このダイアログには［ウォッチ］ウィンドウに似たツリー構造が含まれています。

ツリー構造には、演算子呼び出しのスタック フレームが表示されます。これは、操作の名前、それぞれの引数の子ノード、**result** という子ノードが含まれるルート ノードです。操作がパッシブ クラスに属していて、静的でない場合は、操作が適用されるオブジェクトである **itself** というノードも含まれます。

- ツリーのどの要素も変更できます。それぞれの割り当ては現在のシナリオに登録されています。
- 他のコマンドを使用してモデルの状態を変更することもできます。たとえば、操作がアクティブ クラスに属している場合は、シグナルの送信が必要になる場合があります。
- 続行するには、［OK］ボタンをクリックします。

ブレークポイントの挿入と削除

ブレークポイントを設定して、目的の位置で実行を停止できます。ブレークポイントは、モデル内にいつでも設定できます。モデル ベリファイヤ（Model Verifier）アプリケーションが起動している必要はありません。

ブレークポイントの挿入

ブレークポイントを挿入するには、以下の手順を行います。

1. ブレークポイントの挿入先としたいシンボルが含まれている状態機械図を開きます。
2. シンボルまたはそのテキスト内で右クリックして、ショートカット メニューから［ブレークポイントの挿入 / 削除］をクリックします。ブレークポイントが挿入されたことを示す赤いドットがシンボル フレームまたはシンボル フレーム内のステートメントの隣に追加されます。

［Model Verifier］ツール バーから［ベリファイ］メニューのコマンドを使用してブレークポイントを挿入することもできます。

挿入されたブレークポイントはブレークポイント ウィンドウに表示されます。

ブレークポイントの正確な位置付け

以下の例は、ブレークポイントを設定して目的の位置を正確に特定する方法を示しています。

- アクション シンボルに複数のステートメントが含まれている場合、ブレークポイントを特定のステートメントに設定するには、カーソルをステートメントの終りのセミコロンの前に合わせます。この方法はテキスト図内のブレークポイントの設定にも当てはまります。
- `for` 文上にブレークポイントを挿入するには、キーワード `for` 内の任意の位置にカーソルを合わせます。
- `while-do` 文上にブレークポイントを挿入するには、キーワード `do` 内の任意の位置にカーソルを合わせます。
- nextstate アクション上にブレークポイントを挿入するには、ステート シンボルのフロー ライン上に挿入します。

ブレークポイントの削除

ブレークポイントを削除するには、以下の手順を行います。

1. ブレークポイントが挿入されているシンボルが含まれる状態機械図を開きます。
2. シンボルを右クリックして、ショートカット メニューから［ブレークポイントの挿入 / 削除］をクリックします。

ブレークポイント ウィンドウを使用してブレークポイントを削除することもできます。

ブレークポイントのリストの表示

ブレークポイントのリストを表示するには、以下の手順を行います。

挿入されたすべてのブレークポイントのリストを Breakpoint ウィンドウに表示できます。このリストを使用して、モデル内の既存のブレークポイントを探すことができます。

- ［編集］メニューから［ブレークポイント］をクリックします。

Breakpoints ウィンドウのチェック ボックスを選択または選択解除して、ブレークポイントを有効にしたり無効にしたりできます。ブレークポイントを無効にすることは、ブレークポイントを削除することとは違います。削除されたブレークポイントは Breakpoints ウィンドウに表示されません。

ブレークポイント エントリをダブルクリックすると、そのブレークポイントが設定されているダイアグラムが開きます。

参照

Model Verifier コンソール

メッセージの送信

環境の振る舞いをシミュレートするには、環境からモデルに手動でメッセージを送信します。送信するメッセージはメッセージ リストに挿入する必要があります。メッセージは送信側パスや受信側パスなどのパラメータ付きのシグナルです。

内部メッセージをリストに追加することもできます。内部メッセージは、モデル内にその送信側と受信側の両方があるメッセージです。これにより、不完全なシステムをベリファイできます。まだ実装されていないメッセージを手動で送信できます。この機能を使用して、モデルの堅牢性をテストできます。たとえば、モデルで予期しないシグナルを処理できるかどうかをテストできます。

シミュレーション セッションの実行中に随時メッセージを送信できます。

メッセージの作成

メッセージを作成するには、以下の手順を行います。

1. ［表示］メニューで、［ Model Verifier ウィンドウ］にカーソルを合わせ、［メッセージ］をクリックします。出力ウィンドウにメッセージ リストが表示されます。
2. リストの最初の行を右クリックして、表示されるショートカット メニューから［挿入］をクリックします。
3. それぞれのカラムをクリックして、使用可能なシグナル、送信側などのリストを表示します。必要な選択を行います。

個別のボックス内のリストの要素はそのフル パスと一緒に表示されるので、正確に識別できます。パスの例として a_block.a_process[an_integer] または a_package::a_signal などがあります。

［パラメータ］フィールドの構文は、一連の式の UML 構文です。2 つの定義済み値として、パラメータのリストと [[-]] 記号が提供されます。[[-]] を選択すると、モデル ベリファイヤ（Model Verifier）によって「デフォルト NULL」値が選択されます。たとえば、整数のデフォルト NULL 値は 0 になり、文字列の場合は空の文字列（「 」）になります。

ボックス内の要素を選択する代わりにテキストを入力することもできます。その場合は、不フル パスを指定できます。つまり、パスの先頭部分を省略できます。ただし、終りの部分は省略できません。インスタンス番号の場合も同じです。

注記

シグナルへの参照を入力する際、その後に続けて括弧で囲んだシグナル パラメータ型のカンマ区切りリストを入力してください。これは、パラメータのみが異なる同じ名前のシグナルが多数存在する可能性があるためです。参照しているシグナルがパラメータを持たない場合は、シグナルの後に続けて空の括弧を入力してください。

シグナルの送信

シグナルを送信するには、以下の手順を行います。

1. メッセージ リストで、送信するシグナルを右クリックします。
2. ［送信］をクリックして、シグナルを送信します。

複雑なシグナル パラメータ値

［パラメータ］フィールドに挿入が必要な値が、非常に複雑な値になる場合があります。以下の手順に従って、パラメータ値を簡単に定義できます。

1. ［パラメータ］フィールドで [[-]] 式を選択します。
2. メッセージを送信します。
3. 送信先インスタンスの待ち行列で［ウォッチ］ウィンドウを開きます。
4. ［ウォッチ］ツリーを開いて、送信したメッセージを確認します。

5. F2 キーを押して、パラメータ値を設定します。
6. パラメータを表すツリー ノードを選択します。
7. ［ディープ コピー］コマンドを適用します。
8. ［パラメータ］フィールドで結果を貼り付けます。

パートとして宣言されている複雑なパラメータ（クラス、文字列など）は値参照され、パートとして宣言されていないパラメータはポインタ参照されます。パートとして宣言されていないパラメータに新しい値を割り当てるには、中かっこ（{}）で囲んで指定する必要があります。クラスは受け入れられるようにタイプ キャストする必要があります。パッシブ クラスの要素はフォーマルに要素名に割り当てる必要があります。

例 **115:** シグナル パラメータ

```
class Class2 {
    Integer p1;
    Real p2;
}
interface i {
    signal mysig1( myString par1);
}
syntype myString = String<Class2>;
```

この例は、シグナル `mySig1` を送信するため、Messages ウィンドウのパラメータ フィールドにタイプ `myString` の値を入力する例を示しています。

```
{Class2 (. p1=8, p2=5.1 .), Class2 (. p1=4, p2=5.6 .)}
```

この例は、タイプ Class2 の 2 つの要素がある文字列を示しています。

参照

他の状況での値の入力方法、SDL や ASN.1 などの言語でサポートされる他の言語の使用法については、第 7 章「モデル ベリファイヤ（Model Verifier）リファレンス」の 425 ページ、「パッシブ クラス値」

［ウォッチ］ウィンドウ

シミュレーション セッションの実行中に、［インスタンス］ビューで使用可能になっているモデル オブジェクトを個別のウィンドウからウォッチできます。アクティブ クラス インスタンス、アクティブ クラス インスタンス セット、アクティブ クラス インスタンスの属性、属性要素に加えて、メッセージ待ち行列、タイマー インスタンス、呼び出しスタックなどのランタイム オブジェクトをウォッチできます。

それぞれの値は［ウォッチ］ウィンドウに表示されます。必要な数のオブジェクトをウォッチできます。また、複数の［ウォッチ］ウィンドウを開くこともできます。

ビュー ウィンドウを閉じた場合でも、オブジェクトは［ウォッチ］ウィンドウに格納されます。

インスタンス オブジェクトのウォッチ

インスタンス オブジェクトをウォッチするには、以下の手順を行います。

1. ［インスタンス］ビューでオブジェクトを右クリックします。
2. ［ウォッチ］をクリックします。

［ウォッチ］ウィンドウが開き、オブジェクトとその値が表示されます。［ウォッチ］ウィンドウをすでに開いている場合は、選択されている最新のウォッチにオブジェクトが追加されます。

［ウォッチ］ウィンドウをすでに開いている場合は、オブジェクトを［インスタンス］ビューからドラッグ アンド ドロップすることもできます。［インスタンス］ビュー内のどのオブジェクトでもウォッチできます。

［ウォッチ］ウィンドウからのインスタンス オブジェクトの削除

インスタンス オブジェクトを削除するには、以下の手順を行います。

- ［ウォッチ］ウィンドウで、削除するオブジェクトを右クリックして、［ウォッチ解除］をクリックします。選択されたツリー ルートがビューから削除されます。

空の［ウォッチ］ウィンドウを開く

新しい空の［ウォッチ］ウィンドウを開くには、以下の手順を行います。

- ［ベリファイ］メニューから［新しいウォッチを開く］をクリックします。

コンソール ウィンドウでの表示と編集

デバッグ セッション中に配列または構造型のデータ型などの複雑な型を表示／編集するためには、Model Verifier コンソールが便利です。コンソールコマンドを使用して、全体の属性、要素の範囲、または 1 つの要素を出力／編集できます。

例 **116:** 配列要素の出力 ——————————————————————————

`myData` という属性を考えます。これは、構造体へのポインタで、要素の 1 つとして配列を持ちます。

```
Examine-Variable (MyClass:1) myData -> bits  22
```

上記の例は、配列内の 22 番目の要素を出力します。

——

例 **117:** 要素範囲の出力 ——————————————————————————

`myData` という属性を考えます。これは、構造体へのポインタで、要素の 1 つとして配列を持ちます。

```
Examine-Variable (MyClass:1) myData -> bits  22 25
```

上記の例は配列内の 22 から 25 の範囲の要素を出力します。

例 **118:** 要素の初期化 ———————————————————————————

`myData` という属性を考えます。これは、構造体へのポインタで、要素の 1 つとして配列を持ちます。

```
Assign-Value (MyClass:1) myData -> bits 22 1
```

上記の例は 22 番目の要素を 1 に設定します。

例 **119:** 配列要素の出力 ———————————————————————————

`myData` という属性を考えます。これは、構造体へのポインタで、要素の 1 つとして配列を持ちます。

```
Examine-Variable (MyClass:1) myData
```

上記の例は配列全体を出力します。

ポインタのアドレスと値の出力

次のコマンドは、Model Verifier モードを設定し、ポインタのアドレスのみ出力します。これが、デフォルト モードです。

```
REF-Address-Notation
```

次のコマンドは、Model Verifier モードをポインタに従うように変更し、参照先の要素の値を出力します。

```
REF-Value-Notation
```

要素値の変更

シミュレーション セッションの実行中に、手動で要素の値を変更できます。この機能を使用すると、要素のさまざまな値を短時間でテストできます。

要素値を変更するには、以下の手順を行います。

1. ［ウォッチ］ウィンドウで要素をクリックして、F2 キーを押します。
2. 表示されたテキスト フィールドに新しい値を入力します。Enter キーを押します。

不正な値を入力すると、要素によって前の値が保持されます。

要素値の表示

たとえば、シミュレーション セッション実行中のある時点で、属性、シグナル、状態などの値の確認が必要になることがあります。値は出力ウィンドウの［Model Verifier］タブに表示されます。つまり、デバッグの結果をログ記録する場合は、値が保存されます。

1. ［インスタンス］ビューで、表示する要素を選択します。
2. 選択した要素を右クリックして、［表示］をクリックします。それぞれの値は出力ウィンドウに表示されます。

参照

結果のログ記録

要素値のコピーと貼り付け

［インスタンス］ビューまたは［ウォッチ］ウィンドウで要素の値のコピーと貼り付けを実行できます。使用できるコマンドは、［コピー］、［ディープ コピー］、［貼り付け］です。

注記

編集モードでの定義済みテキスト操作以外の要素変更コマンド（［切り取り］など）は提供されません。

コピー

［コピー］コマンドはアクティブでないオブジェクトに適用できます。操作またはメッセージには適用できません。このコマンドはオブジェクトの値を UML 式として表すテキストのコピーを作成します。複数のオブジェクトを同時にコピーする場合、それぞれの値をカンマで区切ります。

コピーされたテキストを貼り付けて、セッションの実行中に特定の値を別の要素に割り当てることができます。

オブジェクトに他のオブジェクトの参照が含まれている場合、コピーされるのは他のオブジェクトの物理アドレスです。物理アドレスをコピーする場合は、そのアドレスをモデルに貼り付けないことを推奨します。物理アドレスはその後のセッションで無効になります。

［コピー］コマンドを使用してコピーするには

1. ［インスタンス］ビューまたは［ウォッチ］ウィンドウでオブジェクトをクリックします。
2. ［編集］メニューから［コピー］をクリックします。
3. 値を割り当てるオブジェクトを選択します。
4. **F2** キーを押して、編集モードにします。
5. フィールドを右クリックして、ショートカット メニューから［貼り付け］をクリックします。

ディープ コピー

このコマンドは［コピー］コマンドと似ています。唯一の違いは、オブジェクトに別のオブジェクトの参照が含まれているときに、物理アドレスではなく他のオブジェクトの値がコピーされることです。

コピーしたテキストを貼り付けて、たとえば、特定の値を別のオブジェクトに割り当てることができます。また、［メッセージ］ダイアログの［パラメータ］フィールドに貼り付けたり、モデルに貼り付けて名前付き定数を定義したり、シーケンス図のシグナル パラメータの値を定義したりできます。

［貼り付け］コマンドを適用できるのは、コピーされたオブジェクトが UML 静的定数属性となっている場合です。つまり、定数を定義する UML 式がオブジェクトに割り当てられている必要があります。

［ディープ コピー］コマンドを使用してコピーするには

1. ［インスタンス］ビューまたは［ウォッチ］ウィンドウでオブジェクトを右クリックします。
2. 表示されるショートカット メニューから、［ディープ コピー］をクリックします。
3. 値を割り当てるオブジェクトを選択します。
4. **F2** キーを押して、編集モードにします。
5. フィールドを右クリックして、ショートカット メニューから［貼り付け］をクリックします。

参照

UML 式

インスタンスの作成または削除

インスタンスの作成機能または削除機能によって、モデルをベリファイできる可能性が増大します。インスタンスの作成と削除は［インスタンス］ウィンドウまたは［ウォッチ］ウィンドウで実行できます。

新しいインスタンスを作成するには

1. 新しいインスタンスを作成するためのアクティブ クラス インスタンスを右クリックします。
2. ［新規］をクリックします。

インスタンスを削除するには

1. 削除するオブジェクトを右クリックします。
2. ［削除］をクリックします。このコマンドは、以下の動作を実行します。
   - 選択されたオブジェクトがアクティブ クラスのインスタンス場合は、そのオブジェクトを停止する。
   - 選択されたオブジェクトがタイマー インスタンスの場合は、そのオブジェクトをリセットする。
   - 選択されたオブジェクトがポインタ オブジェクトの下にある場合は、そのオブジェクトを削除する。
   - 選択されたオブジェクトがメッセージ待ち行列のメッセージの場合は、そのオブジェクトを削除する。
   - 選択されたオブジェクトがパッシブ リスト オブジェクトの場合は、そのオブジェクトを削除する。

オブジェクトの検索

［インスタンス］ビューと［ウォッチ］ウィンドウ内のオブジェクトの定義は、そのオブジェクトをダブルクリックして簡単に探すことができます。定義はオブジェクトに最適のダイアグラム形式で表示されます。

結果のログ記録

実行結果はログ記録できます。テキスト トレースとシーケンス図の追跡をログ記録できます。

テキスト トレースのログ記録

テキスト トレースは出力ウィンドウの［Model Verifier］タブに表示されます。結果をテキスト ファイルに保存するには、以下のタスクを実行します。

1. 出力ウィンドウで、［Model Verifier］タブをクリックします。
2. テキスト部分の任意の場所を右クリックし、表示されたショートカット メニューから［すべて選択］をクリックします。
3. テキスト部分をもう一度右クリックして、ショートカット メニューから［名前をつけて保存］をクリックします。
4. ログ ファイルの名前を決定して、ファイルを保存します。

シーケンス図トレースのログ記録

結果を保存するには、以下のタスクを実行します。

1. ［シーケンス図でトレース］が有効になっていることを確認します。
2. 実行を開始します。

3. ［モデル ビュー］で使用可能になっているパッケージ DebugTrace を右クリックして、［新しいファイルで保存］をクリックします。

結果をテキスト ファイルに保存するには、以下のタスクを実行します。

1. Model Verifier コンソールで、「`start-batch 2 <filename.txt>`」と入力します。
2. 実行を開始します。
3. 終了したら、「`stop-msc-log`」と入力します。

ファイルはモデルが保存されている同じフォルダに保存されます。

カバレッジ統計ビュー

［カバレッジ統計］ビューは、出力ウィンドウに個別のタブとして表示されます。使用可能なタブは、［カバレッジ統計］、［コード カバレッジ］、［遷移カバレッジ］です。

カラム ヘッダをクリックすると、カラムに従ってレポート タブのラインをソートできます。「プライマリ」カラムと「セカンダリ」カラムに従ってソートするには、最初にセカンダリ カラムでソートしてから、プライマリ カラムでソートします。

カバレッジ統計を表示するには

以下の手順に従って、カバレッジ統計を表示します。

1. ［ベリファイ］メニューから［カバレッジ統計の表示］をクリックします。出力ウィンドウに［カバレッジ統計］タブが表示されます。
2. ［カバレッジ統計］タブでラインを右クリックして、表示されるショートカット メニューから［カバレッジ詳細の表示］をクリックします。ラインの種類に応じて、［コード カバレッジ］タブまたは［遷移カバレッジ］タブが表示されます。

   ［カバレッジ統計］タブでラインをダブルクリックしても、カバレッジの詳細タブを表示できます。
3. カバレッジの詳細タブの 1 つを選択して、ラインを右クリックします。ショートカット メニューから［検索］をクリックします。対応するソースが状態機械図に表示されます。

カバレッジ情報はこのコマンドを実行するたびに更新されます。さらに、［コード カバレッジ ］ビューと［遷移カバレッジ］ビューは、定義済みの場合に更新されます。

［カバレッジ統計］タブ

［カバレッジ統計］タブには、実行中のモデルの操作がリストされます。「カバレッジ統計」レポートには以下のカラムがあります。

- **操作**：操作の名前を示します。
- **パス**：操作のフル パスを示します。

- **種類**：以下の値があります。
  - **statements**：ラインにステートメントのカバレッジが記述される場合。
  - **transitions**：ラインに遷移のカバレッジが記述される場合。
- **数**：操作のステートメントまたは遷移の数を示します。
- **カバー**：セッションの実行中に少なくとも 1 回実行されるカバレッジ内のステートメントまたは遷移の数を示します。
- **% カバー**：実行されたステートメントまたは遷移の割合を示します。
- **キューの最大長**：アクティブ クラスの構造体である操作の場合は、このカラムにセッションの実行中に到達したアクティブ クラスのインスタンス待ち行列の最大長が表示されます。

［コード カバレッジ］タブ

［コード カバレッジ］タブには、操作のステートメントに関する詳細な情報が表示されます。「コード カバレッジ」レポートには以下のカラムがあります。

- **操作**：操作の名前を示します。
- **パス**：操作のフル パスを示します。
- **ステートメント**：このカラムには、実行されるステートメントが表示されます。
- **カバレッジ**：このカラムには、セッション中にステートメントが実行された回数が表示されます。

［遷移カバレッジ］タブ

［遷移カバレッジ］タブには、操作の遷移に関する詳細な情報が表示されます。「遷移カバレッジ」レポートには以下のカラムがあります。

- **操作**：操作の名前を示します。
- **パス**：操作のフル パスを示します。
- **状態**：遷移の起点になる状態を示します。
- **シグナル**：遷移をトリガするシグナルのパスを示します。
- **カバレッジ**：このカラムには、セッション中に遷移が実行された回数が表示されます。

リプレイ モード

リプレイ モードでは、アプリケーションの初期状態から実行されたすべての実行ステップとユーザー コマンドを記録できます。実行ステップとユーザー コマンドはシナリオに保存されます。シナリオは保存してアプリケーション内の任意の状態まで再生できます。

たとえば、アプリケーションの複雑な初期化段階全体のシナリオを記録して、必要に応じて再生できます。または、デバッグ セッションを中断して後から続行する必要がある場合にシナリオを記録できます。

以下の実行ステップがシナリオに記録されます。

- 遷移
- タイマーのタイムアウト

シナリオに記録されるユーザー コマンドは、アプリケーションの状態を変更するコマンドです。以下のコマンドが記録されます。

- Signal Sending
- New
- Delete
- Re-arrange
- Assignment

シナリオを開く

シナリオを開くと、そのシナリオが現在のシナリオになります。現在のシナリオの前の内容は削除されます。

シナリオはシナリオ ウィンドウに表示できます。

シナリオ ファイルのロード

シナリオ ファイルをロードするには、以下の手順を行います。

1. ［ファイル］メニューから［シナリオを開く］をクリックします。
2. 使用するシナリオ ファイルを選択して、［開く］をクリックします。

下位互換性

前バージョンの Tau で保存されたシナリオ ファイルを開くと、下位互換性を確保するためにシグナル送信ステップと割り当てステップが変換されます。

- シグナル送信ステップでは、メッセージのパラメータに SDL 接頭辞が付きます。これは、パラメータが UML 式として処理されないことを意味します。
- 割り当てステップでは、割り当てられた値が SDL 接頭辞付きの UML 情報式として変換されます。これは、値が UML 式として処理されないことを意味します。

重要！

暗黙のインスタンスの名前は、リリースによって変更できます。このためには、旧バージョンで生成されたシナリオの編集または再生性を行う必要があります。
メッセージの受信者は、たとえば、
mm_om.AAA.@part_@implicit_process[1]
が、以下に変わります。
mm_om.AAA.@part_@implicit_process_0[1]

注記

旧バージョンの Tau で生成されたシナリオは、シナリオファイルが内部のモデル変更を反映して変換されていないため、使用できないことがあります。新バージョンの Tau で .u2 ファイルを開くと、モデルファイルの変更が自動的に変換されます。

シナリオの保存

シミュレーションを実行するたびに、実行ステップが現在のシナリオにリストされます。このシナリオは新規に実行を開始するたびに上書きされます。

ただし、現在のシナリオをファイルに保存して、後から再生できます。シナリオには拡張子 `.ttdscn` が付きます。現在のシナリオの有効なデフォルト名は `default.ttdscn` です。

シナリオの保存

シナリオを保存するには、以下の手順を行います。

1. シナリオに含めるステップを実行します。実行したステップがシナリオ ウィンドウに表示されます。
2. ［ファイル］メニューから［シナリオの保存］をクリックします。

シナリオの内容の表示

それぞれの実行ステップはシナリオ ウィンドウに表示されます。シナリオはそれぞれのステップを説明するテキスト要素のリストになっています。このリストはシナリオ ウィンドウで編集できません。

シナリオを実行しているときは、それぞれの実行ステップを追跡できます。実行されたステップはそれぞれのチェック ボックス内でチェック マークが付きます。

シナリオ ウィンドウを開くには

- ［表示］メニューで、［Model Verifier ウィンドウ］にカーソルを合わせ、［シナリオ］をクリックします。

参照

451 ページの「実行ステップ」
451 ページの「ユーザー コマンド」

シナリオの実行

シナリオはステップがアプリケーションに合致していないと実行できません。その場合は、出力ウィンドウの［Model Verifier］タブで通知されます。

シナリオは常に最初の実行ステップから開始されます。実行されると、それぞれのステップにチェック マークが付きます。

シナリオを実行するには

1. ［ベリファイ］メニューから［リプレイ モード］をクリックします（有効になっていない場合）。
2. 以下のように、実行するコマンドを選択します。
   - 現在のシナリオ内のすべてのステップを実行するには、［実行］コマンドを実行します。実行はブレークポイントのみによって中断されます。
   - 現在のシナリオ内の次のステップを実行するには、［次の遷移］コマンドを実行します。シナリオの次のステップとは遷移、タイムアウト、またはユーザー コマンドです。このコマンドは、現在のシナリオ内の位置が終りに到達していると無効になります。
   - 次のシナリオ ステップに記述される遷移に進むには、［ステップ イン］コマンドを実行します。
   - ［ステップ イン］コマンド、［ステップ オーバー］コマンド、［ステップ アウト］コマンドは、遷移がすでに実行中の場合に正しく動作します。それ以外の場合は無効です。

注記

上記のコマンドが正しく動作するのは、モデル ベリファイヤ（Model Verifier）がリプレイ モードに設定されている場合のみです。

モデル ベリファイヤ（**Model Verifier**）の設定

モデル ベリファイヤ（Model Verifier）の設定には、プロパティと実行中に行ったアクションがリストされます。設定には以下の情報が含まれます。

- ［メッセージ］ウィンドウで挿入したメッセージ。
- ブレークポイント ウィンドウで挿入したブレークポイントと、リストされているブレークポイント。Model Verifier コンソール コマンド「Breakpoint-*」を使用して設定されたブレーク条件は含まれません。
- ［親に従う］以外の値に設定したシーケンス図トレース レベル。
- ［親に従う］以外の値に設定した実行追跡レベル。
- それぞれの［ウォッチ］ウィンドウに追加したインスタンス。

注記

［ウォッチ］ウィンドウの図形としてのレイアウトは保存されません。

モデルを実行するたびに現在の設定が作成されます。現在の設定を保存して後で再利用できます。つまり、モデルを実行するたびにブレークポイントやメッセージを挿入する必要はありません。

設定全体またはその一部を保存できます。モデル ベリファイヤ（Model Verifier）を停止すると、現在の設定がターゲット ディレクトリに保存され、モデル ベリファイヤ（Model Verifier）実行形式ファイルに基づいて命名されます。次回にモデル ベリファイヤ（Model Verifier）を起動すると、起動時にそのファイルがロードされます。

参照

397 ページの「モデル ベリファイヤ（Model Verifier）の設定の保存」
393 ページの「リプレイ モード」

モデル ベリファイヤ（**Model Verifier**）の設定の保存

モデル ベリファイヤ（Model Verifier）を停止するたびに、現在のモデル ベリファイヤ（Model Verifier）の設定が、拡張子 `.ttdcfg` の付いたファイルに自動的に保存されます。設定ファイルの名前は、モデル ベリファイヤ（Model Verifier）実行形式ファイルの名前に基づいて付けられます。デフォルトで設定ファイルはターゲット ディレクトリに保存されます。

現在の設定に含まれるすべての情報は、デフォルトの設定ファイルに保存されます。ただし、設定情報のサブセットを保存する場合、またはモデル ベリファイヤ（Model Verifier）を停止せずに設定を保存する場合は、手動で設定ファイルを保存できます。

注記

［親に従う］以外のトレース レベルと追跡レベルのみが保存されます。

手動でモデル ベリファイヤ（Model Verifier）の設定を保存するには

1. 設定に含めるステップを実行します。
2. ［ファイル］メニューから［Model Verifier の設定の保存］をクリックします。
3. 表示されたダイアログで、設定に保存する情報のタイプを選択します。

   デフォルトで［すべて］チェック ボックスが選択されています。設定オプションのサブセットを保存するには、［すべて］チェック ボックスの選択を解除して必要なチェック ボックスを選択します。
4. ［ファイル名］フィールドに設定ファイルの名前を指定し、保存場所を選択します。ファイルのデフォルト名は `default.ttdcfg`、デフォルトの保存場所はターゲット ディレクトリです。

モデル ベリファイヤ（**Model Verifier**）の設定のロード

モデル ベリファイヤ（Model Verifier）を起動すると、設定ファイルが自動的にロードされます。このファイルには、実行用にプロジェクトを最後に使用したときの設定が含まれています。

ただし、手動で保存した設定ファイルをロードすることもできます。この操作には 2 つの方法があります。以下のいずれかを実行できます。

- 設定を開く。
- 設定を取得する。

次回のモデル ベリファイヤ（Model Verifier）の起動時に、手動で保存した設定ファイルを自動的にロードする場合は、モデル ベリファイヤ（Model Verifier）の停止後に、生成された設定ファイルを保存したファイルに置き換える必要があります。

モデル ベリファイヤ（**Model Verifier**）の設定を開く

設定を開くとは、開く設定ファイルに保存されているプロパティで現在の設定ファイル内の設定を置き換えるという意味です。

設定に保存されているオブジェクトのみによって、現在の設定内のオブジェクトが上書きされます。たとえば、［メッセージ］チェック ボックスの選択を解除した状態で設定ファイルが保存された場合、現在の設定の［メッセージ］ウィンドウ内のメッセージは上書きされません。

1. ［ファイル］メニューから［Model Verifier の設定を開く］をクリックします。
2. 表示されるダイアログで、開く `.ttdcfg` ファイルを選択して、［開く］をクリックします。

注記

モデルに対して行った変更を反映していない旧バージョンの Tau で生成された設定を開こうとすると、今後無効になる要素は無視されます。

注記

旧バージョンの Tau で生成された設定は、設定ファイルが内部のモデル変更を反映して変換されていないため、使用できないことがあります。新バージョンの Tau で `.u2` ファイルを開くと、モデルファイルの変更が自動的に変換されます。

モデル ベリファイヤ（**Model Verifier**）の設定の取得

設定を取得するとは、設定ファイル内のオブジェクトを現在の設定内のオブジェクトにマージするという意味です。ブレークポイントとメッセージについては、マージによって重複するオブジェクトは無視されます。トレース レベルと追跡レベルについては、マージによって新しいレベルが現在のレベルの後に適用されます。

たとえば、保存された設定にメッセージ A があって現在の設定にメッセージ B がある場合は、取得後に両方のメッセージが使用可能になります。シグナル B が上書きされる［モデル ベリファイヤの設定を開く］コマンドの場合と比較してください。

1. ［ファイル］メニューから［Model Verifier の設定の取得］をクリックします。
2. 表示されるダイアログで、取得する `.ttdcfg` ファイルを選択して、［開く］をクリックします。

コンソール コマンド

モデル ベリファイヤ（Model Verifier）の設定を Model Verifier コンソール モードで保存したりロードしたりするには、!U2::Debug save コマンドと !U2::Debug open コマンドを使用します。

UML 式

UML 構文の式を使用して、モデル ベリファイヤ（Model Verifier）のオブジェクトの値を割り当てたり、表したりします。そのために、モデル ベリファイヤ（Model Verifier）では、UML 式のモデルと Model Verifier オブジェクトのモデル間に変換アルゴリズムがインクルードされます。

通常の式で値を表せない場合は、非形式的な式（ターゲット式とも呼ばれる）が使用されます。非形式的な式の内容はモデル ベリファイヤ（Model Verifier）に固有なものです。したがって、ツールの他の部分で処理することはできません。

UML 定数式のサブセットがモデル ベリファイヤ（Model Verifier）によって入力としてサポートされます。

注記

以下のセクションでは、式を UML テキスト構文定義で指定されるその名前で記述しています。たとえば、メタモデルでの名前ではなく < integer name> が使用されています。メタモデルの場合は、< integer name> ではなく IntegerValue になります。

式への値のマッピング

オブジェクトの値を表すために使用される式の種類は、以下のようにオブジェクトのタイプによって異なります。

オブジェクト	式
整数型オブジェクトと Null 型オブジェクト	<integer name> 形式の <literal>
実数型オブジェクト	<real name> 形式の <literal>
時間オブジェクトと持続時間型オブジェクト	指数なしの <real name> 形式の <literal>
ブール型オブジェクト	「true」または「false」識別子
Charstring 型オブジェクト	<character string> 形式の <literal>（区切り文字として二重引用符を使用し、「¥」を使用して二重引用符を文字列にインクルード）
8 ビット型オブジェクトと OctetString 型オブジェクト	<hex string> 形式の <literal>
BitString 型オブジェクト	<bit string> 形式の <literal>
ビット型オブジェクト	「0」または「1」
文字型オブジェクト	• <character> 形式の <literal> • 前の表記を適用できない場合は、たとえば、`[[0x7f]]' ように 16 進数が含まれている <target expression>

オブジェクト	式
Pid 型オブジェクト	• 「NULL」リテラル • NULL Pid 型以外の値の場合：たとえば、`[[Match.p2[1] ]]` のように参照されるインスタンスのフル パスが含まれている <target expression> [1] の後のスペースに注意
列挙型オブジェクト	リテラル名を参照する <identifier>
パッシブ クラス オブジェクト	属性名が必須であり、プライベート属性の場合でもオプション以外のすべての属性を指定する必要がある <structure primary>
文字列型オブジェクトと配列型オブジェクト（複数の属性を含む）	<list expression>
選択型オブジェクト	アクティブ属性のみが指定される < structure primary>
他のオブジェクトの参照（「ポインタ」）	• 「NULL」リテラル • ［ディープ コピー］コマンドを使用する場合、またはシーケンス図トレース時：参照されているオブジェクトの値は直接使用 • それ以外の場合：たとえば、[[0x12efbc]] のように物理アドレスを表す 16 進数が含まれている <target expression>
「一般配列」オブジェクト	値は式にマッピングできません。

ポインタが［ディープ コピー］コマンドまたはシーケンス図トレースで処理されると、参照されたオブジェクトの反復処理によってオブジェクトのグラフが表示される場合があります。これを式として表すため、2 回以上参照されるオブジェクトには最初に参照されてその値が表されるときにラベルが付きます。次回に参照されるときは、ラベルのみが与えられます。

ラベルは非形式的な式として表されます。その値は、たとえば、[[_1]] のようにアンダスコア文字と整数で構成されます。

ラベルは、たとえば、「`[[_1]] = MyClass (. …ttribute values… .)`」のように、割り当て式を使用してオブジェクト値に関連付けられます。

そのようなラベルは、式内のみで有効になります。または、メッセージ定義の式のリストなしで使用されます。

例 **120:** オブジェクト ラベルを使用している式 ——————————————

クラスが以下のように指定されていると、

```
class MyClass {
  MyClass previous;
  Integer value;
  MyClass next;
}
```

式は以下のようになります。

```
( [[_1]] = MyClass (. previous NULL, value 1, next MyClass (.
previous [[_1]], value 2, next NULL .) .) )
```

注記

ツールではモデルの意味を推測できないため、データ構造によっては［ディープ コピー］処理が適さない場合があります。この処理は、リンクされたリストやツリーなどの高頻度パターンに適しています。ただし、「所有者」または「親」のようなリンクがあるサブオブジェクトを参照しないことが前提になります。

重要！

［ディープ コピー］アルゴリズムでは、ポインタが大きいオブジェクトの一部を参照していることを検出できません。このアルゴリズムでは、常にポインタがオブジェクト自体を参照するものと見なされます。

値への式のマッピング

モデル ベリファイヤ（Model Verifier）で UML 式を使用してオブジェクトの値をビルドしたり割り当てたりするときは、以下の式がサポートされます。

- 399 ページの「式への値のマッピング」で説明されているとおりにモデル ベリファイヤ（Model Verifier）によってビルドされて値が表される式。
- 次のようなカッコ付きの式：‘(‘<expression> ‘)’
- UML モデルから式がコピーされる場合の、静的な定数属性を参照する <identifier>。定数の値を指定する式が使用されている場合は、定数のアドレスを使用しません。
- ダッシュ符号「[[-]]」のみが含まれている非形式的な式。この符号はオブジェクトのタイプに従って値を作成するようツールに指示します。
- ラベルをオブジェクトに関連付けて（399 ページの「式への値のマッピング」で説明されているとおりに）オブジェクトのグラフを記述するときは、ラベルの最初の文字としてアンダスコアを使用する必要がありますが、アンダスコアの後には任意の文字列を使用できます。したがって、たとえば、「`[[_1]]`」ではなく「`[[_FirstElementOfTypeMyClass]]`」のように使用できます。

注記

式には操作呼び出しや属性オブジェクトの参照を使用できません。

エラー処理

UML の動的ルールに違反すると、シミュレーションの実行中に動的エラーが発生します。動的エラーは出力ウィンドウに表示されます。

動的エラーが検出された後は、現在のステートメントの終りまでシミュレーションの実行が再開されます。

Model Verifier コンソール

テキスト コマンド入力により、モデル ベリファイヤ（Model Verifier）を実行することもできます。使用できるコマンドには 2 つのタイプがあります。「!」で始まるコマンドは、ユーザー インターフェイスに同等な機能があります。

以下の手順に従って、コンソール ウィンドウを起動します。

1. ［表示］メニューで、［Model Verifier ウィンドウ］にカーソルを合わせ、［コンソール］をクリックします。コンソールが開きます。
2. 使用するコマンドと必須コマンド パラメータを入力します。
3. **Enter** キーを押します。

コマンドの出力は出力ウィンドウの［Model Verifier］タブに表示されます。

ヒント

最近使用されたコマンドは保存されています。コンソール フィールドの右の矢印をクリックすると、それらのコマンドを再実行できます。

注記

どのコマンドも取り消すことはできません。

コマンドによっては、パラメータ値を入力する必要があります。必須パラメータを入力しないと、以下のように入力を要求するメッセージが表示されます。

- Type '?' to get a list of possible parameters.（使用可能なパラメータのリストを表示するには、「?」と入力してください）。
- Type '-' to accept default values for the parameters.（パラメータのデフォルト値を確定するには、「-」と入力してください）。デフォルト値がない場合は、使用可能な値のリストが表示されます。

注記

「?」コマンドと「-」コマンドは、「!」で始まるコマンドには当てはまりません。

参照

428 ページの「コマンドの構文」
431 ページの「コンソール コマンド」

トレースと追跡のレベル

シーケンスのトレース レベルと UML モデルの追跡レベルの変更

［インスタンス ビュー］で変更するインスタンスを右クリックして、ショートカット メニューから［トレースと追跡のレベル］を選択します。

シーケンス図トレース レベルを変更するには、以下の手順を行います。

- 必要なトレース レベルをクリックして、ダイアログを閉じる。

実行追跡レベルを変更するには、以下の手順を行います。

- 必要な追跡レベルをクリックして、ダイアログを閉じる。

アクティビティ シミュレーション

モデル ベリファイヤ（Model Verifier）を使用してアクティビティ モデルをシミュレートできます。UML アクティビティの実行セマンティックは、アクティビティ ノードから出入りするトークンをベースとしています。アクティビティのセマンティックの詳細については、アクティビティ モデリングを参照してください。

ADSim アドインの有効化

アクティビティモデルをシミュレートするには、まず ADSim アドインを有効化する必要があります。以下の手順を実行します。

1. ［ツール］メニューから、［カスタマイズ］をクリックします。
2. ［アドイン］タブをクリックして、「ADSim」アドインをチェックします。
3. ［OK］をクリックします。

アクティビティシミュレーションの開始

アクティビティをシミュレートするには、以下の手順を実行します。

1. ［モデル ビュー］で、シミュレートするアクティビティを選択します。
2. 以下のいずれかの手順を実行します。
   - アクティビティを右クリックして、［シミュレーション モデルの作成］コマンドを選択します。
   - メイン メニュー バーの ADSim メニューから、［シミュレーション モデルの作成］コマンドを選択します。
   - アクティビティを右クリックして、［Activity Simulator］>［新しいアーティファクト］を選択します。アクティビティシミュレーション用のビルドあーティファクトが作成されます。このアーティファクトを右クリックして、［ビルド（Activity Simulator）］>［シミュレーションモデルの作成］を選択します。

ADSim アドインはアクティビティを分析し、アクティビティの振る舞いを実装するアクティブ クラスに変換します。また、ADSim アドインは、アクティビティを開始する " ラッパー " アクティブ クラスと、そのクラスをマニフェストしたモデル ベリファイヤ（Model Verifier）のビルドアーティファクトを作成します。元のアクティビティ モデルを変更しないですむように、この " ラッパー " アクティブクラスとビルドアーティファクトは、新しい最上位のパッケージに配置し、専用の .u2 ファイルに格納されます。

分析が完了したら、生成されたビルド アーティファクトからモデル ベリファイヤ（Model Verifier）の実行形式ファイルをビルドし、起動します。詳細については、モデル ベリファイヤ（Model Verifier）の起動を参照してください。ADSim から生成したビ

ルドアーティファクトで起動した場合、モデル ベリファイヤ（Model Verifier）は、デフォルトで Activity-Mode で動作します。アクティビティ モードと状態機械追跡モードの詳細については、状態機械とアクティビティ追跡モードを参照してください。

ヒント

ADSim メニューには［シミュレート］コマンドもあります。このコマンドはシミュレーション モデルを作成し、生成されたビルド アーティファクトを自動的に起動します。

アクティビティモデルをステップスルーするコマンド

アクティビティの実行は、従来のモデルベリファイヤでシミュレーションできます。ただし、実行コマンドの意味が一部異なります。アクティビティシミュレーションで使用する主要な 2 つのコマンドは以下のとおりです。

- **Next-Transition**: このコマンドはアクティビティ内の 1 つのノードを実行します。
- **Go**: このコマンドは、以下のイベントが発生するまで、シミュレーションを実行します。
  - 環境からの入力がなくなる。
  - ユーザーが実行を取りやめると判断する。

ブレークポイントにヒットすると、実行は、そのブレークポイントが設定されたアクティビティノードで停止します。

テキストトレース

アクティビティ シミュレーション セッションの最中に、トレースの目的で、テキストメッセージがモデルベリファイヤ コンソールに表示されます。メッセージは、アクション / トークンのレベルで、シミュレーションモデルで何が起こっているかを説明しています。たとえば、あるアクティビティノードを実行している場合や、モデル内にトークンが送信された場合にメッセージを受け取ることができます。

テキストトレースは常に利用可能です。

アクティビティ図トレース

アクティビティ図トレースを有効にすると、アクティビティ図内でアクティビティノードシンボルが選択されます。ただし、ステートチャートにおける同様な機能とは振る舞いが異なりますので注意が必要です。アクティビティ図については、ダイアグラム中で選択されるノードは、最も最近実行されたアクティビティノードです。緑色の三角形の実行マーカは、実行されたアクティビティノード上に表示されます。

アクティビティ図トレースの有効化 / 無効化は、［ベリファイ］メニューの［次のステートメントを表示］コマンドで制御できます。デフォルトでは、有効化されています。

トレース カラーリング

アクティビティノードは、実行されるとそのシンボルに色が付いてゆきます。この色付けによって、アクティビティ実行のどの部分までが実行されたかを目視確認できます。

色情報は、「Trace data for <name> (<date>)」という名前のパッケージに格納されます（ここで、<name> はシミュレートされるアクティビティで、<date> はシミュレーションセッション開始時の日付時刻）。トレースデータを後で分析したい場合は、パッケージをファイルに保存できます。

ヒント

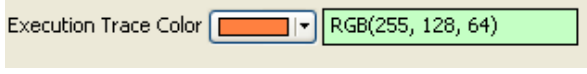
複数のシミュレーションセッションのトレースデータパッケージをそれぞれファイルに保存しておくと、後ですべてのファイルをモデルにロードして、シミュレーションセッションで通ったすべての経路をまとめて表示できます。アクティビティモデルから、実行されない部分を発見するのに役立ちます。

色データを削除するには、モデルからトレースデータパッケージを削除します。

トレース表示色の設定

トレース時に使用する色表示の設定は以下のとおりです：

1. ［モデルビュー］でモデルベリファイヤ ビルドアーティファクトを選択します。このアーティファクトは、作成したアクティビティシミュレーション モデルにあります。
2. プロパティエディタを開きます（**Alt** + **Enter**）。
3. フィルタリストで［**Activity Simulation Build Artifact**］を選択します。
4. ボタンをクリックするか、RGB 値をテキストフィールドに直接入力して［**Execution Trace Color**］プロパティを設定します。



シーケンス図トレース

アクティビティをシミュレートする場合に、モデルベリファイヤのシーケンス図トレースを使うことができます。アクティビティ シミュレーションセッション中にシーケンス図トレース用ツールバー ボタンをクリックすると、ダイアログが表示されて以下のオプションを提供示します。

- **No tracing**
  シーケンス図トレースをオフにするためにはこのオプションを選択します。

- **Node based tracing**
  このオプションを選択すると、シーケンス図トレースはアクティビティノードごとに 1 つのライフラインを表示します。メッセージラインを使用して、アクティビティノード間の制御のフローとデータトークンを表示します。
- **Partition based tracing**
  このオプションを選択すると、シーケンス図トレースはアクティビティ実装内の区画ごとに 1 つのラインフラインを表示します。区画を含まないアクティビティ実装については、デフォルトでは、ノードベースのトレースになります。ダイアログの設定によって、トレースにどの区画やアクティビティノードを含むかをカスタマイズできます。

シミュレーションセッション中の任意のタイミングでトレースダイアログを立ち上げて、シーケンス図トレースのオンオフを切り替えたり、ノードベーストレースにするか、区画ベーストレースにするかを切り替えられます。

ブレークポイント

ブレークポイントは特定のアクティビティ ノードで実行を停止するために使用できます。アクティビティ ノードの実行の直前で、すなわち、必要なすべてのトークンをアクティビティ ノードが使用できるようになったとき、ブレークポイントに到達します。複数のトークンがアクティビティ モデルのさまざまなパートに流れ込むと、実行がシーケンシャルではなく独立したトークンに基づくので、多少混乱を招くことになります。この状態は、マルチスレッド アプリケーションの従来のデバッグと似ています。

ブレークポイントは以下のノードに適用できます。

- Initial,
- Activity/Action,
- AcceptEvent,
- SendSignalAction,
- Decision/Merge,
- Fork/Join,
- ActivityFinal,
- FlowFinal.

サポートされるアクティビティノード

アクティビイティシミュレーションでは、以下のアクティビティノードがサポートされます。

- Initial
- Activity/action
- Decision/merge
- Fork/join
- Connector

- Accept event
- Send signal
- Activity final
- Flow final
- Object

データフローと制御の他に、パーティション、ストリーミングピン、入れ子アクティビティがサポートされます。

アクティビティへのシグナル送信

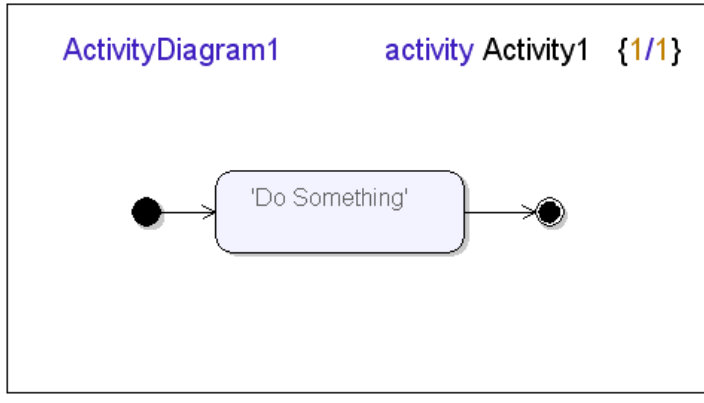
アクティビティにイベント受信ノードがある場合、通常のモデルベリファイヤ メッセージビューを使って、アクティビティの外部環境からシグナルを送信できます。メッセージビューは、［表示］>［モデルベリファイヤウィンドウ］>［メッセージ］コマンドで表示できます。

ただし、アクティビティに送信したメッセージの受信者が、シミュレーションモデル内の「**Communicator**」という名前のインスタンスでなければならないことに注意してください。このインスタンスはメインのアクティビティの一部です。

1 つの例

例 **121:** アクティビティ シミュレーションの実行 ————————————————

以下の実装によってアクティビティをシミュレートするものとします。



ADSim アドインを起動し［モデル ベリファイヤ］で Activity1 を選択します。次に ADSim メニューから［シミュレーション モデルの作成］コマンドを選択します。［メッセージ］タブに以下のメッセージが出力されます。

```
Running the ADSimGenerate
```

```
Preparing activity Activity1 for simulation...
Activity transformation completed!
Start simulation by launching the generated Model
Verifier build artifact Build_MV_Activity1.
```

ここで、生成されたビルド アーティファクトからモデル ベリファイヤ（Model Verifier）を起動し、シングル ステップでの実行のため、［次の遷移］コマンドを使用します（ユーザー インターフェイス コマンドのリストを参照）。

実行時には、Model Verifier コンソールに以下のメッセージが出力されます。

```
Initial_1 executed
    Sent control token to Do Something_2
Do Something_2 executed
    Sent control token to ActivityFinal_3
ActivityFinal_3 executed
```

シミュレーション時に Model Verifier コンソールまで追跡されたメッセージは以下についての情報を提供します。

1. アクティビティ ノードの実行。
2. 1 つのアクティビティ ノードから別のアクティビティ ノードへのコントロール トークンまたはデータ トークンの送信。データ トークンが送信された場合はその型も表示されます。
3. アクティビティ ノードを実行させないトークンの受信。現在アクティビティ ノードが受信したトークン数、および実行までに受信すべき合計トークン数が表示されます。

アクティビティシミュレーション クィックスタート

新しいインポートウィザードで利用できるサンプルに、アクティビティシミュレーションに適したモデルが含まれています。プロジェクト名は「**umlActivitySimulation**」です、［ファイル］>［新規］コマンドから［サンプル］タブをアクセスしてサンプルを選んでください。

Web サービス シミュレーション

モデルベリファイヤでシミュレーションするモデルから Web サービスを呼び出すことができます。この機能を使うと、自分で開発した Web サービスのシミュレーションと外部の Web サービスから提供される機能を UML モデルに組み込むことができます。

UML シミュレーションモデルから Web サービスを呼び出す仕組みは、`WSSim` アドインによって提供されます。

注記

Web サービスシミュレーション機能は、Windows オペレーティングシステムのみでサポートされます。

WSSim アドインの有効化

UML シミュレーションモデルからの Web サービス呼び出しを可能にするには、まず WSSim アドインを有効にする必要があります。以下の手順を実行します。

1. ［ツール］メニューから、［カスタマイズ］をクリックします。
2. ［アドイン］タブをクリックして、「WSSim」アドインをチェックします。
3. ［閉じる］をクリックします。

注記

WSSim アドインを有効にすると、自動的に WSDL アドインも有効になります。これは、呼び出される Web サービスの WSDL ファイルのインポートを可能にするためです。

WSDL/XSD モデリング用に新しいプロジェクトを作成するときに、［Web サービスシミュレーションをサポートする］オプションを有効にできます。このオプションが設定されていると、プロジェクトを開いたときに WSSim アドインは自動的にロードされます。

UML からの Web サービスの呼び出し

UML モデルからの Web サービス呼び出しを可能にするには、まず WSDL ファイルを Tau にインポートする必要があります。この操作を行うには WSDL/XSD インポートウィザードを使います。このウィザードを使う際は［Model］ノードを選択しておいてください。こうしておくと、作成される WSDL パッケージがモデル内の新しい最上位パッケージとして配置されます。WSDL ビューで作業している場合は、［Model］ノードは表示されないため、［Project］ノードを選択しておいてください。

WSDL/XSD インポートウィザードには 2 つのチェックボックスがあります。

- **UML Web サービスインターフェイスの生成**
  このオプションを選択すると、インポートされる Web サービスのインターフェイスとして機能する UML モデルが自動的に生成されます。このインターフェイスモデルの詳細については、WSDL パッケージからの UML Web サービスインターフェイスの生成を参照してください。
- **Web サービスコンシューマの生成**
  このオプションを選択すると、インポータはインポートされる Web サービスを消費するテンプレートモデルを作成します。詳細は Web サービスコンシューマの生成を参照してください。

WSDL パッケージからの **UML Web** サービスインターフェイスの生成

インポートされた WSDL パッケージのために UML インターフェイスモデルが作成される際には、以下の処理が行われます。

1. UML Web サービス インターフェイスパッケージが ”WSDL Packeges” の下に作成されます。このパッケージは、WSDL パッケージが記述している Web サービス用の UML API を構成し、ユーザーの UML モデルから Web サービスを呼び出すための定義を含んでいます。たとえば、1 つの Web サービスは、適当な操作を公開した 1 つのアクティブクラスとして現されます。この操作を通じて UML から Web サービスを呼び出せます。
2. Web サービスを呼び出すための "Glue（糊）" コードが生成されます。このコードは、`__wssim<N>` ディレクトリ（N は名前をユニークにするためにインクリメントされる数字）に配置されます。このディレクトリはプロジェクトディレクトリにあります。

WSDL を、上で説明したオプションを使わずにインポートした場合は、明示的なコマンドから UMLWeb サービスインターフェイスを作成できます。これを行うには、WSDL パッケージを右クリックして、コンテキストメニューから［UMLWeb サービスインターフェイスを生成 / 更新］コマンドを実行します。このコマンドは `WSSim` アドインが提供しています。

WSDL パッケージのコンテキストメニューには、［**Generate UML API**］コマンドも含まれています。このコマンドは上のステップの 1) のみを実行し、Glue コードの生成をスキップします。このコマンドは、UML モデルで Web サービスの使用を表現してシミュレーションしたいが、実際の Web サービスは呼び出さない場合に有用です。たとえば、Web サービスの実装が完了していないときには実際の呼び出しは当然できませんが、その Web サービスの呼び出しを含んでいるクライアントの UML モデルでシミュレーションを行いながら開発を進められます。

生成された **UML Web** サービスインターフェイスの使用法

これで、UML モデルから生成された UMLWeb サービスインターフェイスパッケージの定義を使い始める準備が整いました。以下に通常の動作と手順を示します。

1. モデル内のパッケージから生成した UMLWeb サービスインターフェイスパッケージへの <<import>> 依存を追加します。
2. Web サービスを現すアクティブクラスのインスタンスを作成します。このクラスは、生成された UML パッケージ内の “Services” サブパッケージに配置されます。
3. Web サービスの URL を設定します。これは、Web サービスアクティブクラスの “set_serviceUrl” 操作を呼び出すことによって行います。URL がインポートされた WSDL ファイルに指定されたもの（`<soap:address>` タグ内）と同じ場合は、引数として、生成された UML パッケージ内の “URLs” サブパッケージにある Charstring 属性を渡すことができます。この属性には WSDL ファイルからの URL がデフォルト値として格納されています。

4. Web サービス アクティブクラスの操作を呼び出します。この操作が実際の Web サービス呼び出しを実行します。この操作には接頭辞 “call_” が付いており、通常はいくつかのオーバーロードされた操作もあります。どの操作を使用するかは、実際に Web サービスに対してどの引数を指定したいか、発生するエラーをどのように取り扱いたいかを依存します。詳細については エラーハンドリング を参照してください。
5. ユーザーのクライアントモデルにモデルベリファイヤのビルドアーティファクトを追加します。このビルドアーティファクトについては Make-Template ファイル（`soap.tpm` ファイル）を使う必要があります。このファイルは、プロジェクトの `__wssim` ディレクトリにあります。

Web サービスコンシューマの生成

`WSSim` アドインは、生成された UMLAPI パッケージのコンテキストメニューに、［Web サービスコンシューマを生成］コマンドを提供します。このコマンドを使うと、インポートした Web サービスを消費するテンプレートモデルを生成できます。これによって上に述べた手順は自動化できます。このテンプレートモデルは、モデルベリファイヤビルドアーティファクトを含んでいます。このアーティファクトを起動して Web サービスを簡単にテストできます。

複数の Web サービスを使用する

UML モデルから複数の Web サービスを使用できます。いくつかのやり方がありますが、もっとも簡便なのは、すべての Web サービスの WSDL ファイルを最初から WSDL/XSD インポートウィザードを使用してインポートする方法です。この場合、ウィザードが生成した 1 つのテンプレート Web サービスコンシューマモデルは、すべてのインポートされた Web サービスを消費できます。

WSDL/XSD インポートウィザード 実行後にこの手順を行うことも可能です。複数の WSDL パッケージを［モデルビュー］で選択して、コンテキストメニュー［UMLWeb サービスインターフェイスの生成 / 更新］を選択すると、選択した WSDL パッケージごとに 1 つのインターフェイスパッケージが生成されますが、`__wssim` ディレクトリは 1 つだけ生成されて、そこに、すべての WSDL パッケージ用の "Glue" コードが含まれます。続いて、生成されたパッケージを選択して、［Web サービスコンシューマを生成］コマンドを起動すると、すべての選択した Web サービスを消費できる 1 つのモデルが生成されます。

既存のコンシューマモデルから新しい Web サービスを使う必要がある場合は、その Web サービスを別途インポートして、テンプレートコンシューマモデルを生成できます。その後、生成したコンシューマモデルから必要な部分のみを既存のモデルにコピーします。

データ型の対応付け

Web サービスの入出力を現す WSDL 要素は、WSSim アドインによって UML クラスに翻訳されます。このクラスは UML API パッケージの "Types" サブパッケージにあります。このクラスには通常の UML 型で型付けされた属性が含まれています。XSD 型と UML 型のマッピングは下の表にしたがって行われます。

XSD 型	UML 定義済み型
anyType	Charstring
anySimpleType	Charstring
duration	Charstring
dateTime	Integer
time	Integer
date	Charstring
gYearMonth	Integer
gYear	Integer
gMonthDay	Integer
gDay	Integer
gMonth	Integer
boolean	Boolean
base64Binary	Charstring
hexBinary	Charstring
float	Real
double	Real
anyURI	Charstring
QName	Charstring
NOTATION	Charstring
string	Charstring
normalizedString	Charstring
token	Charstring
language	Charstring
Name	Charstring
NMTOKEN	Charstring

XSD 型	UML 定義済み型
NCName	Charstring
NMTOKENS	Charstring
ID	Charstring
IDREF	Charstring
ENTITY	Charstring
IDREFS	Charstring
ENTITIES	Charstring
decimal	Integer
integer	Integer
nonPositiveInteger	Integer
long	Integer
nonNegativeInteger	Integer
negativeInteger	Integer
int	Integer
unsignedLong	Integer
positiveInteger	Integer
short	Integer
unsignedInt	Integer
byte	Integer
unsignedShort	Integer
unsignedByte	Integer

列挙を含む単純な XSD 型はリテラル付きの UML Datatype にマップされます。列挙を含まない単純な XSD 型はシンタイプにマップされます。

Date と Time

上の表で示したように XSD `date` 型は UML の `Charstring` で現されます。形式は、`YYYY-MM-DD` です。To facilitate working with the XSD `dateTime` 型を使用した作業を容易にするために、一部のユーティリティは `WSSType` UML ライブラリにあります。

SOAP ヘッダー

一部の Web サービスでは、SOAP ヘッダーが必要です。これはパラメータの他に追加のデータを渡す必要があるためです。典型的な使い方は、SOAP ヘッダーを複数の Web サービス呼び出しを通じて一定であるコンテキスト依存のデータを渡すために使う方法です。このようなコンテキスト依存のデータには、ユーザーアカウント情報（ユーザー ID とパスワード）や、1 つの対話の間に複数の Web サービス呼び出しにまたがって使用されるセッション識別子などがあります。

UML SOAP ヘッダーでは、データは Web サービスアクティブクラスの属性として現されます。SOAP ヘッダーデータをセットするには、Web サービス操作を呼び出す前に、これらの属性に値を割り当てる必要があります。この属性の可視性は "private" ですが、値の取得と設定のための public なアクセサ操作が提供されます。

非同期 Web サービス呼び出し

生成された UML Web サービスインターフェイスモデルは非同期的な Web サービス呼び出しをサポートします。Web サービスはシンプルな状態機械を持つアクティブクラスとして現されます。この状態機械は、Web サービスの非同期的な呼び出しを現すシグナルを取り扱います。シグナルの定義は、生成された UML パッケージの “Interfaces & Signals” サブパッケージにあります。Web サービス呼び出しに入力パラメータを渡すためのシグナルがあり、また、出力データを保持する応答シグナルがあります。

エラーハンドリング

生成された Web サービス ”call_” 操作のオーバーロード版の一部には、エラーハンドリング機能があります。Web サービス呼び出しの際に発生する可能性のあるエラーハンドリングをする必要がない場合は、接尾辞 “NR”（NR=No Return value、つまり戻り値なし）の付いた操作を呼び出してください。エラーハンドリングをする必要がある場合は、“call_” 操作のうち `SOAP_CallResult` クラスのインスタンスを戻すものを呼び出してください。以下にこのクラスの定義を示します。

```
class SOAP_CallResult
{
  public Boolean wasError;
  public Charstring errorDescription;
}
```

“wasError” 属性はエラーの場合は ”true” がセットされます。この場合、”errorDescription” 属性にはエラーメッセージがセットされます。

エラー報告

Web サービス起動エラーが検知された場合は、以下の形で報告されます。

- `WSSim` タブに表示される。エラーの発生元は、Web サービス呼び出しを現す UML 操作なので、モデルから複数の Web サービスを呼び出している場合でも、どの呼び出しが失敗したかを特定しやすくなります。
- ログファイル（`tlog_mv_soap.log`）に出力される。このファイルはモデルベリファイヤのターゲットアプリケーションのディレクトリに配置されます。

トラブルシューティング

ほとんどの Web サービスは Tau から呼び出し可能です。ただし、一部の特定のタイプの Web サービスが呼び出せない場合や、呼び出しにワークアラウンドが必要な場合があります。

Web サービス コールバック

Web サービスの中には、クライアントに特定のコールバックインターフェイスの実装を要求するものがあります。このような Web サービスは Tau から直接呼出せません。そういったコールバックインターフェイスの自動作成はサポートされません。

サポートされない型

サポートされない型のパラメータを持つ Web サービスは Tau から呼び出すことはできません。サポートされている型と定義済み UML 型へのマッピングについては、データ型の対応付け を参照してください。

サポートされないバインディング

現在サポートされている Web サービスバインディングは、SOAP 1.1 です。SOAP 1.2 や HTTP などのバインディングを使用している Web サービスは、Tau からは呼び出せません。

7

モデル ベリファイヤ（Model Verifier）リファレンス

このセクションは、モデル ベリファイヤ（Model Verifier）のリファレンス ガイドです。モデル ベリファイヤ（Model Verifier）の詳しい使用方法については 371 ページの「アプリケーションのベリファイ」を参照してください。

トレース レベル

このセクションでは、以下のトレース レベルと追跡レベルについて説明します。

- テキスト トレース レベル
- 実行追跡レベル
- シーケンス図トレース レベル

テキスト トレース レベル

0 ～ 6 のテキスト トレース レベルを使用できます。

トレース レベル **0**

テキスト トレースを無効にします。

トレース レベル **1**

このレベルは環境との間で送受信されるシグナルのみを表示します。

トレース レベル **2**

このレベルは遷移が起こる要因を表示します。この場合の情報の例を以下に示します。

```
*** TRANSITION START
*       PId    :p1:1
*       State  :Idle
*       Input  :Plong
*       Sender :p2:1
*       Now    : 0.0000
```

トレース レベル **3**

このレベルでは、シグナル送信、次のステート、停止などの重要なアクションが出力に追加されます。

```
*** TRANSITION START
*       PId    :p2:1
*       State  :Idle
*       Input  :Pling
*       Sender :p1:1
*       Now    : 0.0000
*   OUTPUT of Plong to p1:1
*** NEXTSTATE  Idle
```

トレース レベル **4**

このレベルはアクションや分岐などの追加タスクを表示します。

トレース レベル **5**

このレベルでは、ヌル遷移、破棄されたシグナルなどのアクションの結果が追加されます。

トレース レベル **6**

このレベルでは、シグナルやタイマーなどのパラメータ値も出力されます。

実行追跡レベル

状態機械図では、以下の追跡レベルを使用できます。

- **しない**
  実行追跡を無効にします。
- **コントロールに従う時**
  現在の実行ポイントが遷移内にあって実行が中断されると、実行される次のステートメントが追跡されます。
  現在の実行ポイントが遷移外にあると、最後の遷移で実行された最後のステートメントが追跡されます。
- **継続的**
  処理中に、実行されている各ステートメントが追跡されます。
- **親に従う**
  このレベルはモデル内の親要素に従って追跡レベルを設定します。

シーケンス図トレース レベル

シーケンス図トレースには、常に送信側インスタンスと受信側インスタンスが含まれます。

- **しない**
  このレベルはシーケンス図内のイベントのトレースを無効にします。
- **ソースまたはターゲット**
  このレベルでトレースが有効になるのは、受信側のシーケンス図トレース レベルが［しない］に設定されていない場合のみです。
- **常に**
  このレベルでは、受信側のシーケンス図トレース レベルに関係なくシーケンス図トレースが有効になります。
- **ブロック レベル**
  このレベルは特定のエージェントのトレースを有効にして、内包するエージェントを非表示にします。
- **親に従う**
  このレベルは送信側のモデル内の親要素と同じトレース レベルを設定します。

ユーザー インターフェイス コマンド

このセクションで説明されているモデル ベリファイヤ（Model Verifier）ユーザー インターフェイス コマンドは、ツールバーまたは［ベリファイ］メニューから実行できます。それらのコマンドをモデル ベリファイヤ（Model Verifier）コンソールから実行することもできます。

参照

コマンドの構文

コンソール コマンド

ユーザー インターフェイス コマンドのリスト

このセクションで説明されているモデル ベリファイヤ（Model Verifier）ユーザー インターフェイス コマンドは、ツールバーまたは［ベリファイ］メニューから実行できます。それらのコマンドをモデル ベリファイヤ（Model Verifier）コンソールで入力することもできます。

- **実行**

  処理の実行はユーザーが停止するか（たとえば、［デバッグの停止］コマンドを使用して）またはブレークポイントに到達するまで続行します。

- **次の遷移**

  このコマンドを使用して、状態機械図内の次の遷移までシステムを実行します。処理の実行は、次のステート シンボルまたはストップ シンボルで停止します。

  Simulation kind に Realtime オプションを使用し、次の遷移が、現在より 2 秒以上に先にスケジュールされたタイマーの場合、Next-Transition コマンドはシミュレーションを 1 秒間実行して停止します。

  遷移内で Next-Transition コマンドを実行すると、遷移の残りの部分が実行されます。

- **ステップ イン**

  このコマンドでは、遷移を通じてステートメントごとに進めます。（アクション シンボルには複数のステートメントが含まれている場合があります）。

  ［ステップ イン］コマンドでは操作にもステップ インします。

- **ステップ ローカル**

  このコマンドは、現在の状態機械内の 1 つのステートメントを進めます。すべての状態機械内の次のステートメントにステップする［ステップ オーバー］コマンドとは異なります。（このコマンドは、ツールバーからは使用できません。）

- **ステップ オーバー**

  このコマンドは［ステップ イン］に似ていますが、操作内のすべてのステップが同時に実行される点が異なります。

- **ステップ アウト**

  このコマンドを使用して、リターン シンボルも含む操作内の残りのステップを実行します。

- **ブレークポイントの挿入／削除**
  このコマンドを使用して、ブレークポイントを挿入または削除します。ブレークポイントを挿入したシンボルの位置に赤いドットが挿入されます。
- **実行の中断**
  このコマンドを使用して、実行をブレークします。実行は［Go］コマンドまたはいずれかのステップ コマンドが実行されたときに直前の一時停止位置から続行します。
- **再実行**
  このコマンドを使用して、アプリケーションをその初期ステートにリセットします。
- **デバッグの停止**
  このコマンドを使用して、デバッグ セッションを停止します。

コンソール

以下のセクションで説明されているコンソール コマンドを使用して、モデル ベリファイヤ（Model Verifier）をモデル ベリファイヤ（Model Verifier）コンソール ウィンドウから実行できます。

パッシブ タイプの値の入出力

値は UML 式で表されますが、UML 式を使用できない場合があります。その場合は、このセクションで説明されているとおりに値を表す必要があります。

以下の場合は、UML 式として値を使用できません。

- コマンド Set-Trace を使用した後でテキスト実行トレースに表示される値。
- 旧バージョンの Tau（2.2 以前）で作成されたシナリオとモデル ベリファイヤ（Model Verifier）構成ファイルに保存されている値。
- 次の構文を使用して［ウォッチ］ウィンドウで指定できる値：`[[SDL:value]]`。このような値を使用できるのは、たとえば、一般的な配列型のように UML 式が適用されない場合です。
- Set-Timer コマンドの引数としての値。

定義済みデータ型のリテラルの構文は、リテラルの SDL 定義に従います。ただし、適宜説明されているように、拡張される場合もあります。オプションとして、値に ASN.1 構文を使用することもできます。入力では両方の値表記を使用できますが、生成される出力のタイプを選択するコマンドがあります（SDL-Value-Notation と ASN1-Value-Notation）。

整数値と自然数値

整数の形式は SDL と ASN.1 標準に完全に準拠します。つまり、整数は一連の桁で構成され、先頭に「+」または「-」が付く場合があります。ただし、コマンド Define-Integer-Output-Mode では、出力上の整数の基数を定義できます（10 進、16 進、8 進）。これは、入力できる方法にも影響します。16 進値の先頭には「0x」が付き、8 進値の先頭には「0」（ゼロ）が付きます。

ブール値

ブール値は大文字または小文字を使用して、`true` または `false` として入力（出力）されます。入力では省略形を使用できます。ASN.1 モードでは、値が大文字で出力されます（`TRUE`、`FALSE`）。

実数値

実数値の SDL リテラル構文は、多くのプログラミング言語と同様に `E` 表記（指数表記）が含まれるよう拡張されています。

例 **122: SDL** 構文の実数値

実数 1.4527 * 10^{24} は、`1.4527E24` と記述できます。

実数 4.46 * 10^{-4} は、`4.46E-4` と記述できます。

ASN.1 の実数値の構文は、423 ページの例 124 のようになります。

例 **123: ASN.1** 構文の実数値

```
{mantissa 23456, base 10, exponent -3}
```

これは値 23.456 に相当します。

時間と持続時間値

時間と持続時間値の形式は、SDL 標準に従います。つまり実数値に E 表記（指数表記）を使用せず、拡張符号を 1 つ使用します。入力では、時間型値に絶対値を使用するか NOW の相対値を使用できます。時間値を符号なしで指定すると、絶対時間値が想定されます。また、プラスまたはマイナス符号を先頭に付けて指定すると、NOW の相対値が想定されます。

例 **124: SDL** 構文の時間値

`123.5` は 123.5 と解釈されます。

`+5.5` は NOW + 5.5 と解釈されます。

`-8.0` は NOW -8.0 と解釈されます。

文字値

文字値は、16 進表記 `0xFF` を使用するか、非印刷可能文字のリテラルを含めた SDL 標準に従って入力または出力されます。

Charstring 値

Charstring 値は SDL 標準に従って入出力できます。つまり、単一引用符（'）の後に多くの文字を使用し、さらに終りに単一引用符を付けます。Charstring 内では、引用符（'）を 2 回指定する必要があります。出力では、Charstring 内の非印刷可能文字が、単一引用符で囲まれた文字リテラルとして出力されます。

Charstring の ASN.1 構文は SDL 構文に似ています。ただし、区切り文字の単一引用符（'）は二重引用符（"）に置き換えられます。

例 **125: SDL** 構文の **Charstring** 値

`''`	空の文字列
`'abc'`	3 文字から成る文字列
`'a'NUL'c'`	2 番目の文字は NUL

Pid 値

有効な Pid 値であるヌルを除いて、Pid 値はアクティブ クラスの名前と 0 より大きい整数としてのインスタンス番号という 2 つの部分で構成されます。

作成されるアクティブ クラスの最初のインスタンスにはインスタンス番号 1 が割り当てられ、2 番目のインスタンスにはインスタンス番号 2 が割り当てられます（このように順次に割り当てられます）。構文の形式は Name:No です。この場合の Name はインスタンス名、No はインスタンス番号を表します。

入力では、コマンド パラメータが Pid 値の場合に、名前とインスタンス番号を代替的な方法として 1 つ以上のスペースで区切ることができます。同じ状況で、アクティブ クラスのインスタンスが 1 つのみの場合、インスタンス番号は不要です（入力するように指示されません）。ただし、コマンド パラメータがアクティブ クラスのインスタンスが想定されるユニットの場合は、名前とインスタンス番号の間にコロン（:）のみを使用できます。コロンは、クラス名の直後に使用する必要があります。そのような状況の例として、Set-Trace と Signal-Log のユニット パラメータがあります。

出力では、Pid 型値の後にプラス符号（+）が付く場合があります。これは、インスタンスが無効になっていることを示します。つまり、停止アクションが実行されています。プラス符号が使用されるのは、「†」文字を想起できるためです。

ビット

ビットには 2 つの値、0、1 が含まれます。この構文は入出力に使用されます。

BitString

BitString 値の場合は、以下の構文が使用されます。

```
'0110'B
```

単一引用符で囲まれた文字は 0 または 1 でなければなりません。入力では、OctetString の構文を使用することもできます。

8 ビット

8 ビット値に使用される構文は、2 つの HEX 数字で構成されます。例：

```
'00'h '46'h 'F2'h 'a1'h 'CC'h
```

文字 0 ～ 9、a ～ f、および A ～ F を使用できます。

OctetString

OctetString の場合は、以下の構文が使用されます。

```
'3A6F'H
```

文字列に含まれている 2 つの HEX 値の各ペアは、8 ビット値と見なされます。奇数の数の文字がある場合は、文字列の最後に 0 がさらに挿入されます。

ObjectIdentifier

ObjectIdentifier は文字列（自然数）と見なされます。つまり、SDL 値表記が使用される場合に、構文は以下のようになります。

```
(. 2, 3, 11 .)
```

入力では、リスト内のアイテムをカンマかスペースまたはその両方で区切る必要があります。ASN.1 値表記が選択されている場合、構文は以下のようになります。

```
{ 2 3 11 }
```

入力では、リスト内のアイテムをカンマかスペースまたはその両方で区切る必要があります。

列挙値

使用可能な値の列挙として SDL で定義されているデータ型は、SDL データ型定義のリテラルを使用して入出力できます。入力では、固有である限りリテラルを略記できます。

パッシブ クラス値

パッシブ クラス値は、2 つの文字「(.」とその後に続くコンポーネントのリストおよび終りの 2 つの文字「.)」として入出力されます。入力では、コンポーネントをカンマか複数のスペースまたはその両方（または復帰改行かタブ）で区切る必要があります。

例 **126:**

```
(. 23, true, 'a' .)
```

ASN.1 構文が使用される場合は、コンポーネント名も含まれます。

例 **127:**

```
{ Comp1 2, Comp2 TRUE, Comp3 'a' }
```

入力では、ASN.1 構文を使用して任意の順序でコンポーネントを使用できます。値が指定されていないコンポーネントは、データ型での意味とは無関係に値 0 が設定されます。

存在しないオプションのコンポーネントは出力されません。つまり、2 つのカンマの間に空の位置が存在します。

Choice 値

`choice` 値の構文の形式は、`ComponentName:ComponentValue` です。たとえば、`choice` に整数型のコンポーネント C1 が含まれている場合、

```
C1:11
```

は有効な `choice` 値です。

配列値

配列値は、「(:」とその後に続くコンポーネントのリストおよび終りの「:)」として入出力されます。入力では、コンポーネントをカンマか複数のスペースまたはその両方（または復帰改行かタブ）で区切る必要があります。最後のコンポーネントと終りの「:)」の間にもスペースを入れる必要があります。ASN.1 構文では、「{」と「}」が区切り文字として使用されます。

配列コンポーネントには、配列の実装用に C コード ジェネレータで選択された実装に応じて 2 つの構文が使用されます。入力に使用する構文の最も簡単な判定方法は、配列の属性を確認することです。

- 配列が単純な配列（つまりインデックスのタイプが単純で 1 つの範囲条件と制限された範囲が指定されている）の場合は、426 ページの例 128 のように配列値の SDL 構文を使用します。

例 **128:** 単純な配列値 ——————————————————————————

```
(: 1, 10, 23, 2, 11 :)
```

ASN.1 値表記が選択されている場合は、「(:」と「:)」を「{」と「}」に置き換えます。

- 配列が一般的な配列型の場合は、426 ページの例 129 のような構文を使用する必要があります。

例 **129:** 一般的な配列値 ——————————————————————————

```
(: (others:2), (10:3), (11:4) :)
```

これは、インデックス 10 の場合は値が 3、インデックス 11 の場合は値が 4、それ以外のインデックスの場合は値が 2 であることを表します。入力では、コンポーネントのカンマ、カッコ、コロンの代わりに 1 つ以上のスペース（または復帰改行かタブ）を使用できます。

単純な配列では、2 番目の構文も使用できます。単純な配列に最初の構文を使用した場合は、配列コンポーネントのすべての値を入力する必要はありません。「:)」または「}」を入力すると、残りのコンポーネントが「ヌル」値に設定されます（つまり、値に対応してコンピュータのメモリが 0 に設定されます）。

文字列値

文字列値は「(.」で開始して「.)」で終了します。文字列のコンポーネントはカンマで区切って列挙されます。

例 **130:**

```
(. 1, 3, 6, 37 .)
```

入力では、カンマの代わりに 1 つ以上のスペース（または復帰改行かタブ）を使用できます。ASN.1 構文では、「(.」と「.)」ではなく「{」と「}」が区切り文字として使用されます。

PowerSet 値

PowerSet 値は「[」で開始して「]」で終了します。PowerSet の要素はカンマで区切って列挙されます。例：

例 **131:**PowerSet 値

```
[ 1, 3, 6, 37 ]
```

入力では、カンマの代わりに 1 つ以上のスペース（または復帰改行かタブ）を使用できます。

bag 値

bag 値は「{」で開始して「}」で終了します。bag の要素はカンマで区切って列挙されます。

例 **132:** bag 値

```
{ 1, 3, 6, 37 }
```

入力では、カンマの代わりに 1 つ以上のスペース（または復帰改行かタブ）を使用できます。同じ値が 2 回以上使用される場合、その値は SDL 構文で複数回列挙されません。代わりに、使用回数が値の後に指定されます。

例 **133:** bag 内の複数同一値

```
{ 1, 3:4, 6:2, 37 }
```

これは以下と同じです。

```
{ 1, 3, 3, 3, 3, 6, 6, 37 }
```

ASN.1 構文では、最後の例のようにそれぞれのメンバーが明示的に指定されます。入力では、アイテムをカンマか 1 つ以上のスペースまたはその両方で区切ります。アイテムの数が多い場合でも少ない場合でも、同じ値を複数回記述することもできます。

Own 値と ORef 値およびパッシブ オブジェクト参照

ポインタ タイプの値（Own と ORef）には、2 つの構文を使用できます。HEX 値としてのポインタ アドレスまたはポインタによって参照されるデータ領域の値が使用されます。値ヌルはどちらの構文でも `Null` として出力されます。モニタ システムでは、用意されている 2 つのコマンドで、使用する構文を判定できます（REF-Address-Notation と REF-Value-Notation）。入力では、選択されている構文に関係なく両方の構文を使用できます。

例 **134:** アドレス表記

```
0x23A20020

HEX(23A20020)
```

参照

UML 構文の［メッセージ］ウィンドウでシグナルに複雑な値を入力する方法について第 6 章「アプリケーションのベリファイ」の 385 ページ、「複雑なシグナル パラメータ値」

コマンドの構文

概要

コマンドには 2 つのタイプがあり、それぞれ使用する構文がわずかに異なります。以下の説明は、主に「!」で開始されないコマンドに当てはまります。

コマンド名

コマンド名は、他のコマンド名との区別に必要な文字を使用して略記できます。コマンド名を明確に区切るには、特種文字のハイフン（-）を使用します。どの部分も、コマンド名が不明瞭にならない限り略記できます。

一例として、コマンド名 `List-Breakpoints` を考えます。このコマンド名は、「`List-B`」または「`L-B`」と入力できます。ただし、「List」という語で始まるコマンドは 1 つに限られないので、「`List`」と入力すると、それらを区別できない可能性があります。その場合は、以下のメッセージが表示されて、

```
Command was ambiguous, it might be an abbreviation of:
```

「List」で始まるすべてのコマンドのリストが提供されます。

大文字と小文字の区別はありません。

パラメータ

コマンド パラメータは、1 つ以上のスペース、復帰改行またはタブで区切ります。Pid 値が必須の場合は、名前とインスタンス番号の間にコロンを使用することもできます。コマンド名の後のパラメータ リストが不完全な場合は、出力ウィンドウで通知されます。必要なパラメータを入力するように要求されます。

パラメータは、パラメータ値が不明瞭にならない限り略記できます。大文字と小文字の区別はありません。

注記

キーワード **Sender**、**Parent**、**Self**、**Offspring** は、略記できません。

パラメータの一致

パラメータ名（略記される場合もある）が入力されて、非省略名に一致するときは、目的のエンティティ クラスの名前のみが考慮されます。パラメータとして名前が必要な場合は、アクティブ クラスを示す名前のみが完全名検索の一部になります。シグナル名とタイマー名は同じエンティティ クラスに属しています。

また、状態機械と属性の仮パラメータは同じエンティティ クラスに属しています。他のタイプの名前はすべてその固有のエンティティ クラスに属しています。

前のパラメータの知識を利用して、特定のパラメータ名の検索範囲を絞り込みます。

修飾子

名前が問題の要因になることがあります。たとえば、2 つの異なる「ブロック」内に同じ名前のアクティブ クラスが 2 つある場合、または「システム」とアクティブ クラスに同じ名前のシグナル定義が含まれている場合に、エラーが発生します。

最初のケースでは、クラス名が常に不明瞭になり、2 番目のケースでは「システム」で定義されたシグナルの名前が常に使用されます。このような問題を解決するには、SDL の場合と同じ構文で修飾子を使用できます。「system」S の「block」B で定義されている「process」P を指定するには、以下の表記を使用できます。

```
system S / block B P
<<system S / block B>> P
```

修飾子では「system」、「block」、「process」、「procedure」、「substructure」などの語を略記できませんが、blocks や processes などのすべての名前は通常のルールに従って略記できます。指定する必要があるのは、固有な修飾子パスの構成要素になる部分だけです。修飾子では、スラッシュ「 / 」の代わりに 1 つ以上のスペースを使用できます。修飾子の入力時は、出力時に修飾子の一部になるアングル ブラケットを省略できます。

インスタンス パス

インスタンス パスは、オブジェクトのマニフェスト関係から得られる修飾子パスの UML 表現です。インスタンス パスの例を以下に示します。

```
Match.p1
{Match.p1[2]}
```

パスは最上位レベルのインスタンスから始まるフル パスを指定する必要があります。オブジェクトを角かっこ [ ] で指定する場合は、上記の例のように中かっこ { } を使用する必要があります。

シグナル パラメータとタイマー パラメータ

シグナル インスタンスとタイマー インスタンスのパラメータは、コマンド パラメータの場合と同じように入力します。パラメータは、シグナル名またはタイマー名の直後に入力することもできます（かっこで囲むことも可能）。

例 **135:** シミュレータ コマンドに使用されるシグナル パラメータとタイマー パラメータ

```
Signal name :S
Parameter 1 (Integer) : 3
Parameter 2 (Boolean) :true
```

同じ指定を以下のように表すこともできます。

```
Signal name :S 3 true
```

または

```
Signal name :S (3 true)
```

シグナル パラメータを入力するときは、すべてのパラメータ値を入力する必要はありません。パラメータの位置に「-」を入力すると、パラメータに「ヌル」値が設定されます（つまり値に対応してコンピュータのメモリが 0 に設定されます）。「)」を入力すると、残りのパラメータに「ヌル」値が設定されます。

例 **136:** コマンドに使用されるシグナル パラメータ

```
Signal name :S -, true
```

この場合は、最初のパラメータに「ヌル」値が設定されます。

```
Signal name :S (3, -)
```

この場合は、2 番目のパラメータに「ヌル」値が設定されます。以下のように入力することもできます。

```
Signal name :S (3)
```

コマンドのエラー

コマンド名またはそのパラメータの 1 つでエラーが検出されると、エラー メッセージが出力されて、コマンドの実行が中断されます。コマンドは完全に実行されるか、まったく実行されません。

コンソール コマンド

?（対話形式のコンテキスト依存ヘルプ）

パラメータ：（なし）

「?」（疑問符）を入力すると、現在の位置の使用可能なすべての値、コマンドまたはタイプのリストが表示されます。リストが表示された後は、使用可能ないずれかの値を入力して続行します。

プロンプト レベルで「?」を入力すると、使用可能なすべてのコマンドのリストが表示されます。その後にコマンドを入力できます。

注記

このコマンドは「!」で始まるコマンドには適用されません。

!U2::Debug add_message

パラメータ：

```
<sender path> <signal path> <connector name>
<receiver path> < parameter string>
```

このコマンドを使用して、メッセージをメッセージ リストに追加します。

<sender path> は、送信側エージェントのパスを示し、<receiver path> は、受信側エージェントのパスを示します。インスタンス内のインスタンスはピリオド「.」で区切ります。

<signal path> は、シグナルのパスを示します。シグナル パスの例を以下に示します。

```
::<object>::<signal name>
```

!U2::Debug assign

パラメータ：

```
<object reference> <value>
```

このコマンドを使用して、新しい値をオブジェクトに割り当てることができます。オブジェクト参照を探すには、コマンド !U2::Debug path2object を使用します。

!U2::Debug break

これは 421 ページの「実行の中断」と同等なコマンドです。

!U2::Debug delete

パラメータ：

```
<object reference>
```

このコマンドは、指定されたオブジェクト上で削除コマンドを実行します。オブジェクト参照を探すには、コマンド !U2::Debug path2object を使用します。

!U2::Debug display

パラメータ：

```
<object reference>
```

このコマンドを使用して、オブジェクトの値をコンソールに表示できます。オブジェクト参照を探すには、コマンド !U2::Debug path2object を使用します。

!U2::Debug echo

パラメータ：

```
<arguments>
```

このコマンドは、出力ウィンドウの Model Verifier タブにすべての引数を表示します。

!U2::Debug exec

パラメータ：

```
<command>
```

このコマンドは、パラメータをコマンドとして実行します。

例 **137:**

```
!U2::Debug exec "Go-Forever"
```

!U2::Debug go

これは 420 ページの「実行」と同等なコマンドです。

!U2::Debug new

パラメータ：

```
<object reference>
```

このコマンドは、指定されたオブジェクト上で新しいコマンドを実行します。オブジェクト参照を探すには、コマンド !U2::Debug path2object を使用します。

!U2::Debug next transition

これは 420 ページの「次の遷移」と同等なコマンドです。

!U2::Debug open

パラメータ：

```
<path name> <optional argument>
```

このコマンドを使用して、シナリオ ファイル（.ttdscn）をロードします。またはモデル ベリファイヤ（Model Verifier）の設定のロード（.ttdcfg）の場合にこのコマンドを使用します。オプションの引数は構成ファイルのみに有効です。

- 引数 -merge を追加すると、開く構成ファイル内のオブジェクトは現在の構成内のオブジェクトとマージされます。
- 引数を省略すると、構成ファイル内のオブジェクトによって現在の構成内の対応するオブジェクトが上書きされます。

例 **138:** パラメータのあるタイマー

```
!U2::Debug open c:/project/test.ttdcfg -merge
```

!U2::Debug output

パラメータ：

```
-from <sender path> -to <receiver path> [-via <connector
name>]:<signal name and parameters in parenthesis>
```

このコマンドは、指定されたメッセージを送信します。

!U2::Debug path2object

パラメータ：

```
<instance path>
```

このコマンドは、インスタンス パスのオブジェクト参照を返します。オブジェクト参照は他のコマンドを実行するために必要です。

!U2::Debug restart

これは 421 ページの「再実行」と同等なコマンドです。

!U2::Debug save

パラメータ：

```
<path name> <optional arguments>
```

このコマンドを使用して、シナリオを保存します。または後から使用できるようにモデル ベリファイヤ（Model Verifier）の設定の保存を行う場合、このコマンドを使用します。シナリオ ファイルには拡張子 `.ttdscn` が必要です。また、構成ファイルには、拡張子 `.ttdcfg` を付ける必要があります。

オプションの引数は構成ファイルのみに有効です。オプションの引数では、構成内のどのオブジェクトを保存するかを決定します。引数が指定されていない場合は、すべてのオブジェクトが保存されます。使用可能な引数は以下のとおりです。

引数	定義
`-messages`	メッセージが保存されます。
`-breakpoints`	ブレークポイントが保存されます。
`-gr_trace`	実行追跡レベルが保存されます。
`-msc_trace`	シーケンス図トレース レベルが保存されます。
`-watches`	［ウォッチ］ウィンドウ内のルート要素が保存されます。

このコマンドのパラメータの例を以下に示します。

```
c:/project/test.ttdscn
```

```
c:/project/test.ttdcfg -messages -msc_trace
```

!U2::Debug set_breakpoint

パラメータ：

```
<U2 statement>
```

このコマンドは、モデルの参照として指定された UML ステートメント上にブレークポイントを設定します。

!U2::Debug set_msc_trace

パラメータ：

```
<object reference> <integer trace level>
```

このコマンドは、指定されたオブジェクトのシーケンス図トレース レベルを設定します。オブジェクト参照を探すには、コマンド !U2::Debug path2object を使用します。

<integer trace level> の値は以下のとおりです。

0= しない
1= ソースまたはターゲット
2= 常に
3= ブロック レベル
4= 親に従う

例 **139:** トレース レベルの設定

```
!U2::Debug set_msc_trace [U2::Debug path2object {CoffeeMachine}] 3
```

!U2::Debug set_replay

パラメータ：

<boolean>

このコマンドは、ブール式が真の場合にモデル ベリファイヤ（Model Verifier）をリプレイ モードに設定します。

!U2::Debug set_tracking_level

パラメータ：

<object reference> <integer trace level>

このコマンドは、指定されたオブジェクトの実行追跡レベルを設定します。オブジェクト参照を探すには、コマンド !U2::Debug path2object を使用します。

<integer trace level> の値は以下のとおりです。

0= しない
1= コントロールに従う時
2= 継続的
3= 親に従う

!U2::Debug start_msc

パラメータ：

<Trace value>

このコマンドはシーケンス図内のイベントのログ記録を動的に開始します。ログ記録を停止するには、コマンド !U2::Debug stop_msc を使用します。有効なトレース レベルは以下のとおりです。

0 = メッセージのみ
1 = メッセージとステート
2 = 完全トレース、アクション、メッセージ、ステート

!U2::Debug step in

これは 420 ページの「ステップ イン」と同等なコマンドです。

!U2::Debug step local

これは 420 ページの「ステップ ローカル」と同等なコマンドです。

!U2::Debug step out

これは 420 ページの「ステップ アウト」と同等なコマンドです。

!U2::Debug step over

これは 420 ページの「ステップ オーバー」と同等なコマンドです。

!U2::Debug stop_msc

このコマンドはシーケンス図内のイベントのログ記録を停止します。

Activity-Mode

パラメータ：（なし）

このコマンドは、モデル ベリファイヤ（Model Verifier）をアクティビティ シミュレーションに適したモードに設定します。アクティビティ モードでは、モデル ベリファイヤ（Model Verifier）は実行する次のステートメントを表示しません。代わりにアクティビティの実行に集中します。ADSim アドインが起動されると、アクティビティ図で実行が追跡されます。

アクティビティ シミュレーション モードを終了するには、Statemachine-Mode コマンドを使用します。

Assign-Value

パラメータ：

```
<object reference> <value>
```

これは、431 ページの「!U2::Debug assign」と同等のコマンドです。

ASN1-Value-Notation

パラメータ：（なし）

値のすべての出力に使用される値表記は、ASN.1 値表記に設定されます。

Breakpoint-Output

パラメータ：

```
<Signal name> <Receiver class name> <Receiver instance
number> <Sender class name> <Sender instance number>
<Counter> <Optional breakpoint commands>
```

ブレークポイントが挿入されて、ブレークポイント コマンドが定義されます。ブレークポイント条件によって 1 つ以上のシグナルの送信が定義されます。ブレークポイント条件はパラメータによって指定されます。

<Counter> パラメータは、実行がブレークされる前にブレークポイント条件が何回真になる必要があるかを示します。このパラメータのデフォルト値は 1 です。つまり、ブレークポイント条件が真になるたびに実行が停止されます。

<Optional breakpoint commands> パラメータを使用して、ブレークポイントがトリガされるときに 1 つ以上のモデル ベリファイヤ（Model Verifier）コマンドを実行できます。コマンドは「;」、つまりスペース、セミコロン、スペースで区切る必要があります。

Breakpoint-Transition

パラメータ：

```
<Class name> <Instance number> <State name> <Signal name>
<Sender class name> <Sender instance number> <Counter> <Optional
breakpoint commands>
```

ブレークポイントがアクティブになり、ブレークポイント条件が定義されます。ブレークポイント条件に遷移が一致すると、遷移が開始される直前に実行が停止されます。ブレークポイント条件は 1 つ以上の遷移に一致します。ブレークポイント条件はパラメータによって指定されます。どのパラメータも省略できます。その場合は、任意の値がブレークポイント条件の未指定フィールドに一致することになります。最初はアクティブになっているブレークポイントはありません。

<Counter> パラメータを使用して、ブレークポイントがアクティブになる前にブレークポイント条件が何回真になる必要があるかを示します。このパラメータのデフォルト値は 1 です。つまり、ブレークポイント条件が真になるたびに実行が停止されます。

<Optional breakpoint commands> パラメータを使用して、ブレークポイントがトリガされるときに実行するコンソール コマンドを指定できます。コンソール コマンドは「;」、つまりスペース、セミコロン、スペースで区切る必要があります。

Breakpoint-Variable

パラメータ：

```
<Attribute name> <Optional breakpoint commands>
```

このコマンドは、現在の範囲で指定されているアクティブ クラスのインスタンス内の指定された属性上にブレークポイントを挿入します。属性の値が変更されると、実行がブレークします。値はシンボル間およびアクション内の割り当てステートメント間のみでチェックされます。

ブレークポイントは属性が存在しなくなるとき、つまり、属性の含まれている Pid が停止されるか、属性の含まれている操作が終りに到達するときもトリガされます。その場合は、ブレークポイントが自動的に削除されます。

`<Optional breakpoint commands>` パラメータを使用して、ブレークポイントがトリガされるときに 1 つ以上のモデル ベリファイヤ（Model Verifier）コマンドを実行できます。コマンドは「 ;」、つまりスペース、セミコロン、スペースで区切る必要があります。

Call-env

パラメータ：

このコマンドは Simulation kind を With Environment に設定して生成されたモデル ベリファイヤのために xInEnv を呼び出します。

Cd

パラメータ：

`<Directory>`

このコマンドは、現在の作業中ディレクトリを指定されたディレクトリに変更します。

Clear-Coverage-Table

パラメータ：（なし）

このコマンドを使用して、すべての位置でカバレッジ テーブルを 0 にリセットします。つまり、現時点からカバレッジのカウントが再開されます。

Close-Signal-Log

パラメータ：

`<Entry number>`

指定されたエントリ番号付きのシグナル ログを停止して、対応するログ ファイルを閉じます。ログ ファイルが 1 つのみ使用されている場合は、エントリ番号パラメータを省略できます。

Command-Log-Off

パラメータ：（なし）

コマンド ログ機能がオフになります。詳細については、コマンド Command-Log-On と比較してください。

Command-Log-On

パラメータ：

`<Optional file name>`

このコマンドは、モデル ベリファイヤ（Model Verifier）コンソールで入力されたすべてのコマンドのログ記録を有効にします。初めてこのコマンドを使用するときは、ログ ファイルのファイル名を指定する必要があります。その後は、ファイル名が指定されていない Command-Log-On コマンドによって直前のログ ファイルに情報が追加されます。また Command-Log-On コマンドにファイル名が指定された場合は、古いログ ファイルが閉じられて、指定された名前の新しいファイルの使用が開始されます。

最初はコマンド ログ機能がオフになっています。アクティブにしたコマンド ログ機能は、コマンド Command-Log-Off を使用して明示的にオフにできます。

生成されたログ ファイルは、コマンド Include-File で直接ファイルとして使用できます。ただし、コマンド エラーによって実行されなかったコマンドも含めてセッションで指定されたコマンドのみが対象になります。最後の Command-Log-Off コマンドもログ ファイルの一部になります。

Create

パラメータ：

```
<object reference>
```

これは、433 ページの「!U2::Debug new」と同等のコマンドです。

Define-Integer-Output-Mode

パラメータ：

```
"dec" | "hex" | "oct"
```

整数値を 10 進、16 進、8 進のどの形式で出力するかを定義します。16 進形式では出力の先頭に「0x」が付き、8 進形式では出力の先頭に「0」（ゼロ）が付きます。

入力では、形式を 16 進または 8 進に設定する場合、以下のように文字列によって基数を決定します。オプションの先行符号の後の先行ゼロは、8 進変換を示します。また、先行する「0x」は 16 進変換を示します。それ以外の場合は、10 進変換が使用されます。

デフォルトは「dec」です。入力変換は実行されません。

Define-MSC-Trace-Channels

パラメータ：

```
"On" | "Off"
```

シーケンス図トレースで、env インスタンスを「env」のコネクタごとに 1 つのインスタンスに分離するかどうかを定義します。デフォルトは「Off」です。

Display-Array-With-Index

パラメータ：

```
"On" | "Off"
```

値が On になっていて配列が実行トレースに表示される場合、配列要素の値はそのインデックスと一緒に出力されます。インデックスは配列要素の値の前に追加されます。デフォルトは「Off」です。

Examine-Timer-Instance

パラメータ：

`<Entry Number>`

指定したタイマー インスタンスのパラメータを出力します。エントリ番号は List-Timer コマンドの使用時にタイマーに関連付けられた番号です。

Examine-Variable

パラメータ：

`<object reference>`

これは、432 ページの「!U2::Debug display」と同等のコマンドです。

Exit

パラメータ：（なし）

これは、421 ページの「デバッグの停止」と同等のコマンドです。

Go

パラメータ：（なし）

これは、420 ページの「実行」と同等のコマンドです。

Go-Forever

パラメータ：（なし）

処理の実行はブレークポイントがアクティブになるまで、または実行を手動で停止するまで続行します。遷移の実行を停止するには、`Enter` キー（このキーのみ）を押します。

UML モデルが完全に休止状態になると（可能な遷移とアクティブなタイマーがない）、外部操作があるまでシミュレーションが中断されます。

Include-File

パラメータ：

`<File Name>`

このコマンドは、テキスト ファイルに格納された一連のコンソール コマンドの実行を可能にします。Include-File 機能は、初期化シーケンスやテスト ケースなどをインクルードするのに有用です。コマンドに含まれるシーケンスで Include-File を使用できます。5 レベルまでネスティングされたインクルード ファイルを処理できます。

List-Breakpoints

パラメータ：(なし)

このコマンドはモデル内のすべてのアクティブ ブレークポイントをリストします。それぞれのブレークポイントにエントリ番号が割り当てられます。

List-GR-Trace-Values

パラメータ：(なし)

このコマンドはテキスト追跡に使用されているトレース値をリストします。

List-MSC-Log

パラメータ：(なし)

このコマンドはシーケンス図ログの現在のステータスを返します（オフ / 対話式 / バッチ)。

List-Ready-Queue

非推奨コマンド。インクルードなし。

List-Signal-Log

パラメータ：(なし)

現在のアクティブ シグナル ログに関する情報を出力します。それぞれのログにエントリ番号が割り当てられます。

List-Timer

パラメータ：(なし)

現在アクティブなタイマーのリストを生成します。タイマーごとに、対応するプロセス インスタンスと関連する時間が示されます。エントリ番号もリストに示されます。これは、Examine-Timer-Instance コマンドで使用されます。

List-Trace-Values

パラメータ：(なし)

現在定義されているすべてのトレースの値がリストされます。リストにはトレース ユニット タイプ（「system」、「block」、「process」、Pid）、ユニット名、トレース値（数値とテキストによる説明の両方）が含まれています。

Log-Off

パラメータ：（なし）

このコマンドは対話ログ機能をオフにします。Log-On と比較してください。

Log-On

パラメータ：

```
<Optional file name>
```

このコマンドは、オプションのファイル名をパラメータとして取り、ユーザーと画面に表示されているシミュレーション プログラム間のすべての対話のログ記録を有効にします。初めてこのコマンドを入力するときは、ログ ファイルのファイル名をパラメータとして指定する必要があります。その後は、ファイル名を指定せずにコマンドを入力すると、情報が直前のログ ファイルに追加されます。ファイル名を指定してコマンドを入力するたびに古いログ ファイルが閉じられ、指定されたファイル名の新しいファイルが開きます。

最初は対話ログ機能がオフになります。対話ログ機能を明示的にオフにするには、コマンド Log-Off を使用します。

Next-Local-Statement

パラメータ：（なし）

これは、420 ページの「ステップ ローカル」と同等のコマンドです。

Next-Statement

パラメータ：（なし）

これは、420 ページの「ステップ イン」と同等のコマンドです。

Next-Transition

パラメータ：（なし）

これは、420 ページの「次の遷移」と同等のコマンドです。

Next-Visible-Transition

パラメータ：（なし）

トレース値が無効化されていない次の遷移（それ自体を含む）までの多くの遷移が実行されます。タイマー出力遷移の場合は、対応するインスタンスのトレース値が考慮されます。

このコマンドは、アクティブ トレースの設定されている遷移が実行されないと、シミュレーション プログラムが無制限に実行される場合があるので注意して使用する必要があります。遷移の実行を停止するには、<Enter> キーを押します。

Now

パラメータ：（なし）

シミュレーション時間の現在の値が出力されます。

Print-Coverage-Table

パラメータ：

```
<File name>
```

このコマンドを使用して、テスト カバレッジ情報とプロファイリング情報を取得できます。このコマンドを実行するたびに、パラメータとして指定された名前のファイルが関連の情報とともに作業ディレクトリに作成されます。カバレッジ ファイルは常にシミュレーション開始時からの状況を反映しています。

注記

指定されたファイルは常に上書きされます。つまり、既存のファイルが指定されている場合、確認メッセージは表示されません。

生成されたファイルは 2 つのセクションで構成されています。最初のセクションは、遷移の数と各アクティブ クラスによって実行されたシンボルの数を含めたプロファイリング情報付きサマリになっています。2 番目のセクションには、それぞれのシンボルと state - input の組み合わせが実行された回数に関する詳細な情報が含まれています。

例 **140:** カバレッジ ファイル内のプロファイリング情報 ——————————

```
*************** PROFILING INFORMATION **************

2  System DemonGame :Transitions = 13, Symbols = 40
3  Block GameBlock
4  Process Main :Transitions = 3 (23%),
                 Symbols = 10 (25%), MaxQ = 2
4  Process Game :Transitions = 6 (46%),
                 Symbols = 15 (37%), MaxQ = 2
3  Block DemonBlock
4  Process Demon :Transitions = 4 (30%),
                 Symbols = 15 (37%), MaxQ = 1
```

——————————————————————————————————————

この情報は以下のように解釈します。

- **transitions** として、実行されたシグナル受信シンボル、トリガされた遷移上のガード シンボル、およびスタート シンボルの数がカウントされます。

- **symbols** として、実行されたシンボル（アクション、シグナル送信、分岐、設定、リセット、呼び出し、ストップ、リターン、作成、次のステート、入力、トリガされた遷移上のガード、スタート）の数がカウントされます。
- 上記の例に示すように、遷移の数と実行されたシンボルの数は、いずれも「system」と各アクティブ クラスに対応して提供されます。もちろん、アクティブ クラスに対応する相対数は「system」に対応する数が基準になります。
- **MaxQ** は、アクティブ クラスの任意のインスタンスの入力ポート待ち行列最大長を表します。作成を実装するための開始シグナルと、トリガされた遷移上のガード シグナルは MaxQ でシグナルとしてカウントされます。
- 各行の先頭の数は、そのユニットの範囲レベルを表します。これを使用して、たとえば、操作が別の操作内または別の操作の後に定義されるかどうかを判定できます。

注記

本来は、遷移の数と実行されたシンボルの数ではなくプロファイリング情報実行時間を測定する必要がありますが、この情報は負荷分散を把握する上で非常に効果的です。

Proceed-To-Timer

パラメータ：（なし）

このコマンドは次のタイマー出力まで（次のタイマー出力自体は含まない）のすべての遷移を実行します。タイマー出力は次の遷移の場合も実行されません。

Proceed-Until

パラメータ：

`<Time value>`

シミュレーションの実行が再開されます。コンソールは時間の値がパラメータとして指定されている時間型値に等しくなったときに初めてアクティブになります。

相対時間値は「+」符号を使用して指定できます。パラメータとして「+5.0」と入力すると、時間型値 NOW+5.0 として解釈されます。

Quit

パラメータ：（なし）

これは、421 ページの「デバッグの停止」と同等のコマンドです。

REF-Address-Notation

パラメータ：（なし）

REF 型値（Own テンプレートと ORef テンプレートを使用して導入されるポインタ）は、アドレスの HEX 型値を使用してアドレスとして出力されます。ヌル値は `Null` として出力されます。入力では、この構文と Value Notation の両方を使用できます（コマンド REF-Value-Notation と比較してください）。

注記

このコマンドは、出力ウィンドウで実行トレースに表示される値のみに適用されます。

REF-Value-Notation

パラメータ：(なし)

REF 型値（Own テンプレートと ORef テンプレートを使用して導入されるポインタ）は、`NEW(<the value the pointer refers to>)` として出力されます。これは REF 型値のデフォルト構文です。つまり、完全なリストまたはグラフが出力されます。

例 **141:**

```
NEW( (. 1, NEW( (. 2, Null .) ) .) )
```

循環グラフの問題が発生しないようにするには、ポインタですでに出力に含まれているアドレスを参照する場合に特殊な構文を使用します。`OLD n`（この場合の n は数字）は、出力される値の n 番目の NEW の参照を表します。

例 **142:**

```
NEW( (. 1, NEW( (. 2, OLD 1 .) ) .) )
```

ヌル値は `Null` として出力されます。入力では、この構文と Address-Notation の両方を使用できます（コマンド REF-Address-Notation と比較してください）。

注記

このコマンドは、出力ウィンドウで実行トレースに表示される値のみに適用されます。

Remove-All-Breakpoints

パラメータ：(なし)

このコマンドは挿入されたすべてのブレークポイントを削除します。

Remove-Breakpoint

パラメータ：<entry number>

このコマンドはモデル内のブレークポイントを削除します。`List-Breakpoints` コマンドは、挿入されたそれぞれのブレークポイントのエントリ番号をリストします。

Reset-GR-Trace

パラメータ：

`<Optional unit name>`

指定ユニットの GR トレース値を未定義にリセットします。ユニットを指定しない場合、システムの GR トレース値が未定義にリセットされます。システムには必ず GR トレース値が定義されていなければならないので、システムに対して Reset-GR-Trace を実行した場合、GR トレース値は 0 であると解釈されます。

Reset-Timer

パラメータ：

```
<Timer name> <Timer parameters>
```

コマンドの結果は、現在の範囲内のプロセス クラスのインスタンスによってリセット アクションが実行された場合とまったく同じになります。リセット アクションが要因となってタイマー シグナルが削除され、このシグナルが次の遷移用に選択されている場合は、プロセス インスタンスによって暗黙の [次の遷移] アクションが実行されます。

Reset-Trace

パラメータ：

```
<Optional unit name>
```

指定されたユニットのトレース値が undefined にリセットされます。ユニットが指定されない場合は、システムのトレース値が undefined にリセットされます。システムでは常に定義されたトレース値が必要になるため、システム上の Reset-Trace はトレース値の 0 への設定と等価に見なされます。

注記

ユーザー インターフェイスはこのコマンドの使用時に更新されません。

SDL-Value-Notation

パラメータ：（なし）

値のすべての出力に使用される値表記は、SDL 値表記に設定されます。これはデフォルトの値表記です。

Save-Breakpoints

パラメータ：

```
<File name>
```

このコマンドはすべてのブレークポイントをファイルに保存します。コマンドとともにテキスト ファイルとして作成されます。

Save-State

パラメータ：

```
   <File name>
```

このコマンドは、現在のシミュレーションの状態を作業ディレクトリのファイルに保存します。ファイル名はパラメータで指定された名前になります。Restore-State コマンドによって状態を元に戻せるように、特殊なコマンドを持つテキスト ファイルとして書き込まれます。

Set-GR-Trace

パラメータ：

```
<Optional unit name> <Trace value>
```

GR トレース値が、指定されたエントリに割り当てられます。ユニットが指定されていない場合、GR トレース値はモデル全体に割り当てられます。システムの初期 GR トレース値は 0、ダイアグラムの実行追跡は無効です。値 1 を指定すると追跡が実行されます。

Set-MSC-Trace

パラメータ：

```
<Optional unit name> <Trace value>
```

これは、434 ページの「!U2::Debug set_msc_trace」と同等のコマンドです。

Set-Timer

パラメータ：

```
<Timer name> <Timer parameters> <Time value>
```

コマンドの結果は、現在の範囲内のアクティブ クラスのインスタンスによってセット タイマー アクションが実行された場合とまったく同じになります。セット アクションが要因となってタイマー シグナルが削除され、このシグナルが次の遷移用に選択されている場合は、インスタンスによって暗黙の [次の遷移] アクションが実行されます。

Set-Trace

パラメータ：

```
<Optional unit name> <Trace value>
```

テキスト トレース レベルが、指定されたユニットまたはノードに割り当てられます。ユニットが指定されていない場合、トレース値はモデル全体に割り当てられます。トレース値は、どのタイプの情報がテキスト トレースに表示されるかを指定します。モデルの初期トレース値は 4 ですが、それ以外のユニットの場合は未定義になります。

Show-C-Line-Number

パラメータ：(なし)

このコマンドは、実行が停止される C コード内の位置を表示します。ファイル名と行は出力ウィンドウに表示されます。

Show-Next-Symbol

パラメータ：（なし）

次に実行されるシンボルが状態機械図で選択されます。

Show-Previous-Symbol

パラメータ：（なし）

最後に実行されたシンボルが状態機械図で選択されます。

Show-Versions

パラメータ：（なし）

SDL to C Compiler のバージョンと、現在実行中のプログラムを生成したランタイム カーネルが表示されます。

Signal-Log

パラメータ：

```
<Unit name> <File Name>
```

このコマンドは、指定されたファイルへのシグナルのログ記録を開始します。`<Unit name>` パラメータは、どのエンティティがログ記録されるかを指定します。このエンティティとの間で送受信されるシグナルのみがファイルにリストされます。ログの完了時にコマンド Close-Signal-Log を使用します。

ログを表示するには、コマンド List-Signal-Log を使用します。

Start-Batch-MSC-Log

パラメータ：

```
<Symbol level> <File name>
```

このコマンドは、対応するシーケンス図イベントに変換できるイベントのログ ファイルへの入力を開始します。ログ記録を停止するには、コマンド Stop-MSC-Log を使用します。

結果はログ ファイルに格納されます。そのログ ファイルは後からシーケンス図に表示できます。

シンボル レベル パラメータによって、ステータスとアクションをログに含めるかどうかが判定されます。可能なシンボル レベル値の記述については、

このコマンドのファイル名パラメータには、任意の有効なファイル名を使用できます。

Start-env

パラメータ：（なし）

このコマンドを使用すると、Simulation kind が With Environment に設定されて生成されたモデル ベリファイヤから環境 API を呼び出す準備を初期化されます。

Statemachine-Mode

パラメータ：(なし)

デフォルトで、モデル ベリファイヤ（Model Verifier）は状態機械 モードで動作します。つまり、実行を状態機械図（およびテキスト図）で追跡できます。しかし、アクティビティをシミュレートする場合は、アクティビティの実行に重点が置かれ、専用のアクティビティ モードが使用されます（Activity-Mode コマンドを参照）。Statemachine-Mode コマンドは、アクティビティ シミュレーション モードを終了するために使用します。

Stop-env

パラメータ：(なし)

このコマンドは、モデル ベリファイヤ実行形式からの環境 API 機能の呼び出しを停止します。このコマンドはモデル ベリファイヤが Simulation kind を With Environment に設定して生成された場合のみ使用できます。

Stop-MSC-Log

パラメータ：(なし)

このコマンドはシーケンス図イベントのログ記録を停止します。バッチ モード ログ記録の場合は、ログ ファイルが閉じられます。詳細については、コマンド Start-Batch-MSC-Log と比較してください。

このコマンドの後に、セッションの残りを新しいファイルにログ記録できます。

特殊なコンソールコマンド

次のリストに、いろいろな理由で完全に説明されていないコンソール コマンドを示します。これらのコマンドは通常使用する必要はありません。Tau サポートから推奨された場合のみ、複雑なシミュレーション場面で、使用されることを想定しています。いくつかのコマンドは、他のメニューやツールバー コマンドと酷似しているため、非推奨扱いとなっているものもあります。

%

Breakpoint-At

Define-At-Delay

Define-Continue-Mode

Define-Delay

Down

Examine-PId

Examine-Signal-Instance

Finish

List-Input-Port

List-MSC-Trace-Values

List-Process

Nextstate

Next-Symbol

Output-From-Env

Output-Internal

Output-None

Output-To

Output-Via

Performance-Simulation

Print-Paths

Rearrange-Input-Port

Rearrange-Ready-Queue

REF-Deref-Value-Notation

Remove-At

Remove-Delay

Remove-Signal-Instance

Reset-MSC-Trace

Restore-State

Save-Delay

Scope

Set-Scope

Show-Breakpoint

Stack

Start-Interactive-MSC-Log

Step-Statement

Step-Symbol

Stop

SymbolTable

Up

xSet

リプレイ モード

リプレイ モードでは、以下のステップとユーザー コマンドを記録できます。

- 実行ステップ
- ユーザー コマンド

実行ステップ

以下の実行ステップがシナリオに記録されます。

- **transition** は、アクティブ クラス インスタンスのパス、その開始元のステート、および使用されるシグナルによって記述されます。以下のように表示されます。

  ```
  active_class_instance_path:from state_path input signal_path
  ```

- **time-out** は、タイマーとタイマーのパスを設定するアクティブ クラス インスタンスのパスによって記述されます。以下のように表示されます。

  ```
  active_class_instance_path:timer timer_path
  ```

ユーザー コマンド

シナリオに記録されるユーザー コマンドは、アプリケーションのステートを変更するコマンドです。

- **Output** は、［メッセージ］ウィンドウに表示される情報によって記述されます。以下のように表示されます。

  ```
  output from sender_path [via connector name] to
  destination_path :signal_path [ (parameters values) ]
  ```

- **New** は、作成されたオブジェクトの親のパスによって記述されます。以下のように表示されます。

  ```
  new object_path
  ```

- **Delete** は、削除されたオブジェクトのパスによって記述されます。以下のように表示されます。

  ```
  delete object_path
  ```

- **Rearrange** は、親オブジェクトのパス、ソートと宛先の位置によって記述されます。以下のように表示されます。

  ```
  collection_path:source_index -> destination_index
  ```

- **Assignment** は、割り当てられたオブジェクトのパスと、新しい値の式によって記述されます。以下のように表示されます。

  ```
  assigned_object_path = new_value
  ```

動的エラー

エラー メッセージは 452 ページの例 143 のように表示されます。

例 **143:** 動的エラーの出力

```
************************ ERROR ************************
Error in SDL Decision:Value is 12:

Entering decision error state
TRANSITION
  Process         :Ctrl:1
  State           :Idle
  Input           :Coin
  Symbol          :#SDTREF(U2,"u2:C:¥Program
Files¥IBM¥Rational¥TAU¥4.3¥¥examples¥CM¥
CM.u2#7t6K1VwL19VLrn8XRELUlHzI|pos(1,18)")
TRACE BACK
  Process         :Ctrl
  System          :CoffeeMachine
*******************************************************
```

動的エラー発生時のアクション

動的エラーが検出されると、現在のシンボルの終りまでシミュレーション プログラムが続行します。以下のアクションが実行されます。

- エラーがシグナル送信エラーの場合、シグナルは送信されません。
- エラーが分岐エラーの場合は、アクティブ クラスのインスタンスがすぐに***分岐エラー ステート***になります。分岐エラー ステートに入ったときに、入力ポートは影響を受けません。分岐エラー ステートでアクティブ クラスのインスタンスに送信されたすべてのシグナルは、入力ポートに保存されます。
- アクティブ クラスの静的インスタンスの作成時にエラーが発生した場合、つまりインスタンスの初期数がインスタンスの最大数より多い場合は、エラー メッセージが表示されて、インスタンスの初期数によって指定された数のインスタンスが作成されます。
- インポートまたは表示アクションの実行中にエラーが発生した場合は、すべての位置内のゼロが含まれている正しいサイズのデータ領域が返されます。
- 割り当て時に範囲条件チェックでエラーが検出された場合は、境界外の場合でも割り当て演算子の左辺の属性にシステム計算された値が割り当てられます。
- 配列インデックスの範囲チェック時にエラーが検出された場合は、インデックス値がインデックス型の最低値に変更されます。つまり、対応する C 配列はその境界外ではインデックスが付きません。
- 構造体内のオプションのコンポーネントの選択時、または `choice` でのコンポーネントの選択時にエラーが発生した場合は、エラー メッセージが表示されて、操作が実行されます。
- ヌル ポインタ（Own と ORef）の参照読み出しが実行されると、ゼロを含んだ正しいサイズの新規データ領域が割り当てられます。このデータ領域はポインタに割り当てられます。エラー メッセージの表示後に、参照読み出しが含まれているステートメントが実行されます。
- 式内でエラーが発生した場合は、エラーを検出した演算子によってデフォルト値が返され、式の評価が続行します。

UML へのインポートとエクスポート

「UML へのインポートとエクスポート」セクションの各章では、それぞれ固有のフォーマットをもつ、他のツール間で情報をやりとりする方法 について説明しています。

8

.NET アセンブリ インポータ

.NET アセンブリ インポータは、.NET アセンブリまたは COM コンポーネントを UML モデルにインポートするためのツールです。.NET アセンブリまたは COM コンポーネントを UML モデルにインポートすると、その内容を可視化して、UML モデルからアクセスできるようになります。

ここでは、.NET アセンブリと COM コンポーネントを表す用語として「**コンポーネント**」を使用します。.NET アセンブリ インポータは、どちらもほぼ同じ方法で処理します。

.NET アセンブリ インポータは、Windows プラットフォームでのみ使用可能です。

動作原理

.NET アセンブリ インポータの主な適用領域は、コンポーネントの UML 表現を作成することです。この UML 表現には、主に以下の 2 つの目的があります。

- 可視化。コンポーネントをインポートして、その内容をダイアグラムとモデル ビュー内で図示できます。
- アクセス。コンポーネントからインポートした定義は、アクセスして UML モデルで使用できます。

コンポーネントのインポートでは、コンポーネントが公開している定義だけがインポートされます。コンポーネントの実装はモデルにインポートされません。コンポーネントからインポートされたすべての定義は、1 つまたは複数の UML パッケージに格納されます。さらに、モデルにインポートされたコンポーネントを表すファイル アーティファクトが作成されます。

コンポーネントのインポート

UML にコンポーネントをインポートするには次の手順を行います。

- ［モデル ビュー］で**モデル** 項目を選択する。
- インポート ウィザードを開く（［ファイル］メニューから［インポート］コマンドを選択）。
- このダイアログで、［.NET アセンブリまたは COM コンポーネントのインポート］を選択する。

以下のダイアログが表示され、インポートするコンポーネントの指定が可能になります。

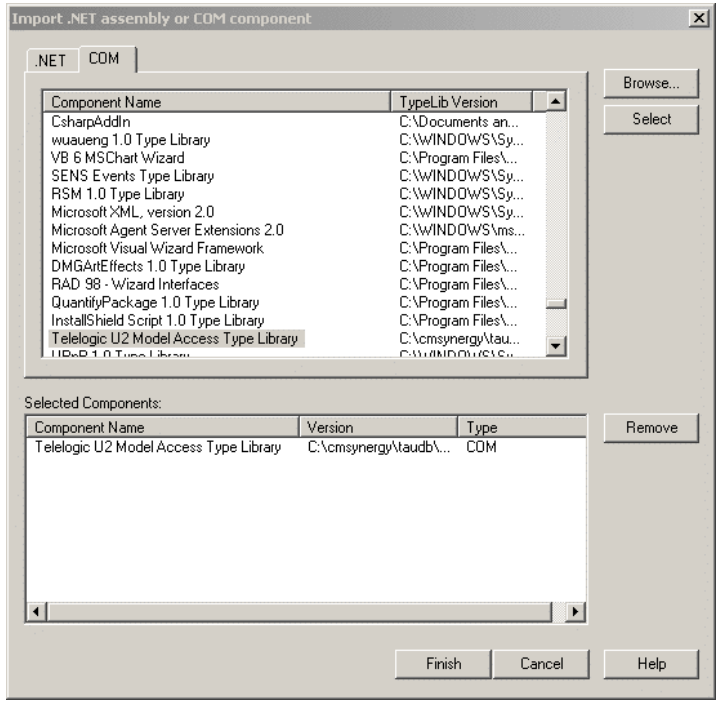


図 168: ［.NET アセンブリまたは COM コンポーネントのインポート］ウィザード

ダイアログの［.NET］タブには、コンピュータのグローバル アセンブリ キャッシュ（GAC）に格納されているすべての .NET アセンブリが表示されます。各アセンブリのバージョン番号も表示されます。

［COM］タブには、コンピュータのシステム レジストリに登録されているすべての COM コンポーネントが表示されます。各 COM コンポーネントのタイプ ライブラリに格納されているファイルも表示されます。

インポートしたいコンポーネントが GAC にもシステム レジストリにも登録されていない場合、どちらのリストにも表示されません。この場合には、［参照］ボタンを使用して、ファイル システム内でそのコンポーネントを探すことができます。

インポートする .NET アセンブリまたは COM コンポーネントは、いくつでも選択できます。［完了］ボタンをクリックすると、下側のリストに表示されているすべてのコンポーネントがインポートされます。

コンポーネントの再インポート

通常、コンポーネントをモデルにインポートするのは一度だけです。しかし、同じコンポーネントを、後でもう一度インポートしたい場合があります。たとえば、コンポーネントに変更を加えた場合、もう一度このコンポーネントをインポートすることで、変更を反映してモデルを更新できます。

コンポーネントを再インポートしてモデルを更新するには、モデルのコンポーネントを表すファイル アーティファクトのショートカット メニューから［コンポーネントのインポート］コマンドを選択します。

翻訳ルール

このセクションでは、コンポーネントを UML モデルにインポートする際に使用される翻訳ルールについて説明します。ここで使用される用語は COM コンポーネントではなく、.NET アセンブリに即しています。その理由は、インポートされる COM コンポーネントが、.NET フレームワークの一部である標準のタイプ ライブラリ インポータ ツールを使用して、まず .NET アセンブリに変換されるからです。この interop アセンブリが、次に UML に翻訳されます。

アセンブリと名前空間

各名前空間は、同じ名前を持つ UML パッケージに翻訳されます。

アセンブリ自身も、アセンブリに含まれるタイプの最上位のスコープを定義します。同様に、アセンブリも同じ名前を持つ UML パッケージに変換されます。

クラス

クラスは、同じ名前を持つ UML クラスに翻訳されます。

クラスが別のクラスを継承するか、あるいはインターフェイスを実装する場合、UML クラスは対応する汎化を含みます。

クラスのフィールドは、UML クラスの対応する属性に翻訳されます。

インターフェイス

インターフェイスは、同じ名前を持つ UML インターフェイスに翻訳されます。

インターフェイスが他のインターフェイスを継承する場合、UML インターフェイスは対応する汎化を含みます。

メソッド

クラスまたはインターフェイスのメソッドは、対応する UML クラスまたはインターフェイスに含まれる UML 操作に翻訳されます。メソッドのパラメータは、UML 操作のパラメータに翻訳されます。インポートされたメソッドとパラメータの名前は、どちらも元のメソッドとパラメータの名前と同じになります。

列挙

列挙は、同じ名前を持つ UML データ型に変換されます。列挙のリテラルは、UML データ型の対応するリテラルに変換されます。

9

C/C++ のインポート

このセクションは、C/C++ インポート機能のリファレンス ガイドです。C/C++ インポート機能は、一連の C/C++ ヘッダー ファイルを取り込んでプリプロセスを行い、ヘッダー内の宣言を対応する UML 表現に翻訳します。

C/C++ インポータはインポートされた関数のインライン実装にある文もインポートするよう設定できます。これは、C/C++ ヘッダーファイルを UML に移行する際に有用です。

このドキュメントは、この翻訳プロセスのルール、および C/C++ インポートの動作原理について説明します。

動作原理

C/C++ インポートの主な目的は、UML ツール セットで開発された UML アプリケーションから外部 C/C++ コードへのアクセスを提供することにあります。C/C++ のインポートでは、一連の C/C++ ヘッダー ファイルを解析し、構文チェックとセマンティック チェックを行い、一連の C/C++ から UML への翻訳ルールに従って C/C++ 定義を対応する UML 定義に翻訳します。

インポートのもう 1 つの目的は、従来のコードを図形表現のドキュメントに変換し、設計、構造、依存関係、および継承のツリーの理解に役立てることです。

さらに、UML で開発を続行し、Tau のコード ジェネレータで新しい C/C++ コードを生成することを目的として、既存の C/C++ アプリケーションを UML に移行する手段として C/C++ インポートを行うこともできます。そのような移行は、アプリケーションの各部分で必ず行わなければならないものではありません。一般的なシナリオとしては、インポートして作成された UML モデルから生成するのはヘッダー ファイルだけにして、C/C++ 実装ファイルは既存のものをそのまま維持管理する方法が考えられます。C/C++ インポートでは、ヘッダー ファイルのインライン実装にあるアクション コード（文）もインポートすることで、この移行シナリオを実現します。C/C++ インポート時に一連のヘッダー ファイル アーティファクトと 1 つのビルド アーティファクトを作成する方法もあります。これらのアーティファクトは、C++ アプリケーション ジェネレータを使用してヘッダー ファイルを再生成するために使用できます。

ターゲット **UML** パッケージ

それぞれの C/C++ のインポートの結果、多数の UML 要素が作成され、ターゲットとして指定されたパッケージに格納されます。開発中の UML モデルを含むパッケージなど、任意のパッケージをターゲットとして割り当てることができますが、最初は空のパッケージを割り当てたほうが、視界を広げて実際にインポートされるものを理解できます。

入力

一連の C または C++ ヘッダー ファイルをインポート スキームの入力として使用できます。入力として指定されたファイルは、実際に翻訳が実行される前に、環境設定に従ってプリプロセスされます。

Preprocessor（プリプロセッサ）

柔軟なインポート スキームを可能するため、C/C++ インポートは C/C++ プリプロセッサの機能である条件付きコンパイルや他のディレクティブを利用し、コードのコンパイル方法に関する基本的な理論をサポートしています。インポート ウィザードの C/C++ モードで、任意のプリプロセッサ、任意のプリプロセッサ オプションの指定をできます。ただし、いくつかのプリプロセッサの制限が適用されます。

注記

C/C++ ヘッダー ファイルのインポートには、プリプロセッシングとインポートの 2 つのステップがあります。プリプロセッシング ステップでは、ヘッダー ファイルがプリプロセスされます。独自のプリプロセッサとプリプロセッサ オプションを使用することもできます。
UML へのインポート ステップでは、すべてのマクロと他のプリプロセッサの構成要素がインポート後の（プリプロセスされた）ファイルに展開されます。オリジナルのマクロ名および定義は保存されず、UML にインポートされません。

構文チェックとセマンティック チェック

C/C++ コードのプリプロセッシングが終了したら、必要な構文チェックおよびセマンティック チェックを行い、UML への翻訳の前にインポートされたヘッダー ファイルの完全性をベリファイします。これらのチェックには、C/C++ インポートでサポートされる C 言語と C++ 言語のサブセットも含まれます。

インポート済み C/C++ コードは、構文的に完全に正しいものである必要があります。構文エラーが検索されて報告されると、インポートは続行できなくなります。

インポート時には、一部のセマンティック チェックが行われます。しかし、C/C++ インポータは、C/C++ コンパイラの代わりではないので、いくつかのセマンティック エラーがあったとしても、プログラムをインポートすることは可能です。インポートするコードの正しさに確証がない場合は、インポートする前に C/C++ コンパイラを使用してコードが正しいかどうかを検証することをお勧めします。

C/C++ から UML への翻訳ルール

インポート可能な C/C++ の構成要素またはトピックの翻訳は、自身のサブセットに記述されます。一般翻訳ルール以降のセクションを参照してください。

ほとんどの言語構成要素はインポートされた C/C++ 定義の UML への翻訳方法の例とともに示されます。

注記

この例では、特定の翻訳ルールのみを示します。翻訳ルールに直接関係していない翻訳後の UML については、省略されています。また、インポート時に適用したオプションによって翻訳後の UML の表示が異なることに注意してください。

作成された UML 定義

生成された UML 定義は、インポートのターゲットとして指定された UML パッケージに入れられます。パッケージがプロジェクトにインクルードされたら、UML モデルからこれらの UML 定義を参照できるようになります。これにより完全な構文分析とセマンティック分析を行うことができます。

それぞれがすべての定義のサブセットのターゲット コンテナとして機能する、複数のパッケージを持つこともできます。これにより、たとえば C と C++ の定義を分離したり、異なるフレーバの C および C++ を使用して開発されたさまざまなオリジンのコードを分離するといった操作ができます。このようなインポートごとに、使用するプリプロセッサ、言語（C または C++）、および C++ 固有表現を個別に設定できます。

ソース ヘッダーへのトレース バック

インポート スキームでは、ソース ヘッダー ファイル内のエンティティの場所を追跡するので、C/C++ プログラミング環境または任意のテキスト エディタによってインポート後のエンティティを検索できます。Tau からの移動は、インポートされたエンティティのコンテキストメニューから［ソースに移動］コマンドを選択して行います。

例 **144:** インポート済みエンティティからのソース ヘッダーへの移動 ——————

以下の C++ ファイル「test.h」が UML へインポートされたとします：

```
class C {};
```

このファイルのインポート後、インポート パッケージの class C を右クリックして、［ソースに移動］を選択できます。インポート済みヘッダー ファイルにあるすべての class C（このケースでは ”test.h: l”）がサブメニューにリストされます。

コンパイラおよび言語のサポート

翻訳ルールは、任意の C/C++ dialect を使用する、C および C++ のターゲット コンパイラをサポートします。ほとんどの場合、翻訳ルールは C または C++ のいずれのターゲット コンパイラが使用されるかに左右されません。ただし、C/C++ のインポートが「C mode」で実行される場合、翻訳ルールを若干修正する必要があります。

修正内容については、523 ページの「C コンパイラ用の翻訳ルール」を参照してください。

C/C++ のインポート

C++ ファイルを UML にインポートするには、以下の手順を行います。

- ［モデル ビュー］で**モデル** アイテムを選択します。
- インポート ウィザードを開きます（［ファイル］メニューの［インポート］コマンド）。
- ダイアログで［C/C++ のインポート］を選択します。

ヘッダー ファイルとオプションは、`cppImportSpecification` ステレオタイプで指定され、パッケージに適用されます。インポートされた定義は、`cppImportSpecification` によってステレオタイプ化されてこのパッケージに入れられます。インポートウィザードは自動的にこのステレオタイプを適用し、タグ

付き値を使用してウィザード中で設定した値を保存します。後でプロパティエディタを使用して、設定を確認、修正でき、さらにウィザードではできない詳細な設定を行えます。詳細は、インポート オプションの指定を参照してください。

インポート入力の指定

インポートウィザードの最初のページでは、入力ファイルを指定します。入力を指定するには、インポートするファイルを参照して選択します。

注記

Windows では、複数の C/C++ ファイルを選択してインポートするか、1 つの Visual Studio .NET プロジェクト ファイル（.vcproj）を選択してインポートするかを、選ぶ必要があります。1 つのプロジェクト ファイルの場合は、そのプロジェクト ファイルで定義された構成も選択する必要があります。インポート時に、選択した構成のプリプロセッサ設定が使用されます。

インポート設定の指定

インポートウィザードの 2 ページ目では、インポートのための一般的なオプションを指定します。以下の指定が可能です。

- **処理系固有の C/C++**：デフォルトは、C++ については ISO/IEC 14882 標準、C については ANSI/ISO 9899-1989 です。サポートされるその他の固有の C/C++ は、GNU、Boland および Microsoft です。詳細については、C/C++ dialect を参照してください。
- **アクションコードのインポート**：このオプションは、インポート対象のファイル内にある関数の本体の処理方法を制御します。デフォルトでは、関数本体は UML モデルにインポートされません。関数本体をインポートしたい場合は、UML のアクションにそれらをインポートするか、それらを非形式なアクションとしてインポートできます。アクション コードをインポートするオプションの典型的例としては、レガシー C/C++ アプリケーションを UML に移行して、その後 C++ アプリケーション ジェネレータでコードを再生成する、というケースがあります。また、C/C++ プログラムを UML 内で可視化したい場合にもこのオプションを使用できます。詳細については Action code strategy を参照してください。
- **ソースファイルは C のみを含む**：デフォルトではインポータはインポートされるヘッダファイルには C/C++ が含まれると仮定します。C++ を使用しない C のみのヘッダファイルをインポートすることが確実な場合はこのオプションを指定します。その指定に従い、インポータは ”C モード ” で動作します。詳細については、C only を参照してください。
- **エクスポート済み定義のみインポート：**一部のコンパイラは、DLL または共有オブジェクトからエクスポートされる定義の指定方法として `__declspec(dllexport)` をサポートします。このオプションを使うと、この構築子の指定に該当する定義のみをインポートできます。あるライブラリのエクスポートされたインターフェイスのみを UML にインポートしたい場合に有用です。詳細については、**Import only exported definitions** を参照してください。

- **スコープ名 ベースの GUID を生成**：デフォルトでは、インポートされたエンティティにはランダムな GUID が割り当てられます。繰り返しインポートを行う場合には、ランダムな ID よりも、インポートされる定義のスコープ付き名前に基づく GUID を生成させる方がより適切です。
- **アーティファクトの生成**：このオプションが指定されている場合、インポータはインポートされたファイルを表現するファイルアーティファクトを生成します。また、C++ アプリケーション ジェネレータ用のビルドアーティファクトも生成します。これらのアーティファクトの使用目的は、C++ アプリケーション ジェネレータを使ったインポートファイルから再生成を容易にすることです。詳細については、Generate artifacts を参照してください。
- **今すぐインポート**：［インポート ウィザード］で［完了］をクリックする前にこのオプションをオフにすると、C/C++ インポートの開始前に、実際のインポートを行わずにインポート オプションを指定できます（初めて C/C++ インポートを使用する場合など）。これらのインポートオプションを指定する方法については、インポート オプションの指定を参照してください。適切なインポート オプションを設定したら、［C/C++ のインポート］コマンド（インポート パッケージのコンテキスト メニューから選択可能）を選択して、インポートを実行します。

  ウィザードで［今すぐインポート］を有効にしていても、後で同じインポート操作を行う場合は（インポート済みファイルで変更を加えた場合など）、［C/C++ のインポート］コマンドを使用する必要があります。そのような繰り返しインポートを行う場合は、特別な注意が必要です。詳細については、繰り返しインポートの考慮点を参照してください。

パッケージ **ImportedDefinitions** 用の **u2** ファイルの指定

C/C++ インポート ウィザードの 3 ページ目で、インポート結果のパッケージの保存先とするモデル（.U2）ファイルを指定できます。パッケージをモデル (`.u2`) ファイルに保存すると、インポートされたヘッダー ファイル パスはこのモデルファイルへの相対パスになります。パスをモデル ファイルと相対的にしない場合は、フラグ「Use absolute paths for input header files」を設定する必要があります。

インポートの出力

インポート結果は現在選択しているモデル ビューのコンテキストにある、「ImportedDefinitions」という名前の新しいパッケージに入れられます。生成された定義を元のパッケージに移動し、ダイアグラムを整理してコメントをつけてもかまいません。

インポート オプションの指定

C/C++ インポートのオプションは、`cppImportSpecification` フィルタ（ステレオタイプ）を使用してプロパティ エディタを開いて変更します。

1. インポート済みパッケージの［プロパティ］ダイアログを開きます。［フィルタ］ドロップダウン メニューで［cppImportSpecification］を選択します。
2. 使用できるオプションを適切な値に調整します。

C/C++ インポート オプションでのインポート入力の指定

cppImportSpecification フィルタ（ステレオタイプ）では、［入力ヘッダー ファイル］フィールドに、ワイルドカード表記を使用できます。したがって、たとえば、このフィールドで `e:¥test¥*.h` または `e:¥test¥*` を指定して、一致するファイル名をインポート用にマークできます。

手動による C/C++ インポート

このステップは、通常は実行する必要はありません。［インポート ウィザード］を使用すれば、必要なアドインがロードされます。

1. ［ツール］メニューから［カスタマイズ］ダイアログを選択します。
2. ダイアログの［アドイン］タブをクリックし、現在使用可能なアドイン モジュールの一覧を表示します。
3. 使用可能なアドインの一覧から［CppImport］アドインを選択し、ダイアログを閉じます。

TTDCppImport という名前の UML プロファイル パッケージがロードされ、［モデル ビュー］の［ライブラリ］フォルダに表示されます。このパッケージには、ステレオタイプ cppImportSpecification と C/C++ インポートの設定を持つ属性が含まれます。

次にこのステレオタイプをパッケージに適用し、ステレオタイプのタグ付き値を編集して適切なインポート オプションを設定します。［モデル ビュー］でパッケージを右クリックし、［C/C++ のインポート］コマンドを選択してインポート操作を実行します。

繰り返しインポートの考慮点

通常、C/C++ コードのインポートは一連の C/C++ ファイルに対して一度だけ行う操作です。しかし、同じファイルセットに対して、後でもう一度インポートを繰り返したい場合があります。たとえば、入力ファイルに変更を加えた場合、この変更を反映してモデルをアップデートするため、繰り返しインポートを行うことができます。

重要！

インポート済みパッケージのコンテキスト メニューから繰り返しインポートを行う場合、パッケージに新しいインポート エンティティが作成される前に、そのパッケージ内にあるすべてのインポート UML エンティティを先に削除する必要があります。UML モデルでインポート エンティティに追加された情報は、失われます。C/C++ コードの繰り返しインポートは、元に戻すことができないので注意が必要です。

GUID 名オプション

C/C++ インポートなどの翻訳の結果作成される UML エンティティは、デフォルトでランダムに生成された GUID を与えられます。

C/C++ 定義を繰り返しインポートすると、次にインポートする際エンティティへの参照（通常プレゼンテーション要素から）が未解決になることがあります。この問題に対処するため、GUID algorithm オプションを `Based on scoped name` に設定し、元の C/C++ 定義の名前から GUID が生成されるようにします。

重要！

スコープ名をベースとする GUID アルゴリズムを使用する場合は、インポート済みのヘッダー ファイルがセマンティック上正しいことが非常に重要です。ファイルが正しくない場合、たとえば、同名の 2 つの定義を持つ C++ スコープの場合、2 つの UML エンティティに同じ GUID が付けられてしまいます。このモデルを保存した場合、GUID が衝突しているので、再ロードできません。したがって、インポートを繰り返し行いたい場合は、インポート パッケージは必ず別々の .u2 ファイルに保存してください。GUID の衝突が生じた場合、プロジェクトからインポート パッケージの .u2 ファイルを削除して、モデルの残りの部分をロードできます。

「スコープ名」が特殊ルールに従って計算されている場合については、次で説明します。このルールの目的は、生成された GUID を一意に保つことです。このルールは、汎化、戻りパラメータ、仮パラメータ、操作に適用されます。

汎化

汎化では、GUID は「X-inherits-Y」になります。ここで、

- X はサブタイプの名前です（インポート先のパッケージの相対アドレス修飾）。
- Y はスーパータイプの名前です。

戻りパラメータ

戻りパラメータでは、GUID は「X-return-parameter」になります。ここで、X は戻りパラメータが属する操作の修飾されたシグニチャです。

仮パラメータ

仮パラメータでは、GUID は「X-Y」になります。

- X はパラメータが属する操作の修飾されたシグニチャです。
- Y は「`fparN`」です。ここで「N」はパラメータの序数詞です。

属性

属性では、GUID は 接頭辞「`attr--`」の付いた属性の修飾名になります。

操作

操作では、GUID は操作の修飾されたシグニチャになります。

一般翻訳ルール

外部

デフォルトでは、インポートされた定義は UML では「外部」とマークされます。これにより、これらの定義が UML モデルの外部にある C/C++ 定義に対応していることを示します。

cppImportSpecification ステレオタイプに "Set "External" attributes for imported definitions" オプションがあります。インポート済み定義を外部とマークしない場合はこのオプションをオフにします。旧来のコードを UML に移行する目的で C/C++ インポートを行う場合などは、このオプションを使用できます。このオプションをオフにすると、C/C++ 定義を明示的に `extern` と宣言した場合のみ外部プロパティが設定されます。ある種の定義（定数など）では、外部と外部以外の定義で UML から C++ への翻訳ルールが異なる場合があるので、このような手法がとられます。

次のセクションの翻訳ルールの例では、このオプションは設定されていることとしています。例では簡潔にするため、UML 定義は通常外部とマークされていません。

名前

C/C++ 識別子には、UML で同じ名前が与えられます。

C/C++ の名前が UML キーワードの場合、有効な UML 名を作成するため、アポストロフィで囲みます。

例 **145:** 名前の翻訳 ——————————————————————————

C/C++ :

```
void OpenFile();
double signal; // UML keyword "signal"
typedef int part; // UML keyword "part"
part port; // UML keywords "part" and "port"
```

UML :

```
public void OpenFile();
public double 'signal';
public syntype 'part' = int;
public 'part' 'port';
```

——————————————————————————————————————

基本型

C/C++ 基本型は同じ名前をもつ UML タイプにマッピングされます。

C/C++ のすべての基本型の UML 表現は、CppTypes アドインによってロードされる TTDCppPredefined プロファイル パッケージから使用できます。

C/C++ インポート時に、このパッケージが使用可能なことを前提とし、パッケージ内の型への参照が生成されます。インポートの実行時にパッケージが使用できない場合、翻訳の結果バインドされない多数の参照が発生する可能性があることを知らせる警告メッセージが表示されます。

C/C++ 基本型から UML への翻訳

次の表に、C++ 基本型と UML タイプ間のマッピングを示します。ほとんどの場合、TTDCppPredefined プロファイル パッケージ内の型は定義済み UML タイプのシンタイプです。

C/C++ 基本型	UML タイプ	定義済み UML タイプ
`signed int,` `int`	`int`	整数
`unsigned int,` `unsigned`	`'unsigned int'`	整数

`signed long int,`
`signed long,`
`long int,`
`long`
	`'long int'`	整数

`unsigned long int,` `unsigned long`	`'unsigned long int'`	整数

`signed short int,`
`signed short,`
`short int,`
`short`
	`'short int'`	整数

`unsigned short int,` `unsigned short`	`'unsigned short int'`	整数

`signed long long int,`
`signed long long,`
`long long int,`
`long long`
	`'long long int'`	整数

`unsigned long long int,` `unsigned long long`	`'unsigned long long int'`	整数
`char`	`char`	文字
`signed char`	`'signed char'`	文字
`unsigned char`	`'unsigned char'`	文字
`wchar_t`	`'wchar_t'`	文字
`float`	`float`	実数

C/C++ 基本型	UML タイプ	定義済み UML タイプ
`double`, `long double`	`double`	実数
`bool`	`bool`	ブール値
`void`	`N/A`	`N/A`

注記

特殊な `void` 型は UML で明示的に表現されません。代わりに、この型は操作への入出力の引数を省略して翻訳します。関数を参照してください。

ポインタ型、配列型、参照型

C のポインタ（`*`）および配列（`[ ]`）型の指定子および C++ の参照（`&`）型の指定子により、以下のものを作成できます。

- ポインタ型指定子,
- 配列型指定子,
- 参照型指定子,
- 型指定子を持たない型,

上記の組み合わせ

ポインタ型指定子

ポインタ型指定子は、TTDCppPredefined プロファイル パッケージの `CPtr` テンプレートのテンプレート インスタンス化に翻訳されます。`CPtr` テンプレートには、C/C++ ポインタで実行可能な操作に対応する操作があります。

例 **146:** ポインタの翻訳 ———————————————————————————

C/C++：

```
typedef int* p_int;
extern void* p_userdata;
typedef char** pp_char;
```

UML：

```
public syntype p_int = CPtr<int>;
public 'void*' extern p_userdata;
public syntype pp_char = CPtr<'char*'>;
```

———————————————————————————————————————

型のないポインタ

型のないポインタ（void*）は TTDCppPredefined プロファイル パッケージ内の特殊な UML タイプ `'void*'` に翻訳されます。以下のように定義されます。

```
syntype 'void*' = CPtr<Any>;
```

SetValue や GetValue のような CPtr<> の操作は 'void*' を付けて使用する必要があります。Any は何も意味しないので、Any タイプの値を直接定義できないためです。

'void*' タイプは C++ の void* と UML 値の間の変換バッファとしてのみ使用できます。UML では 'void*' を CPtr<> タイプおよび UML クラス参照に変換し、特殊なキャストなしに戻すことができます。'void*' と UML タイプの間でのその他の UML タイプの変換は、明示的な cast<> 操作の呼び出しによってのみ行うことができます。

注記

コードを生成する場合、必ず 'void*' 変換のための明示的なキャスト演算子を挿入してください。キャストを使用しなくても UML モデルのチェック時にエラー メッセージは表示されませんが、生成されたコードはコンパイルできません。

例 **147: C++ の void* 型の翻訳と使用**

C/C++ :

```
typedef void* voidstar;
typedef struct Str {
  voidstar  ptr;
} Str;
```

インポート後の UML :

```
public syntype voidstar = 'void*';
public <<struct>> class Str {
  public voidstar ptr;
}
```

UML での 'void*' の使用 :

```
part Str x;  CPtr<int> pi;
pi = new CPtr<int>();
pi.SetValue(10);
x.ptr = cast<'void*'>( pi );    // 'void*' = CPtr<int>
pi = cast<CPtr<int> >( x.ptr ); // CPtr<int> = 'void*'

class C {}
CPtr<C> pc;
C c = new C();
x.ptr = cast<voidstar>( c );    // 'void*' = class
reference
  pc = cast<CPtr<C> >( x.ptr );   // CPtr<C> = 'void*'
  c = pc;                         // class reference =
CPtr<C>
```

473 ページの例 147 に、UML での 'void*' 使用に関する以下のルールを示します。

- UML の 'void*' には、メモリの割り当ておよび割り当て解除関数はありません。UML で 'void*' を初期化する唯一の方法は、クラス（new C()）のオブジェクトを作成するか、CPtr<> タイプ（new CPtr<int>() ）を初期化して cast によって 'void*' に割り当てることです。
- UML の 'void*' 変数が値セマンティクス（int など）を持つ UML オブジェクトを指し示す必要がある場合、CPtr<type> を使用してポインタを作成する必要があります。
- テンプレートのインスタンス化である実テンプレート引数の UML 構文は cast<CPtr<int> > です。この構文では、最初の閉じかっこの次に空白を使用する必要があります。読みやすくするために、CPtr<> 型のシンタイプを定義できます。
- cast<CPtr<Any> > は使用**できません**。使用できるのは cast <'void*'> だけです。
- cast 操作は直接タイプ名（cast<'void*'> ）またはシンタイプとして定義されたいずれかのシノニム（cast<voidstar>）を使用できます。
- クラス参照に 'void*' をキャストしても生成されたコードでは機能しません。これは、C コード ジェネレータの制約です。そのような変換が必要な場合は、'void*' を明示的に CPtr<class> にキャストすると、CPtr<class> が暗黙的にクラス参照にキャストされます。

''void*' は UML の new 演算子を呼び出すことによって初期化できます。また、GetAddress 呼び出しによって初期化することもできます。

例 **148: UML** での **void*** の初期化 ——————————————————————

C/C++ :

```
typedef void* voidstar;
```

インポート後の UML :

```
public syntype voidstar = 'void*';
```

UML での void* の初期化 :

```
'void*' pv1, pv2;  CPtr<int> pi;  int i;
pi = new CPtr<int>();
pi.SetValue(10);
pv1 = cast<'void*'>( pi );       // 'void*' = CPtr<int>

i = 12;
pi = CPtr<int>::GetAddress( i );
pv2 = cast< voidstar >( pi );    // 'void*' = CPtr<int>
```

——————————————————————————————————————

UML の 'void*' 値は void* の形で C++ 関数に渡すことができます。C++ 関数によって戻された C++ の void* 値は UML の 'void*' 値に格納できます。

例 **149: UML** からの **C++** 関数の呼び出し ——————————————————————

C/C++ :

```
typedef void* MyVoidStar;
MyVoidStar init( void );
void finit( MyVoidStar );
```

インポート後の UML：

```
public syntype MyVoidStar = 'void*';
public MyVoidStar init();
public void finit( MyVoidStar );
```

UML からの C++ 関数の呼び出し：

```
CPtr<int> pi;
pi = cast< CPtr<int> > ( init() );
finit( cast<MyVoidStar>( pi ) );
```

---

char へのポインタ

char へのポインタ（`char*, wchar_t*`）は、TTDCppPredefined プロファイル パッケージから特殊な UML タイプ（`'char*'` と `'wchar_t*'`）にマッピングできます。これらのタイプは以下のように定義されます。

```
<<External="true">> datatype 'TCHAR*' :CPtr<TCHAR> {
    public 'TCHAR*'( Charstring str);
    public <<External="true">> Charstring ToString();
}
<<External="true">> syntype TCHAR = Character;
syntype 'char*' = 'TCHAR*';
syntype 'wchar_t*' = 'TCHAR*';
```

UML タイプ 'TCHAR*' は `CPtr` テンプレートに定義されたすべての操作を継承しつつ、2 つの新しい操作（UML タイプの `Charstring` からのコンストラクタと 'TCHAR*' から `Charstring` への変換操作）も追加されます。これらの操作は、UML 文字列と C++ char* 文字間の変換を目的としています。

cppImportSpecification ステレオタイプには "Import char* to CPtr<char>" オプションがあります。char へのポインタを一般的なポインタとして扱いたい場合は、このオプションをオンにできます。UML Charstrings から、または UML Charstrings への変換が必要ない場合は、このオプションを使用してもよいでしょう。

配列型指定子

配列型指定子は、TTDCppPredefined プロファイル パッケージの `CArray` テンプレートのテンプレート インスタンス化に翻訳されます。

UML の通常の多重度指定ではなく特殊な `CArray` テンプレートが使用される理由は、`"[]"` が C/C++ では型指定子であるのに対し、UML では構造的機能の指定子であるからです。多重度指定を使用した場合、476 ページの例 150 に示すように、翻訳時に構造的機能が存在するため `typedef` を翻訳することはできません。

例 **150:** 配列の翻訳

C/C++ :

```
extern char c_arr1[20];
typedef int array_of_ints[1024];
extern char c_arr2[];
```

UML :

```
public CArray<char, 20> extern c_arr1;
public syntype array_of_ints = CArray<int, 1024>;
public ‘char*’ extern c_arr2;
```

無制限配列

配列のサイズを指定しない配列指定子（無制限配列ともいいます）はポインタ指定子と同様に翻訳されます。例については、476 ページの例 150 を参照してください。

参照

ポインタ型指定子

参照型指定子

参照型指定子は UML では暗黙的なので、翻訳する必要ありません。UML タイプのデフォルトは参照セマンティックです。

例 **151:** 参照の翻訳

C/C++ :

```
extern int i;
/* i is initialized elsewhere */
extern int& r;
/* r is initialized to i, elsewhere (int& r = i;) */
```

UML :

```
public int extern i;
public int extern r;
```

参照は仮関数引数の指定子として、また戻り型として使用されることもあります。このような参照型指定子の翻訳については、481 ページの「引数と戻り型」を参照してください。

戻り型の関数に使用された参照は、対応する操作の戻りパラメータに適用された `<<CppReference>>` ステレオタイプを得ます。例については、483 ページの例 160 を参照してください。

型指定子を持たない型

- 定義済み C/C++ 型以外の型に型指定子がまったくない場合、UML タイプに対応するタイプが付けられた UML 構造的機能が組み込まれます。
- 構造的機能ではないエンティティの型付けに UML タイプが使用された場合（シンタイプなど）、型指定子がなくても特殊な翻訳は行われません。
- シンタイプの場合、それによってタイプ付けられた構造的機能の型が組み込まれ、「シンタイプ チェーン」全体に繰り返しルールが適用されます。このルールの理由は、UML のデフォルトが参照セマンティックだからです。

例 **152:** 型指示を持たない型の翻訳 ——————————————————————

C/C++ :

```
class MyClass {};
MyClass value_var; // Value semantics
extern MyClass& ref_var; // Reference semantics
```

UML :

```
public class MyClass {};
public part MyClass value_var;
public MyClass extern ref_var;
```

——————————————————————————————————————

定義済みの型

定義済みの型の場合、デフォルトは値セマンティックです（C/C++ と同様）。この場合、型付けられた構造的機能の型は通常どおり翻訳されます。

列挙型

列挙型は、`enum` リテラルに対応するリテラルを持つデータ型に翻訳されます。

列挙型にリテラルがない場合、C/C++ では整数型として扱われ、`int` のシンタイプに翻訳されます。

例 **153:** 列挙型の翻訳 ——————————————————————

C/C++ :

```
enum {} v;
enum E2 {}; // Enum without literals
enum E3 {a, b = 10, c = b + 5};
```

UML :

```
public syntype incomplete_v = int;
public incomplete_v v;
public syntype E2 = int;
public enum E3 { a, b = 10, c = [[b + 5]]}
```

C/C++ リテラルに指定される定数式の翻訳の詳細については、定数式を参照してください。

参照

int から enum への暗黙の変換

Typedef

`typedef` は UML シンタイプに翻訳されます。

例 **154: C** の型定義の翻訳 ——————————————————————————

C/C++ :

```
typedef int MyInt;
typedef struct r {
  int a;
} r; // Typedef name is the same as the tag name!
typedef struct q {
  bool m_bShall;
} *q; // Typedef name is the same as the tag name,
      // and there is a type specifier.
typedef struct s {
  MyInt a;
}; // Omitted typedef name - legal but rare!
typedef void myvoid;
typedef myvoid myvoid2;
myvoid f(myvoid2);
```

UML :

```
public syntype MyInt = int;
public <<struct>> class r {
  public int a;
}
public <<struct>> class q {
  public bool m_bShall;
}
public <<struct>> class s {
  public MyInt a;
}
public void f();
```

上記「q」の翻訳では、インポート中に「Conflicting typedef name」という警告が生成され、typedef は翻訳されません。詳細については、タグ付き型の Typedef 宣言を参照してください。

また、void の typedef も翻訳されませんが、そのような typedef 名を使用している場合は void が使用されているものとして翻訳されます。詳細については、void 型を持つ Typedef を参照してください。

タグ付き型の **Typedef** 宣言

タグ付き型の `typedef` 宣言について（`typedef` 名はタグと同じ名前、`typedef` 型は定義なしの前方宣言）、以下のルールが適用されます。

`typedef` のような `typedef` 型に型指定子がある場合、それは新しい型名を定義するものではなく、シンタイプも生成されません。シンタイプを生成しない理由は、ポインタ型、配列型、参照型の通常のルールに従って翻訳されると命名の競合が生じるからです。

名前が省略された **Typedef**

`typedef` 名が省略された `typedef` は、新しい型名を定義するものではなく、シンタイプも生成されません。

void 型を持つ **Typedef**

`typedef` の型が void である `typedef`、または void の `typedef` は、新しい型名を定義するものではなく、シンタイプも生成されませんが、`typedef` 名は記憶されます。

`typedef` 名への参照は、void が翻訳された場合と同様に翻訳されます。

関数

関数プロトタイプ

関数プロトタイプは UML 操作に翻訳されます。このルールはメンバー関数および非メンバー関数の両方に有効です。

例 **155:** 関数翻訳の一般例

C/C++ :

```
char myfunc1(char);
int myfunc2(void);
void myfunc3();
void myfunc4(int);
bool myfunc5(void* p1, double p2);
```

UML :

```
public char myfunc1(char);
public int myfunc2();
public void myfunc3();
public void myfunc4(int);
public bool myfunc5 ('void*' p1, double p2);
```

非メンバー関数

メンバー関数はクラスまたは選択の操作として入れられますが、非メンバー関数は、インポート先のパッケージ内の操作に翻訳されます。

メンバー関数

翻訳ルールは非メンバー関数と同様ですが、C++ のメンバー関数は、宣言内で「機能がより充実」している可能性があります。関数のメンバー固有の機能に対する翻訳ルールは、クラス、構造体、共用体の翻訳について説明するセクションのサブセクションで説明しています。

仮引数

関数の仮引数は、対応する UML 操作の仮パラメータに翻訳されます。ただし、void 型の場合は翻訳されません。

戻り型

戻り型の関数は、UML 操作の戻り型パラメータに翻訳されます。

プロトタイプのない関数宣言

プロトタイプのない関数宣言（旧式）は、関数の引数が正しく宣言されていれば、プロトタイプのある通常の関数と同様に翻訳されます。

プロトタイプのない関数宣言は、すべての関数引数の宣言がある場合のみ、C/C++ インポートでサポートされます。

例 **156:** プロトタイプのない関数の翻訳 ——————————————————————

C/C++ :

```
float average(x,y,z)
float x,y;
char z;
{
  return 1.1;
}
```

UML :

```
public float average(float x, float y, char z);
```

——————————————————————————————————————

オーバーロード関数

一連のオーバーロード関数は、一連のオーバーロード UML 操作に翻訳されます。

例 **157:** オーバーロード関数の翻訳 ——————————————————————

C/C++ :

```
int f0();
int f0(double);
int f1(int);
int f1(const int);
```

UML：

```
public int f0();
public int f0(double);
public int f1(int);
```

---

const のオーバーロードは UML ではサポートされませんが、C++ では可能です。理由は、const は C++ の型の一部で、const 型は const を持たない同じ型と区別可能だからです。UML では、const はタイプのプロパティではないので、オーバーロード操作の解決呼び出し時に、名前解決によって考慮されません。

メンバー関数に const を指定子として使用することによってオーバーロードが生じた場合、すべてのオーバーロード関数が UML にインポートされないことがあります。以下の例を参照してください。

例 **158: const** のオーバーロード ——————————————————————

以下の C++ 宣言は UML に正しくインポートされません。

C/C++：

```
class X {
   int func( int x );
   int func( int x ) const;
};
```

出力：

```
Ignored conflicting declaration 'func'
```

UML：

```
public class X {
  private <<IsQuery = "true">> int func( int x );
}
```

参照

オーバーロード関数間の曖昧性

引数と戻り型

デフォルトで、UML では関数引数は参照によって渡されます。例外は、値セマンティックに従ったデータ型です。C++ 関数引数はデフォルトで値セマンティックを持ちます。

これらの違いがあるため、以下の表のルールが適用されます。例では、D という名前の付いたタイプを C++ のタイプが UML データ型に翻訳されます（シンプルタイプ、ポインタ、配列、列挙など）。C という名前のタイプはその他の C++ タイプを表します。

ルール／例	C++	UML
1) タイプ D の C++ 引数は、方向またはパート指定子のない UML 引数に翻訳される。	`void F(D);`	`void F(D);`
2) タイプ C の C++ 引数は、UML の part 引数に翻訳される。	`void F(C);`	`void F (part C);`
3) 参照（&）によって渡されるタイプ D の C++ 引数は、UML の inout 引数に翻訳される。	`void F(D&);`	`void F (inout D);`
4) 参照（&）によって渡されるタイプ C の C++ 引数は、inout 方向のない UML の part 引数に翻訳される。	`void F(C&);`	`void F (inout part C);`
5) 参照（&）によって渡されるタイプ D への C++ ポインタは、タイプ CPtr <D> の UML の inout 引数に翻訳される。	`void F(D*&);`	`void F (inout CPtr<D>);`
6) 参照（&）によって渡されるタイプ C の C++ 引数は、UML の inout 引数に翻訳される。	`void F(C*&);`	`void F(inout C);`

C++ 参照引数のインポートは、以下の UML から C++ への翻訳ルールに従います。

- UML の参照は C++ ポインタに翻訳される。
- inout パラメータは C++ の参照パラメータに翻訳される。
- part 引数は通常の C++ 引数に翻訳される（part は無視され、タイプの名前のみ表示される）。
- datatype は UML の参照ではないので、part 引数と同様にマッピングされる。

詳細については、C++ アプリケーション ジェネレータの型付けされた定義のタイプを参照してください。

例 **159:** 仮引数の翻訳 ——————————————————————————————

C/C++ :

```
class C {
    int x;
};
typedef C* pC;

void F1( int x );
void F2( int& x );
void F3( int*& x );
```

```
void F4( int**& x );

void F5( C x );
void F6( C& x );
void F7( C*& x );

void F8( pC x );
void F9( pC& x );
void F10( pC*& x );
```

UML :

```
public class C {
  private int x;
}
syntype pC = CPtr<C>;

void F1( int x );
void F2( inout int x );
void F3( inout CPtr<int> x );
void F4( inout CPtr<CPtr<int> > x );

void F5( part C x );
void F6( inout part C x );
void F7( inout C x );

void F8( pC x );
void F9( inout pC x );
void F10( inout CPtr<pC> x );
```

定数引数は、読み取り専用の UML の操作パラメータに翻訳されます。

例 **160:** 関数引数と戻り値の翻訳 ——————————————————————

C/C++ :

```
int f1 (int p1, int& p2, const int& p3, const int* p4, int
*const p5);
int& f2();
const int& f3();
class MyClass {};
void f4(MyClass& p1, MyClass p3, MyClass* p2);
```

UML :

```
public int f1(int p1, inout int p2, const inout int p3, const
CPtr<int> p4, CPtr<int> p5);
public (<<CppReference>> int) f2();
public (<<CppReference>> int) f3();
public class MyClass {}
public void f4 (inout part MyClass p1, part MyClass p3,
MyClass p2);
```

`const char*` のように、定数型へのポインタを返す C++ 関数は、UML では const 指定子を持たないタイプを返す操作に翻訳されます。これは、UML の段階では問題ありませんが、この関数を使用する C コードまたは C++ コードを生成する場合に問題となる可能性があります。詳細については、定数へのポインタを参照してください。

デフォルト引数

デフォルト値が指定された関数引数は、デフォルト値を持つ UML の操作パラメータに翻訳されます。デフォルト値が定数式の場合、定数式で説明されているように翻訳されます。

例 **161:** デフォルト引数を持つ関数の翻訳

C/C++ :

```
int func(int a, int b = 5, int c = 7);
int func(int a); // Ambiguous function!
```

UML :

```
public int func(int a, int b = 5, int c = 7);
public int func(int a);
```

実引数を 1 つだけ持つ `func` の呼び出しは、C++ と UML のどちらでも不明瞭になります。詳細については、オーバーロード関数間の曖昧性を参照してください。

オーバーロード関数間の曖昧性

C++ では、呼び出しを未解決にするような実引数を持つ関数を呼び出すことがなければ、オーバーロード関数間の曖昧性が許されます。

UML でも同様です。したがって、484 ページの例 161 の第 2 バージョンの `func` は、UML の定義どおりに許可されますが、呼び出しはできません。

unspecified 引数

unspecified 引数を持つ関数（ellipsis 関数ともいう）はサポートされません。C/C++ インポート時にそのような関数があった場合、ellipsis 引数（...）は無視されます。

これを回避するため、UML にインポートされる C/C++ ヘッダーに一連のラッパー関数を定義します。ラッパー関数によって、UML モデルで使用されるべき ellipsis 関数のバージョンを指定します。

例 **162: ellipsis** 関数の翻訳

C/C++ :

```
int printf(const char *, ...);

/* Wrapper functions */
int printf_int(const char* s, int a) { return printf(s, a); }
```

```
int printf_str(const char* s, const char* a) { return
printf(s, a); }
```

UML :

```
public int printf(const ‘char*’);
public int printf_int(const ‘char*’ s, int a);
public int printf_str(const ‘char*’ s, const ‘char*’ a);
```

インライン関数

inline と宣言された関数は、通常の関数として翻訳されます。

関数のインライン キーワードは C++ コンパイラのディレクティブと見なされ、その関数の呼び出し方法にのみ影響を与えます。

注記

TTDCppPredefined プロファイルからインライン ステレオタイプを使用すると、「インライン」プロパティが維持されます。

例 **163:** インライン関数の翻訳

C/C++ :

```
inline int fac(int n){return (n == 1) ?1 : n * fac(n - 1);};
```

UML :

```
public <<inline>> int fac ( int n);
```

関数ポインタ

関数ポインタは、<<operationReference>> ステレオタイプで、UML インターフェイスにマッピングされます。インターフェイスの名前は型定義の名前になります。関数ポインタ型を（typedef を持たない）インライン コードで使用する場合、インターフェイスの名前は「fpointer_<formal parameter type names>_returns_<return parameter type name>」の形式になります。インポート後のインターフェイスに含まれる操作は「call」のみです。

関数ポインタを使用する関数の呼び出しは、UML では以下のように行われます。

```
i = myOp.call( 10 );
```

ここで、myOp は生成された関数ポインタ インターフェイスによってタイプ指定された属性です。

例 **164:** 関数ポインタの翻訳

C/C++ :

```
typedef int (*PFunc) ( int );
void Func ( bool (*pointer) ( bool ) );
```

```
int (* qq) ( int );
int (* Func1( bool ) ) ( int );
```

UML：

```
public <<operationReference>> interface PFunc {
  int call( int);
}
public void Func( fpointer_bool_returns_bool pointer );
<<operationReference>> interface fpointer_bool_returns_bool {
  bool call( bool);
}
public fpointer_int_returns_int extern qq;
public fpointer_int_returns_int Func1( bool );
<<operationReference>> interface fpointer_int_returns_int {
  int call( int);
}
```

関数型

C/C++ で関数ポインタを指定する別の方法は、関数の型宣言を利用することです。ポインタ宣言子はある特定の関数型を使用して追加できます。

UML の関数型はサポートされません。また、関数型宣言に出合うと C/C++ インポータは警告メッセージを発します。警告は発せられますが、インポータはポインタ宣言子とともに使用される関数型を、通常のポインタ宣言子と同様に、正しく取り扱います。

例 **165:** 関数型を使用した関数ポインタの翻訳

C/C++:

```
typedef void (foo)(int);
foo* pfn;
foo fn; // Yields warning during import
typedef foo foo2;
foo2 fn2; // Yields warning during import
```

UML:

```
public <<operationReference>> interface foo {
  void call(int);
}
public foo extern pfn;
public syntype foo2 = foo;
```

この例で明らかなように、関数型で型付けされた変数 (`fn` と `fn2`) は、UML には翻訳されません。ただし、関数型がポインタ宣言子 (`pfn`) とともに使用されている場合は、通常の関数ポインタの規則に則って翻訳されます。

関数の本体

C/C++ インポートでは、デフォルトでは、インポート済みヘッダー ファイルにある関数の本体（文）をインポートしません。しかし、cppImportSpecification ステレオタイプに "Action code strategy" オプションがあります。関数の本体を UML に翻訳するため、このオプションをオンにできます。旧来のコードを UML に移行する目的で C/C++ インポートを行う場合などは、このオプションを使用できます。

C/C++ 関数の本体は、UML 操作の操作本体に翻訳されます。これは、関数の翻訳です。操作本体には、関数本体の C/C++ 文に対応するアクションのリストが含まれます。文ごとの翻訳ルールを以下に示します。

また、“Action code strategy” オプションをオンにして、非形式な UML を使用して関数本体をインポートすることもできます。この場合、関数の本体全体が非形式な UML アクションにコピーされます。

例 **166:** 関数本体の翻訳 ——————————————————————————

C/C++ :

```
int m() {
  int a = 3;
  return a;
}
```

UML（“Action code strategy” オプションをオンにして、UML アクションを使用してインポート）：

```
int m() {
  int a = 3;
  return a;
}
```

UML（“Action code strategy” オプションをオンにして、非形式な UML を使用してインポート）：

```
int m() {
[[
  int a = 3;
  return a;
]]
}
```

——————————————————————————————————————

ラベル付き文

ラベル付き文は、対応するラベルの付いた UML アクションに翻訳されます。UML は最大 1 つのラベルを持つことができます。つまり、文に複数のラベルが付けられている場合、C/C++ インポータはインポート結果のアクションの前に空のアクションを挿入して、翻訳中にすべてのラベルを保持します。

例 **167:** ラベル付き文の翻訳

C/C++ :

```
A:foo();
B:
EXIT:return;
```

UML :

```
A:foo();
B: ;
EXIT:return;
```

宣言文

宣言文は UML の定義アクションに翻訳されます。C++ と同様に、UML 定義もアクション リスト内の任意の場所で宣言できます。

C/C++ アクションごとの翻訳ルールは、それぞれの章に示します。例については、列挙型、Typedef、関数、変数、定数、クラス、構造体、共用体を参照してください。

例 **168:** 宣言文の翻訳

C/C++ :

```
typedef unsigned int UINT;
UINT a = 5;
```

UML: :

```
syntype UINT = 'unsigned int';
UINT a = 5;
```

式文

式文は UML の式アクションに翻訳されます。式文内の式の翻訳の詳細については、式を参照してください。

式文の式が対応する UML 式で示せない場合、式アクションは非形式な UML アクション（[[...]];）に翻訳されます。ここに、元の C/C++ 式が翻訳されずに保持されます。

式文の式はオプションです。式を省略すると、式文は「空の」UML アクション（セミコロン 1 つ）に翻訳されます。

例 **169:** 式文の翻訳

C/C++ :

```
int a = 5 + 6 - foo();
bool b = true && !false;
a--;
a = sizeof(int);
```

```
a += 5;
;
```

UML :

```
int a = 5 + 6 - foo();
bool b = true && ! false;
a --;
a = [[sizeof(int)]];
[[a += 5]];
;
```

break 文

break 文は UML の break アクションに翻訳されます。例については、489 ページの例 170 を参照してください。

continue 文

continue 文は UML の continue アクションに翻訳されます。例については、489 ページの例 170 を参照してください。

for 文

for 文は、UML の for アクションに翻訳されます。これは、特殊なループ アクションです。

例 **170: for** 文の翻訳

C/C++ :

```
for(int i = 0; i < 10; i++) {
  if (i == 5)
    break;
  if (i == 4)
    continue;
}
```

UML :

```
for ( int i = 0; i < 10; i ++) {
    if (i == 5)
        break;
    if (i == 4)
        continue;
}
```

do 文

do 文は、UML の do アクションに翻訳されます。これは、特殊なループ アクションです。

例 **171: do** 文の翻訳

C/C++ :

```
do {
  compute(x);
} while (x >= 0);
```

UML :

```
do
{
    compute(x);
}
while (x >= 0);
```

goto 文

go to 文は UML の join アクションに翻訳されます。

例 **172: goto** 文の翻訳

C/C++ :

```
if (!valid())
     goto ERROR;

  return 0;

  ERROR:return -1;
```

UML :

```
if (! valid())
    goto ERROR;
return 0;
ERROR :return - 1;
```

ラベル付き文については、ラベル付き文を参照してください。

if 文

if 文は UML の if アクションに翻訳されます。また、else ブランチは UML の対応する else ブランチに直接翻訳されます。

例 **173: if** 文の翻訳

C/C++ :

```
x = getX();
if (x == y)
  equal();
else if (x < y)
  less();
```

```
else
{
  greater();
}
```

UML :

```
x = getX();
if (x == y)
    equal();
else
    if (x < y)
        less();
    else
        {
            greater();
        }
```

switch 文

switch 文は UML の分岐アクションに翻訳されます。また、Case ブランチとオブションのデフォルト ブランチは UML の対応するブランチに翻訳されます。

例 **174: switch** 文の翻訳

C/C++ :

```
switch (e) {
case 1 :
{
  i = 1;
  break;
}
case 2  :
  break;
 default :{ i = 3; };
}
```

UML :

```
switch (e) {
    case 1 :
        {
            i = 1;
            break;
        }
    case 2 :
        break;
    default :
        {
            i = 3;
        }
}
```

while 文

while 文は、UML の while アクションに翻訳されます。これは、特殊なループ アクションです。

例 **175: while** 文の翻訳 ——————————————————————————

C/C++ :

```
while (true) {
  if (done())
    break;
}
```

UML:

```
while (true) {
    if (done())
        break;
}
```

——————————————————————————————————————

return 文

return 文は UML の return アクションに翻訳されます。例については、490 ページの 例 172 を参照してください。

try 文

try 文は UML の try アクションに翻訳されます。また、catch 節は UML の対応する catch 節に変換されます。

catch 節内の例外の「再スロー」は、UML 内でキャッチされた例外パラメータを参照して行う必要があります。したがって、C++ の `catch(...)` 節は、`Any` によって型指定された名前付き例外を持つ UML の catch 節に翻訳されます。例外パラメータのデフォルト名は `Exception` です。catch 節に同じ名前を持つローカル定義がある場合、または同じ名前を持つローカル以外の定義への参照がある場合、catch 節のスコープ内で例外パラメータの名前を一意に保つため、接尾辞が付けられます。

例 **176: try** 文の翻訳 ——————————————————————————

C/C++ :

```
try {
  test();
}
catch (char* error_msg)
{
  printf("%s", error_msg);
}
catch (...)
{
  throw;
}
```

```
try {
  test2();
}
catch (...) {
  bool Exception = true;
  throw;
}
```

UML :

```
try {
  test();
}
catch ('char*' error_msg)
{
  printf([["%s"]], error_msg);
}
catch (Any Exception)
{
  throw Exception;
}

try {
  test2();
}
catch(Any Exception1)
{
  bool Exception = true;
  throw Exception1;
}
```

複合文

複合文は UML の複合アクションに翻訳されます。

例 **177:** 複合文の翻訳

C/C++ :

```
int a = 5;
{
  int a = 6;
  a++;
}
a--;
```

UML :

```
int a = 5;
{
    int a = 6;
    a ++;
}
a --;
```

スコープ ユニット

- 名前空間
- クラス、構造体、共用体
- クラス テンプレート

名前空間

名前空間は UML のパッケージに翻訳されます。

注記

グローバル宣言は、インポート先のパッケージに入れられます。このパッケージには、TTDCppPredefined プロファイルの <<globalNamespace>> ステレオタイプが適用されます。

例 **178:** グローバル名前空間の定義の翻訳

C/C++：

```
int i;
void op(unsigned int);
```

UML （インポート パッケージ内）：

```
public int i;
public void op('unsigned int');
```

クラス、構造体、共用体

メイン トピッククラス、構造体、共用体を参照してください。

例 **179:** ネストされたスコープ ユニットの翻訳

C/C++：

```
class C {
public:
  int ci;
  class CC {
  public:
    int op();
  };
};
```

UML：

```
public class C {
  public int ci;
  public class CC {
    public int op();
  }
}
```

クラス テンプレート

C++ のクラス テンプレートは UML のクラス テンプレートにマッピングされます。

例 **180:** テンプレートがネストされた定義の翻訳 ——————————————————

C/C++ :

```
template <class C> class String {
  struct Srep {
    C* s;
    int sz;
    int n;
  };
  Srep *rep;
public:
  String();
  String(const C*);
  String(const String&);
  C read(int i) const;
};
```

UML :

```
template <type C > public class String {
  private <<struct>> class Srep {
    public C s;
    public int sz;
    public int n;
  }
  private Srep rep;
  public String();
  public String( const C );
  public String( const inout part String<C> );
  public <<IsQuery="true">> part C read( int i );
}
```

変数

変数は UML の属性に翻訳されます。このルールはメンバー変数および非メンバー変数の両方に有効です。

例 **181:** 変数の翻訳 ——————————————————

C/C++ :

```
int ivar, jvar;
class X {
  int j;
public:
  int Get() { return j;};
} xvar;
```

UML :

```
public int ivar;
public int jvar;
public class X {
    private int j;
    public int Get ();
}
public part X xvar;
```

非メンバー変数

非メンバー変数は、属性に翻訳されてインポート先のパッケージに入れられます。

メンバー変数

メンバー変数は `class` または `choice` の属性となります。

翻訳ルールは非メンバー変数の場合と同様ですが、C++ のメンバー変数は、宣言内で「機能がより充実」している可能性があります。変数のメンバー固有の機能に対する翻訳ルールについては、499 ページの「クラス、構造体、共用体」を参照してください。

定数

定数は、読み取り専用の UML の属性に翻訳されます。このルールはメンバー定数および非メンバー定数の両方に有効です。

メンバー定数は `class` または `choice` の属性として入れられますが、非メンバー定数は、インポート先のパッケージ内の属性に変換されます。

翻訳ルールは同じですが、C++ のメンバー定数は、宣言内で「機能がより充実」している可能性があります。定数のメンバー固有の機能に対する翻訳ルールについては、506 ページの「メンバー定数」を参照してください。

定数に式が指定されている場合、定数式に記載されるルールに従って式を評価した結果の値が、対応する UML 属性にデフォルトで含まれます。

例 **182:** 定数の翻訳

C/C++ :

```
class MyClass;
const double pi = 3.14159;
const MyClass m(7, 'x');
extern const int extconst; // Defined elsewhere.
```

UML :

```
public class MyClass {}
public const double pi = 3.14159;
public const part MyClass m with (7, ‘x’);
public const int extern extconst;
```

プリプロセッサ マクロとしての定数

Preprocessor（プリプロセッサ）マクロが定数を表現することは、一般的ではありません。特に旧式の C アプリケーションではあまり使われません。そのような定数は翻訳開始前にプリプロセッサよって削除されるため、UML には翻訳**されません**。そのような定数に UML モデルからアクセスするには、インライン ターゲット コード（[[...]]）を使用する必要があります。

参照

マクロ

式

C++ には、定数式と非定数式の 2 種類の式があります。宣言のみをインポートする場合（関数本体なし）、定数式のみが翻訳されます。旧来のコードを UML に移行する目的で C/C++ インポートを行う場合は、非定数式があればそれも翻訳されます。

C/C++ の式から UML の式への変換に使用される翻訳ルールは、C++ アプリケーション ジェネレータを使用して UML 式から C++ コードを生成するルールの逆となります。詳細については、第 41 章「C++ アプリケーション ジェネレータリファレンス」の 1340 ページ、「式（Expression）」を参照してください。

対応する UML 表現のない C/C++ 式は、非形式な式（[[...]]）に翻訳されます。

二項式と単項式

C/C++ の二項式は 2 つのオペランドを取る演算子を参照しますが、単項式は 1 つのオペランドを取る演算子を参照します。C/C++ 言語の演算子には、UML に対応する表現がないものがあります。そのような演算子を参照する二項式および単項式は、UML の非形式な式（[[...]]）に翻訳されます。下記の表は、対応する UML 表現のある演算子の一覧です。したがって、これらの演算子を参照する二項式および単項式は、UML に翻訳されます。

C/C++ 演算子		**種類**
`!`	`Logical not`	単項
`!=`	`Inequality`	二項
`&&`	`Logical and`	二項
`*`	`Multiplicity`	二項
`+`	`Addition`	二項
`+`	`Plus`	単項
`-`	`Subtraction`	二項
`-`	`Negation`	単項

C/C++ 演算子		種類
/	Division	二項
<	Less than	二項
<=	Less than or equal to	二項
=	Assignment	二項
==	Equality	二項
>	Greater than	二項
>=	Greater than or equal to	二項
[]	Array subscript	二項
<<	Left shift	二項
>>	Right shift	二項
||	Logical or	二項
++	Prefix increment	単項
++	Postfix increment	単項
--	Prefix decrement	単項
--	Postfix decrement	単項
%	Modulus	二項
new	New expression	単項

定数式

定数式は C/C++ ヘッダーに多く見られます。変数の定数初期化や、配列型指定子のビットフィールドのサイズ指定などに使用されます。

定数式は、UML への翻訳時に評価されることもあります。結果に UML レベルでもセマンティックな影響がある場合、たとえば配列のサイズを示す定数式などの場合、式が評価されます。

定数式の評価時、以下の 2 種類の式は完全にサポートされません。これらの式は、コンパイル環境がわからないと正しく評価できないためです。

- **`cast` 式**

  `cast` 式は評価されますが、値の修正（タイプの修正などのため）は行われません。`cast` 式が検出されると、キャスティングを無視することを知らせる警告が表示されます。
- **`sizeof` 式**

  `sizeof` 式の評価は、C/C++ コードをコンパイルするプラットフォームに依存するため、C/C++ インポート時に処理されません。`sizeof` 式が検出されると、この式が非形式的表記と見なされることを知らせる警告が表示されます。

例 **183:** 定数式の翻訳

C/C++ :

```
enum e { a, b, c = 10 };
const int i = (2+c)*b;
struct s{
  int f1 :(2+c)*b; // Evaluating
};
const int uncmn = (int) (bool) 4;
typedef int intarr[sizeof(int)+1];
// Assume that sizeof(int)+1 is an informal expression
```

UML :

```
public enum e {
  a,
  b,
  c = 10
}
const int i = [[(2 + c) * b]];
public <<struct>> class s {
  public <<bitfield(.'size' = 12.)>> int extern f1;
}
const int uncmn = [[(int) (bool) 4]];
public syntype intarr = CArray<int, ([[sizeof(int)+1]])>;
```

メッセージ :

```
Unable to evaluate sizeof expression. It will be imported as
an informal expression.

Can not translate C++ expression, importing it to informal
expression [[ ... ]]
```

参照

式の評価

クラス、構造体、共用体

- クラスまたは構造体は UML のクラスに翻訳されます。
- 共用体は UML の choice に翻訳されます。
- 構造体には <<`struct`>> ステレオタイプが付きます。

例 **184:** クラス、構造体、共用体の翻訳

C/C++ :

```
class C {};
struct S {};
union U {};
```

UML :

```
public class C {}
public <<struct>> class S {}
```

```
public choice U {}
```

タグなしクラス、構造体、共用体

クラス、構造体、共用体がタグを持たない場合、不完全型宣言です。

参照

不完全型宣言

無名共用体

無名共用体が翻訳されるとそのメンバーは UML の class または choice の属性となります。つまり、内包される C/C++ のスコープ ユニットが翻訳されるということです。この翻訳ルールの理由は、通常の共同体とは逆に、C/C++ では無名共用体はスコープ ユニットではないからです。

例 **185:** 無名共用体の翻訳

C/C++ :

```
struct S {
  int i;
  union {
    int j;
    int k;
  };
};
```

UML :

```
public <<struct>> class S {
  public int i;
  public int j;
  public int k;
}
```

注記

無名共用体は、構文は似ていますが、不完全な型宣言です。無名共用体は型または変数を宣言するために使用されることはありません。また、型を定義することもありません。したがって、無名共同体と不完全な型の翻訳ルールは大きく異なります。

参照

不完全型宣言

コンストラクタ

クラスのコンストラクタは UML のコンストラクタに翻訳されます。つまり、対応する UML クラスの操作はクラスと同じ名前を持ちます。

C++ には 2 種類のコンストラクタがあります。ユーザー定義コンストラクタは、ユーザーが明示的に宣言して実装します。暗黙のコンストラクタは、C++ コンパイラによって暗黙的に宣言され、自動生成されます（ユーザーが明示的に宣言していない場合）。

クラスには任意数のユーザー定義コンストラクタを含むことができますが、自動生成されたコンストラクタは 2 つまで、パラメータなし（デフォルト）コンストラクタとコピー コンストラクタは 1 つずつ含むことができます。パラメータなしコンストラクタはクラスにユーザー定義コンストラクタが 1 つもない場合のみ含むことができます。コピー コンストラクタはユーザー定義コピー コンストラクタが 1 つも宣言されていない場合のみ含むことができます。

UML には自動生成コンストラクタはありません。デフォルトおよびコピー コンストラクタはインポート後の C++ クラスに明示的に挿入されません。理由は以下のとおりです。

- UML では、すべてのクラスにデータ型 Any の暗黙の変換があり、このタイプには割り当て演算子が定義されているので、オブジェクト割り当て（c1 = c2）が機能する。
- 自動生成された C++ コンストラクタは生成されたコードに明示的に追加される。
- C++ アプリケーション ジェネレータが生成した C++ コードには自動生成コンストラクタが存在する。

例 **186:** コンストラクタの翻訳

以下の例では、3 つのユーザー定義コンストラクタと 1 つの暗黙コピー コンストラクタを持つ C++ クラスをインポートします。

C/C++ :

```
class C {
public:
  C();
  C(int i);
  C(char c);
};
```

UML :

```
public class C {
  public C();
  public C(int i);
  public C(char c);
}
```

関数と同様コンストラクタもオーバーロードできます。そして、オーバーロード関数のルールもコンストラクタに適用されます。

参照

オーバーロード関数

デストラクタ

クラスのデストラクタは UML のデストラクタに翻訳されます。つまり、対応する UML クラスの操作は、接頭辞としてチルダ（~）が付いたクラスと同じ名前を持ちます。

例 **187:** デストラクタの翻訳

C/C++：

```
class D {
public:
  ~D();
};
```

UML：

```
public class D {
  public ~D();
}
```

メンバー

C++ クラスのメンバー変数は UML クラスの属性に翻訳されます。つまり、そのクラスの翻訳です。メンバー関数は同じクラスの操作に翻訳されます。

クラス内の変数と関数以外の宣言、たとえば型宣言などは、クラスのメンバーと呼ばれることがあります。ただし、上記の翻訳ルールによるとこれらが翻訳されることはありません。これらの宣言は自身の宣言ですが、内包するスコープ ユニット（つまりクラス）に定義されていると見なされます。

例 **188:** クラス メンバーの翻訳

C/C++：

```
class C {
  public:
  int mv1; // Member variable
  void mf1(long long p1); // Member function
  enum e {a,b,c}; // "Member" type declaration
};
```

UML：

```
public class C {
  public int mv1;
  public void mf1 ('long long int' p1);
  public enum e {a, b, c}
}
```

メンバーのアクセス指定子

`public`（`private`、`protected`）アクセス指定子を持つメンバーは、それぞれ `public`（`private`、`protected`）可視形式を持つメンバーに翻訳されます。

デフォルトの振る舞い（可視性が省略された場合）は、以下の C++ ルールに対応します。

- 構造体と共用体の場合、可視性が省略されるとメンバーは `public` としてインポートされる。
- クラス メンバーは、可視性が省略されると `private` としてインポートされる。

例 **189:** 異なるアクセス指定子を持つメンバーの翻訳

C/C++ :

```
class C {
private:
  int i;
protected:
  int j;
public:
  int k;
  int GetI();
  int GetJ();
  int Calc (int x, int y);
};
```

UML :

```
public class C {
  private int i;
  protected int j;
  public int k;
  public int GetI();
  public int GetJ();
  public int Calc(int x, int y);
}
```

仮想メンバー関数

仮想メンバー関数は、「仮想」仮想形式を持つ UML 操作に翻訳されます。

例 **190:** 仮想メンバー関数の翻訳

C/C++ :

```
class CPen {
public:
  virtual void Draw(); // Virtual member function
  double GetRep(); // Non-virtual member function
};
class CPenD :public CPen {
public:
  virtual void Draw(); // Redefinition of CPen::Draw()
```

```
};
```

UML：

```
public class CPen {
  public virtual void Draw();
  public double GetRep();
}
public class CPenD :CPen {
  public virtual void Draw();
}
```

純仮想メンバー関数

純仮想メンバー関数は、通常のメンバー関数と同じように翻訳されます。

「純仮想」はメンバー関数自身の翻訳には影響ありませんが、含有クラス（抽象クラス）の翻訳方法に影響を及ぼします。

例 **191:** 「純仮想」クラスの抽象クラスへの翻訳

C/C++：

```
class C {
public:
  virtual int f(int) = 0; // pure virtual member function
  C() {};
};
class D :public C {
};
```

UML：

```
abstract public class C {
    public int f ( int );
    public C () ;
}
public class D :C {
}
```

純仮想メンバー関数のみを含む C++ クラスは、UML インターフェイスに翻訳できます。これは、cppImportSpecification のプロパティ エディタで、［Import C++ pure virtual classes to UML interfaces］を選択することで実行できます。

例 **192:** 純仮想クラスのインターフェイスへの翻訳

C/C++：

```
class Shape {
  public:
    virtual void rotate(int) = 0;
    virtual void draw() = 0;
    virtual bool isclosed() = 0;
};
```

```
class Box :Shape {
};
```

UML：

```
public interface Shape {
  void rotate( int );
  void draw();
  bool isclosed();
}
public class Box :Shape {
}
```

インターフェイスは、UML では抽象クラスより「強い」概念です。異なる点は、インターフェイス関数のいずれかが継承（実現化）クラスに実装されなかった場合、UML チェッカーによって警告メッセージが表示されることです。抽象クラスでは、継承階層の下位層で実装が可能なため、警告メッセージが表示されることはありません。

例 **193:** 実装されなかった関数の警告メッセージ ———————————————

```
Class Box:Warning:TSC0124:Operation 'rotate' in interface
'Shape' was not realized by class 'Box'.
Class Box:Warning:TSC0124:Operation 'draw' in interface
'Shape' was not realized by class 'Box'.
Class Box:Warning:TSC0124:Operation 'isclosed' in interface
'Shape' was not realized by class 'Box'.
```

静的メンバー

静的メンバーは静的 UML 属性または操作（"Classifier" 所有者スコープを持つ）に翻訳されます。

例 **194:** 静的メンバーの翻訳 ———————————————

C/C++：

```
class C {
public:
  static int k;
  static void InitI(int);
};
```

UML：

```
public class C {
    public static int k;
    public static void InitI ( int ) ;
}
```

定数メンバー

定数メンバーは、可変性が「凍結」された通常のメンバーに翻訳されます。

例 **195:** 定数メンバーの翻訳

C/C++ :

```
class C {
public:
  const int cm; // constant member
  C(int k) :cm(k) {};
  void Do(double);
  void Undo(double) const; //constant member function
};
```

UML :

```
public class C {
  const int cm;
  public C ( int k ) ;
  public void Do ( double );
  public <<IsQuery="true">> void Undo ( double );
}
```

メンバー定数

メンバー定数は、可変性が「凍結」され、デフォルト値を持つ通常のメンバーに翻訳されます。

例 **196:** メンバー定数の翻訳

C/C++ :

```
class X {
public:
  static const int i = 99; // member constant
};
const int X::i; // definition of i
```

UML :

```
public class X {
  public static const int i = 99;
}
```

変更可能メンバー変数

変更可能メンバー変数は通常のメンバー変数として翻訳されます。

ビットフィールド メンバー変数

C++ の構造体、共用体、またはクラスはビットフィールドのメンバー変数を持つことがあります。

ビットフィールド メンバー変数は、（TTDCppPredefined プロファイル パッケージから）<<bitfield>> ステレオタイプが適用された UML メンバー変数に変換されます。ビットフィールドのサイズはタグ値によって指定されます。bitfield サイズは［プロパティの編集］ダイアログで変更できます。

例 **197:** ビットフィールド メンバー変数の翻訳

C/C++：

```
struct mybitfields {
  unsigned a : 4;
  unsigned b : 5;
  unsigned c : 7;
};
```

UML：

```
public <<struct>> class mybitfields {
  public <<bitfield(.'size' = 4.)>>'unsigned int'
  public <<bitfield(.'size' = 5.)>>'unsigned int' b;
  public <<bitfield(.'size' = 7.)>>'unsigned int' c;
}
```

共用体内部のクラス

共用体内部に定義されたクラスは、共用体のオーナーのスコープにインポートされます。

この規則は、ある C++ 共用体が UML の choice にインポートされたが UML の choice 内部にクラスの定義は許可されない、と規定します。

例 **198:** 共用体内部のクラスの翻訳

C/C++：

```
typedef struct S1 {
  union U1 {
    struct S2 {
    } vS2;
  } vU1;
};
```

UML：

```
public <<struct>> class S1 {
  public <<struct>> class S2 {
  }
  public <<IsUnion = "true">> choice U1 {
    public part S2 vS2;
  }
```

```
  public part U1 vU1;
}
```

フレンド

クラス C と他の宣言 D との間の「フレンドシップ」は、D の実装がアクセスできる C のメンバーに影響します。UML では、この関係は <<friend>> ステレオタイプを適用した依存によって表現されます。

例 **199:** フレンド宣言の翻訳

C/C++:

```
class X {};
class Y {
  friend class X;
};
```

UML:

```
public class X { }
public class Y <<friend>> dependency to X { }
```

注記

C++ のフレンドの概念は、UML には存在しません。UML チェッカーは、定義のアクセスにおいて可視性をチェックする際に <<friend>> ステレオタイプを考慮しません。したがって、フレンド宣言を有効に使うには、インラインの C++ コード内で使用する必要があります。

継承

継承関係は UML の汎化に翻訳されます。

例 **200:** 継承の翻訳

C++ :

```
class A {
public:
  int am;
  A(char);
};
class B :public A {
public:
  char bm;
  virtual void calc();
  void set();
};
class C :public B {
public:
  int am;
```

```
  double cm;
  void calc(); // Redefines B::calc()
  void set();
};
```

UML：

```
public class A {
    public int am;
    public A ( char ) ;
}
public class B :A {
    public char bm;
    public virtual void calc ();
    public void 'set' ();
}
public class C :B {
    public int am;
    public double cm;
    public void calc ();
    public void 'set' ();
}
```

C++ では、クラス間での指定なしの可視性の継承は、private 可視性を意味します。この情報は、TTDCppPredefined プロファイルの `<<inheritanceVisibility>>` ステレオタイプを使用して維持する必要があります。この理由は、UML の汎化のため C++ アプリケーション ジェネレータで、デフォルトで公開継承可視性が使用されるからです。詳細については、継承アクセス指定子を参照してください。

複数継承

クラスが 2 つ以上のベース クラスから継承する場合も、C++ の継承の翻訳ルールが使用されます。

例 **201:** 複数継承の翻訳

C/C++：

```
class A {
  public:
  int m;
};
  class B {
  public:
  int m;
  int n;
};

class C:public A, public B {
};
```

UML：

```
public class A {
    public int m;
}
```

```
public class B {
    public int m ;
    public int n;
}
public class C :A, B {
}
```

仮想継承

仮想継承は通常の継承と同じように翻訳されます。

TTDCppPredefined プロファイルから **<<virtualInheritance>>** ステレオタイプを設定すると、仮想継承が翻訳されます。

継承アクセス指定子

C++ では、継承メンバーへのアクセスを減少させる目的で、継承にアクセス指定子（public、protected、private）を使用できます。UML では、これはできません（すべての継承は「public」です）。

TTDCppPredefined プロファイルから **<<inheritanceVisibility>>** ステレオタイプを設定すると protected または private アクセス指定子が翻訳されます。UML 汎化にプロパティ エディタを使用してこの情報を編集します。

前方宣言

前方宣言は UML に翻訳されません。このルールは、ヘッダー ファイルで後で定義が出現するすべての前方宣言に当てはまります。これは最も一般的なユース ケースです。このような前方宣言の目的は、識別子を C/C++ コンパイラに知らせて定義前に使用できるようにすることです。

しかし、ヘッダー ファイルに定義が存在しない前方宣言を作成することも可能です。この場合、欠如した定義を表現する別タイプが生成されます。

例 **202:** 前方宣言の翻訳

C/C++：

```
class A;
class B {
  public:
    static A& g() { return *pA; }
  private:
    static A* pA;
};
```

UML：

```
public class A {
}
public class B {
  public static (<<CppReference>> part A) g();
  private static A pA;
```

```
}
```

ここで、class A は前方宣言です。インポート後のファイルには定義がありません。C/C++ インポータはそれを表すために空のクラスを作成します。

クラス図およびパッケージ図の生成

C++ インポートでは、表示するインポート済み定義がある場合、クラス図およびパッケージ図を作成します。［C++ Imported Types］という名前のクラス図が生成されて、汎化と関連を表すラインを含めて、インポートされたクラス、インターフェイス、データ型（シンタイプと choice 以外）が含まれます。

C/C++ インポートでも、他の（ネストされた）パッケージが含まれているパッケージ用に［C++ Imported Packages］というパッケージ図が生成されます。

例 **203:** クラス図の作成

C/C++ :

```
namespace N{
  class A  { int z; };
  class B :A { long y; };
  class C :A { double x; };
}
class D { long h; };
class E { D d1; };
class F { D* d2; };
```

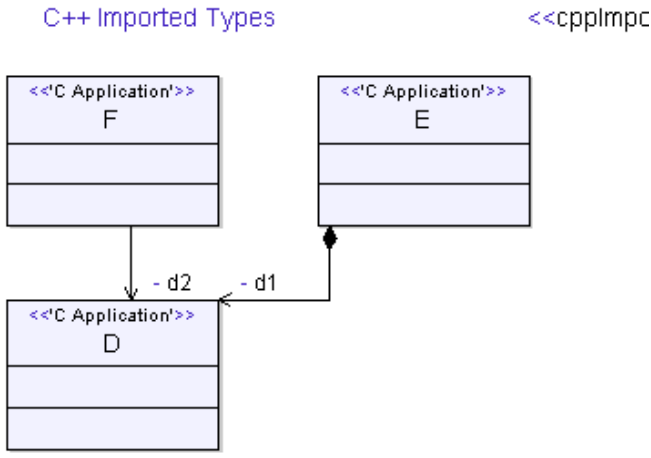


図 169: パッケージ "N" 内のクラス図［C++ Imported Types］

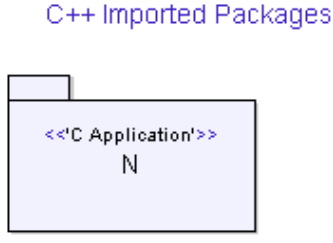


図 170: インポート パッケージ内のクラス図 ［C++ Imported Packages］

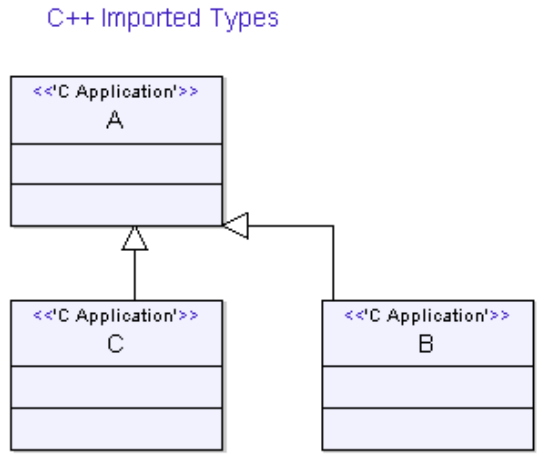


図 171: パッケージ "N" 内のクラス図［C++ Imported Types］

不完全型宣言

C/C++ は不完全なクラス、構造体、共用体、および列挙の宣言を許可します。これらの型はタグを与えずに不完全として宣言されます。このことから、不完全型はタグなし型とも言われます。

不完全型は以下の状況で使用されます。

- データ宣言（変数、定数など）
- 型宣言（`typedef` など）
- "Pointless" 宣言（データまたは型を宣言しない場合など）。"Pointless" 宣言の不完全型は UML には翻訳されず、この宣言を無視することを示す警告が表示されます。

不完全型も、同種の名前付き型と同様にインポートされます。型宣言では、同じ型宣言で複数の型名を定義した場合を除き、シンタイプは生成されません。

例 **204:** 型宣言の不完全型

C/C++ :

```
typedef enum { aa, bb, cc } tEnum1;

typedef class {
```

```
  public:
    int i;
    int j;
    enum tEnum2 { xx, yy, zz } k;
    void init(int ii, int jj, tEnum1 kk);
  } CIncomplete;
```

UML

```
  public enum tEnum1 {
      aa,
      bb,
      cc
  }
  public class CIncomplete {
      public int i;
      public int j;
      public enum tEnum2 {
          xx,
          yy,
          zz
      }
      public tEnum2 k;
      public void init( int ii, int jj, tEnum1 kk);
  }
```

データ宣言または型宣言で使用される不完全型は、最後に宣言された変数または型の名前が与えられ、「`incomplete_`」という接頭辞がつきます。

例 **205:** 不完全型の翻訳 ——————————————————————————

C/C++：

```
  struct S {
    int i;
  struct {
    int j;
  } ss1, *ss2, ss3[2]; // Data declarations
  };
  typedef enum {
    a, b, c
  } ss1, *ss2, ss3[2]; // Type declarations
  typedef struct {
    int i;
  }; // Missing type name - "pointless" declaration
  struct {
    int i;
  }; // Missing variable name - "pointless" declaration
```

UML：

```
  public <<struct>> class S {
    public int i;
    public <<struct>> class ‘incomplete_ss3@1’ {
      public int j;
    }
    public part ‘incomplete_ss3@1’ ss1;
    public ‘incomplete_ss3@1’ ss2;
    public CArray<‘incomplete_ss3@1’, 2> ss3;
```

```
}
public incomplete_ss3 enum  {a, b, c}
public syntype ss1 = incomplete_ss3;
public syntype ss2 = CPtr<incomplete_ss3>;
public syntype ss3 = CArray<incomplete_ss3, 2>;
```

注記

不完全クラス、構造体、共用体は、不完全ですが、スコープ ユニットを定義します。494 ページの「スコープ ユニット」に記載される翻訳ルールが通常どおり適用されます。

オーバーロード演算子

オーバーロード演算子は UML の演算子定義にインポートされます。

例 **206:** オーバーロード演算子

C/C++ :

```
class MyInt {
public:
   int x;
   MyInt operator+ ( const MyInt& i );
};
```

UML :

```
public class MyInt {
    public int x;
    public part MyInt '¥+'( const inout part MyInt i );
}
```

参照

オーバーロード変換演算子

テンプレート

- クラス テンプレート
- 関数テンプレート

参照

定数へのポインタ

クラス テンプレート

C++ のクラス テンプレートは UML のクラス テンプレートに翻訳されます。C++ のテンプレート パラメータは UML のテンプレート パラメータに翻訳されます。値テンプレート パラメータは UML の const 値テンプレート パラメータに翻訳されます。それ以外のすべてのテンプレート パラメータは、UML のタイプ テンプレート パラメータにマッピングされます。

例 **207:** クラス テンプレートの翻訳 ——————————————————————

C/C++ :

```
template <class T, int i> class Buffer {
  T v[i];
  int size;
};
```

UML :

```
template <type T, const int i> public class Buffer {
  private CArray<T, ([[i]])> v;
  private int ‘size’;
}
```

——————————————————————————————————

テンプレートのクラス メンバーは通常のクラスのメンバーと同じようにマッピングされます。テンプレート名がテンプレート クラス本体から参照されている場合は、実テンプレート パラメータが与えられます。それ以外の場合は、対応する UML 参照はテンプレートにバインドされません。

例 **208:** クラス テンプレートの翻訳 ——————————————————————

C/C++ :

```
template <class C> class String {
public:
  String();
  String(const C*);
  String(const String&); // reference to template name
};
```

UML :

```
template <type C> public class String {
  public String();
  public String( const C );
  public String( const inout part String<C> ); // actual
parameter C attached
}
```

——————————————————————————————————

C++ のテンプレート インスタンス化は UML のテンプレート インスタンス化にマッピングされます。値テンプレート パラメータの実際の値は、定数式に翻訳されます（定数式を参照）。

例 **209:** テンプレート インスタンス化の翻訳

C/C++ :

```
template <class T, int i> class Buffer {
  T v[i];
  int sz;
};

Buffer<char, 127> cbuf;
```

UML :

```
template <type T, const int i> public class Buffer {
  private CArray<T, ([[i]])> v;
  private int sz;
}

public part Buffer<char, 127> cbuf;
```

例 **210:** テンプレート インスタンス化の翻訳

C/C++ :

```
template <class C> class String {
  C*rep;
public:
  String();
  String(const C*);
  String(const String&);
  C read(int i) const;
};

String<char> cs;
String<unsigned char> us;
String<wchar_t> ws;

class Jchar {
};
String<Jchar> js;
typedef String<char> CString;
```

UML :

```
template <type C> public class String {
  private C rep;
  public String();
  public String( const C );
  public String( const inout part String<C> );
  public <<IsQuery="true">> part C read( int i);
}

public part String<char> cs;
public part String<'unsigned char'> us;
public part String<wchar_t> ws;

public class Jchar {
```

```
}
public part String<Jchar> js;
public syntype CString = String<char>;
```

関数テンプレート

C++ の関数テンプレートは UML の関数テンプレートに翻訳されます。C++ の関数テンプレート パラメータは、テンプレート クラスと同じ方法で、UML の関数テンプレート パラメータに翻訳されます（クラス テンプレートを参照）。

例 **211:** テンプレート関数の翻訳

C/C++ :

```
template <class T> void sort(int *);
```

UML :

```
template <type T > public void sort( CPtr<int> );
```

デフォルト テンプレート引数

デフォルト値が指定された C++ のテンプレート引数は、デフォルト値を持つ UML のテンプレート パラメータに翻訳されます。

例 **212:** デフォルト テンプレート引数

C/C++ :

```
template <class T, class U = char> class C {
public:
  T t;
  T f();
  C(U chr);
};

C<int> var1;
C<int, char> var2;
C<int, bool> var3;
```

UML:

```
template <type T, type U = char> public class C {
  public part T t;
  public T f();
  public C( part U chr );
}

public part C<int> var1;
public part C<int, char> var2;
public part C<int, bool> var3;
```

例外

関数が特定の例外型をスローするという指定は、対応する UML 操作の指定に翻訳されます。これは関数の翻訳です。

関数が例外をスローしないという指定（`throw()`）は、UML では TTDCppPredefined プロファイルの `<<noException>>` を適用して表現されます。

例 **213:** 関数への例外指定の翻訳

C/C++ :

```
double foo() throw(char, int);
void bar() throw();
```

U2:

```
public double foo() throw char, int;
public <<noException>> void bar();
```

その他

- 言語構成要素
- 非言語構成要素
- C コンパイラ用の翻訳ルール

言語構成要素

揮発

揮発宣言は通常の宣言と同じように翻訳されます。揮発指定子はある種のコンパイラディレクティブと見なすことができます。したがって、UML 翻訳では非表示であるべきです。

リンク

C/C++ 定義のリンクは、通常 UML 翻訳では見えません。しかし、定義に extern リンクが明示的に指定されている場合は、インポート済み定義の「外部」プロパティによって表されます。

例 **214:** 異なるリンクを持つ定義の翻訳

C/C++ :

```
extern int a; // Declaration of a
extern int a; // Legal redeclaration of a
int a; // Definition of a
extern "C" {
  struct S {
    int x;
```

```
  };
}
static void foo();
```

UML：

```
public int extern a;
public <<struct>> class S {
  public int x;
}
public void foo();
```

using ディレクティブ

C++ の using ディレクティブ（“using namespace”）は、<<access>> 依存関係に翻訳されます。依存関係の提供者は、参照される名前空間に対応するパッケージです。

例 **215: using** ディレクティブの翻訳

C/C++：

```
namespace X {
  class C {};
}

using namespace X;

C var;
```

UML：

```
package ImportedDefinitions <<access>> dependency to X {
    package X {
        public <<External="true">> class C {

        }
    }
    public part C extern var;
}
```

Using 宣言

C++ の using 宣言 (“using”) は、1 つの <<access>> 依存に翻訳されます。依存の供給者は、参照される C++ 定義に対応する UML 定義です。

例 **216: using** 宣言の翻訳

C/C++:

```
namespace K
{
  class A {};
}
```

```
using K::A;
A aInst;
```

UML:

```
<<access>> dependency to K::A;

package K {
    public class A { }
}
public part A extern aInst;
```

コンパイラ固有の言語構築子

大半の C/C++ コンパイラは、サポートする C/C++ 言語仕様を越えた、固有の追加機能をサポートしています。C/C++ インポータを使うと、適切な言語方言 (処理系固有の言語) が使用されていれば、入力ファイル内のこういった構築子を処理できます (詳細は C/C++ dialect を参照してください)。ただし、通常はこういった言語拡張の存在は UML への翻訳には影響を与えません。

上記の基本ルールの例外を以下にあげます。

__declspec

Microsoft および GNU コンパイラでサポートされる `__declspec` キーワードを使用すると、TTDCppPredefined プロファイルの `<<__declspec>>` ステレオタイプに翻訳されます。

例 **217: __declspec** キーワードの翻訳

C/C++:

```
void __declspec(dllexport) foo();
```

UML:

```
void <<__declspec(. modifier = __declspecModifier(. kind =
“dllexport”.) .)>> foo();
```

非言語構成要素

マクロ

マクロはプリプロセスされ、C/C++ ファイルで指定された値、およびプリプロセッサの Options で定義された値を使用して、展開されます。作成された C/C++ コードが UML に翻訳されます。それぞれの言語構成要素の翻訳ルールに関する説明を参照してください。

参照

527 ページの「プリプロセッサ」

C/C++ の定義済み型への参照

インポートされる C/C++ ヘッダファイルで特殊な型参照を使用して、定義済みの UML 型にマップできます。このような型参照は SDL_<name> という形式です。ここで <name> は定義済みの UML 型の名称です。たとえば、SDL_Integer は UML の整数型にマップされます。

SDL という接頭辞は、この機能が主として C コードジェネレータで使うために設計されているという理由で使用されています。インポートされたヘッダーファイルが C コードジェネレータライブラリからのこれらの型の定義を含む場合は、オプション Translation of depending declarations をオフして、SDL_ 接頭辞付きの UML 型の生成を抑止できます。

下の表に C/C++ インポータが認識する特殊な型参照の一覧をあげてあります：

C/C++ 名	UML データ型
SDL_Boolean, SDL_boolean	Boolean
SDL_Integer, SDL_integer	Integer
SDL_Real, SDL_real	Real
SDL_Natural, SDL_natural	Natural
SDL_Time, SDL_time	Time
SDL_Duration, SDL_duration	Duration
SDL_PId	Pid
SDL_Character, SDL_character	Character
SDL_Charstring, SDL_charstring	Charstring
SDL_IA5String	IA5String
SDL_NumericString	NumericString
SDL_VisibleString	VisibleString
SDL_PrintableString	PrintableString
SDL_Bit	Bit
SDL_Bit_String	BitString
SDL_Octet	Octet

C/C++ 名	UML データ型
SDL_Octet_String	OctetString
SDL_Object_Identifier	ObjectIdentifier
SDL_Null	Any

変換中に上表の名称を見つけると、情報メッセージが出力されます。

例 **218:** 定義済み **UML** 型の **C** コード名

C/C++ :

```
SDL_Integer func();
```

UML :

```
public Integer func();
```

メッセージ :

```
Information sdltypes.h(8):Recognized C++ name 'SDL_Integer'
of predefined SDL type. Imported to 'Integer'
```

参照

プリプロセッサ

C コンパイラ用の翻訳ルール

言語

C Application ステレオタイプの Language 属性は、C++ ではなく C に設定されます。

STL サポート

C++ 標準テンプレート ライブラリの UML 版を含む特殊なライブラリ パッケージ「std」を利用できます。この定義は、STL 定義を使用している C++ ヘッダーが UML にインポートされる際に UML モデルから参照されます。これらの定義を使用するために、標準の STL ヘッダーを UML にインポートする必要はありません。

パッケージ「std」は以下のものをサポートします。

- STL コンテナ :
  `vector, list, set, multiset, map, multimap, pair, deque, queue, priority_queue, stack, string`

- STL アルゴリズム：
  `for_each, find, find_if, adjacent_find, count, count_if, search, search_n, find_end, find_first_of, iter_swap, swap_ranges, transform, replace, replace_if, replace_copy, replace_copy_if, generate, generate_n, remove, remove_if, remove_copy, remove_copy_if, unique, unique_copy, reverse, reverse_copy, rotate, rotate_copy, random_shuffle, partition, stable_partition, sort, stable_sort, partial_sort, partial_sort_copy, nth_element, lower_bound, upper_bound, equal_range, binary_search, merge, inplace_merge, includes, set_union, set_intersection, set_difference, set_symmetric_difference, push_heap, pop_heap, make_heap, sort_heap, max_element, min_element, next_permutation, prev_permutation`
- STL 入／出力機能、以下に例を示します：
  `basic streams および file input/output streams including cin, cout, cerr, <<, >>`

UML で STL サポートが必要とされる場合、`CppStdLibrary` アドインを起動してパッケージ「`std`」をプロジェクトに追加する必要があります。

注記

STL サポートは現在 C++ アプリケーション ジェネレータでの作業に限られています。STL は C コード ジェネレータでは使用できません。インポートされた C++ コードがテンプレートを使用する場合は、C コード ジェネレータでは使用できません。

C/C++ のインポートとビルド タイプ

- C コード ジェネレータ
- C++ アプリケーション ジェネレータ

C コード ジェネレータ

インポート時に C Application ステレオタイプの以下の C コード生成属性を設定する必要があります。

- Language は C（C モードでのインポート）または C++（その他）に設定します。
- C モードでのインポートでは、C name を使用して `struct`、`union`、`enum` の参照名を宣言します。

C++ アプリケーション ジェネレータ

C++ アプリケーション ジェネレータでは、TTDCppPredefined プロファイルのステレオタイプを使用します。C/C++ インポート時に以下のステレオタイプが使用されます。これらのステレオタイプの詳細については、パッケージ TTDCppPredefined の説明を参照してください。

既知の制限事項

C++ 言語の制限事項

オーバーロード変換演算子

オーバーロード変換演算子はサポートされません。次の警告メッセージが表示されます。

```
Ignoring conversion operator.  Conversion operators are not
yet supported.
```

const のオーバーロード

const のオーバーロードはサポートされません。そのようなオーバーロード定義は、インポート時に無視され、警告メッセージが表示されます。

```
Ignored conflicting declaration
```

ellipsis 関数

ellipsis 関数引数は無視されます。次の警告メッセージが表示されます。

```
Cannot translate ellipsis function 'MyFunc'.
```

しかし、関数自体は（ellipsis の前にある引数も含め）インポートされます。UML から別バージョンの ellipsis 関数を呼び出す方法については、unspecified 引数を参照してください。

関数ポインタ

関数ポインタのインポートにはいくつか制限があります。

1. 参照型を保持する仮パラメータは考慮されません。

例 **219:** 参照型を持つ仮パラメータ

C++ :

```
typedef int (*pf)(int& );
int f1(int& i);
const pf pf1 = f1;
```

UML :

```
public <<operationReference>> interface pf {int call(inout
int);}
public int f1(inout int i);
public const pf pf1 = operation f1(int);
```

2. 定数型を保持する仮パラメータは考慮されません。

例 **220: :** 定数型を持つ仮パラメータ

C++ :

```
typedef int (*pf)(const int);
int f1(const int i);
const pf pf1 = f1;
```

UML :

```
public <<operationReference>> interface pf {
     int call(const int);}
public int f1(const int i);
public  const pf pf1 = operation f1(int);
```

3. 値で渡されるクラス型を保持する仮パラメータは考慮されません。

例 **221:** 値で渡されるクラス型を持つ仮パラメータ

C++ :

```
class X {};
typedef int (*pf)(X);
int f1(X i);
const pf pf1 = f1;
```

UML:

```
public class X {}
public <<operationReference>> interface pf {
     int call( part X);
}
public int f1( part X i);
public const pf pf1 = operation f1(X);
```

4. 関数型へのポインタはサポートされますが、関数型はサポートされません。詳細は関数型を参照してください。

例外

例外として宣言されたすべての定数型変数は `const` 修飾子なしでインポートされます。これを知らせる以下の警告メッセージが表示されます。

```
Constant type is used as exception. This exception will be
imported without const
```

例 **222:** 例外としての定数

C++ :

```
void foo () throw (const int);
```

UML:

```
public void foo() throw int;
```

プリプロセッサ

マクロ名およびマクロ定義は、他のプリプロセッサの構成要素と同様 UML に翻訳されません。

式の評価

ほとんどの式はインポート時に評価されることはなく、非形式な UML 式に翻訳されます。結果に UML レベルでもセマンティックな影響がある場合、たとえば配列のサイズを示す定数式などの場合、式が評価されます。

評価された式には 2 つの制約があります。

- `sizeof` 式はインポート時に評価されず、非形式な式としてインポートされる。

```
struct afx_float {
  char floatBits[ sizeof(float) ];
};
```

```
Warning: Unable to evaluate sizeof expression. It will be
imported as informal expression.
```

```
public <<struct>> class afx_float {
  public CArray<char, ([[sizeof(float)]])> floatBits;
}
```

- `cast` 式は式の評価時に無視される。したがって UML レベルでのタイプ修正による値の修正に影響を与えません。

さらに、C/C++ 言語の演算子には、UML に対応する表現がないものがあります。したがって、これらの演算子を参照する二項式および単項式は、非形式な式に翻訳されます。この演算子の例には、「`+=`」、「`-=`」、「`,`」などがあります。詳細については、二項式と単項式を参照してください。

int から **enum** への暗黙の変換

int から enum へ、または enum から int への暗黙の変換はサポートされません。たとえば以下のようなことはできません。

例 **223:** 暗黙的および明示的な **cast**

```
public enum cars {  volvo, saab, audi, vw, bmw  }
...
int a;
a = volvo; // ERROR:not allowed
...
int b;
cars c;
b = cast<int>(volvo);
c = cast<cars>(1);
```

修飾子としてのテンプレート パラメータの使用

テンプレート パラメータを修飾子として使用すると、インポートされた定義に関してセマンティック分析によりエラー レポートが生成されます。これは、テンプレート内でテンプレート パラメータを修飾子として使用することが許可されないからです。

例 **224:** インポート後のシンタイプからのエラー

C++ :

```
template <class X> class Y {
  typedef X::pointer iterator;
  iterator End;
};
```

インポート後の UML :

```
template <type X > public class Y {
  private syntype iterator = X::pointer;
  private iterator End;
}
```

インポート後のシンタイプによりエラー メッセージが表示されます。

```
Syntype iterator:Error:TNR0047:
Failed to find definition of pointer (while looking for Type).
Syntype iterator:Error:TNR0034:
Failed to find Type X::pointer of Syntype.
```

定数へのポインタ

UML には C++ const ポインタの表現はありません。UML では定数への C++ ポインタはサポートされません。

たとえば、C++ 型 `const char*` を考えます。この C++ 型は、UML の `const 'char*'` 型 にインポートされます。これは定数なので、初期化中に一度だけ値が割り当てられます。しかし、C++ `const char*` は定数へのポインタであり、定数を指し示す変数なので、値が変わります。

```
const char* cc;
cc = "constant string 1";
cc = "constant string 2";
```

他にも、C コード 生成の際に適用される制限事項があります。UML 定数は関数呼び出しによって初期化することはできません。したがって、定数文字列へのポインタを返すインポート後の関数を使用すると、エラー メッセージが出力されます。

例 **225: C++** の定数へのポインタの **UML** での使用

C/C++ :

```
const char* func1( const int a );
```

インポート後の UML :

```
public 'char*' func1( const int a );
```

外部関数の結果の格納：

const 'char*' 変数を使用して func1 戻り値を UML に保存した場合、定数なので定義時に初期化されます。

```
const 'char*' cc = func1( 1 );
```

その結果、C コード ジェネレータによって以下のエラー メッセージが出力されます。

```
ERROR TIL2084:Procedure call not allowed where constant required
```

以下のように 'char*' 変数が使用された場合、

```
'char*' cc = func1( 1 );
```

C コンパイラによって以下のエラー メッセージが出力されます。

```
const01.c(511) :error C2440:'=' : cannot convert from 'const char
*' to 'char *'
```

C++ 定数ポインタは、特に関数によって C++ 定数文字列が返される場合など、UML で使用することはできません。

代替法として、以下の 2 通りの方法があります。

1. C++ ヘッダー ファイルを変更して、直接 const char* を使用する代わりに typedef を使用する。これにより、この外部型を使用して C++ 定数文字列を格納できるようになります。

例 **226: C++** 定数文字列の **UML** での格納 ———————————————————

C/C++：

```
typedef const char* cstr;
cstr func1( const int a );
```

インポート後の UML：

```
public syntype cstr = 'char*';
public cstr func1(const int a);
```

外部関数の結果の格納：

```
cstr cc = func1( 1 );
```

---

2. C++ ヘッダー ファイルはそのままにして、UML のキャスト（`cast<'char*'>`）操作を使用して、外部関数を呼び出す。外部関数を呼び出しは以下のようになります。

   ```
   'char*' cc = cast<'char*'>( func1(1) );
   ```

注記

代入値を使用する場合、変数が指し示す文字列を変更しないことが重要です。外部 C++ コードは、ほとんどの場合定数値に依存します。

便宜性の制限事項

標準インクルードを持つヘッダーのインポート

インポート後のヘッダーファイルにインクルードされた他のヘッダー ファイルが含まれていて、オプション［Do not import definitions from included header files］が無効になっている場合、インクルードされたファイルのすべての定義が一緒にインポートされます。これは、インクルード ファイルのツリーが大きい場合、特に標準インクルードの場合、インポート後のモデルが大きくなりパフォーマンスに影響を及ぼします。定義は、Selective import を使用して個別にインポートできます。

10

DOORS のインポート

DOORS に作成された要件とリンクを Tau にインポートする方法に関する情報については、要求のインポートを参照してください。

注記

この方法は、要件とリンクの正しい UML 表記を作成するという点で、推奨される DOORS データから Tau へのインポート方法です。

11

SDL のインポート

この章では、Tau で SDL 仕様をインポートして、インポート結果を UML モデルに変換する方法について説明します。

動作原理

SDL インポートは、SDL 仕様を含むファイルに基づき、定義済みの一連の変換ルール（539 ページの「SDL から UML への変換ルール」）を使用して、対応する UML モデルに変換します。このインポート結果のモデルは、自動的にロードされるので、UML モデルの完成に向けて、他の作業に移ることができます。

SDL システムのインポート

SDL Suite 対応の SDL インポート

SDL 仕様を確実にインポートするには、SDL システムが完全で、セマンティック上正しいものでなければなりません。このためには、エクスポートする前に、システムで完全な分析を実行する必要があります。

CIF 情報は、SDL Suite の CIF Exporter を使用して、SDL PR ファイルに追加できます。CIF Exporter は、SDL Suite（SDL Suite のバージョン 4.6.1 以降）の［Generate/Convert GR to IBM Rational Tau G2 CIF...］メニューから起動する必要があります。この機能の詳細については、IBM Rational SDL Suite のドキュメントを参照してください。

注記

プロセス図からの状態機械図の生成は、CIF からのインポート後にのみ実行されます。ダイアグラム上の要素の配置は、元の SDL モデルの配置とほとんど同じになります。

バージョン 4.6.0 以前のバージョンの SDL Suite から SDL インポートを行う際、CIF コメント付きの SDL-PR フォーマットを利用するには、［Generate/Convert GR to CIF... ］コマンドを使用します。以下の設定とオプションを適用します。

- *.sdt ファイルに［CIF generation］を選択する。
- ［Generate one CIF file］を**オン**にする。
- ［Include CIF comments］ボックスを**オン**にする。
- ［Include graphical SDT references］を**オフ**にする。
- **適切な CIF ファイルの名前 :** システムのターゲット ディレクトリに入れる。

注記

CIF 情報なしで SDL からインポートされたモデルについては、状態機械実装を状態機械図にドラッグアンドドロップすることで、ステート チャート図を作成できます。この場合、作成されたダイアグラム要素に自動レイアウトが適用されます（自動レイアウトの詳細については、ダイアグラムの自動レイアウトを参照）。

SDL システムが CPP2SDL ユーティリティを使用する場合、または外部 C/C++ 定義がシステムに存在する場合、'EXTERNAL' のタグが付いた定義は、インポート後の UML モデルに表示されません。モデル内でこれらの定義が使用されていると、SDL システムのインポート後に名前解決エラーが生じます。このエラーを避けるには、SDL システムのインポート前に外部 C/C++ コードをインポートし、インポート結果のパッケージに noScope ステレオタイプを適用します。コード生成の前に、このステレオタイプを必要なパッケージ間の依存関係に置き換えないと、エラーが生じます。

SDL インポートを起動するには、プロジェクトを含むワークスペースを開く必要があります。

- ［モデル ビュー］で**モデル**項目を選択します。
- インポート ウィザードを開きます（［ファイル］メニューから［インポート］コマンドを選択）。
- ダイアログで［SDL のインポート］を選択して、［OK］をクリックします。
- 表示されるダイアログで、インポートする SDL ファイルを指定します。
- **処理系固有の SDL：**通常は SDL Suite です。
- 求める結果に応じて［SDL/PR］または［SDL/CIF］を設定する必要があります。

2 番目のダイアログを閉じると、以下のような結果になります。

- パッケージ **ImportedSDLDefinition** がモデル内に作成される。
- ステレオタイプ **sdlImportSpecification** がパッケージに適用される。
- インポートのベースとなった SDL ファイルが、パッケージのステレオタイプ インスタンスに値として格納される。
- インポートが実行され、作成されたパッケージに結果が追加される。

SDL Suite 日本語版からのインポート

注記

この説明は、日本語版 SDL Suite にのみ当てはまります。標準（英語）版 SDL Suite からインポートする場合、この説明は無視してください。

日本語版 SDL Suite は、Solaris と Windows に対応しており、日本語テキスト（文字列やコメントなど）の保存には、「ネイティブ」エンコードを使用します。この場合の「ネイティブ」は、SHIFT-JIS（Windows）と EUC-JP（Solaris）を指します。

ただし、SDL インポートでは、エンコードされるテキストが UTF-8 を使用していることを前提としています。したがって、SDL Suite で作成されたファイルは、インポートする前に UTF-8 方式に変換しておく必要があります。利便性を考慮して、Solaris、Linux、Cygwin でさまざまなエンコードを UTF-8 に変換する標準プログラムである `iconv` ユーティリティが呼び出され、インポートの前にこの変換作業が実行されます。（`iconv` ユーティリティは、Tau インストール プログラムに用意されています。）

iconv に対応するエンコードは `iconv -l` で一覧表示できます。これらの値は、入力 SDL ファイルに使用するエンコードを指定するため、環境変数 `TauImportedSDLEncoding` に割り当てることができます。以下に、状況に応じて使用できる値について説明します。

- **SHIFT-JIS**：Windows で日本語版 SDL Suite を使用して SDL ファイルが作成されている場合
- **EUC-JP**：Solaris で日本語版 SDL Suite を使用して SDL ファイルが作成されている場合
- **UTF-8**：ファイルがすでに UTF-8 に変換されている場合

環境変数の値が iconv ユーティリティに指定どおりに渡され、インポート元の SDL ファイルのエンコードが指定されます。変数が存在しないか値がない場合、ファイルが UTF-8 でエンコードされていると見なされます。

SObjectGeode 対応の SDL インポート

ObjectGeode SDL インポート インターフェイスは、SDL Suite からのインポートのインターフェイスと基本的に同じです。ただし、固有表現が異なります。

- **SDL 固有表現 :**ObjectGeode。
- 求める結果に応じて［SDL/PR］または［SDL/CIF］を設定する必要があります。
- ファイルのインポートを可能にするため、「SDL for Export」という名前で、ObjectGeode に保存する必要があります。

注記

ObjectGeode のデフォルト、ネイティブ ストレージ フォーマットは、CIF 情報付きの SDL-PR です。SDL ファイルの保存時に CIF コメントが出力されます。

サポートされる SDL

SDL インポートは、プレーン SDL-PR（テキスト構文）または CIF（CommonInterchange Format）コメント付き SDL-PR のいずれかで表現された SDL-96 をサポートします。

注記

SDL および CIF の詳細については、国際電気通信連合（ITU-T）勧告 Z.100 および Z.105 を参照してください。また、TauSDL Suite および ObjectGeode の SDL と CIF のサポート内容については、各ツールのユーザーガイドを参照してください。

SDL-PR

SDL インポートはプレーン SDL-PR 仕様をサポートします。この仕様は UML モデルにインポートされ、元の SDL 仕様のセマンティックスも可能な限り保存されます。

例 **227: SDL-PR** テキスト仕様 ——————————————————————————

```
system MySystem;
  signal ok;
  channel c from b to env with ok; endchannel;
  block b;
    procedure out_ok; returns Boolean;
      start;
        output ok;
        return true;
    endprocedure;
    signalroute sg from p to env with ok;
    connect c and sg;
    process p(1,1); signalset;
      dcl v Boolean;
      start;
        task v := "not"(call out_ok);
        stop;
```

```
    endprocess;
  endblock;
endsystem;
```

CIF

SDL インポートは、CIF（SDL 対応の Common Interchange Format）コメントを使用して表現された SDL 仕様のグラフィック情報のインポートもサポートします。CIF のインポートによりグラフィック レイアウト（シンボル、ライン、テキスト属性の位置やサイズなど）が可能な限り保存され、生成された UML モデルとプレゼンテーション要素は見慣れたものになります。

例 **228:** グラフィック情報が付加された **SDL** 仕様

```
/* CIF SystemDiagram */
/* CIF Page 1 (1900,2300) */
/* CIF Frame (150,150),(1600,2000) */
/* CIF PackageReference (175,25),(200,100) */
/* CIF Specific SDT Version 1.0 */
/* CIF Specific SDT OriginalFileName
'C:¥Telelogic¥SDL_TTCN_Suite4.4¥sdt¥examples¥demongame¥DemonG
ame.ssy' */
/* CIF Specific SDT Page 1 Scale 100 AutoNumbered */
system DemonGame;
/* CIF CurrentPage 1 */
/* CIF Text (600,250),(200,100) */
SIGNAL
Newgame, Probe, Result, Endgame,
Win, Lose, Score(Integer), Bump;
/* CIF End Text */
/* CIF Channel (150,475),(600,475) */
/* CIF TextPosition (525,425) */
/* CIF TextPosition (275,500) SignalList1 */
/* CIF Arrow1Position (262,475) */
channel C1 from env to GameBlock with Newgame, Probe,
Result, Endgame;
endchannel C1;
```

サポートされるツールとバージョン

SDL インポートは、以下のツールとバージョンをサポートします。

- Tau SDL Suite バージョン 4.4 以降からのインポート
- ObjectGeode バージョン 4.2 からのインポート

SDL Suite からのインポート

正確な SDL バージョンと固有の拡張に関する SDL のサポート レベルは、SDL Suite の SDL Analyzer により提供されるサポートと同様です。大文字と小文字を区別して、インポートを実行します。

大文字と小文字を区別する SDL システムに変換する方法については、SDL Suite のユーザーガイドを参照してください。（User's Guide、CPP2SDL Migration Guide、Migration Guidelines、Update to case-sensitive SDL）。

ObjectGeode からのインポート

ObjectGeode では、正確な SDL バージョンと固有の拡張に関する SDL のサポート レベルは、ObjectGeode SDL_API で提供された SDL サポートと同様です。

ObjectGeode の詳細については、ユーザーガイドを参照してください。

SDL インポートの起動

SDL インポートは、Tau GUI から呼び出します。SDL インポート機能は、対応するツールの固有アドインを使用して有効にします。通常、インポート ウィザードを最初に使用するとき、適切なアドインが自動で有効になります。

手動で有効にすることもできます。この場合、［ツール］メニューから［カスタマイズ］ダイアログを選択し、適切なアドインを選択します。

注記

SDL インポートを起動するには、ワークスペースを開いて、プロジェクトを作成しておく必要があります。そうしないとアドインが有効になりません。

SDL Suite と ObjectGeode のどちらの場合も、アドインから SDL インポートを起動できます。ツールごとに指定されたアドインがあります。必ずデータのインポートに使用するツールに対応したアドインを選択してください。

- TauSDL Suite の SDL インポートを使用するには、**SDL96Import** アドインを選択します。
- ObjectGeode の SDL インポートを使用するには、**OGSDLImport** アドインを選択します。

SDL から UML への変換ルール

ストラクチャとスコープ

最上位レベルの定義

SDL 仕様から、以下の最上位レベルの定義がインポートされます。

- パッケージ定義
- システム定義
- 型ベースのシステム定義

最上位レベルのシステムは、アクティブ クラスを含むパッケージにインポートされます。システム スコープ内のすべてのシグナル、シグナル リスト、型、同義語、およびリモート定義に定義されたインターフェイスは、パッケージに挿入されます。システム自体は、その他のすべての定義とともに、パッケージ内のアクティブ クラスにマッピングされます。完全修飾子が生成された場合、インポートされた定義がすべて挿入されたターゲット パッケージから開始されます。

例 **229:** システム定義 ——————————————————————————————

SDL

```
system MySystem;
  signal ok;
  ...
  output ok to p;
  output ok to system MySystem/block b1 p;
  ...
endsystem;
```

UML

```
package MySystem {
  active class MySystem {
    ...
    ^ p.ImportedSDLDefinitions::MySystem::ok();
    ^
ImportedSDLDefinitions::MySystem::MySystem::b1_T::p.ok();
    ...
  }
  signal ok;
}
```

——

コマンドラインから SDL インポートを実行する場合、アクティブ クラスをポイントするビルド アーティファクトも生成されます。完全修飾子が生成された場合、グローバル スコープから開始されます。ビルド アーティファクトは、コマンドラインから SDL インポートを実行する場合のみ生成されます。

例 **230:** コマンドラインからのシステム定義のインポート

UML

```
package MySystem {
  active class MySystem {
    ...
    ^ p.::MySystem::ok();
    ^ ::MySystem::MySystem::b1_T::p.ok();
    ...
  }
  signal ok;
}
artifact Build <<manifest>> dependency to MySystem::MySystem
{ }
```

SDL パッケージは、対応する SDL パッケージかインポートされたエンティティを含む UML パッケージにインポートされます。

例 **231:** パッケージ定義

SDL

```
package MyPackage;
  ...
endpackage;
```

UML

```
package MyPackage {
  ...
}
```

パッケージの use 節は、<<access>> ステレオタイプを持つ依存関係に変換されます。システム定義の use 節が、アクティブ クラスを持つ作成されたパッケージに適用されます。

例 **232:** パッケージの使用

SDL

```
package MyPackage;
  ...
endpackage;

use MyPackage;
system MySystem;
  ...
endsystem;
```

UML

```
package MyPackage {
  ...
}
```

```
package MySystem <<access>> dependency to MyPackage {
  active class MySystem
   MyPackage {
  ...
  }
}
```

型ベースのシステム定義は、指定した親システムの型を継承するアクティブ クラスにインポートされます。

例 **233:** タイプ ベースのシステム定義

SDL

```
system MySystem : MySystemType;
```

UML

```
active class MySystem : MySystemType {
}
```

システム型、ブロック型、プロセス型

システム型、ブロック型、プロセス型は、インポート後の SDL エンティティのスコープ定義に対応する定義がインポートされた、アクティブ クラスにインポートされます。

例 **234:** ブロック型とプロセス型

SDL

```
block type bt1;
  process type p1;
    ...
  endprocess type;
  ...
endblock type;
```

UML

```
active public class bt1 {
  active public class p1 {
     ...
  }
  ...
}
```

プロセス型の継承は、クラス生成と状態機械生成にマッピングされます。

例 **235:** プロセス型の継承

SDL

```
process type ptype; fpar i Integer;
  start virtual;
    stop;
endprocess type;

process type psuper inherits ptype; fpar k Integer;
  start redefined;
    stop;
endprocess type;
```

UML

```
active public class ptype {
  virtual statemachine ptype( Integer fpar_i) {
    virtual start {
      ptype::i = fpar_i;
      stop;
    }
  }
  Integer i;
}

active public class psuper : ptype {
  virtual statemachine psuper( Integer fpar_k) :
ptype(Integer) {
     redefined start {
       ptype::i = fpar_i;
       psuper::k = fpar_k;
       stop;
     }
  }
  Integer k;
}
```

サービス型

サービス型はサポートされていません。

ブロック インスタンス、プロセス インスタンス

ブロック インスタンスとプロセス インスタンスは、管理するブロック型またはプロセス型に対応するアクティブ クラスの属性に変換されます。インポート後の属性の集約は「合成」に設定されます。

例 **236:** ブロック インスタンスとプロセス インスタンス

SDL

```
block type bt1;
  ...
endblock type;
block b1 : bt1;
```

UML

```
active public class bt1 {
```

```
...
}
part bt1 b1;
```

プロセス インスタンスの初期数は、UML の属性の初期数にマッピングされます。

例 **237:** プロセス インスタンスの初期数

SDL

```
process type p1;
  ...
endprocess type;
process pi(10):p1;
process pii(1):p1;
```

UML

```
active public class p1 {
  ...
}
part p1 pi / 10;
part p1 pii / 1;
```

プロセス インスタンスの最大数が指定されている場合、属性の多重度にインポートされます。多重度の範囲は「0..max」です。ここで、max はインスタンスの最大数です。

例 **238:** プロセス インスタンスの最大数

SDL

```
process type p1;
  ...
endprocess type;
process pi(10,20):p1;
process pii(0,11):p1;
```

UML

```
active public class p1 {
  ...
}
part p1 [0..20] pi / 10;
part p1 [0..11] pii / 0;
```

サービス型インスタンス

サービス型インスタンスはサポートされません。

ブロックとプロセス

ブロックとプロセスは、ブロック インスタンスやプロセス インスタンスと同様に、アクティブ クラスの属性としてインポートされますが、アクティブ クラスはインポート後のブロックとプロセスから作成され、接尾辞「T」（T は Type を表します）の付いた同じ名前が付けられます。親データ型は、インポート後のモデルにインライン型として指定されます。

- ブロックの場合、属性のインスタンスの初期数は 1 に設定されます。
- プロセスの場合、プロセス インスタンスの初期数と最大数が、以下の表に従ってインポートされます。

SDL プロセス	UML 属性	条件
`process p(N, MAX)`	`[0..MAX] p / N`	
`process p(1);` `process p();`	`p / 1`	`create p;` 文がある。
`process p(1);` `process p();`	`[0..*] p / 1`	`create p;` 文が最低 1 つある。
`process p(N);`	`[0..*] p / N`	`N == 0` または `N > 1`

例 **239:** ブロックとプロセス

SDL

```
block b;
  process p(1,1);
    ...
  endprocess;
  ...
endblock;
```

UML

```
part active class b_T {
  part active class p_T {
    ...
  } [0..1] p / 1;
  ...
} b / 1;
```

ブロック サブストラクチャ

ブロック サブストラクチャは、UML にインポートされません。サブストラクチャのすべての定義は、544 ページの例 240 に示すように、サブストラクチャの所有者に対応するアクティブ クラスにインポートされます。

例 **240:** ブロック ストラクチャ

SDL

```
block type bt;
  substructure;
    block bs;
      ...
    endblock;
    ...
  endsubstructure;
endblock type;
```

UML

```
active public class bt {
  part active class bs_T {
    ...
  } bs / 1;
  ...
}
```

チャネル サブストラクチャ

チャネル サブストラクチャは UML モデルに正しくインポートできないので、無視されます。サブストラクチャ要素はチャネルが定義されているのと同じスコープにインポートされます。

プロシージャ

戻り値のないプロシージャは、UML 操作として所有者側クラスにインポートされます。戻り型は void に設定されます。また、SDL プロシージャの内部定義もすべてインポートされます。

例 **241:** プロシージャ

SDL

```
procedure out_ok;
  start;
    output ok;
    return;
endprocedure;
```

UML

```
void out_ok() statemachine {
  start {
    ^ ok();
    return;
  }
}
```

同様に、プロシージャの仮パラメータがインポートされます。パラメータの型が UML クラスにマッピングされている場合、このパラメータには、集約の種類「合成」が設定されます。

例 **242:** プロシージャの仮パラメータ

SDL

```
newtype S struct
  x Integer;
endnewtype;
procedure p; fpar  a Integer, b S;
  start;
    task x := a;
    task y := b;
    return;
  endprocedure p;
```

UML

```
void out_ok() statemachine {
  start {
    ^ ok();
    return;
  }
}
```

すべての操作パラメータは、UML パラメータにインポートされます。パラメータのプロパティを開くと、「In」、「Out」、「In/Out」、「Return」のいずれかの「Direction」特性が表示されます。

プロシージャの戻り値は、操作の仮パラメータ リストから「Return」方向を持つ属性としてインポートされます。

外部プロシージャは、UML の外部操作にインポートされます（587 ページの「外部定義」と比較してください）。

例 **243:** 外部プロシージャ

SDL

```
procedure prd1;
  fpar a Integer, in/out b Integer;
  returns Integer;
external;
```

UML

```
extern Integer prd1( in Integer a, inout Integer b );
```

SDL の特定の SDL プロシージャについては、最後の IN/OUT パラメータがプロシージャの戻り値パラメータとして使用され、値を渡すことができます。SDL モデルにこの種のプロシージャ呼び出しが存在する場合、プロシージャの定義が修正されます。プロシージャのフォーマル リストの最後に、プロシージャ呼び出しの結果の IN パラメータが追加されます。プロシージャ呼び出しが更新され、最後の実パラメータが操作結果を格納する変数として使用できるようになります。通常の呼び出しでは、デフォルト値が生成された最後の IN パラメータに渡されます。

例 **244: IN/OUT** パラメータとしてのプロシージャの戻り値

SDL

```
procedure p1; returns res Integer;
    start;
      task res := res + 1;
      return res;
endprocedure;
```

UML

```
Integer p1( in Integer ) statemachine {
  start {
    res = res + 1;
    return res;
  }
}
```

逆の状況も考えられます。最後の IN/OUT パラメータをプロシージャの戻り値として使用できます。たとえば、以下のように記述できます。

```
task x := call proc(7); instead of call proc(7, x);
```

インポート後の SDL モデルにそのようなプロシージャ呼び出しがある場合、プロシージャ呼び出しが修正されます。

例 **245** 最後 **IN/OUT** パラメータ、戻り値なし

SDL

```
procedure proc1;
fpar in a Integer, in/out b Integer;
start;
  task b := a+10;
  return;
endprocedure;

procedure proc2;
fpar in a Integer; returns Integer;
start;
  return a+10;
endprocedure;

procedure proc3;
fpar in a Integer; returns b Integer;
start;
  task b := a+10;
  return;
endprocedure;

call proc1(7, var1);
call proc2(7, var1);
call proc3(7, var1);

task var2 := call proc1(7);
task var1 := call proc2(7);
task var1 := call proc3(7);
```

UML

```
void proc1(in Integer a, inout Integer b) statemachine {
  start {
    b = a + 10;
    return ;
  }
}
Integer proc2(in Integer a) statemachine {
  start {
    return a + 10;
  }
}
Integer proc3(in Integer a) statemachine {
  Integer b;
  start {
    b = a + 10;
    return b;
  }
}

proc1(7, var1);
var1 = proc2(7);
var1 = proc3(7);

proc1(7, var2);
var1 = proc2(7);
var1 = proc3(7);
```

戻された結果と最後の IN/OUT パラメータの変換は、外部プロシージャには適用されません。

仮想プロシージャと再定義プロシージャ

仮想プロシージャは継承するエンティティで再定義できます。再定義済みプロシージャが明示的な ATLEAST 制約または明示的な継承を持たない場合、対応する仮想プロシージャ（完全修飾子を持つ）への汎化が UML に生成されます。

例 **246:** 再定義済みプロシージャ

SDL

```
process type pt1;
  virtual procedure pr2; fpar in xxx Integer; returns
Natural;
  endprocedure pr2;
endprocess type;

process type pt2 inherits pt1;
  redefined procedure pr2;
  endprocedure pr2;
endprocess type;
```

UML

```
active public class pt1 {
  virtual Natural pr2(in Integer xxx, in Natural result)
```

```
  statemachine {
  }
}

active public class pt2 : pt1 {
  redefined void pr2() : virtprd::pt1::pr2
  statemachine {
  }
}
```

リモート プロシージャ

リモート プロシージャ コール（RPC）はポートを使用して処理されます。RPC は、リモート プロシージャを実装する他のプロセスに対するシグナル送信（リモート プロシージャの実行に対する発信要求）に似ています。

リモート プロシージャには、インターフェイスが作成されます。プロシージャと同じ名前をもちますが、接尾辞「_I」が付加されます。エクスポート後のプロシージャを実現化するクラスでは、このインターフェイスを実現化するポートが追加されます。プロシージャを呼び出すクラスでは、このインターフェイスを必要とするポートが追加されます。リモート プロシージャごとに固有のポートが生成され、rpc_port_<procedure_name> の名前が付けられます。

例 **247:** リモート プロシージャ

SDL

```
remote procedure setv;
  fpar in Integer;

process P1(1,1);
  exported procedure setv; fpar in i Integer;
  endprocedure;
endprocess;

process P2 (1, 1);
  imported procedure setv; fpar in Integer;
    ...
    call setv(10);
    ...
endprocess;
```

UML

```
interface setv_I {
  void setv( in Integer);
}

part active class P1_T {
  public void setv( in Integer i) statemachine {
    ...
  }
  port rpc_port setv in with setv_I;
} [0..1] P1 / 1;
```

```
part active class P2_T {
  ...
  setv(10);
  port rpc_port setv out with setv_I;
} [0..1] P2 / 1;
```

「Exported as p procedure my_x」: この言語要素は、可視性スコープ内の名前「my_x」によるプロシージャの参照を示していますが、エクスポート中に名前が変更され、インポート先の他の名前で参照されました。1 つの UML 操作は 2 つの異なる名前を持つことができないので、インポート後のモデルではプロシージャ名として常に「exported-as」という名前が使われます。情報メッセージが出力されます。

例 **248:** リモート プロシージャとしてエクスポート

SDL

```
remote procedure p; returns Integer;
process p(1,1); signalset;
  exported as p procedure xx; returns Integer;
    ...
  endprocedure;
  ...
  task x := call xx;
  ...
endprocess;
```

UML

```
interface p_I {
  Integer p();
}
part active class p_T {
  public Integer p() comment "procedure xx" statemachine {
    ...
  }
  ...
  x = p();
  ...
} [0..1] p / 1;
```

メッセージ

```
Information: TSI0200: Importing SDL started
Information: TSI0206: Procedure 'xx' has been imported under
the "exported as" name 'p'
Information: TSI0202: Importing SDL completed
```

通信

シグナル、シグナル リスト

SDL シグナルは UML シグナルにインポートされます。シグナルの仮パラメータは、インポート実行時に保持されます。シグナル パラメータが UML クラスにマッピングされている場合、合成集約とともにインポートされます。

SDL シグナル リストは UML シグナル リストにインポートされます。

例 **249:** シグナル リストとシグナル

SDL

```
newtype MyStruct struct
  x Integer;
endnewtype;
signal ok, s(Integer, MyStruct);
signallist sl = ok, s;
```

UML

```
class S {
  public Integer x;
}
signal ok;
signal s(Integer, part MyStruct);
signallist sl = ok, s;
```

シグナルの微調整はインポートされません。

シグナル継承はインポート時に保持されます。

例 **250:** シグナル継承

SDL

```
signal s(Integer);
signal ss inherits s (Natural);
```

UML

```
signal s(Integer);
signal ss(Natural) : s;
```

Gates

SDL ゲートはポートに変換されます。ゲート制約の信号は、インポートされ、着信シグナル（in with...）の「実現化」シグナル セットに挿入されるか、発信シグナル（out with...）の「必須」シグナル セットに挿入されます。

例 **251:** ゲートによるブロック

SDL

```
block type bt2;
  gate g in from bt1 with ok;
  gate g2 out with ok;
  process type p2;
    gate gp in with ok; out with ok;
    ...
  endprocess type;
  ...
endblock type;
```

UML

```
active public class bt2 {
  port g in with ok;
  port g2 out with ok;
  active public class p2 {
    port gp in with ok out with ok;
    ...
  }
  ...
}
```

チャネルとシグナル ルート

チャネルとシグナル ルートは、UML コネクタとしてインポートされます。UML では、コネクタは必ずゲートに接続されます。SDL 内のチャネルまたはシグナル ルートごとに、UML モデルにゲートが生成されます。暗黙的に生成されたゲートには、対応するチャネルまたはシグナルルートと同じ名前が付けられます。同じ名前のゲートとコネクタは異なるスコープに生成されるので、名前の衝突は発生しません。コネクタの端点は以下に従って計算されます。

1. リンクがインスタンスに接続されて、コネクタの端点としてゲートが指定されている場合、コネクタの端点としてインポートされます。ゲート自体は、インスタンス スコープにインポートされます。

例 **252:** ゲートのシグナル

SDL

```
block b;
  signalroute sr from p via g to env with ok;
  ...
  process type pt;
    gate g out with ok;
    ...
  endprocess type;
  process p(1,1):pt;
endblock;
```

UML

```
part active class b_T {
```

```
  connector sr from p.g to c with ok;
  ...
  active public class pt {
    port g out with ok;
    ...
  }
  part pt [0..1] p / 1;
} b / 1;
```

2. チャネルが環境（ダイアグラムの境界線）に接続され、ゲートが指定されている場合、コネクタの端点としてインポートされます。ゲート自体は、インポートされるリンクを含むインスタンスのクラスにインポートされます。ゲートのシグナルもインポートされます。
3. リンクはインスタンスに接続されているが、ゲートが指定されていない場合、暗黙的なゲートがインスタンスの親クラスに挿入され、インポートされるチャネルまたはシグナル ルートの名前が付けられます。リンク付随するシグナルは、作成されたポートに追加されます。

例 **253:** 暗黙的ゲート

SDL

```
block GameBlock;
  ...
  signalroute R5 from Main to Game with GameOver;
  process Main;
    ...
  endprocess;

  process Game;
    ...
  endprocess;

endblock;
```

UML

```
part active class GameBlock_T {
  ...
  connector R5 from Main.R5 to Game.R5 with GameOver;
  part active class Main_T {
    port R5 out with GameOver;
    ...
  } Main / 1;
  part active class Game_T {
    port R5 in with GameOver;
    ...
  } Game / 1;
} GameBlock / 1;
```

4. チャネルが環境（ダイアグラムの境界線）に接続され、これが最も外側のスコープではないが、リンクが周囲のスコープの他のリンクに接続されている場合、暗黙的ゲートがインスタンスの親クラスに挿入され、インポートされたリンクが接続された周囲のエンティティからの外部リンクの名前が付けられます。リンクに付随するシグナルは、作成されたポートに追加されます。
5. チャネルが環境（ダイアグラムの境界線）に接続され、これが最も外側のスコープである場合、暗黙的ゲートがインスタンスの親クラスに挿入され、「Env」という名前が付けられます。リンクに付随するシグナルは、作成されたポートに追加されます。

例 **254:** 親クラスの暗黙的ゲート

SDL

```
system DemonGame;
  channel C1 from env to GameBlock with Newgame, Probe,
Result, Endgame;
  endchannel C1;
  ...
  block GameBlock;
    signalroute R2 from env to Game with Probe, Result;
    connect C1 and R2;
    ...
    ...
  endblock;
endsystem;
```

UML

```
package DemonGame {
  active class DemonGame {
    ...
    connector C1 from Env to GameBlock.C1 with Newgame,
Probe, Result, Endgame;
    port Env in with Newgame, Probe, Result, Endgame;
    part active class GameBlock_T {
      connector R2 from C1 to Game.R2 with Probe, Result;
      port C1 in with Probe, Result, Newgame, Endgame;
      ...
    } GameBlock / 1;
  }
```

UML のコネクタには遅延プロパティがないため、コネクタの遅延プロパティは変換されません。

接続

接続自体はインポートされませんが、ゲート数とリンクの端点に暗黙的に作成されたゲートの名前に影響を及ぼします（552 ページの「チャネルとシグナル ルート」を参照）。

暗黙的な通信ゲートとリンク

SDL では、通信リンクが暗黙的に作成される場合があります。UML にはそのようなルールはないので、インポート中にすべての暗黙的リンクが構築されなければなりません。SDL インポートでは、これは暗黙的ポートによって実行され、コネクタ自体は UML セマンティック アナライザによって追加されます。

SDL のすべての暗黙的シグナル ルートとチャネルに、「io_port」という名前のポートがインポート後のモデルに追加されます。これらのポートは暗黙的コネクタの端点を示します。

例 **255:** 暗黙的シグナル経路

SDL

```
system MySystem;
  signal ok;
  channel c from b1 to env with ok;
  endchannel;
  block b1;
    process p(1,1); signalset;
      start;
        output ok;
        stop;
    endprocess;
  endblock;
endsystem;
```

UML

```
package MySystem {
  active class MySystem {
    connector c from b1.c to Env with ok;
    part active class b1_T {
      part active class p_T {
        statemachine p_T {
          start {
            ^ ok();
            stop;
          }
        }
        port io_port out with ok;
      } [0..1] p / 1;
      port c out with ok;
    } b1 / 1;
    port Env out with ok;
  }
  signal ok;
}
```

振る舞い

状態機械

SDL プロセスは、インポート後のクラスから付けられた名前を持つ UML 状態機械にインポートされます。

例 **256: UML** 状態機械へのプロセス

SDL

```
process p(1,1); signalset;
  start;
    output ok;
    stop;
endprocess;
```

UML

```
part active class p_T {
  statemachine p_T {
    start {
      ^ ok();
      stop;
    }
  }
} [0..1] p / 1;
```

プロシージャは、状態機械本体を持つ UML プロシージャに直接マッピングされます。

例 **257:** プロシージャ

SDL

```
block b;
  procedure out_ok; returns Boolean;
    start;
      output ok;
      return true;
  endprocedure;
  ...
endblock;
```

UML

```
part active class b_T {
  Boolean out_ok(in Boolean result )
statemachine {
    start {
      ^ ok();
      return true;
    }
  }
} b / 1;
```

SDL テキスト プロシージャ定義は、テキスト本文を持つ UML プロシージャにマッピングされます。

例 **258: SDL-PR** のプロシージャ

SDL

```
block b;
  procedure plus1; fpar x Integer; returns Integer;
  start;
    task x := x + 1;
    join L;
    connection L : return x;
  endprocedure;
  ...
endblock;
```

UML

```
part active class b_T {
  Integer plus1(in Integer, in Integer result ) {
    x = x + 1;
    goto L;
    L : return x;
  }
  ...
} b / 1;
```

SDL 状態機械は、UML 状態機械に直接マッピングされます。

例 **259:** 状態機械

SDL

```
process Main;
  dcl GameP Pid;
  start;
    nextstate Game_Off;

  state Game_Off;
      input Newgame;
      create Game;
      task GameP := offspring;
      nextstate Game_On;
  endstate;

  state Game_On;
      input Endgame;
      output GameOver;
      task GameP := Null;
      nextstate Game_Off;
  endstate;
endprocess Main;
```

UML

```
part active class Main_T {
  Pid GameP;
```

```
  statemachine Main_T {
    start {
      nextstate Game_Off;
    }
    state Game_Off;
    state Game_On;
    for state Game_Off;
      input Newgame() {
        Game = new Game_T();
        GameP = offspring;
        nextstate Game_On;
      }
    for state Game_On;
      input Endgame() {
        ^ GameOver();
        GameP = NULL;
        nextstate Game_Off;
      }
  }
} Main / 1;
```

例 **260:** 状態機械

SDL

```
procedure ReadKeys; fpar in NumberKeys Natural, in/out
KeyData KeyArrayType; returns ReadResultType;
  dcl KeyIndex Natural:=1,
      Key Character;
  start;
    set(KeyTimer);
    nextstate WaitKeyStroke;
  state WaitKeyStroke;
    input KeyStroke(Key);
    task KeyData(KeyIndex) := Key;
    decision KeyIndex >= NumberKeys;
      (false):
        set(KeyTimer);
        task KeyIndex:=KeyIndex+1;
        nextstate WaitKeyStroke;

      (true):
        reset(KeyTimer);
        return Successful;
    enddecision;

    input KeyTimer;
    return TimedOut;
  endstate;
endprocedure ReadKeys;
```

UML

```
ReadResultType ReadKeys( in Natural NumberKeys, inout
KeyArrayType KeyData) statemachine {
  Natural KeyIndex = 1;
  Character Key;
  start {
```

```
      set KeyTimer();
      nextstate WaitKeyStroke;
    }
    state WaitKeyStroke;
    for state WaitKeyStroke;
      input KeyStroke(Key) {
        KeyData[KeyIndex] = Key;
        switch (KeyIndex >= NumberKeys){
          case ==false : {
             set KeyTimer();
             KeyIndex = KeyIndex + 1;
             nextstate WaitKeyStroke;
          }
          case ==true : {
             reset KeyTimer();
             return Successful;
          }
        }
        input KeyTimer() {
        return TimedOut;
      }
  }
```

プロシージャ呼び出し

プロシージャ呼び出しは、類似した UML 操作呼び出しにマッピングされます。これらの呼び出しの相違点は、省略されたパラメータの処理方法にあります。SDL では、プロシージャ呼び出しの IN パラメータをすべて省略できます。UML で、省略できるのは、実パラメータ リストの最後のパラメータのみで、しかも、デフォルト値がプロシージャ プロトタイプで定義されている場合のみです。暗黙的なデフォルト値は、インポート後のプロシージャ プロトタイプに挿入されません。代わりに、デフォルト値がプロシージャ呼び出しに、直接、挿入されます。

以下に示す暗黙的デフォルト値が使用されます。

SDL のデータ型	**UML デフォルト値**
`Integer, Duration, Octet`	`0`
`Boolean`	`false`
`Character`	`'0'`
`Charstring`	“”
`Real, Time`	`0.0`
`Pid`	`NULL`
`Bit`	‘¥0’
`Bit_string`	`''B`
`Octet_string`	`'00'H`
リテラルを持つユーザー定義データ型	データ型定義の最初のリテラル

他のすべてのデータ型では、周囲のスコープに対して変数が生成され、プロシージャ呼び出しに渡されます。変数の名前は default_<number> であり、これは初期化されません。

例 **261:** パラメータが省略されたプロシージャ呼び出し

SDL

```
newtype S struct
  x Integer;
endnewtype;
procedure p; fpar  a Integer, b S;
endprocedure p;
...
dcl Svar S;
call p();
call p(1,);
call p(, Svar);
```

UML

```
class S {
  public Integer x;
}
void p(in Integer a, in part S b) statemachine {
}
...
part S Svar;
{
  part S default_0001;
  p(0, default_0001);
}
{
  part S default_0001;
  p(1, default_0001);
}
{
  p(0, Svar);
}
```

create 文

create 文のマッピング方法は、作成されたプロセスの型により異なります。プロセスインスタンスの最大値が 2 以上の場合、これはマルチ インスタンス プロセスであり、create 文は、アクティブクラスのインスタンスをクラス インスタンスに付加する文にマッピングされます。

例 **262:** マルチ インスタンス プロセスの **create** 文

SDL

```
process MyProcess(1,5);
endprocess MyProcess;

create MyProcess;
```

UML

```
part active class MyProcess_T {
} [0..5] MyProcess / 1;

MyProcess.append(new MyProcess_T());
```

プロセスが複数のインスタンスを持てない場合、create 文は、”new” 演算子の直接呼出しにマッピングされます。

例 **263:** シングル インスタンス プロセスの **create** 文

SDL

```
process Game;
endprocess Game;

create Game;
```

UML

```
part active class Game_T {
} Game / 1;

Game = new Game_T();
```

プロセス インスタンスの最大数が外部定数の場合、このプロセスはマルチ インスタンスと見なされ、create 文は append 呼び出しにマッピングされます。

例 **264:** 外部インスタンス番号を持つプロセスの **create** 文

SDL

```
synonym Max Integer = external 'C';
process type pt;
  create p;
endprocess type;
process p(1, Max) : pt;
```

UML

```
const Integer extern Max;
active public class pt {
  p.append(new pt());
}
part pt [0..Max] p / 1;
```

SDL の create 文で省略された実パラメータごとに、対応するデフォルト値が constructor 呼び出しに挿入されます。

例 **265: create** 文

SDL

```
process pr(1,6);fpar a,b Integer; signalset s,ss;
    …
    create pr();
    create pr(,);
    create pr(2);
    create pr(2,);
    create pr(,2);
     …
endprocess;
```

UML

```
part active class pr_T {
    …
    statemachine pr_T( Integer fpar_a = 0,
       Integer fpar_b) {
      …
      pr.append(new pr_T(0, 0));
      pr.append(new pr_T(0, 0));
      pr.append(new pr_T(2, 0));
      pr.append(new pr_T(2, 0));
      pr.append(new pr_T(0, 2));
    }
    …
  } [1..6] pr / 1;
```

output 文

output 文は、直接、UML のシグナル送信文にマッピングされます。シグナル出力で省略されたパラメータは、プロシージャ コールで省略されたパラメータと同じ方法で処理されます。

例 **266:** シグナル出力

SDL

```
signal ss( Integer, Integer );
  ...
  output ss(5);
  output ss(6,);
  output ss();
  output ss(,7);
```

UML

```
signal ss(Integer, Integer);
^ ss(5, 0);
^ ss(6, 0);
^ ss(0, 0);
^ ss(0, 7);
```

output via リストは、対応する UML via リストにインポートされます。同じエンティティが、via リストで複数回参照されている場合でも、インポートされるのは一度だけです。

非ローカル ポートやリンクからの出力は UML でサポートされません。消失した入出力ポートはすべて、暗黙的に UML モデルに生成されます。SDL チャネルからの出力は UML にインポートされません。via 要素の修飾子も無視されます。

例 **267: Output via**

SDL

```
system OutputVia;
  signal ok;
  channel c1 from t to env with ok;endchannel;
  block t;
    signalroute sr1 from p1 to env with ok;
    connect c1 and sr1;
    process p1(1,1);
      start;
      output ok via c1,c1,c1,sr1,sr1;
      output ok via <<system OutputVia/block t>> sr1;
      stop;
    endprocess p1;
  endblock t;
endsystem;
```

UML

```
package OutputVia {
  active class OutputVia {
    part active class t_T {
        part active class p1_T {
            statemachine p1_T {
                start {
                    ^ ok() via sr1;
                    ^ ok() via sr1;
                    stop;
                }
            }
            port sr1 out with ok;
        } [0..1] p1 / 1;
        connector sr1 from p1.sr1 to c1 with ok;
        port c1 out with ok;
    } t / 1;
    connector c1 from t.c1 to Env with ok;
    port Env out with ok;
  }
  signal ok;
}
```

Pid 式への出力の場合、すべての出力項目に完全修飾子が付加されます。出力項目は to 式のコンテキストで解決されるため、修飾子が必要です。ただし、Pid 式の場合はコンテキストがありません。

例 **268: Pid** 式への出力

SDL

```
system OutputTo;
  signal ok;
```

```
  syntype MyPid = Pid endsyntype;
  block t;
    signal pong;
    ...
    process p2(1,1);
      dcl s MyPid;
      ...
      task s := sender;
      output pong to s;
      ...
      output pong to sender;
      ...
    endprocess p2;
  endblock t;
endsystem;
```

UML

```
package OutputTo {
  active class OutputTo {
    part active class t_T {
      signal pong;
      ...
      part active class p2_T {
        MyPid s = NULL;
        ...
        s = sender;
        ^
s.ImportedSDLDefinitions::OutputTo::OutputTo::t_T::pong();
        ...
        ^
sender.ImportedSDLDefinitions::OutputTo::OutputTo::t_T::pong(
);
      } [0..1] p2 / 1;
    } t / 1;
  }
  signal ok;
  syntype MyPid = Pid;
}
```

input 文

input 文は、直接、UML のシグナル受信文にマッピングされます。シグナル入力で省略されたパラメータごとに、周囲のスコープに対して変数が生成され、input 式に渡されます。変数の名前は `default_<number>` で、これは初期化されません。

例 **269:** シグナル入力

SDL

```
signal s(Natural),ss(Integer,Integer);
block b;
  process p(1,6); signalset s,ss;
    ...
    state A;
      input s(), ss();
```

```
        nextstate A1;

      state A1;
        input ss(v);
        nextstate A2;

      state A2;
        input ss(,v);
        nextstate A3;

      state A3;
        input ss(v,);
        stop;
      endstate;
    endprocess;
  endblock;
```

UML

```
  signal s( Natural);
  signal ss( Integer, Integer);
  part active class b_T {
    part active class p_T {
      private Integer default_0006;
      private Integer default_0005;
      private Integer default_0004;
      private Integer default_0003;
      private Integer default_0002;
      private Natural default_0001;
      statemachine p_T() {
        state A; state A1; state A2; state A3;
        for state A;
          input s(default_0001),
                ss(default_0002, default_0003) {
            nextstate A1;
        }
        for state A1;
          input ss(v, default_0004) {
            nextstate A2;
        }
        for state A2;
          input ss(default_0005, v) {
            nextstate A3;
        }
        for state A3;
          input ss(v, default_0006) {
            stop;
        }
      }
    } [0..6] p / 1;
  } b / 1;
```

非形式文

SDL では、非形式文の中で、アクションを非形式に指定できる場合があります。この文は、空文に添付された UML コメントにマッピングされます。非形式テキストは、コメント文字列に挿入されます。前後の引用符は削除されます。

例 **270:** 非形式文

SDL

```
call p(1);
  decision 'Decide?';
  ('my decision') :
    task 'do something';
    task 'do it once again';
    output ok;
    stop;
  else :
    task 'do something else';
    output ok;
  stop;
enddecision;
```

UML

```
p(1);
  switch ("Decide?") {
  case =="my decision" :
  {
    comment "do something";
    comment "do it once again";
    ^ ok();
    stop;
  }
  default :
  {
    comment "do something else";
    ^ ok();
    stop;
  }
  }
```

注記

この非形式文から C コードが生成される場合、ネストされたコメントが生成され、ビルド プロセスのエラーの原因となります。これは、非形式文をタスク シンボルに入れることによって回避できます。

コード生成ディレクティブ

TauSDL Suite は、特殊なコメントを使用する固有の拡張（コード ジェネレータのディレクティブ）をサポートしています。このディレクティブには、生成後の C コードに挿入する C コードも含むことができます。

#CODE ディレクティブ

C コードは、#CODE ディレクティブを使用して、SDL タスクに含めることができます。このディレクティブは構文「/*#CODE C code */」を持ち、SDL タスク文の近くに格納できます（SDL Suite ドキュメントを参照）。このディレクティブは UML の非形式アクションにマッピングされます。ディレクティブ内の C コードはまったく変更されず、UML の非形式アクションに挿入されます。

例 **271: #CODE** ディレクティブ

SDL

```
start;
  task
/*#CODE
testvalue := testvalue + 1;
*/
  testvalue := testvalue+2
/*#CODE
testvalue = testvalue+3;
*/
  ,
/*#CODE
testvalue = testvalue+4;
*/
  testvalue := testvalue+5
/*#CODE
testvalue = testvalue+6;
*/
  ;
/*#CODE
#(testvalue) = #(testvalue)+7;
*/
  task 'stop the kernel' /*#CODE  SDL_Halt(); */;
```

UML

```
start {
  [[ testvalue = testvalue+1; ]]
  [[ testvalue = testvalue+7; ]]
  testvalue = testvalue + 2;
  [[ testvalue = testvalue+3; ]]
  [[ testvalue = testvalue+4; ]]
  testvalue = testvalue + 5;
  [[ testvalue = testvalue+6; ]]
  comment "stop the kernel";
  [[ SDL_Halt(); ]]
}
```

#ADT ディレクティブ

Tau SDL Suite C コード ジェネレータでは、C 言語で記述された演算子やリテラル関数を実装することもできます。これらの実装は、予約語「ENDNEWTYPE（または、ENDSYNTYPE）」の直前に入れることで認識される #ADT ディレクティブとともに挿入されます。

これらのディレクティブも、UML モデルの ADT ステレオタイプにインポートされて格納されます。この ADT ステレオタイプは、［CApplication］カスタマイズ モジュールで定義されたもので、テキスト文字列の属性と ADT ディレクティブの本文を含みます。

ジェネレータの変換によって定義された newtype の演算子の実装に #ADT ディレクティブを使用すると、正しくインポートされません。このデータ型の演算子は、データ型が定義されているスコープにインポートされ、データ型自体は UML のシンタイ

プにインポートされます（581 ページの「ジェネレータの変換、演算子の追加を持つ newtype」と比較してください）。対応する ADT ステレオタイプは、正しく適用されず、コード生成時に無視されます。

例 **272: newtype** のデータ型の実装

SDL

```
process Central;
  NEWTYPE CardBaseType
    Array(CardBaseIndexType, CardRecordType);
    OPERATORS
      Full: CardBaseType -> Boolean;
      Register: CardBaseType, CardType, CardRecordType ->
CardBaseType;
      Validate: CardBaseType, CardRecordType -> ResultType;
/*#ADT(B)
#BODY
...
*/
  ENDNEWTYPE CardBaseType;
  …
endprocess Central;
```

UML

```
part active <<ADT(.bodyText = "#ADT(B)¥n#BODY …",
generateBodyText = true.)>> class Central_T {
  static <<External="true">> Boolean Full( CardBaseType)
comment "Implemented in #ADT #BODY-section";
  static <<External="true">> CardBaseType Register(
CardBaseType, CardType, part CardRecordType) comment
"Implemented in #ADT #BODY-section";
  static <<External="true">> ResultType Validate(
CardBaseType, part CardRecordType) comment "Implemented in
#ADT #BODY-section";
  syntype CardBaseType = Array<CardBaseIndexType,
Value<CardRecordType> >;
  ...
}
```

#INCLUDE ディレクティブ

Tau SDL Suite のディレクティブ「`#INCLUDE 'file'` 」は、SDL アナライザに `'file'` の内容をディレクティブの位置に入れるよう指示します。インクルードされている定義は、以下のいずれかの定義済みファイルからインクルードされた定義を除き、UML にインポートされます。

```
BasicC++Types.pr
BasicCTypes.pr
C++Pointer.pr
CPointer.pr
CharConvert.pr
```

例 **273:** 定義済みファイルのインクルード

SDL

```
system Mysystem;
  /*#INCLUDE 'BasicC++Types.pr'*/
  /*#INCLUDE 'C++Pointer.pr'*/
  /*#INCLUDE 'CharConvert.pr'*/
  newtype Ptr Ref(Integer) endnewtype;
  ...
endsystem;
```

UML

```
active class Mysystem {
  syntype Ptr = CPtr<Integer>;
  ...
}
```

コード ディレクティブの制限事項

`#TYPE`、`#HEADING`、`#BODY` の各セクションを持つ `#CODE` ディレクティブの拡張形式はサポートされません。

コメント内のディレクティブのみ、記述どおりにインポートされます。それ以外（たとえば、`x + #CODE('a')` など）は、サポートされません。

データ型

単純なデータ型

SDL の定義済みの単純なデータ型と操作は、下表のとおり、対応する UML の基本データ型と操作にマッピングされます。定義済みデータ型と定数のマッピングは以下のとおりです。

SDL	UML
boolean	Boolean
true	true
false	false
character	Character
charstring	Charstring
ia5string	IA5string
numericstring	NumericString
duration	Duration

SDL	UML
time	Time
bit	Bit
octet	Octet
string	String
powerset	PowerSet
bag	Bag
pid	Pid
null	NULL
integer	Integer
float	float
visiblestring	VisibleString
printablestring	PrintableString
real	Real
natural	Natural
plus_infinity	PLUS_INFINITY
minus_infinity	MINUS_INFINITY
array	Array
oref	ORef
bit_string	BitString
octet_string	OctetString
own	Own
object_identifier	ObjectIdentifier
any_type	AnyType

定義済みの操作

SDL	UML
/=	!=
//	+
=	==
not	not

SDL	UML
and	and
or	or
xor	xor
mod	mod
rem	rem
in	in
num	num
chr	chr
mkstring	mkstring
length	length
first	first
last	last
substring	substring
append	append
bitstr (that returns bit_string)	String2BitString
bitstr (that returns octet)	String2Octet
bitstr (that returns octet_string)	bitstr
hexstr (that returns bit_string)	HexString2BitString
hexstr (that returns octet)	HexString2Octet
hexstr (that returns octet_string)	hexstr
shiftl	shiftl
shiftr	shiftr
i2o	I2O
o2i	O2I
incl	incl
take	take
makebag	makebag
fix	fix

SDL には、SDL `Bag` と `Powerset` について、`incl` と `del` の 2 通りの形式があります。

```
incl : Itemsort,  Powerset  -> Powerset;
incl : Itemsort,  in/out Powerset;
```

UML では、`incl` 操作と `del` 操作の形式は 1 つのみです。UML では戻り値を持たない短縮形をサポートしていないため、これに対するすべての参照は、2 つ目の（IN/OUT）パラメータを代入式の左辺に入れる形式に変換されます。

例 **274: Powerset** 操作の短縮形 ——————————————————————

SDL

```
newtype Power
  Powerset (Integer)
endnewtype;
task incl(1, pow1);
```

UML

```
syntype Power = PowerSet<Integer>;
pow1 = Power::incl(1, pow1);
```

——

定義済みの C++ 型

単純なデータ型の「`unsigned_char`」は、C++ 型を表現する SDL パッケージの一部です。このデータ型は、SDL の「8 ビット」型の同義語として定義され、UML の「8 ビット」型に直接マッピングされます。

例 **275: SDLunsigned_char** のマッピング ——————————————————

SDL

```
newtype MyType Ref(unsigned_char) endnewtype;
syntype Myunsigned = unsigned_char endsyntype;
dcl x unsigned_char;
dcl y Myunsigned;
dcl p MyType;
dcl i Integer;
task x := p*>;
task y := x;
task i := O2I(y);
task x := I2O(i);
```

UML

```
syntype MyType = CPtr<Octet>;
syntype Myunsigned = Octet;
Octet x;
Myunsigned y;
MyType p;
Integer i;
x = p.GetValue();
y = x;
i = O2I(y);
x = I2O(i);
```

——

回避ポインタを示す SDL の特殊型 “ptr_void” は、UML の型 ‘void*’ にマッピングされます。C 文字列を表現する SDL の特殊型 “ptr_char” は、UML の型 ‘char*’ にマッピングされます。

SDL Suite には、インポート後の enum 型に対して生成される特殊演算子があります。これらの演算子は `EnumToInt` と `IntToEnum` で、整数値と列挙値の間の変換に使用されます。`EnumToInt` と `IntToEnum` は、UML の `cast<>` 演算子の呼び出しにインポートされます。

例 **276: EnumToInt** と **IntToEnum**

SDL

```
dcl i Integer;
dcl e MyEnum := a;
task i := EnumToInt( e );
task e := IntToEnum( i );
```

UML

```
Integer i;
MyEnum e = a;
i = cast<Integer>(e);
e = cast<MyEnum>(i);
```

ptr_void から Ref<T> へ、および ptr_char から Charstring へ、型をキャスティングする場合、特殊なインポート ルールが適用されます。

- cast(Charstring) から ptr_char へのキャスティング：UML には Charstring から ‘char*’ への暗黙の変換があるため、これは省略される。
- cast(ptr_char) から Charstring へのキャスティング：演算子 ‘char*’.ToString(); の呼び出しにインポートされる。
- cast(Ref<T>) から ptr_void へのキャスティング：<'void*'>(CPtr<T>) にインポートされる。
- cast<ptr_void> から Ref<T> へのキャスティング：imported to UML の from 'void*' から CPtr<T> へのキャスティングにインポートされる。

例 **277: SDL** キャスティングの特殊インポート ルール

SDL

```
newtype Ptr Ref(Integer) endnewtype;
dcl Invar  Integer;
dcl Outvar Ptr;
dcl Vs     ptr_void;
dcl pc     ptr_char;
dcl str    Charstring;
task Outvar := &Invar,
     Vs := cast(Outvar),
     Outvar := cast(Vs);
task str := 'My String',
     pc := cast(str),
     str := cast(pc);
```

UML

```
syntype Ptr = CPtr<Integer>;
Integer Invar;
Ptr Outvar;
'void*' Vs;
'char*' pc;
Charstring str;
Outvar = GetAddress<Integer>(Invar);
Vs = cast<'void*'>(Outvar);
Outvar = cast<Ptr>(Vs);
str = "My String";
pc = str;
str = pc.ToString();
```

下表は、SDL で外部 C++ 型をサポートするために使用されるデータ型と演算子の特殊マッピングのルールをまとめたものです。

SDL	**UML**
データ型	
unsigned_char	Octet
ptr_void	'void*'
演算子	
EnumToInt	cast<>
IntToEnum	cast<>

UML の定義済み型との衝突

SDL のユーザー定義型は、インポート後に UML の定義済み型と競合する場合があります。つまり、インポート後のユーザー定義型が、UML の定義済みパッケージである TTDCppPredefined、TTDRTTypes、Predefined のいずれかと同じ名前になることがあります。

SDL インポート時に競合する型を処理する、[Import user-defined types that conflict with UML predefined types] というオプションがあります。このオプションは、デフォルトで「false」つまり競合する型をインポートしない設定になっています。以下のように、競合状態が発生すると情報メッセージが表示されます。

```
Information: TSI0209: SDL user-defined type conflicts with
UML predefined type 'short int' from package
'TTDCppPredefined'. SDL type is NOT imported.
```

このオプションを「true」に設定すると、競合する型がインポートされ、警告が生成されます。

```
Warning : TSI0210: SDL user-defined type conflicts with UML
predefined type 'short int' from package 'TTDCppPredefined'.
Consider revising imported type.
```

構造体データ型

SDL の構造体型は、public フィールドを持つ UML クラスにマッピングされます。

例 **278:** 構造体

SDL

```
newtype n struct
  f1 Integer;
  f2 Boolean;
endnewtype;
```

UML

```
class n {
  public Integer f1;
  public Boolean f2;
}
```

注記

構造化データ型の比較演算子 v1 == v2（v1 と v2 は構造体の値）は、参照を比較します。ただし、構造化型のネストされた値は除外されます。

オプション フィールドは、`[0..1]` の 多重度を持つ public フィールドにマッピングされます。SDL のフィールド名リストが定義された 1 つの型を持つフィールドは、UML の複数ライン上の複数のフィールド宣言にマッピングされます。インライン フィールドの初期化は、UML の類似の初期化にマッピングされます。

例 **279:** オプション フィールドを持つ構造体

SDL

```
newtype str struct
  a, b Integer;
  c    Boolean optional;
  d    str2 optional;
  e    Charstring := 'IBM Rational';
  f    arr3 := (. 11 .);
endnewtype;
```

UML

```
class str {
  public Integer a;
  public Integer b;
  public Boolean [0..1] c;
  public part str2 [0..1] d;
  public Charstring e = "IBM Rational";
  public arr3 f = arr3 (.11.);
}
```

SDL の `choice` は、public フィールドを持つ UML の `choice` にマッピングされます。

例 **280: SDL choice**

SDL

```
newtype c choice
  f1 Integer;
  f2 Boolean;
endnewtype;
```

UML

```
choice c {
  public Integer f1;
  public Boolean f2;
}
```

特定フィールドの初期化を含む choice を作成する、一般的な UML 形式は以下のとおりです。

```
<choice name> (. <field name> = <expression> .)
```

Example 281: 初期化を含む **SDL choice**

SDL

```
newtype choice_type choice
  a Integer;
  b Boolean;
endnewtype;

dcl an_int choice_type:= a : 1;
/* Initialization of field a by value 1 */
```

UML

```
choice choice_type
  {
    public Integer a;
    public Boolean b;
  }

choice_type an_int = choice_type (. a = 1 .);
```

他の SDL データ型は、UML のデータ型にマッピングされます。演算子もすべてマッピングされます。これらのデータ型の基本操作プロトタイプも挿入されます（リテラルのみを含むデータ型は除く）。

例 **282:** 演算子

SDL

```
newtype dummy
  operators
    op: Charstring -> Charstring;
  operator op; fpar i Charstring; returns Charstring;
    start;
```

```
        return call p( i // 'def' );
    endoperator;
  endnewtype;
```

UML

```
  datatype dummy {
    static <<IsQuery="true">> Charstring op( Charstring i) {
      return p(i + "def");
    }
    extern public static dummy '¥='( dummy, dummy);
    extern public static Boolean '=='( dummy, dummy);
    extern public static Boolean '!='( dummy, dummy);
  }
```

SDL の **newtype** と **syntype** のデフォルト値

SDL の newtype と syntype のデフォルト値は、SDL インポートでサポートされます。型のデフォルト値が指定されている場合、この型の変数とフィールドの初期化に使用され、実値が省略されている場合はプロシージャ呼び出し、シグナルなどに渡されます。

例 **283: SDL** の **newtype** と **syntype** のデフォルト値

SDL

```
  newtype lights
    literals red, yellow, green
    default  yellow
  endnewtype;

  syntype __lights = lights
    default  green
  endsyntype;

  newtype X struct
    x Integer;
    y lights;
  endnewtype;

  signal ok(lights);

  procedure out_ok; fpar ii Integer, ss __lights; returns
  Boolean;
    start;
      output ok;
      return true;
  endprocedure;

  dcl v Boolean;
  dcl s1 lights;
  dcl s2 lights := red;
  dcl s3 __lights;
  dcl s4 __lights := red;

  input ok(s1);
```

```
task v := "not"(call out_ok(1));
```

UML

```
enum lights { red, yellow, green }
syntype __lights = lights;
class X {
  public Integer x;
  public lights y = yellow;
}
signal ok( lights );
Boolean out_ok( in Integer ii, in __lights ss, in Boolean
result )
statemachine {
  start {
    ^ ok(yellow);
    return true;
  }
}

lights s1 = yellow;
lights s2 = red;
__lights s3 = green;
__lights s4 = red;
input ok(s1) {
v = not out_ok(1, green, false);
```

存在のチェック

SDL の型では、いくつかの暗黙的な `Present` 演算子を使用できます。これらの演算子で、choice およびオプションの構造体フィールドの存在を確認できます。

オプションの構造体フィールドに適用された暗黙的なブール演算子 `<field name>present(<variable>)` は、以下の形式の式にインポートされます。

```
<variable>.<field name> != NULL
```

例 **284:** オプションの構造体フィールドの存在を確認

SDL

```
newtype seq struct
  a Integer;
  r Integer;
  z Integer optional;
endnewtype

dcl  v seq;
dcl  b Boolean;
task b := zPresent( v );
```

UML

```
class seq {
  public Integer a;
  public Integer r;
  public Integer [0..1] z;
}
```

```
part seq v;
Boolean b;
b = (v.z != NULL);
```

choice フィールドに適用された暗黙的なブール演算子 `<field name>present(<variable>)` は、以下の形式の演算子呼び出しに変換されます。

```
<variable>.IsPresent("<field name>")
```

例 **285: choice** フィールドの存在を確認

SDL

```
newtype cho choice
  x Integer;
  y Boolean;
endnewtype;

dcl  myc cho;
dcl  b Boolean;
task b := yPresent( myc );
```

UML

```
choice cho {
  public Integer x;
  public Boolean y;
}

part cho myc;
Boolean b;
b = myc.IsPresent("y");
```

choice フィールドと同じ名前のリテラルを持つ暗黙的な `<ChoiceName>present` 型は、UML の明示的な列挙型 `<ChoiceName>_enum` にマッピングされます。All references to `<ChoiceName>present` へのすべての参照は、`<ChoiceName>_enum` という名前になります。

例 **286: choice** 型の **present** フィールドを処理する列挙型

SDL

```
newtype MyChoice choice
  bfield Boolean;
  ifield Integer;
endnewtype;
```

UML

```
choice MyChoice {
  public Boolean bfield;
  public Integer ifield;
}
enum MyChoice_enum {
  bfield,
```

```
    ifield
  }
```

暗黙的な choice フィールド present は、<ChoiceName>present で型指定されており、アクティブな choice フィールドの名前を持ちます。present フィールドをチェックすることにより、どのフィールドがアクティブなのかを判断できます。

暗黙的な choice フィールド「present」を参照するブール式「=」と「/=」は、対応する <variable>.IsPresent("<field name>") のチェックにマッピングされます。

例 **287:** 暗黙的フィールド **present** のチェック

SDL

```
newtype cho choice
  x Integer;
  y Boolean;
endnewtype;

dcl  myc cho;
dcl  b Boolean;
task b := ( myc!present /= y );
task b := ( myc!present = y );
```

UML

```
choice cho {
  public Integer x;
  public Boolean y;
}

part cho myc;
Boolean b;
b = (not myc.IsPresent("y"));
b = (myc.IsPresent("y"));
```

最後の変換は、choice 型にユーザー定義の明示的フィールド「present」がない場合のみ、適用されます。

暗黙的 choice フィールド present に対するその他の参照は、すべての choice フィールドをチェックする条件式にマッピングされます。

例 **288:** 暗黙的 **choice** フィールド「**present**」に対する参照

SDL

```
newtype MyChoice choice
  bfi Boolean;
  ifi Integer;
endnewtype;

dcl ch MyChoice;
dcl v  MyChoicepresent;
task ch!ifi := 10;
```

```
task v := ch!present;
task v := bfi;
```

UML

```
choice MyChoice {
  public Boolean bfi;
  public Integer ifi;
}
enum MyChoice_enum {
  bfi,
  ifi
}
MyChoice ch;
MyChoice_enum v;
ch.ifi = 10;
v = ch.IsPresent("bfi") ? bfi : ifi;
v = MyChoice_enum::bfi;
```

ジェネレータ

定義済みの SDL ジェネレータは、インポートされます。ジェネレータは対応するテンプレート インスタンス化にマッピングされます。すべてのテンプレート関数呼び出し（`mkstring` など）は、型名修飾子の接頭辞が付けられます。

例 **289:** ジェネレータを持つ **newtype**

SDL

```
newtype t
  String( Integer, Empty )
endnewtype;

task v2 := mkstring( 1 );
```

UML

```
syntype t = String<Integer>;
v2 = t::mkstring(1);
```

ジェネレータの変換、演算子の追加を持つ **newtype**

ジェネレータ変換を持つ newtype の演算子は、データ型が定義されているスコープにマッピングされます。

例 **290:** ジェネレータ変換を持つ **newtype**

SDL

```
newtype t4
  array( Index, Integer)
  operators
```

```
        op4 : Integer -> Integer;
    endnewtype;
```

UML

```
public static <<IsQuery="true">> Integer op4( Integer) ;
syntype t4 = Array<Index, Integer> ;
```

#ADT ディレクティブを使用してこのような newtype の演算子の 1 つを実装すると、対応するステレオタイプがインポート後のモデル内の間違ったノードに適用されるため、コード生成時に無視されます。

リテラルを持つ **newtype**

リテラルのみを含む SDL の newtype は、UML の enum 型にマッピングされます。ユーザー定義リテラルは、すべての使用箇所で型修飾子の接頭辞が付けられます。

例 **291:** リテラルを持つ **newtype** ——————————————————————

SDL

```
newtype SomeType
  literals Some1, Some2, Some3
    default Some2
endnewtype;

newtype SomeType2
  inherits SomeType
endnewtype;

dcl var0 SomeType;
dcl var1 SomeType2;
task var1 := Some1;
```

UML

```
enum SomeType {
  Some1,
  Some2,
  Some3
}
datatype SomeType2 : SomeType
{
  public <<External="true">> SomeType2( SomeType);
  public static <<External="true">> SomeType2 '¥='(
SomeType2, SomeType2);
  public static <<External="true">> Boolean '=='( SomeType2,
SomeType2);
  public static <<External="true">> Boolean '!='( SomeType2,
SomeType2);
}
SomeType var0 = SomeType::Some2;
SomeType2 var1;
var1 = SomeType2::Some1;
```

ジェネレータ **Ref** と演算子「**&**」および「*****」

C++ ポインタを表現する定義済みの generator Ref は、`TTDCppPredefined` プロファイルから UML のテンプレート データ型 `CPtr<T>` にインポートされます。

演算子「`&`」は、CPtr データ型で定義されたテンプレート関数 `GetAddress` の呼び出しにマッピングされます。演算子「`*`」は、`GetValue` 関数の呼び出しにマッピングされます。

例 **292:** 演算子「**&**」および「*****」のマッピング

SDL

```
newtype Iptr
  Ref(Integer);
endnewtype;

dcl a Integer;
dcl p Iptr;
task p := &a;
task a := p*;
```

UML

```
syntype Iptr = CPtr<Integer>;
Integer a;
Iptr p;
p = GetAddress<Integer>(a);
a = p.GetValue();
```

データ型の継承

データ型の継承は、相応にマッピングされます。親型への継承チェーン変換演算子内のすべての親は、継承データ型に生成されます。

例 **293:** データ型の継承

SDL

```
newtype SomeType
  literals Some1, Some2, Some3
    default Some2
endnewtype;

newtype SomeType2
  inherits SomeType
endnewtype;

newtype SomeType3
  inherits SomeType2
endnewtype;
```

UML

```
enum SomeType {
  Some1,
```

```
  Some2,
  Some3
}
datatype SomeType2 : SomeType
{
  public <<External="true">> SomeType2( SomeType);
  public static <<External="true">> SomeType2 '¥='(
SomeType2, SomeType2);
  public static <<External="true">> Boolean '=='( SomeType2,
SomeType2);
  public static <<External="true">> Boolean '!='( SomeType2,
SomeType2);
}
datatype SomeType3 : SomeType2
{
  public <<External="true">> SomeType3( SomeType2);
  public <<External="true">> SomeType3( SomeType);
  public static <<External="true">> SomeType3 '¥='(
SomeType3, SomeType3);
  public static <<External="true">> Boolean '=='( SomeType3,
SomeType3);
  public static <<External="true">> Boolean '!='( SomeType3,
SomeType3);
}
```

変数

SDL の変数宣言は、明示的に指定された可視性を持たない、UML の変数宣言にマッピングされます。

例 **294:** 変数

SDL

```
dcl v1 Integer;
dcl v2 Boolean;
```

UML

```
Integer v1;
Boolean v2;
```

viewed/revealed 変数

viewed 変数または revealed 変数はポートを使用して処理されます。viewed 変数（name）ごとに、インターフェイス `<name>_var_I` が作成されます。暗黙的なポート `revealed_port_<name>` は、revealed 変数が定義されているクラスに生成されます。revealed 変数がアクセスされた各クラスに対して、暗黙的なポート `viewed_port_<name>` が生成されます。revealed 変数を持つクラス インスタンスとこの変数を参照するクラス インスタンスの間に、暗黙的なコネクタが生成されます。

例 **295: viewed/revealed** 変数

SDL

```
block b;
  process p;
    dcl revealed j boolean;
  endprocess;

  process type pt2;
    viewed j boolean;
    dcl i boolean;
    task i := view (j);
  endprocess type;

  process p2(1,1) : pt2;
endblock;
```

UML

```
part active class b_T {
  interface j_var_I {
    Boolean j;
  }
  part active class p_T {
    public Boolean j;
    port revealed_port_j in with j_var_I;
  } p / 1;
  active public class pt2 {
    port viewed_port_j out with j_var_I;
    Boolean i;
    virtual statemachine pt2 {
      start {
        i = j;
        stop;
      }
    }
  }
  part pt2 [0..1] p2 / 1;
  connector p2_p_j_var from p2.viewed_port_j to
p.revealed_port_j with j_var_I;
} b / 1;
```

リモート変数

リモート変数は、リモート プロシージャと同じようにポートを使用して処理されます。

一般的なルール

プロセス仮パラメータ

プロセス型の仮パラメータは、状態機械の仮パラメータとしてインポートされ、プロセス型に対応するクラス定義に追加されます。パラメータ名には接頭辞「`fpar_`」が付けられます。

プロセス型の仮パラメータごとに、クラス定義に属性が追加され、これらの属性は状態機械の最初のステートメントで初期化されます。

例 **296:** プロセス型の仮パラメータ

SDL

```
process type pt; fpar a, b Integer;
    …
  endprocess type;
```

UML

```
active public class pt {
    …
virtual statemachine pt( Integer fpar_a,
                                    Integer fpar_b) {
start {
      pt::b = fpar_b;
      pt::a = fpar_a;
    …
   }
  …
  }
  Integer a = 0;
  Integer b = 0;
}
```

プロセス型の仮パラメータは、暗黙的なプロセス型に対して作成されたインライン クラスに追加されます。

修飾子

修飾子は UML 修飾子に変換されますが、エンティティ（たとえば、システムまたはプロセス）の種類の指定は失われます。

仮想性

仮想性は、インポートされます。

コメント

SDL のコメントは、UML モデルにインポートされます。

SDL のテキスト コメント（コメント '...';）は、インポート後に対応するモデルに添付されます。

テキスト図およびシンボルからの C 形式のコメントはインポートされますが、常に場所が確保されているわけではないので、消失することがあります。タスク シンボルからの C 形式のコメントは、ほとんど消失します。ただし結果は、インポート元の SDL（SDL Suite または ObjectGeode）によって異なります。

例 **297: SDL Suite** からの **C** 形式のコメントのインポート

SDL

```
signal sig1 /* 1 */;
signal sig2(Integer) /* 2 */;
signal sig3(Boolean /* 3 */) /* 4 */; /* 5 */
```

UML

```
signal sig1 comment " 1 ";
signal sig2( Integer);
signal sig3( Boolean) comment " 5 ";
```

例 **298: ObjectGeode** からの **C** 形式のコメントのインポート

ObjectGeode からのインポートでは、コメントが要素定義の中に記述されている場合インポート先の **UML** でこの定義の前または後ろに挿入されます。

SDL

```
/* 1 */
signal /* 2 */ s1;
signal s2 /* 3 */; /* 4 */
```

UML

```
/* 1 */
/* 2 */
signal s1;
/* 3 */
/* 4 */
signal s2;
```

外部定義

external 'C' または external 'C++' とマークされた SDL 定義は、SDL インポートでは UML にインポートされません。このような定義は C/C++ のインポートで UML にインポートする必要があります。

定義が無視されると、以下のような情報メッセージが表示されます。

```
Information: TSI0211: External 'C' or 'C++' procedure
'DEK_c_AlgorithmPriority' has been ignored. Use C/C++ import
to map C and C++ definitions to UML.
```

external とマークされた他の SDL 定義は、UML の外部定義にインポートされます。

SDL のインポートに関する制限事項

一般的な SDL 言語の制限事項

セマンティック上正確な SDL

SDL 仕様をインポートするには、完全かつセマンティック上正確な SDL システムでなければなりません。不完全または不正確な仕様はインポートできません。

大文字 / 小文字の区別

SDL Z.100 勧告には、大文字と小文字の区別に関して二つの特色があります。SDL の最近のバージョンでは大文字と小文字の区別が導入され、バージョン 4.4 以降の SDL Suite などの SDL ツールを使用して、大文字と小文字を区別する SDL システムを設計できるようになりました。

UML は大文字と小文字を区別する言語です。SDL のインポートは、大文字と小文字を区別して動作します。このため、インポート元の SDL システムにも大文字と小文字を正確に区別できる機能が必要です。この機能がなければ、定義と参照が一致しない、あるいはキーワードが適切に認識されないなどの不具合が生じて、UML がセマンティック上不正確なものになることもあります。

つまり、以下のようになります。

- SDL の定義と定義の参照は、SDL システム全体で、大文字と小文字を区別して記述しなければならない。
- SDL のキーワードは、一貫して大文字または小文字を使い分けなければならない。

このため、大文字と小文字の区別に一貫性を持たせるため、インポートを開始する前にシステムを更新しなければならない場合があります。大文字と小文字を区別するための SDL システム変換ツールについては、SDL ツールのドキュメントを参照してください。

サポートされない SDL 言語の概念

仮想プロセス型の定義

仮想プロセス型の定義はインポートされますが、対応するアクティブ クラスでの仮想性の使用は、UML ではサポートされません。これは UML モデルの制約です。

仮想プロセス型は、仮想アクティブ クラスにインポートされます。すべての内容が使用可能ですが、セマンティック チェッカはサポートされない仮想型としてエラー メッセージを出力します。

ガイドライン

この型の仮想性を使用しないよう、インポート後のモデルの構造を手動で変更します。各モデル別の対策を行います。

例 **299:** 仮想プロセス型

SDL

```
block type BType1;
  virtual process type pt1;
  endprocess type pt1;
  process p1(1,1) : pt1;
endblock type BType1;
block type BType2 inherits BType1;
  redefined process type pt1;
  endprocess type pt1;
endblock type BType2;
```

UML

```
active public class BType1 {
  active virtual public class pt1 {
  }
  part pt1 [0 .. 1] p1 / 1;
}
active public class BType2 : BType1 {
  active redefined public class pt1 {
  }
}
```

メッセージ

```
Class pt1: Error: TSC2023: The use of virtual, redefined or
finalized types is not supported.
```

サービス、サービス型、サービス型インスタンス

サービスはインポートされません。可能な対処法として、サービスをプロセスなどインポート可能な SDL の構成要素に置き換え、インポート後のモデルの構造を手動で変更します。1 つの方法として、すべてのサービス（たとえばプロセスとしてインポート）を 1 つの UML 状態機械に結合できます。ただし、各モデルに適した方法を考慮しなければなりません。

例 **300:** サービス

SDL:

```
PROCESS P1(1,1); SIGNALSET Sig1R, Sig2R, SigInternal;
  SERVICE S1; SIGNALSET Sig1R;
  ENDSERVICE;
  SERVICE S2; SIGNALSET Sig2R, SigInternal;
  ENDSERVICE;
ENDPROCESS;
```

インポート メッセージ:

```
Warning    : TSI0224: cdtserv01.sdl(25,11): Services are not
imported to UML. Service 'S1' has been ignored.
Warning    : TSI0224: cdtserv01.sdl(35,11): Services are not
imported to UML. Service 'S2' has been ignored.
```

変更済み SDL（サービスはプロセスに置換）：

```
/* PROCESS P1(1,1); SIGNALSET Sig1R, Sig2R, SigInternal; */
PROCESS S1; SIGNALSET Sig1R;
ENDPROCESS;
PROCESS S2; SIGNALSET Sig2R, SigInternal;
ENDPROCESS;
/* ENDPROCESS; */
```

UML （手動で変更）：

```
part active class S1_T {
  statemachine S1_T {
    …
  }
  port io_port in with Sig1R out with SigInternal, Sig1;
} S1 / 1;
part active class S2_T {
  statemachine S2_T {
    …
  }
  port io_port in with Sig2R, SigInternal out with Sig2, ok;
} S2 / 1;
```

Create this

言語要素 "create this" はサポートされません。

シグナルの微調整

シグナルの微調整はインポートされません。

チャネル サブストラクチャ

チャネル サブストラクチャはインポートされません。すべてのサブストラクチャ要素は、チャネルが定義されているスコープにインポートされます。

遅延コネクタ

コネクタの遅延プロパティはサポートされません。これは UML ではコネクタがシグナルの遅延を行わないためです。UML では、コネクタの遅延プロパティをユーザー定義ステレオタイプで示すことができます。インポート後のチャネルに対応するステレオタイプを手動で適用することもできますが、既存のコード ジェネレータでは処理されません。

マクロ

マクロは UML にインポートされません。

select

SDL の select はサポートされません。

SDL の構成要素「Select if (<Boolean expression>)」は条件付きコンパイルに対してアナログです。Selected 本体は、ブール式が true の場合のみ SDL モデルに現れます。この種の処理は SDL のインポート前、および選択したエンティティのインポート後に行われます。Select 構成要素自体は、UML にインポートされません。

例：

SDL:

```
system S;
select if ( false );
  signal ss1;
endselect;
select if ( true );
  signal ss2;
endselect;
  signal ok;
  …
endsystem;
```

UML：

```
package S {
  active class S {
    …
  }
  signal ss2;
  signal ok;
}
```

UML では、たとえば select-if 構成要素が条件付きコンパイル機能を使用して再デザインされます。

名前の衝突

同じ名前を持つ SDL の構成要素が生成後の UML で名前の衝突を起こすことがあります。そのような衝突の例として、リテラルと他のエンティティの例を以下に示します。

例 **301:** 名前の衝突 ———————————————————————————————

SDL

```
signal DR;
newtype IPDUType
  literals CR, CC, DR, DT, AK;
endnewtype IPDUType;
```

UML

```
signal DR;
enum IPDUType {
  CR, CC, DR, DT, AK
}
```

———

SDL ではリテラルと他の定義の名前は正しく解決されますが、UML では名前の衝突が生じてエラーになります。

ネストされたスコープに同じ名前を持つ 2 つのエンティティが存在する場合も、同様の状況になります。

遷移オプション

SDL の遷移オプションはサポートされません。SDL の遷移オプションは、UML の分岐文としてインポートできます。ただし、SDL Suite Analyzer では遷移オプションはサポートされません。

Include 式

SDL の Include 式はサポートされません。

公理

公理は UML にインポートできません。SDL の公理情報は UML では示すことができません

RPC 遷移

「`Input procedure my_x;`」: このコードは、リモート プロシージャが呼び出されたときのみ遷移が行われることを意味します。これは UML ではサポートされません。

配列のインライン初期化

配列要素のインライン初期化は、SDL ではできますが、UML では許可されません。初期化はインポートされますが、モデルは正確なものになりません。

例 **302:** 配列のインライン初期化 ——————————————————————————

SDL

```
newtype St1 struct
  aSt1 Integer;
  bSt1 Charstring;
endnewtype;

newtype A Array (Integer, St1)
endnewtype;

block B1;

process P1 (1, 1);
signalset;

dcl var A := (. (. 10, 'hello' .) .);
start;
```

UML

```
class St1 {
  public Integer aSt1;
  public Charstring bSt1;
}

syntype A = Array<Integer, St1>;


part active class B1_T {

   part active class P1_T {


private A a = A (. St1 (.10, "hello".) .);
   statemachine P1_T {
      start {
...
```

生成後の UML モデルから以下のエラー メッセージが出力されます。

```
InstanceExpr 'St1 (.10, "hello".)': Error: TNR0047: Failed to
find definition of St1 (while looking for InstanceOf).
InstanceExpr 'St1 (.10, "hello".)': Error: TNR0051: Context
resolution failed for InstanceOf in 'InstanceExpr <unnamed>'.
InstanceExpr 'St1 (.10, "hello".)': Error: TNR0034: Failed to
find InstanceOf of InstanceExpr (by ref:St1).
```

UML では、配列を宣言された場所で初期化することはできません。このため、配列変数の初期化を手動で適切な場所に追加する必要があります。これは、状態機械の先頭などで行うことができます。

例 **303:** 配列変数の初期化 ———————————————————————————

```
statemachine P1_T {
   start {
     a[0] = St1 (.10, "hello".);
```

修飾子としてのプロシージャ

UML では、プロシージャを修飾子として使用することはできません。そのような修飾子は、UML にインポートされず、警告メッセージが出力されます。

例 **304:** プロシージャ修飾子 ———————————————————————————

SDL

```
process HelloWorld;
  dcl i Integer := 5;

  procedure op; fpar in a Integer; returns Integer;
    newtype XXX
      literals lit_a, lit_b, lit_c;
    endnewtype;
    dcl i Integer;
```

```
      dcl x XXX;
      start;
        task { procedure op i := a + 1; };
        task process HelloWorld i := a;
        task x := procedure op/type XXX lit_b;
        return i;
    endprocedure op;
    ...
endprocess HelloWorld;
```

UML

```
part active class HelloWorld_T {
  Integer i = 5;
  Integer op(in Integer a) statemachine {
    enum XXX {
      lit_a, lit_b, lit_c
    }
    Integer i;
    XXX x;
    start {
      i = a + 1;
      HelloWorld_T::i = a;
      x = XXX::lit_b;
      return i;
    }
  }
  ...
} HelloWorld / 1;
```

メッセージ

```
Information: TSI0200: Importing SDL started
Warning: TSI0204: Procedure can not be used as qualifier,
qualifier 'op' for identifier 'i' is not imported
Warning: TSI0204: Procedure can not be used as qualifier,
qualifier 'op' for identifier 'XXX' is not imported
Information: TSI0202: Importing SDL completed
```

ERROR 式

ERROR 式は UML でサポートされていないので、正しくマッピングされません。SDL インポートでは、SDL の ERROR キーワードに対して ERROR 識別子が生成され、警告メッセージが出力されます。インポート後の ERROR 識別子は、UML モデルでは解決されずにセマンティック エラーとなるので、適切なコードで置き換える必要があります。

例 **305: ERROR** 式の警告メッセージ

メッセージ

```
Information: TSI0200: Importing SDL started
Warning: TSI0205: ERROR term is not supported in UML, expect
semantic errors on imported 'ERROR' ident
Information: TSI0202: Importing SDL completed
```

SDL Suite からのインポートにおける制限事項

実装を持つデータ型のインポート

SDL Suite で作成された以下の SDL-PR ファイル内のデータ型を参照する #ADT ディレクティブによって SDL をインポートすると、データ型の実装は無視されます。

```
access.pr
byte.pr
cm_pidlist.pr
file.pr
idnode.pr
list1.pr
list1_noname.pr
list2.pr
list2_noname.pr
random.pr
unsigned.pr
unsigned_long.pr
```

#ADT 実装を持つデータ型を以下のいずれかのファイルからインポートしようとすると、#ADT ディレクティブによって生成されるコードはインポートされません。情報メッセージが表示されます。

```
Information: TSI0215: D:¥SDLImport¥file.pr(123,5): #ADT
implementation for 'TextFile' datatype from predefined
'file.pr' has not been imported.
```

演算子の実装に使用される **#ADT** ディレクティブ

ジェネレータの変換によって定義された newtype の演算子の実装に #ADT ディレクティブを使用すると、正しくインポートできません。このデータ型の演算子は、データ型が定義されているスコープにインポートされます。データ型自体は、UML のシンタイプにインポートされます。対応する ADT ステレオタイプは、正しいノードに適用されず、コード生成時に無視されます。

コネクタの不正な再宣言

複数のシグナル ルートが同じコネクタに接続されている場合、SDL ダイアグラムのグラフィック上同じ接続ポイントを使用する必要があります。SDL で 1 つのコネクタに対して繰り返し「グラフィック宣言」を行うと、UML のセマンティックエラーを誘引します。これがコネクタの不正な再宣言です。

これを回避するためには、同じ接続ポイントを使用します。

複数のコメント シンボルによる構文エラー

複数のコメント シンボルを同じ SDL シンボルに接続すると、SDL Suite が構文上不正確な CIF ファイルを生成するため、インポートできなくなります。

この回避策は 2 つあります。

- 複数のコンテンツを 1 つのコメント シンボルに結合する。
- SDL Suite をバージョン 4.4.6 以降にアップグレードする。

［Include graphical SDT References］オプションの選択解除

［Include graphical SDT references］オプションを選択すると、CIF コメントが適切に処理されず、グラフィック要素（ダイアグラム、シンボルなど）が作成されません。

SDL アナライザは大文字と小文字を区別して動作

インポート元の SDL が大文字と小文字を区別できるないと、SDL アナライザ（UML への変換前にソース SDL をチェック）が動作しません。SDL アナライザによりエラーが報告された場合、SDL を修正する必要があります。

ヒント

バージョン 4.4 以降の SDL Suite では、SDL システムを大文字と小文字を区別できるよう変換するツールとスクリプトが提供されています。「Migration Guidelines」章の「Update to case-sensitive SDL」セクションを参照してください。

ObjectGeode からのインポートにおける制限事項

暗黙的演算子の明示的な使用

modify、extract、make など、構造型のための暗黙的な定義済み演算子の明示的な使用は、SDL 標準の演算子定義本体の内部でのみ使用できます。そのような演算子が他の場所で使用されている場合、596 ページの例 306 のように使用されている場合を除き、UML に正しくインポートされません。

例 **306: modify** 演算子の明示的な使用

代入式の左辺にある名前が言語要素 <name>modify(); の最初の引数と同じ場合、modify 演算子はフィールド式の代入にインポートされます。

```
value.name = <expression>;
```

SDL

```
newtype N struct
  a Integer;
  b Boolean;
endnewtype;
dcl m N;
task m := amodify(m,1);
```

UML

```
class N {
  public Integer a;
  public Boolean b;
}
part N m;
```

```
m.a = 1;
```

例のセクション

DemonGame（SDL Suite からインポート）

この例では、DemonGame システムの SDL-PR インポートとダイアグラム インポートについて説明します。ダイアグラムは、テキスト SDL 仕様内に挿入された SDL Suite CIF コメントからインポートされます。

例 **307: DemonGame** の **SDL-PR** 例

```
system DemonGame;

  SIGNAL Newgame, Probe, Result, Endgame, Win, Lose,
Score(Integer), Bump;
  channel C1 from env to GameBlock with Newgame, Probe, Result,
Endgame;
  endchannel C1;
  channel C2 from GameBlock to env with Win, Lose, Score;
  endchannel C2;
  channel C3 from DemonBlock to GameBlock with Bump;
  endchannel C3;

  block GameBlock;
    SIGNAL GameOver;
    signalroute R2 from env to Game with Probe, Result;
    signalroute R1 from env to Main with Newgame, Endgame;
    signalroute R5 from Main to Game with GameOver;
    signalroute R3 from Game to env with Win, Lose, Score;
    signalroute R4 from env to Game with Bump;

    process Main;
      DCL GameP Pid;
      start;
        nextstate Game_Off;
      state Game_Off;
        input Newgame;
        create Game;
        task GameP := offspring;
        nextstate Game_On;
      endstate;
      state Game_On;
        input Endgame;
        output GameOver;
        task GameP := Null;
        nextstate Game_Off;
      endstate;
    endprocess Main;

    process Game;
      DCL Count Integer;
      start ;
        task Count:=0;
        nextstate Losing;
      state Losing;
        input Probe;
        output Lose;
        task Count:= Count-1;
        nextstate -;
        input Bump;
        nextstate Winning;
      endstate;
```

```
      state Winning;
        input Bump;
        nextstate Losing;
        input Probe;
        output Win;
        task Count:=Count+1;
        nextstate -;
      endstate;
      state *;
        input Result;
        output Score(Count);
        nextstate -;
        input GameOver;
        stop ;
      endstate;
    endprocess Game;

    connect C1 and R2, R1;
    connect C2 and R3;
    connect C3 and R4;

  endblock GameBlock;

  block DemonBlock;
    signalroute R1 from Demon to env with Bump;

    process Demon;
      timer T;
      start ;
        set(now+1,T);
        nextstate Generate;
      state Generate;
        input T;
        output Bump;
        set(now+1, T);
        nextstate -;
      endstate;
    endprocess Demon;

    connect C3 and R1;
  endblock DemonBlock;
endsystem DemonGame;
```

例 **308: DemonGame** 例のインポート結果の **UML** モデル

```
package DemonGame {
  active class DemonGame {
  part active class GameBlock_T {
    signal GameOver;

    part active class Main_T {
      Pid GameP;
      port R1 in with Newgame, Endgame;
      port R5 out with GameOver;
      statemachine Main_T {
        start {
          nextstate Game_Off;
        }
        state Game_Off;
        state Game_On;
        for state Game_Off;
          input Newgame() {
            Game = new Game_T();
```

```
            GameP = offspring;
            nextstate Game_On;
          }
        for state Game_On;
          input Endgame() {
            ^ GameOver();
            GameP = NULL;
            nextstate Game_Off;
          }
      }
    } Main / 1;

    part active class Game_T {
      private Integer Count;
      port R2 in with Probe, Result;
      port R5 in with GameOver;
      port R3 out with Win, Lose, Score;
      port R4 in with Bump;
      statemachine Game_T {
        start {
          Count = 0;
          nextstate Losing;
        }
        state Losing;
        state Winning;
        for state Losing;
          input Probe() {
            ^ Lose();
            Count = Count - 1;
            nextstate -;
          }
          input Bump() {
            nextstate Winning;
          }
        for state Winning;
          input Bump() {
            nextstate Losing;
          }
          input Probe() {
            ^ Win();
            Count = Count + 1;
            nextstate -;
          }
        for state *;
          input Result() {
            ^ Score(Count);
            nextstate -;
          }
          input GameOver() {
            stop;
          }
      }
    } Game / 1;

    connector R2 from C1 to Game.R2 with Probe, Result;
    port C1 in with Probe, Result, Newgame, Endgame;
    connector R1 from C1 to Main.R1 with Newgame, Endgame;
    connector R5 from Main.R5 to Game.R5 with GameOver;
    connector R3 from Game.R3 to C2 with Win, Lose, Score;
    port C2 out with Win, Lose, Score;
    connector R4 from C3 to Game.R4 with Bump;
    port C3 in with Bump;
  } GameBlock / 1;

  part active class DemonBlock_T {
    part active class Demon_T {
```

```
      timer T;
      port R1 out with Bump;
      statemachine Demon_T {
        start {
          set T() = now + 1;
          nextstate Generate;
        }
        state Generate;
        for state Generate;
          input T() {
            ^ Bump();
            set T() = now + 1;
            nextstate -;
          }
      }
    } Demon / 1;
    connector R1 from Demon.R1 to C3 with Bump;
    port C3 out with Bump;
  } DemonBlock / 1;
  connector C1 from Env to GameBlock.C1 with Newgame, Probe,
Result, Endgame;
  port Env in with Newgame, Probe, Result, Endgame out with Win,
Lose, Score;
  connector C2 from GameBlock.C2 to Env with Win, Lose, Score;
  connector C3 from DemonBlock.C3 to GameBlock.C3 with Bump;
}
  signal Newgame;
  signal Probe;
  signal Result;
  signal Endgame;
  signal Win;
  signal Lose;
  signal Score(Integer);
  signal Bump;
}
```

エラー メッセージ

概要

SDL-PR インポートと SDL CIF インポート時に、メッセージのソースが複数存在します。

- SDL アナライザによって出力されるメッセージ。SDL 構文セマンティックスに照らし合わせて、インポート後の SDL ファイルの正確性をチェックします。SDL アナライザからのメッセージ コードには接頭辞「TIL」が付きます。
- CIF アナライザによって出力されるメッセージ。SDL 仕様の CIF コメントの構造と内容をチェックします。
- SDL インポートおよび CIF インポート時に出力されるメッセージ。メッセージ コードに接頭辞「OGC」が付いたメッセージは、ObjectGeode のインポートを示します。

SDL インポート時に出力されるメッセージは、すべて出力ウィンドウの［スクリプト］タブに表示されます。

SDL インポートおよび CIF インポート時のメッセージ

SDL インポートおよび CIF インポート時に出力されたメッセージ

コード	テキスト	コメント
TSI19019	Unable to get license	OG と SDL Suite
TSI19001	Invalid option: <string>	OG と SDL Suite
TSI19002	The option is set twice: <string>	OG と SDL Suite
TSI19003	Option requires additional argument: <string>	OG と SDL Suite
TSI19011	There should be at least one input file.	OG と SDL Suite
TSI19012	The errors occurred during profile(s) loading.	OG と SDL Suite
TSI19013	Only one of -check, -quickcheck and -fullcheck could be specified	OG と SDL Suite
TSI19014	-semanPath could not be used with -quickcheck or -fullcheck	OG と SDL Suite
TSI19015	-semanPath or -semanConfig could be used only together with check or transform options	OG と SDL Suite
TSI19016	Output file(s) format is invalid or undefined: '<name>	OG と SDL Suite

コード	テキスト	コメント
TSI19017	Input file(s) format is invalid or undefined: '<name>	OG と SDL Suite
TSI19018	Failed to load predefined package.	OG と SDL Suite
TSI19004	Cannot identify input file format: there is not extension.	OG と SDL Suite
TSI19005	Non-standard extension '<string>' in input file name (use -inputFormat to specify file type)	OG と SDL Suite
TSI19061	Cannot identify output file format: there is not extension.	OG Suite と SDL Suite
TSI19062	Non-standard extension '<string>' in output file name (use -outputFormat to specify file type)	OG と SDL Suite
TSI19007	Error in Tcl script: <string>	OG と SDL Suite
TSI19009	Can not open output file.	OG と SDL Suite
TSI0204	Warning - Procedure can not be used as qualifier, qualifier '%s' for identifier '%s' is not imported	このメッセージは、修飾子としてのプロシージャを使用した場合に出力されます。
TSI0205	Warning - ERROR term is not supported in UML, expect semantic errors on imported 'ERROR' ident	このメッセージは、インポートした SDL で ERROR 式を使用するたびに出力されます。
TSI0206	Information - Procedure '%s' has been imported under the “exported as” name '%s'	このメッセージは、「exported-as」という名前を持つリモート プロシージャをインポートした場合に出力されます。
TSI0209	Information - SDL user-defined type conflicts with UML predefined type '%s' from package '%s'. SDL type is NOT imported.	このメッセージは、UML の定義済み型との衝突および SDL のユーザー定義型がある場合に出力されます。
TSI0210	Warning - SDL user-defined type conflicts with UML predefined type '%s' from package '%s'. Consider revising imported type.	このメッセージは、UML の定義済み型との衝突および SDL のユーザー定義型がある場合に出力されます。
TSI0211	Information - External 'C' or 'C++' `%s` '`%s`' has been ignored. Use C++ Importer to map C and C++ definitions to UML.	外部定義を参照。

コード	テキスト	コメント
OGC0517	(Information, Warning, Error, `FatalError`) - OG native message: “`%s`”, line `%d, %s: %s`.	ObjectGeode 固有のメッセージが出力されます。最初の 2 つのパラメータは処理されたファイルおよびライン番号です。最後の 2 つのパラメータは固有のエラーコードと固有のエラーメッセージです。重大度は固有のエラーメッセージの重大度に依存します。ObjectGeode の説明書を参照してください（Appendix E Geodecheck, E.3.3）。
OGC0518	Information - Converting OG to SDT started.	このメッセージは、ObjectGeode から SDT への変換が開始したときに出力されます。
OGC0519	Information - Converting OG to SDT completed.	このメッセージは、ObjectGeode から SDT への変換が終了したときに出力されます。

12

Rose のインポート

この章では、Rational Rose で作成したモデルを Tau にインポートする方法について説明します。この機能の目的は、Rose モデルを Tau に移行することです。

注記

Rose モデルをインポートするには、Rose インポート ツールを使用します。Rose モデルに XMI インポートを使用した場合、よい結果が得られないので推奨できません。

概要

Rose インポータは、Rational Rose モデルを Tau に移行するための機能です。

Rose インポータの主な機能は以下のとおりです。

- Rose モデル ファイルの全部または一部をインポートする。
  - レイアウト情報を含むモデルの要素とダイアグラム
- インポータが Rose モデル ファイルを直接操作するので、Rational Rose をインストールする必要がない。
- Rose モデルの U2 ファイルへのマッピングを完全にカスタマイズできる。
- インポータは対話形式（607 ページの「インポートの開始」を参照）、またはコマンド ラインから（615 ページの「コマンドライン ユーザー インターフェイス」を参照）実行できる。

インポートの開始

Rose モデルを Tau にインポートするには、以下の手順を行います。

- Tau を起動し、新しいプロジェクトを作成するか、または既存のプロジェクトを使用する。
- ［モデル ビュー］でモデル ノードを選択する。
- ［ファイル］メニューの［インポート］コマンドを選択して、［インポート ウィザード］を起動する。
- ［Rational Rose からのインポート］を選択し、［OK］をクリックする。
- Rose インポート ウィザードの各ページをクリックして進む。

Tau を起動せずにモデルをコマンド ウィンドウからインポートするには、コマンドライン ユーザー インターフェイスを使用します。

Rose インポート ウィザード

このセクションでは、Rose インポート ウィザードのさまざまなステップを詳細に説明します。

Rose インポート ウィザードのステップ 1

インポート プロセスのステップ 1 では、インポートするモデル ファイルおよびその解釈方法を指定します。

- モデル ファイルの指定
- モデル ファイルのロケール
- モデル参照ファイルのロード
- 作成した U2 ファイルのデフォルトの格納場所

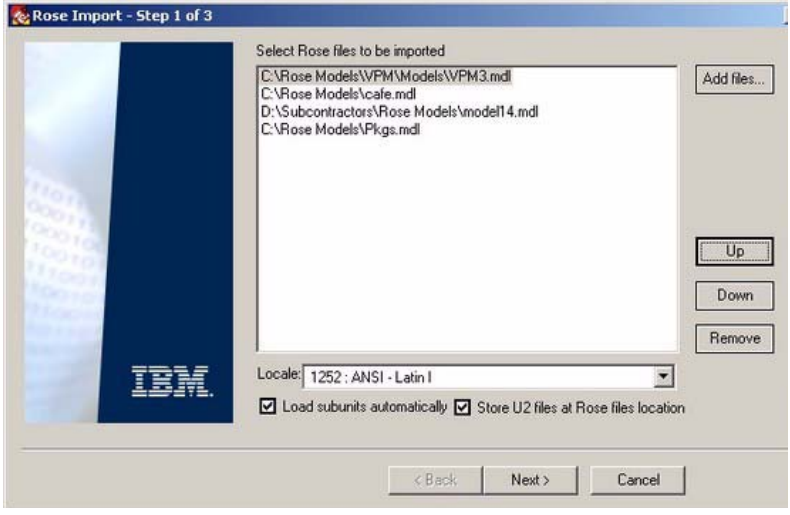


図 172: Rose インポート ウィザードのステップ 1

すべてのモデル ファイル情報を指定して、［Next］ボタンをクリックします。インポート ウィザードは指定したすべてのファイルを解析します。

注記

大きなモデル ファイルの解析には時間がかかります。解析を途中で止める場合は、［キャンセル］ボタンを使用します。

モデル ファイルの指定

最初のステップの主な目的は、インポート対象のファイルを決めることです。一連のファイルがダイアログの中央に一覧表示されます。［Add files...］、［Up］、［Down］および［Remove］の各ボタンを使用して表示内容を変更できます。

- ［Add files...］ボタンをクリックすると、標準のファイルを開くダイアログが表示される。このダイアログで複数のファイルを選択できます。選択したファイルはファイル リストの最後に追加されます。
- ［Up］ボタンまたは［Down］ボタンをクリックすると、リスト内で選択しているファイルが上または下に移動する。
- ［Remove］ボタンをクリックすると、選択しているファイルがリストから削除される。

モデル ファイルのロケール

ロケール リストボックスで、インポート対象のファイルのコード化を設定します。デフォルトでは、現在のシステム ロケールが使用されます。別のロケールを使用して作成したファイルをインポートする場合は、ローカライズした定義を Tau に正しく反映するために、このリスト ボックスから適切なロケールを選択する必要があります。

モデル参照ファイルのロード

［Load subunits automatically］オプションを使用して、外部ファイルへの参照を検索するパーサーの振る舞いを指定します。このオプションを選択すると、パーサーは指定パスでファイルを検索し、その解析を試みます。このオプションを選択しないと、ユニットはロードされません。詳細については、次のセクションを参照してください。

作成した **U2** ファイルのデフォルトの格納場所

［Store U2 files at Rose files location］オプションを使用して、インポートで作成される U2 ファイルのデフォルトの格納場所をカスタマイズできます。

このオプションを有効にすると、デフォルトで、作成されるすべての U2 ファイルは対応する Rose モデル ファイルと同じ場所に格納されます。このオプションを無効にすると、すべての U2 ファイルは現在の Tau プロジェク ファイルのフォルダに作成されます。

例 **309** デフォルトのファイル格納場所

以下の場所に格納されている `my_tau_proj` というプロジェクトで作業しているとします。

```
C:¥Tau¥my_tau_projects¥my_tau_proj.ttp
```

例として、以下の 3 つのフォルダに格納されている 3 つのファイルで構成される、Rose モデルをインポートします。

```
C:¥my_rose_prj¥a¥first.mdl
C:¥my_rose_prj¥b¥second.cat
```

```
C:¥my_rose_prj¥c¥third.cat
```

デフォルトのファイル格納場所を設定してこれらのファイルをインポートすると、3 つの U2 ファイルが作成され、対応する Rose モデル ファイルと同じ以下のディレクトリに格納されます。

```
C:¥my_rose_prj¥a¥first.u2
C:¥my_rose_prj¥b¥second.u2
C:¥my_rose_prj¥c¥third.u2
```

このオプションが無効の場合は、以下のようになります。

```
C:¥Tau¥my_tau_projects¥first.u2
C:¥Tau¥my_tau_projects¥second.u2
C:¥Tau¥my_tau_projects¥third.u2
```

注記

このオプションでは、デフォルトのファイル格納場所のみを指定します。ファイルのマッピングは次のインポートのステップで完全にカスタマイズできます（U2 ファイル マッピングの指定を参照）。

Rose インポート ウィザードのステップ 2

インポート プロセスのステップ 2 では、実際にインポートする必要があるモデルのパートと、またその結果の格納方法を指定します。

- インポートするモデルのパートを指定
- U2 ファイル マッピングの指定
- 紛失ファイルの特定
- 仮想パスの設定
- ログ ファイル

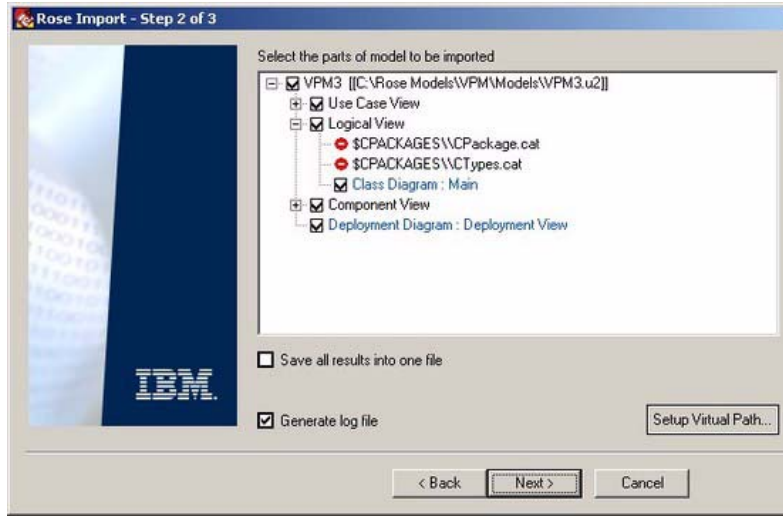


図 173: Rose インポート ウィザードのステップ 2

インポートの対象とファイルのマッピング方法を指定し、［Next］をクリックします。モデルがインポートされ、指定したとおりに、U2 ファイルに保存されます。

インポートするモデルのパートを指定

解析されたモデルはツリー形式で表示されます。ツリー表示では、モデルの全体構造ではなく、ファイル マッピングの指定に必要な詳細のみが表示されます。

各ツリー項目の左には、インポート プロセスでの各ツリー項目の状態を示すチェックボックスが表示されます。

- チェックボックスが空の場合、エンティティとその子のいずれもインポートされない。
- チェックボックスにグレーの印が付いている場合、エンティティがインポートされる。エンティティの子は一部（またはまったく）インポートされません。
- チェックボックスに黒い印が付いている場合、エンティティとその子のすべてがインポートされる。

赤い丸は、ファイルの参照が未解決であることを示しています。紛失したファイル参照を解決する方法については、紛失ファイルの特定を参照してください。

紛失ファイルの特定

インポートしたモデル ファイルに、自動検索できない外部ファイルへの参照が含まれている場合、ツリー表示のノードの前に赤い丸印が表示されます。紛失しているファイル参照を解決する方法には、以下の 2 通りがあります。

- ファイルを手動検索する。

  ツリー表示のノードをダブルクリックし、標準のファイルを開くダイアログを使用してファイルの場所を特定します。場所が特定されると、モデル ファイルが解析され、ツリー表示が更新され、新しく収集されたデータが表示されます。
- 仮想パスを設定する（仮想パスの設定を参照）。

U2 ファイル マッピングの指定

ツリー表示を使用して、インポートしたモデルの U2 ファイル マッピングを指定することもできます。ツリー表示内の黒色の各ノードを U2 ファイルに関連付けることができます。

ノードを U2 ファイルに関連付けるには、以下の手順を行います。

- ツリー表示内のノードをダブルクリックし、ファイルの格納場所を参照します。
- ファイル名を入力し、［保存］クリックします。

U2 ファイルにマッピングされているノードごとに、次のフォーマットに従ってノード名にファイル名が付加されます。

```
nodeName [[U2 file name]]
```

デフォルトでは、Rose モデル ファイルごとに 1 つの U2 ファイルがマッピングされます。同じ名前が付けられ現在のプロジェクト ディレクトリに格納されます。

ダイアグラムは、個別の u2 ファイルに入れることはできません。ツリー ノードはツリー内で青色で強調表示されます。

仮想パスの設定

仮想パスは Rational Rose でサポートされます。これは、新しい環境に対応してプロジェクトを簡単に再構成することで、コラボレーションに対応するためです。

仮想パスを含むモデルをインポートする場合、インポータは Rational Rose のレジストリ項目を使用してパスの解決を試みます。その解決に失敗した場合、たとえば、Rational Rose がインポートを実行するマシンにインストールされていない場合など、未解決の仮想パスを持つユニットは、モデルのツリー表示に未実装のユニットとして表示されます。例については、611 ページの図 173 を参照してください。このようなユニットの位置を指定するために、2 つの方法が用意されています。最初の方法は、前述のように、ユニット ツリー項目をダブルクリックし、ダイアログ内のファイルを選択する方法です。もう 1 つの方法は、仮想パスの変数値を定義する方法です。これは、複数の未実装のユニットが同じ仮想パス変数を使用する場合に有用です。

仮想パスのマッピングを変更するには、以下の手順を行います。

- ［Setup Virtual Path］ボタンをクリックする。

- 以下の説明に従って、仮想パス変数を追加または編集する。
- ［OK］をクリックする。

613 ページの図 174 に［仮想パス設定］ダイアログを示します。このダイアログには、最初は Rational Rose のレジストリ データから抽出可能なすべての変数が含まれていますが、インポータはこれらを以下のファイルに保存します。

```
%APPLICATION DATA%¥IBM¥Rational¥Shared¥tau_virtpath.cfg
```

注記

Rose がインストールされている場合でも、Rose インポータで［仮想パス設定］を変更しても Rose には反映されません。

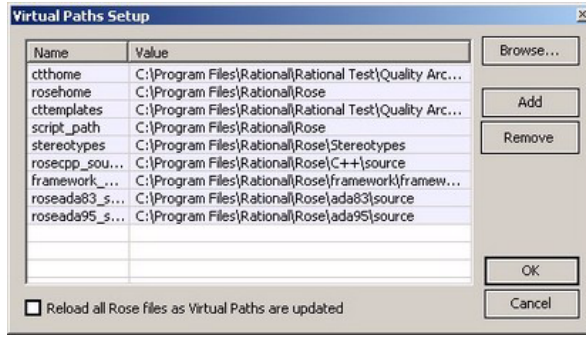


図 174: ［仮想パス設定］ダイアログ

仮想パスのマッピングを追加するには、以下の手順を行います。

- ［Add］をクリックして、変数名を入力する。
- 新しく追加された変数を選択し、［Browse...］ボタンをクリックして仮想パス フォルダの場所を特定する。

既存の仮想パス変数の名前を変更するには、以下の手順を行います。

- 変数の名前カラムをダブルクリックし、新しい名前を入力する。

既存の仮想パス変数のフォルダを変更するには、以下の手順を行います。

- 変数を選択し、［Browse...］ボタンをクリックして新しい仮想パス フォルダの場所を特定する。

既存の仮想パス変数を削除するには、以下の手順を行います。

- 変数を選択し、［Remove］ボタンをクリックする。

［Reload all Rose files as Virtual Paths are updated］オプションにより、仮想パスを変更したとき、解析したモデル ファイルを再び解析するかどうかを指定します。このオプションを選択すると、すべてのモデル ファイルは新しい仮想パスを使って再解析されます。

注記

すべてのモデル ファイルを再解析すると、カスタマイズした U2 ファイル マッピングは失われます。

ログ ファイル

［Generate log file］を選択すると、すべてのエラーメッセージと警告ログが、最初の mdl ファイルと同じ名前で拡張子 `.log` であるログファイルに書き込まれます。このログファイルは、現在の Tau プロジェクト ディレクトリに格納されます。

Rose インポート ウィザードのステップ 3

ステップ 2 を終了すると、インポート プロセスが開始されます。このプロセスが実行されている間、ダイアログ最下部のプログレス バーに、インポートの進捗状況が表示されます。エラー メッセージと警告が一覧表示されます

注記

警告メッセージとエラーメッセージは、ログ ファイルに書き込むようにも設定できます。

［キャンセル］ボタンをクリックするとインポートは中断します。

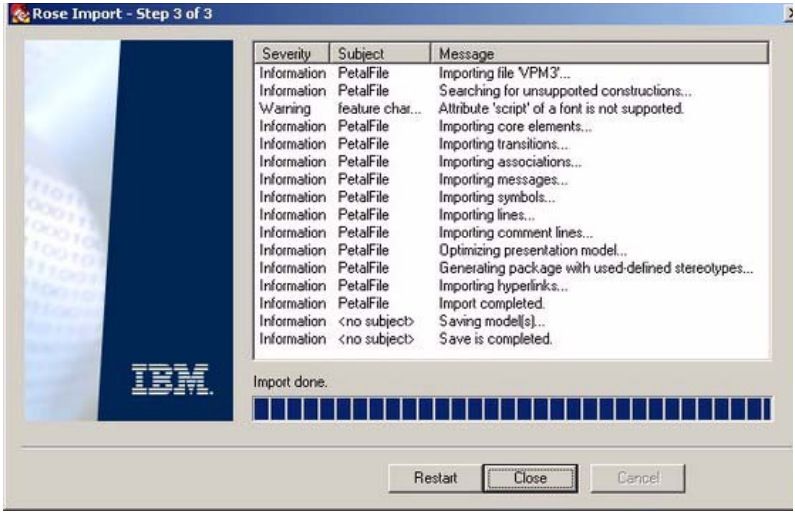


図 175: Rose のインポートの終了

インポートが完了すると［Restart］ボタンと［Close］ボタンが有効になり、インポータの再起動、またはウィザードの終了を実行できます。

［Close］ボタンをクリックすると、インポート時に作成されたファイルが現在の Tau プロジェクトに追加され、Tau にロードされます。

コマンドライン ユーザー インターフェイス

次の構文を使用して、Rose インポータをコマンド ラインから起動できます。

```
MDLImpBatch.exe [-p <vpm file>] { <file>.mdl | <file>.ptl }+ <file>.u2
```

`<vpm file>`
仮想パスの設定に記述されている仮想パス マッピング ファイル

`{ <file>.mdl | <file>.ptl }+`
モデル ファイルまたは petal ファイルのリスト

`<file>.u2`
結果として生成される最上位の .u2 ファイル

コマンド ライン インターフェイスを使用した場合は、生成されたファイルをプロジェクトに手動で追加する必要があります。これは、プロジェクトを開いて、[プロジェクト] -> [プロジェクトに追加] -> [ファイル] コマンドを使用して実行できます。プロジェクトが「発見ベース」である場合は、「手動による再検索」を行ってプロジェクトをリフレッシュします。

変換ルール

ほとんどの場合、Rose の要素は Tau に UML 要素と同じ種類としてインポートされます。たとえば、Rose のクラスは Tau のクラスとしてインポートされます。ただし、特定の構造要素については、できるだけ多くインポートできるように、インポート時に変換が行われます。

このセクションでは、インポート時に適用される変換ルールについて説明します。

クラス図

クラス図のアクター シンボルはクラス シンボルに変換されます。

クラス図のユースケース シンボルは操作シンボルに変換されます。

ユースケース図のクラス シンボルはアクター シンボルに変換されます。

コラボレーション図

コラボレーション図はシーケンス図に変換されます。コラボレーション図にあるすべてのオブジェクトはライフライン シンボルに変換されます。

ステート図

ネストされたステートを含むステート図は、次のアルゴリズムに従ってインポートされます。

ダイアグラムは複数の「レベル」に分割されます。各レベルには、最上位レベルのダイアグラムのクローンである独自のダイアグラムが含まれます。レベルとスコープが同じでネストされている各ステート（同じ合成状態のサブステート）は、同じダイアグラムに配置されます。ステート シンボルの位置とサイズおよび遷移ラインの座標は変更されません。レベルとスコープが同じステート間の遷移ラインは、変換されずにインポートされます。それ以外の場合は、インポート時に別の遷移ラインが作成されます。

アクティビティ図

ネストされているアクティビティを含むアクティビティ図は、ネストされているステートを持つステート図と同じ方法で処理されます。「ステート図」を参照してください。アクティビティ図にあるステート シンボルは ActionActivitySymbol に変換されます。

階層図

階層図はクラス図に変換されます。

共通ルール

テキスト ラベルはコメント シンボルに変換されます。

既知の制限事項

このセクションでは、既知の制限事項とサポートされない構成要素について説明します。

注記

制限事項の多くは、UML 1.x と UML 2.0 の仕様の差異に起因します。

モデル ファイルのフォーマット

古い Rose モデル（Petal バージョン 43 以前）をインポートすると、いくつかのデータが失われます。

非 ASCII フォーマットで保存されている Rose モデルは完全にはインポートできません。非 ASCII 文字を含む要素はインポートできません。

すべてのダイアグラム

コンパートメントはサポートされません。

フォント

クラスの Font 属性、script、bold、italics、underline、strike はサポートされません。

シンボル

インポート後のシンボルのサイズを含め、インポート後のダイアグラムのレイアウトが維持されます。これにより、シンボル内のテキストの一部が画面に表示できなくなります。この問題を解決するには、シンボルに自動サイズ変更を設定します。

例 **310:** シンボルテキストの省略

以下の図はシンボルテキストの省略と表示の例を示しています。左のダイアグラムの省略記号 ... は、テキストが表示されていないことを表します。右のダイアグラムではユースケースシンボルに自動サイズ変更が適用されているので、テキスト全体が表示されています。

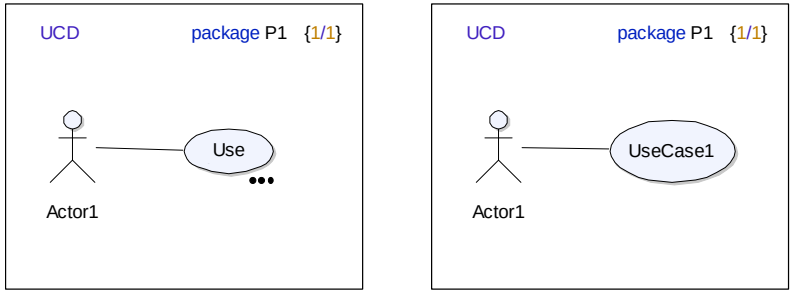


図 176: シンボルテキストの省略

アクティビティ図

History と HistoryAll はサポートされません。

クラス図

AssociationClass と Association 間のラインはサポートされません。

関連間の制約ラインはサポートされません。

「名前の方向」はサポートされません。

シーケンス図

メッセージの属性の「間隔」はサポートされません。

メッセージのドキュメント化はサポートされません。

ステート図

パーティションはサポートされません。

階層図

階層はサポートされません。

ユースケース図

ユースケースとアクター間の関連名はサポートされません。

ユースケースとアクター間の関連から「導出」される機能はサポートされません。

アクター間の汎化はユースケース図ではサポートされません。

13

Together インポート

この章では、Borland Together で作成したモデルを Tau にインポートする方法について説明します。この機能の目的は、Together モデルを Tau に移行することです。

注記

Together モデルをインポートするには、Together インポート ツールを使用します。Together モデルに XMI インポートを使用した場合、よい結果が得られないので推奨できません。

概要

Together インポータは、Borland Together モデルを Tau に移行するための機能です。

Together インポータの主な機能は以下のとおりです。

- Together モデル ファイルの全部または一部をインポートする。
  - レイアウト情報を含むモデルの要素とダイアグラム
- インポータが既存のモデル ファイルを直接操作するので、Together をインストールする必要がない。
- Together モデルの U2 ファイルへのマッピングを完全にカスタマイズできる。
- インポータは対話形式（623 ページの「インポートの開始」を参照）、またはコマンド ラインから（632 ページの「コマンドライン ユーザー インターフェイス」を参照）実行できる。

インポートの開始

Together モデルを Tau にインポートするには、以下の手順を行います。

- Tau を起動し、新しいプロジェクトを作成するか、または既存のプロジェクトを使用する。
- ［モデル ビュー］でモデル ノードを選択する。
- ［ファイル］メニューの［インポート］コマンドを選択して、［インポート ウィザード］を起動する。
- ［Borland Together Architect からのインポート］を選択し、［OK］をクリックする。
- Together インポート ウィザードの各ページをクリックして進む。

Tau を起動せずにモデルをコマンド ウィンドウからインポートするには、コマンドライン ユーザー インターフェイスを使用する。

Together インポート ウィザード

このセクションでは、Together インポート ウィザードのさまざまなステップを詳細に説明します。

- ステップ 1
- ステップ 2
- ステップ 3
- ステップ 4

ステップ 1

インポート プロセスのステップ 1 では、インポートするプロジェクトおよびその解釈方法を指定します。

- プロジェクトの指定
- リンクされているリソース
- 作成した U2 ファイルのデフォルトの格納場所

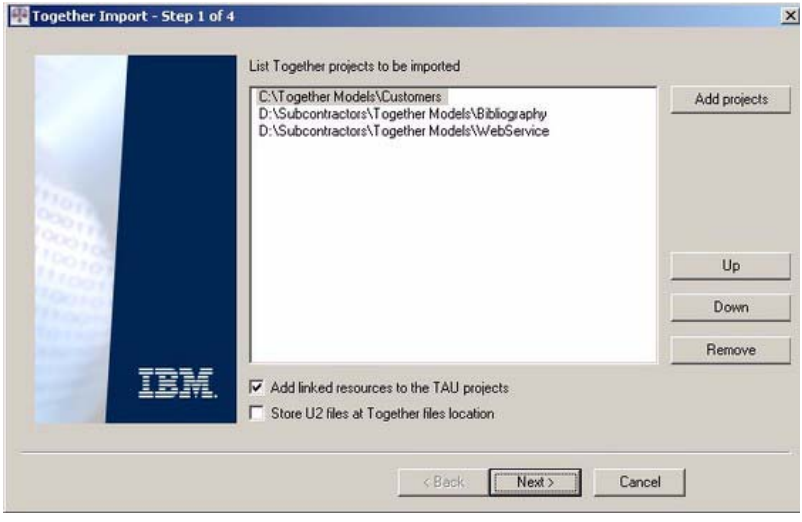


図 177: Together インポート ウィザードのステップ 1

すべてのモデル ファイル情報を指定して、[次へ] ボタンをクリックします。インポート ウィザードは指定したすべてのファイルを解析します。

注記

大きなプロジェクトの解析には時間がかかります。解析を途中で止める場合は、[キャンセル] ボタンを使用します。

プロジェクトの指定

最初のステップの主な目的は、インポート対象のプロジェクトを決めることです。選択した一連のプロジェクトがダイアログの中央に一覧表示されます。[プロジェクトの追加]、[上]、[下] および [削除] の各ボタンを使用して表示内容を変更できます。

- [プロジェクトの追加] ボタンをクリックすると、標準のファイルを開くダイアログが表示される。このダイアログで複数を選択できます。選択したフォルダに格納されているすべてのプロジェクトが、プロジェクトリストに追加されます。
- [上] ボタンまたは [下] ボタンをクリックすると、リスト内で選択しているプロジェクトが上または下に移動する。
- [削除] ボタンをクリックすると、選択しているプロジェクトがリストから削除される。

リンクされているリソース

リンクされているリソース（ファイルとフォルダ）は、デフォルトで Tau プロジェクトに挿入されます。この設定は、[リンクされたリソースを Tau プロジェクトに追加] オプションの選択を解除することで、変更が可能です。

作成した **U2** ファイルのデフォルトの格納場所

[U2 ファイルを Together ファイル位置に格納] オプションを使用して、インポートで作成される U2 ファイルのデフォルトの格納場所をカスタマイズできます。

このオプションを有効にすると、デフォルトで、作成されるすべての U2 ファイルは対応する Together プロジェクトのルート フォルダに格納されます。このオプションを無効にすると、すべての U2 ファイルは現在の Tau プロジェク ファイルのフォルダに作成されます。

例 **311** デフォルトのファイル格納場所

以下の場所に格納されている `ImportedProjects` というプロジェクトで作業しているとします。

```
C:¥Tau¥ImportedProjects¥ImportedProjects.ttp
```

3 つの Together プロジェクトをインポートします。

```
C:¥Project1
C:¥Project2
C:¥Project3
```

デフォルトのファイル格納場所を設定してこれらのファイルをインポートすると、U2 ファイルが作成され、プロジェクト フォルダに格納されます。

```
C:¥Together¥Project1¥Project1.u2
C:¥Together¥Project2¥Project2.u2
C:¥Together¥Project3¥Project3.u2
```

このオプションが無効の場合は、以下のようになります。

```
C:¥Tau¥ImportedProjects¥Project1.u2
```

```
C:¥Tau¥ImportedProjects¥Project2.u2
C:¥Tau¥ImportedProjects¥Project3.u2
```

注記

このオプションでは、デフォルトのファイル格納場所のみを指定します。ファイルのマッピングは次のインポートのステップで完全にカスタマイズできます（U2 ファイル マッピングの指定を参照）。

ステップ 2

ステップ 2 では、選択したプロジェクトが解析され、解析情報が表示されます。

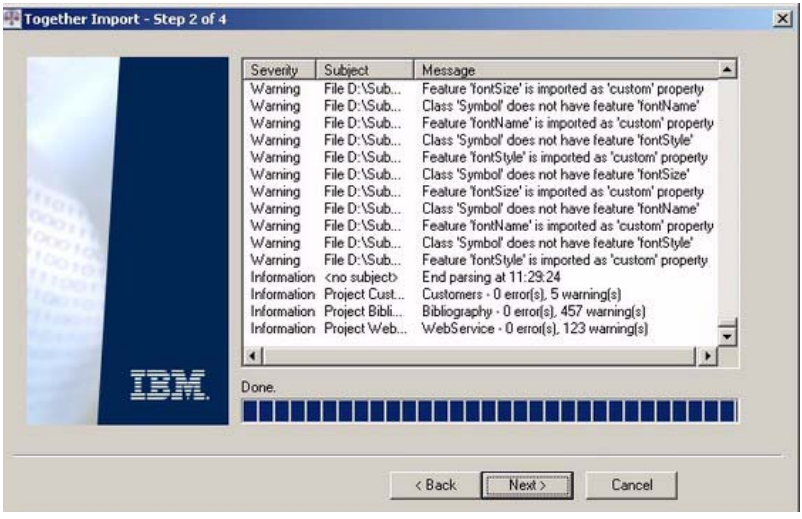


図 178: Together インポート ウィザードのステップ 2

ステップ 3

ステップ 3 では、実際にインポートする必要があるモデルのパートと、またその結果の格納方法を指定します。

- インポートするモデルのパートを指定
- U2 ファイル マッピングの指定
- 紛失ファイルの特定
- インポート オプションの設定

- ログ ファイル

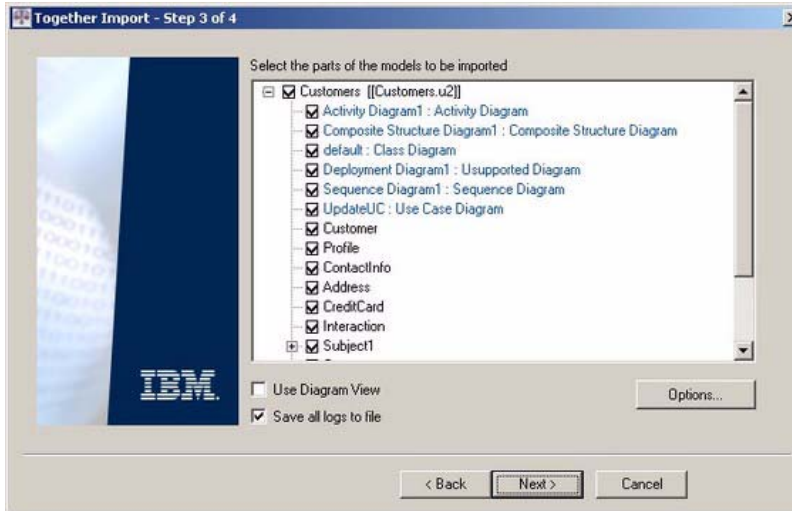


図 179: Together インポート ウィザードのステップ 3

インポートの対象とファイルのマッピング方法を指定し、［次へ］をクリックします。モデルがインポートされ、指定したとおりに、U2 ファイルに保存されます。

インポートするモデルのパートを指定

解析されたモデルはツリー形式で表示されます。ツリー表示では、モデルの全体構造ではなく、ファイル マッピングの指定に必要な詳細のみが表示されます。

表示には、標準ビュー（Standard View）とダイアグラム ビュー（Diagram View）の 2 つのツリー ビュー モードがあり、［ダイアグラム ビューを使用］オプションによって 2 つのモードを切り替えられます。

各ツリー項目の左には、インポート プロセスでの各ツリー項目の状態を示すチェック ボックスが表示されます。

- チェックボックスが空の場合、エンティティとその子のいずれもインポートされない。
- チェックボックスにグレーの印が付いている場合、エンティティがインポートされる。エンティティの子は一部（またはまったく）インポートされません。
- チェックボックスに黒い印が付いている場合、エンティティとその子のすべてがインポートされる。

赤い丸は、ファイルの参照が未解決であることを示しています。紛失したファイル参照を解決する方法については、紛失ファイルの特定を参照してください。

紛失ファイルの特定

インポートしたモデル ファイルに、自動検索できない外部ファイルへの参照が含まれている場合、ツリー表示のノードの前に赤い丸印が表示されます。紛失しているファイル参照を解決する方法には、以下の 2 通りがあります。

- ファイルを手動検索する。
  ツリー表示のノードをダブルクリックし、標準のファイルを開くダイアログを使用してファイルの場所を特定します。場所が特定されると、モデル ファイルが解析され、ツリー表示が更新され、新しく収集されたデータが表示されます。
- パス変数を設定する（インポート オプションの設定を参照）。

U2 ファイル マッピングの指定

ツリー表示を使用して、インポートしたモデルの U2 ファイル マッピングを指定することもできます。ツリー表示内の黒色の各ノードを U2 ファイルに関連付けることができます。

ノードを U2 ファイルに関連付けるには、以下の手順を行います。

- ツリー表示内のノードをダブルクリックし、ファイルの格納場所を参照する。
- ファイル名を入力し、［保存］をクリックする。

U2 ファイルにマッピングされているノードごとに、次のフォーマットに従ってノード名にファイル名が付加されます。

```
nodeName [[U2 file name]]
```

デフォルトのマッピングは、Together モデル ファイルごとに 1 つの U2 ファイルです。同じ名前が付けられ現在のプロジェクト ディレクトリに格納されます。

ダイアグラムは、個別の u2 ファイルに入れることはできません。ツリー ノードはツリー内で青色で強調表示されます。

インポート オプションの設定

Together モデルのインポートを行う場合、追加のインポート オプションとパス変数を指定できます。

629 ページの図 180 に［インポート オプション］ページを示します。

インポート オプションを変更するには、以下の手順を行います。

- ［オプション ...］ボタンをクリックする。Click the **Options...** button
- ［インポート オプション］ページをクリックする。Select **Import Options** Page
- 使用できるオプションを選択、または選択解除する。

- ［OK］をクリックする。

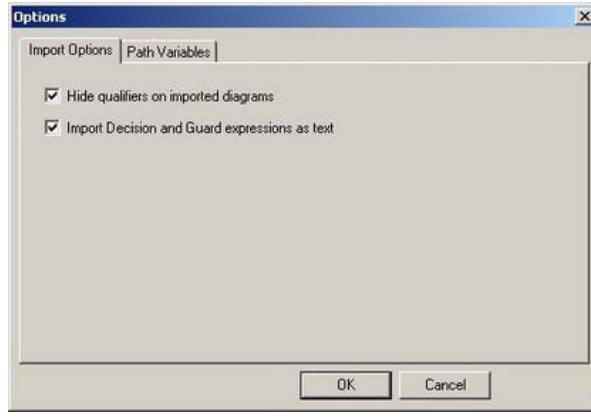


図 180: インポート オプション ページ

パス変数は Together でサポートされます。これは、新しい環境に対応してプロジェクトを簡単に再構成することで、コラボレーションに対応するためです。

パス変数を含むプロジェクトをインポートする場合、インポータは Together のレジストリ エントリを使用してパスを解決しようとします。その解決に失敗した場合、たとえば、インポートを実行するマシンに Together がインストールされていない場合など、未解決のパスを持つユニットは、モデルのツリー表示に未実装のユニットとして表示されます。例については、626 ページの図 178 を参照してください。未解決のユニットの位置を指定するために、2 つの方法が用意されています。最初の方法は、前述のように、ユニット ツリー項目をダブルクリックし、ダイアログ内のファイルを選択する方法です。もう 1 つの方法は、パス変数の値を定義する方法です。これは、複数の未実装のユニットが同じパスを共用する場合に有用です。

パス変数を変更するには、以下の手順を行います。

- ［オプション ...］ボタンをクリックする。
- ［パス変数］ページを選択する。
- 以下の説明に従って、パス変数を追加または編集する。
- ［OK］をクリックする。

630 ページの図 181 に［パス変数］ページを示します。このダイアログには、最初は Together のレジストリ データから抽出可能なすべての変数が含まれていますが、インポータはこれらを以下のファイルに保存します。

```
%APPLICATION DATA%¥IBM¥Rational¥Shared¥tau_virtpath.cfg
```

注記

Together がインストールされている場合でも、Together インポータでパス変数を変更しても、Together には反映されません。

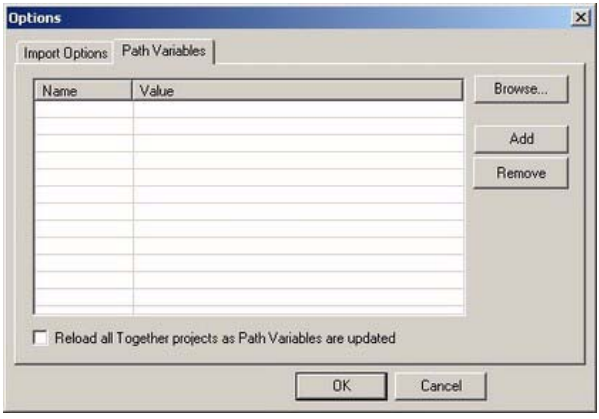


図 181: ［パス変数設定］ダイアログ

パス変数を追加するには、以下の手順を行います。

- ［追加］をクリックして、変数名を入力する。
- 新しく追加された変数を選択し、［参照］ボタンをクリックして変数フォルダの場所を特定する。

既存のパス変数の名前を変更するには、以下の手順を行います。

- 変数の名前カラムをダブルクリックし、新しい名前を入力する。

既存のパス変数のフォルダを変更するには、以下の手順を行います。

- 変数を選択し、［参照］ボタンをクリックして新しい変数フォルダの場所を特定する。

既存のパス変数を削除するには、以下の手順を行います。

- 変数を選択し、［削除］ボタンをクリックする。

［パス変数の変更に伴いすべての Together プロジェクトを再ロード］オプションにより、パス変数を変更したとき、解析したモデル ファイルを再び解析するかどうかを指定します。このオプションを選択すると、すべてのモデル ファイルは新しいパス変数を使用して再解析されます。このオプションを選択しないと、追加した変数に起因するもののみが解析されます。

注記

すべてのモデル ファイルを再解析すると、カスタマイズした U2 ファイル マッピングは失われます。

ログ ファイル

［すべてのログをファイルに保存］オプションを選択すると、エラーと警告のすべてのメッセージがログ ファイルに書き込まれます。このログ ファイルは最初のプロジェクトと同じ名前に .log 拡張子が付けられ、現在の Tau プロジェクト ディレクトリに保存されます。

ステップ 4

インポート ウィザードのステップ 3 を終了すると、インポート プロセスが開始されます。プログレス バーにインポートの進捗状況が表示され、警告メッセージとエラー メッセージが表示されます。

注記

警告とエラー メッセージは、ログ ファイルに書き込むようにも設定できます。

［キャンセル］ボタンをクリックするとインポートは中断します。

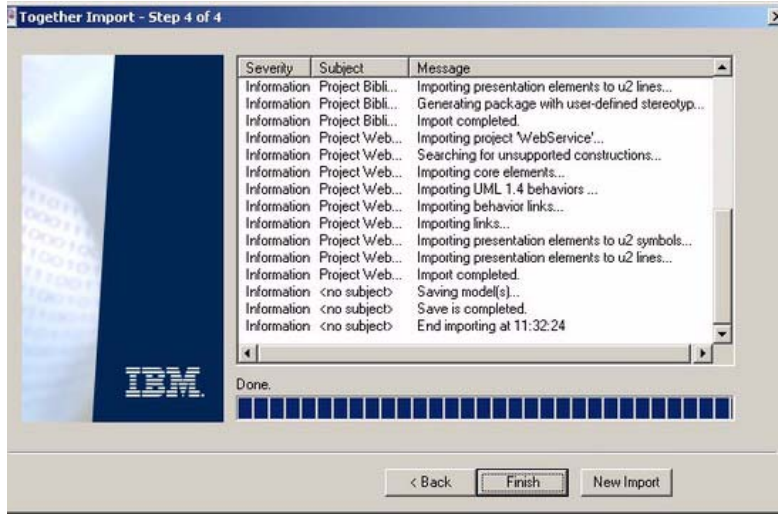


図 182: Together インポート ウィザードのステップ 4

インポートが完了すると［完了］ボタンと［新規インポート］ボタンが有効になり、ウィザードを終了するか、別のプロジェクトのインポートを実行できます。

［完了］ボタンをクリックすると、インポート時に作成されたファイルが現在の Tau プロジェクトに追加され、Tau にロードされます。

コマンドライン ユーザー インターフェイス

次の構文を使用して、Together インポータをコマンド ラインから起動できます。

```
BTXImpBatch.exe [-noGui <inputDir>+ <outputFile>.u2]
```

パラメータを指定せずにインポータを起動すると、GUI（グラフィカル ユーザー インターフェイス）が使用されます。

`-noGui`

GUI を使用せずにインポータを実行するよう指定するフラグ。

`<inputDir>+`

インポートする Together モデルが格納されているフォルダのリスト。

`<outputFile>.u2`

結果として生成される最上位の .u2 ファイル

コマンド ライン インターフェイスを使用した場合は、生成されたファイルをプロジェクトに手動で追加する必要があります。これは、プロジェクトを開いて、［Project］->［Add to project...］->［Files］コマンドを使用して実行できます。プロジェクトが「発見ベース」である場合は、手動による再検索を行ってプロジェクトをリフレッシュします。

変換ルール

ほとんどの場合、Together の要素は Tau に UML 要素と同じ種類としてインポートされます。たとえば、Together のクラスは Tau のクラスとしてインポートされます。ただし、特定の構造要素については、できるだけ多くインポートできるように、インポート時に変換が行われます。

このセクションでは、インポート時に適用される変換ルールについて説明します。

クラス図

アクター シンボル

クラス図のアクター シンボルはクラス シンボルに変換されます。

ユースケース シンボル

クラス図のユースケース シンボルは操作シンボルに変換されます。

ユースケース図

クラス シンボル

ユースケース図のクラス シンボルはアクター シンボルに変換されます。

コミュニケーション図

コミュニケーション図はシーケンス図に変換されます。コミュニケーション図にあるすべてのオブジェクトはライフライン シンボルに変換されます。

ステート図

ネストされたステートを含むステート図は、次のアルゴリズムに従ってインポートされます。

ダイアグラムは複数の「レベル」に分割されます。各レベルには、最上位レベルのダイアグラムのクローンである独自のダイアグラムが含まれます。レベルとスコープが同じでネストされている各ステート（同じ合成状態のサブステート）は、同じダイアグラムに配置されます。ステート シンボルの位置とサイズおよび遷移ラインの座標は変更されません。レベルとスコープが同じステート間の遷移ラインは、変換されずにインポートされます。それ以外の場合は、インポート時に別の遷移ラインが作成されます。

アクティビティ図

ネストされているアクティビティを含むアクティビティ図は、ネストされているステートを持つステート図と同じ方法で処理されます。

ステート シンボル

アクティビティ図にあるステート シンボルは ActionActivitySymbol に変換されます。

共通ルール

テキスト ラベルはコメント シンボルに変換されます。

14

UML インポート

この章では、IBM Rational Tau 以外の UML ツールで作成された UML モデルとダイアグラムのインポートについて説明します。

動作原理

XMI

XMI - XML メタデータ交換は、異なる（個別の）ツール間での UML モデルの交換を可能にする XML に基づいた UML メタデータ表現の標準です。XMI DTD（XML Document Type Definitions）は、XMI ドキュメントの構文仕様を提供します。この仕様に従って汎用 XML ツールで XMI ドキュメントのコンポーズおよびバリデートを行うことができます。

UML メタ モデル クラスは、XMI DTD 内でその名前がクラス名である XML 要素によって表されます。要素定義によってクラスの属性が記述されます。クラスに関連する関連付けの終端の参照、明示的またはコンポジションの関連付けによってネストされたクラスなどが対象になります。

メタモデル クラスの属性は、DTD 内でその名前が属性名である XML 要素によって表されます。

メタモデル クラス間の関連付け（包含ありと包含なしの両方）は、関連付けの終端の役割を表す 2 つの XML 要素によって表されます。

XMI インポート

UML インポート実行時に、XMI 標準に準拠したファイルを読み込み、XMI ファイルの内容を解釈してから UML モデルを作成します。インポートが完了すると、インポートされた内容を可視化するためにプレゼンテーション要素（ダイアグラムとシンボル）が作成されます。また、インポートされた XMI ファイルにダイアグラムとシンボルの情報が含まれている場合は、そのような情報を使用して、インポート後の UML モデルの表示が保持されます。

ダイアグラム情報のない XMI ファイルは、インポートされますが、UML モデル要素のみが作成されます。

XMI インポート アドイン

XMI インポート機能は、**XMIImport** という名前のアドインによって提供されます。

XMI インポート アーキテクチャ

この機能のアーキテクチャの概要を、637 ページの図 183 に示します。XMI Reader モジュールが XMI 仕様のファイルを読み込みます。このモジュールは、それぞれのタグから情報を変換してその情報を UML API に渡します。

UML モデルの要素は、すべて UML API で作成されます。UML API のコア部分は、UML メタモデルの場合と同じクラス階層を持つ C++ クラス セットになっています。UML API は、動的に（タグごとに）UML モデルのスケルトンを作成する（XMI Reader と連携して）モジュール ビルダです。

このフェーズでは、一部の情報を UML モデルに追加できない場合があります。そのような情報は収集されて UML Resolver モジュールに渡されます。

U2 Resolver は、UML モデルのスケルトンに対して一連の変換を実行します。

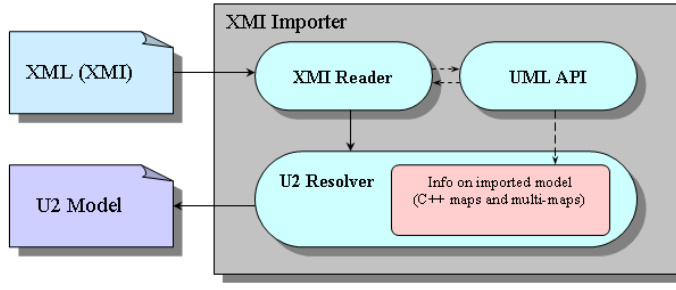


図 183: XMI インポート アーキテクチャ

例 **312: UML Resolver**

U2 Resolver に渡される情報が「列挙型」データ型のインポートの場合の例を示します。たとえば、Rational Rose は、「列挙」を `<<enumeration>>` によってステレオタイプ化されたクラスとしてエクスポートしますが、Tau では「列挙」は `DataType` となります。適用されたステレオタイプに関する情報は、クラス インポート時に提供されません。したがって、このクラスは後から変換する必要があります。必須変換に関する情報は、ステレオタイプ インポート時に U2 Resolver に渡されます。

XMI ファイルのインポート

XMI インポートは、Tau GUI から呼び出します。XMI インポートを開始するには、プロジェクトが含まれるワークスペースを開く必要があります。

- ［モデル ビュー］で**パッケージ**を選択します（詳細レイアウトを使用します。ワークスペース ウィンドウにビュー タブがあります）。
- インポート ウィザードを開きます（［ファイル］メニューの［インポート］コマンド）。
- ダイアログで［XMI のインポート］を選択して、［OK］をクリックします。
- 表示されるダイアログで、インポートする XMI ファイルを指定します。

2 番目のダイアログを閉じると、以下のような結果になります。

- パッケージ **ImportedXMIDefinitions** がモデル内に作成されます。
- ステレオタイプ **xmiImportSpecification** がパッケージに適用されます。

- インポートする XMI ファイルが、パッケージのステレオタイプ インスタンスに値として格納されます。
- インポート操作が実行されて、作成されたパッケージに結果が追加されます。

同じ設定を使用する **XMI** 仕様の再インポート

［モデル ビュー］で、**xmiImportSpecification** ステレオタイプが適用されているパッケージを選択します。

パッケージを右クリックして、ポップアップ メニューから［XMI のインポート］を選択します。

インポート操作が実行されて、結果がパッケージに追加されます。

設定を変更するには（インポートするファイルを選択）、［XMI のインポート］コマンドを実行する前に、パッケージのステレオタイプ インスタンスでプロパティを編集できます。

注記

［インポート ウィザード］ダイアログを使用すると、新しいパッケージが作成されます。ポップアップ メニューから［XMI のインポート］を再度使用すると、既存のパッケージが再利用されます。

サポートされる **XMI** と **UML**

言語とバージョンのサポート

以下の言語とバージョンが XMI インポートでサポートされています。

- XMI 1.0/1.1
- UML 1.4

XMI インポートでサポートされている UML 1.4 エンティティを、以下に示します。特に指定がない限り、エンティティの関係と属性もサポートされます。

基底／コア

- 関連（Association）
- 関連の終端（AssociationEnd）
- 属性（Attribute）
- クラス（Class）
- コメント（Comment）
- コンポーネント（Component）
- 制約（Constraint）
- データ型（DataType）
- 依存（Dependency）
- 要素常駐場所（ElementResidence）

- 列挙（Enumeration）
- 列挙リテラル（EnumerationLiteral）
- 汎化（Generalization）
- インターフェイス（Interface）
- メソッド（Method）
- 操作（Operation）
- パラメータ（Parameter）
- 許可（Permission）
- 構造特性（StructuralFeature）

基底／拡張

- ステレオタイプ（Stereotype）
- タグ付き値（TaggedValue）
- タグ定義（TagDefinition）

基底 / データ型

- ブール値（Boolean）
- 論理式（BooleanExpression）
- 式（Expression）
- 整数（Integer）
- 多重度（Multiplicity）
- 多重度範囲（MultiplicityRange）
- 名前（Name）
- プロシージャ式（ProcedureExpression）
- 文字列（String）
- 未解釈（Uninterpreted）

モデル管理

- モデル（Model）
- パッケージ（Package）
- サブシステム（Subsystem）

振る舞い要素／共通振る舞い

- アクション シーケンス（ActionSequence）
- 引数（Argument）
- 呼び出しアクション（CallAction）
- 生成アクション（CreateAction）
- 消滅アクション（DestroyAction）
- 例外（Exception）

- 戻りアクション（ReturnAction）
- 送信アクション（SendAction）
- シグナル（Signal）
- 終了アクション（TerminateAction）
- 未解釈アクション（UninterpretedAction）

振る舞い要素／コラボレーション

- 分類子ロール（ClassifierRole）
- コラボレーション（Cllaboration）
- 相互作用（Interaction）
- メッセージ（Message）

振る舞い要素／ユース ケース

- アクター（Actor）
- 拡張（Extend）
- 包含（Include）
- ユース ケース（UseCase）

振る舞い要素 / 状態機械

- 合成ステート（CompositeState）
- 呼び出しイベント（CallEvent）
- 終了状態（FinalState）
- ガード（Guard）
- 擬似ステート（Pseudostate）

```
Initial
Choice
Junction
DeepHistory
ShallowHistory
```

- シグナル イベント（SignalEvent）
- ステート（State）
- 単純ステート（SimpleState）
- 状態機械（StateMachine）

サポートされるダイアグラム タイプ

XMI ファイルに必須ダイアグラム情報が含まれている場合に、XMI インポートでは以下の UML ダイアグラム タイプをサポートします。

- クラス図
- コンポーネント図
- 配置図

- パッケージ図
- アクティビティ図
- シーケンス図
- ユース ケース図
- 状態機械図

保存されたレイアウトを使用するインポート

このカテゴリのダイアグラムは、XMI ファイルに図形的なレイアウトが存在するダイアグラムです。

- クラス図
- コンポーネント図
- 配置図
- パッケージ図
- アクティビティ図
- ユース ケース図
- シーケンス図
- 状態機械図

ネストされたステートのインポート

レイアウトは保持されますが、ネストされたステートには特殊な配慮を適用します。

- ステートがネストされているステートごとに、一連のダイアグラムが作成されます（ネスト レベルごとに 1 つ）。
- これらのダイアグラムのステートの位置は、可能な限り元と同じように保持されます。
- 開始シンボルと戻りシンボルは、必要に応じて新しい各ダイアグラム上に作成されます。これらのシンボルの位置は、可能な限り高位ネスト レベル上の対応するシンボルの位置と同じに保持されます。
- 新しいエントリと終了の接続ポイントは、必要に応じて作成されます。
- 大量のテキストが含まれている遷移イベントとアクションは重複する場合があります。

UML 1.x ツールからのインポート

通常の場合、XMI インポート ツールは、サポートされる XMI バージョンに準拠している以下の UML 1.x ツールからの XMI ファイルをサポートします。

- Rational Rose/Unisys (JCR.2 v.1.3.x)
- Tau UML スイート
- Borland Together
- IBM XMI ツールキット

Rhapsody

Rhapsody は XMI をエクスポートしますが、ダイアグラム情報は含まれません。Rhapsody からエクスポートされた XMI ファイル内の情報を使用して、モデル要素が作成されます。その結果、UML 構造がワークスペース ウィンドウに表示されます（ダイアグラムは含まれません）。

Rational Rose

- Unisys の拡張機能を使用する Rational Rose は、XMI をダイアグラム情報と一緒にエクスポートします。この情報は、ダイアグラム作成時の XMI インポート時に使用されます（ダイアグラムがサポートされるダイアグラム タイプのいずれかであることが前提になります）。
- ダイアグラム レイアウトは、クラス図、ユース ケース図、シーケンス図用に保持されます。
- Rational Rose 名はサポートされます。

注記

Rational Rose からのモデルのインポートには、XMI によるインポートではなく Rose のインポート機能を使用することを強く推奨します。Rose のインポート機能を使用すると、モデル ファイルを直接インポートできるので、より多くの情報を保持できます。

DOORS リンクの保持

Rational Rose からの XMI のインポートの間に DOORS のリンクを保持できます。

- UML モデルをエクスポートします。［Generated UUIDs］チェックボタンが選択されていることを確認します。
- 生成された XMI を Tau にインポートします。
- 既存の DOORS インテグレーション コマンドを使用して、Tau から DOORS へ新しい UML をエクスポートします。
- DOORS 内で Tau 代理モジュールを開き、［Import Links from Rational Rose］メニューを選択して指示に従います。

この操作が完了すると、DOORS 内の代理モジュールとの間のすべてのリンク（DOORS Rose Link インテグレーションにて作成された）は、Tau 代理モジュールのためにコピーされます。

Tau UML Suite

- Tau UML Suite が Unisys の拡張機能を使用している場合、XMI をダイアグラム情報と一緒にエクスポートします。この情報は、ダイアグラム作成時の XMI インポートに使用されます（ダイアグラムがサポートされるダイアグラム タイプのいずれかであることが前提になります）。
- ダイアグラム レイアウトは、クラス図、ユース ケース図、シーケンス図用に保持されます。

参照

言語とバージョンのサポート

制限事項

この章では XMI ／ UML サポートのレベルについて説明していますが、以下のセクションではさらにその他の既知の制限について説明します。

タイプと変数の定義

- ローカル データ型定義は状態機械図に表示されません。
- ローカル変数定義は状態機械図に表示されません。

不完全なモデル

XMI 仕様をインポートするには、完全で、セマンティックの正しい UML モデルになっている必要があります。通常の場合、不完全または不正な仕様は Tau にインポートできませんが、そのような仕様を完全な仕様としてインポートするか、インポート時に一部の情報が失われる場合があります。

例 **313:** 不完全なモデルのインポート

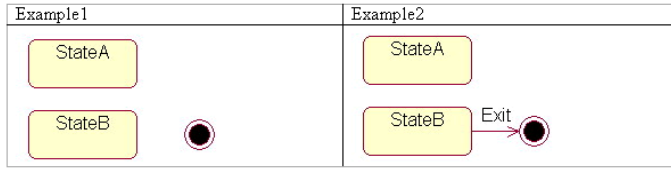


図 184: 不完全なモデル

例 1 では、FinalState がインポートされません。このステートが ReturnAction に変換されるためです。このアクションは FinalState に入る遷移によって所有される必要があります。そのような遷移はこの例では存在しないため、FinalState はインポートされません。

例 2 では、すべてのダイアグラム要素がインポートされますが、このダイアグラムも不完全です（このダイアグラムには InitialState がありません)。

サポートされないクラス

UML 構成要素には 、XMI インポート実行中に処理されないものがあります。

以下の構成要素の場合は、エラー メッセージ TUI0004（サポートされていないクラス）が出力されます。

基底：コア
- アーティファクト（Artifact）
- 関連（Association）（ユース ケース間）
- バインディング（Binding）
- フロー（Flow）
- 汎化（Generalization）（アクター間）

振る舞い要素：共通振る舞い
- 属性リンク（AttributeLink）
- コンポーネント インスタンス（ComponentInstance）
- データ値（DataValue）
- インスタンス（Instance）
- リンク（Link）
- リンク終端（LinkEnd）
- ノード インスタンス（NodeInstance）
- オブジェクト（Object）
- 受信（Reception）
- 刺激（Stimulous）
- サブシステム インスタンス（SubsystemInstance）

振る舞い要素：アクティビティ グラフ（**ActivityGraphs**）
- ステート内の分類子（ClassifierInState）
- オブジェクト フロー状態（ObjectFlowState）
- 擬似ステート（Pseudostate）（浅い履歴と詳細な履歴）（Shallow history と Deep history）

振る舞い要素：コラボレーション
- 関連終端ロール（AssociationEndRole）
- 関連ロール（AssociationRole）
- コラボレーション インスタンス セット（CollaborationInstanceSet）
- 相互作用インスタンス セット（InteractionInstanceSet）

振る舞い要素：状態機械
- 変更イベント（ChangeEvent）
- スタブ ステート（StubState）
- タイム イベント（TimeEvent）

振る舞い要素：ユース ケース
- ユース ケース インスタンス（UseCaseInstance）

サポートされない属性

以下の属性の場合は、サポートされていない属性のエラー メッセージ（TUI0006）が出力されます。

基底：コア

- 関連終端（AssociationEnd）
  - 仕様（Specification）
- 属性
  - 関連終端（AssociationEnd）
- 振る舞い特性（BehavioralFeature）
  - 発生シグナル（RaisedSignal）
- コンポーネント（Component）
  - デプロイメント（Deployment）
- 制約（Constraint）
  - 制約付きステレオタイプ（ConstrainedStereotype）
- 特性（Feature）
  - 所有者（Owner）
- メソッド（Method）
  - 本体（Body）
  - 所有者スコープ（OwnerScope）
- モデル要素（ModelElement）
  - プレゼンテーション（Presentation）
  - テンプレート（Template）
- 操作（Operation）
  - 並列性（Concurrency）
  - オカレンス（Occurence）
  - 仕様（Specification）

基底：データ型

- 式（Expression）
  - 言語（Language）

基底：拡張

- ステレオタイプ（Stereotype）
  - アイコン（Icon）
  - ステレオタイプ制約（StereotypeConstraint）

振る舞い要素：コラボレーション

- コラボレーション（Collaboration）
  - 記述分類子（RepresentedClassifier
  - 記述操作（RepresentedOperation）
- 相互作用（Interaction）
  - コンテキスト（Context）
- メッセージ（Message）
  - アクティベータ（Activator）

振る舞い要素：状態機械

- 合成ステート（CompositeState）
  - 並列プロパティ（Concurrent property）

振る舞い要素：ユース ケース

- アクター（Actor）
  - 抽象プロパティ（Abstract property）
- ユース ケース（Use Case）
  - 拡張ポイント（ExtensionPoint）

モデル管理

- サブシステム（Subsystem）
  - インスタンス化可能プロパティ（Instantiable porperty）

サポートされないコンポジション

以下の構造の場合は、サポートされていないコンポジションのエラー メッセージ（TUI0008）が出力されます。

基底：コア

- 関連終端（AssociationEnd）
  - 修飾子（Qualifier）
- コンポーネント（Component）
  - 実装（Implementation）

振る舞い要素：アクティビティ グラフ（**ActivityGraphs**）

- ステート
  - 内部遷移（InternalTransitions）（アクティビティのアクション）
  - ステート（State）（状態機械に基づく）
  - 擬似ステート：履歴（Pseudostate: History）（浅い履歴と詳細な履歴）（Shallow history と Deep History）

振る舞い要素：コラボレーション

- コラボレーション
  - 制約要素（ConstrainingElement）

振る舞い要素：状態機械

- 状態機械（StateMachine）
  - 同期ステート（SynchState）（同期バー）
- ステート
  - 内部遷移（InternalTransitions）（ステートのアクション）
  - 擬似ステート：ジャンクション（Psuedostate:Junction）

コラボレーション図はサポートされていません。

エクスポートの制限事項

Rational Rose は、不完全なエクスポートを実行する場合があります。その結果、インポート後に一部の情報が失われることがあります。Rational Rose エクスポータ（Unisys 1.3.6）の既知の問題（エクスポートされない機能）を以下に示します。

クラス図

- クラス
  - タイプ（ParameterizedClass、ClassUtility、InstantiatedClass など）
  - 多重度（Multiplicity）
  - スペース（Space）
  - 並列性（Concurrency）
  - 形式（Format）（表示可能性）
- 属性（Attribute）
  - 包含（Containment）

- 操作（Operation）
  - プロトコル（Protocol）
  - 修飾（Qualification）
  - サイズ（Size）
  - 時間（Time）
- バイナリ関連（Binary Association）
  - 制約（Constraints）
  - 包含（Containment）
  - 派生（Derived）
  - フレンド（Friend）
  - リンク要素（LinkElement）
  - 名前ディレクション（Name Direction）
- 継承（Inheritance）
  - 文書化（Documentation）
  - 仮想継承（Virtual inheritance）
  - フレンドシップ必須（Friendship Required）
- 実現化（Realization）
  - 文書化（Documentation）
- 依存 / インスタンス化（Dependency/Instantiates）
  - 多重度（こちらから）（Multiplicity from）
  - 多重度（こちらへ）（Multiplicity to）
  - フレンドシップ必須（Friendship Required）

ステート図

- 遷移（Transition）
  - ステレオタイプ（Stereotype）
  - 文書化（Documentation）

シーケンス図

- メッセージ（Message）
  - 頻度（Frequency）（周期的、非周期的）
- 破棄マーカ（Destruction marker）

ユース ケース図

- アクター（Actor）
  - タイプ（Type）
  - 多重度（Multiplicity）
- ユース ケース（Use Case）
  - ステレオタイプ（Stereotype）
  - ランク（Rank）
- バイナリ関連（Binary Association）
  - 派生（Derived）
  - リンク要素（Link Element）
  - 名前ディレクション（Name Direction）
  - 制約（Constraints）
  - フレンド（Friend）
  - 包含（Containment）
- 依存（Dependency）

パッケージ図

- 依存（Dependency）
  - 文書化（Documentation）

コンポーネント図

- パッケージ（Package）
  - グローバル（Global）
- コンポーネント（Component）
  - 宣言（Declarations）

配置図

- プロセッサ（Processor）
  - スケジューリング（Scheduling）
- プロセス（Process）
  - 優先順位（Priority）
- デバイス（Device）
  - ステレオタイプ（Stereotype）
- 接続（Connection）

アクティビティ図

- スイムレーン（Swimlane）
  - 文書化（Documentation）
- オブジェクト（Object）
- オブジェクト フロー（Object Flow）

エラー メッセージ

概要

XMI インポート中のエラーメッセージは 出力ウィンドウに表示されます。

XMI インポートのエラーメッセージ

コード	テキスト	コメント
TUI0004	属性 '<name>'(<name>)(クラス '<name>') はサポートされていません	XMI 仕様に XMI 標準で指定されていない属性が含まれているか、属性を Tau 内の現在のクラスに適用できないときにエラーが発生します。たとえば、クラス「アクター」の属性「isAbstract」は、Tau に適用できません。
TUI0006	合成 '<name>'(クラス '<name>' からクラス '<name>') はサポートされていません	このエラー メッセージは、クラス間の合成がサポートされていないときに出力されます。たとえば、「修飾子」合成は Tau でサポートされていません。
TUI0008	クラス <name> はサポートされていません	インポートされた XMI 仕様に、たとえば、サポートされていないクラス「インスタンス」が含まれているとエラーが発生します。
TUI0009	グラフィック要素 '<name>'(クラス '<name>') は描かれていません	このエラー メッセージは、PresentationElement の対応する ModelElement が見つからないときに出力されます。たとえば、対応する ModelElement はサポートされていません。
TUI0010	ダイアグラム形式 '<name>'(値 '<name>') はサポートされていません	プレゼンテーション要素が Tau によってサポートされていないときにエラーが発生します。たとえば、ステレオタイプの PresentationElement はサポートされていません。
TUI0016	ファイル <name> を開けませんでした	XMI Importer に渡されたファイルは開けません。
TUI0017	XMI ファイルの分析中に分析エラーが検出されました	このエラー メッセージは、XML パーサで情報を XMI 仕様から読み込めないときに出力されます。たとえば、XML パーサは終了タグを発見できません。
TUI0022	内部エラーです	内部エラーはインポート時に発生します。

15

UML1.x XMI エクスポート

本章では、UML1.x を使用するツールへの XMI 形式のモデル データのエクスポートを Tau がどのようにサポートするかについて説明します。

XMI のエクスポート

操作の原理

UML エクスポータは、XMI 標準に準拠するファイル形式を生成します。エクスポート時、モデル要素とプレゼンテーション要素の両方がファイルに書き出されます。Unisys XML プラグインに基づいて UML モデルの外観を保持するため、ダイアグラムとシンボルのレイアウト情報が含まれます。

XMI エクスポート アドイン

XMI エクスポート機能は、**XMIExport** という名前のアドインで提供されます。

XMI ファイルへのエクスポート

XMI エクスポータは、Tau GUI から呼び出します。

- ［XMI エクスポート］ウィンドウを開きます（［ツール］メニューから［モデルを XMI にエクスポート］コマンドを選択します）。
- 表示されるダイアログで、エクスポート先の XMI ファイルを指定します。

ワークスペースに複数のプロジェクトが存在する場合、XMI エクスポートは選択したプロジェクトに対して行われます。プロジェクトを選択しなかった場合、また複数のプロジェクトを選択した場合、メニュー項目がグレー表示となります。

サポートされる XMI とツールのバージョン

XMI エクスポータは以下のバージョンをサポートします。

- XMI 1.1

XMI エクスポータは、以下のターゲット ツール環境でテストされています。

- Rational Rose Enterprise Edition 2003
- Rose XML Tools (UniSys XML plug-in) 1.3.6 for Rational Rose

サポートされる UML エンティティ

以下の表に、XMI エクスポータがサポートする Tau UML エンティティの一覧を示します。

- **UML エンティティ**
  Tau 内の UML エンティティ。
- **エクスポート**
  Tau からエクスポートして Rose にインポートした場合、Rose 内でのインポート結果のエンティティ。

- **ラウンドトリップ**
  XMI ラウンドトリップを実行した場合、Tau 内でのインポート結果のエンティティ

このリストにない他のエンティティはエクスポートされません。

UML ダイアグラム	エクスポート	ラウンドトリップ
アクティビティ図	同じ	同じ
クラス図	同じ	同じ
コンポーネント図	クラス図	クラス図
配置図	クラス図	クラス図
パッケージ図	クラス図	クラス図
シーケンス図	同じ	同じ
状態機械図	同じ	同じ
テキスト図	ノート付きクラス図	ノート付きクラス図
ユース ケース図	クラス図	クラス図

全般	エクスポート	ラウンドトリップ
コメント シンボル	ノート	同じ
注釈ライン	アンカー	同じ
テキスト シンボル	ノート	コメント シンボル
<Any>		
コメント	ドキュメント	なし
ステレオタイプ	同じ	同じ
リンク	ファイル	なし
色	同じ	同じ
フォント	同じ	同じ

アクティビティ図	エクスポート	ラウンドトリップ
アクティビティ シンボル	同じ	同じ
• 名前	同じ	同じ
アクション	「Entry」アクション	なし

アクティビティ図	エクスポート	ラウンドトリップ
アクティビティ	<<activity>> でステレオタイプ化されたクラス	<<activity>> でステレオタイプ化されたクラス
開始ノード	同じ	同じ
アクティビティ終了	終了ステート	アクティビティ終了
フロー終了	終了ステート	アクティビティ終了
アクティビティ ライン	遷移	同じ
テキスト	遷移ラベル	同じ
フォーク／ジョイン	同期	同じ
分岐	同じ	同じ
• 名前	同じ	同じ
SendSignalSymbol	「Do/send」アクションを持つ名称未設定アクティビティ	アクションのない自動名前変更済みアクティビティ
AcceptEventSymbol	「Do/receive」アクションを持つ名称未設定アクティビティ	アクションのない自動名前変更済みアクティビティ
AcceptTimeEventSymbol	「Do/receive」アクションを持つ名称未設定アクティビティ	アクションのない自動名前変更済みアクティビティ

クラス図	エクスポート	ラウンドトリップ
クラス	同じ	同じ
• 名前	同じ	同じ
• 抽象	同じ	同じ
• テンプレート パラメータ	仮引数	同じ
• 可視性	エクスポート コントロール	同じ
クラス属性	同じ	同じ
• 名前	同じ	同じ
• タイプ	同じ	同じ

クラス図	エクスポート	ラウンドトリップ
• 可視性	エクスポート コントロール	同じ
• デフォルト値	初期値	同じ
• 派生	同じ	同じ
クラス操作	同じ	同じ
• 名前	同じ	同じ
• 戻り型	同じ	同じ
• 可視性	エクスポート コントロール	同じ
• 指定された例外	例外	同じ
クラス操作パラメータ	同じ	同じ
• 名前	同じ	同じ
• タイプ	同じ	同じ
• デフォルト値	同じ	同じ
要求インターフェイス	インターフェイス	<<interface>> でステレオタイプ化されたクラス
実現化インターフェイス	インターフェイス	<<interface>> でステレオタイプ化されたクラス
インターフェイス	同じ	<<interface>> でステレオタイプ化されたクラス
タイマー	<<timer>> でステレオタイプ化されたクラス	<<timer>> でステレオタイプ化されたクラス
シグナル	<<signal>> でステレオタイプ化されたクラス	同じ
ステレオタイプ	<<stereotype>> でステレオタイプ化されたクラス	<<stereotype>> でステレオタイプ化されたクラス
操作	<<operation>> でステレオタイプ化されたクラス	<<operation>> でステレオタイプ化されたクラス
状態機械	<<statemachine>> でステレオタイプ化されたクラス	<<statemachine>> でステレオタイプ化されたクラス
基本／列挙	<<primitive>>/<<enumeration>> でステレオタイプ化されたクラス	<<primitive>>/DataType でステレオタイプ化されたクラス

クラス図	エクスポート	ラウンドトリップ
アーティファクト	<<artifact>> でステレオタイプ化されたクラス	<<artifact>> でステレオタイプ化されたクラス
コラボレーション	<<collaboration>> でステレオタイプ化されたクラス	<<collaboration>> でステレオタイプ化されたクラス
選択	<<choice>> でステレオタイプ化されたクラス	<<choice>> でステレオタイプ化されたクラス
関連ライン	同じ	同じ
• 名前	同じ	同じ
関連ロール	同じ	同じ
• 名前	同じ	同じ
• 可視性	エクスポート コントロール	同じ
制約	同じ	同じ
多重度	同じ	同じ
集約	集約、包含	同じ
所有者スコープ	静的	なし
汎化／実現化ライン	同じ	同じ
依存ライン	同じ	同じ
拡張ライン	<<extend>> でステレオタイプ化された依存	<<extend>> でステレオタイプ化された依存

コンポーネント図	エクスポート	ラウンドトリップ
コンポーネント シンボル	<<component>> でステレオタイプ化されたクラス	同じ

配置図	エクスポート	ラウンドトリップ
DeploymentSpecificationSymbol	<<deploymentSpecification>> でステレオタイプ化されたクラス	<<deploymentSpecification>> でステレオタイプ化されたクラス
ExecutionEnvironmentSymbol	<<executionEnvironment>> でステレオタイプ化されたクラス	<<executionEnvironment>> でステレオタイプ化されたクラス
NodeSymbol	<<node>> でステレオタイプ化されたクラス	<<node>> でステレオタイプ化されたクラス

パッケージ図	エクスポート	ラウンドトリップ
パッケージ	同じ	同じ
• 名前	同じ	同じ
依存ライン	同じ	同じ

シーケンス図	エクスポート	ラウンドトリップ
ライフライン	同じ	同じ
• 名前	同じ	同じ
• タイプ	クラス	同じ
メッセージ	単純なメッセージ	同じ
• 名前	同じ	同じ
メソッド呼び出し	プロシージャ呼び出しメッセージ	メッセージ
• 名前	同じ	同じ
メソッド応答	リターン メッセージ	同じ
• 名前	同じ	同じ
タイムアウト	タイムアウト メッセージ	メッセージ
• 名前	同じ	同じ
生成ライン	「:{Create}」という接尾辞の付いた名前のメッセージ	「:{Create}」という接尾辞の付いた名前のメッセージ
相互作用	<<interaction>> でステレオタイプ化されたクラス	<<interaction>> でステレオタイプ化されたクラス

状態機械図	エクスポート	ラウンドトリップ
ステート	同じ	同じ
• 名前	同じ	同じ
マルチ ステート（ステート リストまたはアスタリスク ステートを持つステート）	元のステート テキストに設定されたステート名を持つステート	元のステート テキストに設定されたステート名を持つステート
遷移ライン	同じ	同じ
ラベル	同じ	同じ
分岐	同じ	同じ
分岐質問	名前	同じ
分岐回答シンボル	遷移ガード条件	遷移ガード条件
開始	同じ	同じ
停止	終了ステート	リターン
リターン	終了ステート	同じ
フロー ライン	遷移	遷移
シグナル受信	遷移イベント	遷移イベント
ガード シンボル	遷移ガード条件	遷移ガード条件
アクション シンボル	遷移アクション	なし
シグナル送信	遷移送信イベント	なし

ユース ケース図	エクスポート	ラウンドトリップ
アクター	同じ	同じ
• 名前	同じ	同じ
• 可視性	エクスポート コントロール	同じ
ユース ケース	同じ	同じ
• 名前	同じ	同じ
パフォーマンス ライン	<<performance>> でステレオタイプ化された関連	同じ

ユース ケース図	エクスポート	ラウンドトリップ
依存ライン	同じ	なし
• 名前	同じ	なし
汎化ライン	同じ	同じ

モデルの階層

Rational Rose の包含階層は、662 ページの図 185 に示すような構造になります。

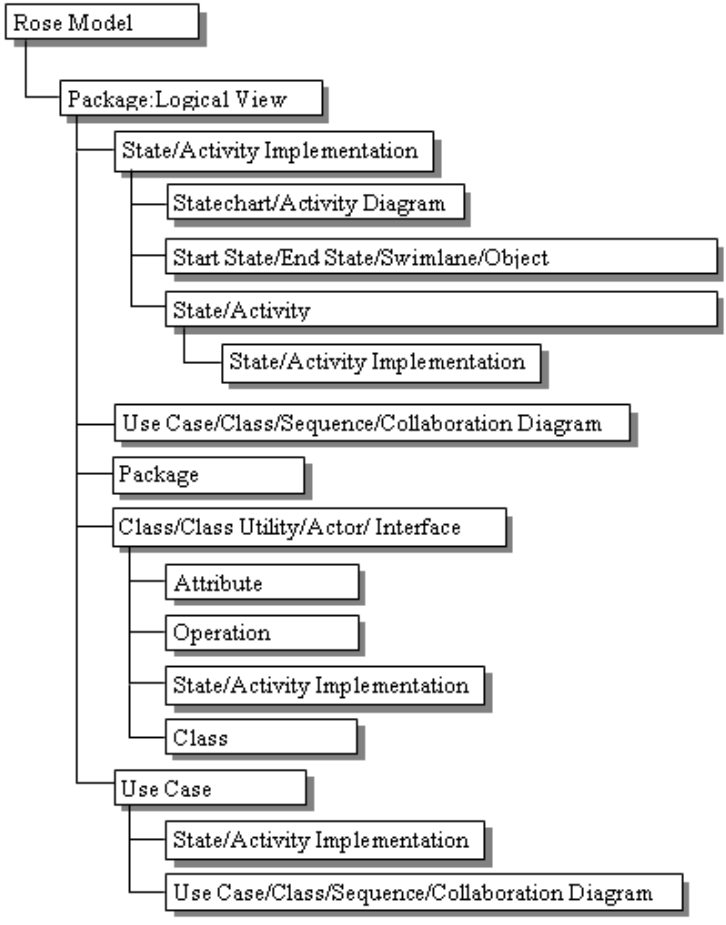


図 185: 包含階層

Rational Rose のビューはパッケージとして定義されます。Logical View は、ハードコードされた定義済みパッケージです。Rational Rose XMI モジュールがモデル要素とダイアグラムをインポートする際、デフォルトのインポート先となります。

State/Activity Implementations は状態機械仕様を表し、適用する要素の直下に置かれます。Packages とクラス Classes を（Classes、Class Utilities、Actors、Interfaces を介して）ネストできるので、階層は限りなく深くできます。どの要素の下にも File と URL を置くことができます。

一般的なルールとして、サポートされない XMI ファイルに包含すると、XMI インポート時に消失します。

モデル変換

モデル情報を可能な限り保持するため、モデルの変換が行われることがあります。

以下の表には次の内容を示します。

- Tau エンティティ
- エクスポート後の XMI 内においてエンティティを移動する理由の説明
- Tau エンティティに包含されていて上位階層に移動されるエンティティ

Tau	説明	移動されるエンティティ
インターナル	同等な機能はありません。	`Class diagram, Package diagram, Text diagram, UseCase diagram, Activity, Actor, Artifact, Association, Attribute, Choice, Class, Collaboration, DataType, Interaction, Interface, Operation, Signal, StateMachine, Stereotype, Timer, UseCase`
状態機械実装	可能なエンティティのタイプに関して、このレベルは大きく制限されます。	`Activity, Actor, Artifact, Association, Attribute, Choice, Class, Collaboration, DataType, Interface, Operation, Signal, Stereotype, Timer, UseCase, Class diagram, Package diagram, Text diagram, Use Case diagram`
アクティビティ実装	可能なエンティティのタイプに関して、このレベルは大きく制限されます。	`Actor, Artifact, Choice, Class, Collaboration, DataType, Interface, Signal, Stereotype, Timer, Use Case diagram`
相互作用実装	同等な機能はありません。	`Activity, Actor, Artifact, Attribute, Choice, Class, Collaboration, DataType, Interface, Operation, Signal, StateMachine, Stereotype, Timer, UseCase, Sequence diagram, UseCase diagram`
ネストされたクラス	ネストされたクラスの下の状態機械図はインポートされません。	`State machine diagram`

Tau	説明	移動されるエンティティ
クラス	クラスの下のインターフェイスはインポートされません。	`Interface`
属性	何かを包含することはできません。	`Artifact, Choice, Class, Collaboration, DataType, Interface, Stereotype`
選択	クラスに変換されます。	`UseCase`

Rational Rose への **XMI** エクスポートの制限事項

Tau がエクスポートする XMI データについて、Rational Rose XMI Import では多くの制限事項があります。既知の問題について以下の表に示します。

一般的な機能	説明
可視性オプション	この設定は XMI を通しては伝達できません。Tau で設定される可視性オプションと同じものがない場合は、ダイアグラム要素が重なることがあります。例として、クラス属性、操作、および操作シグニチャなどがあります。Tau で無効にしていた可視性オプションを XMI データのインポート時に有効にした場合、インポート後のクラスシンボルのサイズが異なるため、シンボルの重なりが生じます。
ダイアグラム タイプ	ユース ケース図は、クラス図として Logical View にインポートされます。
ライン	インポートによってラインの頂点は消失します。
	ラインの色はインポートされません。
要素	同じダイアグラム内の 2 つ以上のシンボルのインスタンス（クラスなど）はインポートできません。
ノート	サイズはインポートされません。
	ノートはそのアンカーごとに 1 回複製されます。

アクティビティ図	説明
アクティビティ	アクティビティにステレオタイプとその下のサブアクティビティの両方がある場合、正しくインポートまたは表示されません。
	塗りつぶしの色、フォント、フォント サイズはインポートされません。
分岐	塗りつぶしの色、フォント、フォント サイズはインポートされません。
オブジェクト	インポートされません。

クラス図	説明
クラス	多重度はインポートされません。
	インターフェイスの下のクラスはインポートされません。
	ネストされたクラスの属性と操作はインポートされません。
クラス属性	Static(静的属性) はインポートされません。
インターフェイス	属性はインポートされません。
パッケージ	フォント、フォント サイズ、塗りつぶしの色はインポートされません。
関連	派生はインポートされません。
	制約はインポートされません。

パッケージ図	説明
パッケージ	塗りつぶしの色、フォント、フォント サイズはインポートされません。

シーケンス図	説明
ライフライン	関連付けられているテキストが長い場合、水平方向の間隔がエクスポート後の XMI で正しく表示されないことがあります。
	塗りつぶしの色、フォント、フォント サイズはインポートされません。

シーケンス図	説明
メッセージ	インポート時にメッセージ間の垂直方向のスペースが追加されます。
	同じ Y 座標を持つメッセージは、異なるレベルのライフラインに入れられます。
	線の色、フォント、フォント サイズはインポートされません。
破棄マーカー	インポートされません。
ノート	インポート時にメッセージのアンカーが接続されません。

状態機械図	説明
状態	同じダイアグラムにステートが複数回存在する場合、1 つのシンボルのみイポートされます。
	塗りつぶしの色、フォント、フォント サイズはインポートされません。
分岐	塗りつぶしの色、フォント、フォント サイズはインポートされません。
遷移ライン	線の色、フォント、フォント サイズはインポートされません。

ユース ケース図	説明
アクター	サイズはインポートされません。
	多重度はインポートされません。
ユース ケース	ステレオタイプはインポートされません。
	サイズはインポートされません。
依存	ユース ケース間に描画されている場合はインポートされません。

エラー メッセージと警告メッセージ

エラー メッセージと警告メッセージが出力ウィンドウの **XMIExport** というタブに表示されます。これらのメッセージはすべてナビゲート可能です。

XMI で表現できない UML エンティティについては、エラー メッセージが生成されます。

UML エンティティが変換された場合、または包含階層で移動された場合、警告メッセージが表示されます。これは、Rational Rose がこのような構造を処理できないことによります。

16

CORBA IDL エクスポータ

ここでは、UML モデルから CORBA IDL をエクスポートする方法について説明します。

CORBA IDL エクスポータ

CORBA IDL エクスポータは提供されるアドインの 1 つで、以下に示す複数の部分から構成されています。

- CORBA プロファイル。UML から IDL へのマッピングに関係する複数のステレオタイプが含まれます。このパッケージには、ファイル オプション、名前付け規則などのコード生成オプションを制御するステレオタイプや、実際のコード生成を実行するエージェントも含まれます。
- CCM プロファイル。CCM（CORBA Component Model）と CIF（CORBA Implementation Framework）の組み込みを可能にする複数のステレオタイプが含まれます。
- CORBA パッケージ。複数の定義済みの IDL 型と型テンプレートが含まれます。コードの生成をカスタマイズし、UML モデルでネイティブの IDL 型を使用できるようにします。
- TCL スクリプト。以下のことを実行します。
  - IDL エクスポータに適用するコンテキスト メニューを制御します。
  - CORBA 固有メニューと CCM 固有メニューによるモデル要素へのステレオタイプの適用を簡素化します。

CORBA IDL アドインの起動

このアドインは `CORBAIDLGenerator` と呼ばれ、［ツール］メニューから［カスタマイズ］ダイアログを開き、［アドイン］タブで対応する入力項にチェックマークを付けることによって起動されます。

CORBA IDL アーティファクトの作成

CORBA IDL アーティファクトは、UML モデルからエクスポートする対象を決定するために使用されます。

このアーティファクトを作成するもっとも簡単な方法は、［モデル ビュー］で該当するモデル要素を右クリックし、［新しい CORBA IDL アーティファクト］メニュー項目を選択することです。このメニュー項目は、選択したモデル要素がパッケージ、クラスまたはインターフェイスのときだけ適用できます。また、CORBA IDL アドイン起動時に利用可能な CORBA メニューから、コマンドを使用することもできます。

IDL アーティファクトを作成する別の方法が配置図の作成です。この配置図のアーティファクトに、手動で、`<<IDLGenerator>>` ステレオタイプを適用します。さらに、IDL にエクスポートされるモデル要素とのマニフェスト依存関係を作成する必要があります。

注記

IDL を生成可能にするには、その前に、IDL アーティファクトをファイルに保存する必要があります。

IDLGenerator ステレオタイプ

<<IDLGenerator>> ステレオタイプは Artifact を拡張します。このステレオタイプの目的は、IDL に UML をエクスポートする際に、複数のオプションを使用できるようにすることにあります。

- **targetDir**：エクスポートした IDL ファイルを格納するディレクトリ。相対パス使用の場合、ターゲット ディレクトリはアーティファクトのモデル（.u2）ファイルが格納されるディレクトリに対する相対パスとなります。
- **fileName**：生成されたファイルのファイル名（ファイル マッピングが ONE_FILE の場合）。それ以外の場合は、最上位のファイルの名前。
- **fileSuffix**：生成されるすべてのファイル名に付加される接尾辞。
- **fileMapping**：モデル要素のファイルへのマッピング方法を規定します。利用できるオプションは、ONE_FILE、MODULE_STRUCTURED、および MODULE_FLAT です。
- **isAsyncDefault**：操作のデフォルトが一方向か、または一方向でないかを指示します。デフォルトは一方向（true）です。デフォルト値は、<<CORBAOperation>> ステレオタイプを適用し、タグ付き値 overrideIsDefaultAsync を設定することによって変更できます。

ファイル マッピング

ファイル マッピングが ONE_FILE の場合、アーティファクトが指示するすべてのモデル要素が生成され、ファイルオプションに基づく 1 つの IDL ファイルに保存されます。

MODULE_STRUCTURED の場合、モデル内の <<CORBAModule>> ステレオタイプが適用された各パッケージが生成され、独自のファイルに保存されます。各パッケージはファイル構造のディレクトリにも直接対応します。MODULE_FLAT の場合も、<<CORBAModule>> ステレオタイプが適用された各パッケージが生成され、独自のファイルに保存されます。ただし、この場合、すべてのファイル名はパッケージのスコープ名で修飾された名前となり、単一のディレクトリに格納されます。

複数のファイルが生成される場合、このファイル名は最上位のファイルにのみ適用されます。その他のすべての場合には、モジュール名がファイル名として使用されます。

IDL のエクスポート

IDL アーティファクトを作成したら、IDL のエクスポートが可能になります。エクスポートするには、IDL アーティファクトを右クリックし、[CORBA IDL の生成] を選択します。このコマンドは CORBA メニューからも利用できます。

エクスポートされたファイルは、デフォルトでは、アーティファクトの .u2 ファイルが格納されているディレクトリ内の独自のディレクトリに保存されます。ターゲット ディレクトリとファイル名は、上記のオプションを使用して変更できます。

モデル要素のマーキング

現在のところ、CORBA 固有のステレオタイプが適用されたモデル要素（674 ページの「CORBA プロファイル」参照）だけが、IDL マッピング時に考慮されます。

モデル要素をマークするもっとも簡単な方法は、［モデルビュー］またはダイアグラムからモデル要素を選択し、次に、CORBA メニューまたは CCM メニューを使用して、モデル要素が表す IDL 概念を選択することです。これにより、CORBA プロファイルまたは CCM プロファイルから該当するステレオタイプが自動的に適用されます。プロパティ エディタを使用して、操作が一方向か、そうでないかを指示するなど、設定の調整が必要なことがあります。

モデル要素を右クリックし、メニュー コマンド［ステレオタイプ ...］を選択することにより、モデル要素をマークすることもできます。表示されるダイアログボックスには適用可能なステレオタイプだけが表示されるので、適切なステレオタイプにチェックマークを付けます。

または、3 番目の方法として、該当するモデル要素を選択するときに、プロパティ エディタ内から［ステレオタイプ ...］を選択できます。

CORBA IDL データ型の使用

CORBA プロファイルを起動すると、UML でモデリング時に使用可能な IDL データ型を内蔵したパッケージ CORBA（モデル ビュー内のライブラリに所属）にアクセスします。このパッケージには、シーケンス、配列などの複数のテンプレート型も含まれます。これらのタイプを示す完全なリストが 671 ページの「定義済みの IDL 型」にあります。

定義済みの **IDL** 型

CORBA IDL エクスポータには、UML でモデリング時に使用できる複数の定義済みの IDL 型が用意されています。これらの型は、IDL エクスポート時に再使用されます。このことは、これらの IDL 型が、エクスポートした IDL をその後にマッピングするターゲットのプログラミング言語に依存しないことを意味しています。

単純型

定義済みの型は以下のとおりです。

```
short
long
long long
unsigned short
unsigned long
unsigned long long
float
double
long double
boolean
char
octet
wchar
string
wstring
Object
native
any
TypeCode
```

テンプレート型

以下のテンプレート型が定義されています。

```
sequence
array
string
wstring
fixed
```

sequence

sequence テンプレートには次の 2 つの形式があります。1 つは上限があり、もう 1 つは上限がありません。これらのシグニチャを以下に示します。

```
sequence<type T>
sequence<type T, const Natural index>
```

これらの sequence テンプレート UML での使用例を以下に示します。

```
// UML
<<CORBATypedef>> syntype Seq1 = sequence<long>;

<<CORBATypedef>> class Seq2 {
         sequence<long, 5> dummy;
```

```
}

<<CORBATypedef>> syntype Seq3 = sequence<Seq1>;
```

array

多次元配列は直接にはサポートされません。また角かっこは使用されないので、array テンプレートの動作は IDL で配列が動作する方法と異なります。array テンプレートのシグニチャを以下に示します。

```
array<type T, const Natural index>
```

注記

多次元配列を実現するには、下記の `MultiArray` のような「配列の配列」を定義して使用する必要があります。

この array テンプレート UML での使用例を以下に示します。

```
// UML
<<CORBATypedef>> syntype MyArr1 = array<4, long>;

<<CORBATypedef>> class MyArr2 {
        array<5, char> dummy;
}

<<CORBATypedef>> syntype MyArr3 = array<6, MyArr2>;

<<CORBATypedef>> syntype MultiArray = array<3, MyArr1>;
```

string

string 型には 2 つの形式がありますが、そのうちの 1 つだけが template 型です。これらのシグニチャを以下に示します。

```
string
string<const Natural size>
```

string と string テンプレート UML での使用例を以下に示します。

```
<<CORBATypedef>> syntype MyStr1 = string;

<<CORBATypedef>> class MyStr2 {
        string<5> dummy;
}

<<CORBAInterface>> interface I3 {
        string a;
        MyStr1 b;
        MyStr2 c;
};
```

wstring

wstring 型には 2 つの形式がありますが、そのうちの 1 つだけが template 型です。これらのシグニチャを以下に示します。

```
wstring
wstring<const Natural size>
```

wstring と wstring テンプレート UML での使用例を以下に示します。

```
<<CORBATypedef>> syntype MyStr1 = wstring;

<<CORBATypedef>> class MyStr2 {
        wstring<5> dummy;
}

<<CORBAInterface>> interface I3 {
        wstring a;
        MyStr1 b;
        MyStr2 c;
};
```

fixed

fixed テンプレートのシグニチャを以下に示します。

```
fixed<const Natural digit, const Natural scale>
```

この fixed テンプレート UML での使用例を以下に示します。

```
// UML
<<CORBAInterface>> interface I5 {
        fixed<7, 3> a;
};
```

定義済みの **UML** 型

Integer、Natural、Real などの定義済みの UML 型を使用できます。IDL へのこれらのマッピングを 691 ページの「定義済みの型」に説明します。

CORBA プロファイル

UML を IDL にエクスポートするとき、以下のステレオタイプを使用してマッピングを制御します。対応するマッピングについては、681 ページの「マッピング ルール」にその概要を説明します。

ここで説明するステレオタイプは、OMG の CORBA 仕様向けの UML プロファイルから使用できるものです。ただし、UML 2.0 に対応するように更新されているので、サポートされるステレオタイプ、拡張するメタクラスなどに違いがあります。

CORBA

`<<CORBA>>` ステレオタイプは抽象であり、他のステレオタイプに対するスーパークラスとしてのみ使用されます。

CORBAAttribute

`<<CORBAAttribute>>` ステレオタイプは Attribute を拡張します。詳細については、682 ページの「属性」を参照してください。

属性が定数値を表すことを示すために、タグ付き値 `isConstant` が使用されます。属性を直接パッケージに配置できるので、ユーティリティ クラスを使用してモジュール レベル定数を表す必要はありません。

属性が読み取り専用であることを表すために、メタ特性 `isReadOnly` が使用されます。これは、ベース メタクラス `Attribute` のプロパティです。

属性のステレオタイプは、ブラウザからクラス図などにドラッグすることで、可視化が可能です。

CORBABoxedValue

`<<CORBABoxedValue>>` ステレオタイプは、Syntype と Class を拡張します。

CORBAEnum

`<<CORBAEnum>>` ステレオタイプは DataType を拡張します。詳細については、685 ページの「列挙」を参照してください。

CORBAException

`<<CORBAException>>` ステレオタイプは Class を拡張します。詳細については、685 ページの「例外」を参照してください。

CORBAInclude

`<<CORBAInclude>>` ステレオタイプは Dependency を拡張します。詳細については、686 ページの「インクルード」を参照してください。

CORBAInterface

<<CORBAInterface>> ステレオタイプは Interface を拡張します。詳細については、687 ページの「インターフェイス」を参照してください。下位互換性を保つため、このステレオタイプは Class も拡張します。CORBA 仕様ではこのアプローチのみを使用していますが、この使用方法は推奨できません。UML 2.0 では、Interface メタクラスによって CORBA インターフェイスへのもっと自然なマッピングが提供されています。

インターフェイスがローカルか、ローカルでないかを示すためにタグ付き値 isLocal が使用されます。デフォルトでは、インターフェイスはローカルではありません。

CORBAModule

<<CORBAModule>> ステレオタイプは Package を拡張します。詳細については、688 ページの「パッケージ」を参照してください。

CORBAOperation

<<CORBAOperation>> ステレオタイプは Operation と Signal を拡張します。詳細については、688 ページの「操作」と 692 ページの「シグナル」を参照してください。

これには、操作（一方向の操作）についてのデフォルトのマッピングを上書きするために使用されるタグ付き値 overrideIsDefaultAsync が含まれています。

これには、操作に関する文字列を運ぶ別のタグ付き値 context が含まれています。

属性が読み取り専用であることを表すために、メタ特性 isReadOnly を使用します。これは、ベース メタクラス Attribute のプロパティです。

操作のステレオタイプは、ブラウザからクラス図などにドラッグすることで、可視化が可能です。

CORBASequence

<<CORBASequence>> ステレオタイプは Class を拡張します。詳細については、691 ページの「シーケンス」を参照してください。

CORBAStruct

<<CORBAStruct>> ステレオタイプは Class を拡張します。詳細については、692 ページの「構造体」を参照してください。

CORBATruncatable

<<CORBATruncatable>> ステレオタイプは、Generalization を拡張します。

CORBATypedef

<<CORBATypedef>> ステレオタイプは Syntype、Class、DataType、Interface を拡張します。詳細については、692 ページの「シンタイプ」を参照してください。

CORBAUnion

<<CORBAUnion>> ステレオタイプは Class を拡張します。詳細については、693 ページの「共用体」を参照してください。

これにはタグ付き値 isSimple が含まれています。このタグ付き値は、弁別子を使って自動的に long 型に設定される単純な共用体で、case ラベルは 0、1、2 などに設定されます。このモデルではデフォルトの case は定義できません。

関連するステレオタイプは case と discriminator です。

CORBAValue

<<CORBAValue>> ステレオタイプは、Interface を拡張します。

値がカスタム マーシャリングを使用するかどうかを示すため、タグ付き値 isCustom が使用されます。詳細については、694 ページの「値」を参照してください。

CORBAValueFactory

<<CORBAValueFactory>> ステレオタイプは、Operation を拡張します。

case

<<case>> ステレオタイプは Attribute を拡張します。case を使用できるのは <<CORBAUnion>> ステレオタイプが適用されたクラス上のみです。

このステレオタイプは、case の選択（「スイッチ」）に使用される条件を表すタグ付き値 label を所有します。

また、case がデフォルトのものであることを示すために使用されるタグ付き値 isDefault も所有します。この値が true の場合、case ラベルは無視されます。

共用体は、isDefault が true に設定された case を 1 つしか所有できません。

discriminator

<<discriminator>> ステレオタイプは Attribute を拡張します。discriminator を使用できるのは、<<CORBAUnion>> ステレオタイプが適用されたクラス上のみで、そのような各共用体は 1 つの弁別子のみを持つことができます。

IDLFile

<<IDLFile>> ステレオタイプは Artifact を拡張します。詳細については、686 ページの「インクルード」を参照してください。

これはタグ付き値ファイルを所有します。このファイルを使って、アーティファクトが表す実際の IDL ファイルを指示します。

IDLGenerator

<<IDLGenerator>> ステレオタイプは Artifact を拡張します。詳細については、669 ページの「IDLGenerator ステレオタイプ」を参照してください。

外部ステレオタイプ

CORBA プロファイル仕様では以下のステレオタイプが定義されていますが、IDL エクスポータではサポートされていません。これらのステレオタイプは、CORBA プロファイル仕様で抽象として定義されているか（この場合はエクスポータに対する実用的な目的はありません）、あるいは前述のように他のステレオタイプまたは構成要素に置き換えられます。

- CORBAAnonymousArray（Array テンプレートを参照）
- CORBAArray（Array テンプレートを参照）
- CORBAAnonymousSequence（Sequence テンプレートを参照）
- CORBAConstant （CORBAAttribute テンプレートを参照）
- CORBAConstants（CORBAAttribute テンプレートを参照）
- CORBAConstructedType（抽象）
- CORBACustomValue（CORBAValue テンプレートを参照）
- CORBAUserDefinedType（抽象）
- CORBAIndexedType（抽象）
- CORBAObjectType（抽象）
- CORBAStructType（抽象）
- CORBAStructuredType（抽象）
- CORBAUserDefinedType（抽象）
- CORBAValueSupports（CORBAValues と CORBAInterfaces 間での汎化によって黙示的に）
- CORBAWrapper（抽象）
- oneway（CORBAOperation を参照）
- readonly（CORBAOperation および CORBAAttribute を参照）
- readonlyEnd（冗長）
- switch（discriminator を参照）
- switchEnd（冗長）

CCM プロファイル

IDL に UML をエクスポートするとき、以下のステレオタイプを使用してマッピングを制御します。対応するマッピングについては、681 ページの「マッピング ルール」にその概要を説明します。

ここで説明するステレオタイプは、OMG の CCM 仕様向けの UML プロファイルから使用できるものです。ただし、UML 2.0 に対応するように更新されているので、サポートされるステレオタイプ、拡張するメタクラスなどに違いがあります。

サポートされるステレオタイプ

CORBAArtifact

<<CORBAArtifact>> ステレオタイプは Class を拡張します。詳細については、681 ページの「アーティファクト」を参照してください。

CORBAComponent

<<CORBAComponent>> ステレオタイプは Class を拡張します。詳細については、683 ページの「コンポーネント」を参照してください。

CORBAComponentImpl

<<CORBAComponentImpl>> ステレオタイプは Class を拡張します。詳細については、683 ページの「コンポーネント」を参照してください。

このステレオタイプは、タグ付き値 `category` を所有します。これは、ステレオタイプが指し示すコンポーネントの種類を表します。列挙で使用できるリテラルは、`entity`、`process`、`service`、または `session` です。

このステレオタイプは、タグ付き値 `composite` も所有します。これは、包含される合成の名前を示すために使用されます。

CORBAEvent

<<CORBAEvent>> ステレオタイプは Interface を拡張します。詳細については、685 ページの「イベント」を参照してください。

CORBAEventSink

<<CORBAEventSink>> ステレオタイプは Port を拡張します。詳細については、689 ページの「ポート」を参照してください。

このステレオタイプは、CCM 仕様の <<CORBAConsumes>> を置き換えます。ポートの概念によってより自然なマッピングを提供できるからです。

CORBAEventSource

<<CORBAEventSource>> ステレオタイプは Port を拡張します。タグ付き値 `sourceKind` を所有します。これは、使用可能なリテラルである `Emitter` または `Publisher` のついた列挙です。詳細については、689 ページの「ポート」を参照してください。

このステレオタイプは、CCM 仕様の <<CORBAPublishes>> および <<CORBAEmits>> を置き換えます。ポートの概念によってより自然なマッピングを提供できるからです。

CORBAFacet

<<CORBAFacet>> ステレオタイプは Port を拡張します。詳細については、689 ページの「ポート」を参照してください。

このステレオタイプは、CCM 仕様の `<<CORBAProvides>>` を置き換えます。ポートの概念によってより自然なマッピングを提供できるからです。

CORBAFactory

<<CORBAFactory>> ステレオタイプは Operation を拡張します。詳細については、686 ページの「ホーム」を参照してください。

CORBAFinder

<<CORBAFinder>> ステレオタイプは Operation を拡張します。詳細については、686 ページの「ホーム」を参照してください。

CORBAHome

<<CORBAHome>> ステレオタイプは、<<CORBAInterface>> の特殊な種類です。詳細については、686 ページの「ホーム」を参照してください。

CORBAHomeImpl

<<CORBAHomeImpl>> ステレオタイプは Class を拡張します。詳細については、686 ページの「ホーム」を参照してください。

CORBAImplements

<<CORBAImplements>> ステレオタイプは Dependency を拡張します。詳細については、687 ページの「実装」を参照してください。

CCM 仕様では、このステレオタイプは Association を拡張しますが、メタクラス Dependency が運ぶオーバーヘッドはかなり少なくなります。

CORBAIsProvidedBy

<<CORBAIsProvidedBy>> ステレオタイプは Dependency を拡張します。

CORBAManages

<<CORBAManages>> ステレオタイプは Dependency を拡張します。詳細については、687 ページの「管理」を参照してください。

CCM 仕様では、このステレオタイプは Association を拡張しますが、メタクラス Dependency が運ぶオーバーヘッドはかなり少なくなります。

CORBAPrimaryKey

<<CORBAPrimaryKey>> ステレオタイプは、Attribute を拡張します。

CORBAReceptacle

<<CORBAReceptacle>> ステレオタイプは Port を拡張します。詳細については、689 ページの「ポート」を参照してください。

このステレオタイプは、タグ付き値 `isMultiple` を所有します。これは、複数接続が許可されるかどうかを示すために使用されます。

このステレオタイプは、CCM 仕様の <<CORBAUses>> を置き換えます。ポートの概念によってより自然なマッピングを提供できるからです。

CORBASegment

<<CORBASegment>> ステレオタイプは Class を拡張します。詳細については、691 ページの「セグメント」を参照してください。

このステレオタイプは、タグ付き値 `features` を所有します。これは、セグメントによってサポートされる機能を示すために使用されます。

また、セグメントが連続しているかどうかを示すタグ付き値 `isSerialized` も所有します。

CORBASupports

<<CORBASupports>> ステレオタイプは Generalization を拡張します。

外部ステレオタイプ

CCM プロファイル仕様では以下のステレオタイプが定義されていますが、IDL エクスポータではサポートされていません。これらのステレオタイプは、CCM プロファイル仕様で抽象として定義されているか（この場合はエクスポータに対する実用的な目的はありません）、あるいは前述のように他のステレオタイプに置き換えられます。

- CORBAConsumes（CORBAEventSink を参照）
- CORBAEmits（CORBAEventSource を参照）
- CORBAEventPort（抽象）
- CORBAProvides（CORBAFacet を参照）
- CORBAPublishes（CORBAEventSource を参照）
- CORBAUses（CORBAReceptacle を参照）

マッピング ルール

このセクションでは、さまざまな UML 構成要素を IDL にエクスポートするとき、それらの構成要素をマッピングする方法について説明します。

ほとんどの場合、IDL を生成するため、UML 構成要素に関係する CORBA ステレオタイプを適用する必要があります。

注記

1 つのモデル要素に複数の CORBA ステレオタイプを適用しても、IDL 生成時にはエクスポータが 1 つの CORBA ステレオタイプだけを選択します。

アーティファクト

UML で `<<CORBAArtifact>>` ステレオタイプが適用されているクラスは、IDL 内のセグメントのアーティファクトにマッピングされます。`<<CORBAIsProvided>>` 依存関係によってセグメントに関連付けられていないクラスの場合は、IDL は生成されません。691 ページの「セグメント」も参照してください。683 ページの「コンポーネント」に例を示しています。

関連

関連のマッピングは直接的には行なわれません。その代わりに、誘導可能な各関連端は、所有者に基づいて、IDL の属性、ケースまたはフィールドとしてマッピングされます。関連端が多重度の場合、各端のマッピングは687 ページの「多重度」に記述されているように行なわれます。

大部分の場合、エクスポートされる関連端について、所有する分類子をマークすることで十分です。693 ページの「共用体」に説明しているように、注意すべき例外は共用体の case と弁別子です。

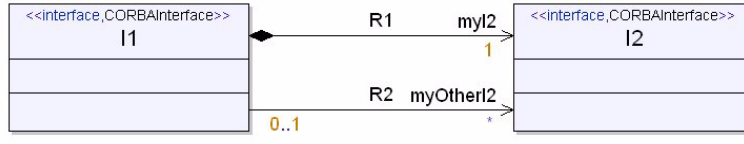


図 186

681 ページの図 186 の UML モデルは、次の IDL にマッピングされます。

```
interface I2 {
};

interface I1 {
        attribute I2 myI2;
        attribute sequence<I2> myOtherI2;
};
```

属性

UML で <<CORBAAttribute>> ステレオタイプが適用されている属性は、それが定数でなければ、IDL 内の属性にマッピングされます（684 ページの「定数」参照）。マークされているクラスのすべての属性がエクスポートされるので、通常、ステレオタイプの使用は必ずしも必要ではありません。

UML の以下のコード例は、

```
<<CORBAInterface>> class C1 {
        long a;
        char [0..1] b;
        wchar [*] c;
        boolean d;
}

<<CORBAStruct>> class S1 {
        string e;
        爽nsigned longí [8] f;
        C1 [1] g;
}
```

次の IDL にマッピングされます。

```
interface C1 {
        attribute long a;
        attribute sequence<char, 1> b;
        attribute sequence<wchar> c;
        attribute boolean d;
};

struct S1 {
        string e;
        sequence<unsigned long, 8> f;
        C1 g;
};
```

制限事項

定数でない属性を IDL エクスポート対象とするには、クラスまたはインターフェイスの一部として定義します。つまり、パッケージ直下で定義される属性は IDL にはマッピングされません。

クラス

UML で <<CORBAInterface>> ステレオタイプが適用されているクラスは、IDL 内のインターフェイスにマッピングされます。この例を 682 ページの「属性」に示します。

UML で <<CORBAStruct>> ステレオタイプが適用されているクラスは、IDL 内の構造体にマッピングされます。併せて 692 ページの「構造体」も参照してください。

UML で <<CORBAUnion>> ステレオタイプが適用されているクラスは、IDL 内の共用体にマッピングされます。併せて 693 ページの「共用体」も参照してください。

UML で <<CORBAException>> ステレオタイプが適用されているクラスは、IDL 内の例外にマッピングされます。併せて 685 ページの「例外」も参照してください。

UML で <<CORBATypedef>> ステレオタイプが適用されているクラスは、IDL 内の例外にマッピングされます。併せて 693 ページの「型定義」も参照してください。

制限事項

クラスを IDL エクスポート対象とするには、パッケージの一部として定義します。つまり、インラインとして宣言されるクラスは IDL になりません。

コメント

エクスポートするモデル要素に付加されるコメントは IDL コメント（前に「//」が付く）となります。コメントはコメント シンボルで記述されるか、またはコメント用のプロパティ エディタ フィールドに記述されます。

コンポーネント

UML 内で <<CORBAComponent>> ステレオタイプが適用されているクラスは、IDL 内のコンポーネントにマッピングされます。同様に、UML 内で <<CORBAComponentImpl>> ステレオタイプが適用されているクラスは、IDL 内のコンポーネント合成にマッピングされます。これには、管理対象のホーム実装への参照が含まれています。UML 内の <<CORBAComponent>> ステレオタイプが適用されている、コンポーネント実装からコンポーネントへの依存関係は、IDL 内の合成コンポーネントから対応するコンポーネントへの実装関係にマッピングされます。

UML 内でコンポーネント実装がパーツを持っており、<<CORBASegment>> ステレオタイプが適用されたクラスによってパーツが型付けされている場合、IDL 内で合成コンポーネントは（そのアーティファクトとともに）それらのセグメントを示します。

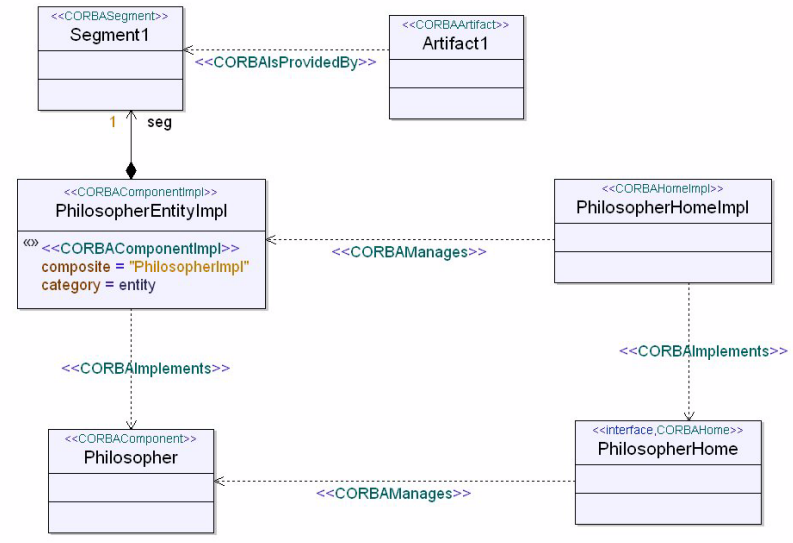


図 187

684 ページの図 187 に示す例は、以下の IDL にマッピングされます。

```
composite entity PhilosopherImpl {
        home executor PhilosopherHomeImpl {
                implements PhilosopherHome;
                manages PhilosopherEntityImpl {
                        segment Segment1 {
                                provides (Artifact1);
                        };
                };
        };
};
```

定数

UML で <<CORBAAttribute>> ステレオタイプが適用され、しかも定数である属性は、IDL 内の定数にマッピングされます。

注記

IDL へのマッピング時に、多重度として使われる定数は整数値で評価されます。

UML の以下のコード例は、

```
<<CORBAModule>> package M5 {
        <<CORBAAttribute(.isConstant=true.)>>
        const Integer nval = 5;

        <<CORBAAttribute(.isConstant=true.)>>
```

```
        const Integer mval = nval;

        <<CORBAInterface>> class C1 {
                <<CORBAAttribute(.isConstant = true.)>>
                const Real cval = 22.7;

                long [nval] x;
        }
}
```

次の IDL にマッピングされます。

```
module M5 {
        const Integer nval = 5;
        const Integer mval = nval;

        interface C1 {
                const double cval = 22.7;
                attribute sequence<long, 5> x;
        };
};
```

注記

定数式は評価されません。つまり、整数リテラルおよび他の定数だけが現在サポートされています。

列挙

UML で <<CORBAEnum>> ステレオタイプが適用されている列挙は、IDL 内の列挙にマッピングされます。

UML の以下のコード例は、

```
<<CORBAEnum>> enum Color {
        red,
        green,
        blue
}
```

次の IDL にマッピングされます。

```
enum Color {
        red,
        green,
        blue
};
```

イベント

UML で <<CORBAEvent>> ステレオタイプが適用されているインターフェイスは、IDL 内のイベント型にマッピングされます。

例外

UML で <<CORBAException>> ステレオタイプが適用されているクラスは、IDL 内の例外にマッピングされます。

UML の以下のコード例は、

```
<<CORBAException>> class Error {
        string e;
```

```
}

<<CORBAInterface>> class C1 {
        <<CORBAOperation(.overrideIsDefaultAsync = true.)>>
        void op() throw Error();
}
```

次の IDL にマッピングされます。

```
exception Error {
        e string;
};

interface C1 {
        void op() raises (Error);
};
```

ホーム

UML で <<CORBAHome>> ステレオタイプが適用されているインターフェイスは、IDL 内のホーム宣言にマッピングされます。ホーム インターフェイスはプライマリキーがある場合があります、これは、UML 内では <<CORBAPrimaryKey>> が適用された依存関係または関連で表され、依存関係の供給者は値型を持つ必要があります。

683 ページの「コンポーネント」に例を示しています。UML で <<CORBAFactory>> ステレオタイプが適用されている操作は、IDL 内のファクトリ操作にマッピングされます。同様に、UML で <<CORBAFinder>> ステレオタイプが適用されている操作は、IDL 内のファインダ操作にマッピングされます。

UML の以下のコード例は、

```
<<CORBAHome>> interface Ifc {
        <<CORBAFactory>> void myFactory(in boolean b);
        <<CORBAFinder>> void myFinder(in long a);
}
```

次の IDL にマッピングされます。

```
home Ifc {
        factory myFactory(in boolean b);
        finder myFinder(in long a);
};
```

インクルード

アーティファクトから複数のファイルを生成した場合に必要となる include 文は、IDL 生成時に自動的に作成されます。また、既存の IDL ファイルを参照する必要がある場合に、生成された IDL に手動で include 文を追加するのではなく、モデル内でその関係を表現しておいて生成させる方法もあります。このために、<<CORBAInclude>> ステレオタイプが用意されています。

UML で <<IDLGenerator>> ステレオタイプが適用されているアーティファクトと <<IDLFile>> ステレオタイプが適用されているアーティファクトの間のこのような依存関係は、指示されたファイルの include 文にマッピングされます。

重要！

<<CORBAInclude>> ステレオタイプは、正しくマークされているアーティファクト間で使用されるときだけ考慮されます。特に、これは、パッケージ間の通常のアクセスまたはインポートの依存関係の置換ではありません。

実装

UML で <<CORBAImplements>> ステレオタイプが適用されているクラスは、IDL 内の implements 文にマッピングされます。依存関係の方向は、必ずコンポーネント実装からそのコンポーネント、あるいはホーム実装からそのホーム インターフェイスとなります。

683 ページの「コンポーネント」に例を示しています。

インターフェイス

UML で <<CORBAInterface>> ステレオタイプが適用されているインターフェイスは、IDL 内のインターフェイスにマッピングされます。

制限事項

インターフェイスを IDL エクスポート対象とするには、パッケージの一部として定義します。

管理

UML で <<CORBAManages>> ステレオタイプが適用されている依存関係は、IDL 内の manages 文にマッピングされます。依存関係の方向は、必ずホーム実装からコンポーネント実装、あるいはホーム インターフェイスからコンポーネントとなります。

683 ページの「コンポーネント」に例を示しています。

多重度

多重度は複数の範囲から構成できますが、指定できる値の型は整数リテラルまたは定数だけです。多重度は評価されて、すべての有効な値を包含する 1 つの範囲にまとめられます。すべての多重度はシーケンスにマッピングされます。シーケンスはバインドされるか、バインドされないかのいずれかです。ただし、多重度 1 のシーケンスを除きます。

複数範囲の場合、これらの範囲の最大値が上限となります。

UML	IDL
`long [1]`	`long`
`long [0..1]`	`sequence<long, 1>`
`long [0..*]`	`sequence<long>`

UML	IDL
long [1..*]	sequence<long>
long [6..*]	sequence<long>
long [3..8]	sequence<long, 8>
long [7..7]	sequence<long, 7>
long [1, 7..9, 3, 5]	sequence<long, 9>

操作

UML で <<CORBAOperation>> ステレオタイプが適用されている操作は、そのタグ付き値に基づいてマッピングされます。デフォルトで、IDL では操作は one way(一方向の操作) としてマッピングされます。overrideIsDefaultAsync ステレオタイプを「true」に設定することにより、この IDL 操作は IDL の通常の操作として生成されます。

UML の以下のコード例は、

```
<<CORBAInterface>> interface I1 {
        signal op1;

        <<CORBAOperation(.overrideIsDefaultAsync = true.)>>
        long op2();

        void op3(in long a);

        <<CORBAOperation(.overrideIsDefaultAsync = true.)>>
        string<5> op4(inout char b, in long c);
}
```

次の IDL にマッピングされます。

```
interface I1 {
        oneway void op1();
        long op2();
        oneway void op3(in long a);
        string<5> op4(inout char b, in long c);
};
```

制限事項

操作が一方向の操作にマッピングされた場合、in パラメータだけが使用可能となります。

操作を IDL エクスポートの対象とするには、クラスまたはインターフェイスの一部として定義します。

パッケージ

UML で <<CORBAModule>> ステレオタイプが適用されているクラスは、IDL 内のモジュールにマッピングされます。

UML の以下のコード例は、

```
<<CORBAModule>> package P1 {
        <<CORBAInterface>> interface I2 {
                long a;
        }
}
```

次の IDL にマッピングされます。

```
module P1 {
        interface I2 {
                attribute long a;
        };
};
```

パラメータ

パラメータは、所有する操作またはシグナルに常に関連づけられているので、CORBA ステレオタイプを適用する必要はありません。

UML のパラメータの「方向」は、そのまま IDL でのパラメータの「方向」に一致します。

UML	**IDL**
in	in
inout	inout
out	out
return	return

ポート

ポートを IDL にマッピングできるステレオタイプはいくつかあります。

- `<<CORBAFacet>>` ステレオタイプが適用されているポートは provides にマッピングされます。
- `<<CORBAReceptacle>>` ステレオタイプが適用されているポートは uses にマッピングされます。
- `<<CORBAEventSink>>` ステレオタイプが適用されているポートは consumes にマッピングされます。
- `<<CORBAEventSource>>` ステレオタイプが適用されているポートは emits または publishes にマッピングされます。

マッピングをより視覚的に識別するには、690 ページの図 188 に示すように、ポートにアイコンを割り当てます。これは、icon ステレオタイプを使用して設定します。

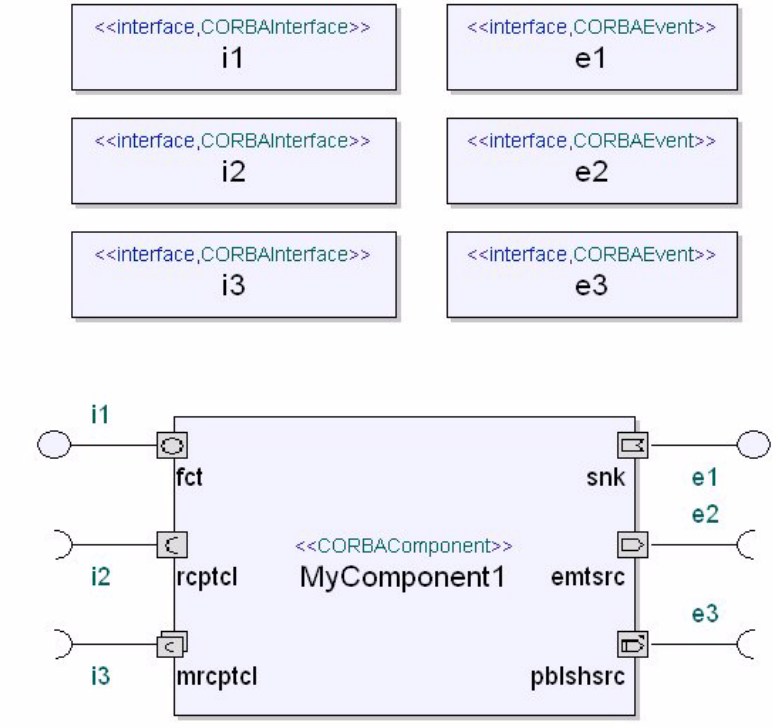


図 188

上記の例は、以下の IDL にマッピングされます。

```
interface i1 { };
interface i2 { };
interface i3 { };
eventtype e1 { };
eventtype e2 { };
eventtype e3 { };

component MyComponent1 {
        provides i1 fct;
        uses i2 rcptcl;
        uses multiple i3 mrcptcl;
        consumes e1 snk;
        emits e2 emtsrc;
        publishes e3 pblshsrc;
```

```
};
```

定義済みの型

単純な定義済みの IDL 型（671 ページの「定義済みの IDL 型」参照）はそのままマッピングされます。

テンプレート型（671 ページの「定義済みの IDL 型」参照）は対応する IDL の型にマッピングされます。

また、以下の表の定義済みの UML データ型も使用てきます。IDL へのマッピングは下表のとおりです。

UML	**IDL**
Natural	unsigned long
Integer	long
Boolean	boolean
Character	char
Real	double
Charstring	string

制限事項

`<<CORBAOperation>>` ステレオタイプをシグナルに適用する場合、タグ付き値 `isOneway` を「true」に設定する必要があります。

セグメント

UML で `<<CORBASegment>>` ステレオタイプが適用されているクラスは、IDL 内のコンポーネント実装のセグメントにマッピングされます。コンポーネント実装のパーツではないクラスの場合は、IDL は生成されません。681 ページの「アーティファクト」モサンショウしてください。683 ページの「コンポーネント」に例を示しています。

シーケンス

UML で `<<CORBASequence>>` ステレオタイプが適用されているクラスは、シーケンスにマッピングされます。シーケンスの型は属性を使って指定されます。属性の名前は重要ではなく、その型だけが考慮されます。これは sequence テンプレート型の使用に代わるものです。

UML の以下のコード例は、

```
<<CORBASequence>> class MySeq {
        string dummy;
```

```
}

<<CORBASequence>> class MyOtherSeq {
        MySeq x;
}

<<CORBASequence>> class MyThirdSeq {
        MyOtherSeq y;
}
```

次の IDL にマッピングされます。

```
typedef sequence<string> MySeq;

typedef sequence<MySeq> MyOtherSeq;

typedef sequence<MyOtherSeq> MyThirdSeq;
```

シグナル

UML で <<CORBAOperation>> ステレオタイプが適用されているクラスは、one way(一方向の操作) にマッピングされます。例については、688 ページの「操作」を参照してください。

注記

この結果は、<<CORBAOperation>> ステレオタイプが適用された UML 操作と同じマッピングとなります。この場合、タグ付き値 isOneway は「true」に設定され、in パラメータだけが使われます。

制限事項

シグナルを IDL エクスポートの対象とするには、パッケージの一部として定義します。

構造体

UML で <<CORBAStruct>> ステレオタイプが適用されているクラスは、IDL 内の構造体にマッピングされます。例については、682 ページの「属性」を参照してください。

シンタイプ

UML で <<CORBATypedef>> ステレオタイプが適用されているクラスは、IDL 内の typedef にマッピングされます。これは型定義の使用に代わるものです。

UML の以下のコード例は、

```
<<CORBATypedef>> syntype MyOtherType = string;
<<CORBATypedef>> syntype SeqLong = sequence<long>;
```

次の IDL にマッピングされます。

```
typedef string MyOtherType;
typedef sequence<long> SeqLong;
```

型定義

UML で <<CORBATypedef>> ステレオタイプが適用されているクラスは、IDL 内の typedef にマッピングされます。これはシンタイプの使用に代わるものです。

UML の以下のコード例は、

```
<<CORBATypedef>> class MyType {
        long dummy;
}
```

次の IDL にマッピングされます。

```
typedef long MyType;
```

汎化を使用して型定義を定義することもできます。ただし、汎化は、2 つのデータ型、2 つのインターフェイス、あるいは 2 つのクラスなど、同種の要素間でのみ使用できます。

UML の以下のコード例は、

```
<<CORBATypedef>> datatype MyBool : boolean {
}
```

次の IDL にマッピングされます。

```
typedef boolean MyBool;
```

重要！

シーケンスと配列などのテンプレート型は、属性およびパラメータに対して無記名で使用するのではなく（attr: sequence<long> のような書式）、変数の型定義を行って（attr: SeqLong のような形）から使用する方が優れています。

共用体

UML で <<CORBAUnion>> ステレオタイプが適用されているクラスは、IDL 内の共用体にマッピングされます。<<discriminator>> ステレオタイプが適用されている属性の型は共用体の弁別子にマッピングされます。discriminator と表記されている属性の名前は無視されます。<<case>> ステレオタイプが適用されている属性のタグ付き値のラベルは、共用体のメンバーの case ラベルにマッピングされます。

以下の UML のコード例は、

```
<<CORBAUnion (.isSimple = true.)>> class U1 {
        long a;
        char b;
        string c;
}

<<CORBAUnion>> class U2 {
        <<discriminator>> Color d;
        <<‘case’(.label = "green”.)>> long e;
        <<‘case’(.label = “blue”.)>> boolean f;
        <<‘case’(.isDefault = true.)>> string g;
}
```

次の IDL にマッピングされます。

```
union U1 switch (long) {
        case 0: long a;
        case 1: char b;
```

```
        case 2: string c;
};

union U2 switch (Color) {
        case green: e long;
        case blue: f boolean:
        default: g string;
};
```

制限事項

共用体が simple でない場合、<<discriminator>> または <<case>> ステレオタイプが適用されているクラスの属性だけが IDL エクスポートの対象です。一方、共用体が simple の場合、<<discriminator>> と <<case>> は無視されます。

値

UML で <<CORBAValue>> ステレオタイプが適用されているインターフェイスは、IDL 内の値型にマッピングされます。

既知の制限事項

anonymous 型

anonymous テンプレート型は限られた IDL コンパイラでしか動作しません。

値型

IDL コード ジェネレータは CORBA 向け UML プロファイルのサブセットの大半を実装していますが、すべてを実装しているわけではありません。このエクスポータでは値型はサポートされません。

OMG での仕様は UML1.X に基づいているため、値型の実装は UML2.1 に適するように修正されています。

C++ サポート

ORB と連携する C++ コード例の生成や CORBA IDL インポータの提供は行われません。

17

MSVS ソリューションファイルのインポート

本章では Microsoft Visual Studio ソリューションファイル (.sln ファイル) を Tau にインポートしてコンポーネント間の依存を可視化する方法を説明します。

概要

MSVS sln インポータは Microsoft Visual Studio (MSVS) Solution ファイルを Tau にインポートするために使用します。インポートされると、ソリューションによって生成されたコンポーネントとコンポーネント間の依存は、UML ダイアグラム上で図形として表示できます。Tau で作成された 1 つのコンポーネントがソリューション内の 1 つの .vcproj ファイル (MSVS プロジェクトファイル) に相当します。

インポータのデフォルト設定は、コンポーネントと .vcproj ファイルの一対一関係を作成するので、生成されたグラフは .vcproj 間の依存を表示するのにも役立ちます。インポートされたライブラリを表現するアーティファクトを生成することも可能です。このオプションはソリューションのすべてのコンポーネントの表示を可能にします。1 つの .vcproj ファイルは結果として 2 つのアーティファクトとなります。1 つは、生成の主たる出力であり、もう 1 つは、対応する「インポート」ライブラリです。

はじめに

sln ファイルをインポートするには :

- Tau を起動して、新規プロジェクトを作成するか既存のプロジェクトを開きます
- ［ファイル］>［インポート］を選択して、［インポートウィザード］を開始します。
- ［MSVS ソリューション］のインポートを選択して［OK］をクリックします。
- MSVS ソリューションインポートウィザード を続行します。

注記

MSVS ソリューションインポータがサポートするのは。Visual Studio .NET 2008 だけであることに注意してください。

MSVS ソリューションインポートウィザード

このセクションでは、MSVS ソリューションインポートウィザードを説明します。

MSVS ソリューションインポートウィザードの第 1 ステップ

インポートプロセスの最初のステップは、インポートするソリューションファイルと構成とプラットフォームの指定、および追加のインポートオプションの指定です。.

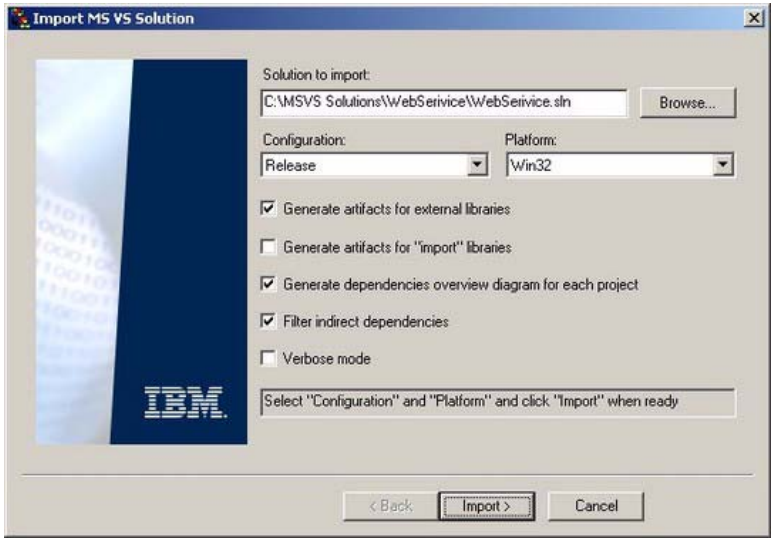


図 189: MSVS sln インポートウィザードの第 1 ステップ

- はじめに、［参照］ボタンを押してソリューションファイルを選択します。その後、ウィザードはファイルを読み込み、ソリューションに合った構成とプラットフォームを表示します。
- 構成とプラットフォームを選択します。
- オプションで、下記のオプションを選択または選択解除します。
- ［インポート］を選択します。

［外部ライブラリ用アーティファクトの生成］を選択解除すると、vcproj ファイル内で参照されるライブラリのためのアーティファクトと依存は生成されません。ソリューション内の vcproj ファイルによって使用されるのは、Visual Studio .NET ライブラリ、Windows DLL、および他の外部コンポーネントです。

［「インポート」ライブラリ用アーティファクトの生成］を選択して vcproj ファイルがあるライブラリを vcproj ファイル内の ［リンカ］->［詳細］セクションの［ライブラリのインポート］オプションで指定していると、そのライブラリと依存を表現するアーティファクトが生成されます。

［各プロジェクトの Dependency overview dialog を生成］が選択解除されていると、1 つのダイアグラムのみが生成されます。そのダイアグラムはすべてのアーティファクトとその依存を表示します。このオプションが選択されると、インポータは各 vcproj ファイルごとに 1 つのダイアグラムを生成し、コンポーネントとその依存を特定のプロジェクト用に表示します。

vcproj ファイルには、指定したコンポーネントをビルドするのに必要な依存をすべて指定する必要があります。間接的な依存も vcproj ファイルに存在する必要があります。ソリューションファイルのインポートの結果、すべての間接的な依存を表現した非常に複雑な概観図を生成できます。［間接参照をフィルタリング］オプションは間接依存を生成せずに依存グラフを最適化します。このオプションを選択解除すると、vcproj ファイル中のすべての依存が表示されます。

MSVS ソリューションインポータを使用するには Visual Studio .NET がインストールされている必要があります。インポータが Visual Studio .NET を使用して、ソリューションと vsproj ファイルから情報を抽出するからです。［詳細（Verbose）モード］オプションがチェックされていると、インポート中のインポータの振る舞いがログファイルに記録されます。MSVS ソリューションインポートウィザードの第 2 ステップ を参照してください。

MSVS ソリューションインポートウィザードの第 2 ステップ

インポートウィザードの第 2 ステップは、単なる進捗表示です。このステップに入るとすぐにインポータは Visual Studio .NET と通信して、ソリューションと vsproj ファイルから情報を引き出します。

インポート実行中に［キャンセル］を選択してインポートを中止できます。［新規インポート］を選択して、第 1 ステップからインポートをやり直すこともできます。

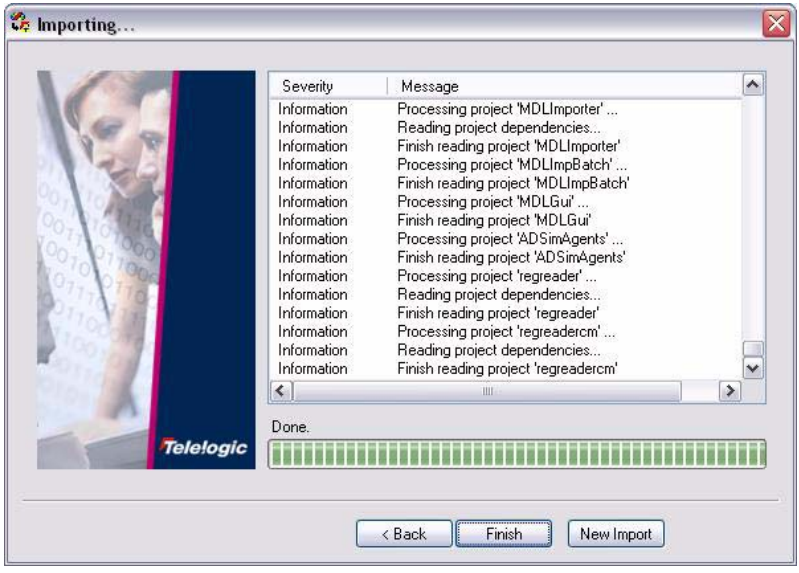


図 190: MSVS sln インポートウィザードの第 2 ステップ

［完了］ボタンを押すとインポートウィザードは終了します。

インポートの結果

MSVS ソリューションインポートは <<MSVSSolution>> ステレオタイプ付きのパッケージを生成します。このステレオタイプはインポートウィザードの第 1 ステップで指定したオプションと同じタグ付き値をもちます。

すべてのアーティファクトはパッケージ内に配置されます。パッケージもその中のエンティティも自動的にはファイルに保存されません。保存はユーザーの責任です。保存するには、パッケージの最上位レベルで［新規ファイルに保存］を選択し、有効な名前を入力して［OK］をクリックします。

ソリューションの再インポート

ソリューションインポートの結果のパッケージ (<<MSVSSolution>> ステレオタイプ付き) を右クリックすると、コンテキストメニューから［**Update model from Visual Studio solution**］が表示されます。このオプションを選択すると、インポートを再実行し、パッケージ内のすべてがいったん削除されて新しいインポートの結果に書き換わります。

18

ファイル / フォルダ インポータ

この章では、ファイルシステムエンティティ（ファイルとフォルダ）を Tau にインポートして、モデルとして表記させる方法を説明します。ファイル / フォルダインポータは、デフォルトのモデル表記を拡張するために、インポートされたファイル / フォルダの処理のために使用できる拡張モジュールをサポートします。

概要

ファイル / フォルダインポータは、ファイルとフォルダを UML 表記としてインポートするためのツールです。基本的な動作として、インポート結果のモデルはファイルアーティファクト（インポートされたファイルを表現）とパッケージ（インポートされたフォルダ）から構成されます。

インポータは、インポートされたファイルとフォルダの、ドメイン固有の拡張モジュールを使用した処理もサポートします。拡張モジュールの機能の例としては、ファイルの内容に関する情報を可視化する目的で、インポートされたファイルの特定の一部を分析して、対応するファイルアーティファクトの間の依存関係を作成する、などがあります。インポータとともに動作するカスタマイズされた拡張モジュールを追加することも可能です。

クィックスタート

インポートウィザードで、ファイル / フォルダインポータを使うには：

- 新規プロジェクトを作成するか、既存のプロジェクトを使用します。
- ［ファイル］>［インポート］を選択して、［インポート ウィザード］を起動します。
- ［**Import Files/Folders**］を選択して、［OK］をクリックします。
- ファイル / フォルダ インポート ウィザードの指示に従います。

ファイル / フォルダ インポート ウィザード

このセクションでは、ファイル / フォルダインポートウィザードについて詳細に説明します。

ファイル / フォルダ インポートウィザードの最初のステップ

インポート処理の第一歩は、ファイルシステムからインポートするファイルとフォルダを指定することです。

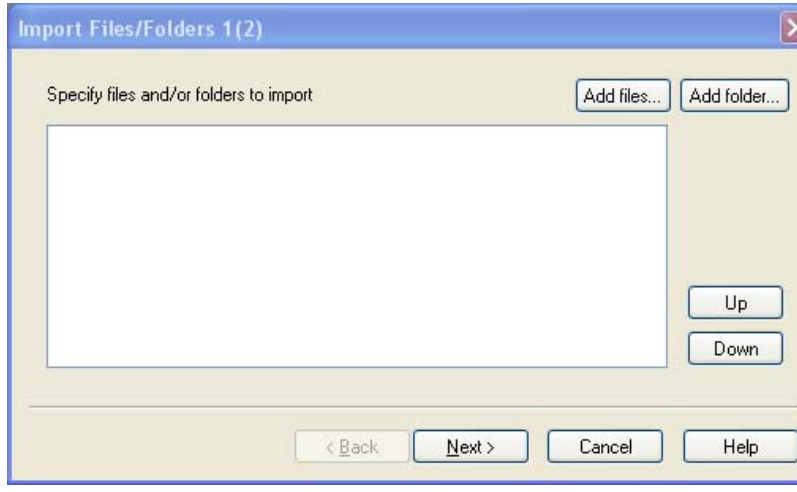


図 191: ファイル / フォルダ インポートウィザードの最初のステップ

- ［ファイルの追加］ボタンを使用して、標準のダイアログを開き、ファイルシステムからファイルを選択します。ダイアログでは複数のファイルを選択できます。
- ［フォルダの追加］ボタンを使用して、標準のダイアログを開き、ファイルシステムからフォルダを選択します。
- 中央の領域をダブルクリックして、直接インポートするファイル、フォルダのパス名を入力することもできます。この場合、パス名にワイルドカードを使うことができます。
- リストから項目を選択して［削除］ボタン押すと、その項目がリストから削除されます。
- ［上へ］または［下へ］ボタンを使って、リスト内の横目の位置を上下に移動できます。リスト内の順番はそのままインポートされる順番になります。

リスト内のパスには URN を使用できます。相対パスも可能です。相対パスはプロジェクトファイルの場所を基準として、解釈されます。

ファイルやフォルダの選択が終わったら、［次へ］をクリックして次のページに進みます。

ファイル / フォルダ インポートウィザードの第二ステップ

二番目のステップでは、インポート処理で使用できる拡張モジュールが提示されます。Tau とともに出荷されている組込済み拡張モジュールについては、組込済み拡張モジュール を参照してください。

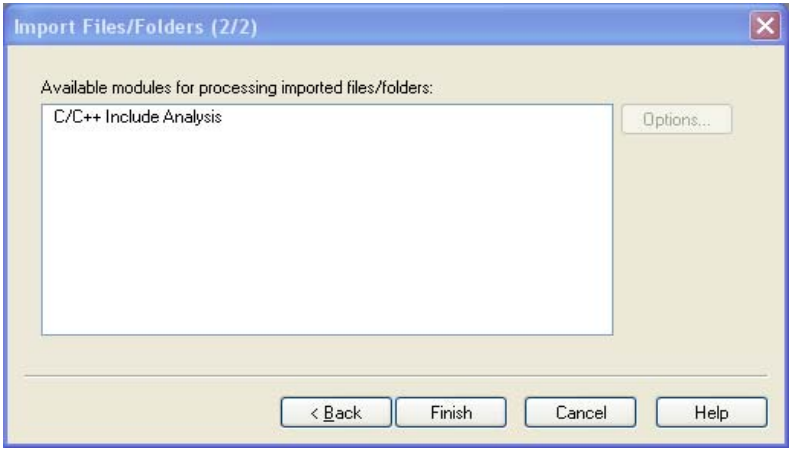


図 192: ファイル / フォルダ インポートウィザードの第二ステップ

インポート処理で使用したい拡張モジュールを選択できます。選択したモジュールにオプションを指定したい場合は、［オプション］ボタンを押します。

［完了］を押してインポートウィザードを閉じて、インポートを実行させます。

インポートの結果

ファイル / フォルダインポータは生成したファイルアーティファクトとパッケージを最上位パッケージ、`Imported Files/Folders<index>` に配置します。ここで、`<index>` はパッケージ名を一意にするための番号です。

インポート後、ダイアグラムジェネレー使用して（ダイアグラムの生成を参照してください）、選択した拡張モジュールによる追加情報を含んだ、インポート要素を可視化できます。

ヒント

テキストファイルを表すファイルアーティファクトをダブルクリックすると、テキストエディタが開いて編集できます。

再インポート

以下の手順でファイル / フォルダを再インポートできます :

- ［モデルビュー］でインポートの結果作成されたパッケージを選択します。
- パッケージを右クリックして、コンテキストメニューから［モデルの更新 (ファイル / フォルダ)］コマンドを選択します。

ファイル / フォルダインポートウィザードで選択したオプションは、インポータが作成したパッケージのタグ付き値として保存されています。必要に応じて、再インポートの前にプロパティエディタを使用してこれらのオプションを編集できます。

組込済み拡張モジュール

このセクションでは、Tau とともに出荷されるファイル / フォルダインポータ用の拡張モジュールを説明します。カスタム拡張モジュールの追加に関する情報は、ファイル／フォルダ インポータの拡張モジュールの追加を参照してください。

C/C++ インクルード分析

この拡張モジュールは、インポートする C/C++ ヘッダファイルの #include ディレクティブの分析を行います。各 #include ディレクティブは、インクルードする側のファイルのファイルアーティファクトとインクルードされる側のファイルのファイルアーティファクトの間の <<inlcude>> 依存にマップされます。

`includePath` オプション（オプション 参照）を使用すると、相対パスを使用したインクルードされるファイルの検索パスを定義できます。

依存図の生成

include 依存を可視化するには、インポート後、以下の手順でダイアグラムジェネレータでダイアグラムを作成します：

- ［モデルビュー］でファイル / フォルダインポータが作成したパッケージを選択します。
- パッケージを右クリックして［**Generate Diagram / Generate Dependency View for Contained Definitions**］を選択します。

作成されるクラス図は、ファイルアーティファクトと include 依存を表示します。

オプション

C/C++ Include 分析拡張モジュールは以下のオプションをサポートします：

`includePath`

インクルードされるファイルを検索するパスのリスト。C/C++ のプロプロセッサがプリプロセスを行うのと同等のパスです。

UML によるアプリケーションの作成

「UML によるアプリケーションの作成」セクションの各章では、UML モデルに基づくアプリケーションの作成方法について説明しています。このセクションで説明している情報は、付属のコードジェネレータを使用する全 UML プロジェクトに共通しています。

19

ビルドとコード生成の概要と例

この章には以下の情報があります。

- UML モデルで C コード ジェネレータ、C++ コード ジェネレータ、および Java コードジェネレータを使用してアプリケーションをビルドするためのユーザー ガイド。
- C コード ジェネレータおよび AgileC コード ジェネレータに焦点を当てた、Tau でのビルド プロセスとコード生成プロセスの概要。
- C Advanced および AgileC コード ジェネレータで提供されるサポートに焦点を当てた UML 合成構造図の使い方。パート／全体関係、およびインスタンスの動的生成機能とともにこれらのコンセプトを使用する方法について説明します。
- 低レベルのポインタ型の機能を UML モデリングで提供する CPtr タイプ。これは、手動でコード化された C コンポーネントまたは C++ コンポーネントとインターフェイスする UML アプリケーションのためのタイプです。
- スレッド OS とリアルタイム OS のインテグレーション。
- Tau で生成された C アプリケーションの例。生成されたコードの環境へのインターフェイス方法、およびターゲットへのデプロイ方法も示しています。

参照

Tau での C++ サポート

Java サポート

注記

生成アプリケーションを Windows Vista で実行するためには、実行ユーザーには管理者権限が必要であり、高い実行権限でアプリケーションを実行する必要があります。

Tau を使用したアプリケーションのビルド

概要

UML モデルからアプリケーションの実行までのビルドプロセスへの入力には、以下のものがあります。

- UML モデル自体。
- ビルドとコード生成の設定。ビルド アーティファクトの使用を参照してください。
  - モデルごとに複数のアーティファクトをビルドできます。
- 構成。ビルド アーティファクトのグループを一度のビルドで処理できます。
- 外部コード（ユーザーが提供）、定義、およびライブラリ。これらは、生成されたコードによってコンパイルおよびリンクを行うビルド プロセスにインクルードしなければなりません。

インタラクティブ モードまたはバッチ モードでのビルド

ビルドは、Tau の GUI を使用するか、コマンド ライン プロンプトのバッチ モードを使用して行うことができます。

参照

インタラクティブ ビルド インターフェイス

バッチ ビルド インターフェイス .

ビルド アーティファクトの使用

ビルド アーティファクトは UML モデルのアーティファクトです。ビルド ステレオタイプ（<<build>> またはその子など）が適用され、ビルド プロセス固有のプロパティを持ちます。ビルド アーティファクトはクラスを配置できる場所ならどこでも配置でき、参照するルート オブジェクトに関する標準のスコープ ルールに従う必要はありません。

ビルド アーティファクトには以下の情報が含まれます。

- ビルド ルート：ビルド ルートは、UML モデルの要素を定義してビルドのスコープを定めます。
- ビルド タイプ：ビルド タイプは、ビルドに使用するコード ジェネレータを定義します。
- ビルド設定：ビルド設定は、コード ジェネレータが使用する設定を定義します。コード ジェネレータに固有または汎用の設定を行うことができます。
- ターゲット ディレクトリ：ビルドによって起動したツールで作成されたファイルを保管する場所を指定します。
- エラーの上限：ビルドに関連するエラーがこの上限を超えると、ビルドが中止されます。

ビルド アーティファクトの追加

以下の手順を実行して、簡単にビルド アーティファクトをモデルに追加できます。

1. 正しいビルド タイプ（Model Verifier など）が有効になっていることを確認します。
2. ビルド ルートとして使用するモデル要素を右クリックし、ビルド タイプを選択し、［新しいアーティファクト］を選択します。
3. プロパティ エディタを使用して、ビルド タイプに関する詳細オプションを指定するために必要なステレオタイプを追加します。［フィルター］ドロップダウン メニューには現在適用されているステレオタイプがあります。

ビルド アーティファクトで定義されたプロパティの表示と指定

それぞれのビルド アーティファクトには、ビルド ステレオタイプのインストレーションが 1 つあります。ビルド ステレオタイプは実際のビルド タイプに関連するビルド設定を定義します。

これらの属性を表示するには以下の手順を行います。

1. ワークスペース ウィンドウのビルド アーティファクトを右クリックします。
2. ［ビルド設定］を選択します。プロパティ エディタが表示されます。
3. 必要に応じて［ステレオタイプ］をクリックし、このアーティファクトでインスタンス化されるべきビルド ステレオタイプを追加します。
4. ［フィルタ］ドロップダウン メニューからビルド ステレオタイプを選択します。
5. プロパティ（ビルド設定）が表示され、変更できます。エディタで変更を適用するとそれがコミットされます。

ビルド アーティファクトの場所

［モデル ビュー］でのビルド アーティファクトの場所にはセマンティックな意味はありません。ビルド アーティファクトは、ビルド ルート、またはワークスペース内の任意の場所など、有効な場所なら必要に応じてどこに移動してもかまいません。

アーティファクトの必須使用

実行中のビルドには少なくとも 1 つのビルド アーティファクトが存在しなければなりません。

- ビルド アーティファクトなしでモデルをビルドする場合、ビルド ウィザードによってビルド アーティファクトを作成できます。このウィザードで、必要な情報を指定するように要求されます。ビルド アーティファクトはアクティブな構成にも挿入され、構成のビルドによってそのアーティファクトもビルドされるようになります。

複数ビルド アーティファクトの構成

UML モデルごとに複数のビルド アーティファクトを使用できるので、1 つの UML モデルから複数のアプリケーションを生成できます。たとえば、ホスト コンピュータ上でのデバッグを目的としたモデル ベリファイヤ（Model Verifier）を作成するビルド アーティファクト、ターゲットへのデプロイメントを目的としてアプリケーションをビルドするビルド アーティファクトなどがあると便利です。

さらに、1 回の操作で複数のビルド アーティファクトを指定できるように、複数のビルド アーティファクトを構成にグループ化できます。

選択によってビルドが開始されます。以下のルールに従って、選択したオブジェクトによってリストが形成されます。

- オブジェクトがビルド アーティファクトの場合、リストに追加される。
- オブジェクトがビルド アーティファクトではない場合、オブジェクト（または親オブジェクト）を「マニフェスト」する一連のビルド アーティファクトが形成される。
- セットが空の場合、「ビルド ウィザード」が表示される。
- セットにビルド アーティファクトが 1 つだけ含まれる場合、リストに追加される。
- セットに複数のビルド アーティファクトが含まれる場合、「ビルド ウィザード」が表示される。

「Build」および「Verify」操作では、ビルド アーティファクトのリストには、リスト内のビルド アーティファクトが依存するビルド アーティファクトも追加されます。つまり、たとえばビルド アーティファクト「A」がビルド アーティファクト「B」に依存している場合、「A」をビルドすると「B」も自動的にビルドされます。

ビルド アーティファクトのリスト順序は、ビルド操作の開始前の依存関係に従います。循環依存関係がある場合、ビルド結果の順序は定義されません。

参照

構成を使用したビルド

スレッド アーティファクトの使用

ビルド アーティファクトは複数の**スレッド アーティファクト**で構成されることがあります。スレッド アーティファクトは、ステレオタイプ << thread >> が適用されたアーティファクトです。このようなクラスは、クラス図エディタを使用してモデリングします。

スレッド アーティファクトは複数のビルド アーティファクトで使用できます。

スレッド アーティファクトは AgileC コード ジェネレータと C コード ジェネレータでのみ使用します。

C++ アプリケーション ジェネレータ と Java コードジェネレータについては、スレッドは、TOR ライブラリのユーティリティを使ってプログラム的に定義され、操作されます。

使用例

スレッド アーティファクトの使用方法については、この章の最後にあるスレッド アーティファクトの使用例を参照してください。

参照

アプリケーション例

ファイル アーティファクトの使用

モデル要素のファイルへの実装方法を指定するため、また外部コンポーネントを追加してソースとターゲット間に「make」依存関係を指定するために、ファイル アーティファクトを使用します。

注記

ファイル アーティファクトは、モデル ベリファイヤ（Model Verifier）、C Code Generator、および AgileC Code Generator によって生成されるアプリケーションの実装状態または依存関係を指定するために使用します。ファイル アーティファクトを使用しない場合、関連付けられているコード ジェネレータ固有のコード生成設定を使用しなければなりません。

ファイル アーティファクトを使用する C++ コード生成の調整

ファイル アーティファクトを使用する典型的な場面は、ステレオタイプの C++ implementation file と C++ header file を使用して、C++ アプリケーション ジェネレータで生成された C++ コード ファイルとクラスとの間のマッピング スキームを指定する場合です。C++ アプリケーション ジェネレータで使用されるマッピング スキーム（デフォルトは 1 対 1 マッピング、つまり各クラスが 1 つの C++ ソース ファイルと 1 つの C++ ヘッダーにマッピングされる）を置き換えることにより、生成されたコードをファイルに保存する方法を詳細に設定できます。

この精度により、マルチ ユーザー環境での構成管理が簡素化され、C++ コンパイラのタスクも同様に簡素化されます。しかし、所属が同じクラス、または安定していてほとんど変更が見込まれないモデルの一部を含むクラスをパッケージにグループ化し、これらのクラスまたはパッケージに関連付けられる C++ ファイル アーティファクトの作成を減らすと便利です。

ファイル アーティファクトを使用する C++ ラウンド トリップ

C++ ファイル アーティファクトは、クラスを生成されたコードに「接続」します。これにより、ユーザーが行った変更を C++ ファイルがコード ジェネレータによって生成された後で反映させるため、モデルとコードの同期（構成の更新コマンドを使用）を取ることができます。

C++ ファイルをマニフェストする際求められる精度レベルは、構成管理の考慮点および要求される柔軟性によって異なります。715 ページの「ファイル アーティファクトを使用する C++ コード生成の調整」を参照してください。

ファイル アーティファクトを使用する C++ ターゲットと make スキームの指定

ファイル アーティファクトを使用して、モデルを複数のターゲットにコンパイルおよびリンクするよう指定できます。この場合のターゲットは、ライブラリまたは実行形式のアプリケーションのことです。これにより、作成されるアプリケーションをいくつかのライブラリに分割できます。

C++ ターゲットを指定するには、以下の手順を行います。

1. library または executable に特化したファイル アーティファクトを追加します。
   - ライブラリ ファイル アーティファクトは静的または動的にリンクされたライブラリとしてさらに特化できます。
2. ライブラリ／実行形式ファイルからそのソースへの依存関係を追加します。
   - ライブラリ ファイル アーティファクトはライブラリの一部である C++ 実装ファイル アーティファクトに依存しなければなりません。
   - 実行形式ファイル アーティファクトは、適切な UML 要素（C++ コードで `main()` 関数としてマニフェストされる操作など）をマニフェストする C++ 実装ファイル アーティファクトに依存しなければなりません。
   - 実行形式ファイル アーティファクトは、アプリケーションが正しく実行されるために必要なライブラリ アーティファクトにも依存しなければなりません。

その結果、ソース、依存関係、およびライブラリの作成方法を定義する make ファイルが作成されます。生成された make ファイルは、Make settings ステレオタイプの属性を使用して、Windows または UNIX での make をポートするように適応し、ユーザー定義コードを使用する引数を使用できるようになります。

ファイル アーティファクトを使用する C++ オブジェクトの指定

現バージョンの Tau は、C++ 実装ファイルと任意のオブジェクト ファイルとの間の make 依存関係のカスタマイズをサポートしません。make 依存関係は、ファイル ベース名を維持することを前提としています。

ファイル アーティファクトを使用する Java コード生成

C++ の場合と同様に、Java コードジェネレータはファイルアーティファクトを使用して、モデル要素とファイルの間のマッピングを行います。ただし、C++ の場合とは異なり、Java 言語では Java ソースファイルの内容にいくつかの制約を課しています。たとえば、二つ以上の最上位クラスを 1 つの Java ファイルに保持することはできず、その最上位クラスの名前はファイル名と一致する必要があります。

この制約のため、Tau では、Java コード生成時にファイルアーティファクトを手動で管理することを通常はお勧めしません。ファイルアーティファクトは、コード生成やラウンドトリッピングを行った際に、必要に応じて自動で追加されます。

ファイル アーティファクトの使用例

ここでは、ファイル アーティファクトを使用して C++ ファイルでのモデル要素の make 依存関係とマニフェストをカスタマイズする方法について説明します。

以下のようにアプリケーションにデプロイすると指定されたモデルを考えます。

- モデルには main() 操作があり、C++ コードの `main()` 関数を生成します。このコードを個別の `main.cpp` ファイルと `main.h` ファイルに入れることにします。
- モデルにはグローバル定義を含むパッケージ「a」があるとします。パッケージには、`a.lib` に保存されるライブラリとしてコンパイルされるクラス「a」が含まれます。生成されたソース コードを `a.cpp` ファイルと `a.h` ファイルによってマニフェストされるようにします。
- モデルには、外部で定義および実装されるクラス「support」が含まれます。クラスの実装と定義は既存の `support.cpp` ファイルおよび `support.h` ファイルで行うことができます。「support」クラスについてはコードを生成する必要はありませんが、コンパイルとリンクのスキームにはファイルが必要です。また、グローバル定義がある `a.h` に対して依存関係がなければなりません。
  - クラス「support」を外部で定義および実装されるように指定するには、その属性 **External** を **true** に設定しなければなりません。
- 上記のファイルは、アプリケーション `my_application.exe` にコンパイルおよびリンクされます。

ファイル アーティファクトの作成

上記で確認したファイル（`main.cpp` など）ごとに、まず必要なファイル アーティファクトを作成します。ファイルごとに、以下の手順を行います。

1. ファイル アーティファクトを作成します（[新規] -> [アーティファクト]）。
2. 作業を行うファイルの名前に従って名前を付けてください。
3. プロパティ エディタを使用して、ファイル アーティファクトを C++ implementation file、C++ header file、executable、library のいずれかに特化します。
   - 必要に応じて［ステレオタイプ］をクリックし、必要なステレオタイプを追加します。

すべてのファイルを指定した結果を 718 ページの図 193 に示します。

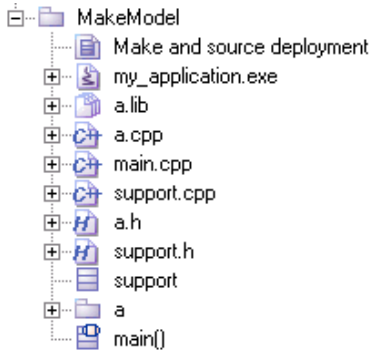


図 193: 必要なファイル アーティファクトを表示する［モデル ビュー］

make 依存関係の指定

コンパイルとリンクのスキームを正しく確実に行うため、以下の手順により make 依存関係を指定します。

1. 「Make and source deployment」などのようなクラス図を作成します。
2. 前のステップで作成したファイル アーティファクトをドラッグ アンド ドロップによりクラス図に入れます。
3. 以下のように、ソース、ライブラリ、実行形式ファイル間の make 依存関係を表す依存ラインを引きます。
   - `my_application.exe` からライブラリ `a.lib` へ
   - `my_application.exe` から外部コード `support.cpp` へ
   - `a.lib` から実装 `a.cpp` へ
4. 以下のように、定義と実装間の make 依存関係を表す依存ラインを引きます。これらの依存関係には、<< include >> ステレオタイプが適用されていなければなりません。
   - `a.cpp` から `a.h` へ
   - `support.cpp` から `support.h` へ
   - `main.cpp` から `a.h` へ（`a.h` にグローバル定義があるため）
   - `support.h` から `a.h` へ（同上）

make 依存関係を指定したクラス図を 719 ページの図 194 に示します。

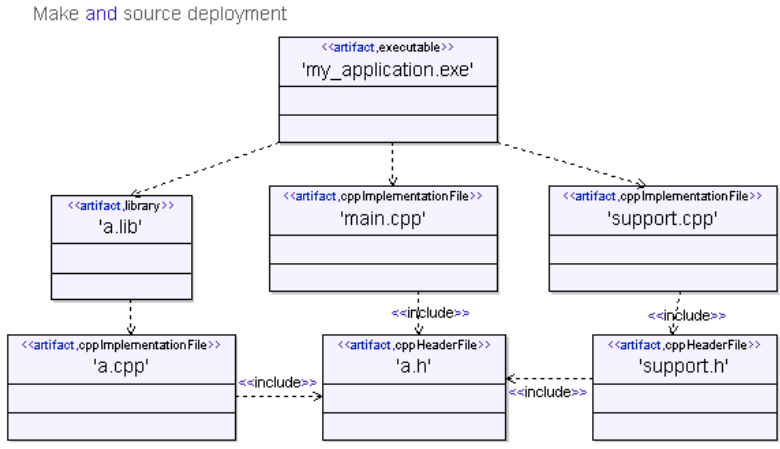


図 194: アプリケーションをビルドするファイルの make 依存関係

マニフェストの指定

前回のタスクでモデル要素「main()」、「a」、「support」が実装と定義をマニフェストするファイルにどのように接続されるかを指定しました。すべての依存関係とマニフェストを 1 つのダイアグラムで示すほうが簡単であれば、make 依存関係を指定するために使用したクラス図をこの作業にも使用できます。

- クラス「a」の定義は（<<manifest>> 依存関係を描くことにより）`a.h` によってマニフェストされます。実装は（<<manifest implementation>> 依存関係を描くことにより）`a.cpp` によってマニフェストされます。
- 同様に、クラス「support」の実装と定義は `support.h` および `support.cpp` によってマニフェストされます。
- 「main()」操作は実装 `main.cpp` によってのみマニフェストされます（`main.h` はありません）。

結果を 720 ページの図 195 に示します。

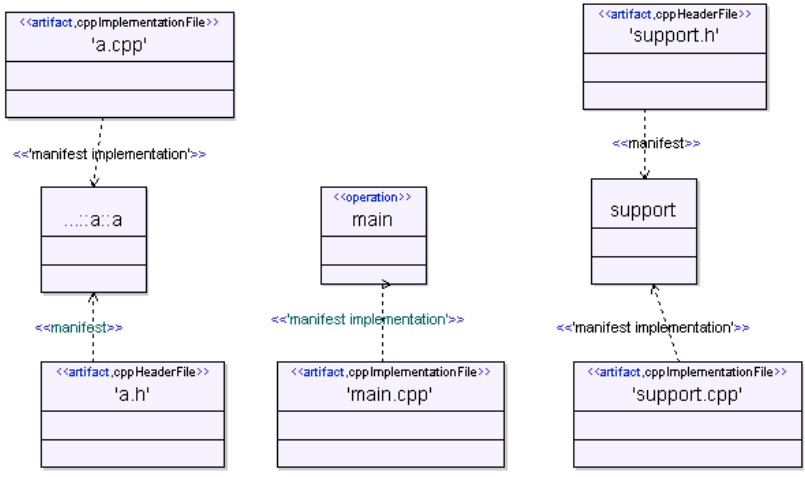


図 195: C++ ファイルによってマニフェストされたモデル要素

ビルド ルートの使用

ビルド アーティファクトごとにビルド ルートを設定する必要があります。ビルド ルートはビルドされる「サブモデル」（ビルド ルートによって定義されるか、UML 言語のスコープ ルールに従って、参照またはインクルードされるモデル要素）を定めます。

- アーティファクトのビルド ルートは、ビルド アーティファクトからビルド ルートとして使用する要素への << manifest >> 依存関係によって識別されます。

ビルド ルートの変更

ビルド ウィザードを使用してビルド アーティファクトを使用する際、ビルド ルートを指定します。

ビルド ルートの変更が必要な場合、以下の方法で最も簡単に行うことができます。

1. ビルド アーティファクトを右クリックし、ショートカット メニューから［ビルド ルートの選択］を選択しします。
2. 表示されたダイアログで、ビルドのルートとして使用するモデル要素（パッケージまたはクラス）を指定します。
3. ［OK］ボタンをクリックしてダイアログを閉じます。

C ビルド タイプに適したビルド ルート

モデルベリファイヤ（Model Verifier）、C コード ジェネレータ、および AgileC コード ジェネレータを起動するビルド アーティファクトのビルド ルートとして使用できる要素は、次のとおりです。

- 最上位のアクティブ クラス
- パッケージ

注記

パッケージをビルド ルートとして使用した場合、アプリケーションではなくライブラリがビルドされます。Target 属性を Executable ではなく Library に設定してください。ライブラリのビルドについては、第 22 章「大規模アプリケーション開発のガイドライン」を参照してください。

C++ ビルド タイプに適したビルド ルート

C++ アプリケーション ジェネレータを起動するビルド アーティファクトのビルド ルートとして使用できる要素は、次のとおりです。

- 最上位のアクティブ クラス
- パッケージ
- クラス

Java ビルド タイプに適したビルド ルート

Java コードジェネレータを起動するビルド アーティファクトのビルド ルートとして使用できる要素は、次のとおりです。:

- 最上位のアクティブ クラス
- パッケージ
- クラス

ビルド タイプの使用

ビルド プロセスとそれに関わるコンポーネントに関するすべての設定結果を管理するため、**ビルド タイプ**という概念を導入しました。ビルド タイプは、基本的にはビルド アーティファクトが使用するコード ジェネレータを定義します。

Tau では、**モデル ベリファイヤ（Model Verifier）**、**AgileC コード ジェネレータ**、**C コード ジェネレータ**、および **C++ アプリケーション ジェネレータ**、**Java コードジェネレータ**を使用するビルド タイプをサポートします。これらの「実際にコードを生成するコード ジェネレータ」以外にも、Tau では make ファイルを生成して make ユーティリティを起動するビルド タイプ **Makefile ジェネレータ**と **Make** もサポートします。

注記

技術的な見地から言うと、モデル ベリファイヤ（Model Verifier）は C コード ジェネレータの特定のアプリケーションです。このアプリケーションには生成された C コードが備わっており、UML レベルでのデバッグ、トレース、シミュレートをサポートするために必要なもの（シンボル情報、コマンドライン インタープリタ、GUI など）を提供します。モデル ベリファイヤ（Model Verifier）によるビルドは、C アプリケーションのビルドと同じようなフローに従いますが、作成されるアプリケーションは異なります。

ビルド タイプごとに、ユーザーができる設定の概要を以下に示します。

ビルド タイプの有効化（ロード）

ビルド タイプを使用できるようにするには、まず対応するアドインをアクティブにする必要があります。これは、プロジェクトの作成時にプロジェクト ウィザードを使用して行います。

プロジェクトの作成時に設定されているビルド タイプ以外を有効にしたい場合、対応するアドインをロードします。このためには、［ツール］メニューから［カスタマイズ］ダイアログを選択します。［アドイン］タブに、現在使用可能なアドイン モジュールが表示されます。これらのモジュールは個々に使用できます。

1. 使用するビルド タイプにチェックを付けて選択します。
   - **Make**：Tau からの make ユーティリティの使用を可能にします。
   - **Makefilegen**：Makefile ジェネレータを有効にします。
   - **AgileCApplication**：AgileC コード ジェネレータを有効にします。
   - **CApplication**：C コード ジェネレータを有効にします。
   - **ModelVerifier**：モデル ベリファイヤ（Model Verifier）を有効にします。
   - **CppGen**：C++ アプリケーション ジェネレータを有効にします。
   - **JavaApplication** ：Java コードジェネレータを有効にします。
2. ［閉じる］をクリックすると、ダイアログで選択したアドイン モジュールがロードされ、モデルにそのプロファイルが追加されます。ロードされているプロファイルの情報は、出力ウィンドウの［スクリプト］タブに表示されます。

   プロファイルをロードすると、関連付けられているビルド タイプ／コード ジェネレータとサポートする設定がプロパティ エディタの［フィルター］ドロップダウン メニューで選択可能になります。

ビルド タイプの指定

ビルド タイプは以下のいずれかの方法で指定できます。

- ビルド アーティファクトを持たないモデルをビルドしようとすると、ビルド ウィザードが起動してビルド タイプを指定するよう要求されます。

- ビルド ルートとして使用するモデル要素から表示されるショートカット メニューで、ビルド タイプを選択し、［新しいアーティファクト］を選択します。これにより、目的のビルド タイプを持つビルド アーティファクトが作成されます。

複数のビルド タイプを持つプロジェクトに関する考慮事項

複数のアドインがロードされたプロジェクトでは（ビルド タイプの有効化（ロード）を参照）、複数のビルド タイプにアクセスできます。別のアーティファクトを構成に追加することにより、複数のビルド アーティファクトをビルドする 1 つのビルド コマンドを使用して、モデルからさまざまなアプリケーションをビルドできます。

たとえば、モデル ベリファイヤ（Model Verifier）の生成と、AgileC コード ジェネレータアプリケーションを生成する 1 つのビルド アーティファクトの生成を行うことができるビルド アーティファクトがあれば、両方のアプリケーションを 1 つのビルド コマンドだけでビルドできます。

注記

1 つのビルド アーティファクトに複数のビルド タイプを与えるべきではありません。multiple build type ステレオタイプを 1 つのビルド アーティファクトに与えると、実際に使用するビルド タイプが定義されていないことになります。この場合、未使用ビルド タイプの削除を行ってください。

未使用ビルド タイプの削除

ビルド アーティファクトからビルド タイプを削除するには、以下の手順を行います。

- アーティファクトを選択し、プロパティ エディタを起動します。
- ［ステレオタイプ］ボタンをクリックしてダイアログを開きます。ビルド アーティファクトに現在適用されているステレオタイプが表示されます。
- 削除するビルド タイプのチェック ボックスの選択を解除し、ダイアログを閉じます。

ビルド タイプの変更

属性としてデフォルト値を持つステレオタイプが適用されており、削除された属性の値は失われて復活できなくなるので、ビルド アーティファクトのビルド タイプを変更することは推奨できません。代わりに、目的のビルド タイプを持つ新規ビルド アーティファクトを作成することを推奨します。しかしながら、プロパティ エディタを使用して［ステレオタイプ］をクリックし、ステレオタイプをチェックすれば、変更は可能です。

個別ビルドの実行

最上位のアクティブ クラス（所有者のないクラス）以外のビルド ルートを定義すると、ビルド スキームを別々のビルドに分割できます。たとえば、コードを生成して 1 つのパッケージのみをコンパイルしたい場合があります。モデル全体のコードを生成すると、前回のビルド後にグローバルな変更を行っていないのに、すべてのパッケージのビルドが行われてしまいます。

個別ビルドを成功させるには、「個別ビルド」機能（ビルド アーティファクトで定義）がコード ジェネレータでサポートされていなければなりません。

- ビルドを個別ビルドに分割するには、必要な数のビルド アーティファクトを作成し、それぞれを適切なビルド ルートに関連付けます。
- C++ アプリケーション ジェネレータを使用してビルドされたモデルは、Makefile ジェネレータで管理される C++ ターゲット（実行形式またはライブラリ）を定義することで、モジュラー化できます。

参照

ファイル アーティファクトを使用する C++ ターゲットと make スキームの指定

選択したモデル要素のビルド

ビルド設定の使用

ビルド設定はすべてのビルド タイプ用の汎用設定と、特定のビルド タイプのための固有設定があります。

ビルド ステレオタイプ

ビルド ステレオタイプにはさまざまなタイプがあり、それぞれが異なるコード ジェネレータに対応しています。ビルド ステレオタイプによって定義される値は、対応するコード ジェネレータによってサポートされるビルド設定を示します。

すべてのビルド ステレオタイプは <<build>> という名前のステレオタイプを継承します。

- ［UML 基本編集］のオプション［ステレオタイプ インスタンスを表示する］にチェックを付け、ビルド アーティファクトに適用されているステレオタイプをワークスペース ウィンドウに表示されるようにすると、作業がしやすくなります。

ビルド設定の表示と変更

ステレオタイプを適用したら、必要に応じてプロパティ（対応するビルド設定）を表示して変更できます。

- 設定を表示して変更するには、プロパティ エディタを使用します。

ビルド設定の削除

ビルド アーティファクトからステレオタイプを削除すると、すべてのプロパティが削除され、対応するビルド設定がデフォルト値に戻されます。

参照

ユーザーができるビルド設定については、ステレオタイプと属性を参照してください。

C ターゲットの指定

C ターゲット名

ビルド タイプ AgileC コード ジェネレータ、C コード ジェネレータ、および Model Verifier については、ターゲットのベース名、つまり実行形式ファイルの名前は、ビルド アーティファクトが参照するビルド ルートの名前から自動的に導き出されます。

C ターゲットの指定と **make**

アプリケーションのコンパイルとリンクに使用される make ファイルは、現バージョンの Tau ユーザー インターフェイスには完全に組み込まれていません。make ファイルのコンテンツは「Make テンプレート ファイル」を使用して取り込むことができます。

使用する Make template file を指定するために C コード生成が設定されたステレオタイプを用いることができますが、make テンプレート ファイルのカスタマイズはツール以外で行う必要があります。make テンプレート ファイルのカスタマイズの詳細については、第 25 章「C および AgileC ランタイム ライブラリ」の 912 ページ、「ライブラリ ファイル」を参照してください。

C++ ターゲットの指定

C++ ターゲット名

C++ ターゲットの名前を指定しない場合、作成されたアプリケーションの名前は次のようになります。

- Windows : `application.exe`
- UNIX : `application`

C++ ターゲットの指定と **make**

デフォルト スキーム（すべてのクラスが C++ ファイルと 1 対 1 でマッピングされ、1 つにコンパイルおよびリンクされる）は、任意の数の C++ ターゲットを調整することによって置き換えることができます。これは、ファイル アーティファクトを使用して行うことができます。

参照

ファイル アーティファクトを使用する C++ ターゲットと make スキームの指定

ターゲット ディレクトリ

ターゲット ディレクトリは、ビルド アーティファクトのために生成されたすべてのファイルが保管される、ファイル システム上の場所のことです。

ビルド アーティファクトで指定しなければ、ターゲット ディレクトリ名はビルド アーティファクトの名前と同じになります。

ビルド アーティファクトごとに、上記の名前付け規則に優先する属性 **Target Directory** を使用できます。

ヒント

暗黙のターゲット ディレクトリの名前は、今後のリリースで変更することがあります。明示的なターゲット ディレクトリを使用することを推奨します。可能であれば、コードを記述してルールを規定し、生成後のディレクトリの名前に依存しないようにします。

ターゲットディレクトリと **make** テンプレート ファイル

ターゲット ディレクトリは、絶対パスかプロジェクト ディレクトリへの相対パスのいずれで指定してもかまいません。Make template file の相対パスを指定する場合、このファイルの場所はターゲット ディレクトリとの相対で指定されます。

make テンプレート ファイルには、通常ソース コード ファイルへの参照が含まれます。make テンプレート ファイルのコンテンツは生成された make ファイルにコピーされます。ベアアプリケーションの場合、ソース コード ファイルへの参照はターゲット ディレクトリの相対パスにします。

スレッド システムをビルドする場合、ターゲット ディレクトリへのサブディレクトリを作成します。このサブディレクトリの名前はビルド ルートから導き出されます。make ファイルも含む生成されたコードは、サブディレクトリに置かれ、ここでコンパイルおよびリンクされます。

エラーの上限

ビルド アーティファクトに関連するエラー メッセージ数がこの数値を超えると、ビルドが中止されます。正確には、ビルドが中止されたタイミングでは、すべての保留エラー メッセージが処理されていることから、エラーの総数がこの数値を超えている可能性はあります。エラーの上限を「0」に設定すると、生成されたエラー メッセージの数に関わらず、ビルドは中止されません。

ビルド アーティファクトを使用したビルド

アーティファクトのビルドを開始するには、ビルド アーティファクトを右クリックし、ショートカット メニューから［ビルド (< build type >)］を選択し、サブメニューから必要な項目を選択します。

- 構成のチェック：セマンティック チェックを行います。
- 構成の生成：[構成のチェック] と同じ操作の後、コードの生成を行います。
- 構成のビルド：[構成の生成] と同じ操作の後、コンパイルとリンクを行います。
- 構成の実行：[構成のビルド] と同じ操作の後、アプリケーションを起動します。このコマンドはモデル ベリファイヤ（Model Verifier）でのみサポートされます。
- 構成の更新：ラウンドトリップ エンジニアリングに使用される外部 C++ コードとの同期を取ります。
- 構成のクリーン：`"make clean"` を実行します。

ツールは汎用設定およびビルド アーティファクトで定義されたビルド タイプに特化された設定（コード ジェネレータ固有の設定など）を読み込み、ビルド アーティファクトで定義されたビルド ルートをコード ジェネレータに提出します。

選択したモデル要素のビルド

選択した要素のビルドは、ビルドするパッケージまたはクラスを右クリックして行うことができます。コンテキスト メニューには、アクティブなビルド タイプごとに 1 つの項目があります。

- これらのビルド タイプごとに、選択した要素をマニフェストするために使用できるビルド アーティファクトのリストがあります。
  - ビルド アーティファクトごとに、使用できるビルド コマンドがサブメニューにリストされます。726 ページの「ビルド アーティファクトを使用したビルド」を参照してください。
- [新しいアーティファクト] というメニュー項目を選択することもできます。このメニュー項目を選択して、選択したエレメントをビルド ルートとする新規ビルド アーティファクトを作成できます。

構成を使用したビルド

構成のビルド

構成のビルドを行うには、以下の手順を行います。

1. ビルドする構成がプロジェクト ツール バーでアクティブな構成になっていることを確認します。
2. ビルド メニューから適切なメニュー項目を選択します（[構成のチェック]、[構成の生成]、[ビルド]）。

ビルド アーティファクトの構成への追加／削除

ビルド アーティファクトの構成への追加／削除は、プロジェクトの設定ページを使用して行います。

1. プロジェクト ツール バーから、ビルド アーティファクトを追加／削除したい構成を選択します。
2. ［プロジェクト］メニューから［設定］を選択します。ダイアログが表示されます。
   - ダイアログの左側のツリーには、プロジェクトに含まれるファイルの一覧が表示されます。この情報は UML モデルをビルドする際は使用しないので、無視してください。
   - ［ビルド］タブに表示されるリストには、アクティブな構成に現在含まれているすべてのビルド アーティファクトが表示されます。
3. アーティファクトを選択し、左右の矢印を使用して構成に追加／削除します。リストから複数の項目を選択し、一度の操作で複数の項目を追加／削除できます。
4. 構成に入れたいアーティファクトがすべて左側のリストに表示されたら、［OK］ボタンをクリックして確定します。

注記

［キャンセル］をクリックしても［OK］と同じ結果になります。従って、いったん行った追加／削除をキャンセルしたい場合は、ダイアログを閉じた後で、［元に戻す］機能を使用して構成の内容を元に戻してください。

ビルドに関するエラーと警告

UML モデルからアプリケーションの実行までのビルド プロセスには、以下のようないくつかのフェーズがあります。これらの各フェーズでは、ツールによってビルド タブの出力ウィンドウにメッセージが表示されます。エラーの上限を使用して、エラーが一定数に達したらビルドを中止するようにできます。

- 初期分析フェーズでは、モデルをチェックして選択したビルド タイプのコンテキストのセマンティックな正確性を確認します。このフェーズでは、警告またはエラーが返されます。多くは［選択部分をチェック］または［すべてをチェック］コマンドを実行する際のエラーおよび警告です。
- 分析フェーズが終了してモデルと設定がビルドに適していると判断されると、コード ジェネレータが起動します。セマンティック チェッカーが検出しないコードを生成する際、高度な UML 構成要素については制限がある場合があるので、このフェーズでさらにエラーが表示されることがあります。
- ビルド タイプで定義された make ファイル ジェネレータは make ファイルを作成し、make が実行されます。デフォルト設定が使用される場合、つまりユーザー定義 make テンプレート ファイルまたは make 設定が定義されていない場合、このフェーズではエラーは表示されません。make ユーティリティで報告される診断も、出力ウィンドウに表示されます。

- make が実行された結果、モデルから生成されたコードまたはユーザーが提供したコードがコンパイルされます。事実上すべてのケースで、ツールによって生成されたコードはコンパイルに成功するはずです。コンパイラのメッセージは出力ウィンドウに逐語的にエコーされます。
- 最後に、オブジェクト ファイルがランタイム ライブラリとユーザーが提供したライブラリにリンクされます。UML モデルで「External」と宣言されたエンティティが、正しいタイプと名前のライブラリでも使用可能であれば、このフェーズではエラーは報告されないはずです。リンカーのメッセージは出力ウィンドウに表示されます。

表示されるメッセージの重要度は、情報／警告／エラーの順です。

注記

コード生成前に行うセマンティック チェックでは、[選択部分をチェック] または [すべてをチェック] を実行した場合と同じ結果が報告されないことがあります。これは、セマンティック チェックを行う方法が違うことに起因します。
[選択部分をチェック] および [すべてをチェック] 操作は、Tau のメモリ内のモデルに対して行われます。[選択部分をチェック] は選択されたモデル エンティティをチェックし、[すべてをチェック] は全体のモデルをチェックします。[すべてをチェック] を実行すると、バインドの不一致や問題も検出するため、まず全体のモデルのバインドが解除されます。
コード ジェネレータが実行するセマンティック チェックも基本的には同じですが、コード ジェネレータがロードしたモデルのコピーに対してチェックが行われます。これは、通常一連のオリジナルのモデルです。この場合、コード ジェネレータ固有のチェックも実行されます。
混乱を避けるため、コード生成を行う前に [すべてをチェック] を実行して、モデルが正しいことを確認するとよいでしょう。[すべてをチェック] によってエラーが報告された場合は、コード生成を行う前にエラーを修正してください。

参照

エラー メッセージと 警告メッセージ

C アプリケーション ビルド時の UML サポートの制限事項

Makefile ジェネレータ

Makefile ジェネレータは、本質的には、他のコード ジェネレータと同様な動作をするコード ジェネレータですが、プロジェクト全体ではなく、1 つのモデル（.u2）ファイルを読み込むことができます（プロジェクトを読み込むこともできます）。

利用法

Makefile ジェネレータは、多くの「ジェネレータ パラメータ」を使用して構成できます。これらのパラメータにより、内部での変換の方法と名前の変換の方法、ルールとコマンドを Makefile に書き込む方法を決定できます。

Makefile ジェネレータは次の 2 つの方法で使用できます。

explicit（明示的）

ユーザーはプロジェクト内に make モデルを作成し、"Makefile Generator" ステレオタイプが与えられたビルド アーティファクトを作成します。ビルド アーティファクトは、"manifest" ステレオタイプが適用された make モデルと依存関係があります。これがすべてのコード ジェネレータが動作する方法です。

implicit（暗黙的）

これが「通常」の操作方法です。ユーザーが意識的に Makefile ジェネレータを起動するのではなく、他のコード ジェネレータ（たとえば、C++ コード ジェネレータ）からのコード生成の結果として、いわば副産物的に生成される Makefile モデル ファイルから、Makefile ジェネレータが起動されます。Application Builder は、コード生成動作の終了後に”make-model”型のビルド結果ファイルを受け取ると、make model u2 ファイルを入力として使用して Makefile ジェネレータを必ず起動します。また、"main" コード ジェネレータを起動するために使われたビルド アーティファクトに "Makefile Generator Settings" ステレオタイプが適用されている場合、このステレオタイプの設定が Makefile コード ジェネレータに適用されます。すなわち、"Makefile Generator Settings" ステレオタイプによって Makefile コード ジェネレータのデフォルトの振る舞いが変更されます（このステレオタイプは、Makefile コード ジェネレータを実行するための必要条件ではありません）。

コード ジェネレータのステレオタイプ

Makefile コード ジェネレータは、プロファイル パッケージ "MakefileGen" にコード ジェネレータとして定義されます。これによって、2 つのステレオタイプ "Makefile Generator" と "Makefile Generator Settings" が定義されます（下図を参照）。ステレオタイプ "Makefile Generator Settings" には Makefile コード ジェネレータ特有のすべての設定が含まれ、ステレオタイプ "Makefile Generator" には "build" と "Makefile Generator Settings" から継承するものだけが含まれていますが、それ自体は属性を定義しません。

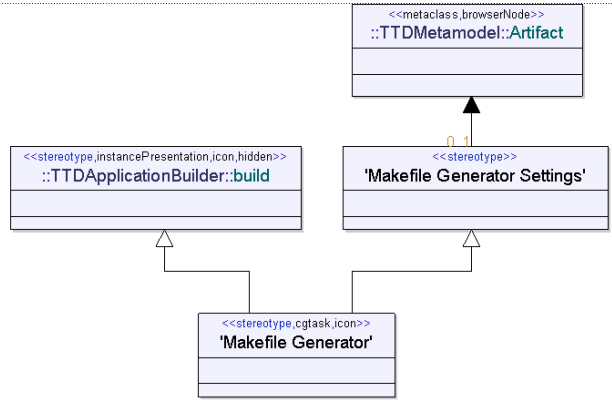


図 196: “Makefile Generator” と ”Makefile Generator Settings”

Makefile コード ジェネレータが明示的に使われると、ビルドのステレオタイプ "Makefile Generator" が使用されます。Makefile コード ジェネレータが暗黙的に使用されると、ステレオタイプ "Makefile Generator Settings" がビルド アーティファクト に追加されてデフォルトの振る舞いを変更できます。Makefile が生成されるときに使用されるデフォルトのジェネレータのパラメータは、ホスト システム（Win32、Solaris または Linux）によって決まります。

"Makefile Generator Settings" で定義される設定を以下に説明します。

Dialect

これは生成される Makefile の固有表現、GNU C/C++（およびクラシックな）互換 Makefile については "gmake"、または Win32 nmake 互換 Makefile のについては "nmake" を決定します。デフォルトはターゲットの種類によって決まります。

Target

これは、実行形式ファイルまたはライブラリ（静的または動的）のどちらをビルドするかを決定します。デフォルトは、実行形式ファイルをビルドするために最適な Makefile を生成することです。

Target Kind

これは Makefile 生成の全体の構成を制御します。デフォルトは、Win32 システムの場合は "Win32 - cl"、Linux システムの場合は "Linux - g++"、Solaris の場合は "Solaris - CC" というように、ホストシステムによって異なります。

User code

このフィールドは、ターゲットの種類によって暗黙に定義される設定を上書きするために使用されます。追加の make 変数をここで定義できます。

例 **314: C++** コード生成のためのビルド アーティファクト **----------**

ビルド アーティファクト "`HelloWorldArt`" はアクティブ クラス "Hello" をマニフェストします。アーティファクトは C++ コードの生成に使用されます。またこのアーティファクトは、Makefile ファイル生成時に使用されるデフォルトの上書きに使用される "Makefile Generator Settings" ステレオタイプも適用されています。このステレオタイプの設定は次のようになります。

図 197: ビルド アーティファクト "`HelloWorldArt`" がアクティブ クラス "Hell" をマニフェストする

これにより、Solaris に適した Makefile と GNU C/C++ ツール チェーンを作成します。

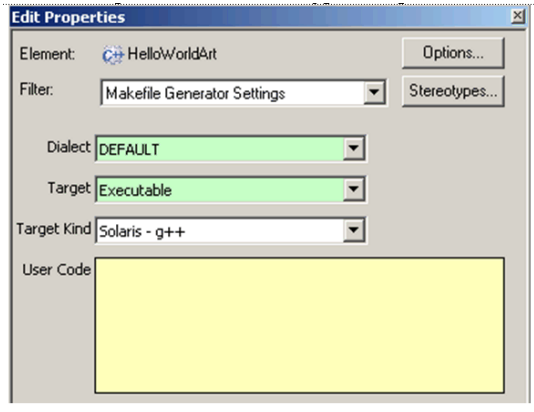


図 198: Makefile Generator ステレオタイプの設定

ファイルのステレオタイプ

Makefile ジェネレータは、ファイル アーティファクトのステレオタイプが適用されたアーティファクトを認識します。

名前	定義場所	コメント
PHONY	MakefileGen	"all" および "clean" などの "phony" ターゲットに対して使用されます。これは "file" ステレオタイプを継承するステレオタイプではなく、ただアーティファクトを拡張するだけです。
objectFile	MakefileGen	オブジェクト ファイルを表すために使用されます。
executable	TTDFileModel	実行形式ファイルを表すために使用されます。
library	TTDFileModel	一般的にライブラリを表すために使用されます。このステレオタイプを持つアーティファクトは、"staticLibrary" ステレオタイプが適用されているかのように処理されます。
staticLibrary	TTDFileModel	静的ライブラリ（非 Win32 システムのアーカイブと Win32 システムの lib ファイル）を表すために使用されます。
dynamicLibrary	TTDFileModel	動的ライブラリ（ELF システム上の共有オブジェクト ファイル、Win32 システムの DLL）を表すために使用されます。
cppImplementationFile	TTDFileModel	C++ 実装ファイルを表すために使用されます。
cppHeaderFile	TTDFileModel	C++ ヘッダーファイルを表すために使用されます。

Make モデル

Makefile コード ジェネレータへの入力は、ファイルのステレオタイプが適用されたファイル アーティファクトを内包するモデル（またはモデルの一部）です。ファイル アーティファクト間の依存関係は、make 依存関係として解釈されます。依存関係に関するステレオタイプはすべて無視されます。

多くのモデル変換は、最後の Makefile が記述される前に実行されます。このセクションでは、"cpp file" の概念は、"cppImplementationFile" ステレオタイプが適用されたアーティファクト、"objectFile" ステレオタイプが適用された "objfile" などに使用されます。lib ファイルという用語は、"library"、"staticLibrary"、または "dynamicLibrary" ステレオタイプが適用されたアーティファクトに使用されます。A が B に「依存する」という概念は、A がクライアントで、B がサプライヤという依存関係に対して使用されます。

include モデルの収集

自動オブジェクト ファイル

自身に依存する obj ファイルを持たない各 cpp ファイルについて、cpp ファイルに依存する新しい obj ファイルが作成されます。

オブジェクト ファイルへのルーティング

実行形式ファイル、ライブラリ ファイル、または PHONY が cpp ファイルに依存する場合、cpp ファイルに依存する obj に依存するように、依存関係が変更されます。変換「自動オブジェクト ファイル」により、このような obj ファイルの存在が確実に作成されます。

デフォルトのターゲット

実行形式ファイルまたはライブラリ ファイルと依存関係のない obj ファイルが存在する場合、新しいデフォルトのターゲット（exe または lib のどちらかが「ターゲット」属性に基づいて決まる）が作成され、依存関係が obj ファイルに設定されます。新しいデフォルトのターゲットが実行形式ファイルの場合、実行形式ファイルと各ライブラリとの依存関係が作成されます。

デフォルトの "all" ターゲット

"all" という名前の PHONY ターゲットが存在しない場合、すべての実行形式ファイルとライブラリ ファイルによって決まるこのターゲットが作成されます。

デフォルトの "clean" ターゲット

"clean" という名前の PHONY ターゲットが存在しない場合、すべての実行形式ファイルとライブラリ ファイルによって決まるこのターゲットが作成されます。

名前の変換

実行形式ファイル名、ライブラリ ファイル名およびオブジェクト ファイル名は、ジェネレータにあるパラメータが設定された場合に変換されます。ファイル名の変換は、ジェネレータのこのパラメータに基づきます。フラグ パラメータは以下のとおりです。

"transformExeName"、"transformObjName"、"transformStaticLibName"、"transformDynamicLibName"

名前の変換は、パラメータ "executableName"、"objectFileName"、"staticLibName" および "dynamicLibName" を使って行われます。名前の変換前にアーティファクト名からパスと既知の接尾辞が削除され、次にパスを再構築した後に新しい名前に変換されます。接尾辞として以下のものがあります。

".exe"、".obj"、".lib"、".dll"、".u2"、".ttp"、".ttw"、".o"、".a"、".so"、".asm"、".m"、".mak"、".cxx"、".hh"、".hpp"、".C"、".rc"、".res"、".cc"、".hh"。

パスの変換

パスの変換は pathDialect パラメータに基づいて行われます。パス セパレータは変更され（'/' 対 '¥'）、UNIX 下では指定されているドライブが削除され、不正な文字がパスに含まれている場合はパスが引用符で囲まれます。この変換は .obj、.exe、.lib、.cpp および hpp ファイルと、include のパスの（内部の）リストに適用されます。

例 **315: Make** モデルとそのターゲットと依存関係

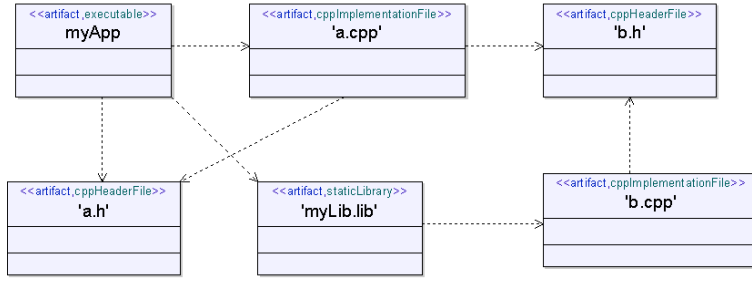


図 199: Make モデルのターゲットと依存関係

```
all : myApp.exe myLib.lib
myApp.exe : a.obj myLib.lib
myLib.lib : b.obj
a.obj : a.cpp
b.obj : b.cpp
clean :
```

中間オブジェクト ファイルに注目してください。

ジェネレータのパラメータ

ジェネレータ パラメータは Makefile の生成方法を制御します。このパラメータはいくつかのセットで構成されます。デフォルトのセットは、各ターゲットの種類に対して 1 つとユーザー セットです。ジェネレータは、パラメータを取り出すとき、ユーザー セット、ターゲットの種類に対応したセット、最後にデフォルトのセットの順に検索します。

ステレオタイプ "Makefile Generator Settings" または "Makefile Generator" の "User Code" 属性に書き込まれているデータが、"user" セットに組み込まれます。

内蔵パラメータ

以下の説明で、「デフォルト」という言葉は、パラメータの特定の値が各ターゲットの種類（ターゲットの種類）に設定された「デフォルト」パラメータ セットに入っていることを示しています。

内蔵パラメータの値のいくつかは、文字列の変更の方法を説明しています。組み込みパラメータのすべてが、「%」とそれに続く 1 文字で構成された 1 組のコードを使用します。これらのコードを以下に示します。

コード	説明
%%	このコードは 1 つの % に置換されます。
%x	このコードは名前に置換されます。
%q	このコードは名前に置換されますが、「異常な」文字が含まれている場合は引用符で囲まれます。
%b	このコードはベース名に置換されます。
%t	このコードはターゲット名に置換されます。
%d	このコードは先頭の依存関係に置換されます。
%D	このコードは、スペースで区切られた、すべての依存関係のリストに置換されます。

最後の 3 つはコマンド パラメータでのみ使用できます。

たとえば、"dynamicLibName" が "lib%x.so" で、動的 lib アーティファクトの名前が "MyLib" の場合は、MyLib は "libMyLib.so" に変わります。

またコマンド パラメータでは、シーケンス 「¥n」（円記号とその後に小文字の「n」）は拡張されて、復帰改行とそれに続く水平タブとなります。これにより、コマンドを複数行にできます。

基本名の置換コード %b は、ライブラリ ファイル名を「基本名」（"-l%b" に見られるように）に置換することが主な目的です。この置換は、"libX.a" から "X"、"libX.so" から "X"、および "X.lib" から "X" となります。

applicationBaseName

デフォルトのターゲット（実行形式ファイル）の基本名を示します。デフォルトは "application" です。

compileCppCommand

cpp ファイルのコンパイルに使用されるコマンドを定義します。このデフォルトを以下に示します。

```
"$(CXX) -c $(CPPFLAGS) $(CXXFLAGS) -o %t %d"
```

コード："%%" "%t" "%d" "%D", expands '¥n'.

cppDefineFormat

コンパイラに対するコマンドライン定義のフォーマットを示します。デフォルトは "-D%x" です。

コード："%%" "%x" "%q" "%b"

cppIncludeFormat

コンパイラへのコマンドライン include パスのフォーマットを示します。デフォルトは "-I%x" です。

コード："%%" "%x" "%q" "%b"

dynamicLibBaseName

デフォルトのターゲット（動的ライブラリ）の基本名を示します。デフォルトは "rary" です（基本名は後で "lib%x.so" を使って変換されます）。

コード："%%" "%x" "%q" "%b"

dynamicLibName

動的ライブラリ名を変換する方法を定義します。デフォルトは "lib%x.so" です。

コード："%%" "%x" "%q" "%b"

executableName

実行形式ファイル名を変換する方法を定義します。デフォルトは "%x" です。

コード："%%" "%x" "%q" "%b"

linkDynamicLibCommand

動的ライブラリの生成に使用されるコマンドを定義します。デフォルトを以下に示します。

```
"$(CXX) $(LDSOFLAGS) -o %t %D $(LDLIBS)".
```

コード："%%" "%t" "%d" "%D", expands '¥n'.

linkExeCommand

実行形式ファイルを生成するコマンドを定義します。デフォルトを以下に示します。

```
"$(CXX) $(LDFLAGS) -o %t %D $(LDLIBS)"
```

コード："%%" "%t" "%d" "%D", expands '¥n'.

linkFormat

実行形式ファイルまたは動的ライブラリがライブラリとの間で依存関係にあるときに使用されます。このライブラリの名前は、このフォーマットで、依存関係リストに追加されます。デフォルトは "-l%b" です。

コード："%%" "%x" "%q" "%b"

linkStaticLibCommand

静的ライブラリの生成に使用されるコマンドを定義します。このデフォルトを以下に示します。

```
"$(AR) $(ARFLAGS) %t %D".
```

コード："%%" "%t" "%d" "%D", expands '¥n'.

makeCommentFormat

Makefile のコメントのフォーマットを定義します。デフォルトは "# %x" です。

コード："%%" "%x" "%q" "%b"

makeDependencyFormat

make 依存関係のフォーマットを定義します。デフォルトは "%x" です。

コード："%%" "%x" "%q" "%b"

makeDereferenceFormat

Makefile で使われる make 変数参照読み出しのフォーマットを定義します。デフォルトは "$(%x)" です。

コード："%%" "%x" "%q" "%b"

makeDialect

Makefile の固有表現を示します。デフォルトはありません。

makeTargetFormat

make ターゲットのフォーマットを定義します。デフォルトは "%x" です。

コード："%%" "%x" "%q" "%b"

makeTargetType

デフォルトのターゲットのタイプを決定します。有効な値は、"Executable"（これはデフォルト）、"Static Library" および "Dynamic Library" です。このパラメータは、ステレオタイプ "Makefile Generator Settings" と "Makefile Generator" の "Target" 属性を使って設定されます。

makefileName

生成される Makefile の名前を示します。デフォルトは "Makefile" です。

objectFileName

オブジェクト ファイル名を変換する方法を定義します。デフォルトは "%x.o" です。

コード："%%" "%x" "%q" "%b"

staticLibBaseName

デフォルトのターゲット（静的ライブラリ）の基本名を示します。デフォルトは "rary" です（基本名は後で "lib%x.a" を使って変換されます）。

staticLibName

静的ライブラリ名を変換する方法を定義します。デフォルトは "lib%x.a" です。

コード："%%" "%x" "%q" "%b"

transformDynamicLibName

動的ライブラリ ファイル名の変換を有効にするための論理フラグです。デフォルトは "true" です。

transformExeName

実行形式ファイル名の変換を有効にするための論理フラグです。デフォルトは "true" です。

transformObjName

オブジェクト ファイル名の変換を有効にするための論理フラグです。デフォルトは "true" です。

transformStaticLibName

静的ライブラリ ファイル名の変換を有効にするための論理フラグです。デフォルトは "true" です。

pathDialect

パスの変換を定義します。取り得る値を以下に示します。

- dos：
  - 「/」はすべて「¥」に変換されます。
  - 前に置かれたドライブ指定はすべて削除されません。
  - パスに「0-9A-Za-z/._-:¥」以外のトークンが含まれている場合、そのトークンは二重引用符で囲まれます。
- UNIX：
  - 「¥」はすべて「/」に変換されます。
  - 前に置かれたドライブ指定はすべて削除されます。
  - パスに「0-9A-Za-z/._-」以外のトークンが含まれている場合、そのトークンは二重引用符で囲まれます。
- Cygwin：
  - 「¥」はすべて「/」に変換されます。
  - 前に置かれたドライブ指定はすべて削除されません。
  - パスに「0-9A-Za-z/._-:¥」以外のトークンが含まれている場合、そのトークンは二重引用符で囲まれます。

他の値の場合、変換は無効になります。

Make パラメータ

内蔵パラメータ以外のすべてのパラメータは、そのまま Makefile に渡されます。Makefile コード ジェネレータが、パラメータの値を解釈したり、値を変換することはありません。

make パラメータ セットは固定ではなく、ターゲットの種類によって決まります（つまり、特定のターゲットの種類については、一部のパラメータのみが有効です）。使用される変数は、対応するプラットフォームとツールで通常使われる変数です。このため変数の意味がさまざまに異なる可能性があります（つまり、「CPP」と書くと「C プリプロセッサ」の意味になるシステムが多い中、Win32 環境においてのみ、「C++ コンパイラ」の意味になります）。

ターゲットの種類

ターゲットの種類はパラメータ セットの定義に使われます。ターゲットの種類は、使用されるターゲットのオペレーティング システムとツール セット（コンパイラ リンカーなど）の概念を結合します。以下のターゲットの種類が使用されます。

ターゲットの種類	説明
`Win32 - cl`	Microsoft 32 bit Windows オペレーティング システムは、Microsoft Visual Development Environment に付属するツールチェーンを使用します。
`Cygwin - g++`	Microsoft 32 bit Windows 上で実行される Cygwin 環境をターゲットとし、Cygwin に付属する gnu ツール チェーンを使用します。
`Linux - g++`	Linux の下で gnu ツール チェーンを使用します。
`Solaris - CC`	Solaris の下で SUNWsPro ツール チェーンを使用します。
`Solaris - g++`	Solaris の下で gnu ツール チェーンを使用します。
`VxWorks SimPC - g++`	Microsoft 32 bit Windows 上で実行される VxWorks ホスト シミュレータをターゲットとし、VxWorks に付属する gnu ツール チェーンを使用します。
`VxWorks SimSol - g++`	Solaris の下で実行される VxWorks ホスト シミュレータをターゲットとし、VxWorks に付属する gnu ツール チェーンを使用します。

次の表では、内蔵パラメータの名前はすべて「$」で始まります。ユーザーが内蔵パラメータの値を上書きする必要がある場合、その値は頭文字が「$」の "User Code" 属性に追加されます。「$」で始まらない名前を持つパラメータはすべて make パラメータとして処理され、Makefile ジェネレータによって完全に無視されます（Makefile に解釈なしでエコーされたものを除く）。

makefile generator パラメータのデフォルト値

名前	値	説明
`RM`	`rm -f`	"clean" ターゲットによって使用されるファイル削除コマンド。
`DEFINES`	`$(TAUDEFINES -D_REENTRANT`	
`INCLUDES`	`$(TAUINCLUDES)`	
`CPPFLAGS`	`$(INCLUDES) $(DEFINES)`	

名前	値	説明
CXXFLAGS	-O2 -fpic	
TAUDEFINES		この値は Makefile コードジェネレータによって設定されます。他の方法で割り当てをしてはなりません。
TAUINCLUDES		この値は Makefile コードジェネレータによって設定されます。他の方法で割り当てをしてはなりません。

makefile generator パラメータの **Win32 - cl** 値

名前	値	説明
$makeDialect	nmake	nmake は Win32 上で使用されます。
$staticLibBaseName	library	
$dynamicLibBaseName	library	
$makefileName	makefile.mak	
$executableName	%x.exe	
$staticLibName	%x.lib	
$dynamicLibName	%x.dll	
$objectFileName	%x.obj	
$cppIncludeFormat	/I %q	
$cppDefineFormat	/D %q	
makeTargetFormat	%q	
makeDependencyFormat	%d	
$linkExeCommand	link $(LINKEXEFLAGS) /out:%t %D	
$linkStaticLibCommand	lib $(LINKLIBFLAGS) /out:%t %D	
$linkDynamicLibCommand	link $(LINKDLLFLAGS) /out:%t %D	
$compileCppCommand	$(CPP) $(CPPFLAGS) /Fo%t /c %d	

名前	値	説明
$pathDialect	dos	
RM	del /f	
CPP	cl	
DEFINES	$(TAUDEFINES) /D ¥"WIN32¥" /D ¥"NDEBUG¥" $(SUBSYSDEF)	
CPPFLAGS	/nologo /MT /W3 /GR /GX /O2 $(INCLUDES) $(DEFINES)	
CXXFLAGS	$(CPPFLAGS)	
CFLAGS	$(CPPFLAGS)	
SUBSYSFLAG	/subsystem:console	
SUBSYSDEF	/D "_CONSOLE"	
LINKLIBS	kernel32.lib user32.lib gdi32.lib winspool.lib comdlg32.lib advapi32.lib shell32.lib ole32.lib oleaut32.lib uuid.lib odbc32.lib odbccp32.lib ws2_32.lib	
LINKEXEFLAGS	$(LINKLIBS) /nologo $(SUBSYSFLAG)	
LINKLIBFLAGS	/nologo $(SUBSYSFLAG)	
LINKDLLFLAGS	$(LINKLIBS) /nologo $(SUBSYSFLAG) /DLL	

makefile generator パラメータの **Cygwin - g++** 値

名前	値	説明
$makeDialect	gmake	
pathDialect	cygwin	
AR	ar	
ARFLAGS	crv	
CXX	g++	
CXXFLAGS	-O2	

名前	値	説明
LDFLAGS		値は空です
LDSOFLAGS	-shared $(LDFLAGS)	
LDLIBS	-lrt -lpthread -lm	

makefile generator パラメータの **Linux - g++** 値

名前	値	説明
$makeDialect	gmake	
AR	ar	
ARFLAGS	crv	
CXX	g++	
LDFLAGS		値は空です
LDSOFLAGS	-shared $(LDFLAGS)	
LDLIBS	-lrt -lpthread -lm	

makefile generator パラメータの **Solaris - CC** 値

名前	値	説明
$makeDialect	gmake	
AR	ar	
ARFLAGS	-xar -o	
CXX	CC	
CXXFLAGS	-O2 -pic -instances=static	
LDFLAGS		値は空です
LDSOFLAGS	-G $(LDFLAGS)	
LDLIBS	-lsocket -lnsl -lrt -lpthread	

makefile generator パラメータの **Solaris - g++** 値

名前	値	説明
$makeDialect	gmake	
AR	ar	
ARFLAGS	crv	
CXX	g++	
LDFLAGS		値は空です
LDSOFLAGS	-shared $(LDFLAGS)	
LDLIBS	-lsocket -lnsl -lrt -lpthread	

Makefile

生成される Makefile の名前は "$makefileName" パラメータによって与えられます。生成される Makefile は以下の部分から構成されます（順番に）。

プリアンブル
Makefile の発生元を記述する一組のコメントです。

固有表現仕様
このコメントを Makefile の実行時に使用して、Makefile が使用する固有表現を特定します。コメントには、文字列「MakeDialect=」とそのすぐ後に続く「nmake」または「gmake」が含まれます。

Make パラメータ
すべての make パラメータは Makefile に記述されます。特に決まった順番はありません。

all ターゲット
ターゲットとの依存関係が、実行形式ファイル、動的ライブラリ ターゲット、静的ライブラリ ターゲット、ライブラリ ターゲットおよび他のターゲットの順に発行されます。このターゲットはコマンドを使用しません。

実行形式ファイル ターゲット
このターゲットは、パラメータ "$linkExeCommand" を使用します。

動的ライブラリ ターゲット
このターゲットは、パラメータ "$linkDynamicLibCommand" を使用します。

ライブラリ ターゲット
このターゲットは、パラメータ "$linkStaticLibCommand" を使用します。

静的ライブラリ ターゲット
このターゲットは、パラメータ "$linkStaticLibCommand" を使用します。

オブジェクト ファイル ターゲット
このターゲットは、パラメータ "`$compileCppCommand`" を使用します。

clean ターゲット

ポストアンブル
これは "`END`" を示すコメントです。

Tau でのコード生成

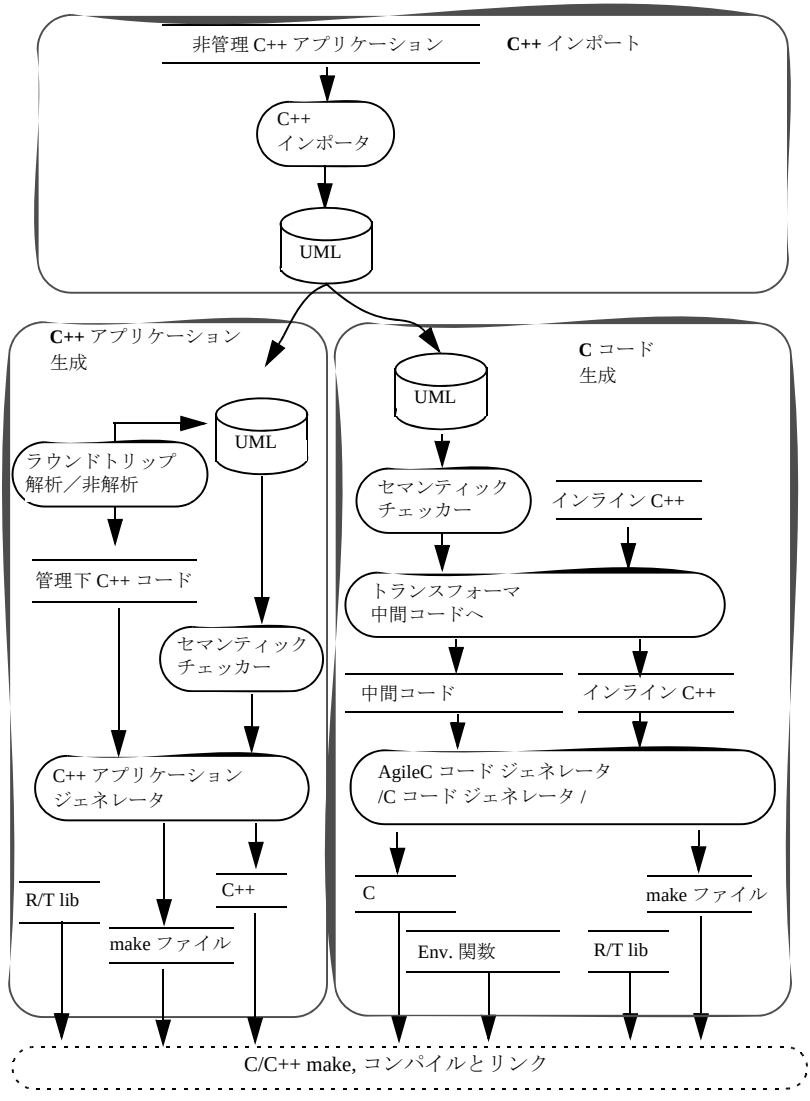


図 200: Tau でのコード生成

C++ インポート

C/C++ インポートの主目的は、Tau で開発した UML アプリケーションから外部 C/C++ アプリケーションへのアクセスを提供することです。この仕組みによって、新規アプリケーション開発に UML を採用するプロジェクトに既存のコードをインクルードできるようになります。C/C++ コードのインポート処理では、一連の C/C++ ヘッダー ファイルを解析し、変換ルールに従って、C/C++ 定義を対応する UML 定義に変換します。

ターゲット

インポート時に作成された UML 要素は、ユーザーが指定した UML パッケージに追加されます。指定するパッケージは、すでに開発中の UML モデルが含まれているパッケージか、インポート結果を保管するためのパッケージのどちらでもかまいません。

インポートの次のステップは、UML 内のアプリケーション モデルから、インポートした UML への参照です。インポートした UML をコピーして、実際に開発中のアプリケーションを定義するモデルを含むパッケージに貼り付けられます。

プリプロセス

C/C++ インポートでは、C/C++ の世界で頻繁に使用される条件付きコンパイルのプリプロセッサとパラダイムを利用できます。これにより、インポート後の設計とコンパイル方法を管理する理論に沿って柔軟にインポートできます。

C および C++ のサポート

インポート時に、ソース言語が ISO C 言語なのか、C++ 言語なのかを指定できます。C と C++ の構文ルールとセマンティック ルールの必須のサブセットがチェックされ、UML に正しく変換できる十分な情報がソースに含まれているかが確認されます。アプリケーションの範囲を拡張するため、C 言語と C++ 言語の一般的な固有表現がサポートされています（第 21 章「コード生成のステレオタイプ」の 844 ページ、「C/C++ dialect」を参照）。

変換ルール

UML への変換は一連の変換ルールに従って行われます。この変換ルールは、C/C++ 言語のほぼ全体をカバーし、外部コードを今開発している UML モデルから簡単に参照できる UML モデルに変換するように設計されています。

ソースへのトレース バック

インポート時に生成された UML 要素には、ソース ヘッダー ファイルおよびファイル内のソース定義の場所に関する情報を持つ属性があります。

参照

第 9 章「C/C++ のインポート」

C コードの生成

UML モデルから、C コード ジェネレータ リファレンスおよび AgileC コード ジェネレータ リファレンスによって完全なリアルタイム アプリケーション コードが作成されます。生成されたコードは、外部 C/C++ コードから簡単に引数を付けたり、選択したランタイム ライブラリにリンクを付けることができるので、アプリケーションに必要なプロパティを与えることができます。プリプロセッサを使用する条件付コンパイルは、高度な柔軟性と拡張性を得るために頻繁に行われます。

UML モデル

UML モデルは、「ネイティブ」ソースとして管理されます。広範囲なセマンティック チェックおよび C 言語を使用するアプリケーション コードの完全な生成など、UML は完全にサポートされます。

生成後に C コードの一部となるモデル内の要素には、以下のようなものがあります。

- **アクティブ クラス**は実行形式のコードになります。アプリケーションの振る舞いは**状態機械図**で示され、これが詳細設計または実装仕様となります。
- 他のクラス（**アクティブではないクラス**）は、データ型およびデータ型の演算子となります。使用またはインスタンス化されると、これらのクラスはデータに加えてコードとなります。
- **合成構造図**は環境へのインターフェイス、および状態機械間の内部インターフェイスを示します。合成構造図から、状態機械間の接続とシグナリングを実装するコードと、環境へのインターフェイスが生成されます。
- **ビルド アーティファクト**は、ビルド プロパティの制御に使用されます。ビルド アーティファクトは、［モデル ビュー］に表示されるか、クラス図に示されることもあります。
- モデルを含むプロジェクトは、それぞれが任意数のビルド アーティファクトを含む**構成**を、任意数インクルードすることもできます。
- **スレッド アーティファクト**は、アプリケーションのスレッドへのデプロイメントを制御するために使用します。スレッド アーティファクトは、一連のインスタンスをアクティブ クラスに基づいて示します。ビルド アーティファクトは、1 つまたは複数のスレッド アーティファクトに明示的にモデリングできます。スレッド アーティファクトは、モデル内のインスタンスのビルド方法を決めるために使用されます。スレッド アーティファクトは 1 つまたは複数のビルド アーティファクトと関係があります。

設定

コード生成の設定は、モデルに組み込まれたビルド アーティファクトで定義されます。ビルド アーティファクトは、抽象モデルから切り離すべき情報を保持しています。

ビルド アーティファクトはコード ジェネレータ設定、ランタイム ライブラリおよび外部コードなどを指定します。さらに、ビルドアーティファクトは、ターゲット アプリケーションのコンパイルとリンクに必要な情報を保持します。

インライン C++

Tau では、インライン C またはインライン C++ のコードをサポートします。これらのコードは、UML モデル内で意味のある場所であれば、事実上どこにでも追加できます。インライン コードはほとんど逐語的に C コード ジェネレータおよび AgileC コード ジェネレータに転送され、そこで最終生成されたアプリケーション コードに埋め込まれます。インライン C コードには次のようなルールがあります。

- インライン C コードは、2 重角かっこ `[[ ]]` に入れる。
- インライン C 内のコメント（`/* */` を使用）はサポートされない。
- UML コード内で文字をエスケープするためシャープ記号（#）を使用する。たとえば、プリプロセッサ演算子はダブル シャープ記号（##）で記述し、UML エンティティは「`#(<UML 名 >)`」を使用してアクセスできます。

例 **316:** インライン **C**、**UML** 操作の実装

下記の UML 操作（`SetClock`）は、ユーザー定義クロック関数の初期化に使用する C 関数（`SetTime`）に整数値を送ります。

```
void SetClock( Integer TimeNow) {
[[
##ifdef USER_CLOCK_FUNC
SetTime((xint32)#(TimeNow));
##endif
]]
}
```

コンパイル時にコンパイラ スイッチ `USER_CLOCK_FUNC` が設定されていなければ、操作は空になります。UML 整数引数は、32 ビット整数タイプ（`xint32`）に応じたタイプ（C コード ジェネレータで定義されたタイプ）です。下記に、`SetTime` 関数プロトタイプに対応する C コードを示します。

```
void SetTime (xint32 newTime);
```

C コードでの # のエスケープ

インライン C コードでは、特殊な意味を持ってシャープ記号（#）が使用されます。Tau でインラインで使用する C コードを記述する場合、従って、シャープ記号（#）をエスケープする必要があります。UML コードでは、シャープ記号（#）はエスケープ文字としても使用します。

```
#(<UML name>)
##0, ##1, ..., ##9
####0, ####1, ..., ####9
```

上記の構成要素は、ソースで定義されたユニットに対する C コード名で置き換えられます。その他のケースでは、# は # 自体を示します。

ただし、意図した C コードをインクルードすることが、上記変換ルールによって不可能になる場合があります。フォーマット文字列では、たとえば # に続けて数字を入れることができます。そのためには、# をエスケープしなければなりません。

中間コードで 3 つ連続した #（つまり ###）は 1 つの # としてコピーされます。これを実現するためには、UML コード内でそれぞれの # をエスケープして、連続した 6 つの # を指定すると、生成されたコード内に 1 つの # が記述されます。

例 **317: ######** のエスケープ

```
######1
```

生成後の記述：

```
#1
```

ビルド前のモデルのチェック

UML セマンティック チェック

ビルド コマンドを実行したら、徹底的なセマンティック チェックを行い、入力した UML モデルが正確、完全なことをベリファイします。C アプリケーションのビルドのコンテキスト内でチェックを行う場合、セマンティック チェッカーがさらにチェックを行い、C コード ジェネレータまたは AgileC コード ジェネレータ、および生成するアプリケーションで使用されるランタイム システムよって課せられた制限を認識します。

モデルがセマンティック上正しいとベリファイされたら、コード生成が開始されます。

中間コード

アプリケーション コードを C コードに生成するための前提条件として、UML ツールは UML モデルをまず中間フォーマットに変換します。中間コードは UML モデルの精製版と見なすことができます。C コード ジェネレータおよび AgileC コード ジェネレータがリアルタイム プロパティを持ったアプリケーション コードを生成するために必要なアクション セマンティックとランタイム ダイナミクスが付け加えられて強化されています。

変換はユーザーに透過的に、いくつかのパワフルな機能を活用して行われます。変換は、アドインの 1 つ **IMGen** によって制御されます。

- この表現を作成する際、変換ツールによってユーザーが追加したすべてのインライン C コードが検出され、C コード ジェネレータおよび AgileC コード ジェネレータに逐語的に転送されます。
- ソース UML モデルでは、ビルド時にコードを精製したくないモデルのパートにタグをつけることができます。
- 変換ツールは、ユーザーがモデル内で「External」というタグを付けた UML 要素の検出も行います。このタグが付いた要素は、コードの定義をどこからでも（完全なアプリケーションをビルドするためにコンパイルおよびリンク スキームに追加された C/C++ ファイルでも）使用できます。

IMGen は、パフォーマンスの観点から、名前によるビルドではなく GUID によるビルドを使用します。この結果、名前解決エラーの一部がビルドプロセスの後のほうで見つかる可能性があり、特定の TIL エラーが起こる可能性があります。この場合は［すべてをチェック］ボタンを押して確認の上、モデルを修正してください。

C コードの生成

C コード ジェネレータおよび AgileC コード ジェネレータは、中間表現コードが正確で完全なことをまずチェックします。中間コードがこのチェックに合格すると、C コード ジェネレータおよび AgileC コード ジェネレータは、そこから完全なアプリケーション C コードを生成します。

UML から C への変換

UML から C への変換、ランタイム セマンティック、および最適化

UML から C への変換の際、C コード ジェネレータおよび AgileC コード ジェネレータは多数の変換ルールを使用し、コードがランタイム セマンティックに従っており、例外的な状況も含めてスケジューリングとシグナリングの問題に安全に準拠していることを確認します。実行時に最高のパフォーマンスを得られるように、かつコード サイズはできるだけ低く抑えるように、コードが生成されます。動的メモリはメモリの断片化を避ける方法で管理されます。

［モデル ビュー］からソース コードへのナビゲート

コード ジェネレータによって、モデルからコードへのナビゲートが可能になります。コード生成の結果としてヘッダー ファイル、または実装ファイルが作成されている場合、［モデル ビュー］で UML 要素を右クリックし、［ソースに移動］を選択すると、対応するコードの行にナビゲートできます。このコマンドは、生成されたコードからモデルへのナビゲートにも使用できます。

生成されたコードへのファイル参照は、「`Result of C/C++ Generation`」という名前の［モデル ビュー］内のパッケージに表示されます。

この機能は、オプション［Generate reference package］に関連づけられている C コード ジェネレータのステレオタイプ（Model Verifier、C コード ジェネレータ）のためのものです。このオプションはデフォルトではアクティブです。

C および C++ のサポート

ユーザーの必要に応じて、C コードはプレーン ISO C コードまたは C++ コンパイラをサポートする C コードとしてとして生成されます。この選択肢は、使用する C/C++ コンパイラに関わる問題に対処するためのものです。また、外部 C++ コードを最終アプリケーションに組み込む場合などにも対応します。

ランタイム ライブラリ

C アプリケーションをビルドする際、コードをコンパイルする最終ターゲットのタイプを指定できます。ホスト コンピュータで実行およびデバッグを行うことができるように、コードをコンパイルすることが、ワークフローの最初の作業になります。アプリケーションの予測どおりの振る舞いをベリファイしたら、ターゲット システムに向けてコンパイルし、さらにテストを行って環境との統合を完了します。

多様な組み合わせがあり、定義済み C および AgileC ランタイム ライブラリが多数提供されているので、あらゆる場面での使用に対応しています。

環境のサポート

自己完結したアプリケーションとなるためには、アプリケーション コードには相互作用するよう設計された環境とのインターフェイスが必要です。相互作用は、環境から送受信されるシグナルを使用して実行されます。いくつかの機能を提供するだけで、環境が適切に初期化されて閉じていることを確証し、UML モデルと実世界との間の相互作用を確実にするため、自己完結したアプリケーション開発してきました。

［Generate environment Template Functions］オプションを設定すると、C コード ジェネレータおよび AgileC コード ジェネレータは以下のものを作成します。

- C アプリケーションのための 環境関数のスケルトン。環境からのシグナルの送受信をハンドリングするコードを記述するために役立ちます。
- システム インターフェイス ヘッダー ファイル。このファイルでシステムとその環境とのインターフェイスが定義されます。また、UML コンセプトの表現を C コードにマッピングするデータ型の定義もこのファイルに含まれます。アプリケーション コードをその環境に統合する機能を持たせるには、このヘッダー ファイルが必要です。通常システム インターフェイス ヘッダー ファイル（`.ifc`）といいます。
- make テンプレート ファイル。アプリケーションのコンパイルとリンクの方法を指定します。この make テンプレート ファイルは、ユーザーがコンパイルおよびリンク スキームに外部 C コードまたは C++ コードをインクルードするために使用できます。

make

ビルド プロセスの最終ステップとして、アプリケーション ビルダは、生成された make ファイルと、make テンプレート ファイル（C コード ジェネレータと AgileC コード ジェネレータの設定で使用するよう指定された場合）を使用して、make を起動します。

生成された C コード、選択したランタイム ライブラリ、および make テンプレート ファイルで定義された外部コードが、make ファイルの作成時に定義されたプロパティに従ってここでコンパイルされます。

実際に使用される make プログラムは、使用および指定する OS とコンパイラの環境によって異なります。

参照

第 20 章「アプリケーション ビルド リファレンス」

第 25 章「C および AgileC ランタイム ライブラリ」

第 31 章「AgileC コード ジェネレータ リファレンス」

実行モード

C コード ジェネレータまたは AgileC コード ジェネレータによって生成されたアプリケーションは、ベアとスレッドの 2 つのモードで実行できます。

ベア

アプリケーションは 1 つのユニットとして実行されます。すべての構成要素は内部スケジューラによって準並列にスケジューリングされます。つまり、遷移は他の遷移に邪魔されることなく必ず最後まで実行されます。遷移が終了すると、イベントが選択され、対応するトランザクションが実行されます。この実行モードをベア モードといいます。

ベア実行モードでは、実行プラットフォームに多くを要求することはありません。アプリケーションがタイマーを使用しているか、他の理由で現在の時刻が必要な場合、プラットフォームにクロックを読み込む方法をインクルードする必要があります。アプリケーションに動的メモリが必要な場合、静的配列内のメモリを管理する AgileC コード ジェネレータに含まれるメモリ パッケージを使用するか、C の標準関数 `malloc` および `free` に似た 2 つの関数を提供することにより、これを実装しなければなりません。

スレッド

もう 1 つの実行モードであるスレッド モードは、実行は並列実行（スレッド、タスク、プロセス、または他の並列実行エンティティ）をサポートする OS に依存します。このモードでは、構成要素がいくつかのグループに分割され、それぞれのグループが 1 つのスレッドで実行されます。スレッド内のスケジューリングは上記の場合と同様準並列実行ですが、スレッド間では、稼動する OS によりスレッド間のコンテキスト スイッチが可能です。このように、優先度の高い操作を優先度の低い操作より先に直ちに実行することが可能です（OS が対応している場合）。

OS との統合およびスレッド モードの実装の詳細については、1100 ページの「コンパイラおよびオペレーティング システムとのインテグレーション」を参照してください。

実行モード選択時の考慮事項

OS の並列処理機能に依存するのが適切かどうかは、最長の遷移を比較して判断します。最長の遷移がアプリケーションの最大のレイテンシとなり、最も重要な状況が検出されてからこれをハンドリングするコードが実行されるまでの最長時間が許可されます。最長の遷移に予想される実行時間が必要なタスク スイッチのレイテンシより短い場合、それ以上の OS を必要とせず、全体的に最良のパフォーマンスを得ることが

できます（ベア）。あるいは、少なくともすべての関連アクティブ クラスを同じスレッドにマッピングすることが可能です。レイテンシが十分ではない場合、外部 OS 内でのプリエンティブ コンテキスト スイッチ（スレッド間）に依存しなければなりません（スレッド）。

条件付きコンパイル

この機能がサポートする主たるシナリオは、ユーザーが複数の異なるバージョンのアプリケーションを記述する UML モデルを作成する必要がある場合です。複数バージョンの例は、異なるプラットフォームに対応する場合やデバッグ／テスト用とリリース用をそれぞれ作成する場合などです。

Tau では、複数バージョンのアプリケーションは、異なるバージョンを記述する異なるアーティファクトを使用して作成します。それぞれのアーティファクトには、アプリケーションの詳細を定義するための情報が含まれます。アプリケーションは、アーティファクトをベースにしたコード ジェネレータを実行して生成します。

条件付きコンパイルのサポートも、同様です。条件付きコンパイルを定義する条件を記述できるよう、アーティファクトが拡張されます。

UML レベルのサポート

UML レベルでは、条件付きコンパイルはステレオタイプ <<conditional>> で定義されます。これは、以下のエンティティに適用できます。

- 定義（クラス、シグナル、操作、インターフェイスなど）
- 複合文（UML テキスト構文で { } で囲まれたブロック）
- 遷移

さらに、条件付き分岐文を定義して、状態機械図で条件付きコンパイルを可能にすることもできます。

ステレオタイプ <<conditional>> には、expr:Boolean という 1 つの属性があります。この属性は、エンティティをアプリケーション生成の一部とするかを決定する式を定義します。

<<conditional>> 要素（および条件付き分岐回答）の式に使用される式は、論理定数、<<conditionalConstant>> でステレオタイプ化されたグローバルまたはパッケージ レベルの属性、定義済み属性 activeCG のテストのみを参照します。また、「true」と「false」の 2 つのリテラルと、以下の操作のみをインクルードします。

- not
- and
- or

activeCG のテストは、現行のビルドにどのコード ジェネレータが使われているかをチェックするために使用されます。activeCG 属性は Charstring 属性なので、その値は自動的に初期化されます。条件式はこの属性を使用してテストを行い、「CGEN」または「CPPAPPGEN」に C コードおよび C++ コード生成時モデルのパートを含めるか、除外するかを判断できます。テストには、以下の例のように、演算子「==」および「!=」を使用できます。

```
<<conditional(.expr = activeCG=="CGEN".)>> {
    i = 10;
}
<<conditional(.expr = activeCG!="CGEN".)>> {
    i = 20;
```

```
}
```

条件定義

構文の観点から、グラフィック構文の定義を使用するステレオタイプを示します。756 ページの図 201 に示すように、属性の値はプロパティ エディタを使用して編集します。

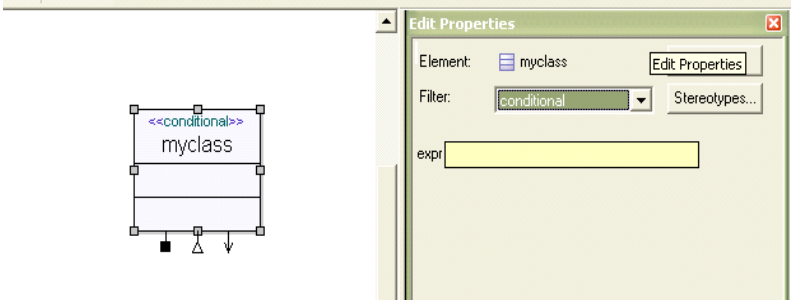


図 201: 属性値の編集

以下のように、UML テキスト構文を使用してステレオタイプを適用することもできます。

```
<<conditional(.expr = "A".)>> class myclass {

}
```

図 202: UML テキスト構文で適用された <<conditional>> ステレオタイプ

条件付きアクション文

複合文に使用できるのは、以下に示すようにテキスト構文だけです。

```
i = 10;
<<conditional(. expr = A .)>> {
  j = 11;
}
```

条件付き遷移

<<conditional>> ステレオタイプは、遷移のグラフィックには出現しません。

条件付き分岐

分岐シンボルを条件付きにできます。定義済み識別子 **CONDITIONAL** を使用して、分岐に条件付きというマークを付けます。757 ページの図 203 に示すように、分岐回答の 1 つに条件式を与えて、このブランチに条件付きアクションをインクルードします。

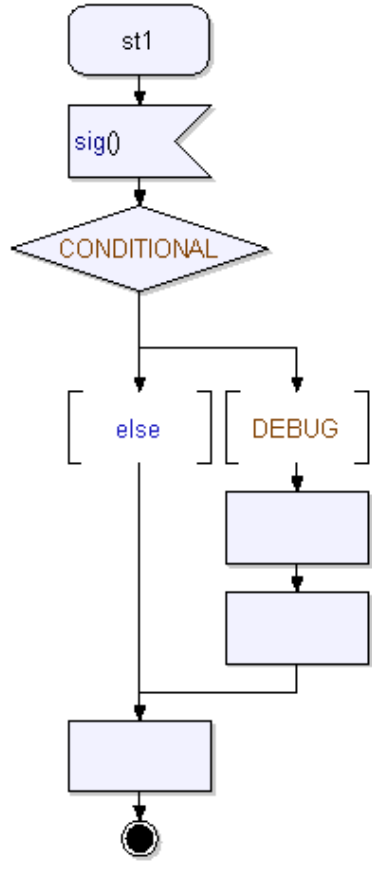


図 203: 条件付きとマークされた分岐シンボル

条件付き分岐の制限事項

条件付き分岐には、以下のようないくつかの静的制約があります。

- 分岐式は、他の条件式と同様のルールに従う `Boolean` 式でなければなりません。

- 分岐には 2 つのブランチのみ許可されます。
- 条件付きアクションを含まないブランチには、「else」とマークします。
- 条件付きブランチは、分岐回答に続く「else」ブランチの前に、「else」ブランチのフローに合流しなければなりません。

アーティファクトと条件付きコンパイル

Tau でアプリケーションをビルドする際、ビルド設定はビルド アーティファクトで記述されます。

この種類のアーティファクトは、条件付きコンパイルを制御する値の設定にも使用されます。

条件式は、グローバル スコープで定義された定数または <<conditionalConstant>> 属性のみ参照できます。アーティファクトは、グローバル スコープ属性にビルド固有の値を定義する手段を提供します。この操作の本体には、<<conditionalConstant>> 属性に値を与える一連の代入文のみが含まれます。

注記

これが、<<conditionalConstant>> 属性に値を与えることができる唯一の方法です。たとえば、モデル内で使用されるすべての <<conditionalConstant>> 属性に conditionalInit 操作で値を与えるようなデフォルト値を設定することはできません。

例 **318:** グローバル属性への代入値 ——————————————————————

以下の 3 つのグローバル属性がある場合、

```
<<conditionalConstant>> Boolean Debug;
<<conditionalConstant>> Boolean Test;
<<conditionalConstant>> Boolean Instrument;
```

これらの属性を以下の conditionalInit() 操作で初期化します。

```
void conditionalInit(){
  Debug = true;
  Test = false;
  Instrument = Debug or Test;
}
```

——————————————————————————————————————

コード生成に及ぼす影響

複数のコード ジェネレータでの <<conditional>> ステレオタイプ／条件付き分岐のサポートの詳細は、特定のコード ジェネレータによって決定され、以下の 2 種類のサポートが提供されます。

- プリプロセッシング手法
- ターゲット言語マッピング手法

プリプロセッシング手法がとられた場合、<<conditional>> ステレオタイプ／条件付き分岐はコード ジェネレータの特定の受け渡し中に処理されます。式が解釈され、条件式が無効になっているモデル要素はモデルから削除されます。生成されるコードには、これらのモデル要素の表現は含まれません。

ターゲット言語マッピング手法がとられた場合、<<conditional>> ステレオタイプ／条件付き分岐は、ターゲット言語での条件文（C/C++#ifdef 文など）にマッピングされます。

前方生成シナリオでは通常プリプロセッシング手法が有用です。これは、ターゲット言語と関係なく適用できます。たとえば、Java は条件付きコンパイルのコンセプトをサポートしていませんが、プリプロセッシング手法では Java コード生成にこれを適用できます。

ターゲット言語マッピングのシナリオは、ラウンドトリップ シナリオと前方生成シナリオの両方で使用できます。ただし、条件付きコンパイルのコンセプトをサポートする言語への生成に限られます。

現在の実装では、C コード ジェネレータでは常にプリプロセッシング手法をとり、C++ コード ジェネレータではラウンドトリップが有効な場合はターゲット言語のマッピング手法、ラウンドトリップが無効な場合はプリプロセッシング手法を用いています。

制限事項

条件付きコンパイルは、コード生成時のみ処理されます。異なる条件式を使用して、同じスコープ内で同じ名前を持つ 2 つの異なるクラスを定義することはできません。

条件付きコンパイルは、Java コードジェネレータではサポートされません。

合成構造

このセクションでは、C Advanced および Agile C コード ジェネレータで提供されるサポートに焦点を当てた、UML 合成構造図の使い方について説明します。特に、合成構造図とパート／全体関係との関わり、およびインスタンスの動的生成機能とともにこれらのコンセプトを使用する方法について説明します。

パートと全体の関係

一般的にソフトウェア エンジニアリングでは、1 つのエンティティが他のエンティティのパートであることを示すことが必要です。通常、複数のコンポーネントが階層状態の他のコンポーネントによって構成されることを示すために使用します。この関係性は、UML では合成関連として示されます。このグラフィック構文を 760 ページの図 204 に示します。

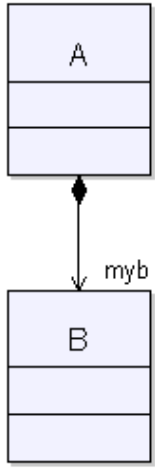


図 204: myb という B パートを包含するクラス A

同じ例をクラス シンボルを使用して示すことができます。パートは属性入力領域内に示されます (761 ページの図 205)。

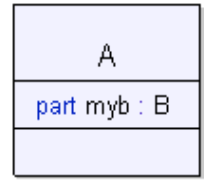


図 205: パートが属性入力領域内に示される例

クラス A のダイアグラム内のテキスト シンボルのように、同じ例をテキスト構文で示すこともできます（761 ページの図 206）。

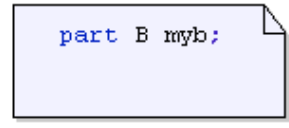


図 206: パートのテキスト構文

これまでの例では、パート／全体関係が 1 対 1 の関係になっています。しかし、関係はもっと複雑になる場合があります。最も一般的なケースとして、コンテナに固定数のパートが含まれる場合、インスタンス数の上限がある／ない動的生成によってパートが生成された場合があります。いずれの場合も、合成関連のパート側の多重度で示されます（762 ページの図 207）。

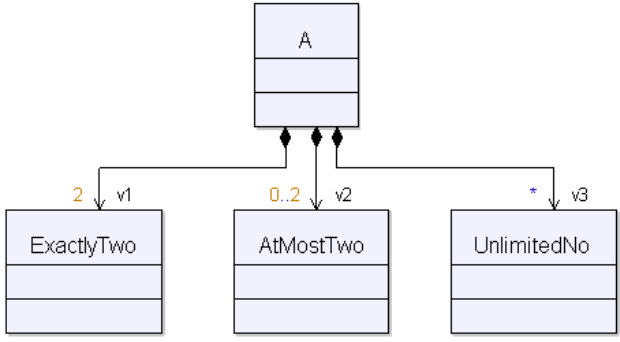


図 207: さまざまな多重度のパートを含むコンテナ

クラス シンボル内で属性定義を使用した同じ例を 762 ページの図 208 に示します。

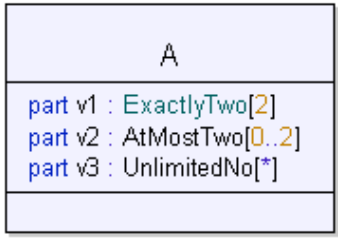


図 208: パートを持つクラス

テキスト シンボル内で定義を使用した同じ例を 762 ページの図 209 に示します。

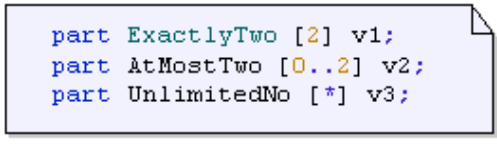


図 209: テキスト シンボル内のパート定義

この例では、多重度が 2 に固定されているので、クラス A の作成時には、必ず `ExactlyTwo` の 2 つのインスタンスが作成されなければなりません。この時点で `AtMostTwo` または `UnlimitedNo` クラスのインスタンスを作成する必要はありません。これらのクラスのインスタンスは A インスタンスのライフタイムの後方で作成されるべきインスタンスです。

A インスタンスが終了すると、その中に包含されているインスタンスも終了します。

これまでの例は、パッシブ クラスの使用のみを示していました。別スレッドの制御があるアクティブ クラスでもセマンティックは同じです。763 ページの図 210 に示すようにこれらのクラスをアクティブにできます。

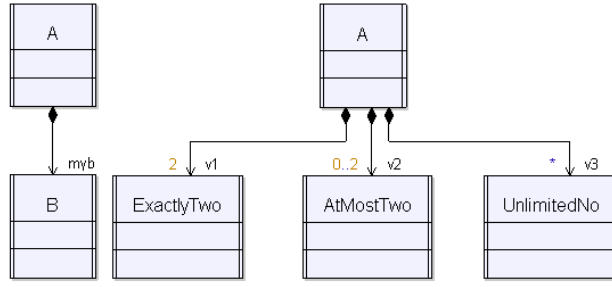


図 210: パートを持つアクティブ クラス

合成構造とパート／全体関係

合成構造図は UML クラスの内部構造を示すために使用します。ここに示す「内部構造」はクラスの属性とそれを接続するコネクタで構成されます。合成属性、非合成属性、クラスによって定められる属性、データ型によって定められる属性など、すべての種類の属性は合成構造図で可視化できます。

一般的なケースの例を 764 ページの図 211 に示します（破線のシンボルは「`aReference`」が合成パートではないことを示します）。

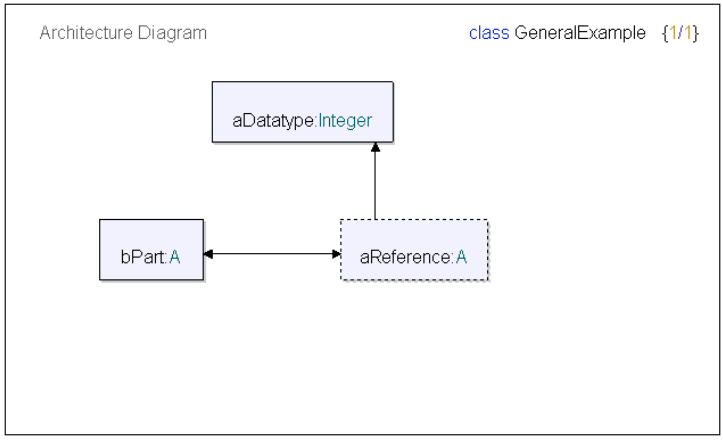


図 211: UML クラスの内部構造を持つ合成構造図

設計の観点からは、合成構造図には以下の 2 つの利点があります。

- 内部構造が可視化されるので、複雑なアプリケーションの構造が理解しやすい。
- アドレッシング メカニズムによってコンポーネント間の「接着剤」の役目を果たすので、パートを個別に設計し包含するエンティティを構成する際に結合させることができる。

可視化は例からわかりますが、アドレッシング メカニズムについてはさらに詳細な説明が必要です。これは、受信側の ID ではなく、ポートによって通信できる能力に基づいています。したがって、クラス A にポート p がある場合、クラス A の振る舞いコードの一部に以下のような文を入れることができます。

```
output s() via p;
```

この文のセマンティックは、シグナル「s」がポート「p」から送信され、「p」に接続されたコネクタを介して転送され、先端のコネクタに接続された誰かに受信されるという意味です。受信側の ID は、「A」がインスタンス化されている合成構造図によって完全に決定されます。

この「接着剤」が機能するためには、クラスが以下のようなルールに従って作成されている必要があります。

- 必須ポートおよび／または実現化されたインターフェイスを持っていること。
- 外部との通信を確立できるポートを使用すること。
- 自己完結型で独自スレッドの制御を持つよう作成されていること。したがって、アクティブ クラスでなければなりません。

通常、コンポーネントは以下の定義で構成されるはずです。

- ポートを持つアクティブ クラス

- 一連のインターフェイス

単純な例を 765 ページの図 212 に示します。

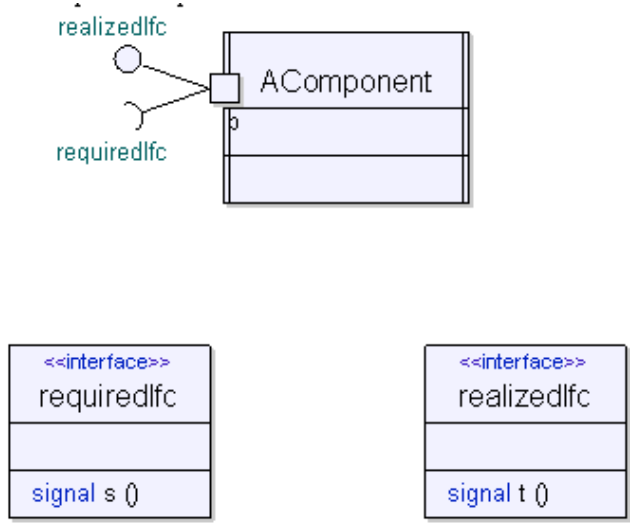


図 212: ポートを持つアクティブ クラス

これで、このコンポーネントを大きいシステムの一部として合成構造図に使用できます（765 ページの図 213）。

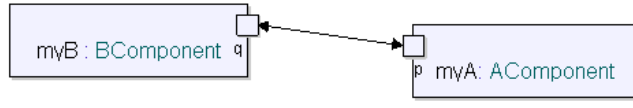


図 213: 合成構造図のコンポーネント

ほとんどの場合、合成構造図はクラスのパートを定義する合成属性に焦点を当てます。このことが、UML でのダイアグラムが（少し誤解されやすいですが）「合成構造図」と呼ばれる 1 つの理由です。

合成構造図は、アクティブ クラスとパッシブ クラスの両方に使用できますが、ゆるやかに結合されたアーキテクチャ エンティティを示すことが多いので、実際にはアプリケーションのアーキテクチャを定義するトップレベルのアクティブなパートを示すために主に使用します。また、非同期通信を使用して通信を行うパート、およびネットワークを介して配布されるパートを示すためにも使用されます。

C コード ジェネレータでは、以下の一般的なケースのサポートに限定されています。すなわち、合成構造図は、アクティブ クラスの内部構造を定義するためだけに使用でき、他のアクティブ クラスによってタイプが定められた合成パートのみ包含できます。

動的に生成されたアクティブ インスタンスとパート／全体関係

UML では、「new」文を使用してアクティブ インスタンスを動的に生成できます。アプリケーションの構造により、新しいインスタンスは包含するエンティティの合成パートとして作成するか、すべての合成階層と合成構造の外側で作成できます。

どちらを選択するかは、インスタンスを使用する目的に依存します。対応するクラス（または通信相手のクラス）は、ポートなどのコンポーネントとしてコード化し、通信ポートに依存し、正常に機能するには内部構造に挿入します。

また、クラスとその周囲のクラスが通信をポートに依存せず、代わりに、シグナルの送信や操作の呼び出し時には必ず受信インスタンスを指定する場合は、そのクラスを包含するエンティティの内部構造に挿入する必要はありません。

AgileC コード ジェネレータでは、合成関連は厳密に実行されます。新しく生成したインスタンスは直接合成関連に挿入しないと、すべての合成関係の外側にあると見なされます。使用する構文は、属性の多重度によって異なります。766 ページの例 319 に例を示します。

例 **319:** 属性の多重度

```
A[0..1] Aref;
// Aref is a reference to one A instance

A[*] Arefs;
// Arefs is a list of reference to A instances

part A[0..1] Apart;
// Apart is a composite part for one A instance
part A[*] Aparts;
// Aparts is a composite part for a list of A instances
```

例 **320:** インスタンスの作成

```
Aref = new A();
// This created an A instance outside all
// composition hierarchies/internal structures
// and keeps a reference to it in Aref´

Arefs.append( new A() );
```

```
// This created an A instance outside all
// composition hierarchies/internal structures
// and added a reference to it in ‘Arefs´

Apart = new A();
// This created an A instance as a composite
// part of the containing entity
// as defined by the composite part ‘Apart´
‘
Aparts.append( new A() );
// This created an A instance as a composite
// part of the containing entity and added
// it to the composite part ‘Aparts´
```

一般的には、合成パートに追加された新しいインスタンスには参照が必要です。これは、767 ページの例 321 に示すように、UML の「offspring」の仕組みによって行います。

例 **321: offspring** からの参照

```
part A[*] Aparts;
A Aref;
Aparts.append( new A() )
Aref = offspring;
// Aref will now reference the newly create instance
```

offspring 変数は、常に生成される最新のインスタンスへの参照を保持します。

offspring の値が、アプリケーション コード中の他の生成操作の影響を受けないことに頼らないことを推奨します。新しいインスタンスを作成する前には、必ず offspring を属性または変数に保存してください。

もう 1 つ注意が必要なのは、合成構造に挿入するインスタンスの動的作成は、包含するクラス内で実行しなければならないことです。つまり、インスタンスが挿入される先のパート属性をもつクラス内で実行しなければなりません。典型的なコンポーネントの構造には、状態機械で定義された振る舞い、および動的に作成されたパートを包含する一連のパートが含まれます。インスタンスの動的作成は状態機械のコードによって実行します。768 ページの図 214 に典型的な例を示します。最初のダイアグラムには、管理ポート（mgm）とサービス ポート（s）で提供される（AServer）を定義するコンポーネント定義が含まれます。

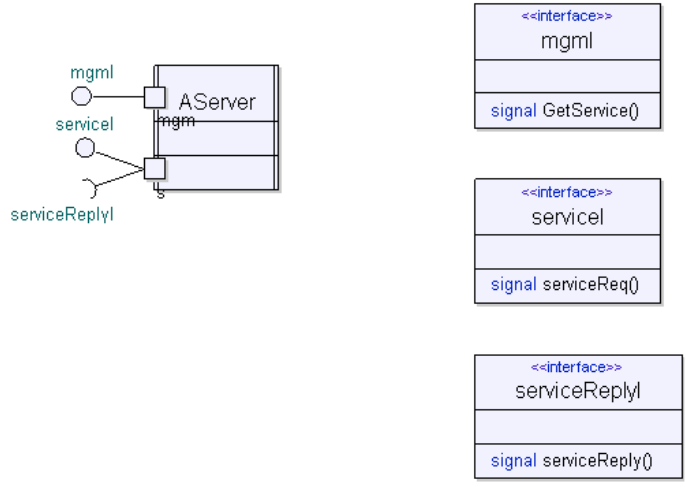


図 214: 管理ポート（mgm）とサービス ポートを持つコンポーネント定義

AServer の内部合成構造は、768 ページの図 215 に示すとおりです。

注記

ローカル定義したクラス LocalServer は serviceI インターフェイスを実装するクラスです。

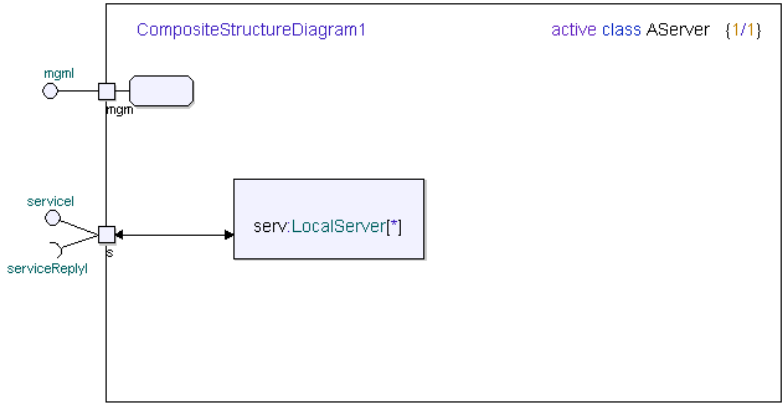


図 215: 1 つのサービスを持つサービス クラス

メイン サーバ クラス（AServer）はサービスのみサポートします。実際のケースでは、もっと多くのサービスがあり、AServer にも多くのローカル パートがあるはずです。AServer クラスの状態機械を 769 ページの図 216 に示します。

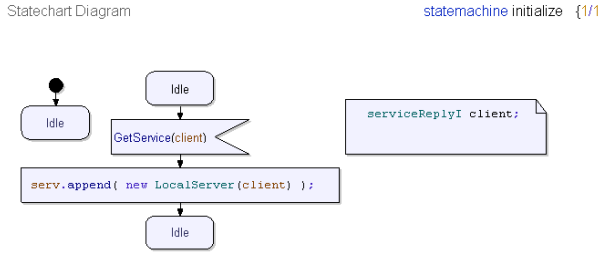


図 216: サーバ クラスの状態機械

769 ページの図 217 に示すように、状態機械は、サービスを実装する単純な LocalServer クラスです。

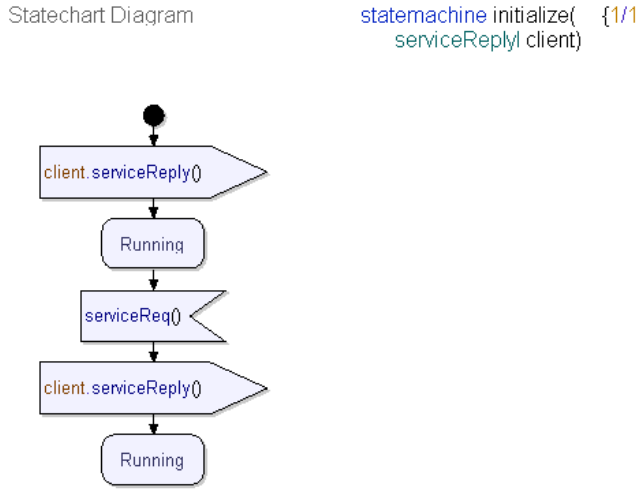


図 217: サービスの状態機械

作成したインスタンスとの通信

上記で説明したように、アクティブ クラスでのシグナルの送信または操作の呼び出しは 2 通りの方法で行うことができます

- 合成構造を使用して受信者を見つける。
- 受信者の明示的なアドレッシングを使用する。

ポートやコネクタという観点では、合成構造を使用する利点は、アプリケーションの初期化が単純になることです。初期化の段階で実行アプリケーションを構成するさまざまなインスタンスに対する参照と通信する必要がないため、動的に割り当てられたメモリの使用を厳しく制約する組み込みアプリケーションなどには、特に重要です。ある極端な（しかし一般的な）ケースとして、すべてのインスタンスが初期化時にすでに静的に割り当てられ、実行中に動的メモリの割り当てがまったくないケースがあります。この場合には、ポートに基づいたアドレッシングが非常に効率的です。

合成構造と動的インスタンス化の組み合わせによりアドレッシングの使用を可能にするには、動的に作成されたインスタンスを挿入することが重要です。このインスタンスには、前のセクションで説明したように、合成構造のポート／コネクタ構造を使用してアクセスできなければなりません。これをしておかないと、通信は失敗します。

インスタンスは合成構造のパートとして動的に作成しないと、ポート／コネクタを通してアクセスできません。その場合、シグナルの送信や操作の呼び出しを行う際、明示的なアドレッシングを行う必要があります。

770 ページの図 218 に、明示的アドレッシングのみを使用する単純な例の静的構造を示します。これは、クラス図の一部です。

図 218: 明示的アドレッシングを使用する静的構造

770 ページの図 218 にはクラス A とクラス C という 2 つのアクティブ クラスがあります。クラス A にはクラス C への参照があります。771 ページの図 219 に示すように、クラス A はクラス A の状態機械からクラス C のインスタンスを作成し、インスタンスにシグナルを送信できます。

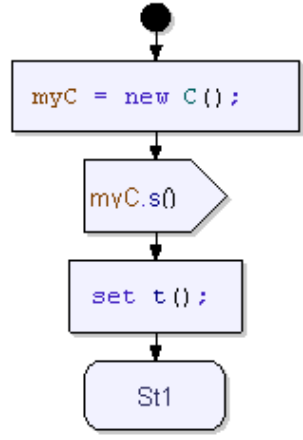


図 219: クラス A の状態機械

この例では、クラス A とクラス C のどちらにもポートを作成していません。また、合成構造も作成していません。シグナルの送信は、ポートではなく直接「myC」参照に基づいて行われます。

上記の例ではクラスにポートを定義していませんが、シグナル送信または呼び出しの指定にポートを使用していなければ、インターフェイスとポートを使用することは可能です。インターフェイスをインターフェイスの実装と切り離していれば、より拡張性のある再利用可能な設計を実現できます。772 ページの図 220 の例では、クラスはすべての振る舞いを維持しながらこれを実現しています。

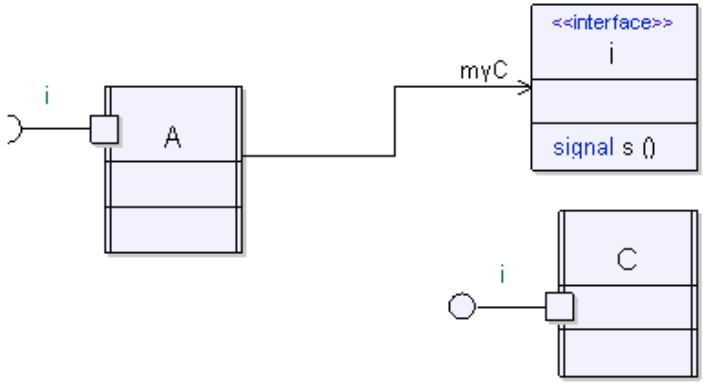


図 220: 切り離されたインターフェイス

この利点は、クラス A およびクラス C が合成構造でコンポーネントとして再利用可能となることです。前の例ではインターフェイスまたはポートを使用していないので、これは不可能です。クラスが再利用を奨めるように設計されている場合は、インターフェイスとポートを使用したほうがよいです。

パートの繰り返し使用

さまざまな状況で、合成属性に含まれるパートなどのすべてのインスタンスは繰り返し使用する必要があります。これは以下の例に示すような方法で行うことができます。773 ページの図 221 に示すような、内部アーキテクチャを持つアクティブクラス Sys があるとします。

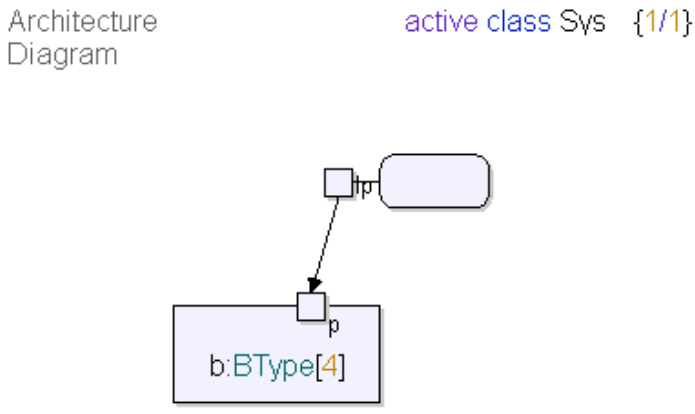


図 221: クラス Sys

773 ページの図 221 に関する簡単な説明：

- 振る舞いポート lp：このポートは、合成構造図で Sys の状態機械を可視化するので、内部パートのポートを Sys 自体の状態機械に接続できます。
- コネクタに関連する名前と情報の流れは省略します。ダイアグラムの詳細によって例が複雑になるのを避けるため、簡単にしています。

この設定で、状態機械 Sys の内部からパート b のインスタンスを繰り返し使用できます。774 ページの図 222 に示すように、これは length 演算子と指数を使用して行います。

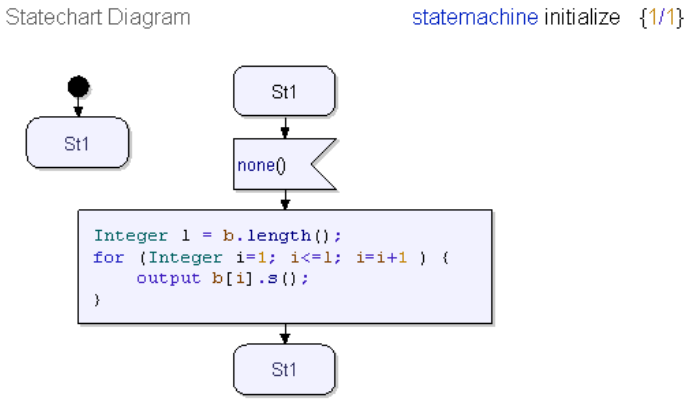


図 222: インスタンスの繰り返し使用

C コード ジェネレータの制限事項

アクティブ クラスのアーキテクチャとコンポジションに関して、C コード ジェネレータ（C コード ジェネレータ、AgileC コード ジェネレータ）が UML モデルに課する制約があります。ここでは、最も重要ないくつかの制約を要約して説明します。これらの制約を守らないと、期待どおりの振る舞いを得られないことになります。

合成構造図とコンポジション

合成構造図では、アクティブ クラスのみ使用できます。

合成構造図では、属性のタイプとしてアクティブ クラスのみ使用できます。

合成構造図の属性は、合成パートのみ、参照は使用できません。

合成クラスの動的作成

内部アーキテクチャがある場合、クラスのインスタンスを動的に作成することはできません。厳密には、アクティブ クラスに他のアクティブ クラスによってタイプが定められた合成パートがある場合、外部クラスのインスタンスを作成することはできません。たとえば、アクティブクラス A を考えます。クラス A の内部には、B（別のアクティブ クラス）によってタイプが定められたパート シンボルを持つ合成構造図があります。この場合、インスタンス A を動的に作成することはできません。

実際には、これは必要なインスタンスをすべて動的に作成する「manager」クラスを持つパターンを使用して処理されます。動的に生成されたアクティブ インスタンスとパート／全体関係で説明しているように、「`AServer`」が包含する「`LocalServer`」インスタンスのマネージャとして動作します。

インスタンスの停止

アクティブ クラスのインスタンスを停止するためには、インスタンスに終了アクションを実行させなければなりません。他のインスタンスから停止させることはできません。アクティブ クラスを「外部」から停止するには、必要なクリーンアップ アクションと終了を実行するシグナル（「terminate」など）による遷移が、アクティブクラスに含まれている必要があります。

インスタンスの作成

合成属性への追加などのように、アーキテクチャに挿入するインスタンスは、属性の所有者が作成しなければなりません。したがって、新しいインスタンスを作成する際、包含するスコープで定義された属性をネストされたクラス内から参照することはできません。これに対応するため、属性の所有者側で演算子を定義し、それによって属性の作成と追加を行い、新しく作成されたクラスに参照を返します。この演算子はネストされたクラスから呼び出すことができます。

Tau での CPtr タイプの使用

CPtr タイプは、UML モデリングで低レベルのポインタ型の機能を提供するためのものです。これは、手動でコード化された C コンポーネントまたは C++ コンポーネントとインターフェイスする UML アプリケーションのためのタイプですが、C コード生成用のすべての UML アプリケーションで使用できます。外部 C/C++ 宣言をインポートする場合、外部コードのポインタはすべて UML モデルの CPtr インスタンス化に変換されます。

概要

例として、以下のような .h ファイルのフラグメントを考えます

```
int * ip;
typedef struct c {
  int i;
} * cp;
int op( cp p );
```

これは、UML では以下のように変換されます。

```
CPtr<int> ip;
class c {
  int i;
}
syntype cp = CPtr<c>;
int op( cp p );
```

UML の観点からは、CPtr タイプは、データ型にクラス プロパティを付与するラッパー クラス、または UML の通常のクラス参照の代わりに追加機能を提供するものと見ることができます。CPtr タイプは CppTypes アドインによってロードされる TTDCppPredefined UML パッケージで定義され、以下の 4 つの演算子を持ちます。

```
SetValue
GetValue
GetAddress
'[]'
```

これらの演算子を UML モデルで使用する方法について、これから説明します。

CPtr とデータ型

CPtr をデータ型とともに使用すると、以下の機能を提供するラッパー クラスとして動作します。

- new 文を使用して動的に作成する。
- GetValue メソッドを使用して包含される値にアクセスする。
- SetValue メソッドを使用して包含される値を変更する。
- `GetAddress` 演算子を使用して既存の属性または変数に基づいて作成する。
- 配列および指数として表示する。これは、C/C++ コードに配列が割り当てられており、外部 C/C++ コードとの統合を主な目的としている場合のみ可能です。

例 **322: CPtr** の割り当てとアクセス

```
class c {
  Integer i;
  Integer j;
  CPtr<Integer> ip;
  CPtr<Integer> jp;
  void test() {

    /* Allocation of a CPtr, usage of SetValue/GetValue */
    ip = new CPtr<Integer>();
    ip.SetValue(10);
    i = ip.GetValue(); // ‘i’ is now 10

    /* Accessing an attribute using GetAddress */
    j = 20;
    jp = CPtr<Integer>::GetAddress(j);
    jp.SetValue(30); // ‘j’ is now also set to 30

    /* Viewing a CPtr as an array */
    CPtr<Integer> intArray;
    Integer len,index;
    [[
      #(intArray) = malloc(10*sizeof(#(Integer)));
      #(len) = 10;
    ]]
    for (index=0; index<len; index=index+1) {
      intArray[index] = 44;
    }
  }
}
```

CPtr とクラス

CPtr はクラスとともに使用し、オブジェクトへの参照の低レベル表示を提供します。しかし、通常の参照の使用と比べて大きな利点はないので、純粋な UML モデリングで使用することは推奨できません。

CPtr とクラスは、外部 C/C++ コードにアクセスするために主に使用します。外部 C/C++ コードの構造体とクラスは、対応する UML のクラス定義を生成します。次のような C コードの例を考えます。

```
typedef struct c {
  int i;
} * cp;
int op( cp p );
```

インポート後の UML モデルには次の宣言が含まれます。

```
class c {
  public int i;
}
syntype cp = CPtr<c>;
int op( cp p );
```

これは、UML モデルでは振る舞いコードから使用できます。例：

```
cp mycp;
int i;
mycp = new cp();
i = op( mycp );
```

CPtr の再帰的使用

CPtr は再帰的に使用できます。これは C/C++ コードにおける「ポインタへのポインタ」に相当し、データ型に適用された CPtr 型と、クラスに適用された CPtr 型の両方で機能します。クラスに対しては 1 レベル分間接性が少なくなります。これは、あるクラスへの参照が、意味的に、そのクラスへの CPtr として扱われるからです。したがって、"new c()" は "c" への参照を返し、"new CPtr<c>()" は "c" への参照へのポインタを返します。

例 **323:** 再帰的に使用される **CPtr**

```
class c {
   CPtr<Integer> ip;
   CPtr< CPtr<Integer> > ipp;
   CPtr<c> cp;
   CPtr< CPtr<c> > cpp;
   void test() {
     ip = new CPtr<Integer>();
     ipp = new CPtr< CPtr<Integer> >();
     ip.SetValue(11);
     ipp.SetValue(ip);
     cp = new c();
     cpp = new CPtr<c>();
     cpp.SetValue(cp);
   }
}
```

CPtr タイプの代入式との互換性

CPtr インスタンス化は代入式と互換があり、以下のような式にできます。

```
CPtr<Integer> i1;
CPtr<Integer> i2;
i1 = i2;
```

演算子を呼び出すと、暗黙的な代入式が実行されます。

例 **324:** 外部関数のインポート

```
int op(int * i);
```

例 **325: UML** モデルの振る舞いコード内の外部関数

```
CPtr<Integer> ip;
ip = new CPtr<Integer>();
ip.SetValue(10);
```

```
op( ip );
```

SetValue/GetValue の繰り返し使用

SetValue/GetValue を繰り返し呼び出し、以下のような構成で使用できます。

```
class c {
  syntype ipType = CPtr<Integer>;
  syntype ippType = CPtr<ipType>;
  ipType ip;
  ippType ipp;
  void test() {
    ip = new ipType();
    ipp = new ippType();
    ipp.SetValue(ip);
    ipp.GetValue().SetValue(10);
  }
}
```

CPtr と参照間の変換

クラスに適用した CPtr とクラスへの通常の参照との間で、変換が可能でなければなりません。これはツールで完全にサポートされており、以下のように使用できます。

```
class c {
  public Integer i;
}
syntype CPtr_c = CPtr<c>;
class test {
  CPtr_c cp;
  c cr;
  void test() {
    cr = new c();
    cp = cr;
  }
}
```

以下の例のように、パートでも同様のことを行うことができます。

```
part c cv;
CPtr<c> cp;
void test() {
   cv.i=10;
   cp = cv;
}
```

OS のスレッド インテグレーション

概要

RTOS

ここでいう RTOS インテグレーションとは、特定の RTOS（リアルタイム OS）に対応するカーネル ソースの適用のことです。このインテグレーションにより、Tau の UML モデルで生成されたアプリケーションが、RTOS 上で実行できます。この適用は、限定 OS に配布されたコードで一部準備されます。

バージョンの差異

ここで説明するスレッド インテグレーションは従来のものと大きな違いはありません。インテグレーション原理はほとんど同じです。一番大きな違いは、旧バージョンでのインテグレーションはマクロで表しましたが、新バージョンでは関数インターフェイスを使用することです。また、旧バージョンではすべて 1 つのヘッダー ファイルに入れられていましたが、新バージョンでは各 RTOS インテグレーションは 1 つのヘッダー（.h）ファイルと 1 つの本体（.c）ファイルを使用して実装されます。これらの点は、読みやすくするためとデバッグを単純化する目的で変更されました。ファイル構造の変更により、インテグレーションの実装と、お客様によるインテグレーションの変更が簡単になりました。

従来のインテグレーション：

- SUN Solaris（UNIX 系 OS で使用可能な POSIX pthread インテグレーション）
- Win32

新しいインテグレーション：

- POSIX pthreads（SUN Solaris および Linux でテスト済み、ほとんどの UNIX 系 OS で使用可能）
- Win32

RTOS インテグレーション ファイル

RTOS インテグレーションは rtapidef.h と rtapidef.c という 2 つのファイルで構成されます。rtapidef.h ファイルは scttypes.h ファイルに内包され、rtapidef.c ファイルは sctsdl.c ファイルに内包されています。rtapidef.c ファイルは sctsdl.c ファイルに内包されているので、新しくコンパイルするファイルはなく、後で説明するコンパイル オプション以外に make ファイルは影響を受けません。

コンパイル スイッチ

コンパイルの段階では、現在のスレッド インテグレーションで使用されているコンパイル スイッチは定義しないでください。新しいインテグレーションを使用するには、以下を指定します。

- アプリケーション カーネルを選択するスイッチ。例：SCTAPPLCLENV
- スイッチ THREADED
- rtapidef.h ファイルと rtapidef.c ファイルの場所をコンパイラに知らせるコンパイラ オプション

例：`cc -DSCTAPPLCLENV -DTHREADED -I/some/suitable/path`

スレッド インテグレーションの詳細について、次に説明します。

スレッド インテグレーション

新しいインテグレーションの実装、および既存のインテグレーションの理解のためには、この説明のほか、既存のインテグレーションのコードについての説明書も併せてお読みください。リアルタイム OS とのインテグレーションを実装するには、以下のことを行う必要があります。

- クロック関数を実装する。
- 複数のミューテックスまたはバイナリ セマフォを使用して共有データを保護する。
- 関連するプロパティを持つスレッドを作成して同期するため、スタートアップ コードを使用する。
- 何もすることがない場合、スレッドの実行を停止する。スレッド内のパートにシグナルを送信する場合には再実行できなければなりません。

インテグレーションの詳細を説明するため、例として POSIX インテグレーションを使用します。以下に示すコードの他に、必要なコンセプトにアクセスするため、rtapidef.h には必要なシステム インクルード ファイルがなければなりません。

例 **326: POSIX** の **rtapidef.h** のインクルード

```
#include <pthread.h>
#include <sched.h>
#include <semaphore.h>
#include <time.h>
#include <sys/time.h>
```

RTOS に main 関数が必要な場合（例では必要）、XMAIN_NAME を以下のような名前に定義して、uml_kern.c に含まれるメイン関数の名前を変更できます。

```
#define XMAIN_NAME agilec_main
```

次に、`agilec_main` 関数を呼び出す適切な main 関数を実装しなければなりません。

クロック関数

タイマーの SDL コンセプトをサポートするため、クロック関数が必要です。生成されたコードとカーネルは、現在の時間を返す xNow 関数があることを前提としています。時刻は、以下のように定義される SDL_Time のタイプによって表されます。

```
typedef struct
  s, ns xint32;
} SDL_Time;
```

xint32 は 32-bit int として実装されます。実装されたクロック関数により、コンポーネント「s」と「ns」は、ある時刻から経過した秒数またはナノ秒数を表します。

クロック機能には、UNIX 系システム用と Windows 用の 2 つの標準実装があります。システム クロックの読み出しに、Windows では標準関数 _ftime が使用され、UNIX 系のシステムでは標準関数 clock_gettime が使用されます。

クロック関数を実装するには、以下の詳細に従って、自分の rtapidef.h ファイルおよび rtapidef.c ファイルにコードを入れます。

タイマーを使用せず、SDL または C で明示的にクロックにアクセスしない場合、クロックを実装する必要はありません。その場合、以下のマクロ定義だけ含めます。

```
#define xInitSystime()
```

rtapidef.h ファイル：

クロックの実装が必要な場合、以下のプロトタイプを rtapidef.h ファイルに入れます。

```
extern void xInitSystime(void);
extern SDL_Time xNow (void);
```

初期化関数が必要ない場合、xInitSystime 関数の代わりに以下のマクロを入れます。

```
#define xInitSystime()
```

rtapidef.c ファイルには、これらの関数の実装が必要です。実装は、ソフトウェアとハードウェアのアーキテクチャに対するサポートに大きく依存します。

共有データの保護

使用できるシグナルやタイマーのリストなどを保護する必要があります。このためには、4 つのグローバル ミューテックスまたはバイナリ セマフォが必要です。これらの変数は rtapidef.h で定義された extern で、rtapidef.h で宣言されていなければなりません。変数の名前は、以下に示す例と同じでなければなりません。

例 **327: rtapidef.h** ファイル： ——————————————————————————

```
extern pthread_mutex_t xFreeSignalMutex;
extern pthread_mutex_t xFreeTimerMutex;
extern pthread_mutex_t xCreateMutex;
#ifdef USER_CFG_USE_MEMORY_PACK
  extern pthread_mutex_t xMemoryMutex;
```

```
#endif
```

例 **328: rtapidef.c** ファイル：

```
pthread_mutex_t xFreeSignalMutex;
pthread_mutex_t xFreeTimerMutex;
pthread_mutex_t xCreateMutex;
#ifdef USER_CFG_USE_MEMORY_PACK
  pthread_mutex_t xMemoryMutex;
#endif
```

これらの 4 つの変数は、アプリケーションの起動時に初期化されロック解除状態にならなければなりません。この初期化には `xThreadInit` 関数が使用されます。

例 **329: xThreadInit**

```
void xThreadInit (void)
{
  (void)pthread_mutex_init(&xFreeSignalMutex, 0);
  (void)pthread_mutex_init(&xFreeTimerMutex, 0);
  (void)pthread_mutex_init(&xCreateMutex, 0);
  #ifdef USER_CFG_USE_MEMORY_PACK
    (void)pthread_mutex_init(&xMemoryMutex, 0);
  #endif
  ....
}
```

また、ミューテックスまたはバイナリ セマフォのため、ロックおよびロック解除操作を実装する必要があります。以下の 2 つの関数を実装します。

`rtapidef.h` ファイルの例：

```
extern void xThreadLock (pthread_mutex_t *);
extern void xThreadUnlock (pthread_mutex_t *);
```

`rtapidef.c` ファイルの例：

```
void xThreadLock (pthread_mutex_t *M)
{
  (void)pthread_mutex_lock(M);
}

void xThreadUnlock (pthread_mutex_t *M)
{
  (void)pthread_mutex_unlock(M);
}
```

起動フェーズ - スレッドの作成

基本的な初期化の後で、カーネルは main 関数内で指定スレッドを起動します。スレッドごとに xThreadInitOneThread 関数および xThreadStartThread 関数が呼び出されます。xThreadInitOneThread はスレッド固有の初期化を行い、xThreadStartThread はスレッドを起動します。スレッドごとにカーネルで宣言された xMainLoop が実行されます。これは、ラッパー関数 xThreadEntryFunc を使用して実行されます。このラッパー関数はインテグレーションで定義され、スレッドで実際に開始される関数です。

すべてのスレッドが起動したら main 関数で xThreadGo 関数が呼び出されます。これらの関数の詳細を次に説明します。

すべてのスレッドが作成するまで、起動したスレッドは SDL 遷移を実行しません。したがって、xThreadEntryFunc は最初のアクションとしてセマフォを待ちます。すべてのスレッドが作成されると、xThreadGo 関数がこのセマフォをリリースします。

多くの関数には、パラメータとして xSystemData タイプへのポインタがあります。ここにはスレッドのローカル情報があります。また、RTOS インテグレーションで定義された xThreadVars タイプのフィールドもあります。

例：rtapidef.h ファイル内の xThreadVars タイプ

```
typedef struct {
  pthread_mutex_t SignalQueueMutex;
  pthread_cond_t  SignalQueueCond;
  pthread_t       ThreadId;
} xThreadVars;
```

最初の 2 つのフィールドについては後で説明します。ThreadId は起動時にスレッドの ID を保存するために使用されます。

このセクションで記述される振る舞いのコードは、以下の例のようになります。

rtapidef.h ファイルの例：

```
extern sem_t xInitSem;
#if !defined(USER_CFG_USE_xInEnv) && !defined(XENV)
  extern sem_t xMainThreadSem;
#endif

extern void xThreadInitOneThread (
   struct _xSystemData *);
extern void xThreadStartThread (
   struct _xSystemData *,
   unsigned int, unsigned int,
   unsigned int, unsigned int);
```

rtapidef.c ファイルの例：

```
sem_t xInitSem;
#if !defined(USER_CFG_USE_xInEnv) && !defined(XENV)
  sem_t xMainThreadSem;
#endif

void xThreadInit (void)
{
```

```
  ....
  (void)sem_init(&xInitSem, 0, 0);
}

void xThreadInitOneThread(struct _xSystemData *xSysDP)
{
  (void)pthread_mutex_init(
    &xSysDP->ThreadVars.SignalQueueMutex, 0);
  (void)pthread_cond_init(
    &xSysDP->ThreadVars.SignalQueueCond, 0);
}


static void *xThreadEntryFunc (void *xSysDP)
{
  (void)sem_wait(&xInitSem);
  (void)sem_post(&xInitSem);
  xMainLoop((xSystemData *)xSysDP);
}


void xThreadStartThread(struct _xSystemData *xSysDP,
                        unsigned int StackSize,
                        unsigned int Prio,
                        unsigned int User1,
                        unsigned int User2)
{
  pthread_attr_t Attributes;
  ....
  (void)pthread_create(&xSysDP->ThreadVars.ThreadId,
                       &Attributes, xThreadEntryFunc,
                       (void *)xSysDP);
  ....
}

void xThreadGo(void)
{
  (void)sem_post(&xInitSem);

  #if defined(USER_CFG_USE_xInEnv)
    xInEnv();  /* AgileC */
  #elif defined(XENV)
    xInEnv(xNow());  /* Cadvanced */
  #else
    (void)sem_init(&xMainThreadSem, 0, 0);
    (void)sem_wait(&xMainThreadSem);
  #endif
}
```

`xInitSem` セマフォはスレッドの同期に使用されます。`xThreadInit` の最初に 0 に初期化され、ステートをブロックします。その後、各スレッドで 1 回ずつ `xThreadStartThread` が起動します。関数 `pthread_create` は、3 つ目のパラメータとして指定された関数（`xThreadEntryFunc`）を、4 つ目のパラメータとして指定された void * パラメータ（`xSysD`）とともに呼び出します。`pthread_create` はまた、最初のパラメータとして渡される変数にスレッドの ID を保存します。2 つ目のパラメータはスレッドのプロパティです。これについては後で説明します。

すべてのスレッドが作成される前にいずれかのスレッドが実行されてしまった場合、そのスレッドは xThreadEntryFunc 内の sem_wait を保留し、メイン スレッドが xThreadGo を呼び出してセマフォ xInitSem を 1 回提示するのを待ちます。このセマフォを待つスレッドのいずれかが実行可能になると、直ちにまたセマフォが提示されます。すべてのスレッドが実行されるまで、これが続きます。

OS とアプリケーションのプロパティにより異なりますが、すべてのスレッドが実行されたら、メイン スレッドは別のことを実行できるようになります。まず xInEnv 関数を呼び出し、それをスレッドで実行することを推奨します。詳細については xInEnv の説明を参照してください。あるいは、上記のように、セマフォ xMainThreadSem を使用して（xInEnv が使用されていない場合）メイン スレッドのセマフォを保留します。この場合、どこかで xMainThreadSem セマフォを提示してメイン スレッドの実行を再開します。

メイン スレッドが xThreadStart 関数から戻った場合、プログラムは main 関数での実行を継続し、終了の呼び出しを行います。メイン スレッドが終了を実行した場合、スレッド プログラムの振る舞いは、OS に依存します。POSIX pthreads では、このような場合すべてのスレッドが終了します。xThreadStart 関数の最後にメイン スレッドを保留するのが重要なのは、このためです。

次に、スレッドのプロパティについて説明します。ほとんどの RTOS では、スタック サイズや優先度などのプロパティは個々のスレッドに設定できます。スレッド アーティファクトの定義とともに、4 つの整数値を指定できます。

- 最初の値はスタック サイズと解釈されます。
- 2 つ目の値は優先度と解釈されます。
- 3 つ目と 4 つ目の値は RTOS またはユーザーが定義したプロパティのために使用できます。

現在定義済みのインテグレーションでは最初の 2 桁のみ使用されています。これらの値はパラメータとして xTreadStartThread 関数に渡されます。

プロパティの詳細な設定は、RTOS に依存します。xThreadStartThread 関数などで使用可能なインテグレーションを見てください。

rtapidef.h ファイルで 4 つの xThreadData フィールドの適切なデフォルト値を設定します。スレッド 定義で値を指定しないと、これらのデフォルト値が使用されます。

例：

```
#define DEFAULT_STACKSIZE     1024
#define DEFAULT_PRIO             0
#define DEFAULT_USER1            0
#define DEFAULT_USER2            0
```

スレッドの停止と再開

ある一定期間スレッドで実行することがない場合、スレッドは自身を停止して他のスレッドにプロセッサを解放します。タイマーが切れるか処理が必要になった場合、あるいは他のスレッド（xInEnv など）が停止中のスレッドで処理されるべきシグナルを送信した場合、スレッドが再開されなければなりません。

これらの機能を実装するには、ある種の条件変数とともに 1 つのミューテックスまたはバイナリ セマフォを使用します。time-out とともに、または単独で条件 wait を実行できなければなりません。停止中のスレッドを再開するシグナルを送信する方法がなければなりません。スレッドごとにこれら 2 つのエンティティが必要です。これは、前に説明したように、xThreadVars 構造体に含まれています。

```
typedef struct {
  pthread_mutex_t SignalQueueMutex;
  pthread_cond_t  SignalQueueCond;
  pthread_t       ThreadId;
} xThreadVars;
```

SignalQueueMutex は、スレッドの外側からのシグナルが挿入されるシグナル キューを保護するためのものです。SignalQueueCond は条件付き wait を可能にします。

SignalQueueMutex は xThreadInitOneThread で初期化されます。SignalQueueCond を初期化する必要がある場合、同じ場所で実行されます。

例 **330**

```
void xThreadInitOneThread(struct _xSystemData *xSysDP)
{
 (void)pthread_mutex_init(&xSysDP->ThreadVars.SignalQueueMutex, 0);
 (void)pthread_cond_init(&xSysDP->ThreadVars.SignalQueueCond, 0);
}
```

SignalQueueMutex は、前に説明したように、xThreadLock 関数によってロックされます。ロック解除は以下の 3 つの方法で行われます

- xThreadUnlock（前述）
- xThreadWaitUnlock
- xThreadSignalUnlock

xThreadWaitUnlock は、スレッドが自身を停止すべきだと判断した際にスレッド自身から呼び出されます。xThreadSignalUnlock は停止中のスレッドを再開したい別のスレッドから呼び出されます。どちらの関数も操作を実行するスレッドの xSysD ポインタを渡します。

rtapidef.h ファイルの例：

```
extern void xThreadWaitUnlock (struct _xSystemData *);
extern void xThreadSignalUnlock (struct _xSystemData *);
```

rtapidef.c ファイルの例：

```
void xThreadWaitUnlock (struct _xSystemData *xSysDP)
{
  #if defined(CFG_USED_TIMER) || defined(THREADED)
  #ifdef THREADED
      /* Cadvanced */
    if (xSysDP->xTimerQueue->Suc==xSysDP->xTimerQueue) {
  #else
      /* AgileC */
    if (! xSysDP->TimerQueue) {
  #endif
      (void)pthread_cond_wait(
        &xSysDP->ThreadVars.SignalQueueCond,
        &xSysDP->ThreadVars.SignalQueueMutex);
    } else {
      struct timespec timeout;
      #ifdef THREADED
          /* Cadvanced */
        timeout.tv_sec =
          ((xTimerNode)xSysDP->xTimerQueue->Suc)->
            TimerTime.s;
        timeout.tv_nsec =
          ((xTimerNode)xSysDP->xTimerQueue->Suc)->
            TimerTime.ns;
      #else
          /* AgileC */
        timeout.tv_sec = xSysDP->TimerQueue->Time.s;
        timeout.tv_nsec = xSysDP->TimerQueue->Time.ns;
      #endif
      (void)pthread_cond_timedwait(
        &xSysDP->ThreadVars.SignalQueueCond,
        &xSysDP->ThreadVars.SignalQueueMutex,
        &timeout);

    }
  #else
    (void)pthread_cond_wait(
      &xSysDP->ThreadVars.SignalQueueCond,
      &xSysDP->ThreadVars.SignalQueueMutex);
  #endif
  (void)pthread_mutex_unlock(
    &xSysDP->ThreadVars.SignalQueueMutex);
}

void xThreadSignalUnlock (struct _xSystemData *xSysDP)
{
  (void)pthread_cond_signal(
    &xSysDP->ThreadVars.SignalQueueCond);
  (void)pthread_mutex_unlock(
    &xSysDP->ThreadVars.SignalQueueMutex);
}
```

`xThreadWaitUnlock` または `xThreadSignalUnlock` が呼び出されると、`SignalQueueMutex` がロックするので、関数の最後にロックを解除しなければなりません。

`xThreadWaitUnlock` では、スレッドは自身を停止しようとします。タイマーを使用し、タイマー キューにアクティブなタイマーがある場合、タイマーが切れるか別のスレッドによって再開されるまで待ちます。POSIX pthread では、`pthread_cond_wait`

関数によって同じことが実行されます。タイマーを使用していないかアクティブなタイマーがない場合、誰かが再開するまでスレッドは停止されたままになります。POSIX pthreads では、`pthread_cond_wait` 関数によって同じことが実行されます。

`xThreadSignalUnlock` では、パラメータによって指定されたスレッドが再開します。ここでは `pthread` 関数 `pthread_cond_signal` を使用できます。

ここで説明したインテグレーションは、SDL Suite でも使用できます。スレッド インテグレーションの実装には、いくつかの要件があります。まず、スレッドを停止できる関数が必要です。

rtapidef.h ファイル：

```
#if defined(THREADED) || defined(CFG_USED_DYNAMIC_THREADS)
extern void xThreadStopThread(struct _xSystemData *);
#endif
```

rtapidef.c ファイル：

```
#if defined(THREADED) || defined(CFG_USED_DYNAMIC_THREADS)
void xThreadStopThread(struct _xSystemData *xSysDP)
{
  pthread_mutex_destroy(&xSysDP->ThreadVars.SignalQueueMutex);
  pthread_cond_destroy(&xSysDP->ThreadVars.SignalQueueCond);
  pthread_exit(0);
}
#endif
```

`xThreadStopThread` 関数はスレッド固有のセマフォをクリーンアップし、スレッドを停止します。この関数を呼び出してスレッドを停止するのは、スレッド自身です。

また、もう 1 つの違いは、2 つのコード ジェネレータのタイマーにアクセスする方法が異なることです。これは、`xThreadWaitUnlock` 関数の詳細に影響します。上記の関数の `#ifdef THREADED` の下のセクションに注目してください

例 **331: scttypes.h** の **#define** 文

以下の #define も関連します（scttypes.h から）：

```
#define THREADED_GLOBAL_VARS
#define THREADED_GLOBAL_INIT ¥
     xThreadInit();
#define THREADED_THREAD_VARS ¥
     xThreadVars ThreadVars;
#define THREADED_THREAD_INIT(SYSD) ¥
     xThreadInitOneThread(SYSD);
#define THREADED_THREAD_BEGINNING(SYSD)
#define THREADED_AFTER_THREAD_START ¥
     xThreadGo();
#define THREADED_START_THREAD(F, SYSD, STACKSIZE, PRIO, USER1,
USER2) ¥
xThreadStartThread(SYSD, STACKSIZE, PRIO, USER1, USER2);
#define THREADED_STOP_THREAD(SYSD) ¥
     xThreadStopThread(SYSD);
#define THREADED_LOCK_INPUTPORT(SYSD) ¥
     xThreadLock(&SYSD->ThreadVars.SignalQueueMutex);
#define THREADED_UNLOCK_INPUTPORT(SYSD) ¥
     xThreadUnlock(&SYSD->ThreadVars.SignalQueueMutex);
#define THREADED_WAIT_AND_UNLOCK_INPUTPORT(SYSD) ¥
     xThreadWaitUnlock(SYSD);
#define THREADED_SIGNAL_AND_UNLOCK_INPUTPORT(SYSD) ¥
     xThreadSignalUnlock(SYSD);
```

```
#define THREADED_LISTREAD_START  xThreadLock(&xFreeSignalMutex);
#define THREADED_LISTWRITE_START xThreadLock(&xFreeSignalMutex);
#define THREADED_LISTACCESS_END  xThreadUnlock(&xFreeSignalMutex);
#define THREADED_EXPORT_START    xThreadLock(&xCreateMutex);
#define THREADED_EXPORT_END      xThreadUnlock(&xCreateMutex);
```

アプリケーション例

このセクションで示す例を理解して活用するためには、Tau に関する知識があり、プロジェクトおよびモデルとダイアグラムの操作に精通し、アプリケーションのビルド方法がわかっている必要があります。

ここで示すサンプルはすべて、Tau の［ファイル］メニューから［新規］ダイアログを開いて開始できます。［テンプレート］タブをクリックし、目的のテンプレートを選択して実行します。

環境を備えた例（EchoServer）

サンプルを使用して実習を行うことができます。アプリケーションをビルドして実行したり、用意されているコードを学習し、UML、C 言語および C++ 言語のインテグレーションの実装方法を理解できます。

振る舞い

サンプルに含まれるアプリケーションは、限定的で単純な機能を持つ小さいサーバをモデルとしています。アプリケーションの振る舞いは以下のとおりです。

1. 環境からシステムに送信されるシグナル `Say(Charstring, Duration)` を受信すると、アクティブ クラス Server がアクティブ クラス RequestHandler のインスタンスを動的に作成します。
2. RequestHandler は Duration パラメータで指定された期間待ちます。
3. Charstring パラメータは自身に連結します。
4. 最終的に、アクティブ クラス RequestHandler から環境に対してシグナル Echo(Charstring) が送信されます。

デプロイメントとスレッドの例

「デプロイメント」サンプルは、コード生成のデプロイメントとスレッドの側面に重点が置かれています。生成されたコードの有用性を重要と位置づけるているため、コード ジェネレータの主なフレーバとして AgileC コード ジェネレータが使用されます。

振る舞い

デプロイメント サンプルは典型的なクライアント／サーバ システムです。最上位のアクティブ クラス Top には、Master タイプの m と Slave タイプの s があります（792 ページの図 223 の左部分）。マスターはスレーブに要求を送信し、応答を待ちます。これは 100 回繰り返されます（200 のシグナルが送信されます）。マスターは、最初のシグナルの送信前に文字列「Starting」を `stdout` に出力し、最後のシグナルの受信後に文字列「Done」を出力します。

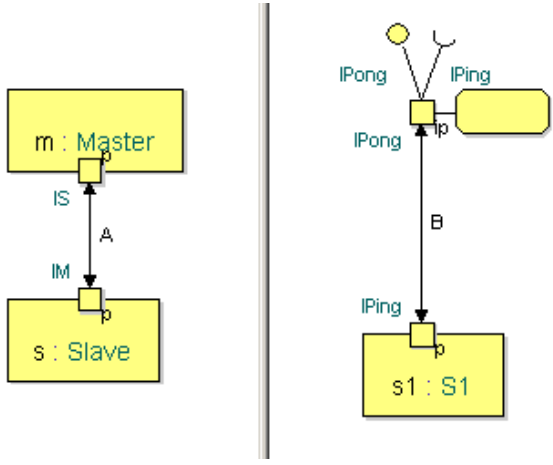


図 223: 合成構造図でのデプロイメント

スレーブ クラスには、状態機械とタイプ S1 のコンポーネント s1 の両方が含まれます。スレーブ クラスはマスター クラスから要求を受信すると、パート s1 とのシグナリング交換シーケンスを開始します。シグナリング シーケンスは、スレーブ クラスがマスター クラスに応答するまで、100,000 回繰り返されます。この例では合計 200,000 のシグナルが送信されます。

792 ページの図 224 は、代替表記で状況を示しています（四角はアクティブ クラス、丸は状態機械、線はシグナル フローを表します）。

注記

マスター 状態機械のスレーブ 状態機械とのコミュニケーション方法、また、スレーブ状態機械のパート s1 とのコミュニケーション方法に着目してください。

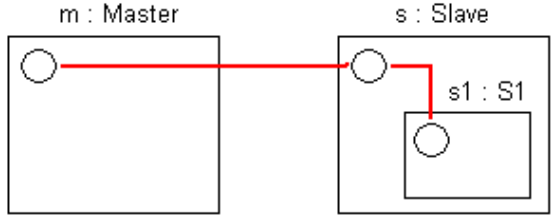


図 224: デプロイメントの代替表記

スレッドへのデプロイメント

サンプルはさまざまな方法でデプロイできます。使用できる方法は、生成するアプリケーションのタイプ（Model Verifier または AgileC コード ジェネレータアプリケーション）およびターゲット システムの種類の両方に依存します。また、アプリケーション内のさまざまなスレッドへの状態機械のマッピング方法も異なります。

このサンプルでは、以下のスレッドへのマッピング方法を使用できます。

- 1 つのスレッド：すべての状態機械を保持します。
- 2 つのスレッド：一方がパート m を、もう一方がパート s を保持します。
- 3 つのスレッド：状態機械ごとに 1 つのスレッド。

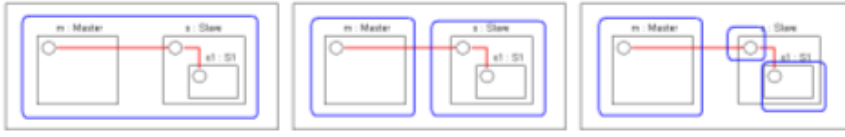


図 225: デプロイメントのマッピング

Deployment One Thread、Deployment Two Threads、Deployment Three Threads という 3 つのクラス図でこれらのケースについて説明します。

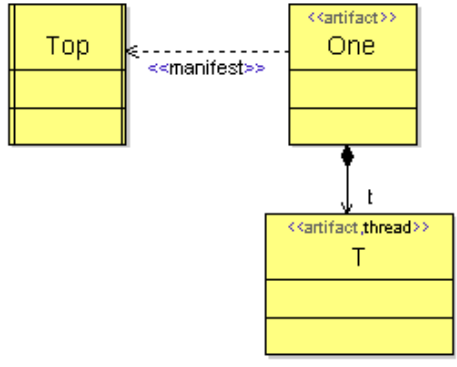


図 226: 1 つのスレッドへのデプロイメント

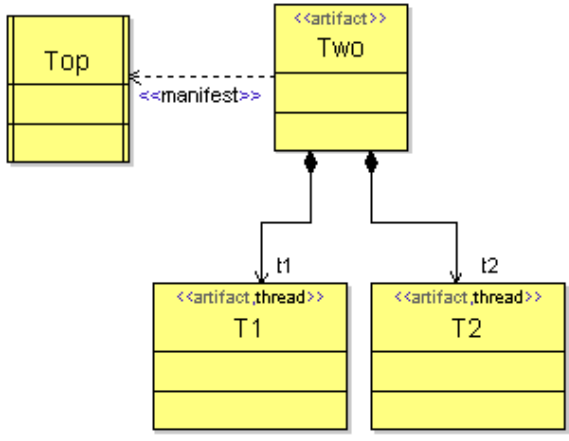


図 227: 2 つのスレッドへのデプロイメント

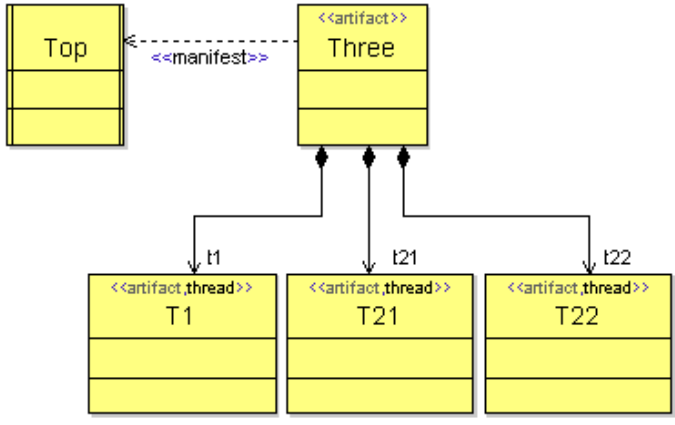


図 228: 3 つのスレッドへのデプロイメント

スレッド アーティファクト

スレッド アーティファクトは、ビルド アーティファクトの名前付き合成です。スレッド アーティファクト T1 は、2 つのスレッドと 3 つのスレッドの両方のケースに使用されます。

各スレッド アーティファクトは、スレッドが包含すべき状態機械のインスタンス名のリストを保持します。これらの名前は完全修飾された、「マニフェストされた」クラスに関連する UML インスタンス名です。この場合、以下の名前が考えられます。

- m：最上位クラスのパート m を表します。
- s：最上位クラスのパート s を表します。
- s.self：最上位クラスのパート s の状態機械を表します。
- s.p1：最上位クラスのパート s のパート p1 を表します。

このインスタンス名リストは、［ビルド設定］を使用して編集できます。たとえば、アーティファクト「T」を右クリックして［ビルド設定］を選択します。以下のようなエディタが表示されます。

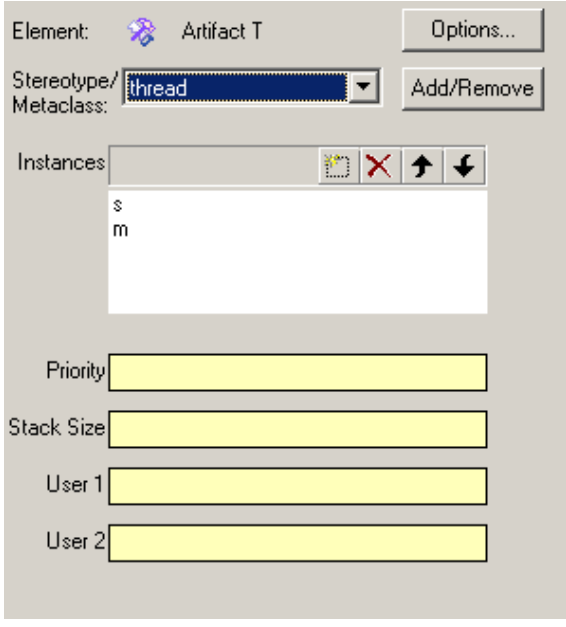


図 229: スレッド アーティファクト「T」のプロパティ

スレッドで実行されるインスタンスのリストの他に、スレッドごとに優先順位とスタック サイズを設定できます。スレッドに追加プロパティを指定するため、「User 1 と「User 2」という名前の 2 つの属性が使用できます。（このサンプルでは、これらの追加設定値は使用しません。）

サンプルにはこの他 2 つのビルド アーティファクトが用意されています。

- Single：このビルド アーティファクトには、スレッド アーティファクトがありません。ベア バージョンのビルドに使用します。この場合、振る舞いは「1 つの」スレッドの場合と同じです。
- ModelVerify：このアーティファクトは、モデル ベリファイヤ（Model Verifier）実行形式ファイルのビルドに使用します。

ビルド

これまでに説明したビルド アーティファクトは、アーティファクトを右クリックしてショートカット メニューから［ビルド］を選択するか、［ビルド］から［構成のビルド］コマンドを選択してビルドできます。

その場合、「ModelVerify」、「One、「Two」、「Three」という 4 つのビルド アーティファクトのビルドに、「Configuration ALL」というクラス図を使用できます。

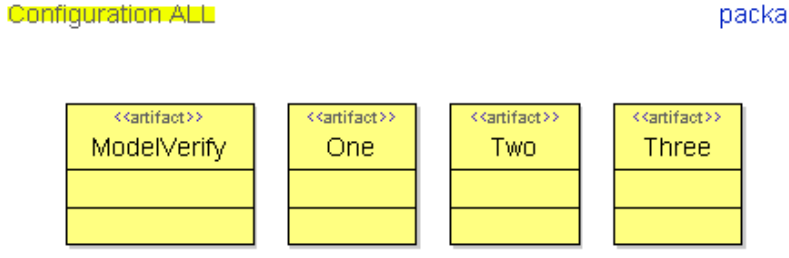


図 230: 構成図

実行

Tau フレームワークで起動できるのは、「ModelVerify」ビルド アーティファクトによって生成した実行形式ファイルだけです。その他の実行形式ファイルを起動するには、コマンド シェルを使用します。

実行パフォーマンス

3 つのスレッドのシナリオには、異なるタイミング特性があり、このため実行パフォーマンスが異なります。サンプルでは、シグナルの送信が主な点です。以下の実行回数は、800MHz Intel CPU を搭載した Windows 2000 システムで測定されたものです。

アーティファクト	One	Two	Three
回数	0.7	0.8	15.0

これらの回数の測定はラフなものですが、スレッド間のシグナル送信によるオーバヘッドの増加を示しています。

20

アプリケーション ビルド リファレンス

このドキュメントは UML モデルから C、C++、および Java アプリケーションをビルドするためのリファレンス ガイドです。

インタラクティブ ビルド インターフェイス

アプリケーション ビルダ は以下のコンポーネントで自身をマニフェストします。ビルド アーティファクト、構成、ビルド ウィザード、ビルド メニュー、およびプロパティ エディタで表示するステレオタイプと属性に含まれるビルド設定。

ビルド アーティファクト

ビルド アーティファクトは、ビルド ステレオタイプが適用されたアーティファクトです。ビルド ステレオタイプは、適用されたときに対象のアーティファクトが何のビルドアーティファクトになるのかを定義しています。1 つのアーティファクトに複数のビルド ステレオタイプが適用されている場合、そのアーティファクトでビルド操作を開始したときに、どのビルドステレオタイプが使用されるのかは、定義されていません。

ビルド アーティファクトは、特殊な図形表示を使用して、ワークスペース ウィンドウに表示されます。これにより、ビルド アーティファクトとその関数（<<Icon>> ステレオタイプの属性で定義）を簡単に関連付けることができます。これは、ビルドアーティファクトを区別するのに便利です。下図に、モデル ビューで表示されているビルド アーティファクトのアイコンと対応するビルド ステレオタイプ上の <<Icon>> ステレオタイプとの関係を示します。

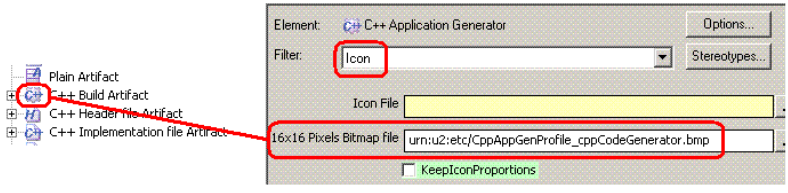


図 231: Artifact Icons

ビルド ステレオタイプ

ビルド ステレオタイプは、一連のビルド操作とビルド設定を定義しています。ビルド設定はビルド ステレオタイプの属性で、ビルド ステレオタイプと関連付けられたすべてのツール、つまりコードジェネレータやランタイムシステムなどの設定を現します。ビルド操作は、ビルド アーティファクトで起動できるさまざまなアクションを指定します。ビルド操作は、ビルド ショートカット メニューに表示されます。

すべてのビルド ステレオタイプは（抽象）ジェネリック ビルド ステレオタイプ <<build>> を継承します。

注記

ジェネリック ビルド ステレオタイプ <<build>> は、表示されるステレオタイプの数を減らすために通常は隠されています。

ビルド操作

ビルド操作には、「動的」と「静的」の 2 種類があります。

動的なビルド操作は、ビルド ステレオタイプの操作を使用して定義されています。つまり、ビルド ステレオタイプが操作の定義を「所有」しています。操作の名前は［ビルド］ショートカットメニューで使用されます。この仕組みによって、静的なビルド操作で必須の「生成」、「ビルド」、「Make」だけではない、任意の名前の操作を使用できます。

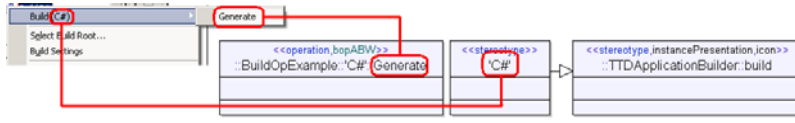


図 232: ビルド操作

上図はビルド アーティファクト ‘C#’ とビルド操作 ‘Generate’ を示しています。また、ビルド メニューでこれらの名前がどのように使用されているかも示しています。

動的ビルド操作のアクションを定義するためには、<<bopAgent>> と <<bopABW>> の 2 つのステレオタイプを使用します。2 つのいずれかが操作の定義に適用される必要があります。どちらも適用されない場合は、操作はビルド操作とは見なされません。両方が適用された場合の振る舞いは定義されていません。

<<bopAgent>> は、ビルド操作のアクションが、Tau のアドレス空間内で起動するエージェントを使用して実行されることを指定します。<<bopABW>> は、ビルド操作のアクションが、外部実行形式ファイルを使用して、そのプロセス内で実行されることを指定します。これはコード生成の間に Tau の実行がブロックされないこと、およびコード生成が［ビルドの停止］ボタンで中断できることを意味します。

<<bopABW>> を使用して指定した実行形式が "ABWGen" である場合、<<bopAgent>> の場合と同じようにエージェントが使用されますが、実行は外部プロセスで行われます。

静的ビルド操作は、静的な操作 "check"、"generate"、"build"、"make"、"clean"、"launch"、"update" を記述するために使用されます。

注記

静的ビルド操作は将来のサポートの保証がないため使用を推奨しません。代わりに、より柔軟な動的ビルド操作を使用することを推奨します。静的ビルド操作の説明は、製品の機能性を完全に記述するという観点からのみ行っています。

参照

ABWGen

構成

Tau プロジェクトには、少なくとも構成が 1 つ含まれ、このうちの 1 つは必ずアクティブ構成です。現在のプロジェクトと現在のプロジェクトのアクティブ設定は、プロジェクト ツール バーに表示されます。

1 つの構成は、任意の数のビルド アーティファクトをグループ化します。構成のビルドを行う際、構成に含まれるすべてのビルド アーティファクトが擬似並列的にビルドされます。ビルド順序については、複数ビルド アーティファクトの構成 で説明しています。

ビルド ルート

ビルド ルートで、ビルドするモデルのサブセットとその範囲を指定します。ビルド ルートの使用方法については、ユーザー ガイドを参照してください。

参照

ビルド ルートの使用

ビルド タイプ

ビルド タイプはビルド アーティファクトによって使用されるコード ジェネレータを指定します。使用できるビルド タイプは以下のとおりです。

- Model Verifier
- C Code Generator
- AgileC Code Generator
- C++ Application Generator
- Java
- Make settings
- Makefile generator

1 つのビルド アーティファクトに追加できるビルド タイプは 1 つだけです。

上記のビルド タイプは、すべて汎用ビルド タイプ **build** から継承されます。このビルド タイプは特定のコード ジェネレータを定義していないため、実際のビルドには使用できません。

参照

ビルド タイプの使用

ビルド設定

ビルド設定でモデルのビルド方法を制御します。ビルド タイプごとに個別のビルド設定があります。これらのビルド設定は、ビルド ステレオタイプおよびプロパティとして実装されます。ビルド アーティファクトごとに、複数のステレオタイプを適用できます。

以下は、サポートされるビルド タイプで利用可能なビルド設定の要約です。

Model Verifier の設定

このビルド タイプでは、以下の設定を指定できます。

- ターゲット ディレクトリの指定
- 環境とのシグナル送受信のサポート
- ホスト プラットフォームの指定
- 使用する C/C++ コンパイラの選択
- make テンプレート ファイルの提供
- C++ モードでコンパイルするオプション
- アプリケーションまたはライブラリとして生成コードのコンパイルを指定
- 警告の冗長表示または非表示などの便宜設定

C コード ジェネレータの設定

このビルド タイプでは、Model Verifier と同じ設定と以下の設定を指定できます。

- スレッド アプリケーションを生成するオプション
- C 言語で生成された名前の接頭辞付加
- C Code Generator の定義済みライブラリの代わりに、ユーザー定義（カスタム）ランタイム ライブラリを指定。

AgileC コード ジェネレータの設定

このビルド タイプでは、モデル ベリファイヤ（Model Verifier）と C コード ジェネレータの設定を指定できます。さらに以下の設定も行うことができます。

- 生成コードの名前付けの詳細設定
- プロセス、シグナル、タイマーのハンドリングの設定
- 動的メモリ管理などのランタイム パフォーマンスに影響を及ぼす設定
- 生成コードのランタイム エラー検出

C++ アプリケーション ジェネレータの設定

このビルド タイプでは、C++ Application Generator の設定を指定できます。以下のような指定の例があります。

- ターゲット ディレクトリの指定
- 名前の接頭辞の設定
- 生成コードのレイアウトと編成のオプション
- Tau Object Run Time (TOR) ライブラリとのリンク方法
- タイマを使用する生成アプリケーションで使用する時間単位

- Tau で生成 C++ アプリケーションをデバッグする際に使用するポート番号とホスト名
- 生成ファイルに変更を加えた場合に UML モデルにラウンドトリップバックするかどうかの指定
- コード生成とモデル更新を自動で行うかどうかの指定

詳細は翻訳オプションを参照してください。

Java コードジェネレータの設定

このビルドタイプでは、Java コードジェネレータの設定を指定できます。S 以下のような指定の例があります。

- ターゲット ディレクトリの指定
- 使用する Java バージョン
- コード生成とモデル更新を自動で行うかどうかの指定

詳細は、Java ビルドアーティファクト設定を参照してください。

make の設定

このビルド タイプでは以下の設定を指定できます。

- 「make」呼び出し時に使用するプラットフォーム固有の規則の指定
- 入力として提示する make ファイルの指定
- make に対するユーザー定義オプションの指定

Makefile ジェネレータの設定

このビルド タイプでは以下の設定を指定できます。

- 生成された make ファイルの保管場所の指定
- 使用する make の方言の指定
- 生成された make ファイルに追加するユーザー定義セクションの指定

参照

Model Verifier

AgileC Code Generator

Make settings

Makefile generator

ビルド ウィザード

以下の場合、ビルド ウィザードにより、ビルドに必要な情報を入力するよう要求されます。

- 不完全な設定のビルド アーティファクトでビルドが開始された場合。この場合、ビルド ルートおよび／またはビルド タイプは予めビルド アーティファクトの値に設定されています。ウィザードを使用して定義した値が、ビルド アーティファクトに保存されます。
- ビルド アーティファクトではなく、関連付けられたビルド アーティファクトもないモデル要素でビルドが開始された場合。この場合ビルド ルートが挿入されます。
- 構成ビルドの開始時、この構成内のビルド アーティファクトが空の場合。

ビルド ウィザードで指定するプロパティ

ビルド ウィザードにより、以下のプロパティを指定するよう要求されます。

- **マニフェスト**

  このフィールドにはビルドするビルド ルートの完全修飾名が表示されます。[マニフェスト] フィールドで、ビルド アーティファクトとビルド ルートとして使用する要素（クラスまたはパッケージなど）との関係を指定します。
- **設定**

  [設定] ボタンを押して、現在のプロジェクト モデルに従って、クラス階層ツリーからビルド ルートを指定できます。
- **ビルド タイプ**

  [ビルドタイプ] を選択して、ビルド タイプを指定します。選択できるビルド タイプは、プロジェクトで有効なビルド タイプとビルド タイプの列挙型（ビルド アーティファクトのプロパティとして設定される）のリテラル値によって異なります。
- **アーティファクトをアクティブな構成に追加します。**（オプション）

  このオプションを有効にすると、現在アクティブな構成にアーティファクト（ウィザード終了後に作成される）を挿入できます。714 ページの「複数ビルド アーティファクトの構成」を参照してください。
- **プロパティ**（オプション）
  - このボタンをクリックすると、ビルド アーティファクト名のようにビルド ウィザードで表示できないプロパティを変更できるプロパティ エディタが表示されます。

ファイル アーティファクト

ファイル アーティファクトはファイル ステレオタイプが適用されたアーティファクトです。すべてのファイル ステレオタイプは汎用ステレオタイプ << file >> を継承します。

このアーティファクトは、モデル要素をどのようにファイルに実装するかや、モデル要素をどのようにファイルに「接続」するかを指定するために使用します。また、このアーティファクトは、UML ソースと実行形式ファイルまたはライブラリとの間の依存関係をモデル化して、正しいビルドの依存関係を確立するためにも使用します。

アーティファクトにファイルステレオタイプを追加することによって、アーティファクトは特殊化されて、抽象モデルを具体的な実装に変換する際（つまりアプリケーションのビルドの際）に特定の目的を与えられます。

C++ アプリケーション ジェネレータでアプリケーションをビルドする際、以下のファイル アーティファクトを使用できます。

- cppHeaderfile
- cppImplementationfile
- executable
- library
  - dynamicLibrary
  - staticLibrary
- makefile

Java コードジェネレータを使用したアプリケーションビルドを行うときには、以下のファイルアーティファクトが利用できます。

- javaFile
- jarFile

注記

現バージョンの Tau でアプリケーションをビルドする際、ファイル アーティファクトを使用できるのは、C++ アプリケーション ジェネレータと Java コードジェネレータのみです。モデル要素とファイル間の接続は、AgileC コード ジェネレータ、C コード ジェネレータ、およびモデル ベリファイヤ（Model Verifier）で生成されたアプリケーションのファイル アーティファクトを使用して指定することはできません。

参照

ファイル アーティファクトの使用

スレッド アーティファクト

スレッド アーティファクトは、ステレオタイプ << thread >> が適用されたアーティファクトです。スレッド アーティファクトは、実行形式アプリケーション内のスレッドを現すステレオタイプ化されたクラスです。このようなクラスは、クラス図エディタを使用してモデリングします。

スレッドアーティファクトは、AgileC コード ジェネレータ と C コード ジェネレータでのみ使用できます。

C++ アプリケーション ジェネレータと Java コードジェネレータについては、スレッドは TOR ライブラリを使用してプログラム的に定義され操作されます。

参照

スレッド アーティファクトの使用

プロジェクト ツール バー

プロジェクト ツール バーは、アクティブ プロジェクト、アクティブな環境設定、アクティブ ツール、およびクイック ボタンのあるビルド ツール バーで構成されます。

アクティブ プロジェクト

このボックスで、現在ロードされているワークスペースにあるプロジェクト間でアクティブ プロジェクトを切り替えられます。

アクティブな環境設定

このボックスで、アクティブな構成に切り替えられます。選択できる項目には、アクティブ プロジェクトで定義された環境設定があります。アクティブな環境設定の切り替えは、1 つのモデルからの複数のビルドを管理するために便利な機能です。

アクティブ ツール

このコンボ ボックスで、アプリケーションのビルド時に使用する「ツール」を指定します。現バージョンでは、このボックスから選択できるのは **Application Builder** のみです。

ビルド ツール バー

ビルド ツール バーには、構成のビルドで頻繁に使用するコマンドにアクセスするためのクイック ボタンがあります。

コマンド	ショートカット
構成の更新	Shift + F6
構成の生成	F6
構成のビルド	F7
停止	Ctrl + Scroll Lock
構成の実行	F5

参照

プロジェクトの操作については、プロジェクトの操作を参照してください。

アプリケーション ビルド時の構成の使用については、構成を使用したビルドを参照してください。

これらのコマンドについては、ビルド メニューを参照してください。

ビルド メニュー

ビルド メニューには、構成ビルドでサポートされるすべてのビルド コマンドが表示されます。

- 最も利用頻度の高いコマンドは、プロジェクト ツール バーのクイックボタンで実行されます。
- ワークスペース ウィンドウのビルド アーティファクトを右クリックすると、アーティファクトのビルドでサポートされるコマンドがショートカット メニューに表示されます。

ビルド操作にはいくつかの決められたステップがあります。

1. ビルドとチェックの準備。このステップは必ず実行します（停止コマンドを除く）。ここでは構成とビルド アーティファクトが有効かどうか、モデルが正しく、このビルドに適しているかをチェックします。
2. **生成**：このステップでは、モデルからのコードが生成され、C/C++ コード生成のためには、make ファイルも生成されます。
3. **ビルド**：生成のステップで make ファイルが作成されていれば、このステップでは ”make” を実行します。
4. **起動**：このステップでは、前のステップで実行形式が生成されていれば、その実行形式ファイルを実行します。

これらのステップは、この順序どおり、構成内のビルド アーティファクトごとに繰り返し実行します。最後に実行するステップは、どのビルド操作を実行するか、どの「程度」進めるか、状況によって異なります。

停止

このコマンドは、実行中のすべてのビルド操作を停止し、現在アクティブのモデル ベリファイヤ（Model Verifier）セッション（ある場合）を終了します。

構成のチェック

このコマンドは、構成のセマンティック チェックを実行します。セマンティック チェッカは、構成に含まれるビルド アーティファクトで定義されたビルド タイプに従って動作します。

［構成のチェック］は通常の［チェック］コマンドと異なり、ビルド タイプが定義するコード ジェネレータやターゲット言語の制約事項が生む規則も適用して、モデルをチェックします。このコマンドの目的は、ビルド作業のできるだけ早期において、上に述べた制約に起因するような問題を検出することです。

注記

コード ジェネレータ固有の制約事項から発生する UML の制約の多くは、セマンティック チェッカで見つけることができます。ただし、このような制約の中には、コード生成して初めて見つかるものもあります。制限事項に関する詳細な情報については、816 ページの「C アプリケーション ビルド時の UML サポートの制限事項」および 822 ページの「C++ アプリケーション ビルド時の UML サポートの制限事項」を参照してください。

構成の生成

このコマンドは、まず構成がビルドに適しているかチェックします（構成のチェックを参照）。

次に、ビルド アーティファクトのビルド ルートがマニフェストしているモデル部分がが、セマンティック的に正しく、ビルド アーティファクトで定義されたコード ジェネレータでサポートされている場合には、ビルド アーティファクトで定義された設定に従ってコードが生成されます。C と C++ については、make ファイルも生成されます。

構成のビルド

このコマンドは、まず構成がビルドに適しているかどうかチェックし、次にコードと make ファイルを生成します（構成のチェックおよび構成の生成を参照）。

コードと make ファイルが生成されたら、生成されたコードはビルド アーティファクトで定義された make 設定に従ってコンパイルおよびリンクされます。

構成の実行

このコマンドは、まず構成のチェック、生成、ビルドを行います（構成のチェック、構成の生成、および構成のビルドを参照）。ビルドに成功したら、生成されたアプリケーションが起動します。

注記

この機能は、モデル ベリファイヤ（Model Verifier）アプリケーションでのみ利用可能です。

構成の更新

このコマンドは、ファイル アーティファクトの使用によってマニフェストされたソース ファイルで、構成内のビルド アーティファクトにあるビルド ルートで定義されたモデルの一部を更新します。このソースに加えられた変更はモデルに反映されます。

注記

構成の更新は、AgileC コード ジェネレータ、C コード ジェネレータ 、Model Verifier ではサポートされません。

構成のクリーン

このコマンドは、make ファイルの make clean 操作を実行します。この操作で、前回のビルドで作成されたオブジェクト ファイル、ライブラリおよび実行形式ファイルを削除します。

モデルベリファイヤの起動

このコマンドは、ビルドせずにモデル ベリファイヤ（Model Verifier）アプリケーションを起動します。コマンドに続けて、モデル ベリファイヤ（Model Verifier）の実行形式ファイルを含むファイルを指定するよう要求されます。

注記

ワークスペースに現在ロードされているモデルからアプリケーションが作成されたかベリファイするためのチェックは行われません。そうではない場合、またはビルド後にモデルが変更された場合は、モデル ベリファイヤ（Model Verifier）に不正確あるいは不十分な情報が表示されます。

ビルド ショートカット メニュー

ビルド ショートカット メニュー（ビルド アーティファクト、または、ビルド ルートとしての使用に適したモデル要素を右クリックすると表示される）には、1 つまたは 2 つの表示形態があります。

ビルド アーティファクトのビルド ショートカット メニュー

ビルド アーティファクトから起動すると、ビルド コンテキスト メニューは以下のように表示されます。

［ビルド ([Build type])］が表示され、次に利用可能なビルド コマンドのサブメニューが表示されます。これらのビルド コマンドはビルド メニューのコマンド（［構成のチェック］、［構成のビルド］、［構成の更新］、［構成のクリーン］、［構成の実行］）と同じです。

モデル要素の ビルド ショートカット メニュー

ビルド ルートとしての使用に適したモデル要素（パッケージまたはクラス）から起動すると、ビルド ショートカット メニューは以下のように表示されます。

- プロジェクトで現在有効になっているすべてのビルド タイプのリスト。
- これらのビルド タイプはそれぞれサブメニューを持ちます。このサブメニューには、アクティブな構成内のビルド タイプと合致するすべてのビルド アーティファクトが含まれます。
- 新しいビルド アーティファクトを作成するコマンド。

バッチ ビルド インターフェイス

taubatch コマンドを使用して Tau を起動し、コマンドライン プロンプトからアプリケーションをビルドできます。

```
taubatch [ オプション ]
```

入力

taubatch への入力はコマンドのオプションとして指定され、以下の組み合わせで構成されます。

- プロジェクト ファイル（.ttp ファイル）。-p “project file” オプションを使用します。
- 構成。-c “Configuration Name” オプションを使用します。
- ビルド アーティファクト。`-g GUID` または、`-o element` オプションを使用します。

出力

ビルド インデックス ファイル

taubatch は、ターゲット ディレクトリにビルド インデックス ファイルを生成します。このディレクトリには前回このビルド タイプが利用された際に生成されたファイルのリストが含まれます。ビルドが何らかの理由で失敗した場合、このファイルは空になります。

ビルド インデックス ファイルの名前は以下のようになります。

```
build_index_" <guid-of-artifact> ".xml
```

この中で、"0-1, @-Z, a-z, _" の組み合わせ以外のトークンをすべて "_" に置き換えるため、アーティファクトの GUID（グローバル一意識別子）が壊されます。

ビルド インデックス ファイルは、以下のビルド インデックス ファイル DTD に従い、情報を XML で表現します。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE BI [
  <!ELEMENT list (file)+ >
  <!ELEMENT file EMPTY>
    <!ATTLIST file name CDATA #REQUIRED>
    <!ATTLIST file time INT #REQUIRED>
    <!ATTLIST file type CDATA #IMPLIED>
]>
```

リストの各エントリは生成されたファイルを記述し、以下の属性を含みます。

属性	説明
Name	ファイルの名前
Type	ファイルのタイプ（モデル ベリファイヤ（Model Verifier）実行形式ファイルの「デバッガ」など）
Time	ファイルが生成された時間。秒単位で、1970-01-01:00:00:00 から数えます。

コード ジェネレータの出力

コード ジェネレータからの出力は、C/C++/Java ソースファイル、make ファイル、および他のさまざまなファイルとレポートで構成されます。

参照

第 27 章「C コード ジェネレータ リファレンス」
第 31 章「AgileC コード ジェネレータ リファレンス」
第 41 章「C++ アプリケーション ジェネレータリファレンス」
第 34 章「Java コードジェネレータ リファレンス」

オプション

`taubatch` で認識されるオプションは以下のとおりです。

-B

`Build` の短縮形

このオプションは、セマンティック チェック、コードと make ファイルの生成、make ファイルの make コマンドの実行順にビルドするよう指定します。

このオプションは、-C、-G、および -S のオプションと組み合わせて使用することはできません。

-c “Configuration Name”

`--config “Configuration Name”` の短縮形

このオプションは、`Configuration Name` で指定された構成をビルドの入力として使用するように指定します。このオプションを指定すると、構成に含まれるすべてのビルド アーティファクトがビルドされます。

-C

`--Clean` の短縮形

このオプションは、make ユーティリティで生成されたファイルの削除を実行します。通常、生成されたオブジェクト ファイル、ライブラリおよび実行形式ファイルを削除します。どのフレーバの make、make ファイルを使用するかは、ビルドで使用するビルド アーティファクトで定義します。

このオプションは、-B、-G、および -S のオプションと組み合わせて使用することはできません。

-g GUID

`--guid GUID` の短縮形

このオプションでは、GUID（グローバル一意識別子）で参照されるビルド アーティファクトをビルドの入力として使用します。

一般に GUID を入手するには、ビルド アーティファクトが格納されている `.u2` ファイルを検索します。（ビルド ルートのような）ビルドの基本要素の GUID は生成後の目次で見つけることができます（オプション「-T」を参照）。

-G

`--Generate` の短縮形

このオプションでは、ビルドの入力で定義されたビルド ルートによってマニフェストされるモデルのサブセットのコードと make ファイル（C/C++ の場合）をチェックして生成します。

このオプションは、-B、-C、および -S のオプションと組み合わせて使用することはできません。

-h

`--help` の短縮形

このオプションは、`stdout` で使用可能なオプションのヘルプ ファイルを表示し、`taubatch` を終了します。

-l (小文字の L)

`--listbt` の短縮形

このオプションは、`stdout` のプロジェクト ファイルで使用可能なビルド タイプを表示し、`taubatch` を終了します。

-o element

`--object element` の短縮形

`element` は `taubatch` の入力として提示するビルド アーティファクトの修飾名です。

このオプションは -g GUID および -c “Configuration Name” オプションと相互排他的です。

-p “project file”

--project “project file” の短縮形

"project file" は、ビルドの入力として提示するプロジェクト ファイル（.ttp ファイル）の名前です。

-r

--rebind-by-name の短縮形

このオプションは、モデルの読み込み時に、ツールが完全名前解決を使用するよう設定します。通常、このオプションは必要ありません。この設定でビルドに時間がかかる可能性があります。

-S

--Semantic-check の短縮形

このオプションは、入力のセマンティックの正確性とビルドへの適性のチェックのみ実行するようビルドを設定します。

このオプションは、-B、-C、および -G のオプションと組み合わせて使用することはできません。

-T

--TOC の短縮形

このオプションは、プロジェクト ファイルの目次を含むファイルを生成します。このファイルには以下の情報が格納されます。

- インデントされたリストの名前空間の構造（パッケージおよびクラス）
- 使用可能なビルド アーティファクト
- 使用可能な構成

-V

--Version の短縮形

このオプションは、stdout 時に taubatch のバージョンを表示し、終了します。

taubatch の使用例

例 **332: -T** オプションの使用例

```
$ taubatch -p PingPong.ttp -T
IBM Rational Tau Application Builder

TAB1026:Table of contents
DJhxxINnrzALDYVY-Le5U27I :pingpong_artifact
L0SZSIZK9BlLsxk94E9IS*dE :PingPong
b7xI3LmvPyFLuTiXGEwHCW1I :Match
w2rrRLYFz60L-H2dZL99rzgV :Player2
bKPz0VVWI2FLMsNWILWqtuoI :Player1
7MNyDEJTh40Ld5wkyIy1RczL :Agile

Configurations:
  ALL
  Default
  MyConfig
```

例 **333:** 構成とビルド アーティファクトのビルド

構成 “MyConfig” をビルドする例 :

```
taubatch -p PingPong.ttp -c MyConfig
```

ビルド アーティファクト “Agile” をビルドする例 :

```
taubatch -p PingPong.ttp -o Agile
```

MyConfig と Agile は、どちらもプロジェクト ファイル `PingPong.ttp` で定義します。

C アプリケーション ビルド時の UML サポートの制限事項

このセクションでは、C 言語ベースのビルド タイプを使用してアプリケーションをビルドする際の UML サポートの制限について説明します。

- モデル ベリファイヤ（Model Verifier）
- C コード ジェネレータ
- AgileC コード ジェネレータ

C ビルド タイプの制限事項

アクティブ コード ジェネレータ

効率的にコードを生成するためには、各プロジェクト内のアクティブなコード ジェネレータの数を制限することを推奨します。たとえば、プロジェクト内で C++ アプリケーション ジェネレータ と Model Verifier コードジェネレータの両方がアクティブな場合、コード生成を終了するまでに長い時間が必要になり、多くのメモリ資源を消費します。

すべての C ビルド タイプ

以下の UML の構成要素、または UML の使用は、どの C ビルド タイプ（Model Verifier、C コード ジェネレータ、および AgileC コード ジェネレータ）でもサポートされません。

クラス

- 生成された C コードの比較演算子 ('==' および '!=') は外部クラスのインスタンスには機能しないことがあります。これにより、コンパイル エラーが発生することがあります。このような演算子もある条件下では使用できます。たとえば、C++ クラスのユーザー定義の独立した演算子 '==' をヘッダー ファイルに入れた場合、このヘッダーファイルを CApplication ステレオタイプを使用してモデルに挿入すると、この演算子はインポート後の C++ クラスのインスタンスでも機能します。
- 内部構造をもつアクティブ クラスのインスタンスの動的作成。
- スタンドアロン アクションでのパッシブ クラスの作成は許可されません。
- ステート付き操作はパッシブ クラスに含むことができません。
- パッシブ クラスはインターフェイスを実行できません。
- パッケージが C または C++ 言語で外部定義されていなければ、パッケージの Non const 属性は無視されます。
- パッシブ クラスのメソッドで、アクティブ クラスのメソッドを呼び出せません。
- パッシブ属性では多重度制約はサポートされません。

例 **334:** パッシブ属性の多重度制約

```
class A { }
part A [1..10] mypart; // constraint [1..10] is ignored
```

コンストラクタとデストラクタ

- コンストラクタには実行メソッドが必要です。
- 静的コンストラクタとデストラクタは許可されません。
- コンストラクタの状態機械は 1 つのクラスに 1 つだけ表示されます。
- パッシブ クラス内のコンストラクタから継承されたコンストラクタを呼び出すことはできません。

デストラクタは部分的にサポートされます。操作は以下のとおりです。

- 必須クリーンアップ後、状態機械を停止するシグナルを受け取れるように、アクティブ クラスを実装する必要があります。
- パッシブ クラスでは、デストラクタは他の操作を呼び出すことができません。

例 **335:** デストラクタは他の操作を呼び出すことがでない

```
class B
{
  public void Foo() { }
}
class A
{
    B refB;
    ~A()
    {
      refB.Foo(); // Not supported, error TCC0924
      delete refB;
    }
}
```

ポート

- パッシブ クラスではポートは許可されません。
- ポートは再定義できません。

操作

- 修飾子を使用した superclass 操作の呼び出しはサポートされません。
- グローバル操作からアクティブ クラスの操作を呼び出すことはサポートされません。
- 値を戻す操作の戻り値は、代入式の左辺で処理する必要があります。
- delete 操作はパートには適用できません。
- 確定した操作の再定義は許可されません。
- 静的操作から非静的属性にアクセスできません。

シグナル

- 受信シグナルのパラメータは省略できません。
- "all" からのシグナル送信はサポートされません。
- オプション パラメータはサポートされません。シグナル パラメータの多重度 [`0..1`] は C コード生成時に無視されます。

演算子

- Own および Oref テンプレート：
  Own および Oref テンプレートに対する C コード ジェネレータのサポートは限られています。これらのテンプレートは使用しないことを推奨します。
- ビット型：
  関係演算子 <、>、<=、>= は、C コード ジェネレータではサポートされません。
- 8 ビット型の操作 “Octet '[]' (Octet, Integer, Bit)”：
  8 ビット型のビットに新しい値を割り当てることは、C コード ジェネレータではサポートされません。
- 8 ビット型の操作 “Octet '¥=' (Octet, BitString)”：
  BitString 値から 8 ビットを割り当てることは、C コード ジェネレータではサポートされません。
- べき乗演算子はコード ジェネレータでサポートされていないため、使用する場合は、外部で実装する必要があります。

タイマー

- タイマーはパッケージで定義できません。タイマーは、パッケージではなく、使用されるアクティブ クラスで定義します。
- タイマーの `active` 操作は、パラメータを持つタイマーについてはサポートされません。テストできるのはパラメータを持たないタイマーだけです。

アクション

- Try アクションはサポートされません。
- Join、Return、Stop アクションは内部ループで許可されません。
- Start Transition には terminating アクションが必要です。
- シグナル "none" の受信はサポートされません。
- Deep History Nextstate アクションはサポートされません。

属性

- 静的属性はサポートされません。C 言語と生成された C コードのグローバル データに対応する属性は、リアルタイム アプリケーションで安全に変数にアクセスすることをサポートしません。
  - アプリケーションにグローバル データが必要な場合、グローバル データの読み込みと書き込みを行うシグナル インターフェイスを通じて、すべてのアクティブ クラスからアクセスできるアクティブ クラスにグローバル データを実装します。
  - 代わりに、静的属性を静的 C 変数として実装する方法もあります。

- 親クラスから（仮想）操作を継承するためには以下の条件が必要です。
  - 操作属性は、（確定された）子クラスで再度属性宣言する必要があります。
  - しかし、この属性で、親と子の操作に同じ名前を使用できません。
- 整数型と列挙型に互換性はありません。このような型の属性を、式や演算子で一緒に使うことはできません。

その他

- STL サポートは現在 C++ アプリケーション ジェネレータでの作業に限られています。STL は C コード ジェネレータでは使用できません。インポートされた C++ コードがテンプレートを使用する場合は、C コード ジェネレータでは使用できません。
- C/C++ インポートで説明されているように、関数ポインタは C コード ジェネレータではサポートされません。
- C コード生成時に範囲チェック式 はサポートされません。
- モデル ベリファイヤ（Model Verifier）の 2.2 から 2.3 への移行について、特定のデータ型について名付け規則が変更されました。この結果、モデルが同じリテラルの二つ以上の列挙型を含む場合に `.ttdcfg` ファイルが機能しない可能性があります。
- リテラル値やリテラル値のみを含む式を比較演算子の引数として使用する場合、解析エラーが生じます。
- `noScope` ステレオタイプは、ブラウザのグループ化の問題を解決するために導入されました。コード生成の対象となるモデルについては推奨されません。

例 **336:** サポートされない式 ———————————————————————————

分岐などで利用された場合、以下のような構成要素を使用しても、正しく実行されません。

```
2+2==4
0!=1
```

- 複数の継承はサポートされません。
- 選択したインスタンスの動的作成は許可されません。
- モデルには少なくとも 1 つのアクティブ クラスが必要です。そうしないと、アプリケーションに振る舞いがなくなります（アプリケーションのビルドは可能ですが、実行されません）。
- ユーザー定義テンプレートはサポートされません。
- ステート式はサポートされません。
- C コード生成を意図したモデルは、アクター、シーケンス図、およびユースケースに依存してはなりません。これらの UML 言語の構成要素は非形式的なので、C コード生成ではサポートされません。

- アプリケーションを使用してプロファイル パッケージ（アドイン）をロードしたり、`u2::LoadLibrary` を使用してツールに依存性を計算させると、C コード ジェネレータはパッケージを無視することになります。
  C コード生成の対象であるプロファイル パッケージは、以下のような Tcl コマンドを使用してロードされる必要があります。
  ```
  u2::LoadFile <model> <profile pacakge> false<=loadAsProfile>
  ```
  このコマンドは、プロファイル パッケージをモデル内に導入するファイルとしてロードします。その後ユーザーは、そのパッケージプロファイルへの必要な依存性を設定できます（パッケージのコード生成を強制します）。

AgileC コード ジェネレータ

すべての C ビルド タイプに適用される制限のほか、以下のような UML 構成要素、または、UML の使用は、AgileC コード ジェネレータを使用するビルド タイプではサポートされません。

クラス

- 内部構造のあるアクティブクラスが持てる多重度は 1 つ（`[1]`）のみです。
- アクティブ クラスの無限数のインスタンス宣言。
  - この操作はサポートされますが、無限数インスタンスの実行時のパフォーマンス上の理由から推奨できません。

パート

- 「新しい < アクティブ クラス名 >」以外の構成要素をパートに追加することはできません。以下はサポートされません。

```
active class A { }
part A [*] mypart;
A tmp;
tmp = new A();
mypart.append(tmp); // not possible
```

操作

- ステート付き操作はサポートされません。
- 操作からの仮想操作または再定義操作の呼び出しはサポートされません。
- 操作からの他アクティブ クラス内の操作の呼び出しはサポートされません。

タイマー

- 整数型以外のパラメータ型を持つタイマー
- 複数のパラメータを持つタイマー
- タイマー持続時間は実数値で設定できません。タイマー持続時間は type Duration の式で設定します。

その他

- “any” 分岐
- インフォーマル分岐
- イベント式のないガード条件つき遷移

- パート インデックスはサポートされません。

予約語

すべての C ビルド タイプ

これらの制限はすべての C ビルド タイプ（モデル ベリファイヤ（Model Verifier）、C コード ジェネレータ、および AgileC コード ジェネレータ）に適用されます。

モデルから C アプリケーションをビルドする場合、以下の語は UML モデルのアイテムの名前（クラスや属性の名前など）として使用することは**できません**。UML の観点から正しいモデルであっても、コード ジェネレータで許可されません。

```
break, choice, optional, remote
```

モデル ベリファイヤの実行

C コードジェネレータは以下にあげた用語を使うことを許可します。したがってこれらの用語を使用してアプリケーションを作成できます。しかし、モデル ベリファイヤのシーケンス図トレースは、これらの用語を含むエンティティに遭遇するとエラーを報告する可能性があり、そのような場合には、トレースが期待される結果を表示しない可能性もあります。以下にあげた用語以外に、ハイフン '-' を含む用語でも同じようなエラーが起こる可能性があります。

```
action, all, alt, as, before, begin, block, by, comment,
concurrent, condition, connect, create, decomposed,
empty, end, endconcurrent, endmsc, endexpr, env, exc,
expr, external, found, from, gate, in, inf, inline, inst,
instance, loop, lost, msc, mscdocument, msg, opt, order,
par, process, reference, related, reset, service, seq,
set, shared, stop, subst, system, text, timeout, tim, to,
via
```

注記

この制限は英語の大文字小文字の違いには左右されずに適用されます。モデル ベリファイヤシーケンス図トレースは、大文字小文字を区別します。たとえば、System と SYSTEM はともに予約語です。

C++ アプリケーション ビルド時の UML サポートの制限事項

このセクションでは、C++ アプリケーション ジェネレータを使用する際の UML サポートの制限について説明します。

以下の制限が適用されます。サポートされる操作の詳細については、一般的な翻訳ルールおよび C++ テキスト構文を参照してください。

アクティブコードジェネレータ

効率的にコードを生成するためには、各プロジェクト内のアクティブなコード ジェネレータの数を制限することを推奨します。たとえば、プロジェクト内で C++ アプリケーション ジェネレータ と Model Verifier コードジェネレータの両方がアクティブな場合、コード生成を終了するまでに長い時間が必要になり、多くのメモリ資源を消費します。

UML に関する制限

- 状態機械の遅延イベント（シグナルの「保存」）はサポートされません。
- データ型内の操作はサポートされません。
- UML の受信機構はサポートされません。
- C++ コード ジェネレータは、適切な前方向参照を定義すれば、循環依存をほぼ処理できます。ただし、特定のケースでは依存関係を手動で解決する必要があります。循環依存 (Class1.h が Class2.h をインクルードし、それが Class1.h をインクルードするような場合）があると、コンパイルエラーが起こり得ます。これは、以下のような前方宣言を追加することで解決できます。

```
//<USER>
    class Class1;
//</USER>
```

ラウンドリップ C++ サポートの制限

- 関数ポインタはサポートされません。
- テンプレートの特化はサポートされません。
- クラス メンバーへのポインタはサポートされません。

参照

第 40 章「C++ テキスト構文」

第 41 章「C++ アプリケーション ジェネレータリファレンス」の 1315 ページ、「一般的な翻訳ルール」

UML Java アプリケーション ビルド時の UML サポートの制限事項

このセクションでは、Java コードジェネレータを使用する際の UML サポートの制限について説明します。

以下の制限が適用されます。サポートされる操作の詳細については、Java コードジェネレータ リファレンスを参照してください。このドキュメントに説明のない UML 構築子についてはサポートされません。

アクティブコードジェネレータ

効率的にコードを生成するためには、各プロジェクト内のアクティブなコード ジェネレータの数を制限することを推奨します。たとえば、プロジェクト内で C++ アプリケーション ジェネレータ と Java コードジェネレータの両方がアクティブな場合、コード生成を終了するまでに長い時間が必要になり、多くのメモリ資源を消費します。

UML に関する制限

- 状態機械の遅延イベント（シグナルの「保存」）はサポートされません。
- データ型内の操作はサポートされません。
- UML の受信機構はサポートされません。
- ポート、コネクタはサポートされません。
- デフォルト値をもつパラメータまたはオプションパラメータはサポートされません。
- 状態なしの状態機械として実装される操作はサポートされません。
- シンタイプはサポートされません。

21

コード生成のステレオタイプ

このセクションは、コード ジェネレータとビルド アーティファクトと ステレオタイプのリファレンスです。アルファベット順に説明します。

ステレオタイプ

AgileC Code Generator

このステレオタイプは、**<<**build**>>** ステレオタイプを継承し、AgileC コード ジェネレータによる C コードの生成を制御する属性が含まれます。

AgileC コードジェネレータで使用される属性には、C コードジェネレータで使用される属性と同じ名前とセマンティックを持つものがあります。これらを含む属性の完全リストを以下に示します。このようなステレオタイプの詳細については、C コード ジェネレータステレオタイプのセクションの説明を参照してください。

Additional Preprocessor Defines

この属性は C コード ジェネレータ ステレオタイプのセクションで説明します。Additional Preprocessor Defines を参照してください。

Code generation properties

このコントロールは、AgileC コード ジェネレータで生成されたコードのプロパティを定義する属性をグループ化します。

- **Name mangling**
  - 型：{Prefix | Suffix}
  - デフォルト：Suffix

  この属性は、C コードで生成された名前の変換を制御します。UML と C コードの名前スコープは異なるため、生成後の C コードで UML の名前を直接使用することはできません。

  デフォルトで、AgileC コード ジェネレータは UML の名前に接尾辞を付加して C コードの識別子として使用し、C コードの識別子が一意であることを確実にします。長い UML 名を使用していて、同時に、C コンパイラが限られた文字数を調べて 2 つの名前が同じであるかどうかを判定する場合、名前の衝突が発生する可能性があります。このような事態を回避するには、接尾辞の代わりに接頭辞を使用します。
- **Comments**
  - 型：{Sparse | Structure | Explanation}
  - デフォルト：Sparse

  状態機械について生成される C コード内で、コードに付加するコメントのレベルを指定できます。以下のコメント レベルを使用できます。
  - **Sparse**：遷移を特定するために使用される一部のコメントだけを含みます。
  - **Structure**：Sparse に加え、変換された各 UML シンボルのコメントが追加されます。
  - **Explanation**：Structure に加え、コードを説明するコメントが追加されます。

- **Connector name, Constant name, Type name, Literal name, Signal name**
  - 型：String
  - デフォルト：以下の表を参照。

UML エンティティ	C コード名
Connector name	`cha_%n`
Constant name	`con_%n`
Type name	`typ_%n`
Literal name	`lit_%n_%s`
Signal name	`sig_%n`

これらの属性は、コネクタ、定数、タイプ、リテラル、シグナルのインターフェイス ヘッダー ファイル（.ifc）内の生成された C コード名の接頭辞と接尾辞を制御します。

`%n` はエンティティの名前、`%s` はスコープの名前です。このファイルから定義を完全に排除するには、`%n` を省略します。

- **Operators in environment header file**
  - 型：Boolean
  - デフォルト：True

  この属性は、インターフェイス ヘッダー ファイル（.ifc）に演算子を入れるかどうかを設定します。
- **Include references to UML source as comments**
  - 型：Boolean
  - デフォルト：False

  この機能を有効にすると、生成されたコードにトレース関数への呼び出しを含めるよう、AgileC コード ジェネレータに指示します。Print UML level trace on stdout（stdout に UML レベル トレースを出力する）機能を有効にしている場合、この属性を True に設定しなければなりません。
- **Always include `stdio.h`**
  - 型：Boolean
  - デフォルト：False

  この属性は、`make` ファイルに提出するインクルード ファイルに `stdio.h` ファイルをインクルードするかどうかを定義します。
- **Print UML level trace on `stdout`**
  - 型：Boolean
  - デフォルト：False

  この属性は、重要な UML アクションとイベントのトレースを `stdout` に出力するかどうかを定義します。重要なアクションとは、シグナルの送受信、作成、タイマーの各アクションです。

- **Generate MISRA compliant code**
  - 型：Boolean
  - デフォルト：False

  この属性は、生成後のコードを MISRA に適合させるかどうかを定義します。生成後のコードは完全に盲従するわけではありません。たとえば、ラベルへの無条件ジャンプを使用せず、同じコードセグメントを複製してインライン処理する場合があります。

Compile and link

この属性については、C コード ジェネレータステレオタイプのセクションで説明しています。

Dynamic memory allocation

このコントロールは、動的メモリの管理方法を定義する属性をグループ化します。

- **Use memory management package provided with AgileC**
  - 型：Boolean
  - デフォルト：False

  この属性は、OS 関数 `malloc` と `free` の代わりに、ライブラリに用意されているメモリ管理パッケージを使用するように定義します。このパッケージを使用した場合、Memory pool size（メモリ プール サイズ）属性を適切な値に設定しなければなりません。
- **Memory pool size**
  - 型：Integer
  - デフォルト：8192（バイト）

  この属性は、動的メモリのプールが使用するバイト数を定義します。不要な問題を回避するために、この値は 16 の倍数で設定します（Minimum block size（最小ブロック サイズ）を参照）。
- **Minimum block size**
  - 型：Integer
  - デフォルト：(空)

  この属性は、割り当てられたメモリの最小のブロック サイズを定義します。この値は 16 の倍数で指定します（最小値は 16）。

Error detection

このコントロールは、ランタイム時に実行されるアプリケーション エラー検出を定義する属性をグループ化します。

- **Basic run-time error check**
  - 型：Boolean
  - デフォルト：False

  この属性は、以下の基本状態機械 プロパティのランタイム チェックを行います。
  - シグナルの作成または出力にメモリが使用可能かをチェックします。
  - シグナルのパラメータを割り当てる際、メモリが使用可能かチェックします。
  - タイマー インスタンスの作成にメモリが使用可能かチェックします。
  - シグナルが環境に送信される際、`xOutEnv()` があるかチェックします。
  - シグナルが破棄される際にチェックします。
  - （インスタンスが）最大値に達したとき、インスタンスの追加作成試行をチェックします。
- **Index checks in arrays**
  - 型：Boolean
  - デフォルト：False

  この属性は、配列内のインデックスが許容範囲内かどうかをチェックします。
- **Syntype range checks**
  - 型：Boolean
  - デフォルト：False

  この属性は、シンタイプの値が許容値であるかどうかをチェックします。
- **Checks in predefined operators**
  - 型：Boolean
  - デフォルト：False

  この属性は、定義済み演算子の実行時のエラー状況をチェックします。
- **Check that the decision value matches an answer**
  - 型：Boolean
  - デフォルト：False

  この属性は、分岐からの有効なパスがあるかをチェックします。
- **Checks for null pointers**
  - 型：Boolean
  - デフォルト：False

  この属性は、参照読み出しの前に、ポインタに NULL 値が含まれていないかをチェックします。

- **Enable checks inside the memory management package**
  - 型：Boolean
  - デフォルト：False

  この属性は、メモリ管理パッケージのチェックを有効にします。また、Use memory management package provided with AgileC（AgileC で用意されているメモリ管理パッケージを使用）属性を True にした場合も、このパッケージが使用されます。
- **Print errors on `stdout`**
  - 型：Boolean
  - デフォルト：False

  この属性は、`stdout` にエラー メッセージを出力するかどうかを定義します。
- **Print errors on `stderr`**
  - 型：Boolean
  - デフォルト：False

  この属性は、`stderr` にエラー メッセージを出力するかどうかを定義します。
- **Print errors with error messages, not only numbers**
  - 型：Boolean
  - デフォルト：False

  この属性は、エラー メッセージを出力するか、エラー メッセージ 番号のみ出力するかを定義します。
- **Call a user provided function at warnings**
  - 型：Boolean
  - デフォルト：False

  この属性を有効にすると、ランタイム時に警告が検出された際、ユーザーが提供した関数が呼び出されます。

  ```
  void xUserWarnAction (unsigned char WarningNumber);
  ```
- **Call a user provided function at errors**
  - 型：Boolean
  - デフォルト：False

  この属性を有効にすると、ランタイム時にエラーが検出された際、ユーザーが提供した関数が呼び出されます。

  ```
  void xUserErrAction (unsigned char ErrorNumber);
  ```

Environment

このコントロールは、生成後のアプリケーションの適切な場所での環境関数 `xInitEnv` / `xCloseEnv` / `xInEnv` / `xOutEnv` の呼び出しを定義する属性をグループ化します。

- 型：Boolean

- デフォルト：False

これらの関数の呼び出しは個別に定義できます。これらの関数はユーザーが提供します。その目的と設計ガイドラインの詳細については、AgileC コード ジェネレータの説明を参照してください。

Extra code

型：String（複数行）

デフォルト：(空)

このコントロールは、**Head** と **Tail** の 2 つの属性を指定します。それぞれ、ファイルの先頭または最後にオプションのユーザー コード セクションを指定します。
`uml_cfg.h`

Generate environment template functions

この属性については、C コード ジェネレータステレオタイプのセクションで説明しています。

Make template file

この属性については、C コード ジェネレータステレオタイプのセクションで説明しています。

Operators in environment header file

この属性については、C コード ジェネレータステレオタイプのセクションで説明しています。

Process properties

型：Integer (positive)

デフォルト：5

インスタンスの最大値が制限されていないパートがある場合、起動時に、開始インスタンス数に必要な容量のメモリが動的に割り当てられ、この属性で定義された値が追加されます。値は正数でなければなりません。

Signal properties

このコントロールは、ランタイム時のシグナルの処理方法を管理する属性をグループ化します。

- **Use priorities on signals**
  - 型：Boolean
  - デフォルト：False

  この属性は、シグナル キューを最初に優先順でソートしてから着信順でソートするか、着信順のみでソートするかを定義します。
- **Length of static signal queue**
  - 型：Integer
  - デフォルト：20

  この属性は、静的シグナル キューの長さを定義します。動的なシグナルが使用されていない場合（Prevent dynamic signals（動的シグナルを抑止）を参照）、シグナルを送信しようとしたとき、シグナル キューが満杯になっていると、エラーが発生します。
- **Prevent dynamic signals**
  - 型：Boolean
  - デフォルト：False

  この属性は、シグナルに動的メモリを使用しないよう設定します。この機能は、動的メモリを使用しない小規模なシステムでは通常オフにします。その場合、Length of static signal queue（静的シグナル キューの長さ）を適切な値に設定する必要があります。
- **Signals with dynamic parameters are never discarded**
  - 型：Boolean
  - デフォルト：False

  この属性は、シグナルの破棄時、シグナル パラメータを解除するコードを削除するよう定義します（シグナルに動的パラメータが含まれる場合のみ有効）。

  この属性を True に設定した場合、動的パラメータを含むシグナルが破棄されると、メモリ リークが発生します（メモリ割り当てが解除されず、メモリ プールに戻りません）。
- **Priority for timer signals**
  - 型：Integer
  - デフォルト：50

  この属性は、すべてのタイマー シグナルに優先順位を設定します。
- **Priority for startup/create signals**
  - 型：Integer
  - デフォルト：50

  この属性は、すべての起動シグナルと生成シグナルに、優先順位を設定します。

- **Default priority for signals**
  - 型：Integer
  - デフォルト：50

  この属性は、明示的に優先順位が定義されていないすべてのシグナルに優先順位を設定します。
- **Signal parameter size**
  - 型：Integer
  - デフォルト：(空)

  この属性は、シグナル パラメータのインライン フィールドのサイズを設定します。これより大きいパラメータには動的メモリが必要です。

Timer properties

- **Length of static timer queue**
  - 型：Integer
  - デフォルト：(空)

  パラメータを持たないタイマーに対して、この属性は使用可能なアクティブ タイマーの最大値を定義します。

  パラメータを持つタイマーについては、極端な場合、この値は小さすぎることがあります。動的タイマーを使用していない場合（Prevent dynamic timers（動的タイマーの抑止）を参照）、タイマーを設定しようとしたときにタイマー キューが満杯になっていると、エラーが発生します。
- **Prevent dynamic timers**
  - 型：Boolean
  - デフォルト：False

  この属性は、タイマーに対する動的メモリの使用を制御します。この機能は、動的メモリを使用しない小規模なシステムでは通常オフにします。その場合、Length of static timer queue（静的タイマー キューの長さ）を適切な値に設定する必要があります。

Support C++

この属性については、C コード ジェネレータステレオタイプのセクションで説明しています。

Suppress C level warnings

この属性については、C コード ジェネレータステレオタイプのセクションで説明しています。

Target directory

型：String

デフォルト：(空)

この属性は、AgileC コード ジェネレータで生成されるファイルを書き出す、ファイル システム内の場所を指定します。絶対パスと相対パスのどちらも使用できます。相対パスを指定する場合、「ルート」は現行のプロジェクト (`.ttp`) ファイルがある場所です。

この属性には空のデフォルト値があります。この場合、ターゲット ディレクトリの名前付けと場所の規則が適用されます。

Target kind

この属性については、C コード ジェネレータステレオタイプのセクションで説明しています。

User defined kernel

この属性については、C コード ジェネレータステレオタイプのセクションで説明しています。

Verbose mode

型：Boolean

デフォルト：False

この属性は、コード生成時に AgileC コード ジェネレータから詳細レポートと診断をメッセージ出力領域に出力するかどうかを制御します。

build

このステレオタイプは、メタクラス アーティファクトを拡張し、アーティファクトから継承した「実際の」build ステレオタイプはすべての「基底クラス」としての役割を果たします。<<build>> ステレオタイプは実際には無効で、単独で使用する目的のものではありません。実際に使用できるのは、以下のステレオタイプです。

- <<AgileC Code Generator>>
- <<C Code Generator>>
- <<C++ Application Generator>>
- <<Makefile generator>>
- <<Model Verifier>>

<<build>> ステレオタイプは、プロファイル内ではデフォルトで非表示ですが、ここでは、ユーザーが参照できるリストを完全にするために取り上げています。

Target directory

型：String

デフォルト：(空)

この属性は、現在のビルド アーティファクトを使用したビルドの結果として生成されるファイルを書き出す、ファイル システム内の指定を設定します。ファイルの場所を相対パスで指定する場合、「ルート」は、現行のプロジェクト（.ttp）ファイルがある場所です。

この属性には空のデフォルト値があります。この場合、ターゲット ディレクトリの名前付けと場所の規則が適用されます。

C Code Generator

このステレオタイプは、ベース ステレオタイプ <<build>> を継承し、C コード ジェネレータによる C コードの生成を制御する属性が含まれます。

Additional Preprocessor Defines

型 : String

デフォルト：（空）

この属性を使用して追加のコンパイラ スイッチを指定できます。このコンパイラ スイッチはコンパイラに渡されて、カーネル ファイルを含むすべての C/C++ ファイルのコンパイルに作用します。

プリプロセッサ定義は、以下のような、-D フラグを使用する構文で行います。

```
-DUSERDEF1 -DUSERDEF2=15
```

この文字列は引用符で囲まないでください。引用符は自動的に挿入されます。

Advanced options

型：String（複数行）

デフォルト：(空)

この属性は C コード ジェネレータに渡される、詳細オプションを定義します。このオプションは、他のコード生成オプションが指定された後に、テキスト ボックスでの表示順に追加されます。

注記

上記で説明した以外の方法でこの機能を使用すると、コード生成オプションが不必要に上書きまたは変更され、予期せぬ振る舞いが発生する危険性があります。

以下のオプションがサポートされています。

- **Set-Signal-Number**
  このオプションは、環境へのシグナルの送信時に、シグナルを検索しやすくするため、環境への（または、からの）シグナルに番号を割り当てるよう C コード ジェネレータに指示します。
  この結果、C コード ジェネレータに割り当てられたシグナル番号の情報を含む <basename>.hs という名前のファイルが作成されます。この機能の使用方法については、899 ページの「シグナルが多数ある場合の xOutEnv のパフォーマンス向上」を参照してください。

注記

このファイルを正確なものにするには、アプリケーション全体をビルドする必要があります。部分ビルドを実行すると、シグナル番号が不正確になることがあります。

- **Set-SDL-Coder**
  このオプションは、環境への（または、からの）シグナルのデータのエンコード方法に関する情報を生成するよう C コード ジェネレータに指示します。
  このオプションを選択すると、<basename>_cod.h と <basename>_cod.c という 2 つのファイルが作成されます。この機能の使用方法については 891 ページの「シグナル パラメータのエンコードとデコード」を参照してください。

Code generation properties

型：String

デフォルト：

UML エンティティ	C コード名
Connector name	cha_%n
Constant name	con_%n
Type name	typ_%n
Literal name	lit_%n_%s
Signal names	sig_%n

これらの属性は、システム インターフェイス ヘッダー ファイルで生成された C コード名の接頭辞と接尾辞を制御します。

Operators in environment header file：837 ページの「Operators in environment header file」（環境ヘッダー ファイルの演算子）を参照してください。

Compile and link

型：Boolean

デフォルト：True

この属性は、C ファイルのコンパイルとリンクのための make ファイル生成に AgileC コード ジェネレータと C コード ジェネレータのどちらを使用するか、また、ビルド プロセス中に自動実行するかを定義します（make ファイルの作成と実行のオプションは個別に制御できません）。

Expand macros

型：Boolean

デフォルト：False

この属性は、コードを読みやすくして C レベルでのデバッグを簡単にするため、コード生成後に C コードのマクロを拡張して処理するかどうかを制御します。この処理は C コンパイラ ドライバユーティリティで実行されます。

注記

マクロの拡張は、Win32 に設定された Target kind（ターゲットの種類）ではサポートされていません。

Generate environment template functions

型：Boolean

デフォルト：False

この属性は、アプリケーションまたはモデル ベリファイヤ（Model Verifier）の生成時に、環境関数のスケルトンを持つファイルを {AgileC コード ジェネレータ | C コード ジェネレータ } で作成するかどうかを制御します。

環境へのインターフェイスの最新定義を持つシステム インターフェイス ヘッダー ファイル（`.ifc`）は、この属性の値とは関係なく常に作成されます。

Make template file

型：String

デフォルト：（空）

この属性は、システムの make ファイルの作成時に、指定された make テンプレート ファイル（ユーザーが提供したファイル）を、C コード ジェネレータまたは AgileC コード ジェネレータで使用するかどうかを制御します。make テンプレート ファイルにより、生成される C コードにコンパイルおよびリンクされる外部コードをインクルードできます。相対パスは、プロジェクトのディレクトリとの相対パスです。ターゲット ディレクトリ内の生成されたデフォルト名（ビルド ルート名に依存）を持つ .tpm ファイルを使用する場合は「+」記号を入れます。

Operators in environment header file

型：Boolean

デフォルト：False

この属性は、C コード ジェネレータで生成されるシステム インターフェイスのヘッダー ファイルに UML 演算子の定義を入れるかどうかを制御します。

Simulation kind

型：SimulationKind

デフォルト：Standard

この属性は、モデル ベリファイヤ（Model Verifier）のタイマー処理を制御します。`SimulationKind` は次のいずれかの値を取ります

```
Standard
Realtime
With Environment
```

Standrad の指定はデスクリートシミュレーションを意味します。環境はモデルベリファイヤから制御されます。

:**Realtime** を指定すると、設定されたタイマーにリアルタイム クロック タイマーの遅延が生じます。この遅延はタイム ユニットにつき 1 秒です。コンパイル時にマクロ XCLOCK が設定されます。

With Environment を指定すると、アプリケーション API を通じた環境制御付きモデルベリファイヤを生成します。コンパイル時にマクロ XENV が設定されます。

Support C++

型：Boolean

デフォルト：False

この属性は、C コード ジェネレータまたは AgileC コード ジェネレータで生成されたコードに、C++ コンパイラでのコンパイルまたは ISO C コンパイラによるコンパイルを可能にするプロパティを付与します。主要事項のプロパティは、とりわけ、アプリケーション コードとランタイム ライブラリにコンパイルおよびリンクされる外部コードのハンドリングです。

また、この属性は、コードのコンパイルおよびリンク時に `_cpp` 接尾辞を持つ適切なライブラリが使用されるよう制御します。

Suppress C level warnings

型：Boolean

デフォルト：False

この属性は、以下のツールから出力される警告メッセージを抑止するかどうかを制御します。（このような警告は、多くの場合無視されます。）

- C コード ジェネレータ / AgileC コード ジェネレータ / Model Verifier
- C コンパイラ
- C リンカー

Target directory

型：String

デフォルト：(空)

この属性は、C コード ジェネレータで生成されるファイルを書き出す、ファイル システム内の場所を指定します。絶対パスと相対パスのどちらも使用できます。相対パスを指定する場合、「ルート」は現行のプロジェクト (`.ttp`) ファイルがある場所です。

この属性には空のデフォルト値があります。この場合、ターゲット ディレクトリの名前付けと場所の規則が適用されます。

Target kind

型：TargetKind

デフォルト：(ホストに依存)

この属性は、C コード ジェネレータでビルドするターゲット アプリケーションの種類を制御します。

`TargetKind` の指定可能な値は、Win32、Solaris - cc、Solaris - gcc、Linux - gcc、Cygwin です。

注記

ターゲットの種類、Solaris - cc、Solaris - gcc および Linux - gcc は、UNIX でのみサポートされます。

User defined kernel

型：String

デフォルト：(空)

この属性は、アプリケーションのコンパイルおよびリンク時に、C コード ジェネレータで用意されているライブラリから選択せず、ユーザー定義ランタイム ライブラリを使用するかどうかを定義します。

このようなライブラリは、コードのプリプロセスとコンパイルの方法を管理する C マクロとコンパイラ スイッチの定義を含むファイルが格納されているディレクトリで識別されます。

Verbose mode

型：Boolean

デフォルト：False

この属性は、C コード ジェネレータから詳細なレポート、診断およびメッセージをメッセージ出力領域に出力するかどうかを制御します。

参照

第 25 章「C および AgileC ランタイム ライブラリ」の 909 ページ、「サポートされるライブラリ」

第 25 章「C および AgileC ランタイム ライブラリ」の 912 ページ、「ライブラリ ファイル」

C Application

このステレオタイプは、AgileC コード ジェネレータ、C コード ジェネレータ、Model Verifier のいずれかのコード ジェネレータを使用して C アプリケーションをビルドする際、コードの生成を制御します。

このステレオタイプで以下を制御します。

- 個別のモデル要素に対するコード生成を無効にします。
- C アプリケーションの生成時の外部 C または C++ 宣言のインポート方法を柔軟に設定します。C または C++ コードは、手書きのコードやサード パーティ ライブラリでもインポート可能です。

Generate C code

型：Boolean

デフォルト：True

この属性は、UML モデルのある要素を C コード ジェネレータまたは Model Verifier ビルドタイプを使用するビルドにインクルードするかどうかを制御します。

このようなビルドのデフォルトの振る舞いは、ビルド アーティファクトのビルド ルートの使用として指定されたクラスに含まれるクラスやパッケージをすべてインクルードします。

この属性で、モデル内の個々の要素の破棄を指定できます。これにより、まだ実装されていないか、適切に動作していないと判断されたアプリケーションのパートを、インスタンスから除外できます。

要素を破棄すると、セマンティック上不正なモデルが作成されたり、コードを生成できなくなることがあります。

Include File

型：String

デフォルト：(空)

この属性は、この属性を適用する要素に対して生成されたコードに、外部定義を持つファイル（通常、C または C++ ヘッダー ファイル）をインクルードするよう定義します。

生成後の C コードには、外部定義を持つファイルをインクルードする C/C++ の `#include` 文が作成されます。

ヒント

この文字列属性で指定されるファイルは 1 つだけです。複数のヘッダー ファイルをインクルードする必要がある場合、生成後のコードが外部コードのコンパイルとリンクに必要とするすべてのヘッダー ファイルに対して、#include 文を含む（通常この文以外には何もない）ヘッダー ファイルを 1 つ作成し、プロパティ エディタでこのファイルをインクルード ファイルとして使用するよう指定します。

Language

型：langKind

デフォルト：C

この属性は、外部宣言に使用するプログラミング言語を指定します。**langKind** で使用可能な属性値は C および C++ です。

C++ 言語を使用する場合、生成後のコードが外部コードを適切にコンパイルおよびリンクし、生成後の実行形式ファイルの振る舞いを予測どおりにするには、通常、C コード ジェネレータステレオタイプ（または、ビルド タイプによってはモデル ベリファイヤ（Model Verifier））の Support C++（C++ のサポート）属性を True に設定します。

C name

型：String

デフォルト：(空)

この属性は、C コードで要素に対して与えられる名前を定義して、C コード ジェネレータで定義された名前付けスキームを置き換えます。

また、UML モデルで指定された名前ではなく、C コード ジェネレータで生成されたコードで、要素に名前を与えるかどうかを定義します。この場合、C コード ジェネレータで採用された名前付けスキームを追跡せずに、名前を指定する外部宣言を含めることができます。

参照

生成された C コードの名前

ターゲット コード式

C Application Customization

このステレオタイプは、C コードの生成に使用する詳細設定をカスタマイズします。このステレオタイプの設定は、すべての C ビルド タイプ（C コード ジェネレータ、AgileC コード ジェネレータ、モデル ベリファイヤ（Model Verifier））で共有され、ホスト上でのシステムのデバッグに使用するアプリケーションとターゲットに配置されるアプリケーションの振る舞いを一貫したものにします。

Priority

型：Integer（`0..255`）

デフォルト：(空)

この属性によって、シグナル定義での優先順位の指定が可能になります。C コード ジェネレータ、AgileC コード ジェネレータおよび Model Verifier には、着信順にキューをソートするデフォルトの機能以外に、シグナル優先順位に従って キューを配列する機能があります。

値が低いほど優先順位は高くなります。

デフォルトでこの属性は空です。つまり、コード ジェネレータが、着信順にシグナル キューにシグナルを追加します。

C++ Application Generator

このステレオタイプには、C++ アプリケーション ジェネレータによる C++ コード生成に使用する設定が含まれます。

このステレオタイプの属性は C++ アプリケーションジェネレータの翻訳オプションに対応付けられています。

C++ header file

このステレオタイプは、モデル要素に対して C++ アプリケーション ジェネレータから生成された C++ コードのタイプ定義を、指定されたファイルに保存するよう指定します。

Makefile ジェネレータは C++ コードをマニフェストするファイルを考慮し、生成後の make ファイルにはこれらのファイルに対する依存関係を設定します。

File name

型：String

デフォルト：(空)

この属性は、C++ アプリケーション ジェネレータがモデル要素に対して生成した C++ 定義を保存するヘッダー ファイルの名前を指定します。この名前は、ファイル拡張子（`.h`、`.hpp` など）も含めて指定しなければなりません。

Precompiled

型：Boolean

デフォルト：False

この属性は、指定されたファイルが定義済みバージョンにあることを示します。この機能は Windows の C++ コンパイラでのみサポートされます。

C++ implementation file

このステレオタイプは、モデル要素に対して C++ アプリケーション ジェネレータから生成された C++ コードの実装コードを、指定したファイルに保存するよう指定します。

Makefile ジェネレータは、C++ コードをマニフェストするファイルをソースと見なします。生成後の make ファイルにはこれらのファイルに対する依存関係を設定します。

File name

この属性は、C++ アプリケーション ジェネレータがモデル要素に対して生成した C++ コードを保存するソース ファイルの名前を指定します。この名前は、ファイル拡張子（`.c`、`.cpp` など）も含めて指定しなければなりません。

参照

library（ライブラリ）

cppImportSpecification

このステレオタイプ **cppImportSpecification** は、C/C++ のインポートの結果作成されるモデル要素の「ターゲット」として使用される UML パッケージに適用されます。

Add source file references to enable navigation from the UML model to the C++ source

型：Boolean

デフォルト：True

この属性を True に設定すると、［モデル ビュー］からインポート後のパッケージの要素を選択し、ショートカット メニューから［ソースに移動］を選択できるようになります。これによりソース ヘッダー ファイルが開き、選択した要素の元のソース ファイルに移動できます。

Always generate constant expressions within [[]]

型：Boolean

デフォルト：False

この属性を True に設定すると、インポータはすべての C/C++ 定数式を UML ターゲット コード式 ([[...]]) に翻訳します。False の場合は、UML 式に翻訳します。

C only

型：Boolean

デフォルト：False

この属性により、C モードでインポートを実行できます。C モードでは、定義内に C++ 構成要素がないことを前提とします。入力内に C++ コードがある場合の解釈は**定義されていません**。

C および C++ ヘッダー ファイルのインポート時、各インポート パスで生成された要素を保存するパッケージを 1 つ割り当て、C と C++ を個別に処理し、Input header files（入力ヘッダー ファイル）と Preprocessor（プリプロセッサ）も個別に設定できるようにします。

C/C++ dialect

ここでは、個別に「True」または「False」を設定することで、指定された C/C++ 固有表現に対応できる一連のブール属性をグループ化します。

すべての固有表現が無効になっている場合、（コードが C only（C コードのみ）としてインポートされているかどうかに依存）ISO C または C++ ISO/IEC 14882 標準規格がサポートされます。

- **GNU C/C++**

  型：Boolean

  デフォルト：False

  この属性は、インポートが GNU C/C++ 固有表現に従うかどうかを制御します。
- **Microsoft C/C++**

  型：Boolean

  デフォルト：False

  この属性は、インポートが Microsoft Visual C/C++ 固有表現に従うかどうかを制御します。
- **Borland C/C++**

  型：Boolean

  デフォルト：False

  この属性は、インポートが Borland C/C++ 固有表現に従うかどうかを制御します。

Do not import definitions from included header files

型：Boolean

デフォルト：True

このオプションを使用して、大量の定義がインポートされないよう、ネストされたライブラリからの定義のインポートを抑止できます。

このオプションが False に設定されている場合、すべての参照先ヘッダー ファイルのすべての定義が、結果のパッケージにインポートされます。

このオプションを True に設定すると、インクルードされたヘッダー ファイルの定義は結果のパッケージにインポートされません。このオプションは、標準ヘッダー ファイル（#include <header.h>）、またはユーザー ヘッダー ファイル（#include "header.h"）のどちらが使用されているかに影響されません。複数のヘッダー ファイルをインポート

する場合、相互に包含状態になっていても、入力 ヘッダー ファイルにすべてのファイルをリストしなければなりません。また、Selective import（選択インポート）を使用していて、インポート後のヘッダーの定義が、インクルードされた別ヘッダーの別定義を参照する場合、Translation of depending declarations（依存宣言の変換）が設定されていれば、この依存関係の定義が結果モデルにインポートされます。

注記

標準のヘッダー ファイルを使用すると、インポートされる定義が大量になりパフォーマンスに影響を及ぼします。

Generate artifacts

型：Boolean

デフォルト：False

この属性を True にすると、インポータはインポートされるファイルを表すファイル アーティファクトを生成します。また、C++ アプリケーションジェネレータ用のビルドアーティファクトも生成します。このアーティファクトの使用目的は、C++ アプリケーションジェネレータを使用するインポートファイルから容易に再生成できるようにすることです。したがって、このオプションの主な使用法は、C++ から UML への移行のケースです。インポートが繰り返し行われる場合、ファイルアーティファクトはインポートのたびにインポートされたファイルについて更新されますが、ビルド アーティファクトは更新されません。したがって、C++ コード生成を設定するためのには、アーティファクトのオプションを手動で変更するのが安全です。この設定は繰り返しのインポートの間には上書きされません。

GUID algorithm

型：`GuidStrategyKind`

デフォルト：Random

この属性は、GUID の生成について、ランダムパターンを使用するか、C/C++ ヘッダーファイルで定義された名前から名付けするかを制御します。`GuidStrategyKind` に使用可能な値は、**Random** と **Name** です。

Action code strategy

型：`ActionCodeStrategy`

デフォルト：`DoNotImportAc` (“Don’t Import Action Code”)

この属性を `ImportAC`（“Import Action Code“）にすると、インポートされるファイル内のすべての関数本体が、UML アクションを使用してインポートされます。典型的な使用例としては、レガシー C/C++ アプリケーションから UML へ移行し、後で C++ アプリケーションジェネレータでコードを再生成するというケースがあります。関数本体を UML 中で可視化するためにも使用できます。

関数本体は、UML モデルの非形式アクションを使用してインポートするほうが便利な場合があります。その場合は、この属性を `ImportACAsInformal`（“Import Action Code as Informal”）に設定します。この設定により、関数本体の内容が UML 操作本体に非形式アクションとしてコピーされます。この方法は、操作本体にあるコード レイアウト、マクロなどを維持できるという利点があります。この設定をしない場合は、これらはインポート時に失われます。ただし、これは、同時に UML チェッカが操作本体の正確性をチェックしないことも意味します。

Import char* as CPtr<char>

型：Boolean

デフォルト：False

このオプションを使用して、C/C++ の `char*` を「`char*`」ではなく `CPtr<char>` としてインポートできます。

このオプションを True にしてデフォルトの振る舞いを上書きすると、外部 `char*` 定義が「`char*`」ではなく `CPtr<char>` としてインポートされます。バージョン 2.3.00 以前のツールを使用してインポートした文字列がモデルにすでに含まれている場合、このオプションが有効です。

Import C++ pure virtual classes to UML interfaces

型：Boolean

デフォルト：True

このオプションで、C++ の純仮想クラスのインポート方法を指定できます。

- このオプションを True に設定すると、純仮想メソッドのみを含む C++ クラスが UML インターフェイスにインポートされます。
- このオプションを False に設定すると、C++ 純仮想クラスが UML 抽象クラスにマッピングされます。

詳細については、504 ページの「純仮想メンバー関数」を参照してください。

Import unsigned char to Octet

型：Boolean

デフォルト：False

このオプションを True に設定すると、C++ の unsigned char 型が UML の 8 ビットタイプにマッピングされます。このようなマッピングは SDL Suite から移行されたシステムで役立ちます。なぜなら、C++ の `unsigned char` 型が、C++ をサポートする SDL Suite の定義済みパッケージで定義された SDL タイプの 8 ビットで表示されるからです。

このオプションを False に設定すると、C++ の `unsigned char` 型が対応する UML の unsigned char タイプにマッピングされます。

Import class pointers to UML references

型：Boolean

デフォルト：True

この設定で、C++ ポインタのクラスへのインポート結果を UML クラスの直接参照とする（バージョン 2.3.00 で導入されたマッピングを使用）か、前バージョンと同様に CPtr<T> テンプレートを使用するかを定義します。

この設定は、主にバージョン 2.3.00 以前で使用された C++ インポート マッピング スキームとの下位互換性を提供することを目的としていますが、インポート後の機能性は低下します（847 ページの例 337 を参照）。

以下の C++ 定義のインポートを考えます。

```
class C {}
class D {
  C* c;
}
void op (C* par);
```

デフォルトで、インポートによって以下の UML 定義が生成されます。

```
class C {};
class D {
  C c;
}
void op (C par);
```

この属性を False に設定すると、代わりに、以下の UML が生成されます。

```
class C {};
class D {
  CPtr<C> c;
}
void op (CPtr<C> par);
```

例 **337:** インポート後のクラス間の関連の可視化

以下の C++ 定義を考えます。

```
class a {
public:
  int x;
};

class b {
public:
 a* a1;
};

int Getx( b* pb );
```

これら 2 つのクラスをインポートとすると、2 つの UML クラスと 1 つの操作になります。これらの定義をクラス図にドラッグすると、848 ページの図 233 に示すように、クラス b とクラス a 間、および操作 Getx とクラス b 間に関連ラインが自動的に描画されます。

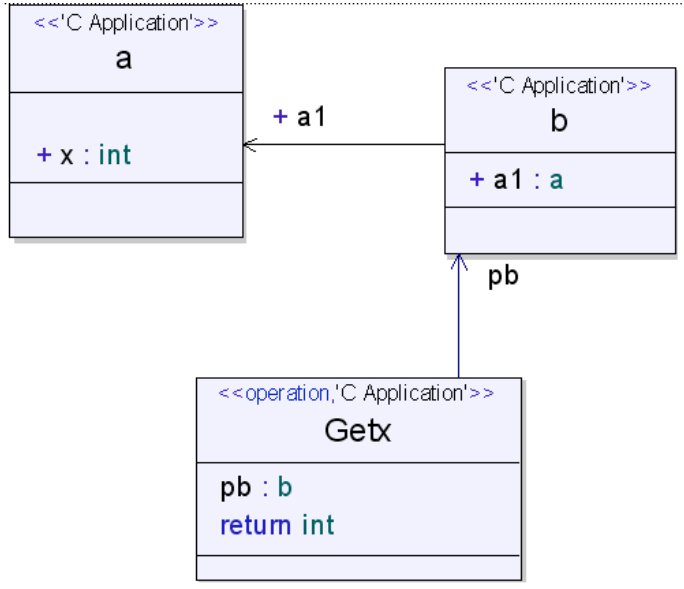


図 233:

Input header files

型：Charstring のリスト

デフォルト：（空）

この設定で、宣言を含む入力ファイル（つまり、C/C++ ヘッダー）のリストを定義します。この宣言は、インポート時に処理され、Output header file（出力ヘッダー ファイル）に入れられ、最終的に UML に変換されます。

ツールバーには、インポートするファイルのリストに項目を挿入するためのボタンがあります。［新規］ボタンまたは［挿入］ボタンをクリックして、以下のいずれかを実行します。

- 選択するヘッダー ファイルの名前を入力します。
- 新たに作成した項目の右に表示される［参照］ボタンをクリックしてから、［開く］ダイアログでファイルを指定します。

インポートしないファイルは、［削除］ボタンを使用してリストから削除します。

ファイルのリストは、現在定義されている Preprocessor（プリプロセッサ）に `#include <filename>` のリストとして渡されます。プリプロセッサまたは C/C++ のインポートの技術的な制限により、不正な再宣言などから競合が発生した場合、インポートの全体または一部が失敗することがあります。［上に移動］ボタンと［下に移

動］ボタンを使用して、インポートするファイルの順番を変更できます。インポートスキームは上から順に行われます。定義のインポートの順序が重要な場合、正常にインポートできるよう、リストを再配列できます。

ヘッダー ファイルのリストでは、C only（C コードのみ）と Preprocessor（プリプロセッサ）に適切な値を設定する必要があります。

Options

型：`Charstring`

デフォルト：（空）

このテキスト ボックスでプリプロセッサに渡すオプションを指定します。通常、このようなオプションとして、指定値に拡張されるプリプロセッサ マクロが使用されます。

例 **338:** プリプロセッサ オプション

```
/Dmymacro1/Dmymacro2
```

マクロ定義の構文（上記の例では `/D`）は、実際に使用するプリプロセッサ（Preprocessor（プリプロセッサ）属性で指定）によって異なります。

Output header file

型：Charstring

デフォルト：（空）

この設定で、C/C++ のインポート時に生成される出力ヘッダー ファイルを定義します。これは、インポートする各 Input header files（入力ヘッダー ファイル）の `#include` 文で構成されます。この出力ヘッダー ファイルは解析されて、UML に変換されるファイルです。

C コード ジェネレータによって生成される最終 C コードに出力ヘッダー ファイルを含める必要があるため、通常は、ビルド ターゲット ディレクトリにヘッダー ファイルを置くとよいでしょう。コード生成を行わない場合、Output header file（出力ヘッダー ファイル）属性は指定されないままです（この場合、ツールによって一時出力ファイルが作成されます）。

Preprocessor

型：`Charstring`

デフォルト：（空）（空の値は、インポート時に Windows では「`cl`」、UNIX では「`cpp`」と解釈されます。）

このテキスト フィールドでは、ヘッダー ファイルをインポートする前に使用するプリプロセッサを指定できます。このテキスト フィールドには、通常、プリプロセッサの起動に使用するコマンド、または実際に使用するプレプロセッサを「埋め込む」ための適切なスクリプトを指定します。

コマンドまたはスクリプトは、直接入力するか、［参照］ボタンをクリックして、［開く］ダイアログからファイルを検索して指定できます。

C/C++ インポート時に、以下のプリプロセッサが名前で認識されます。

- cl（Microsoft Visual C/C++ プリプロセッサ）：Windows のみ
- cpp32（Borland C/C++ プリプロセッサ）：Windows のみ
- cpp：UNIX のみ
- g++/gcc（GNU C/C++ プリプロセッサ）：UNIX（Cygwin を含む）

使用するプリプロセッサがプラットフォームで認識可能なプリプロセッサの中にない場合、プリプロセッサを呼び出すシェル スクリプトを記述できます。スクリプトには、上記のリスト以外の名前を付けるように注意してください。C/C++ インポート時に、以下の 2 つのパラメータを持つスクリプトが呼び出されます。

1. プリプロセスする入力ヘッダーの名前
2. プリプロセスの結果を保存するファイルの名前

例 **3391: Windows** 上で **gcc** を呼び出すシェル スクリプト ——————————

このスクリプト ファイルは gcc プリプロセッサを（cygwin から）呼び出します。プリプロセッサは ImportedDefinitions パッケージの cppImportSpecification 内の `mycpp.bat` で指定します。

［ImportedDefinitions］パッケージをクリックして［Import］を選択すると、`mycpp.bat` が呼び出されます。出力タブに以下のログが表示されます。

```
gcc -E -v -I . -x c++ %1 -o %2.i
@copy %2.i %2
```

——————————————————————————————————

例 **340: UNIX** 上でユーザー定義プリプロセッサを呼び出すシェル スクリプト ———

UNIX プラットフォームのサンプル スクリプト `mycpp.sh` を以下に示します。

```
gcc -E -v -I . -x c++ $1 -o $2.i
cp $2.i $2
```

——————————————————————————————————

Selective import

型：Charstring のリスト

デフォルト：（空）

この編集リストには任意の数の C/C++ 識別子を含めることができます。これらの識別子は、対応する宣言を変換すると、モデル内で使用可能になります。

デフォルトでこのリストは空です。つまり、解析された C/C++ 宣言のすべてがモデルにインポートされます。同じ識別子を何回入力しても、インポートされるのは 1 回のみです。選択インポートの場合、Translation of depending declarations（依存宣言の変換）を実行して、インポート結果の UML 宣言が完全なものであることを確実にします。

Set External attributes for imported definitions

型：Boolean

デフォルト：True

この属性を True に設定すると、外部属性がインポート済み定義に設定されます。

インポート後の定義を再利用しない場合、このオプションを False に設定します。

Translation of depending declarations

型：Boolean

デフォルト：True

この属性を True に設定すると、インポート仕様の識別子が他の宣言を参照し、その宣言がさらに他の宣言に依存している場合、これらのすべての依存宣言も変換されます。この原理は、依存宣言に依存するすべての宣言に繰り返し適用され、結果の UML 宣言は完全で適合したものとなります。

この属性を False に設定すると、依存宣言は自動変換されません。この場合、結果の宣言が完全なものになる保証はありません。Selective import（選択インポート）を行う場合、これは特に注意が必要です。Do not import definitions from included header files（インクルードされたヘッダー ファイルの定義をインポートしない）オプションが true（デフォルト）の場合、全依存宣言に必要なヘッダーファイルがすべてリストされていることを確認する必要があります。

Import only exported definitions

型：Boolean

デフォルト：True

この属性を true に設定すると、C/C++ インポータは `__declspec(dllexport)` キーワードでマークされた定義のみをインポートします。このオプションを使用するには C/C++ コンパイラが `__declspec` キーワードをサポートしている必要があります。

Java

このステレオタイプは、Java ビルドアーティファクトの定義のために使用されます。Java ビルドアーティファクトは Java ソースコードを生成し、UML との間でラウンドトリップするために使用します。

このステレオタイプの属性は、Java コードジェネレータの Java ビルドアーティファクト設定に対応付けられています。

Configuration

このステレオタイプには、ビルド アーティファクトが属する構成を制御する設定が含まれます。

注記

ビルド アーティファクトには複数の configuration ステレオタイプが適用され、各構成にこれが 1 つ含まれています。このステレオタイプはプロパティ エディタの［フィルタ］ドロップダウン メニューに複数のオカレンスで表示されます。

name

型：String

デフォルト：(空)

この属性にはビルド アーティファクトが所属する構成の名前が含まれます。

dynamicLibrary

このステレオタイプは、親ステレオタイプ <<library>> を継承します。ファイル アーティファクトを特化して動的にリンクされたライブラリをマニフェストできるようにするために使用します。

このステレオタイプは UML パッケージでの使用に適しています。

dynamicLibrary ステレオタイプを使用するファイル アーティファクトは Makefile ジェネレータで管理され、ステレオタイプを適用する UML パッケージで表示される C++ ソース ファイルに対する依存関係が確立されます。

リンクされた C++ ソースは以下のようになります。

プラットフォーム	結果ファイル
Windows	`<File name>.DLL`（Windows アプリケーションの拡張子）
UNIX	`lib<xxx>.so` （共有オブジェクト）

注記

このステレオタイプは C++ アプリケーション ジェネレータによってビルドされたモデルでのみ使用します。

File name

型：String

デフォルト：(空)

この属性は、ファイル アーティファクトによって指定された動的ライブラリを実装するファイルのベース名を指定します。

executable

このステレオタイプは、ファイル アーティファクトを特化して、実行アプリケーションを表示させるために使用します。コンパイルとリンクに必要なすべての定義と実装を含むスコープを定義するモデル要素に追加するために適したステレオタイプです（通常最上位クラス）。

executable ステレオタイプを使用するファイル アーティファクトは、Makefile ジェネレータに管理され、ステレオタイプを適用する要素によって定義されたスコープに表示される C++ ソース ファイルとライブラリに対する依存関係が確立されます。

C++ オブジェクトとライブラリがリンクされ、以下のようになります。

プラットフォーム	結果ファイル
Windows	`<File name>.DLL`（Windows アプリケーション）
UNIX	`<File name>` （ELF；Executable Library Format）

注記

このステレオタイプは C++ アプリケーション ジェネレータによってビルドされたモデルでのみ使用します。

file

このステレオタイプはメタクラス アーティファクトを拡張してアーティファクトを特化し、ファイル アーティファクトにします。これはベース ステレオタイプです。プロファイルでは非表示のマークが付けられ、基本形式では実際に利用されません。

実際のアプリケーションのモデリングとビルド時には、以下の特化された file ステレオタイプが使用されます。

- <<C++ header file>>
- <<C++ implementation file>>
- <<dynamicLibrary>>
- <<executable>>
- <<staticLibrary>>

注記

C コード ジェネレータ、AgileC コード ジェネレータまたはモデル ベリファイヤ（Model Verifier）を使用してモデルをビルドする場合、これらのファイル アーティファクトは使用されません。

File name

型：String

デフォルト：（空）

この属性は、ステレオタイプを適用する UML 要素を表現するファイルのベース名を指定します。

Icon

このステレオタイプには、ワークスペース ウィンドウおよびエディタでのグラフィック要素の形状を調整できる属性が含まれます。

16 x 16 Pixels Bitmap file

型：String

デフォルト：(モデル要素に依存)

この属性は、ワークスペース ウィンドウに表示する際の要素の形状を指定します。通常、ビルド アーティファクトおよびファイル アーティファクト（<<file>> または、<<build>> ステレオタイプが適用された要素）の機能と容易に関連付けられる形状を指定するため、この属性が用いられます。

この文字列は、適した 16 x 16 ピクセル ビットマップまたはアイコンを含むファイル指定します。デフォルト値は、Tau インストール ディレクトリの `etc` ディレクトリに格納されたファイルを参照します。

この属性値は、モデルやそのビルド方法にセマンティック上の影響を与えません。「外観」にのみ影響を与えます。

Icon File

型：String

デフォルト：(空)

この属性は、エディタ ウィンドウに表示する際のシンボルの形状を指定します。パッケージ、クラス、パート、ステートなど、「重要な」UML シンボルのサブセットによりサポートされます。

この文字列は、適切なビットマップまたはアイコン ファイルを含むファイルを指定します。

この属性が空の場合、シンボルは出荷時設定で表示されます。つまり UML シンボルは OMG の標準設定で表示されます。

この属性はセマンティック上の意味をもちません。「外観」のみに影響します。

KeepIconProportions

型：Boolean

デフォルト：False

この属性を true に設定すると、元のイメージの縦横比が維持されます。

参照

第 3 章「ダイアグラムの操作」の 152 ページ、「アイコン」

LabelPosition

labelVertPosition

型：{TopOutside | TopInside | VCenter | BottomInside | BottomOutside}

デフォルト：(空)

この属性は、シンボルのテキスト ラベルの上下方向の配置方法を指定します。

labelHorzPosition

型：{LeftOutside | LeftInside | HCenter | RightInside | RightOutside}

デフォルト：(空)

この属性は、シンボルのテキスト ラベルの左右方向の配置方法を指定します。

jarFile

このステレオタイプは Tau でサポートされる Java テクノロジによって管理およびビルドされる UML モデルに使用されます。C または C++ テクノロジを使用するアプリケーションのビルドには使用されません。

参照

第 33 章「Java サポート」の 1145 ページ、「Java ファイル」

javaFile

このステレオタイプは Tau でサポートされる Java テクノロジによって管理およびビルドされる UML モデルに使用されます。C または C++ テクノロジを使用するアプリケーションのビルドには使用されません。

参照

第 33 章「Java サポート」の 1145 ページ、「Java ファイル」

library

このステレオタイプは、ファイル アーティファクトを特化して、ライブラリを指定するために使用します。ここでいう「ライブラリ」は、アトミックな方法による管理に適していて、リンカーへの入力が可能な「汎用オブジェクトファイルコンポーネント」を意味します。ただし、ライブラリプロパティやこのテクノロジに関する詳細はここでは指定しません。

このステレオタイプは UML パッケージで使用するのに適しています。

library ステレオタイプを使用するファイル アーティファクトは、Makefile ジェネレータにより管理されます。その方法は、生成後の make ファイルで、モデル要素を実装するソース ファイルとファイル アーティファクトによって指定されるファイル間で make 依存関係を定義する方法と同じです。

ライブラリプロパティをよく理解し、使用するライブラリ テクノロジを選択できるようになると、library ステレオタイプの代わりに、以下のステレオタイプを使用することも可能です。

- <<dynamicLibrary>>
- <<staticLibrary>>

注記

このステレオタイプは C++ アプリケーション ジェネレータによってビルドされたモデルでのみ使用します。

File name

型：String

デフォルト：(空)

この属性は、ファイル アーティファクトで指定されたライブラリを実装するファイルを指定します。

参照

C++ implementation file（C++ 実装ファイル）

makefile

このステレオタイプは、ファイルの生成を安全かつ正確に管理するため、ツールの内部で使用されます。

ユーザーが参照できるリストを完全にするために取り上げていますが、ユーザーによる使用を意図したものではありません。

このステレオタイプを使用して実際に動作可能な make ファイルを Makefile ジェネレータから作成できるのは、ある特定の場合のみです。IBM Rational から具体的な指示を受けない限り、このステレオタイプの使用は推奨できません。

Make settings

このステレオタイプには、make ユーティリティの実行に使用する設定が含まれています。make ユーティリティは、生成コードのコンパイルを含むビルド コマンドの命令結果として実行されます（例：構成のビルド）。

make ユーティリティは以下のいずれかの方法で実行できます。

- Makefile ジェネレータで生成された make ファイルで透過的に操作
- ユーザーが提供する make ファイルで明示的に操作

Command

型：String

デフォルト：(空)

この属性は、デフォルト コマンドの代わりに、実際に実行する make コマンドを指定します。

Dialect

型：{DEFAULT | gmake | nmake | sunmake}

デフォルト：DEFAULT

この属性は、make ファイルを make ユーティリティに渡す際に使用する make の固有表現を指定し、適切な方法でファイル パスとオプションを指定します。

- DEFAULT は、ホスト OS に従い、make ファイルを「ネイティブ」とします。
  - nmake（Windows ホスト）（ファイル パスには円記号を使用し、オプションにはスラッシュを使用）
  - gmake（Linux ホスト）（ファイル パスにはスラッシュを使用し、オプションにはダッシュを使用）
  - sunmake（Solaris ホスト）（ファイル パスにはスラッシュを使用し、オプションにはダッシュを使用）
- gmake と sunmake は「make」の呼び出しに UNIX 規則を指定します。
- nmake は「make」の呼び出しに Windows 規則を指定します。

Makefile

型：String

デフォルト：(空)

この属性は、Makefile ジェネレータで生成される make ファイルの代わりに、入力として使用する特定の make ファイルを指定します。

Options

型：String（複数行）

デフォルト：(空)

この属性は、make ユーティリティの起動時に使用する、任意数のユーザー定義オプションを指定します。

Makefile generator

このステレオタイプはベース ステレオタイプ <<build>> を継承し、生成後の make ファイルの内容と場所を制御する設定が含まれます。コードと make ファイルの生成を含むビルド コマンド命令の結果として、make ファイルが生成されます（例：構成の生成）。

Dialect

型：{Default | gmake | nmake | sunmake}

デフォルト：DEFAULT

この属性は、make ファイルの内容の生成に使用する固有表現を置き換えます。これは、生成後のアプリケーションを Tau 以外のホスト コンピュータに簡単にコンパイルするために必要です。

- DEFAULT は、ホスト OS で「ネイティブ」な make 固有表現を使用します。
  - nmake（Windows ホスト）
  - gmake（Linux ホスト）
  - sunmake（Solaris ホスト）
- gmake は、GNU「gmake」ユーティリティと GNU C/C++ ツール チェーンで定義された make 固有表現を使用するよう指示します。これは、Linux ホストおよび Solaris ホストで使用します。
- sunmake は、SUN ワークショップと ForteC++ ツール チェーンで定義された make 固有表現を使用するよう指示します。
- nmake は、Microsoft「nmake」で使用される固有表現を使用するように指示します。

Target directory

型：String

デフォルト：(空)

この属性は、Makefile ジェネレータで生成される make ファイルを書き出す、ファイル システム内の場所を指定します。絶対パスと相対パスのどちらも使用できます。相対パスを指定する場合、「ルート」は現行のプロジェクト (`.ttp`) ファイルがある場所です。

この属性には空のデフォルト値があります。この場合、ターゲット ディレクトリの名前付けと場所の規則が適用されます。

User code

型：String（複数行）

デフォルト：(空)

この属性は、Makefile ジェネレータで生成された make ファイルに挿入するユーザー コードを指定します。この機能により、ユーザーが使いやすいように生成後の make ファイルの内容をカスタマイズできます。これは、主に公開性を目的に提供されています。

ユーザーが提供するセクションが、生成後の make ファイルの「compiler macro definitions」セクションと「dependencies」セクションの間に挿入されます。

Model Verifier

このステレオタイプは、ベース ステレオタイプ <<build>> を継承し、Model Verifier（シミュレーション、トレースおよび UML レベルでアプリケーションを詳細にデバッグする機能をサポートするアプリケーション）の生成を管理する属性が含まれます。

Additional Preprocessor Defines

この属性は C コード ジェネレータ ステレオタイプのセクションで説明します。Additional Preprocessor Defines を参照してください。

Expand macros

型：Boolean

デフォルト：False

この属性は、C コードを読みやすくして、Model Verifier の実行中に C デバッガを利用しやすくするため、コード生成後に C コードのマクロを拡張して処理するかどうかを制御します。この処理は C コンパイラ ドライバユーティリティで実行されます。

注記

マクロの拡張は、Win32 に設定された Target kind（ターゲットの種類）ではサポートされていません。

Generate environment template functions

型：Boolean

デフォルト：False

この属性は、モデル ベリファイヤ（Model Verifier）のビルド時に、環境関数のスケルトンを持つファイルを作成するかどうかを制御します。

環境へのインターフェイスの最新定義を持つシステム インターフェイス ヘッダー ファイル（`.ifc`）は、この属性の値とは関係なく常に作成されます。

Make template file

型：String

デフォルト：（空）

この属性は、Model Verifier で使用する make ファイルの作成時に、make テンプレート ファイル（ユーザーが提供したファイル）を使用するかどうかを制御します。make テンプレート ファイルにより、生成される C コードにコンパイルおよびリンクされる外部コードをインクルードできます。相対パスは、プロジェクトのディレクトリとの相対パスです。ターゲット ディレクトリ内の生成されたデフォルト名（ビルド ルート名に依存）を持つ .tpm ファイルを使用する場合は「+」記号を入れます。

Suppress C level warnings

この属性については、C コード ジェネレータステレオタイプのセクションで説明しています。

Support C++

型：Boolean

デフォルト：False

この属性は、ビルドされたモデル ベリファイヤ（Model Verifier）に、C++ コンパイラでのコンパイルまたは ISO C コンパイラによるコンパイルを可能にするプロパティを付与します。主要事項のプロパティは、とりわけ、アプリケーション コードとランタイム ライブラリにコンパイルおよびリンクされる外部コードのハンドリングです。

また、この属性は、コードのコンパイルおよびリンク時に `_cpp` 接尾辞を持つ適切なライブラリが使用されるよう制御します。

Target directory

型：String

デフォルト：(空)

この属性は、現在のビルド アーティファクトを使用したビルドの結果として生成されるファイルを書き出す、ファイル システム内の指定を設定します。場所を相対パスで指定する場合、「ルート」は現行のプロジェクト（`.ttp`）ファイルがある場所です。

この属性には空のデフォルト値があります。この場合、ターゲット ディレクトリの名前付けと場所の規則が適用されます。

Target kind

型：`TargetKind`

デフォルト：Win32

この属性は、C コード ジェネレータでビルドするターゲット アプリケーションの種類を制御します。

`TargetKind` で使用可能な値は、ドロップダウン メニューで設定します。Win32、Win32-gcc、Solaris-cc、Solaris-gcc、Linux-gcc のいずれかの値です。

注記

1. Solaris と Linux の値は UNIX 版でのみサポートされます。
2.Win32-gcc の値は『インストール ガイド』に従って GNU C コンパイラをインストールしている場合のみ有効です。また、Win32-gcc は環境関数をまったく含まないモデル ベリファイヤ（Model Verifier）のみサポートします。

Verbose mode

型：Boolean

デフォルト：False

この属性は、モデル ベリファイヤ（Model Verifier）のコード生成時に C コード ジェネレータから詳細レポートと診断をメッセージ出力領域に出力するかどうかを制御します。

参照

Generate environment template functions（環境テンプレート関数の生成）

Make template file （make テンプレートファイル）

第 25 章「C および AgileC ランタイム ライブラリ」の 909 ページ、「サポートされるライブラリ」

objectFile

このステレオタイプは、ソース ファイルと C++ コンパイラで生成されたオブジェクト ファイル間の「make」依存関係を管理するため、ツールの内部で使用されます。また、ターゲット（実行形式ファイルまたはライブラリ）に対する依存関係の定義にも、内部的に使用されます。

ユーザーが参照できるリストを完全にするために取り上げていますが、ユーザーによる使用を意図したものではありません。

このステレオタイプを使用して実際に動作可能な make ファイルを Makefile ジェネレータから作成できるのは、ある特定の場合のみです。IBM Rational から具体的な指示を受けない限り、このステレオタイプの使用は推奨できません。

Source reference

ステレオタイプ **Source reference** は、インポートする定義と要素に関する情報を持つ属性を含みます。

メインのアプリケーションドメインでは、ショートカット メニューの［C/C++ ソースの表示］を選択すると、インポート元の C/C++ 定義に移動できます。C/C++ ソース ファイルとヘッダー ファイルの作業に使用するテキスト エディタで、File（ファイル）属性で指定されたファイルのウィンドウを表示します。

技術的に可能で、テキスト エディタがサポートしていれば、Line（行）および Column（列）属性で定義された行と列にテキスト挿入ポイントが置かれます。

注記

属性の値は、C/C++ のインポート時に指定されます。ユーザーはこれらの値を変更できません。変更した場合、参照にエラーが発生します。

File

型：Charstring

デフォルト：(空)

この属性は、インポート結果として作成される要素の定義が含まれる、C/C++ ヘッダー ファイルを指定します。インポート結果として要素が作成されていない場合、この属性は空です。

Line

型：Integer

デフォルト：(空)

この属性は、インポータ時に C/C++ ヘッダー File（ファイル）のどの行で定義の宣言を検出できるかを指定し、要素のインポート元の C/C++ 定義に関する情報を微調整します。行の開始番号（通常 0 または 1）はプリプロセッサによって異なります。

インポート結果として要素が作成されていない場合、この属性は空です。

Column

型：Integer

デフォルト：(空)

この属性は、インポータ時に C/C++ ヘッダー File（ファイル）のどの列で定義の宣言を検出できるかを指定し、要素のインポート元の C/C++ 定義に関する情報を微調整します。列の開始番号（通常 0 または 1）はプリプロセッサによって異なります。

インポート結果として要素が作成されていない場合、この属性は空です。

staticLibrary

このステレオタイプは、親ステレオタイプ <<library>> を継承し、ファイル アーティファクトを特化して静的にリンクされたライブラリをマニフェストできるようにします。

このステレオタイプは UML パッケージでの使用に適しています。

staticLibrary ステレオタイプを使用するファイル アーティファクトは Makefile ジェネレータで管理され、ステレオタイプを適用する UML パッケージで表示される C++ ソース ファイルに対する依存関係が確立されます。

リンクされた C++ ソースは、以下のようになります。

プラットフォーム	結果ファイル
Windows	`<File name>.LIB` (library)
UNIX	`lib<xxx>.a` (archive)

注記

このステレオタイプは C++ アプリケーション ジェネレータによってビルドされたモデルでのみ使用します。

File name

型：String

デフォルト：(空)

この属性は、ファイル アーティファクトにより指定された静的ライブラリまたはアーカイブを実装する、ファイルのベース名を指定します。

thread

このステレオタイプは、メタクラス アーティファクトを拡張し、アーティファクトを**スレッド アーティファクト**に特化するために使用します。別スレッドでアプリケーションを実行する方法を指定する属性が含まれます。

Instances

型：String（複数行）

デフォルト：(空)

この属性には、独自のスレッドで実行されるインスタンスのリストが含まれます。リストの各要素は、UML で完全修飾されたインスタンス名です。

One thread per instance

型：Boolean

デフォルト：False

この属性は、<<thread>> ステレオタイプにマッピングされた各インスタンスを個別のランタイム スレッドで実行するのか、thread ステレオタイプにマッピングされたインスタンスをすべて同じランタイム スレッドで実行するのかを指定します。

この属性を True に設定すると、各インスタンスに <<thread>> ステレオタイプを作成するのと同じ効果があります。

Priority

型：Integer

デフォルト：(空)

この属性は、スレッドの作成時にスレッドに適用する優先順位を指定します。この属性の値は生成後のコードと、パラメータ、スレッドを生成する OS の呼び出しまで、逐語的に存在します。

この属性が空の場合、デフォルトの優先順位でスレッドが作成されます。

Stack size

型：Integer

デフォルト：(空)

この属性は、スレッドの作成時にスレッドに適用するスタック サイズを指定します。この属性の値は、生成後のコード内、つまりスレッドを生成する OS プリミティブの呼び出しパラメータ内に、指定したとおりに渡されます。

この属性が空の場合、デフォルトのスタック サイズでスレッドが作成されます。

User 1, User 2

型：Integer

デフォルト：(空)

これらの属性により、スレッドの生成時に稼動する OS に対して 2 つまでの整数パラメータを指定どおりに渡すことができます。

これらの属性が空の場合、スレッド作成プリミティブの呼び出しに対応するパラメータのデフォルト値が使用されます。

22

大規模アプリケーション開発のガイドライン

このドキュメントでは、大規模なモデルからアプリケーションを作成、操作、生成するプロジェクトを対象に設計された Tau の機能の活用法について説明します。主に、C コード ジェネレータを使用するプロジェクトに焦点を当てて説明します。

ツールの機能については第 20 章「アプリケーション ビルド リファレンス」を参照してください。

概要

大規模なアプリケーションを設計する際に生じる問題として、次のような例があります。

- 構成管理とバージョン管理の効率を高めるため、モデルを適切なファイルに分割する
- モデルからコードを生成する際のビルド時間を最短化するため、適切なモデル構造を作成する

このドキュメントでは、上記の 2 つの面からアプリケーションの設計について考えます。

ライブラリのビルド

大規模なアプリケーション開発時に共通する戦略は、アプリケーションを異なる複数のモジュールに分割して、さまざまなチームやスタッフが扱えるように、相互に安定したインターフェイスを作ることです。これは、各チームが、担当するモジュールと他のモジュールとのインターフェイスにのみ注意を払えばいいようにするという考えです。他のモジュールの実装については、他のチームが取り組んでおり、彼らの担当以外のモジュールの安定的なビルドに必要なライブラリを提供します。

このアプローチには以下のようないくつかの利点があります。

- 各チームの責任範囲が明確になる
- チーム間のインターフェイスが明確になる
- 各チームは担当のモジュールのみをビルドすればよいので、ビルド時間が短縮される

ライブラリ ビルドを使用する開発の一般的なスキームとは、アプリケーションを一連のモジュールに分割し、個々のモジュールのインターフェイスを明確に定義し、1 つのライブラリにコンパイルできるように設計することです。特定のモジュールの作業を行う場合、現在のモジュールのみを更新し、このモジュールから作成されたライブラリのみをビルドすればよいのです。

また一般に、アプリケーションの初期化を定義する「メイン」モジュールが存在します。これには、`main()` 関数や他の適切な方法を使います。このモジュールにはアプリケーションの起動時に実行されるコードが含まれます。また、各モジュールにテストフレームワークがある場合もあります。これは、基本的にテスト専用に作成された「メイン」モジュールです。

ライブラリ アーティファクト

このスキームは、「ライブラリビルドアーティファクト」のコンセプトに基づく UML でサポートされます。UML の観点からは、ビルド アーティファクトは <<build>> ステレオタイプを継承したステレオタイプでステレオタイプ化されたアーティファクトです。この典型的な例は、<<'C Advanced Application'>>、<<'ModelVerifier'>> または、<<'C++ Application'>> アーティファクトです。

ライブラリ ビルド アーティファクトは、'Executable'（実行形式ファイル）ではなく、'Library'（ライブラリ）の 'Target'（ターゲット）プロパティ セットを含むビルド アーティファクトです。つまり、アーティファクトのビルド時に作成されたターゲット ファイルは、実行形式ファイルではなく、他のライブラリとリンクさせて実行形式ファイルを作るためのライブラリです。

注記

ライブラリ ビルド アーティファクトはすべて同じタイプでなければなりません。1 つのプロジェクトで異なるビルド タイプを組み合わせることはできません。

ライブラリ アーティファクトは、ライブラリに含むべきモデル要素を <<manifest>> するものでなければなりません。現時点では、**特定のパッケージ**のライブラリにすべての要素が含まれる場合のみがサポートされます。この要素は、ライブラリ ビルド アーティファクトからパッケージ自体への <<manifest>> 依存関係を適用することで、表現されます。これは、C コード ジェネレータの使用時にサポートされる唯一のライブラリの作成方法です。

注記

ライブラリ アーティファクトでマニフェストされたパッケージは、最上位 パッケージである必要があります。

ライブラリ ビルド アーティファクトに基づくビルドを実行すると、ライブラリのみが作成されます。完全な実行形式ファイルを作成するためには、「メイン」モジュールをビルドする必要があります。UML においては、「メイン」モジュールは「実行形式ファイル」への「ターゲット」セットをもつビルドアーティファクトによって表現されます。アプリケーションの起動時に作成すべきクラスを特定するためには、ビルド アーティファクトからこのクラスへの <<manifest>> 依存関係を使用します。

また、ライブラリ間の関係についてのツール情報を提供するため、ビルド アーティファクト間の <<include>> 依存関係を指定する必要があります。

注記

現時点では、<<include>> 依存関係をモデルから手動で導き出し、必要な場所に明示的に追加しなければなりません。

下図では、ここで説明したコンセプトを例示しています。この例では 2 つのメインライブラリ（Pkg1 と Pkg2）で構成されるアプリケーションを想定します。各メインライブラリに一連の定義を持つパッケージが 1 つ含まれます。このモデルにはライブラリ（`Interfaces`）が 1 つ含まれます。このライブラリには、共通インターフェイス、シグナルなどが含まれます。そしてさらにメイン・モジュール (`LibraryExample`）が含まれます。

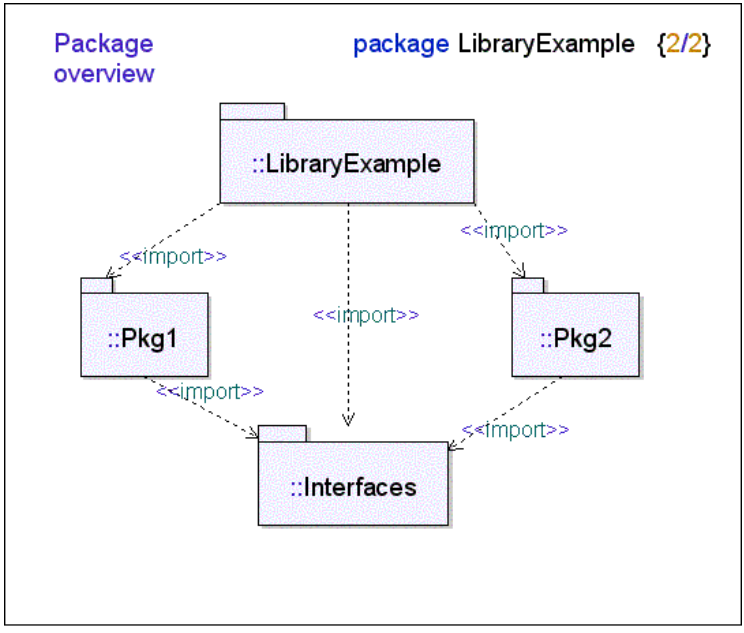


図 234: パッケージの概要（例）

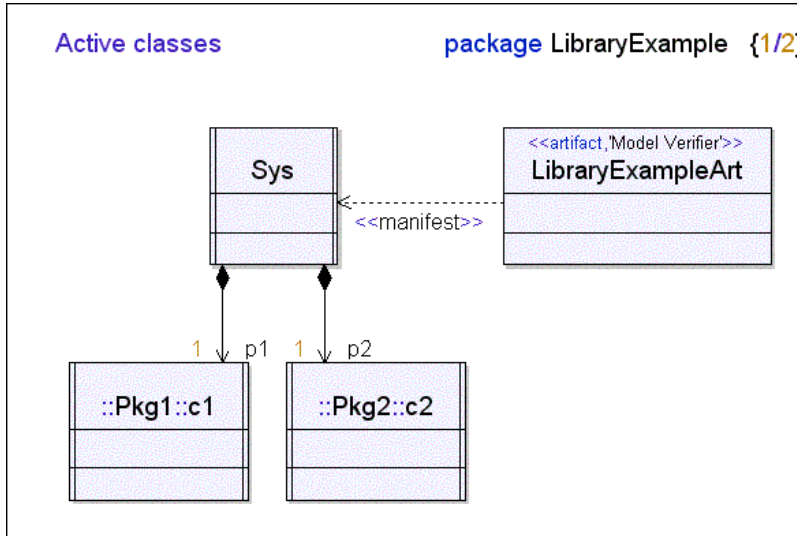


図 235: システムの概要

「Interfaces」のパッケージは以下のように定義されています。

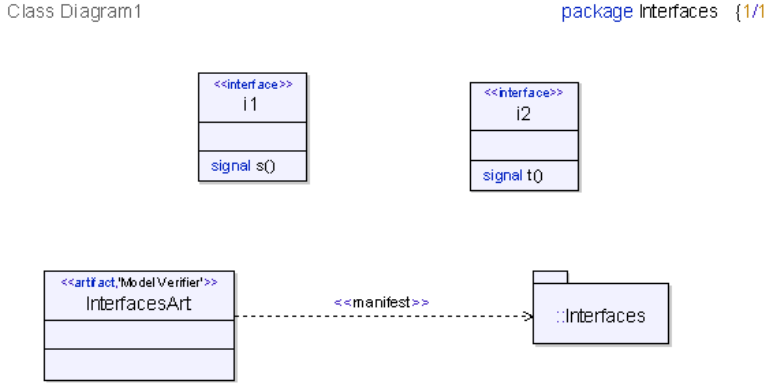


図 236: パッケージ「Interfaces」の内容

2 つのインターフェイス「i1」と「i2」があります。また、パッケージをマニフェストする **Model Verifier** ビルド アーティファクト インターフェイス があります。このアーティファクトのプロパティをチェックする必要がある場合、ライブラリのターゲット プロパティ セットを確認します。

870 ページの図 237 に示すように「Pkg1」が定義されます。

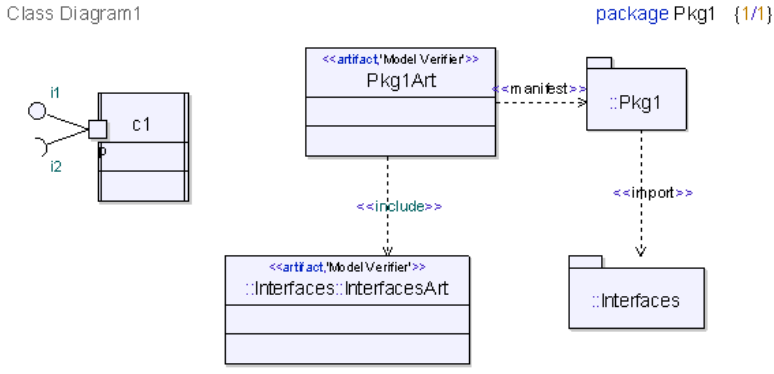


図 237: パッケージ「Pkg1」の内容

`Pkg1` には、アクティブ クラス「c1」と <<'Model Verifier'>> ビルド アーティファクト「`Pkg1Art`」が 1 つずつ含まれます。「c1」クラスは「`i1`」インターフェイスを実現化し、「`i2`」インターフェイスを要求します。これらのインターフェイスは「`Interfaces`」パッケージで定義されることから、この定義にアクセスするためには、「Pkg1」から「`Interfaces`」への <<import>> 依存関係が必要です。また、「`Pkg1Art`」が「`InterfacesArt`」に依存していることを示すために、「`Pkg1Art`」から「`InterfacesArt`」への <<include>> 依存関係が必要です。

「`Pkg2`」は、「`Pkg1`」によく似ています。違いは 871 ページの図 238 に示すよう他のクラスを含む点です。

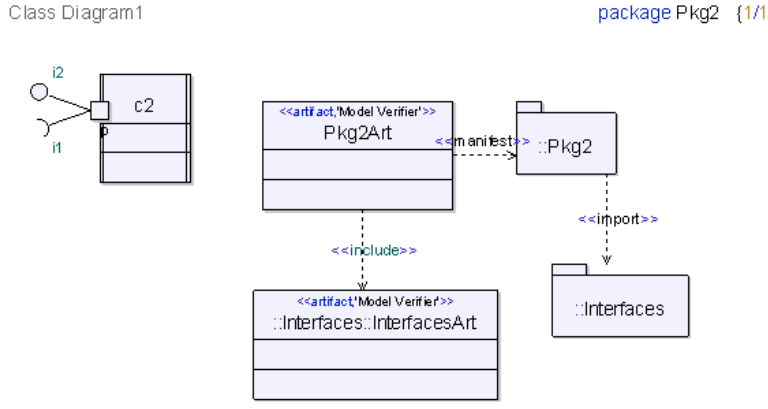


図 238: パッケージ「Pkg2」の内容

871 ページの図 239 に、「LibraryBuildExample」アーティファクトとライブラリアーティファクトの依存関係を示します。

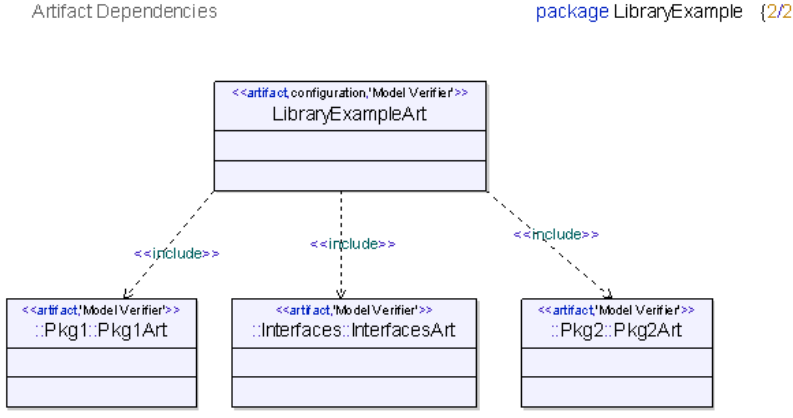


図 239: アーティファクト依存関係

この例について特筆すべき点：

- 「Interfaces」パッケージは、アクティブ クラスを含むモジュールに必要な共通定義を含むライブラリ モジュールとして使用されます。

- 一般に、すべてのパッケージは個別のファイルに保存されます。このため、UML ファイルは 4 つになります（パッケージに 1 つずつ、つまり「Interfaces」、「Pkg1」、「Pkg2」、「LibraryExample」の 4 つ）。
- 「メイン」モジュールのあるパッケージには、ライブラリ ビルド アーティファクトはありません。
- すべてのビルド アーティファクトには、同じターゲット ディレクトリを使用します。872 ページの「ビルド プロセス」を参照してください。

実装とシグニチャ ファイル

ライブラリ コンセプトの利益を享受するためには、モジュール間に明確に定義されたインターフェイスがあり、このインターフェイスをモジュールの実装から分離できることが重要です。UML では、1 つのモジュールインターフェイスは 1 つのパッケージにより記述されます。このパッケージには、一組のクラスの型定義とともに、クラス定義のパブリック使用の部分で使用される一組のインターフェイス、シグナル、データ型などの型定義が含まれます。実際には、この種のパッケージを別々のファイルに保存して、パッケージのバージョン管理をモデルの他の部分のバージョン管理から切り離すと便利です。Tau では、［モデルビュー］でパッケージ上のショートカットメニューから［新しいファイルで保存］コマンドを実行して、たとえばパッケージを別のファイルに保存できます。

このソリューションの 2 つ目の特徴は、パッケージで定義されたクラスの実装面を 1 つまたは複数のファイルに保存して、このファイルのバージョンを、モジュールインターフェイスを含むファイルとは独立に扱えるという点です。実際には「実装面」は 2 つの部分から構成されます。

- 操作の振る舞いを定義する、状態機械および操作本体
- プライベート属性

Tau では、状態機械の実装や操作本体を別のファイルに移動できます。このためにはモデルビュー内の実装本体を［モデルビュー］のファイルフォルダに表示されるファイルの 1 つにドラッグします。または、パッケージと同様に、［モデルビュー］内の実装で［新しいファイルで保存］コマンドを選択できます。

前のセクションの例においては、通常は、状態機械の実装である c1 と c2 を、それぞれ別のファイルに保存することになるでしょう。

ビルド プロセス

完全なアプリケーションをビルドするために、まず個々のライブラリをビルドしてから、メイン モジュールをビルドします。ただし、ビルドの前に必ず、共通ディレクトリ（たとえば、「mytargetdir」）の複数のビルド アーティファクトにターゲット ディレクトリ プロパティを設定しなければなりません。この設定により、C コンパイラはリンクのフェーズにおいてライブラリを見つけることができるようになります。

上記の例では、4 つのビルドコマンドを与えます。まず「Interfaces」、「Pkg1」、「Pkg2」の 3 モジュールのライブラリのビルド、そして最後に「LibraryExampleArt」アーティファクトに記述された実行形式のビルドです。

注記

ライブラリは下から上の順番（「ボトムアップ」オーダー）でビルドする必要があります。つまり、ある特定のパッケージをビルドする前に、このパッケージが依存するすべてのパッケージをビルドしなければなりません。

C コードの制限事項

ライブラリ ベースのビルドには多くの利点があります。ただし、C コード ジェネレータでサポートされるライブラリ ビルドを活用するために理解する必要のある制限事項がいくつかあります。これらの制限事項は以下のとおりです。

コンテナ タイプとライブラリ ビルド

クラスなどのエンティティのコードを生成する場合、C コード ジェネレータは、このクラスのためにユーティリティ タイプをいくつか生成します。たとえばクラスへのポインタのコード、およびクラスの明示的でないコレクション タイプに対応するコード（モデルがクラスを多重度 >1 の属性として持つ場合に使用）があります。また、テンプレート コンテナ タイプがパラメータとしてのクラスとともにインスタンス化されるときに使用されるタイプが含まれることもあります（たとえば、クラスの「Bag」または「Array」が属性のタイプとして使用される場合）。

ライブラリ ビルドが使用されない場合、コード生成時にクラスの使用方法のグローバル分析が実行されます。また、テンプレートのインスタンス化がある場合、これに対応するタイプが C コードに生成されます。しかし、ライブラリ ビルドが使用された場合は、ライブラリ モジュールのコード生成時にインスタンス化をすべて明確にすることはできません。このため、クラスがどのように使用されているかをグローバル分析することはできません。

したがって、コードは以下のケースに対してのみ生成されます。

- クラスへのポインタ
- 文字列のインスタンス化

つまり、多重度 1 と多重度 >1 の両方の属性の型としてクラスを使用できます。また、クラスの String も使用できます。しかし、パラメータとしてのクラスでテンプレート コレクションタイプのインスタンス化を実行する他のやり方は、エラーとなります。これは、C の型に対応する C コードが生成されないためです。

ただし、特定のテンプレート タイプのインスタンス化については、対応する C コードをツールから簡単に生成する方法があります。必要なテンプレート インスタンス化を含み、かつ、<<CollectionType>> でステレオタイプ化されたシンタイプを定義すればよいのです。たとえば、クラス C を定義して、属性などのタイプとして `Bag<C>` を使用する場合、モデルのどこかにシンタイプ定義「<<CollectionType>> `syntype Bag_C = Bag<C>;`」を定義します。通常、このシンタイプの定義に最適な場所は、クラス C が定義されている場所ですが、`Bag<C>` 定義が使用されている場所から見えるモデルの中であれば、どこでも構いません。

このセクションではクラスについて説明しましたが、コレクション タイプの制限事項はデータ型にも適用されます。

再帰的なパッケージのインポートとアクセス

C コード ジェネレータには、<<import>> 依存関係と <<access>> 依存関係の使用方法に関する制限事項があります。<<import>> 依存関係と <<access>> 依存関係を再帰的に使用することはできません。この規則はライブラリ ビルドが使用されている場合も、使用されていない場合も適用されます。

ライブラリ パッケージとアクティブ クラスの継承

アクティブクラスを含むライブラリパッケージがあり、そのアクティブクラスを、たとえば、他のライブラリパッケージなどで使用したいとしても、そのクラスに対して可能なのはインスタンス化だけです。元のクラスを継承する別のクラスは定義できません。このため、互いに継承し合うアクティブ クラスはすべて、同じライブラリ パッケージで定義されなければなりません。

この理由は、継承のためのコードを生成するには、継承されたクラスの状態と遷移の詳細を知る必要がありますが、通常、このような情報はクラスの実装パートに隠れているためです。

<<noScope>> パッケージを使用するファイル サイズの管理

大規模モデル開発では、大規模なチームによる 1 つのモデルに対する同時並行的な作業をサポートできることが重要です。このような状況をサポートするために、Tau では、複数のファイルにモデルを保存して、異なるチーム メンバーがそれぞれのファイルで作業できるようにしました。ファイル分割に使用される最も一般的な方法は、複数のファイルにパッケージを保存すること、複数のファイルにクラスなどを実装することです。しかし、他の仕組みの方が有益な場合もあります。モデルを複数のファイルに分割するやり方などです。このセクションでは <<noScope>> パッケージに基づくファイル管理の方法について説明します。

注記

<<noScope>> パッケージは単一のスコープ単位を構成しないため、<<noScope>> パッケージに対して、または <<noScope>> パッケージからの <<import>> 依存関係と <<access>> 依存関係は作れません。

ビルド パフォーマンスの改善

<<bindByGuid>> パッケージ

モデルに基づいたコードを生成する場合、コード生成は複数のフェーズで構成されます。主要なフェーズの 1 つに名前解決のフェーズがあります。このフェーズでモデル内の参照がすべて 解決されます。Tau は、モデル要素に対して生成された一意の識別子に基づくモード、または、要素とスコープ階層の位置の名前と識別子に基づくモードの 2 通りで名前解決ができます。通常、Tau は、名前の解決時にこの両方の仕組みを使用して、名前の不一致を検出して、名前の変更を反映させます。ただし、この設定はモデルのロードに必要な時間を増大します。このため、この設定はほとんど変更のないライブラリ パッケージには不要です。

<<bindByGuid>> ステレオタイプをパッケージに適用する場合、パッケージのエンティティの名前による解決を避けて、一意の識別子による解決のみ実行するよう設定できます。この設定は、パッケージの要素とのやりとりにまったく影響を与えません。このため、名前変更の反映やその他の類似機能は問題なく動作します。ファイルからパッケージをロードする場合にのみ影響があります。モデルの C コード生成時にも、まったく同じように設定しているので、状況によっては、C コード ジェネレーションのパフォーマンスを改善する効果的な方法となります。

23

生成されたコード内での要件のトレーサビリティ

Tau の C、C++、Java コード ジェネレータは、DOORS に記述された要件を参照したコード内注釈の生成をサポートします。この機能によって、生成された C、C++、Java コード内の定義から関連する要件へのトレーサビリティが実現します。

本章では、この機能の使い方について説明します。

概要

複雑なアプリケーション開発においては要件の管理と分析が成功への鍵となります。要件の仕様が不明瞭であったり、分析が不十分であった場合、開発したアプリケーションが結果的に要求を満たさない可能性が高くなります。

プロジェクトの要件分析の段階では、DOORS などの要件管理 / 分析ツールが力を発揮します。Tau を使用してモデリングと実装を行う設計段階に移行した際にも、設計された要素とその要件との対応付けを明確化できます。この対応付けをトレーサビリティ リンクといい、DOORS と Tau のインテグレーション機能を使用して実現します。インテグレーション機能の詳細については、DOORS との協調を参照してください。

アプリケーション開発において、Tau のモデルと DOORS の要件の間のトレーサビリティ リンクは、主としてナビゲーションのために使用します。モデルから C、C++、Java コードを生成する場合には、このナビゲーション情報を、生成するコードに DOORS 要件を参照する「注釈」として入れ込むことができます。生成コードにこのような注釈を付けることで、以下のような問いに対して容易に答えることができます。

- 生成されたアプリケーションはすべての要件を実装しているか。
- 新しく追加された要件が生成コードにどのような影響を及ぼすか。
- ある 1 つの C、C++、Java 定義が複数の要件が実現化しているかどうか。
- 1 つの要件の実装が 1 つのソース ファイルに限定されるか、複数のファイルにまたがるか。
- 1 つの要件の実装が 1 つのモジュール（ライブラリまたは実行ファイル）に限定されるか、アプリケーション全体にまたがるか。
- 要件を削除すると、生成コードにどのような影響があるか。

生成コード内に要件への参照があることで、生成コードはより読みやすくなります。また、特定の標準や認証プログラムに則したコード検証を行う場合には、こういった要件参照が求められるケースもあります。

U2ReqTrace アドイン

生成コードに要件を参照する注釈を付けるには、U2ReqTrace という名前の Tau アドインを使用します。

このアドインの使用手順は以下のとおりです。

1. DOORS で要件を定義し、1 つまたは複数の DOORS フォーマル モジュールに保存します。
2. UML モデルを設計して実装する際に、これらの DOORS フォーマル モジュールを Tau にインポートします。この作業の詳細については、「要求のインポート」を参照してください。
3. UML モデル要素と DOORS 要件の間にトレーサビリティ リンクを設定します。この作業の詳細については、「要求の関係」を参照してください。

4. U2ReqTrace アドインを起動します。アドインの起動方法については、アドインの起動を参照してください。
5. ビルド アーティファクトに `<<requirementRefAnnotation>>` ステレオタイプを適用します。プロパティ エディタを使用して、このステレオタイプのタグ付き値として適切なオプションを設定します。
6. C、C++、Java アプリケーションを生成します。トレーサビリティ リンクをもつ UML モデル定義に対応した C/C++ コード定義に、コメントの形式で注釈が付けられます。この注釈には、DOORS 要件への参照のほかに、要件テキストも含まれます。DOORS 要件からどの情報を取得して表示させるかは、U2ReqTrace のオプションによって変更できます。

実際は、上記の手順を繰り返し実行します。後で新しい要件を追加する場合は、DOORS モジュールを再インポートして、新しい要件と UML モデル要素にトレーサビリティ リンクを追加できます。その後、コードを（必要に応じて全部または一部）再生成できます。

注釈のフォーマット

生成コードには、コメントの形式で注釈が付けられます。デフォルトでは、これらの注釈は、要件への参照を持つ C/C++/Java 定義の直前に表示されます。すべての注釈をソース ファイルの先頭、または末尾に生成することもできます（詳細については、オプションを参照してください）。

注釈の形式は、コメントをどこに生成するかによって異なります。定義の直前に生成される注釈の場合、以下のようになります。

```
/*** Realizes requirement /<DB>/<P>/<M>#<ID>
    <Req Heading>
    "<Req Short Text>"
    "<Req Object Text>"
************************************************************/
```

ファイルの先頭または末尾に生成される注釈の場合、以下のようになります。

```
/**** Requirements realized by definitions in this file ****
************************************************************
<Def Name>(<Line>) : /<DB>/<P>/<M>#<ID>
    <Req Heading>
    "<Req Short Text>"
    "<Req Object Text>"
************************************************************/
```

C++、Java コードを生成する場合は、C のコメントに代わって、C++、Java のコメント（//）が使用されます。

上記で使用されている「変数」の意味は以下のとおりです。

<DB>	DOORS データベースの名前
<P>	DOORS フォーマル モジュールへのパス
<M>	DOORS フォーマル モジュールの名前

<ID>	要件の ID
<Req Heading>	要件の見出しテキスト
<Req Short Text>	要件のショート テキスト
<Req Object Text>	要件のオブジェクト テキスト
<Def Name>	注釈付き定義の名前
<Line>	生成した定義のファイル内の場所を示す行番号

例 **341:** 要件の注釈コメントの形式 ——————————————————————

注釈を付ける定義の直前に表示される注釈は以下のようになります。

```
//// Realizes requirement /DOORS
Database//Tau2_7_Requirements/U2ReqTrace#6
//  Support both C and C++ code generators
//  "Requirement traceability should be supported both for C and
C++ code generators"
////////////////////////////////////////////////////////////
class U2ReqTracer {};
```

ファイルの先頭または最後に表示される注釈は次のようになります。

```
///// Requirements realized by definitions in this file ////
////////////////////////////////////////////////////////////
//U2ReqTracer(12) : /DOORS
Database//Tau2_7_Requirements/U2ReqTrace#6
//  Support both C and C++ code generators
//  "Requirement traceability should be supported both for C and
C++ code generators"
////////////////////////////////////////////////////////////
```

——————————————————————————————————————

注記

ラウンドトリッピングを使用する C++ ビルド アーティファクトに U2ReqTrace アドインを使用した場合、注釈は GENERATED タグ内に表示されます。これは、これらのコメントに含まれる情報が UML モデルのどれとも対応していないからです。この仕組みによって、注釈付きファイルでもラウンドトリッピングができます。

オプション

コード生成時に使用するビルド アーティファクトに `<<requirementRefAnnotation>>` ステレオタイプを適用した場合、生成したコードには要件参照の注釈が付きます。このステレオタイプは、U2ReqTrace アドインを起動したときにロードされる ReqTraceability プロファイルパッケージで定義されています。

プロパティ エディタなどを使用して、`<<requirementRefAnnotation>>` ステレオタイプのタグ付き値として、以下のオプションを設定できます。

- **Annotate Code**
  コードに注釈を付ける場合は、このオプションを有効にする必要があります。このオプションを無効にした場合、その他のオプションの効果がなくなります。他のオプションの値を維持したまま、要件参照の表示を一時的に無効にしたい場合、このオプションが有用です。
- **Text Size Limit**
  このオプションにより、すべての DOORS 要件テキストの最大表示文字数を指定します。デフォルトでは、上限は 255 文字に設定されます。すべてのテキストを表示するには、このオプションを「0」に設定します。
- **File Placement**
  このオプションにより、生成したファイル内のどこに要件参照を表示するかを指定できます。デフォルト値は［Before Definition］、要件参照は対応する C/C++/Java 定義の直前に表示されます。このオプションを［File Header］（すべての要件をファイルの先頭に表示する）、または［File Footer］（すべての要件をファイルの最後に表示する）に設定することもできます。表示される注釈のフォーマットはこのオプションによって決まります。詳細および例については、注釈のフォーマットを参照してください。
- **Print Headings**
  このオプションにより、要件の見出しを表示するかどうかを指定します。デフォルトでは有効に設定されています。
- **Print Short Text**
  このオプションにより、要件の概略（ショートテキスト）を表示するかどうかを指定します。デフォルトでは有効に設定されています。
- **Print Object Text**
  このオプションは、要件の完全な説明（オブジェクトテキスト）を表示するかどうかを指定します。デフォルトでは有効に設定されています。

利用法

U2ReqTrace アドインは、ユーザ インターフェイスに特定のコマンドを提供しません。また、U2ReqTrace アドインは C、C++、Java コード ジェネレータに統合されており、C/C++/Java のコード生成後に自動的に起動されます。より正確には、コード ジェネレータが生成したコード生成結果パッケージが挿入された後です。

U2ReqTrace アドインによって生成コードに注釈が付けられると、ビルド ログに以下のメッセージが出力されます。

```
Successfully annotated "C:¥CGTest¥MyFile.h" with requirements
references.
```

エラーが発生した場合にも、ビルド ログにメッセージとして出力されます。

要件情報を抽出するためには、U2ReqTrace アドインは DOORS に接続されている必要があります。コード生成時に DOORS が起動されていない場合、DOORS は自動的に起動されます。

API アクセス

U2ReqTrace アドインはエージェントとして実装されます。したがって、InvokeAgent API メソッドを使用して、パブリック Tau アプリケーションからアクセスできます。このエージェントは ReqTraceability プロファイルで定義されており、`AnnotateGeneratedCodeWithRequirementsReferences` といいます。

また、U2ReqTrace をカスタムメイドのコード ジェネレータト統合することもできます。プロファイルと同様、エージェントも Insert cross reference file ツール イベントでトリガされます。したがって、このエージェントは、C、C++、Java コード ジェネレータと同じ形式で相互参照ファイル（別名、コード生成結果パッケージ）を作成する、すべてのコード ジェネレータに対して機能します。U2ReqTrace は、結果パッケージの内容を分析し、要件へのトレーサビリティ リンクを持つ定義について DOORS 情報を抽出し、その抽出情報をソース コードへの コメントとして生成コードに追加し、必要に応じて、結果パッケージ内の情報を更新します（挿入されたコメントのために行番号の調整が必要になることがあります)。

UML による C コードの生成

「UML による C コードの生成」セクションの各章では C コード ジェネレータと AgileC コード ジェネレータを使用して UML プロジェクトを C コード アプリケーションに変換する方法について説明しています。

24

C アプリケーションのための環境関数

このセクションでは、C コード ジェネレータで生成したアプリケーション コードとシステムの環境間のインターフェイスを設計する手順について説明します。

C アプリケーションのアーキテクチャの概要と、環境とのインターフェイスの方法について解説します。

次に、環境関数のガイドラインと推奨設計について説明します。望ましい結果が得られるように関数をアプリケーション コードにインクルードする方法のガイドラインとヒントを示すことが目的です。

参照

C コード ジェネレータの動作原理についての参考情報は、以下の章を参照してください。

第 27 章「C コード ジェネレータ リファレンス」

第 28 章「C コード ジェネレータ ランタイム モデル」

第 29 章「C コード ジェネレータ シンボル テーブル」

第 30 章「C コード ジェネレータマクロ」

概要

C コード ジェネレータを使用して生成したアプリケーションは、以下の 3 つの部分から構成されていると考えることができます。

- システムを実装する生成コード

- システムの物理的な外部環境
- システムとシステムの外部環境とを接続する環境関数

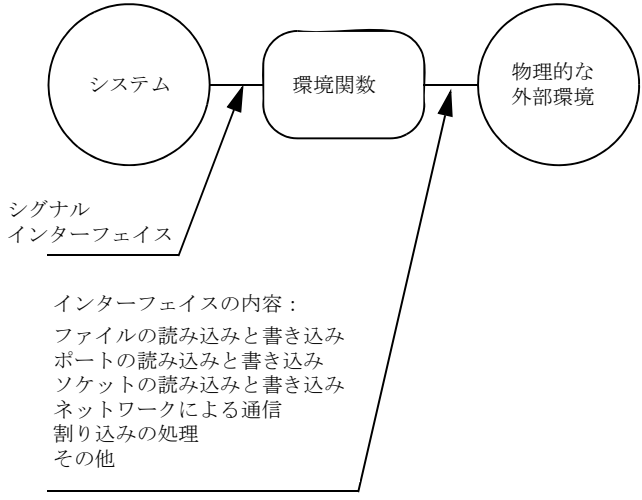


図 240: アプリケーションと環境のインターフェイス

生成コード

システムは状態機械として動作します。つまり、遷移は優先度順に実行され、シグナルはアクティブ クラスのインスタンスから別のインスタンスへ送信され、新たな遷移が開始され、タイマー シグナルが送信される、などの動作をします。これらは、システムの実行に影響する内部動作の例です。

システムは、シグナルの送受信によって環境と通信します。

物理的な外部環境

アプリケーションの物理的な外部環境は、オペレーティング システム、ファイル システム、ハードウェア デバイス、コンピュータのネットワークなどから構成されると考えられます。この「現実世界」の環境では、シグナル送信以外の動作が必要となります。アプリケーションが実行する必要のある動作の例としては、ファイルの読み込みと書き込み、ネットワークを通じたデータ送受信、割り込みシグナルへの応答、ハードウェア ポートやソケットに対する情報の読み込みと書き込みなどがあります。

環境関数

環境関数とは、システムと物理的な外部環境という 2 つの世界が接点です。この関数では、システムから外部環境に送信されたシグナルによって物理的な環境でのあらゆる種類のイベントが引き起こされ、同時に環境のイベントによってシグナルがシステムに送信されます。

C コード ジェネレータには物理的な外部環境や実行すべき動作に関する知識がないため、このような環境関数を指定する必要があります。

分散アプリケーション

分散システムでは、アプリケーションは複数の通信システムから構成される可能性があります。この場合、各システムが実行形式プログラムに対応します。システムは、他のオペレーティング システム タスクと通信するオペレーティング システム タスクとして実行することも、専用のプロセッサでネットワークを介して他のプロセッサと通信して実行することも可能です。さらにこれらの組み合わせも考えられます。ここでは理解のしやすさのため、このオペレーティング システム タスクやプロセッサを、ネットワークを通じて通信する「ノード」と呼びます。通信するオペレーティング システム（OS）タスクがある場合、OS から提供されるプロセス間通信の手段はネットワークです。

C コード ジェネレータを使用して、ネットワークを通じて通信する複数のノードから構成されるアプリケーションを生成できます。ただしこの場合、環境関数にノード間の通信を実装する必要があります。

注記

ネットワーク中のすべてのノードが UML モデルから C コード ジェネレータで生成されたプログラムである必要はありません。ノードと他のノードと通信さえできれば、どのような技術によって実装されていても構いません。

複数の通信システムを含む分散世界では、Pid 値（アクティブ クラスのインスタンスの参照）が問題になります。たとえば、送信側が別のシステムにあるアクティブ クラスのインスタンスを参照しても「Sender.Output」が正しく動作することが要求されます。この問題に対処するため、Pid 値のコンポーネントとしてグローバル ノード番号が導入されています。グローバル ノード番号は各ノードに割り当てられた一意の整数値で、現在のインスタンスがあるノードを特定します。Pid 値のもう一方のコンポーネントはノード内のインスタンスのローカル識別子です。

その他の利点

アプリケーションをシステムと環境変数に分割することには他にも利点があります。動作を実行する論理分岐（環境にシグナルを送信する分岐など）が、動作の実装詳細（環境関数でのシグナルの処理など）から分離されます。

このような分離により、問題の複雑さが軽減され、個別のテストも可能になります。さらに、論理的な振る舞い（システムとして実装）と外部環境へのインターフェイス（環境関数として実装）の並行開発が可能になります。システムと外部環境の間のシグナル インターフェイスが定義されると、両方の並行開発が可能となり最終的に結果を簡単に統合できます。

アプリケーションのビルド

アプリケーションのビルド方法、すなわち UML モデルからコードを生成し、適切なコード ジェネレータ設定やコンパイラ オプションなどを使用してコードをコンパイルおよびリンクする方法については、第 20 章「アプリケーション ビルド リファレンス」で説明しています。

アプリケーションのシミュレーションとデバッグ

アプリケーションの開発時、UML モデルから生成されたコードの振る舞いが想定どおりであることを検証するため、まず外部環境を使わず、ホスト コンピュータ上でモデル ベリファイヤ（Model Verifier）を使用して UML のシミュレーションとデバッグを行うのが適切です。このようなシミュレーション モードでは、外部環境はモデル ベリファイヤ（Model Verifier）ユーザー インターフェイスによって提供されます。この手法を使うと、外部環境からシグナルを受信でき、環境に送信されるシグナルの確認とトレースが可能になります。

ターゲット アプリケーション

次のステップで、アプリケーションと外部環境のインターフェイスに必要なサポート機能をインクルードし、ターゲット環境に合わせてアプリケーションのコンパイルとリンクを行います。

参照

第 25 章「C および AgileC ランタイム ライブラリ」

生成 C コードに関する基本事項

このセクションでは、生成 C コードにおけるアクティブ クラスのインスタンスとシグナルの表現について説明します。システムの静的構造の表現であるシンボル テーブルについても、生成コードと環境関数をインターフェイスする作業をどのように進めるかを理解するために、説明します。このセクションの情報は、後で環境関数を説明する際に使用します。

生成 C コードに関する詳細については、第 28 章「C コード ジェネレータ ランタイム モデル」および第 29 章「C コード ジェネレータ シンボル テーブル」を参照してください。

シグナルを表すデータ型

シグナルは、C 構造体で表現されます。この構造体は、シグナルに関する一般情報と、シグナルによって伝達されるパラメータを要素として持ちます。

パラメータを持たないシグナル、およびそのシグナルのポインタ、`xSignalNode` の一般型定義 `xSignalRec` とを以下に示します。これらのデータ型は、`scttypes.h` ファイルにあります。このデータ型によって、あらゆる信号のデータ型をキャストし、一般の要素にアクセスできます。

```
typedef struct xSignalRec *xSignalNode;
typedef struct xSignalRec {
  xSignalNode Pre;
  xSignalNode Suc;
  SDL_PId     Receiver;
  SDL_PId     Sender;
  xIdNode     NameNode;
  int         Prio;
} xSignalRec;
```

xSignalRec には、以下のコンポーネントが含まれます。

- Pre および Suc：これらのコンポーネントは、アクティブ クラスの受信インスタンスの入力ポート リスト内にあるシグナルをリンクするのに使用します。入力ポートは二重にリンクされたリストとして実装されます。シグナルが処理され、シグナルに含まれている情報が不要になると、シグナルは使用可能メモリ リストに戻され、将来のシグナル送信で再利用できるようになります。Suc コンポーネントはシグナルを使用可能メモリ リストにリンクするのに使用し、Pre コンポーネントはシグナルが使用可能メモリ リストにある限り (xSignalNode)0 です。
- Receiver：受信インスタンスです。
- Sender：送信インスタンスです。
- NameNode：このコンポーネントは、シグナル型を表すシンボル テーブル内のノードへのポインタです。シンボル テーブルはシステムに関する情報を持つツリーで、特にシステムで定義されたシグナル型ごとに対応する 1 つのノードが定義されています。
- Prio：シグナルの優先度を表します。

生成コードには、以下の例に示すように、シグナルのパラメータを表すデータ型が入ります。

例 **342:** シグナル定義に対して生成された **C** コード ——————————————

以下のシグナル定義を考えます。

```
signal S1(Integer);
signal S2;
signal S3(Integer, Boolean, OwnType);
```

この場合、次の C コードが生成されます。

```
typedef struct {
  SIGNAL_VARS
  SDL_Integer Param1;
} ySignalPar_z0f_S1;
typedef ySignalPar_z0f_S1 *yPDP_z0f_S1;

typedef struct {
  SIGNAL_VARS
  SDL_Integer Param1;
  SDL_Boolean Param2;
  z09_OwnType Param3;
} ySignalPar_z0h_S3;
typedef ySignalPar_z0h_S3 *yPDP_z0h_S3;
```

SIGNAL_VARS は scttypes.h で定義された C マクロで、シグナル構造体内の共通コンポーネントとして展開されます。

注記

シグナル S2 には定義がありませんが、パラメータを持たないシグナルにはデータ型が生成されないので正しい動作といえます。

パラメータを持つシグナル 1 つごとに、構造体とポインタの 2 つのデータ型が生成されます。

構造体型には、シグナル定義内のパラメータごとにコンポーネントが 1 つ生成されています。コンポーネントには Param1、Param2 などの名前が付けられます。コンポーネントは、シグナル定義でのパラメータの順序と同じ順序で構造体に配置されます。

例 **343:** シグナルパラメータが構造体の場合の生成 **C** コード

```
<<struct>> class MyStruct {
  Integer x;
  Integer y;
}

signal MySig1( MyStruct );
```

生成 C コード :

```
typedef struct z_paraext_H0_MyStruct_s {
  SDL_Integer x;
  SDL_Integer y;
  z09_OwnType Param3;
} z_paraext_H0_MyStruct;

typedef struct z_paraext_H0_MyStruct_s
*z_paraext_H1_ptr_MyStruct;

typedef struct {
  SIGNAL_VARS
  z_paraext_H1_ptr_MyStruct Param1; /* Note: this is a
pointer to structure */
} ySignalPar_z_paraext_S2_MySig1;

typedef ySignalPar_z_paraext_S2_MySig1
*yPDP_z_paraext_S2_MySig1;
```

デフォルトでは、UML ではすべての構造体とクラスは参照集約をもち、シグナル MySigl は構造体 MyStruct への参照（生成コードではポインタとなる）を含みます。この使用が当てはまる場合で、参照がどうしても必要な場合は、環境からこのようなシグナルを送信するときに、シグナルパラメータ用のメモリ割り当てと割り当て解除は、ユーザーが手作業で取り扱う必要があります。

MyStruct パラメータを値で渡したい場合、シグナルパラメータは、以下のような合成集約で定義する必要があります。

signal MySig1(part MyStruct);

これは、テキスト図で手作業で行うか、シグナルパラメータのプロパティメニューの［集約］ドロップダウンリストから “Composition (part)” を選択します。

C 言語における UML データ型の表現

UML データ型の C 言語での表現のリファレンスについては、942 ページの「データ型の解釈」セクションを参照してください。

シグナル パラメータのエンコードとデコード

パラメータを含むシグナルの自動処理（エンコーダやデコーダなど）を実装するには、シグナル中のデータの配置に関するランタイム情報が必要です。C コード ジェネレータは、`Set-SDL-Coder` が C コード ジェネレータの Advanced options で指定されていれば、この情報を生成します。

このオプションをオンにすると、C コード ジェネレータによって `<basename>_cod.h` と `<basename>_cod.c` の 2 つのファイルが生成されます。さらに生成された make ファイルを自動的に更新して、`<basename>_cod.c` ファイルのコンパイルとリンクをインクルードします。

注記

Tau にあるエンコーダ ライブラリは、エンコーダやデコーダのフレームワークとしての使用がいっさいサポートされていません。

インターフェイス外のシグナルの表現

最上位パッケージ直下または下位のパッケージ直下に定義されたシグナルがある場合、この定義と、同じパッケージ内または下位のパッケージ内のあるインターフェイス内で定義された同名のシグナルとは、生成コードでは同じものを表します。

パッケージにこのような共通の定義となるシグナル定義がない場合は、異なるインターフェイス内に定義された同名のシグナルは生成コードでは別のシグナルとして処理されます。

例 **344:** 最上位レベルパッケージで定義されたシグナル

下記のシグナル「S」はインターフェイス `I1` と `I2` の両方に定義されたシグナルと同じです。

```
package signals {
    signal S;
    interface I1 {
        signal S;
    }
    active class C <<import>> dependency to p1 {
        active class C1 {
            port in with I1;
        }
        part C1 C1;
        part p1::C2 C2;
    }
    package p1 {
```

```
            active class C2 {
                port out with I2;
            }
            interface I2 {
                signal S;
            }
        }
    }
```

アクティブ クラスのインスタンスを表す型

Pid 値は、2 つのコンポーネントで構成される構造体になります。つまり、

整数型のグローバル ノード番号（通常は xGlobalNodeNumber 関数が生成）と、ポインタ型のローカル Pid 値です。

```
typedef xLocalPIdRec *xLocalPIdNode;

typedef struct {
  int GlobalNodeNr;
  xLocalPIdNode LocalPId;
} SDL_PId;
```

グローバル ノード番号はアクティブ クラスのインスタンスが属するシステムを識別し、ローカル Pid 値はシステム内のインスタンスを識別します。ローカル Pid ポインタ値はシステム内でのみ有効であるため、その値が定義されているシステム外で参照すべきではありません。

グローバル ノード番号を Pid 値に導入すると、Pid 値は複数のシステムから構成されるアプリケーション全体で解釈できるようになります。アプリケーションの非 UML 定義部分で独自の Pid 値を定義することも可能です。この場合もシグナルを使用して通信できます。

変数 `SDL_NULL` は、Pid のヌル値を表しており、ランタイム ライブラリで定義され、`scttypes.h` ファイルによって利用できます。この変数ではグローバル ノード番号とローカル Pid コンポーネントの両方がゼロになっています。`SDL_NULL` 以外のすべての Pid 値のグローバル ノード番号は、ゼロより大きくなければなりません。

シンボル テーブル

シンボル テーブルは、生成されたプログラムの実行の初期化フェーズ中に構築されるツリー型データであり、システムの静的構造に関する情報を含みます。シンボル テーブルには、シグナル型、アクティブ クラス、コネクタ、操作を表すノードが含まれます。シンボル テーブルのシグナルを表すのに使用する C 言語の型を以下に示します。

```
typedef struct xSignalIdStruct *xSignalIdNode;
typedef struct xSignalIdStruct {
 /* components */
} xSignalIdRec;
```

システムの外部環境とやりとりされるシグナルから見て、環境関数との結合に関して特に重要なのは、シグナル型を表すノードです。各シグナル型に対応する 1 つのシンボル テーブル ノードが定義されます。このノードは、名前 `ySigN_` の後に接頭辞を付

けたシグナル名を指定することで参照できます。ノードへの参照は、パラメータとして受け渡されるシグナルのシグナルデータ型を見つけるため、たとえば `xOutEnv` などで使用できます。

トップレベルのクラス内のポートに対応するシンボル テーブル ノードへの参照が重要な場合もあります。シグナルの場合と同様に、このようなノードは、名前 `yChaN_` の後に接頭辞の付いたポート名で参照できます。

環境関数

システムは、外部環境とのシグナルの送受信によって外部環境と通信します。外部環境に関する情報がシステム内で定義されていない場合、C コード ジェネレータは、たとえば、シグナルが外部環境に送信される場合に実行する必要のある動作を生成できません。したがって、外部環境に送信されるシグナルと実行すべき動作との間のマッピングを行う関数を提供する必要があります。このような動作の例として、ポートに対するビット パターンの書き込み、ネットワークを通じた別のコンピュータへの情報の送信、OS プリミティブを使用した別の OS タスクへの情報の送信などがあります。

システムの外部環境を扱うには、以下の関数を提供する必要があります。

- `xInitEnv`：環境を適切に初期化するため、アプリケーション起動時に呼び出されます。
- `xCloseEnv`：ファイルやソケットを閉じるなど、環境を適切な方法で確実に「閉じる」ため、アプリケーション終了時に呼び出されます。
- `xOutEnv`：システムから外部環境に送信されるシグナルを管理します。
- `xInEnv`：外部環境からシステムに送信されるシグナルを管理します。
- `xGlobalNodeNumber`：複数の通信システムがある場合に発生する問題を処理します。

これらの関数については、896 ページの「環境関数のガイドライン」セクションで詳しく説明しています。

関数スケルトン

環境関数のスケルトンを生成する方法には以下の 2 つがあります。

- 通常の方法は、アプリケーション ビルダーに環境関数を生成させることでスケルトンを生成する方法です。生成された環境関数の利点は、環境関数に実装されるシグナル インターフェイスが C コード ジェネレータにわかっているため、すべてのシグナルのコードやマクロをインターフェイスに挿入できることです。この情報をユーザー自身で計算するのは簡単ではありません。

  単純な場合では、この生成ファイルのマクロを調整するだけで実行可能な環境関数を得られることもありますが、通常はこれをスケルトンとして使用し、必要に応じて編集します。次のコード生成時に上書きされないよう、必ずファイルに別の名前を付けて保存してください。

- 以下のディレクトリにある sctenv.c ファイルのコピーを使用する方法もあります。
  .../sdlkernels/include
  このファイルには、実行のトレースに使用できるトレース機構も含まれています。ただしこのトレース機構が使用できるのは、ランタイム ライブラリ（C コード ジェネレータに含まれる）のソース コードを使用でき、さらに適切なコンパイラ マクロを使用して新しいオブジェクト ライブラリをコンパイルできる場合のみです。

システム インターフェイス ヘッダー ファイル

このセクションの内容は、C コード ジェネレータと AgileC コード ジェネレータの両方に有効です。

システム インターフェイス ヘッダー ファイルには、システム レベルで定義されているオブジェクトのコードが含まれます。外部 C コードの実装に必要なすべてのデータ型定義とそれ以外の外部定義が含まれます。これらのオブジェクト定義により、環境関数の実装が簡単になります。したがって、システム インターフェイス ヘッダー ファイルは環境ヘッダー ファイルと考えることもできます。

生成されたインターフェイス ヘッダー ファイルのデフォルト名は <system_file_name>.ifc です。

システム インターフェイス ヘッダー ファイルの内容

システム インターフェイス ヘッダー ファイルの内容と構造を以下に示します。

- マクロに変換されるすべての定数属性に対するマクロ
- パッシブ クラスとシンタイプ宣言から生成されるすべてのデータ型定義
- 変数に変換されるすべての定数属性の外部定義
- システム ダイアグラムで定義される各シグナルに対する、シグナルを表す xSignalIdRec 変数の extern 定義
- システム レベルで定義される各パラメータ付きシグナルに対する ySignalPar_SignalName 型と yPDP_SignalName 型など、シグナルを表すデータ型定義
- 外部環境とのやりとりであるリモート操作に対して生成される、以下の名前のシグナルのコード
  pCALL_procedurename、pREPLY_procedurename
- システム ダイアグラムで定義される各コネクタとポートに対する、コネクタを表す xChannelIdRec 変数の extern 定義
- C コード名への名前変換を簡単にするマクロ
- アプリケーションから外部環境へのシグナル パスに関する情報（オプション）。899 ページの「外部環境へのシグナル経路の有効活用」を参照。

C 言語での UML オブジェクトの名前

UML と C 言語では名前付け規則が異なることから、C 識別子を一意にするために C コード ジェネレータによって接頭辞や接尾辞が生成されます。ただしこれらの接頭辞や接尾辞は、システムの更新時に予定外の変更が行われる可能性があります。したがって、環境関数では接頭辞付きのオブジェクト名を使用しないでください。代わりに、システム インターフェイス ヘッダー ファイルで生成されたマクロが静的な名前から接頭辞付きの名前へのマッピングを行います。システム インターフェイス ヘッダー ファイルは、システムのコードを再生成するたびに再生成されます。インターフェイス名は静的であるため、環境関数を変更する必要はありません。

名前付けと接頭辞付与の規則については、第 27 章「C コード ジェネレータ リファレンス」の 959 ページ、「生成された C コードの名前」セクションを参照してください。

接頭辞

`.ifc` ファイル以外のシステムの生成コードでは、C 言語で名前を一意にするためにすべての UML 名に、接頭辞または接尾辞として、増加する番号が振られます。ただし、このために C 言語での名前の判定ができず、また、たとえば新しい定義が挿入された時のように、改めてコード生成が行われると番号が変更されてしまう可能性もあります。`.ifc` ファイルでは、UML 名には次の表に示すとおりあらかじめ定義された接頭辞が付けられます。

UML エンティティ	C コード名
コネクタ	`cha_%n`
定数	`con_%n`
データ型	`typ_%n`
リテラル	`lit_%n_%s`
シグナル	`sig_%n`

`%n` はエンティティの UML 名で置き換えられ、`%s` はデータ型の UML 名で置き換えられます。

これらの接頭辞はプロパティ エディタの Code generation properties によって制御されます。

システムのパートの名前は `.ifc` ファイルにあります。これらの名前は、`xInEnv` 関数でシグナルを送信する際に受信側として使用できます。

例 **345:** インターフェイス ヘッダー ファイルのマクロ

システムに Sig1 というシグナルが定義されている場合、以下のマクロが生成されます。

```
extern XCONST struct xSignalIdStruct ySigR_z5_Sig1;
#ifndef sig_Sig1
#define sig_Sig1 (&ySigR_z5_Sig1)
```

```
#endif
```

このマクロにより、接頭辞付きの名前 `ySigR_z5_Sig1` でなく静的な名前 `Sig1` を使用して `xIdNode` を参照できます。

マクロにより、以下のタイプに対して静的な名前が生成されます。

- パッケージ内の定数属性（マクロと変数に変換される）
- パッシブ クラスとシンタイプ宣言。パッシブ クラスが列挙型に変換される場合、すべてのリテラルが UML 名を使用して C 言語内で直接使用できます。
- シグナルを表す `xSignalIdNode`（`ySigN_` 接頭辞なし）
- コネクタとポートを表す `xChannelIdNode`（コネクタの内向きまたは外向きにアクセスするには、それぞれ接頭辞 `xIN_` または `xOUT_` を使用する）
- `yPDP_SignalName` ポインタ型。このデータ型は、名前 `yPDP_SignalName` を使用して参照できます。ここで `SignalName` は UML 名です。

注記

環境関数の編集や生成の前に必ずシステム インターフェイス ヘッダー ファイルを生成しなければなりません。

シグナル番号ファイル

注記

このセクションは、C コード ジェネレータのみに有効です。

シグナル番号ファイルは、アプリケーションのビルド時に任意で C コード ジェネレータによって生成されます（835 ページの「Advanced options」を参照）。シグナル番号ファイルには、シグナル番号に関する情報（C コード ジェネレータによって割り当てられる）と、これらの番号の元になっているシグナル名が含まれます。この機能の利用方法については、899 ページの「シグナルが多数ある場合の xOutEnv のパフォーマンス向上」サブセクションを参照してください。

シグナル番号ファイルの名前は `<basename>.hs` となります。

シグナル パラメータ レイアウト ファイル

注記

このセクションは、C コード ジェネレータのみに有効です。

環境関数のガイドライン

環境関数を含むファイルは、次のような構造になります。

```
#include "scttypes.h"
#include "systemfilename.ifc"

void xInitEnv XPP((void))
{
}
```

```
void xCloseEnv XPP((void))
{
}

void xOutEnv (xSignalNode *S)
{
}

void xInEnv (SDL_Time Time_for_next_event)
{
}

int xGlobalNodeNumber XPP((void))
{
}
```

xInitEnv 関数と xCloseEnv 関数

これらは、環境の初期化と終了の処理を行う関数です。

```
void xInitEnv ( void );

void xCloseEnv ( void );
```

これらの関数の実装では、ソフトウェアとハードウェアの初期化と終了に必要となる適切なコードを記述する必要があります。

xInitEnv 関数は、プログラムの起動時に最初の動作として呼び出されます。xCloseEnv 関数は SDL_Halt 関数内で呼び出されます。SDL_Halt の呼び出しは、プログラムを終了する適切な方法です。SDL_Halt を呼び出す最も簡単な方法は、以下のように、呼び出しをインラインとしてインクルードすることです。
[[SDL_Halt]]

SDL_Halt はランタイム ライブラリの一部で、定義は以下のとおりです。

```
void SDL_Halt ( void );
```

xInitEnv はシステム初期化の前に呼び出されます。すなわち、この関数内ではシステムへの参照は使用できません。

xOutEnv 関数

シグナルがシステムから外部環境に送信されるたびに、関数 xOutEnv が呼び出されます。この関数のプロトタイプは以下のとおりです。

```
void xOutEnv ( xSignalNode *S );
```

xOutEnv 関数は現在のシグナルをパラメータとして持っており、実行すべき動作を実装する際には、シグナルに含まれるすべての情報を自由に使用できます。シグナルには、シグナル データ型、送信インスタンスと受信インスタンス、シグナルのパラメータが含まれます。

アクティブ クラスのインスタンスやシグナルを表すのに使用するデータ型については、この章の前の部分にある 888 ページの「シグナルを表すデータ型」セクションと892 ページの「アクティブ クラスのインスタンスを表す型」セクションを参照してください。

xOutEnv のパラメータは xSignalNode のアドレス、つまりシグナルを表す構造体へのポインタのアドレスです。これは、xOutEnv 関数から復帰する前に、xOutEnv のパラメータとして指定されたシグナルを、使用可能メモリのプールに戻す必要があるからです。メモリの開放は、xReleaseSignal 関数を呼び出すことによって行います。この関数は、xSignalNode のアドレスをパラメータとし、シグナルを使用可能メモリのプールに戻し、さらに xSignalNode パラメータに 0 を割り当てます。したがって、呼び出し

```
xReleaseSignal(S);
```

が、xOutEnv からの復帰の前に必要です。xReleaseSignal 関数の定義は以下のとおりです。

```
void xReleaseSignal ( xSignalNode *S );
```

xOutEnv 関数内では、パラメータとしてこの関数に渡されるシグナルに含まれる情報を使用できます。まず、シグナルのデータ型を決定します。システム インターフェイス ヘッダー ファイルを使用し、シグナルの名前が Sig1 であるとすれば、以下の形式の式を含む if 文を使う方法が最も適しています。

```
(*S)->NameNode == Sig1
```

Receiver、Sender、シグナル パラメータにアクセスするのに適切な式は以下のとおりです。

```
(*S)->Receiver
(*S)->Sender
((yPDP_Sig1)(*S)) -> Param1
((yPDP_Sig1)(*S)) -> Param2
```

(以下同様)

Sender は常に送信インスタンスを参照します。一方 Receiver は外部環境中のアクティブ クラスの特定のインスタンスを参照するか、xEnv 値を取ります。xEnv は、インスタンスを明示的に指定せずに一般的な「外部環境アクティブ クラス」を表現するために使う Pid 値です。

Receiver は、次の場合に「アクティブ クラス」xEnv を参照します。つまり、アドレスされたシグナル送信の Pid 式が xEnv を参照しているか、またはシグナルが直接アドレスされずに送られ、受信相手のスキャンの結果、外部環境が受信側として選択される場合です。

環境関数内の、外部環境へまたは外部環境からのリモート操作呼び出しは、pCALL_procedurename と pREPLY_procedurename という 2 つのシグナルで処理します。

外部環境へのシグナル経路の有効活用

xOutEnv 関数の作成時に、所定シグナルに対するアプリケーションからの経路（出力ポート）を知っておくと便利なことがあります。コンパイル スイッチ XPATH_INFO_IN_ENV_FUNC を使用してアプリケーションをコンパイルすることにより、この情報が利用できます。

この場合、xOutEnv 関数のプロトタイプは次のようになります。

```
extern void xOutEnv (xSignalNode *, xChannelIdNode);
```

注記

この機能を使用する場合、必ず定義に従って xOutEnv 関数の実装を更新してください。

出力経路に関する情報は、生成された .ifc ファイルにあります。

例 **346:** 生成されたシステム ヘッダー ファイルにある出力経路

```
#ifndef XOPTCHAN
extern XCONST struct xChannelIdStruct yChaR_c_0;
extern XCONST struct xChannelIdStruct yChaRR_c_0;
#define xIN_c (&yChaRR_c_0)
#define xOUT_c (&yChaR_c_0)
#endif
```

xIN_c と xOUT_c は、経路（この場合は「c」）の内向き／外向き方向を参照するのに使用できるマクロです。xOutEnv 関数においては、xOUT_c に利用価値があります。処理するシグナルがパス「c」を通じて環境に伝送されたかどうかテストするには、次に示すようなテストが使用できます。

```
void xOutEnv(xSignalNode *SignalOut, xChannelIdNode Path)
{
  ...
  if (Path == xOUT_c) ...
  ...
}
```

シグナルが多数ある場合の **xOutEnv** のパフォーマンス向上

通常、シグナルはポインタ（xSignalIdNode 型）によって識別されます。つまり、xOutEnv のシグナルのデータ型を見つけるには、一連の "else if" 文を使用する必要があります。これは、xOutEnv 関数で多数のシグナルを処理する必要がある場合は非効率的です。シグナルを番号によって識別できれば、switch 文を使用することにより、実行パフォーマンスの向上が図れます。

シグナル番号機能を利用するには、以下の 2 つの操作を実行する必要があります。

1. C コード ジェネレータの Advanced options で、**Set-Signal-Number** を指定します。これでシグナル番号ファイルが生成されます。
2. コンパイル スイッチ XUSE_SIGNAL_NUMBERS でアプリケーションをコンパイルします。

このコンパイル スイッチにより、シグナル番号が xSignalIdStruct にインクルードされます。すなわち、xOutEnv で、次の例のようなスイッチが使用できます。

```
switch ((*SignalOut)->NameNode->SignalNumber) {
  case SN_signalname1 : ....
  case SN_signalname2 : ....
  ....
}
```

xOutEnv 関数のガイドライン

xOutEnv 関数は自由に記述できます。次に示す構成例は、xOutEnv 関数の設計方法の一例です。シグナル名でなくシグナル番号を使用するように関数を設計することも可能です。899 ページの「シグナルが多数ある場合の xOutEnv のパフォーマンス向上」を参照してください。

例 **347: xOutEnv** 関数の構成

```
void xOutEnv ( xSignalNode *S )
{
 if ( (*S)->NameNode == Sig1 ) {
  /* perform appropriate actions */
  xReleaseSignal(S);
  return;
 }
 if ( (*S)->NameNode == Sig2 ){
  /* perform appropriate actions */
  xReleaseSignal(S);
  return;
 }
 /* and so on */
}
```

xInEnv 関数

xInEnv 関数は、外部環境から受信したシグナルのシステムへの送信を可能にするために使用します。ベア統合では、この関数はシステム実行中に繰り返し呼び出されます。スレッド統合では、xInEnv 関数はそれ自体のスレッドで実行されるため、復帰しません。実行中、xInEnv は外部環境をスキャンし、システム内のアクティブ クラスのインスタンスへのシグナル送信のトリガとなる事象が発生したかどうか調べます。

```
void xInEnv (SDL_Time Time_for_next_event);
```

システムへのシグナル送信を実行するには、2 つの関数が使用できます。1 つは xGetSignal で、これはシグナルを表すのに適切なデータ領域を取得するのに使用します。もう 1 つは SDL_Output で、これは指定された受信者にシグナルを送信するものです。これらの関数については後述します。

パラメータ Time_for_next_event は、関数が 1 回しか呼び出されないスレッド統合では意味がありません。xInEnv 関数には、次の無限ループが含まれている必要があります。

```
function xInEnv
{
  while (1) {
    Wait_for_something_to_do;
    /* Handle different signals */
  }
}
```

ここで **"Wait_for_something_to_do"** は、何らかの送信シグナルがあるまでこのスレッドの実行を保留するユーザー定義文を表します。たとえば、xInEnv はあるセマフォで待機できます。このセマフォは、シグナルをシステムに送信する必要が発生した時に割り込みルーチンや別の外部コードで解除されます。

ベア統合では、Time_for_next_event パラメータには、システムで次にスケジュールされるイベントの時間が含まれます。パラメータは以下のいずれかです。

- 0 または Now：直ちに実行できる遷移（またはタイマー出力）があることを示します。
- Now よりも大きい数字：次のイベントが、指定時間に予定されているタイマー出力であることを示します。
- 非常に大きい数字：システムに予定されている動作がないこと、つまりシステムが外部からの起動を待っていることを示します。このような大きな数字は、xSysD.xMaxTime 変数に設定されることがあります。

時刻が Time_for_next_event を過ぎている場合、環境をスキャンし、現在のシグナル送信を実行し、できるだけ早く制御を戻さなければなりません。

時刻が Time_for_next_event を過ぎていなければ、xInEnv 関数からすぐに制御を戻して xInEnv を再度呼び出すこともできます。または、シグナル（システムに送信されるシグナル）が起動されるか、時刻が Time_for_next_event を過ぎるまで xInEnv 内で待機することもできます。

注記

ベア統合でのデバッグにおいては、できるだけ早く xInEnv 関数から、制御を戻し、関数が正しく動作することを確認することをお勧めします。この関数に長く待機すると、キーボード ポーリング、すなわち、実行の中断を目的として RETURN を押した場合の中断処理が機能しません。できるだけ早く制御を戻しても何も実行する操作がない場合は、"busy wait" を引き起こすため、実アプリケーションでは注意が必要です。

xGetSignal 関数

xGetSignal 関数は、信号を送信するときに使用するサービス関数の 1 つです。この関数は最初のパラメータによって指定されたデータ型のシグナル インスタンスを表すデータ領域のポインタを返します。

```
xSignalNode xGetSignal
 ( xSignalIdNode SType,
   SDL_PId Receiver,
   SDL_PId Sender );
```

シグナル インスタンスの Receiver コンポーネントと Sender コンポーネントにも、対応するパラメータの値が割り当てられます。

- SType : このパラメータは、現在のシグナル型を表すシンボル テーブル ノードへの参照です。システム インターフェイス ヘッダー ファイルを使用すると、シグナル名を使用して、このようなシンボル テーブル ノードを直接参照できます。
- Receiver : このパラメータは、システム内のアクティブ クラスのインスタンスの Pid 値か、xNotDefPId 値のいずれかです。xNotDefPId 値は、シグナルを直接アドレシングなしで送信する必要があることを示します。Pid 値を指定した場合は、直接アドレシングを含むシグナル送信と見なされます。システム中のアクティブ クラスのインスタンスの Pid 値は計算できません。したがって、システムから送信されたシグナルによって伝達される情報（Sender またはパラメータ）から取り込む必要があります。
- Sender : Sender は、現在のシステムの環境にあるアクティブ クラスのインスタンスを表す Pid 値か、xEnv 値のいずれかです。xEnv は、外部環境中のインスタンスを明示的に指定せずに、システム環境の一般的な概念を表す「外部環境アクティブ クラス」を参照する Pid 値です。

SDL_Output 関数

ランタイム ライブラリ関数 SDL_Output は、PAD 関数 から呼び出され、シグナルの送信を実装します。

SDL_Output 関数は、シグナル インスタンスの参照を使用し、受信側を特定した後にシグナルを送信します。

SDL_Output が、シグナルの受信側が現在のシステムの一部でないアクティブ クラスのインスタンスと判断すると、SDL_Output は xOutEnv 関数を呼び出します。

```
void SDL_Output
 ( xSignalNode S,
   xIdNode     ViaList[] );
```

- S : このパラメータは、コンポーネントがすべて格納されたシグナル インスタンスへの参照です。
- ViaList : このパラメータは、VIA 節がシグナル送信文の一部であるかないかを指定するのに使用します。(xIdNode *)0 値（ヌル ポインタ）は、VIA 節がないことを表すのに使用します。903 ページの例 349 を参照してください。

これだけの情報があれば、シグナルを送信するコードを作成できます。

例 **348:** 環境からシグナルを送信する **C** コード ——————————————————

パラメータのないシグナル S1 を、明示的な受信側なしで、xEnv からシステムに送信する場合を考えます。コードは以下のようになります。

```
SDL_Output( xGetSignal(S1, xNotDefPId, xEnv),
  (xIdNode *)0 );
```

2 つの整数パラメータを持つ S2 を、xEnv から変数 P が参照するインスタンスに送信する場合のコードは以下のようになります。

```
xSignalNode OutputSignal; /* local variable */
...
OutputSignal = xGetSignal(S2, P, xEnv);
```

```
((yPDP_S2)OutputSignal)->Param1 = 1;
((yPDP_S2)OutputSignal)->Param2 = 2;
SDL_Output( OutputSignal, (xIdNode *)0 );
```

シグナルに via リストを導入するには、xIdNode の配列の変数が必要です。xIdNode には、via リスト内の現在のコネクタを表すシンボル テーブル ノードへの参照が格納されます。

具体的には以下のような変数が必要です。

```
ViaList xIdNode[N];
```

N は、via リストの最長の表現の長さに 1 を加えた値です。変数のコンポーネントには対応する値を設定します。つまり、コンポーネント 0 には via リストにある 1 番目のコネクタ（シンボル テーブル ノード）への参照を指定し、コンポーネント 1 には 2 番目のコネクタへの参照を指定し、以下同様に指定します。コネクタを参照する最後のコンポーネントの次には、ヌル ポインタ（(xIdNode)0 値）を含むコンポーネントを指定する必要があります。ヌル ポインタ以降のコンポーネントは参照されません。C1 と C2 という 2 つのコネクタの via リスト生成の例を以下に示します。

例 **349: 2** つのコネクタの **via** リスト

```
ViaList xIdNode[4];
/* longest via has length 3 */
...
/* this via has length 2 */
ViaList[0] = (xIdNode)xIN_C1;
ViaList[1] = (xIdNode)xIN_C2;
ViaList[2] = (xIdNode)0;
```

これで、変数 ViaList を、後続の SDL_Output 呼び出しで ViaList パラメータとして使用できます。

xInEnv 関数のガイドライン

xInEnv 関数は、基本的に、外部環境を評価する多数の if 文で構成されます。シグナルを送信する適切なコード（902 ページの例 348）は、システムへのシグナル送信を意味する何らかの情報が検出された場合に、実行される必要があります。

以下に示す例は、ベア統合の xInEnv 関数です。スレッド実行モデルでは、上記の無限ループと wait 文を追加する必要があります。

例 **350: xInEnv** 関数の構造

```
void xInEnv (SDL_Time Time_for_next_event)
{
  xSignalNode S;

  if ( Sig1 should be sent to the system ) {
    SDL_Output (xGetSignal(Sig1, xNotDefPId,
      xEnv), (xIdNode *)0);
  }
```

```
    if ( Sig2 should be sent to the system ) {
      S = xGetSignal(Sig1, xNotDefPId, xEnv);
      ((xPDP_Sig2)S)->Param1 = 3;
      ((xPDP_Sig2)S)->Param2 = SDL_True;
      SDL_Output (S, (xIdNode *)0);
    }
    /* and so on */
  }
```

この基本構造は、用途に合わせて変更できます。たとえば、`if` 文の代わりに `while` 文を使用できます。シグナル データ型は何らかの「優先順」にソートして処理でき、また `return` 文は `if` 文の最後に導入して制御を戻すことができます。つまり、`xInEnv` の 1 回の呼び出しではシグナル 1 つのみが送信され、待ち時間を短縮できます。

xGlobalNodeNumber 関数

パラメータのない `xGlobalNodeNumber` 関数を使って、各実行システムに固有の整数を 1 つ返す必要があります。

```
int xGlobalNodeNumber ( void )
```

この戻り値は、アプリケーションを構成する通信システム間で固有で、0 より大きい整数でなければなりません。アプリケーションが 1 システムのみで構成されている場合は、この値はそれほど重要ではありません（設定の必要はあります）。このグローバル ノード番号は、アクティブ クラスのインスタンスが属するノード（OS タスクまたはプロセッサ）を特定する Pid 値に使用されます。これで Pid 値は、グローバルにアクセス可能となり、たとえば、異なるシステムにあるアクティブ クラスのインスタンス間で「Sender.Output」を簡単に機能させることができます。

複数の通信システムから構成されるアプリケーションを設計する場合、グローバル なノード番号を、現在の OS タスクかプロセッサに割り当てて、グローバルな PId のアドレスを持つシグナルを正しい OS タスクやプロセッサに転送できるようにする必要があります。この処理は、`xOutEnv` 関数で実行します。

xMainInit 関数と xMainLoop 関数

生成コードには、初期化関数と PAD 関数 という 2 つの重要な関数が含まれます。PAD 関数は、ステート遷移中にアクティブ クラスのインスタンスによって実行される動作を実装します。初期化関数は各生成 `.c` ファイルに 1 つ含まれます。この関数はシステムを表現するファイル内にあり、名前は `yInit` です。システム内の各インスタンスは、PAD 関数で表現され、現在のインスタンス集合のインスタンスが遷移を実行する場合に呼び出されます。

`main()` 関数での `xMainInit()` 関数と `xMainLoop()` 関数の使用方法は、例に示すとわかりやすいでしょう。

例 **351:** 起動およびエンドレス ループ

```
void main ( void )
{
```

```
 xMainInit();
 xMainLoop();
}

void xMainInit ( void )
{
  xInitEnv(); /* Init of internal data structures in the run-
time library */
  yInit();
}

void xMainLoop ( void )
{
  while (1) {
    xInEnv(...);
    if ( Timer output is possible )
      SDL_OutputTimerSignal();
    else if ( transition is possible )
      Call appropriate PAD function;
  }
}
```

実行の停止

前述のとおり、xMainLoop 関数には無限ループが含まれます。プログラムの実行を適切に停止するには、ランタイム ライブラリ関数 SDL_Halt を呼び出します。通常この C 言語関数の呼び出しは、インライン C コードを使用して適切なアクションにインクルードします。

SDL_Halt 関数の構造は以下のとおりです。

```
void SDL_Halt ( void )
{
  xCloseEnv();
  exit(0);
}
```

25

C および AgileC ランタイム ライブラリ

この章では、C コード ジェネレータおよび AgileC コード ジェネレータが使用するランタイム ライブラリの実装とインターフェイスについて説明します。ライブラリは C ソース コードで提供されます。

また、C コード ジェネレータおよび AgileC コード ジェネレータとともに提供されるデフォルト セットから新しいライブラリを生成する方法についても取り上げます。

最後に、アプリケーションのコンパイルやリンクに使用する C コンパイラの要求事項や規則に適合するようライブラリを適応させる方法について説明します。

ランタイム ライブラリ

ランタイム ライブラリ ディレクトリ構造

C コード ジェネレータ ランタイム ライブラリに使用するファイルとディレクトリの構造を図で説明します。この構造のルート ディレクトリの名前は `sdlkernels` で、これには、コンパイルし、C コード ジェネレータによって生成されたコードとリンクするランタイム ライブラリを実装するすべてのファイルが含まれます。

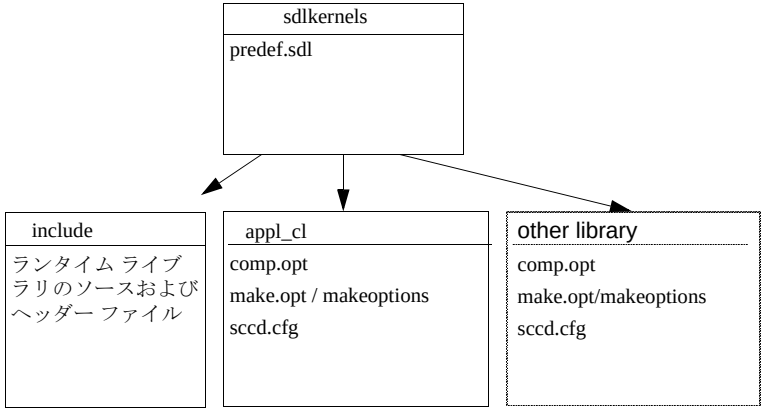


図 241: ランタイム ライブラリのディレクトリ構造

sdlkernels

`sdlkernels` ディレクトリには、`predef.sdl` ファイルがあります。このファイルにはあらかじめ定義されたソート（データ型）の定義が含まれ、C コード ジェネレータが、C コード生成前にシステムの構文とセマンティクスをチェックするために使用します。

ランタイム カーネルは、ランタイム ライブラリのいずれかのオプションを適用し、include ディレクトリのソース コードによって定義されます。

include

include ディレクトリには、ランタイム ライブラリとモデル ベリファイヤ（Model Verifier）のソース コード ファイルが含まれます。これらのファイルについては、917 ページの「インクルードされるソース ファイルとヘッダー ファイル」で説明します。

ランタイム ライブラリ

`include` ディレクトリと同じレベルにいくつかのディレクトリがあり、ここには `comp.opt` ファイルと `makeoptions` ファイル（Windows では `make.opt` ）が含まれます。これらのファイルによって定義されたコンパイルとリンクの設定を使用すると、選択した特定のターゲット アプリケーションに固有のプロパティが実行形式コードに付与されるように、生成アプリケーション コードと include ディレクトリ内のファイルの両方がコンパイルされます。

- comp.opt ファイル ファイルは、生成 make ファイルの内容と「make」の呼び出し方法を決定します。

- makeoptions/make.opt ファイルは、使用コンパイラ、コンパイラ オプション、リンカー オプションなど、ライブラリのプロパティを記述します。
- sccd.cfg ファイルファイルは、生成 C コードを読みやすくするのに使用する C コンパイラ ドライバ ユーティリティの設定ファイルです。

ディレクトリの名付け規則

コンパイラとライブラリのプロパティを含むディレクトリの名前は、生成されるコードのプロパティを意味する言葉を短縮して表記した名付け規則に従って付けられます。

名付け規則に使用される構文は次のとおりです。

```
<"dbg" | "appl" | "applt"> || <compiler> || [_cpp]
```

- `dbg`：生成結果がホストでのアプリケーションのデバッグに適したモデル ベリファイヤ（Model Verifier）になることを示します。
- `appl`：生成結果がベア アプリケーションになることを示します。
- `applt`：生成結果がスレッド アプリケーションになることを示します。
- `compiler`：使用するコンパイラとリンカーを示します。
- `_cpp`：コードが C++ 互換モードでコンパイルされることを示します。

例 **352: Windows** のアプリケーションのコンパイル

908 ページの図 241 の「`appl_cl`」ディレクトリの名前が上で述べた規則の例です。

このディレクトリ内の `comp.opt` ファイルと `make.opt` ファイルを使用することにより、Windows 用のアプリケーションのコンパイルとリンクができます。

アプリケーション ビルダを使用してアプリケーションをビルドする場合は、これらの短縮表記の代わりに、プロパティ エディタの GUI で明示的に指定した名前が使用されます。

サポートされるライブラリ

次の表は、Tau で使用できるライブラリを示したものです。それぞれ、ランタイムと環境の統合モデルである、ベアまたはスレッドのいずれかとともに使用します。

注記

ビルド タイプ C コード ジェネレータおよびモデル ベリファイヤ（Model Verifier）の場合、使用するステレオタイプの属性で定義された設定によって、ライブラリを持つディレクトリが決まります。

属性によって定義されたランタイム ライブラリ

次の表は、ビルド タイプ C コード ジェネレータおよびモデル ベリファイヤ（Model Verifier）の設定を定義する各属性の全組み合わせと、コードのコンパイルとリンクに使用する対応ランタイム ライブラリをまとめたものです。OS 統合のサポートが、AgileC コード ジェネレータで記述されたデプロイメント モデルを使用する点に注意してください。

ビルド タイプ（Build Type）	C++ のサポート（Support C++）	ターゲット形式（Target kind）	スレッド モデル（Threading model）	定義されるランタイム ライブラリ（Resulting run-time library）
AgileC コード ジェネレータ	なし	Solaris Forte	ベア	`agilec/agilec_appl_cc`
AgileC コード ジェネレータ	あり	Solaris Forte	ベア	`agilec/agilec_appl_cc_cpp`
AgileC コード ジェネレータ	なし	Solaris Forte	スレッド	`agilec/agilec_applt_cc`
AgileC コード ジェネレータ	あり	Solaris Forte	スレッド	`agilec/agilec_applt_cc_cpp`
AgileC コード ジェネレータ	なし	Solaris gcc	ベア	`agilec/agilec_appl_gcc`
AgileC コード ジェネレータ	あり	Solaris gcc	ベア	`agilec/agilec_appl_gcc_cpp`
AgileC コード ジェネレータ	なし	Solaris gcc	スレッド	`agilec/agilec_applt_gcc`
AgileC コード ジェネレータ	あり	Solaris gcc	スレッド	`agilec/agilec_applt_gcc_cpp`
AgileC コード ジェネレータ	なし	Win32	ベア	`agilec/agilec_appl_cl`
AgileC コード ジェネレータ	あり	Win32	ベア	`agilec/agilec_appl_cl_cpp`
AgileC コード ジェネレータ	なし	Win32	スレッド	`agilec/agilec_applt_cl`
AgileC コード ジェネレータ	あり	Win32	スレッド	`agilec/agilec_applt_cl_cpp`
AgileC コード ジェネレータ	なし	VxWorks (Win32)	スレッド	`agilec/agilec_applt_ccsimpc`
AgileC コード ジェネレータ	あり	VxWorks (Win32)	スレッド	`agilec/agilec_applt_ccsimpc_cpp`
AgileC コード ジェネレータ	なし	VxWorks (Solaris)	スレッド	`agilec/agilec_applt_ccsimso`

ビルド タイプ (Build Type)	C++ のサポート (Support C++)	ターゲット形式 (Target kind)	スレッド モデル (Threading model)	定義されるランタイム ライブラリ (Resulting run-time library)
AgileC コード ジェネレータ	あり	VxWorks (Solaris)	スレッド	agilec/agilec_applt_ccsimso_cpp
C コード ジェネレータ	なし	VxWorks (Win32)	ベア	applt_ccsimpc
C コード ジェネレータ	あり	VxWorks (Win32)	ベア	applt_ccsimpc_cpp
C コード ジェネレータ	なし	VxWorks (Solaris)	ベア	applt_ccsimso
C コード ジェネレータ	あり	VxWorks (Solaris)	ベア	applt_ccsimso_cpp
C コード ジェネレータ	なし	Solaris Forte	ベア	appl_cc
C コード ジェネレータ	あり	Solaris Forte	ベア	appl_cc_cpp
C コード ジェネレータ	なし	Solaris Forte	スレッド	applt_cc
C コード ジェネレータ	あり	Solaris Forte	スレッド	applt_cc_cpp
C コード ジェネレータ	なし	Solaris gcc	ベア	appl_gcc
C コード ジェネレータ	あり	Solaris gcc	ベア	appl_gcc_cpp
C コード ジェネレータ	なし	Solaris gcc	スレッド	applt_gcc
C コード ジェネレータ	あり	Solaris gcc	スレッド	applt_gcc_cpp
C コード ジェネレータ	なし	Win32	ベア	appl_cl
C コード ジェネレータ	あり	Win32	ベア	appl_cl_cpp
C コード ジェネレータ	なし	Win32	スレッド	applt_cl
C コード ジェネレータ	あり	Win32	スレッド	applt_cl_cpp
Model Verifier	なし	Solaris Forte	ベア	dbg_cc
Model Verifier	あり	Solaris Forte	ベア	dbg_cc_cpp

ビルド タイプ（Build Type）	C++ のサポート（Support C++）	ターゲット形式（Target kind）	スレッド モデル（Threading model）	定義されるランタイム ライブラリ（Resulting run-time library）
Model Verifier	なし	Solaris gcc	ベア	`dbg_gcc`
Model Verifier	あり	Solaris gcc	ベア	`dbg_gcc_cpp`
Model Verifier	なし	Win32	ベア	`dbg_cl`
Model Verifier	あり	Win32	ベア	`dbg_cl_cpp`
Model Verifier	なし	Win32-gcc [a]	ベア	`dbg_cyg`
Model Verifier	あり	Win32-gcc [b]	ベア	`dbg_cyg_cpp`

a. ÉÇÉfÉã ÉxÉäÉtÉ@ÉCÉÑÅiModel VerifierÅjÇÝÇÃTarget kindÇÃégópÇ…ä÷Ç²ÇÈêßå¿ÇšéQèÝÇµÇƒÇ¦ÇæÇŠÇ¢ÅB

b. ÉÇÉfÉã ÉxÉäÉtÉ@ÉCÉÑÅiModel VerifierÅjÇÝÇÃTarget kindÇÃégópÇ…ä÷Ç²ÇÈêßå¿ÇšéQèÝÇµÇƒÇ¦ÇæÇŠÇ¢ÅB

参照

第 56 章「定義済みのステレオタイプと 属性」

ライブラリ ファイル

このセクションでは、ライブラリが実装されるディレクトリの内容について説明します。以下のトピックを取り上げます。

- comp.opt ファイル
- makeoptions/make.opt ファイル
- sccd.cfg ファイル
- make ファイル
- make テンプレート ファイル

comp.opt ファイル

このファイルは、生成される make ファイルの詳細と make ファイル実行のために発行するコマンドを決定します。`comp.opt` ファイルには、「#」で始まる行が含まれることがあります。これらの行はコメントと見なされます。その後に必須データが 5 行続きます。

- 1 行目：`makeoptions`（`make.opt`）ファイルのインクルード方法
- 2 行目：コンパイル スクリプト
- 3 行目：リンク スクリプト
- 4 行目：make を実行するコマンド
- 5 行目：コーダやデコーダのライブラリ（アーカイブ）のビルド方法

これらの各行で % コードを使用して特定の情報を挿入できます。

5 行すべてで使用可能

```
%n : newline
%t : tab
%d : target directory
%s : source directory
%k : kernel directory
%f : base name of generated executable (no path, no
     file extension). NOT on line 2 or 5.
```

2 行目のコンパイル スクリプトで使用可能

```
%c : c file in compile script
%C : c file in compile script, without extension
%o : resulting object file in compile script
```

3 行目のリンク スクリプトで使用可能

```
%o : list of all object files in link script
%O : list of all object files in link script, with
     ¥ followed by newline between files
%e : executable file in link script
```

4 行目の make コマンドで使用可能

```
%m : name of generated makefile
```

5 行目のアーカイブコマンドで使用可能

```
%o : list of object files, $(sctCODER_OBJS).
%a : the archive file, libstcoder$(sctLIBEXTENSION)
```

例 **353: UNIX** の **comp.opt** ファイル

```
# makefile for unix make
include $(sctdir)/makeoptions
%t$(sctCC) $(sctCPPFLAGS) $(sctCCFLAGS) $(sctIFDEF) %c -o %o
%t$(sctLD) $(sctLDFLAGS) %o -o %e
make -f %m sctdir=%k
%t$(sctAR) $(sctARFLAGS) %a %o
```

makeoptions/make.opt ファイル

このファイルの構造は以下のとおりです。

例 **354: UNIX** の **make.opt** ファイル

```
# #
sctLIBNAME     = Simulation
sctIFDEF       = -DSCTDEBCOM
sctEXTENSION   = _smd.sct
sctOEXTENSION  = _smd.o
sctLIBEXTENSION=  _smd.a
sctKERNEL      = $(sctdir)/../INCLUDE
sctCODERDIR    = $(sctdir)/../coder

#Compiling, linking
#Take advantage of the C Compiler Driver
```

```
sctSCCD         =
sctCC           = $(sctSCCD) cc
sctCODERFLAGS   = -I$(sctCODERDIR)
sctCPPFLAGS     = -I. -I$(sctKERNEL) $(sctCODERFLAGS)
 $(sctCOMPFLAGS) $(sctUSERDEFS)
sctCCFLAGS      = -c -Xc
sctLD           = cc
sctLDFLAGS      =
sctAR           = ar
sctARFLAGS      = rcu

all : default

# below this point there are a large number of
# compilation rules for compiling the libraries
# The following names of importance are defined:

sctLINKKERNEL  =
sctLINKKERNELDEP =
sctLINKCODERLIB =
sctLINKCODERLIBDEP =
```

上図の等号記号の右側にある情報は 1 つの例です。makeoptions（make.opt）ファイルに設定される環境変数では以下の内容を指定します。

- sctLIBNAME：make ファイルが動作内容を報告するためだけに使用します。
- sctIFDEF：C コード ジェネレータによって定義されたコンパイル スイッチのうちどれを使用するかを指定します。通常、ライブラリ バージョンを定義するスイッチが 1 つあります。
- sctEXTENSION：実行形式ファイルのファイル拡張子を決定します。
- sctOEXTENSION：オブジェクト ファイルのファイル拡張子を決定します。
- sctLIBEXTENSION：アーカイブまたはライブラリのファイル拡張子を決定します。
- sctKERNEL：ランタイム ライブラリ ソース コードのディレクトリです。
- sctCODERDIR：コーダまたはデコーダのソース コードのディレクトリです。
- sctCC：使用するコンパイラを定義します。
- sctCODERFLAGS：コーダ ファイルまたはデコーダ ファイルのコンパイルに必要なコンパイル オプションです。
- sctCPPFLAGS：C プリプロセッサがどこで scttypes.h、sctlocal.h、sctpred.h の各インクルードファイルを見つけられるかを指定するコンパイル フラグを付与するものです。
- sctCCFLAGS：デバッグ情報の -g（Sun cc）または -v（Borland bcc32）、最適化の -O など、他に使用するコンパイル フラグを指定します。
- sctUSERDEFS：make テンプレート ファイルでこのフラグを使用してコンパイル用の追加インクルード パスを設定できます。この変数は、標準カーネルの sctCPPFLAGS で起動します。
- sctLD：使用するリンカーを定義します。
- sctLDFLAGS：リンク操作で使用する他のフラグを指定するものです。

- `sctAR`：アーカイブ アプリケーションです。
- `sctARFLAGS`：sctAR のフラグです。
- `sctLINKKERNEL`：ライブラリ ソース ファイルの `.o` ファイルを指定します。生成 make ファイルのリンク コマンドで使用します。
- `sctLINKKERNELDEP`：必要に応じて依存関係を実装してカーネルをリコンパイルするのに使用します。
- `sctLINKCODERLIB`：コーダ ライブラリ ソース ファイルの `.o` ファイルを指定します。生成 make ファイルのリンク コマンドで使用します。
- `sctLINKCODERLIBDEP`：依存関係を実装してコーダ ライブラリを必要に応じて再コンパイルするのに使用します。

sccd.cfg ファイル

C コンパイラ ドライバ ユーティティの CCD 設定ファイルです。C マクロを拡張し、生成 C コードを読みやすく、C 言語のレベルでデバッグができるようにするために使用します。

make ファイル

make ファイルの生成と実行は、C コード ジェネレータとアプリケーション ビルダによって自動的に処理されます。make ファイルの内容は、例で示すと理解しやすいでしょう。

例 **355:**「**example**」という名前のシステム用の **UNIX** 生成 **make** ファイル ————————

```
# makefile for System: example

sctAUTOCFGDEP =
sctCOMPFLAGS = -DXUSE_GENERIC_FUNC

include $(sctdir)/makeoptions

default: example$(sctEXTENSION)

example$(sctEXTENSION): ¥
  example$(sctOEXTENSION) ¥
  $(sctLINKKERNELDEP)
  .$(sctLD) $(sctLDFLAGS) ¥
  example$(sctOEXTENSION) $(sctLINKKERNEL) ¥
  -o example$(sctEXTENSION)

example$(sctOEXTENSION): ¥
  example.c
  .$(sctCC) $(sctCPPFLAGS) $(sctCCFLAGS) ¥
  $(sctIFDEF) example.c -o example$(sctOEXTENSION)
```

——

このように生成された make ファイルを ” メーク ”（実行）する際、以下に示す一連のアクションが実行されます。

3. makeoptions（make.opt）ファイルが、環境変数 sctdir が参照するディレクトリにインクルードされます。
4. 次に、sctIFDEF、sctLINKKERNEL、sctCC、sctCPPFLAGS、sctCCFLAGS、sctLD、sctLDFLAGS の各変数を使用して、C コード ジェネレータによって生成されたアプリケーション コード（この場合は 915 ページの例 355 の example.c）をコンパイルし、リンクします。
5. C コード ジェネレータで生成されたコードには、適切な場所に、必要な .c ファイルと .h ファイルをインクルードするための #include 文が挿入されます。他のファイルとともにランタイム カーネルを実装するこれらのファイルは、自動的に、アプリケーション コードとともにコンパイル、リンクされます。

注記

特定のシステム向けに生成されたコードとランタイム カーネルの間に矛盾が生じないように、makeoptions (make.opt) ファイルを使用して、ライブラリに含まれる C ファイルと生成されたアプリケーション コードをコンパイルする make ファイルを生成します。使用するランタイム ライブラリとアプリケーション コードの間に矛盾がある場合、結果が予測不可能になります。

make テンプレート ファイル

make テンプレート ファイルは、外部コードをアプリケーション コードやランタイム ライブラリとともにコンパイル、リンクするのに便利です。このようなコードとして考えられるものに、環境関数や外部ライブラリがあります。make テンプレート ファイルは、コンピュータで現在使用されている「Make」の構文に従わなければなりません。

USERTARGET は、C コード ジェネレータが生成した make ファイルに追加する必要のある追加 make ターゲットを指定するのに使用します。

USERLIBRARIES は、アプリケーションにリンクする必要のある追加ライブラリの指定に使用します。

Make template file の使用はビルド アーティファクトのコード ジェネレータのステレオタイプによって制御されます。

例 **356: make** テンプレート ファイル（**nmake** を使用）

example_env.c ファイルと example_env.h ファイルに存在する環境関数を考えます。

テンプレートとして使用する make ファイルは次のようになります。

```
USERTARGET = C:¥example¥example_env$(sctOEXTENSION)
USERLIBRARIES = -lm

C:¥example¥example_env$(sctOEXTENSION):
C:¥example¥example_env.c C:¥example¥example_env.h
  .$(sctCC) @<<
  .$(sctCPPFLAGS) $(sctCCFLAGS)
  .$(sctIFDEF)
  ./FoC:¥example¥example_env$(sctOEXTENSION)
```

```
  .C:¥example¥example_env.c
<<
```

インクルードされるソース ファイルとヘッダー ファイル

C コード ジェネレータが使用するランタイム ライブラリのコードと定義は、多くのファイルに及びます。これらのファイルはすべて `...¥sdlkernels¥include¥` ディレクトリにあります。ファイルの内容について以下のセクションで簡単に説明します。

sctda.c

このファイルには、モデル ベリファイヤ（Model Verifier）によって送信された要求に応じるコマンド インタープリタとコマンド エグゼキュータを実装する関数が含まれます。

sctadacom.c

このファイルは、モデル ベリファイヤ（Model Verifier）とアプリケーション間の相互通信に使用される通信層を実装します。この実装は TCP/IP ソケットです。

sctadacom.h

このヘッダー ファイルには、`sctadacom.c` が使用する定義が含まれます。

sctdamsg.c

このファイルは、モデル ベリファイヤ（Model Verifier）とアプリケーション間の相互通信に使用されるメッセージ 層を実装します。`sctadacom.c` で実装される通信層の上に構築されます。

sctdamsg.h

このヘッダー ファイルには、`sctdamsg.c` が使用する定義が含まれます。

sctdamsgcode.h

このヘッダー ファイルには、`sctdamsg.c` が使用する定義が含まれます。

sctlocal.h

このファイルには、変数の型定義と `extern` 宣言、およびランタイム カーネルのみで使用される関数が含まれます。このファイルは生成されるコードにはインクルードされません。

sctos.c

このファイルには、ハードウェア、オペレーティング システム、およびコンパイラの依存関係を表現する関数が配置されます。

アプリケーションがターゲット上で正しく機能するには、以下の関数が必要です。

- クロックを読み込む関数
- メモリを割り当てる関数

sctpred.c

このファイルには、定義済みデータ型のために定義された操作を実装する関数があります。

sctpred.h

このファイルには、定義済みデータ型を処理する型定義と `extern` 宣言が含まれます（`scttypes.h` にある Pid を除く）。このファイルは `scttypes.h` を経由して生成コードにインクルードされます。

sctsdl.c

このファイルには、操作の実装とスケジューリングに使用される関数があります。特に、以下の関数があります。

- 動的エラーの処理と報告のための関数
- signal sending、create、stop、nextstate、set、reset などの操作のための関数やこれらのアクティビティを補完する関数
- 初期化と main ループ（スケジューラ）のための関数

scttypes.h

このファイルには、変数や関数の型定義と `extern` 宣言が含まれます。このファイルは、`sctsdl.c`、`sctpred.c`、`sctutil.c`、`sctda.c`、`sctos.c`、および C コード ジェネレータが生成する各 C ファイルによって、インクルードされます。このファイルは通常すべてのユーザー作成環境ファイルにインクルードし、データ型、操作、シグナル プリミティブが使用できるようにする必要があります。

sctutil.c

このファイルには、基本的な読み書きの関数に加えて、定義済みデータ型を含む抽象データ型の値の読み書きに使用する関数が含まれます。

生成された C プログラムとランタイム ライブラリを新規のターゲット プラットフォーム（コンパイラも含む）に移動するには、このファイルに変更を加える必要があります。`scttypes.h` に新しいセクションを作成し、新しいコンパイラのプロパティを記述する必要もあります。

ユーザー定義（カスタム）ライブラリの作成

コンパイル／リンクの際に、生成コードにカスタマイズされた プロパティを与えるため、新しいライブラリを作成するとよい場合があります。この新しいライブラリの作成プロセスは、Tau のグラフィカル フレームワークではサポートされていません。したがって、ホスト上で 使用できる適切なツール（シェル、テキスト エディタ、make、コンパイラ、リンカーなど）を利用し、アプリケーションを実行して想定どおりに動作するかどうか検証する必要があります。

このプロセスは以下の手順で実行します。

1. まず、サポートされるライブラリ（`appl_cl`、`applt_cl` など）や以前に作成したライブラリなどの既存ディレクトリの 1 つの内容をコピーします。目指している効果に類似したプロパティをすでに持っているライブラリを利用すると、以降の作業量が少なくてすみます。
2. 次に、`comp.opt` ファイルファイルの内容を変更し、ファイルを実行して以下に示すような想定どおりの結果が得られることを検証します。
   - 正しい `makeoptions/make.opt` ファイルがインクルードされているか
   - 正しいコンパイル スクリプトが使用されているか
   - 正しいリンク スクリプトが使用されているか
   - make を実行する正しいコマンドが使用されているか
   - コーダ ライブラリが正しくビルドされているか
3. 次に、`makeoptions/make.opt` ファイルの内容を変更し、環境変数、C マクロ、コンパイラ フラグを適切な値に変更します。
   - 生成 C コードのプリプロセス時に使用する C マクロの参照と、`include` ディレクトリにある定義は、第 30 章「C コード ジェネレータマクロ」にあります。
   - コーダとアーカイブの生成を定義するセクションは、通常、変更する必要はありません。
4. C コンパイラ ドライバをカスタマイズすることも可能です（任意）。
5. `comp.opt` ファイルファイルと `makeoptions/make.opt` ファイルを使用してアプリケーションをビルドし、動作が想定どおりかテストします。

注記

ライブラリの新しいバージョンを作成する場合、必ず、インクルードするファイルと生成コードの両方を同じコンパイル スイッチでコンパイルします。そうしないと、コンパイル済みアプリケーションが想定外の振る舞いや未定義の振る舞いをする場合があります。

コンパイラへの適合

このセクションでは、ソース コードを新しい環境に適合させるために変更する方法について説明します。具体的には、コードを新しいターゲット ハードウェアや OS に移動することや、新しいコンパイラを使用することが考えられます。

変更が必要となりうるソース コードは、以下の 2 つです。

- `scttypes.h` に、コンパイラのプロパティを定義するセクションがあります。ここに新しいコンパイラを追加できます。
- `sctos.c` に、オペレーティング システムやハードウェアに依存する関数が集められています。新しいコンパイラ、OS、ハードウェアに合わせてこれらを変更します。

scttypes.h のコンパイラ定義セクション

`scttypes.h` では、コンパイラのプロパティは、コンパイラや、コンパイラが設定した、コンピュータに依存するスイッチによって認識されます。

```
#if defined(__linux)
#define SCT_POSIX

#elif defined(__sun)
#define SCT_POSIX

#elif defined(__hpux)
#define SCT_POSIX

#elif defined(__CYGWIN__)
#define SCT_POSIX

#elif defined(QNX4_CC)
#define SCT_POSIX

#elif defined(__BORLANDC__)
#define SCT_WINDOWS

#elif defined(_MSC_VER)
#define SCT_WINDOWS

#else
#include "user_cc.h"

#endif
```

上記のセクションでは、まず UNIX-like/POSIX コンパイラと Windows コンパイラを区別しています。上のリストにないコンパイラの場合、`user_cc.h` ファイルを作成して自分で設定する必要があります。このファイルはターゲット ディレクトリに置くとよいでしょう。

上記コンパイラについて下表にまとめます。

コンパイラ	説明
__linux	Linux gcc
__sun	SUN のコンパイラ
__hpux	HP のコンパイラ
__CYGWIN__	Windows gcc。詳細については `http://sources.redhat.com/cygwin/` を参照してください。
QNX4_CC	QNX
__BORLANDC__	Windows Borland C/C++ コンパイラ
_MSC_VER	Windows Microsoft Visual C/C++ コンパイラ

このコンパイラ設定セクションの後に、一般設定セクションが続きます。

```
#if defined(SCT_POSIX) || defined(SCT_WINDOWS)
#define XMULTIBYTE_SUPPORT
#endif

#include <string.h>
#include <stdlib.h>
#include <limits.h>
#include <stdarg.h>
#ifdef XREADANDWRITEF
#include <stdio.h>
#ifdef XMULTIBYTE_SUPPORT
#include <locale.h>
#endif
#endif

#ifndef GETINTRAND
#define GETINTRAND      rand()
#endif
#ifndef GETINTRAND_MAX
#define GETINTRAND_MAX  RAND_MAX
#endif

#ifndef xptrint
#if (ULONG_MAX != UINT_MAX)
#define xptrint         unsigned long
#define X_XPTRINT_LONG
#else
#define xptrint         unsigned
#endif
#endif

#ifndef xint32
#if (INT_MAX >= 2147483647)
#define xint32          int
#define X_XINT32_INT
#else
#define xint32          long int
#endif
#endif
```

一般設定セクションでは、まず、マルチバイト文字のサポートを設定します。次に、標準インクルード ファイルをいくつかインクルードし、続いて乱数生成のプロパティを設定します。最後に、アドレスと同じサイズの `unsigned int` タイプを定義する `xptrint` と、32 ビットの `int` タイプを定義する `xint32` の 2 つのタイプを設定します。

sctos.c ファイルの変更

`sctos.c` には以下の重要な関数が定義されています。

```
extern void * xAlloc (xptrint Size);

extern void xFree (void **P);

extern void xHalt (void);

#ifdef XCLOCK
extern SDL_Time SDL_Clock (void);
#endif

#if defined(XCLOCK) && !defined(XENV)
extern void xSleepUntil (SDL_Time WakeUpTime);
#endif

#if defined(XPMCOMM) && !defined(XENV)
extern int xGlobalNodeNumber (void);
#endif

#if defined(XMONITOR) && !defined(XNOSELECT)
extern xbool xCheckForKeyboardInput (
  long xKeyboardTimeout);
#endif
```

これらの関数の一部には、SCT_POSIX、SCT_WINDOWS、およびその他の場合に対応する 3 つの異なった実装があります。「その他の場合」は、何も実行しない「空の実装」です。sctos.c の標準的な実装が特定のアプリケーションのニーズに対応しない場合、上記のどの関数も代替となるユーザー実装が可能です。マクロの定義によっては、対応する関数が `sctos.o` から削除されるため、次に示すように、ユーザーが実装する必要があります。

```
XUSER_ALLOC_FUNC
XUSER_FREE_FUNC
XUSER_HALT_FUNC
XUSER_CLOCK_FUNC
XUSER_SLEEP_FUNC
XUSER_KEYBOARD_FUNC
```

xAlloc

`xAlloc` 関数は動的メモリを割り当てるのに使用し、ランタイム ライブラリと生成されたコード全体で使用されます。この関数は、入力としてバイト単位でサイズが与えられ、要求されたサイズのデータ領域のポインタを返します。このデータ領域のすべてのバイトはゼロに設定されます。標準実装では、C 関数 `calloc` を使用します。

動的メモリの必要量を見積るのに、xAlloc にステートメントを導入し、呼び出し数と動的メモリの合計要求サイズを記録できます。ただし、注意すべき点がいくつかあります。

- モデル ベリファイヤ（Model Verifier）として動作するようにコンパイルされたプログラムは、アプリケーションとして動作するようにコンパイルされたプログラムよりも多くの動的メモリを必要とするため、見積りには適切なコンパイル スイッチを使う必要があります。
- calloc を呼び出すと、C ランタイム システムがメモリの割り当ての解除と再利用を行うために、実際の要求よりも大きいサイズのメモリが割り当てられます。追加のサイズはコンパイラに依存します。
- 割り当て要求時に使用可能なメモリがなくなった場合の処理を xAlloc に実装できます。xAlloc の標準的な実装では、calloc が 0 を返すかどうかをテストできるので、終了前にプログラムから適切なメッセージを出力させることができます。

xFree

xFree 関数を使用して、メモリを使用可メモリのリストに戻し、以降の xAlloc の呼び出しで再利用できるようにします。標準的な実装では、C 関数 free を使用します。動的メモリを使用するデータ型が使用されておらず、なおかつユーザーによる他の動的データが導入されていない場合は、この関数は使用しません。

xFree 関数のパラメータは、割り当てられたメモリを指すポインタのアドレスです。

例 **357 xFree** 関数の使用

```
unsigned char *ptr;
ptr = xAlloc(100);
xFree (&ptr);      /* NOTE: Not xFree(ptr); */
```

xHalt

xHalt 関数はプログラムを終了するのに使用します。標準的な実装では C 関数 exit を使用します。

SDL_Clock

SDL_Clock 関数は、OS またはハードウェアから読み込まれた現在の時刻を返します。戻り値は SDL_Time 型になります。アプリケーションが実時間との関連を必要としない場合（たとえばタイマーを使用しておらず、できるだけ速く実行する必要がある場合）、クロック関数は不要です。このような場合は、コンパイル スイッチを定義しないことにより、シミュレート時間を使用するとよいでしょう。これで、SDL_Clock が呼び出されることがなくなり、実装が不要になります。他の方法として、SDL_Clock が常に時間値 0 を返すようにすることもできます。

組み込みシステムでの代表的な実装では、ハードウェアがあらかじめ定義された間隔で割り込みを生成します。割り込みのたびに、現在時刻を示す変数が更新されます。この変数を SDL_Clock で読み込み、現在時刻を返すことができます。

注記

変数は、SDL_Clock がクロック変数を読み込む間、更新されないよう保護する必要があります。SDL_Clock がクロック変数を読み込んでいる間に割り込みルーチンを呼び出すと、システムに重大な不調が発生する場合があります。

SDL_Time

SDL_Time は、時間値の秒とナノ秒を示す 2 つの 32 ビット整数コンポーネントを含む構造体です。

```
typedef struct {
  xint32  s;       /* for seconds */
  xint32  ns;      /* for nanoseconds */
} SDL_Time;
```

xint32 は 32-bit int として実装されます。コンポーネント「s」と「ns」は、実装されたクロック関数に基づく過去のある時刻から経過した秒数とナノ秒数をそれぞれ表します。

xSleep_Until

xSleep_Until 関数は、SDL_Time 型の時間値を入力として与えられ、この時間が経過するまで実行を保留するものです。その後、リターンします。

この関数は、実時間が使用されている場合（スイッチ XCLOCK が定義されている）と環境関数がない（XENV が定義されていない）場合にのみ使用します。
xSleep_Until 関数は、イベントを発生できる環境がないときに、次のイベントがスケジュールされるまで待機するために使用します。

xGlobalNodeNumber

xGlobalNodeNumber 関数は、分散アプリケーションの一部である各システムに固有の番号を割り当てるために使用します。環境関数が使用できる場合は、環境関数内にこの関数を実装します。

xCheckForKeyboardInput

xCheckForKeyboardInput 関数は、キーボードで入力した行（stdin）があるかどうかを判定するのに使用します。この実装が困難な場合は、キーボードで入力された文字があるかどうかを判定させてもかまいません。この関数は、XMONITOR が定義されている場合のモデル ベリファイヤ（Model Verifier）だけが使用します。

xCheckForKeyboardInput 関数を使用して、<Return> を入力してステート遷移の実行を中断したり、モデル ベリファイヤ（Model Verifier）のコマンド プロンプトでプログラムが入力待ちになっている場合に環境のポーリングを処理したりできます。

26

C コードジェネレータの動的メモリ管理

このセクションでは、C コードジェネレータが生成するコードの動的メモリの割り当てと割り当て解除の方法について説明します。割り当て解除や使用可能メモリ リストによって動的メモリを再利用する方法や、アプリケーションに必要な動的メモリ所要量の合計の予測方法を解説します。

注記

C コードジェネレータフレームワーク内のエンティティ、特にメモリ割り当て、Mutex、セマフォア、またはその他の同期機構を、明示的または暗黙的に使用するエンティティは、割り込みルーチン、シグナルハンドラ、またはそのほか同様の関数の内部では使用すべきではありません。OS にも依存しますが、そのような使用はアプリケーションを誤動作させる可能性があります。

動的メモリ所要サイズ

動的メモリは、C コード ジェネレータが生成するアプリケーションのランタイム モデル中のオブジェクトに使用します。オブジェクトを以下に示します。

- アクティブ クラスのインスタンス
- シグナル インスタンスとタイマー インスタンス
- アクティブ クラスの操作のインスタンス
- Charstring、OctetString、BitString、ObjectIdentifier の各データ型を持つ属性
- String、Bag、一般的 Array、一般的 PowerSet の各データ型を持つ属性
- 動的メモリの使用を指定した、他のユーザー定義データ型を持つ属性

アプリケーションのメモリ所要量を予測するため、上で示したインスタンスのサイズと生成インスタンスの数に関する情報が必要です。算出されたサイズ情報は生成されたアプリケーションだけに当てはまります。モデル ベリファイヤ（Model Verifier）コマンドライン インタープリタを含むようなアプリケーションには当てはまりません。所要量を算出するにあたり、アプリケーションに含まれないコンポーネントは、与えられたデータ型定義から削除されます。

完全な定義は、`scttypes.h` ファイルにあります。

アクティブ クラス

アクティブ クラスの各インスタンスは、ヒープに割り当てられる 2 つの構造体によって表されます。`scttypes.h` では `xLocalPIdRec` タイプが、生成コードでは `yVDef_ProcessName` 構造体が定義されます。

```
typedef struct {
   xPrsNode      PrsP;
}  xLocalPIdRec;

typedef struct {
   xPrsNode       Pre;
   xPrsNode       Suc;
   int            RestartAddress;
   xPrdNode       ActivePrd;
   void (*RestartPAD) (xPrsNode  VarP);
   xPrsNode       NextPrs;
   SDL_PId        Self;
   xPrsIdNode     InstNameNode;
   xPrsIdNode     TypeNameNode;
   int            State;
   xSignalNode    Signal;
   xInputPortRec  InputPort;
   SDL_PId        Parent;
   SDL_PId        Offspring;
   int            BlockInstNumber;
   xSignalIdNode  pREPLY_Waited_For;
   xSignalNode    pREPLY_Signal;
/* parameters and attributes of active classes */
} yVDef_ProcessName;
```

上記の構造体のサイズを計算するには、構造体のコンポーネントについてさらに情報が必要です。xPrsNode、xPrdNode、xSignalNode、xPrsIdNode、xStateIdNode、xSignalIdNode の各データ型はすべてポインタで、SDL_PId は int 型 1 つとポインタ型 1 つを含む構造体です。xInputPortRec はポインタ型 2 つと int 型を 1 つ含む構造体です。

したがって、コンパイラが一般的でない配置（Alignment）ルールを使用しない限り、以下の式で xLocalPIdRec および xPrsRec 構造体のサイズを計算できます。

$$Size_{xLocalPIdRec} = Size_{address}$$

$$Size_{xPrsRec} = 15 \cdot Size_{address} + 7 \cdot Size_{int}$$

yVDef_ProcessName のサイズは xPrsRec のサイズにアクティブ クラスの属性とパラメータのサイズを加算したものです。C システムによって導入されたオーバーヘッドがあれば、それも加算します。状態機械のパラメータとその属性のサイズは、アクティブ クラスでの宣言によって異なります。

システム中のアクティブ クラスの各インスタンス セットごとに、2 種類の異なる構造体が割り当てられます。

- 各生成インスタンスに xLocalPIdRec が 1 つ割り当てられます。この構造体は、かつて存在したアクティブ クラスのインスタンスの識別子として使用されるため、再利用されません。
- アクティブ クラスのインスタンス セットの最大並行実行インスタンス数（プログラムを完全に実行する間の最大数）と同じだけの yVDef_ProcessName 構造体が割り当てられます。

yVDef_ProcessName 構造体は、使用可能メモリ リストを使用して再利用されます。この構造体は、表現対象のアクティブ クラスのインスタンスの stop アクション実行時に使用可能メモリ リストに入れられます。アクティブ クラスの各タイプに使用可能メモリ リストが 1 つあります。インスタンスを生成する場合、ランタイム ライブラリはまず使用可能メモリ リストを参照し、リストのアイテムを再利用します。使用可能メモリ リストが空の場合のみ新しいメモリが割り当てられます。

注記：

コンパイル スイッチ XPRSOPT を定義すると、xLocalPIdRec が xPrsRec とともに再利用されます。xLocalPIdRec には、さらに int が含まれます。

シグナル

シグナルは、アクティブ クラスとほぼ同様に処理されます。シグナル インスタンスは、1 つの構造体で表します。

```
typedef struct {
   xSignalNode   Pre;
   xSignalNode   Suc;
   int           Prio;
   SDL_PId       Receiver;
   SDL_PId       Sender;
   xIdNode       NameNode;

   /* Signal parameters */
} ySignalPar_SignalName;
```

この構造体型には、各シグナル パラメータに対応するコンポーネントが 1 つ含まれます。コンポーネントのデータ型は、パラメータのデータ型の変換バージョンになります。

すなわち、パラメータを持たないシグナルの構造体と同じ xSignalRec のサイズは以下の式で計算できます。

$$\mathrm{Size}_{\texttt{xSignalRec}} = 5 \cdot \mathrm{Size}_{\mathrm{address}} + 3 \cdot \mathrm{Size}_{\texttt{int}}$$

したがって、ySignalPar_SignalName 構造体のサイズは xSignalRec のサイズにパラメータのサイズを加算したものと同じになります。

システム中のシグナル型ごとに、以下に示すデータ領域が割り当てられます。

- 送信済みだがまだ受信されていないシグナルの最大数（プログラムを完全に実行する間の）と同じ数の ySignalPar_SignalName 構造体が割り当てられます。

ySignalPar_SignalName 構造体は、表現対象のシグナル インスタンスが受信されると戻される使用可能メモリ リストを使用して再利用されます。シグナル インスタンスが使用可能メモリ リストに戻されるのは、シグナル インスタンスによる遷移が nextstate アクションまたは stop アクションによって終了したときです。各シグナルに使用可能メモリ リストが 1 つあります。シグナル インスタンスを生成する場合、たとえばシグナル送信操作中、ランタイム ライブラリはまず使用可能メモリ リストを参照し、可能であればメモリを再利用します。使用可能メモリ リストが空の場合のみ新しいメモリが割り当てられます。

注記

パラメータを持たないすべてのシグナルに共通の使用可能メモリ リストが 1 つあります。

タイマー

タイマーに必要なメモリはシグナルの場合と同様に計算できます。ただしタイマーごとに他に SDL_Time コンポーネントが含まれることを考慮する必要があります。

アクティブ クラスの操作

アクティブ クラスとアクティブ クラスの操作には、メモリ割り当てに関して共通点が多くあります。アクティブ クラスの操作は、呼び出しから戻りまでの存在期間では、yVDef_ProcedureName 構造体によって表されます。

```
typedef struct {
   xPrdIdNode  NameNode;
   xPrdNode    StaticFather;
   xPrdNode    DynamicFather;
   int         RestartAddress;
   int (*RestartPRD) (xPrsNode  VarP);
   xSignalNode pREPLY_Signal;
   int         State;

   /* Formal parameters and attributes */
} yVDef_ProcedureName;
```

この構造体型には、各仮パラメータまたは属性に対応するコンポーネントが 1 つ含まれます。コンポーネントのデータ型は、パラメータのデータ型の変換バージョンになります。ただし、IN/OUT パラメータの場合はアドレスとして表現されます。

属性や仮パラメータを持たない操作と同じ `xPrdRec` 構造体のサイズは、次の式で計算できます。

$$\mathrm{Size}_{\texttt{xPrdRec}} = 5 \cdot \mathrm{Size}_{\mathrm{address}} + 2 \cdot \mathrm{Size}_{\texttt{int}}$$

`yVDef_ProcedureName` 構造体のサイズは `xPrdRec` のサイズに、操作で定義された仮パラメータと属性のサイズを加算したものです。

システム中の操作のタイプごとに、以下に示すデータ領域が割り当てられます。

- 操作の最大並行呼び出し数（プログラムを完全に実行する間の）と同じ数の `yVDef_ProcedureName` 構造が割り当てられます。並行呼び出しは、アクティブクラスの 1 インスタンス内で操作が自身を再帰的に呼び出す場合と、一時点で同じアクティブ クラスの複数のインスタンスが同じ操作を呼び出す場合の両方に発生します。

`yVDef_ProcedureName` 構造体は、操作インスタンスの戻り（return）アクションを実行すると戻される使用可能メモリ リストを使用して再利用されます。各操作型に使用可能メモリ リストが 1 つあります。操作のインスタンスを生成する場合、つまり呼び出し操作時、ランタイム ライブラリはまず使用可能メモリ リストを参照し、可能であればメモリを再利用します。使用可能メモリ リストが空の場合のみ新しいメモリが割り当てられます。

定義済みデータ型

定義済みデータ型 Charstring は、C で `char *` として実装され、動的メモリ割り当てを必要とします。定義済みデータ型 BitString、OctetString、ObjectIdentifier も動的メモリを使用して実装されます。

Charstring、BitString、OctetString、ObjectIdentifier の各ソート（データ型）の実装は、長さが動的に変化し、なおかつすべてのメモリが再利用できます。

使用されていないメモリをリリースするには、`sctos.c` ファイルの `xFree` 関数を呼び出します。これは、標準関数 `free` を使用してメモリをリリースするものです。

Charstring、BitString、OctetString、ObjectIdentifier は、構造体または配列の一部であっても正しく処理されます。たとえば、Charstring コンポーネントを持つ構造体に新しい値を与えると、古い Charstring 値はリリースされます。これらのデータ型のいずれかを含むすべての構造体型に対し、構造体変数のすべての動的メモリをリリースするのに使用する `Free` 関数もあります。

メモリ管理の実装

メモリの割り当てと割り当て解除は、sctos.c ファイルの xAlloc 関数と xFree 関数によって処理されます。このファイルの関数は、オペレーティング システムまたはハードウェアに向けて生成アプリケーションを適用する際に利用します。生成されたコードとランタイム ライブラリにおいて、メモリが必要になったり、リリース可能になった場合は、それぞれ xAlloc 関数または xFree 関数が使用されます。

sctos.c ファイルの詳細については、第 25 章「C および AgileC ランタイム ライブラリ」の 922 ページ、「sctos.c ファイルの変更」を参照してください。

割り当てと割り当て解除の関数

xAlloc

```
void* xAlloc(xptrint Size)
```

xAlloc 関数は、必要なメモリサイズをバイト単位指定のパラメータとして受信し、要求されたサイズのデータ領域のアドレスを返します。**データ領域のすべてのバイトはゼロに初期設定されます**。

xFree

```
void xFree(void** P)
```

xFree 関数は、ポインタへのアドレスを使用して、そのポインタが参照するデータ領域を使用可能メモリのプールに返します。**同時にそのポインタの値をゼロに設定します**。

実装面について

xAlloc 関数と xFree 関数は通常、割り当て（malloc、calloc）や割り当て解除（free）のための C の標準関数で実装します。万が一 sctos.c のデフォルト実装では不十分な場合は、指定されたインターフェイスさえ実現していれば、他の実装を提供してもかまいません。

注記

デフォルト以外の実装を指定する場合は、データのアドレスを返す前に、独自の xAlloc では必ずデータをゼロに初期設定し、xFree ではポインタの値をゼロに設定してください。

メモリの断片化

メモリの断片化は、プログラムが、サイズの異なる複数のデータ領域をランダムに割り当て、割り当て解除する場合に発生する現象です。こうなると、サイズが小さいために割り当て要求に当てはまらないメモリの断片が発生し、アプリケーションからのメモリ要求が徐々に増えていきます。

メモリの断片化を防止するためには、メモリの割り当て解除の実装を必要としない使用可能メモリ リストを、ほとんどの状況で使用します。

メモリの割り当て解除はデータ型のみに適用されます。以下を参照してください。

- Charstring
- BitString
- OctetString
- ObjectIdentifier
- String（#STRING ではない）と Bag テンプレートで作成されたデータ型
- Array テンプレートで作成されたデータ型。ただし、C の配列が使用できないようなインデックス型の場合。
- PowerSet テンプレートで作成されたデータ型。ただし、コンポーネント タイプのプロパティが一般的配列のインデックス型と同じ場合。

つまり、上記データ型の属性が使用されておらず、メモリの割り当て解除が必要なコーディングになっていない場合は、メモリ割り当て解除はまったく行われません。この場合、`xFree` 関数の実装は不要です。

メモリ所要量のトレース

動的メモリの所要量のトレースは簡単です。すべてのメモリ割り当ては `xAlloc` 関数によって実行されており、この関数のソースコード（`sctos.c` 内）が利用可能なので、適切な count 文または printout 文を使用してトレースする仕組みを作成できます。

27

C コード ジェネレータ リファレンス

この章は、C コード ジェネレータのリファレンス ガイドであり、UML から C のコードを生成する原理を説明します。特に次のことを説明します。

- 操作の原理
- ランタイム セマンティック（タイム、スケジューリング、シグナリングなど）の実装
- UML の定義済みデータ型の解釈
- Tau 独自のデータ型拡張の定義済み UML データ型への解釈
- 操作へのパラメータの受け渡し
- データ型のジェネリック関数
- 生成されたコード内の名前

C コード ジェネレータ 操作原理

C コード ジェネレータは、ビルド タイプ［C Code Generator］と［Model Verifier］（モデル ベリファイヤ）が使用します。

参照

第 19 章「ビルドとコード生成の概要と例」の 721 ページ、「ビルド タイプの使用」

C コード ジェネレータのオプションと設定

C コード ジェネレータは、ビルド アーティファクトの使用により、ビルドするアプリケーションにグローバルな設定を行います。このような設定の例の 1 つとして、コンパイル後に生成コードとリンクして使用する、ターゲット ライブラリとランタイム ライブラリの指定があります。

ビルド タイプ［C Code Generator］では、モデルの個々の要素にオプションを適用することもできます。例として、ビルドからのモデル要素の除外、外部 C および C++ コードの宣言や定義をインクルードするときに発生する命名や言語関連問題の解決などがあります。

C コード ジェネレータの起動

インタラクティブ モード

GUI から C コード ジェネレータを起動するには、以下の手順を行います。

1. 現在のビルドに適した設定を含むビルド アーティファクトを作成します。
   - ステレオタイプ << C Code Generator >> には、C コード ジェネレータの設定を定義する属性が含まれています。
2. ビルド アーティファクトにビルド ルートを適用します。ビルド ルートは、ビルドするモデル内の最上位レベルのアクティブ クラスを指定します。
3. 必要な場合、ビルドから除外する要素にはタグをつけ、外部 C または C++ コードで定義されている要素には適切な名前および言語関連属性を適用します。
   - これにはステレオタイプ << C Application >> を使用します。
4. ビルド アーティファクトで［C Code Generator］ビルド タイプを選択します。
5. コードを生成するには、次の手順を行います。
   - ビルド アーティファクトを右クリックし、メニューの［ビルド］をクリックします。

上の操作の結果、ユーザーの介入を必要とせずに次の操作が行われます。

1. モデルのビルド ルートと指定されているパートが SDL の中間表現に変換されます。内部表現は SDL 型の構文で表現されます。
   - 属性 Generate C code を持つ要素は、false に設定され破棄されます。
   - 外部宣言のインクルードとその後の名前付け問題は、<< C Application >> ステレオタイプにある設定で処理します。
2. エクスポートされた内部表現に対して、C コードを生成する前に C コード ジェネレータがセマンティック上正しいかどうかをチェックします。
3. C コード ジェネレータが C コードを生成します。
   - コード生成では、ビルドに指定された設定を適用します。
   - コード生成変換ルールについては、本書の主 セクションで説明しています。
4. 生成された C コードは、Target directory のファイルに書き出されます。
5. C コード ジェネレータが make ファイルを作成します。この make ファイルは、生成されたコードとビルド アーティファクトで指定されたランタイム ライブラリが適切にコンパイルされリンクされることを保証します。ビルド アーティファクトで make テンプレート ファイルを指定してコンパイルとリンク スキームに外部コードを追加することもできます。
   - これらの設定は、ステレオタイプ << C Code Generator >> が管理します。
6. 生成された make ファイルは、ビルドの最終ステップとして実行されます。

バッチ モード

C コード ジェネレータは、インタラクティブ ビルドと同じ原理に従って、同じ種類のオプションを使用してバッチ（コマンドライン）モードでも操作できます。唯一の違いは、メッセージがメッセージ エリアに出力される代わりに `stdout` に書き出されることだけです。

コマンド ライン プロンプトから C コード ジェネレータを起動するには、`taubatch` コマンドを使用します。

参照

第 11 章「SDL のインポート」の 536 ページ、「サポートされる SDL」

第 20 章「アプリケーション ビルド リファレンス」の 800 ページ、「インタラクティブ ビルド インターフェイス」

第 20 章「アプリケーション ビルド リファレンス」の 811 ページ、「バッチ ビルド インターフェイス」

ランタイム セマンティックの実装

時間

C コード ジェネレータが生成するアプリケーションは、時間の処理に関して 2 つの方法で実行できます。

- シミュレートされた時間
- 実時間

シミュレートされた時間

シミュレートされた時間の利用は、モデル ベリファイヤ（Model Verifier）を使ったシミュレーション / デバッギングのセッションを行うためのもっとも便利な方法です。ただし、シミュレーションにおける時間が実時間と対応しなくなります。ここでは実時間の代わりに、ディスクリート イベント シミュレーション技法が使われます。この技法は、シミュレーション時間の現在値（Now）は現在実行しているイベントがスケジュールされている時間と同じであるという概念に基づいています。1 つのイベントが終了すると、シミュレーション時間は次のイベントがスケジュールされる時間にまで増やされ、このイベントが開始します。イベントには、状態機械内の遷移、タイマー出力、および環境からシステムへ送られるシグナルがあります。

たとえば、ディスクリート イベント シミュレーション技法を使うと、次のイベントが今から 1 時間後にスケジュールされているタイマー出力であっても、次の遷移の実行が許されていれば、そのタイマー出力が直ちに起きます。シミュレーション時間は 1 時間増分されますが、ユーザーは 1 時間待つ必要はありません。

実時間

Simulation kind の Real-time オプション時間を使用すると、実行しているプログラムのクロックと実時間の間には連係ができます。上の例で Go コマンドを指定した場合、タイマー出力があるまで 1 時間待つ必要があります。実時間を使用するには、オペレーティング システムが提供するクロック機能を使用します。

次の遷移が将来にスケジュールされたタイマーの場合（現在より 2 秒以上先）、次の遷移コマンドはシミュレーションを 1 秒間実行して停止します。

スケジューリング

システム内のアクティブ クラスの振る舞いは、アクション、分岐、シグナル送信、操作の呼び出しなどのアクションで構成される遷移を実行する、状態機械によって実装されています。遷移には時間がかからないこと、したがって出力操作が行われると直ちに受信者の入力ポートにシグナル インスタンスが置かれることを前提としています。

遷移は、アプリケーションのデバッグにおいてレディ キュー（待ち行列）を手動で再配置（モデル ベリファイヤ（Model Verifier）が提供する適切なコマンドが使用可能）して、別の遷移を実行するような場合を除き、常に優先的（preemptive）に実行されます。中断された遷移は、後に終りまで実行できます。

UML 自身は実行ストラテジを何も定義していないので、選択されたストラテジが許可されるストラテジになります。ただし、それが実行できる唯一のストラテジとは限りません。

結論として、やはり実行ストラテジに従うモデル ベリファイヤ（Model Verifier）は、異なるインスタンスのアクションの時間または順序が極めて重要な「タイミング効果」のシミュレーションには直接的には適していません。

例 **358:** 予測できない状況でのスケジューリング

同じ遷移内で 2 つのシグナル インスタンスを送るアクティブ クラス A があり、1 つはアクティブ クラス B のインスタンスへ、もう 1 つはインスタンス C へ送ると仮定します。B と C への対応する遷移の最中に、シグナル インスタンスがアクティブ クラス D のインスタンスへ送られます。

アプリケーションの振る舞いが D の入力ポートでシグナル インスタンスを受け取る順序に依存している場合、これはアクティブ クラスのインスタンスの実行速度とシグナルの遅延が振る舞いを決定する「予測できない」状況といえます。

レディ キュー（待ち行列）

レディ キュー（待ち行列）は、遷移を起こすことができるシグナルを受け取っているがまだその遷移を完了していないアクティブ クラスのすべてのインスタンスを含みます。レディ キュー（待ち行列）は、優先順および挿入時間の順に並んでいます。

アクティブ クラスの優先順位は、次の遷移を発生するシグナルの優先順位です。レディ キュー（待ち行列）は、まず優先順位（優先順位の値が大きいほど優先順位は低い）に従って配置され、次に挿入時間に従って配置されます。

レディ キュー（待ち行列）の優先順位

アクティブ クラスのインスタンスは、同じ優先順位を持つインスタンスのうちの最後に挿入されます。プリエンティブ スケジューリングは使用しないのでアクティブ クラスのインスタンスが現在実行しているインスタンスの前に挿入されることはありません。

- アクティブ クラスのインスタンスが、直ちにシグナルを受信できる別のインスタンスへシグナルを送る場合、受信インスタンスはレディ キュー（待ち行列）の同じ優先順位を持つアクティブ クラスのインスタンスのうちの最後に挿入されますが、現在実行しているインスタンスの前に挿入されることはありません。

- アクティブ クラスのインスタンスの振る舞いを実装する状態機械が現在次のステートを実行していて、引き続き直ちに別の遷移を実行できる場合、そのインスタンスは同じ優先順位を持つ最後のインスタンスとしてレディ キュー（待ち行列）に挿入されます。つまり、そのインスタンスはレディ キュー（待ち行列）内の最初のインスタンスとして残ることができます。
- タイマー出力を受け取る状態機械が受信シグナルの応答として直ちに遷移を実行できる場合、そのインスタンスは最後に挿入されます。

Public 属性

C コード ジェネレータは、public 属性を解釈する方法として、public 属性のメモリ領域に直接アクセスする方法をとります。属性のアドレスは、カーネルのユーティリティ関数を使用して計算され、public 属性へのアクセスまたは変更に直接使用されます。

注記

スレッド内部では問題ありませんが、スレッド アプリケーションで他のアプリケーションの属性にアクセスする場合は、この方法は適していません。稼動する OS によってスレッド間でのメモリへのアクセスが阻害されるか、あるいは複数スレッドから同時にメモリにアクセスすることにって問題が発生する可能性があります。スレッド アプリケーションでは、変数の共有は set および get 操作を使用して実行することを推奨します。この方法によって共有変数のアクセスと更新の同期がとれ、更新による問題を防ぐことができます。

ガードとトリガされた遷移のガード

ガードとトリガされた遷移のガードは、UML への追加概念です。この概念のモデルの１つとして考えられるのは、反復シグナル送信を使用した式の繰り返し計算です。しかし、このモデルはシミュレーションやデバッギング セッションの場合には適さず、当然ながらアプリケーションでも使用できません。したがって、これは概念の説明している振る舞いに近い実装ストラテジとして使われます。

実装

まず、ガードとトリガされた遷移のガードのうち動的なものと静的なものを区別します。

- 静的ガードは与えられた値を持つ式を含み、対応する式を nextstate 操作で計算する必要がある以外は実装上の問題や実行上のオーバーヘッドはありません。
- 動的ガードは、対応する状態機械がそのステートで待機しているとき値を変えることができる式を含みます。式は、状態機械がステートメントを何も実行しないにも関わらず値を変更できる部分を含みます。たとえば、public 属性を使用している場合や、ガード式内の「Now」などがあります。

動的ガードは繰り返し再計算する必要があります。これらの式に選択したストラテジは、任意の状態機械が実行する各遷移またはタイマー出力の後と、さらに遷移内でモデル ベリファイヤ（Model Verifier）コマンドライン インタープリータに入る前に式を再計算します。

動的ガードを含むステートで待機している各状態機械は、他の状態機械が実行する各遷移またはタイマー出力の間に暗黙の次のステート操作を実行します。

定数属性

定数属性または読み取り専用属性は C マクロ（`#define`）または C 変数として実装されています。

- マクロに変換するには、属性の値を定義する式が以下を満たす必要があります。
  - 定義済みソート（Integer 型、Real 型など）の 1 つでなければならない。
  - 定義済みソートで定義されているリテラルと操作およびコードを生成するとき計算可能な他の定数属性のみ含む。
- パッケージ内の他の定数属性はすべてプログラム起動時に値を与えられる C 変数として実装されています。

この問題は、Array（配列）と PowerSet の実装に関連するために提起しています。これらの概念に対してそれぞれ 2 つの実装方法があります。

- 配列は C の配列に変換するか C のリストにリンクできます。
- PowerSet は、ビット配列に変換するかリストにリンクできます。

変換方法は、インデックス タイプを見て選びます。インデックス タイプが 1 つの制限された範囲を持つシンタイプならば、配列とビット配列スキームを使用し、そうでなければリンクされたリストを使用します。

C 変数に変換される定数属性がシンタイプの範囲条件に使われていてシンタイプが配列または PowerSet インスタンス化のインデックス ソートに使われている場合、その配列または PowerSet の実装にリンクされたリストが使われます。この理由は、配列の長さが C の変数に依存できないためです。

外部定数属性

外部定数属性はシステムのパラメータ化に使用できます。したがって生成されたプログラムをパラメータ化することもできます。外部定数属性に使用する値は、生成されたプログラムが起動時に読み込むか、生成されたコードにマクロ定義として含めることができます。C コード ジェネレータは、この両方に対応しています。プログラムをコンパイルするまで各属性にどの方法を使用するか選択する必要はありません。

属性値の指定に C マクロの使用

パッケージ内の属性値の指定に C のマクロ定義を使用するには、次の手順を行います。

1. 属性を外部と宣言し、対応するマクロ定義をファイルに書き出します。

例 **359:** マクロ定義 ——————————————————————————————

外部定数属性の UML パッケージ宣言：

```
const Integer extern attribute1;
```

```
const Real extern attribute2;
```

属性名は接頭辞のない UML 名で、文字、数字、アンダースコア以外の文字を除いたものです。

```
#define attribute1 3
#define attribute2 3.14
```

定義済みの値（3, 3.14）が属性の型（Integer, Real）に適合していることを確認する必要があります。

---

2. 属性が属するパッケージを選択します。
3. プロパティ エディタを開き［フィルタ］ドロップダウン メニューから［C Application］を選択します。［C Application］の選択肢がない場合、［カスタマイズ］ダイアログダイアログの［アドイン］タブを使用してアドインを起動する必要があります。
4. マクロ定義のあるファイルの名前とパスを、［Include File］フィールドに入力します。パスとして絶対パス、またはターゲット ディレクトリへの相対パスのどちらを指定してもかまいません。

注記

ファイルに格納されている属性値を読む関数はないのでアプリケーションを作成したら、パッケージ内のすべての属性にマクロ定義を使用する必要があります。

プログラム起動時の属性値の読み込み

パッケージの外部定数属性の値を与えるもう 1 つの方法は、プログラム起動時に値を読み込むことです。対応するマクロ定義を持たない外部定数属性がある場合、残りの属性の値をキーボードから与えるか値を含むファイルを使用するかのいずれかを選ぶことができます。

アプリケーションを起動すると、モデル ベリファイヤ（Model Verifier）に次のプロンプトが表示されます。

```
External file :
```

- Model Verifier コンソールで <Return> を押して、値を端末から読み込むことを指定します。
- あるいは、値を含むファイルの名前を入力して <Return> を押します。

値を端末から読み込む場合、各値ごとにプロンプトされます。ファイルを指定する場合、次の例のように属性名と対応する値を含むファイルを作成しておく必要があります。

例 **360:** プログラム起動時の値 ———————————————————————

```
attribute1 value1
attribute2 value2
```

属性は任意の順序で定義できます。

値が返る操作の呼び出し

C コード ジェネレータでは、データ型、パッシブ クラス、またはアクティブ クラスの、値が返る操作の呼び出しは、その操作呼び出しを含むステートメントの直前に特別の呼び出しを挿入して行います。呼び出しの結果は匿名変数に格納され、それが式で使用されます。

注記

値を返す操作呼び出しは、呼び出しの最後に操作結果の新 IN/OUT パラメータを追加することにより通常の呼び出しに変換されます。

任意の値演算子（**any**）

any には 2 種類の用途があります。次の書き方ができます。

- 分岐内の any
- 式内の any (SortName)

注記

any 演算子は、モデル ベリファイヤを使用したシミュレーション用にのみ使用します。C コード ジェネレータで生成されたアプリケーションには使用すべきではありません。

分岐内の **any**

分岐内の any は、デバッギングおよびシミュレーション用にのみ使用すべきであり、従うパスを選ぶ機会を与える質問によって実装します。

式内の **any (SortName)**

式内の any (SortName) は、乱数ジェネレータを使用して特定の型の乱数を生成して実装します。

注記

Sort がシンタイプの操作呼び出し any(Sort) は、シンタイプが a:b 形式（1 つの制限された範囲）の範囲条件を最大 1 つ含む場合にのみ実装されます。

参照

第 4 章「UML 言語ガイド」の 299 ページ、「任意値（any）式」

データ型の解釈

概要

C 定義の実装

C コード ジェネレータが、定義済みおよび Tau 独自拡張の UML 向けに使用するランタイム ライブラリによって提供される C と C のマクロの実装は、ファイル `sctpred.h` にあります。ただし、Pid sort はファイル `scttypes.h` にあります。

以降のサブセクションのすべての例では、C コード ジェネレータが C の名前に付ける接頭辞は表示しません。（これらの接頭辞は、生成されたプログラムで名前が重複しないことを保証するために追加されます。接頭辞と接尾辞の詳細は、以下を参照してください。

参照

生成された C コードの名前

初期値

初期値は、UML で指定された初期値を持たないすべての属性に割り当てられます。その属性は 0 への `memset` を使用して 0 に設定されます。

注記

C コード ジェネレータでは、名前クラス リテラルまたは文字列を使用してリテラルに名前を付けることはできません。

CPtr

このテンプレートは、テンプレート Own および ORef と同様に、異なるプロパティを持つポインタを示します。これらはすべて C のポインタに変換されます。

CPtr には次の操作があります（定義より）。

```
public <<External="true">> void SetValue( T);
public <<External="true">> T GetValue();
template<type T1> public static <<External="true">> T1 GetValue(CPtr<T1>);
template <type T1>public static <<External="true">> CPtr<T1> GetAddress(T1
entity);
public <<External="true">> T '[]'(CPtr<T>, Integer);
public CPtr<T> '¥+'(CPtr<T>, Integer);
public CPtr<T> '¥-'(CPtr<T>, Integer);
public static <<External="true">> void free(CPtr<T>);
```

以下は `CPtr<char>` または `char*` のみのためにサポートされます :

```
template<type T1> public static <<External="true">> CPtr<T1> malloc(Integer);
```

Array（配列）

配列テンプレートのインスタンス化は、次の制限付きで C コード ジェネレータ が処理します。配列のコンポーネントとインデックス ソートには C コード ジェネレータが扱える任意のソートが可能ですが、配列型自体を直接または間接的に参照することはできません（Struct（構造体）の扱いを説明している前のセクションも参照してください）。

インデックス ソートが値の閉じた間隔を持つディスクリート ソートの場合、UML 配列は C の配列である要素を含む構造体に変換されます。以下がディスクリート ソートです。

- Character（文字）
- Boolean（ブール値）
- Octet（8 ビット）
- Bit（ビット）
- 列挙型と見なされるソート
- 任意の Integer（整数）、Character（文字）、Boolean（ブール値）、Octet（8 ビット）、Bit（ビット）、または列挙型のシンタイプ。サブタイプは、値の閉じた間隔を指定する 1 つの範囲条件のみ持つことができます。

インデックス ソートが上記のリストのソートのいずれでもない場合、UML 配列はリンクされたリストに変換されます。リストの先頭には、可能なすべてのインデックスのデフォルト値があり、リストの要素にはデフォルト値と等しくないコンポーネント値を持つ各インデックスの値のペア（`index_value`、`component_value`）があります。

例 **361:** 配列

```
syntype MyInt= Integer constants( 1..10 );
  class Arr {
      Array <MyInt, Real> MyArray;
 }
```

上記は以下に変換されます。

```
typedef SDL_Integer MyInt;
typedef struct {
 SDL_Real A[10];
} Arr;
```

Bag

Bag テンプレートは PowerSet に似ています。ただし、bag は同じ値を持つ複数の要素を含むことができます。bag は、そのメンバーである各値に対して 1 つの要素を持つリンクされたリストに変換されます。各要素は、値とこの値のオカレンスの数を含みます。

Charstring

Charstring 型は、一般的にテキスト文字列を示すために使われます。Charstring は、C で `char*` と示されます。

注記

Charstring 型では、コード ジェネレータによってインデックス 0 に追加の文字が挿入されています。この文字は、ランタイム システムが内部で使用するので、アプリケーション レベルで C の Charstring 変数を指定するときは無視する必要があります。

Choice

Choice は、暗黙のタグで共用体を示すとき使用します。

例 **362: Choice**

```
choice Str {
  Integer a;
  Boolean b;
  Real c;
}
```

上記は以下に変換されます。

```
typedef enum {a, b, c} StrPresent;
typedef struct {
  StrPresent Present;
  union {
    SDL_Integer a;
    SDL_Boolean b;
    SDL_Real c;
  } U;
} Str;
```

`Present` コンポーネントは、choice のコンポーネントに値が与えられると C コード ジェネレータが自動的に設定します。

注記

モデル ベリファイヤ（Model Verifier）を使用したシミュレーションやデバッギング セッションの実行中、コンポーネントにアクセスしようとしたとき、実行時にそのコンポーネントが「存在する」かどうかがテストされます。存在しない場合には、ランタイム エラー メッセージが出力されます。

Enum

リテラル リストを含むソートは列挙型と見なされます。944 ページの例 363 を参照してください。このような型は、リテラルが 0、1、2... と続く「定義」のリストとともに `int` に変換されます。

例 **363:** 列挙型

```
enum EnumType {Lit1, Lit2, Lit3}
```

上記は以下に変換されます。

```
typedef XENUM_TYPE EnumType;
#define Lit1 0
#define Lit2 1
#define Lit3 2
```

マクロ XENUM_TYPE は、ファイル sctpred.c で以下のように定義されています。

```
#ifndef XENUM_TYPE
#define XENUM_TYPE int
#endif
```

これはすべての enum 型が int 型になることを意味します。ただし、マクロ XENUM_TYPE がユーザーによって再定義（たとえば unsigned char に）されている場合を除きます。256 個以上の値を持つ enum 型は常に int 型でマクロ XENUM_TYPE の影響を受けません。

ORef

このテンプレートおよびテンプレート Own は異なるプロパティを持つポインタを示します。これらはすべて C のポインタに変換されます。

Own

上記の ORef を参照してください。

PowerSet

PowerSet のインスタンス化は、次の制限付きで C コード ジェネレータが処理します。コンポーネント ソートには、C コード ジェネレータが扱える任意のソートが可能ですが、直接または間接的に PowerSet 型自体を参照することはできません。

PowerSet には 2 つの解釈スキームがあります。コンポーネント ソートが 943 ページの「Array（配列）」 で述べているインデックス ソートの条件を満たす場合、32 ビット整数の配列が使われます。各ビットは、PowerSet のメンバーであるか否かにかかわらず特定の要素を示すために使われます。そうではない場合、セットのメンバーであるすべての要素のリンクされたリストを使用して PowerSet を示します。

String（文字列）

文字列のインスタンス化は、次の制限付きで C コード ジェネレータが処理します。コンポーネント ソートには、C コード ジェネレータが扱える任意のソートが可能ですが、直接または間接的に文字列型自体を参照することはできません。

文字列は、文字列値の各要素に対して 1 つの要素を含むリンクされたリストに変換されます。

Struct（構造体）

パッシブ クラスは、946 ページの例 364 に示すように C の `struct` に変換されます。

例 **364:** 構造体

```
class Str {
 Integer a;
 Boolean b;
 Real c;
}
```

上記は以下に変換されます。

```
typedef struct {
  SDL_Integer a;
  SDL_Boolean b;
  SDL_Real c;
} Str;
```

構造体のコンポーネントには、C コード ジェネレータが扱える任意のソートが可能です。ただし、コンポーネントは構造体ソート自体を直接または間接的に参照することはできません。たとえば、上記のソート `Str` はソート `Str` のコンポーネントを持つことはできません。このような場合、C struct への変換は無効になります。

Syntype（シンタイプ）

C コード ジェネレータが扱える任意のソートに対して、ソートに新しい名前を与えてシンタイプを定義できます。値の許容範囲を制限する範囲条件も可能です。

シンタイプは、`typedef` を使用して親型に等しい型に変換されます。シンタイプ属性に正当な値だけが割り当てられていることを確認する検査が型定義とともに生成されるテスト関数に実装されています。このような属性に不正な値を割り当てようとするとランタイム時に動的エラーとして扱われます。シンタイプが配列でインデックス ソートとして使われ、生成される型が C で配列になる場合、インデックス値が配列コンポーネント選択でその範囲内にあることをチェックするテスト関数も生成されます。

操作へのパラメータの受け渡し

パフォーマンス上の理由からデータ型は 2 つのグループに分けられています。

- 値として渡される、単純でサイズの小さな型。
- 参照（アドレス）として渡される、構造化された、よりサイズの大きな型。

値として渡される型

次の型は値として渡されます（単純型）。

- Integer（整数）
- Real（実数）

- Natural（自然数）
- Boolean（ブール値）
- Character（文字）
- Time（時間）
- Duration（持続時間）
- Pid
- Charstring（文字列）
- Bit（ビット）
- Octet（8 ビット）
- IA5String（IA5 文字列）
- NumericString（数値文字列）
- PrintableString（出力可能文字列）
- VisibleString（可視文字列）
- NULL
- 列挙型
- テンプレート Own、ORef のインスタンス化。

以下も値として渡されます。

- 上記のリスト内の型の任意のシンタイプ
- このリストの型を継承する型。

アドレスとして渡される型

次の型はアドレスとして渡されます（構造化型）。

- `BitString`（ビット文字列）
- `OctetString`（8 ビット文字列）
- `ObjectIdentifier`（オブジェクト ID）
- `Struct`（構造体）型
- `Choice`（選択）型
- テンプレート `PowerSet` のインスタンス化
- テンプレート `Bag` のインスタンス化
- テンプレート `Array` のインスタンス化
- テンプレート `String` のインスタンス化。

以下もアドレスとして渡されます。

- 上記のリスト内の型の任意のシンタイプ
- このリストの型を継承する型。

注記

ポインタとして表される型（Charstring、Charstring の任意のシンタイプ、Own、ORef）では、「ポインタのアドレス」ではなく「ポインタそのもの」がパラメータとして渡されます。

パラメータの受け渡し

C で実装される操作のパラメータの受け渡しは次のように動作します。

In パラメータ

- 型が 946 ページの「値として渡される型」のリストに含まれていれば C の値として渡されます。これは、C のパラメータ 型が UML の型と同じであることを意味します。
- 型が 947 ページの「アドレスとして渡される型」のリストに含まれていれば C のアドレスとして渡されます。これは UML の型が（`UML_type`）ならば C のパラメータが
（`UML_type *`）であることを意味します。

In/Out パラメータ

- パラメータは常にアドレスとして渡されます。つまり、UML の型が (`UML_type`) ならば C のパラメータは
(`UML_type *`) です。

操作の結果

操作の結果の型が 946 ページの「値として渡される型」のリストに含まれる場合、C 関数の結果の型は UML の場合と同じです。

結果の型が 947 ページの「アドレスとして渡される型」のリストに含まれる場合、次の 2 点が変更されます。

- C の結果型が (`UML_type *`) になります。つまり結果はアドレスになります。
- C 関数の最後に追加のパラメータが挿入されます。このパラメータも (`UML_type *`) 型で関数の結果を格納する場所として使われます。操作の呼び出しでは、実際のパラメータとして「ダミー」変数を渡す必要があります。すると、C 関数はこれを使用して操作の結果を格納でき、結果として再度変数を返します。

例 **365:** ——————————————————————————————

構造体データ型 struct1 を考えます。

```
class c {
  int X;
  struct1 Y;
}
```

これらの操作の C プロトタイプは次にようになります。

```
SDL_Integer X (SDL_Integer, SDL_Integer *);
struct1 * Y (struct1 *, struct1 *, struct1 *);
```

実装の例は次のようになります。

```
SDL_Integer X
  (SDL_Integer Param1, SDL_Integer *Param2)
{
```

```
  *Param2 = *Param2+Param1;
  return *Param2;
}

struct1 * Y (struct1 *Param1,
             struct1 *Param2,
             struct1 *Result)
{
  /* implementation assuming struct1 to contain
     two integers */
  (*Param2).comp1 = (*Param2).comp1+(*Param1).comp1;
  (*Param2).comp2 = (*Param2).comp2+(*Param1).comp2;
  *Result = *Param2;
  return Result;
    /* always return the last, extra, parameter */
}
```

注記

構造体型では IN パラメータはアドレスとして渡されるので、操作の中でそのようなパラメータを変更すると予期せぬ結果となることがあります。この場合、実際のパラメータとして渡される属性も変わります。仮パラメータを変更する場合には、まずそれをローカル属性にコピーします。

ジェネリック関数

型情報ノード（**Type Info Nodes**）

ジェネリック関数は、複数の異なる型に対して特定の作業を行うことができます。ジェネリック関数を書くには、型の固有情報を利用できるようにしなければなりません。この固有情報として、サイズ、構造体型のコンポーネント型、コンポーネント オフセットなどがあります。この情報は、型情報ノード（Type Info Nodes）にあります。

型情報ノードは、型を定義する情報を含む構造体です。各型には対応する型情報ノードがあります。各型情報ノードには 2 つのセクションが含まれます。

- 最初のセクションには、すべての型情報ノードで共通の一般コンポーネントがあります。
- 2 番目のセクションには、各型を定義する個別の型固有コンポーネントがあります。

型情報ノードの内容の詳細なリファレンスは、第 29 章「C コード ジェネレータ シンボル テーブル」の 1012 ページ、「型情報ノードの型定義」を参照してください。

UML で導入されるすべてのパッシブ クラスまたはシンタイプは、生成された C コードの型情報ノードで記述されます。あらかじめ定義されたデータ型の型情報ノードは、`sctpred.h` および `sctpred.c` にあります。

```
extern tSDLTypeInfo ySDL_SDL_Integer;
extern tSDLTypeInfo ySDL_SDL_Real;
extern tSDLTypeInfo ySDL_SDL_Natural;
extern tSDLTypeInfo ySDL_SDL_Boolean;
extern tSDLTypeInfo ySDL_SDL_Character;
extern tSDLTypeInfo ySDL_SDL_Time;
```

```
extern tSDLTypeInfo ySDL_SDL_Duration;
extern tSDLTypeInfo ySDL_SDL_PId;
extern tSDLTypeInfo ySDL_SDL_Charstring;
extern tSDLTypeInfo ySDL_SDL_Bit;
extern tSDLTypeInfo ySDL_SDL_Bit_String;
extern tSDLTypeInfo ySDL_SDL_Octet;
extern tSDLTypeInfo ySDL_SDL_Octet_String;
extern tSDLTypeInfo ySDL_SDL_IA5String;
extern tSDLTypeInfo ySDL_SDL_NumericString;
extern tSDLTypeInfo ySDL_SDL_PrintableString;
extern tSDLTypeInfo ySDL_SDL_VisibleString;
extern tSDLTypeInfo ySDL_SDL_Null;
extern tSDLGenListInfo ySDL_SDL_Object_Identifier;
```

ユーザー定義型の型情報ノードは次の名前を持ちます。

```
ySDL_#(TypeName)
```

ジェネリック代入関数

UML の各型は、代入マクロ「`yAssF_typename`」にアクセスできます。

例 **366:** 論理型とユーザー定義型 **A**

```
#define yAssF_SDL_Boolean(V,E,A)  (V = E)

#define yAssF_A(V,E,A)  yAss_A(&(V),E,A)
#define yAss_A(Addr,Expr,AssName) ¥
   (void)GenericAssignSort(Addr,Expr,AssName,
                           (tSDLTypeInfo *)&ySDL_A)
```

このマクロは、生成されたコード（およびランタイム ライブラリ）内の代入を行う箇所で使われます。次の 3 つのマクロ パラメータがあります。

- **V**: 左辺の変数
- **E**: 右辺の式
- **A**: 代入のプロパティを与える整数。

このマクロは、C の代入文になるか代入関数の呼び出しになります。代入文は、対応する型に対して C で代入文が記述でき、UML の代入文と比較しても正しい意味を持つ場合に使われます。

代入文が使用できない場合、代入マクロは代入関数の呼び出しになります。元となるジェネリック代入関数は `sctpred.c` および `sctpred.h` に記述されています。

```
extern void * GenericAssignSort(void *, void *,
   int, tSDLTypeInfo *);
```

ここで、

- 最初のパラメータは左辺の変数のアドレス。
- 2 番目のパラメータは右辺の式のアドレス。
- 3 番目のパラメータは代入のプロパティ。
- 4 番目のパラメータは実際の型の型情報ノード。

`GenericAssignSort` は最初のパラメータとして渡されたアドレスを返します。`GenericAssignSort` 関数は次の処理を行います。

- 左辺変数の古い値は解放される。ただし、代入のプロパティでその旨が指定され、値がポインタを含む場合のみ。
- 値は式から変数へコピーされる。通常は `memcpy` 関数が使用されるが、使用できない場合は型の種類に従って特別のコードが実行される。
- 代入のプロパティに従って、変数の `IsAssigned` フラグが設定される。

Charstring および Own テンプレートのインスタンス化は特別な扱いを必要とするため、これらに対して `GenericAssignSort` を呼び出す固有のラッパー関数を導入します。

```
extern void xAss_SDL_Charstring (SDL_Charstring *,
   SDL_Charstring, int);
extern void * GenOwn_Assign (void *, void *, int,
   tSDLTypeInfo *);
```

`GenericAssignSort` 関数を使う場合、オブジェクトを正しく扱うには次の点を考慮する必要があります。

- どのようにオブジェクトをコピーするか。
- 新しく作成したオブジェクトのステータスは何か。
- 左辺の変数が参照している古い値をどうするか。

これらのトピックについて、以下で述べてゆきます。

オブジェクトのコピー

この操作には注意が必要です。誤ったアクションを行うとメモリ リークやアクセス エラーが発生する可能性があるためです。以下の 3 つのパターンがあります。

- **AC**: 常に参照されているオブジェクトをコピーする。
- **AR**: 常にポインタをコピーする。つまり、参照されているオブジェクトを再利用する。
- **MR**: オブジェクトが一時的なものならポインタをコピーし、そうでなければオブジェクトをコピーする。

新しいオブジェクトの状態

これはこのオブジェクトに対する次の操作の準備であり、一番目の考慮点に従って正しく判断するために行います。以下の 2 つのパターンがあります。

- **ASS**: オブジェクトがある変数に代入された場合、つまり、Charstring ソートを表す C 文字列の最初の文字の値「V」と「L」に対応する場合、オブジェクトは「代入された」状態になります。このあとオブジェクト自身をコピーすることになります。典型的なケースは通常の代入文です。
- **TMP**: オブジェクトが永続的な変数に代入されていない場合、つまり、Charstring ソートを表す C 文字列の最初の文字の値「T」に対応する場合、「一時的」な状態となります。変数に代入されないので、このあと、オブジェクト自身をコピーすることはありません。典型的なケースは操作の結果値です。

古い値の処理

通常はこの値に対して free（解放）を行うべきです。そうしないとメモリ リークが起きる可能性があるからです。ただし、変数を初期化するときは、ランダムなアドレスに対して free が呼ばれる可能性があるので free を行うべきではありません。以下の 2 つのパターンがあります。

- **FR**: 古い値を free する。
- **NF**: 古い値を free しない。

`GenericAssignSort` 関数の 3 番目の「代入のプロパティ パラメータ」には、上の考え方に従って値を与えます。可能な限り指定されているマクロを使用します。

```
#define XASS_AC_ASS_FR  (int)25
#define XASS_MR_ASS_FR  (int)26
#define XASS_AR_ASS_FR  (int)28

#define XASS_AC_TMP_FR  (int)17
#define XASS_MR_TMP_FR  (int)18
#define XASS_AR_TMP_FR  (int)20

#define XASS_AC_ASS_NF  (int)9
#define XASS_MR_ASS_NF  (int)10
#define XASS_AR_ASS_NF  (int)12

#define XASS_AC_TMP_NF  (int)1
#define XASS_MR_TMP_NF  (int)2
#define XASS_AR_TMP_NF  (int)4
```

上記のマクロ名はすべて XASS_1_2_3 の形をしています。ここで 1、2、3 に対応する略語は次のような意味です。

- 1 = AC: 常にコピーする。
- 1 = MR: 再利用可能（一時オブジェクトならポインタを取る）。
- 1 = AR: 常に再利用（ポインタを取る）。
- 2 = ASS:「variable」に代入された新しいオブジェクト。
- 2 = TMP: 新しいオブジェクト テンポラリ。
- 3 = FR: 変数が参照している古い値に対して free を呼ぶ。
- 3 = NF: 古い値に対して free を呼ばない。

これらの代入のバリエーションの区別は、対象の型がポインタを使用している場合やコンポーネントとしてポインタを含む型を扱う場合にのみ意味があります。

ジェネリック等値関数

各型は等値（equal）マクロ「`yEqF_typename`」および非等値（not equal）マクロ「`yNEqF_typename`」にアクセスできます。以下に Boolean 型とユーザー定義型 A の例を示します。

```
#define yEqF_SDL_Boolean(E1,E2)   ((E1) == (E2))
#define yNEqF_SDL_Boolean(E1,E2)  ((E1) != (E2))

#define yEqF_z3_A(Expr1,Expr2)  yEq_z3_A(Expr1,Expr2)
#define yNEqF_z3_A(Expr1,Expr2)  ( ! yEq_z3_A(Expr1,Expr2) )
```

```
#define yEq_z3_A(Expr1,Expr2) ¥
  GenericEqualSort((void *)Expr1,(void *)Expr2, ¥
                   (tSDLTypeInfo *)&ySDL_z3_A)
```

これらのマクロは、生成されたコード（およびカーネル内）で等値判定が必要な場所ごとに使われます。等値および非等値マクロのパラメータがテストすべき 2 つの式です。

C の等値または非等値演算子を使うことができない場合、マクロは関数呼び出しになります。基本的なジェネリック等値関数は `sctpred.c` および `sctpred.h` に記述されています。

```
extern SDL_Boolean GenericEqualSort(void *, void *,
   tSDLTypeInfo *);
```

ここで、

- 最初の 2 つのパラメータは比較対象の 2 つの式のアドレス。
- 3 番目のパラメータは実際の型の型情報ノード。

Charstring と Own

Charstring および Own テンプレートのインスタンスは特別な扱いを必要とするため、これらの型に対して `GenericEqualSort` を呼び出す固有のラッパー関数を導入します。

```
extern SDL_Boolean xEq_SDL_Charstring
   (SDL_Charstring, SDL_Charstring);
extern SDL_Boolean GenOwn_Equal (void *, void *,
    tSDLTypeInfo *);
```

ジェネリック開放関数

ポインタとして実装される各型、または自動的に処理されるメモリを参照するポインタを含む各型には、「`yFree_typename`」関数またはマクロが定義されています。ジェネリック関数モデルでは、これは常にマクロになります。

```
#define yFree_SDL_Charstring(P)  xFree_SDL_Charstring(P)
#define xFree_SDL_Charstring(P) ¥
   GenericFreeSort(P,(tSDLTypeInfo *)&ySDL_SDL_Charstring)

#define yFree_A(P) ¥
   GenericFreeSort(P,(tSDLTypeInfo *)&ySDL_A)
```

「`yFree`」マクロは常に関数 `GenericFreeSort` の呼び出しに変換されます。

```
extern void GenericFreeSort (void **, tSDLTypeInfo *);
```

この関数は、変数のアドレスと型情報ノードを使用して、変数内で動的に確保して使用していたメモリを解放します。

ジェネリック構築関数

構造化された型の値を生成する 4 つのジェネリック関数があります。

```
extern void * GenericMakeStruct (void *, tSDLTypeInfo *, ...);
extern void * GenericMakeChoice (void *, tSDLTypeInfo *,
```

```
   int, void *);
extern void * GenericMakeOwnRef (tSDLTypeInfo *, void *);
extern void * GenericMakeArray  (void *, tSDLTypeInfo *,
   void *);
```

GenericMakeStruct

この作成操作は、struct 型、ObjectIdentifier 型、および String、PowerSet、Bag テンプレートのインスタンス化で使用できます。

- `void *` パラメータは、結果を入れる変数のアドレスです。この値が戻り値です。
- `tSDLTypeInfo *` パラメータは、作成する型の型情報ノードのアドレスです。
- “...” は、構造体内のコンポーネントの値のアドレスのリストを意味します。すべてのパラメータは、コンポーネント 型がアドレスとして渡せるか値として渡せるかに関わらず、アドレス（`void *`）として渡します。

  唯一の例外は、それ自体がポインタとして表現される型（Charstring、Own、ORef、およびこれらの型の任意のシンタイプ）であり、この場合はポインタのアドレスではなくポインタ自体を渡します。

  選択可能 フィールドまたは初期値のあるフィールドの場合は、コンポーネントの値が存在するか否かを示すため、「0」または「1」を渡します。「1」を指定する場合は、次のパラメータで実際の値を渡します。「0」を指定する場合は値を指定するパラメータは必要ありません。

GenericMakeChoice

この関数は choice 型に使われます。

- 最初の `void *` パラメータは、結果を入れる変数のアドレスです。この値が戻り値です。
- `tSDLTypeInfo *` パラメータは、作成する型の型情報ノードのアドレスです。
- `int` パラメータは、どの choice コンポーネントが存在するかを指定します。
- 最後の `void *` パラメータは、値を格納した領域のアドレスです。

GenericMakeOwnRef

この関数はテンプレート Own のインスタンス化に使用します。

- `tSDLTypeInfo *` パラメータは、作成する型の型情報ノードのアドレスです。
- `void *` パラメータは、この関数が割り当てるメモリに代入される値のアドレスです。

GenericMakeArray

この関数は、テンプレート `Array`、`CArray`、および `GArray` のインスタンス化に使用します。

- 最初の `void *` パラメータは、結果を入れる変数のアドレスです。この値が戻り値です。

- `tSDLTypeInfo *` パラメータは、作成する型の型情報ノードのアドレスです。
- 最後の `void *` パラメータは、配列のすべてのコンポーネントに代入される値のアドレスです。

Copy

すべてのユーザー定義型に対して暗黙の copy 演算子が挿入されます。この演算子は値を入力とし、その値のコピーを返します。Own ポインタではない型、Own ポインタを含まない型にとっては、この演算子は意味がありません。単に同じ値を返します。しかし、Own ポインタまたは Own ポインタを含む構造体値では、copy 関数は Own ポインタが参照する値をコピーします。

暗黙の copy 演算子は、ユーザー定義型用にのみ実装されています。定義済みデータ型では、copy 演算子は意味がないので実装されていません。

定義済みテンプレートの操作用のジェネリック関数

定義済みテンプレートの操作のジェネリック関数は、以下の例外を除いて一般的な操作の規則に従います。

- C の関数がある特定のテンプレートのすべてのインスタンス化を扱えるように、パラメータとして型情報ノードが必要です。
- テンプレート パラメータ型（たとえばコンポーネントやインデックス 型）のパラメータは、多くの場合アドレスとして渡す必要があります。その型のプロパティが不明なためです。

一般配列

```
extern void * GenGArray_Extract (xGArray_Type *, void *,
   tSDLGArrayInfo *);
extern void * GenGArray_Modify  (xGArray_Type *, void *,
   tSDLGArrayInfo *);
```

- パラメータ 1 : 配列
- パラメータ 2 : アドレスとして渡されるインデックス値
- パラメータ 3 : 型情報ノード
- 結果 : コンポーネントのアドレス。

PowerSet

これは、単純なコンポーネント 型の PowerSet で使用できるジェネリック関数です。PowerSet はビットシーケンス「`unsigned char[Appropriate_Length]`」で表現されます。

```
#define GenPow_Empty(SDLInfo,Result) ¥
   memset((void *)Result,0,(SDLInfo)->SortSize)
extern SDL_Boolean GenPow_In (int, xPowerset_Type *,
   tSDLPowersetInfo *);
extern void * GenPow_Incl (int, xPowerset_Type *,
```

```
   tSDLPowersetInfo *, xPowerset_Type *);
extern void * GenPow_Del (int, xPowerset_Type *,
   tSDLPowersetInfo *, xPowerset_Type *);
extern void GenPow_Incl2 (int, xPowerset_Type *,
   tSDLPowersetInfo *);
extern void GenPow_Del2 (int, xPowerset_Type *,
   tSDLPowersetInfo *);
extern SDL_Boolean GenPow_LT (xPowerset_Type *,
   xPowerset_Type *, tSDLPowersetInfo *);
extern SDL_Boolean GenPow_LE (xPowerset_Type *,
   xPowerset_Type *, tSDLPowersetInfo *);
extern void * GenPow_And (xPowerset_Type *, xPowerset_Type *,
   tSDLPowersetInfo *, xPowerset_Type *);
extern void * GenPow_Or (xPowerset_Type *, xPowerset_Type *,
   tSDLPowersetInfo *, xPowerset_Type *);
extern SDL_Integer GenPow_Length (xPowerset_Type *,
   tSDLPowersetInfo *);
extern int GenPow_Take (xPowerset_Type *, tSDLPowersetInfo *);
extern int GenPow_Take2 (xPowerset_Type *, SDL_Integer,
   tSDLPowersetInfo *);
```

- `GenPow_In`、`GenPow_Incl`、`GenPow_Del`、`GenPow_Incl2`、`GenPow_Del2` の `int` 型のパラメータ : コンポーネント値。
- `GenPow_Take`、`GenPow_Take2` の `int` 型の戻り値 : コンポーネント値。
- `tSDLPowersetInfo *` 型のパラメータ : 型情報ノード。
- 型情報ノードの後に現れる `xPowerset_Type *` 型のパラメータ : 結果を格納するアドレス。戻り値。
- 他の `xPowerset_Type *` パラメータ : パラメータとして渡される `PowerSet`。

Bag および General PowerSet

Bag と合成型コンポーネントをもつ PowerSet には、次のジェネリック関数を利用できます。これらの型は C ではリンクリストとして表現されます。

```
#define GenBag_Empty(SDLInfo,Result) ¥
   memset((void *)Result,0,(SDLInfo)->SortSize)
extern void * GenBag_Makebag (void *, tSDLGenListInfo *,
   xBag_Type *);
extern SDL_Boolean GenBag_In (void *, xBag_Type *,
   tSDLGenListInfo *);
extern void * GenBag_Incl (void *, xBag_Type *,
   tSDLGenListInfo *, xBag_Type *);
extern void * GenBag_Del (void *, xBag_Type *,
   tSDLGenListInfo *, xBag_Type *);
extern void GenBag_Incl2 (void *, xBag_Type *,
   tSDLGenListInfo *);
extern void GenBag_Del2 (void *, xBag_Type *,
   tSDLGenListInfo *);
extern SDL_Boolean GenBag_LT (xBag_Type *, xBag_Type *,
   tSDLGenListInfo *);
extern SDL_Boolean GenBag_LE (xBag_Type *, xBag_Type *,
   tSDLGenListInfo *);
extern void * GenBag_And (xBag_Type *, xBag_Type *,
   tSDLGenListInfo *, xBag_Type *);
extern void * GenBag_Or(xBag_Type *, xBag_Type *,
   tSDLGenListInfo *, xBag_Type *);
extern SDL_Integer GenBag_Length (xBag_Type *,
   tSDLGenListInfo *);
extern void * GenBag_Take (xBag_Type *, tSDLGenListInfo *,
   void *);
```

```
extern void * GenBag_Take2 (xBag_Type *, SDL_Integer,
   tSDLGenListInfo *, void *);
```

- GenBag_Makebag、GenBag_In、GenBag_Incl、GenBag_Del、GenBag_Incl2、GenBag_Del2 の int 型の パラメータ：コンポーネント値のアドレス。
- GenBag_Take、GenBag_Take2 の int 型の 戻り値：コンポーネント値のアドレス。
- tSDLGenListInfo * 型のパラメータ：型情報ノード。
- 型情報ノードの後に現れる xBag_Type * 型のパラメータ：結果を格納するアドレス。戻り値。
- 型情報ノードの後に現れる xBag_Type * 型のパラメータ：結果を格納するアドレス。戻り値。
- その他の xBag_Type * パラメータ：パラメータとして渡される Bag/PowerSet。

String

String のインスタンス化には次のジェネリック関数を利用できます。String は、リンクリストとして実装されています。

```
#define GenString_Emptystring(SDLInfo,Result) ¥
   memset((void *)Result,0,(SDLInfo)->SortSize)
extern void * GenString_MkString (void *, tSDLGenListInfo *,
   xString_Type *);
extern SDL_Integer GenString_Length (xString_Type *,
   tSDLGenListInfo *);
extern void * GenString_First (xString_Type *,
   tSDLGenListInfo *, void *);
extern void * GenString_Last (xString_Type *,
   tSDLGenListInfo *, void *);
extern void * GenString_Concat (xString_Type *,
   xString_Type *, tSDLGenListInfo *, xString_Type *);
extern void * GenString_SubString (xString_Type *,
   SDL_Integer, SDL_Integer, tSDLGenListInfo *,
   xString_Type *);
extern void GenString_Append (xString_Type *, void *,
   tSDLGenListInfo *);
extern void * GenString_Extract (xString_Type *, SDL_Integer,
   tSDLGenListInfo *);
```

- GenString_MkString、GenString_Append の void * 型のパラメータ：コンポーネント値のアドレス。
- 型情報ノードの後の void * 型または xString_Type * パラメータ : 結果を格納するアドレス。戻り値。
- tSDLGenListInfo * 型のパラメータ：型情報ノード。
- その他のパラメータ：パラメータの定義に従います。

限定文字列（**Limited string**）

限定文字列のジェネリック関数が利用できます。限定文字列は、文字列の最大サイズを使用する文字列です。これらの文字列は、C では配列として実装されます。

```
#define GenLString_Emptystring(SDLInfo,Result) ¥
   memset((void *)Result,0,(SDLInfo)->SortSize)
extern void * GenLString_MkString (void *, tSDLLStringInfo *,
   xLString_Type *);
#define GenLString_Length(ST,SDLInfo) (ST)->Length
extern void * GenLString_First (xLString_Type *,
   tSDLLStringInfo *, void *);
extern void * GenLString_Last (xLString_Type *,
   tSDLLStringInfo *, void *);
extern void * GenLString_Concat (xLString_Type *,
   xLString_Type *, tSDLLStringInfo *, xLString_Type *);
extern void * GenLString_SubString (xLString_Type *,
   SDL_Integer, SDL_Integer, tSDLLStringInfo *,
   xLString_Type *);
extern void GenLString_Append (xLString_Type *, void *,
   tSDLLStringInfo *);
extern void * GenLString_Extract (xLString_Type *,
   SDL_Integer, tSDLLStringInfo *);
```

- `GenLString_MkString`、`GenString_Append` の `void *` 型のパラメータ : コンポーネント値のアドレス。
- 型情報ノードの後に現れる `void *` 型または `xLString_Type *` 型のパラメータ : 結果を格納するアドレス。戻り値。
- `tSDLLStringInfo *` 型のパラメータ : 型情報ノード。
- その他のパラメータ : パラメータの定義に従う。

最適化

未使用操作の削除

システムを実装するとき、常に使用可能なすべての UML 操作を使うわけではありません。C コード ジェネレータは、未使用の操作の宣言を削除して生成されたアプリケーションのコード サイズを最小化します。以下の未使用操作が削除されます。

- 定義済みデータ型の操作、たとえば、substring、concatenate、Charstring の長さの計算など。
- 定義済みテンプレートの String、Array、PowerSet、Bag で定義されている操作
- assign、equal、default、make、extract、modify、free などの特殊操作とヘルプ関数

C コード ジェネレータは、以下の手順でコードを最適化します。

1. 1 つの操作を実装するすべての C 関数を `#ifndef` 定義で囲みます。

例 **367** `#ifndef` 定義

```
#ifndef XNOUSE_AND_BIT_STRING
  /* function implementing the operation */
#endif
```

2. コードを生成する際、アクティブ クラスを実装する状態機械の遷移における操作の使用を記録します。

3. 異なる操作間の依存関係を更新します。たとえば、構造体型の等式操作は、そのすべてのコンポーネント型の等式に依存することがあります。
4. 未使用と判断された各操作に対して、その操作のコードを削除する `#define` 定義が生成されます。すべての定義は、`auto_cfg.h` というファイルに保存されます。

例 **368 #define** コマンド

```
#define XNOUSE_AND_BIT_STRING
```

生成された **C** コードの名前

このセクションは C コード ジェネレータと AgileC コード ジェネレータの両方に適用されます。

生成された **C** の名前の接頭辞と接尾辞

UML の名前が C の識別子に変換されると、通常、UML で与えられる名前に接頭辞または接尾辞が追加されます。この接尾辞は、生成されたコードで名前が重複するのを防いでいます。これは、UML が C 以外にもスコープ ルールを持ち、オブジェクトのエンティティ クラスが異なれば同じスコープ内で定義される異なるオブジェクトが同じ名前を持つことが許されるために必要です。たとえば、UML では、アクティブ クラスで同じ名前のソート、属性、および操作を定義できます。したがって、接頭辞または接尾辞は変換された UML 名を C プログラム内で一意なものにすることを目的としています。

コードを読みやすくするために、接尾辞を使うことを推奨します。接頭辞を使う理由としては、使用するコンパイラが識別子の名前を解析するときに、限られた数の文字しか見ない場合などが考えられます。この場合には、名前の先頭に一意な部分を設ける必要があります。

生成されたコード内の接尾辞を使う名前

UML オブジェクトの生成された名前は次の順序の 3 つの部分からなります。

1. C 識別子で許されない文字から UML 名を取り除いたもの
2. 下線 '_'
3. 名前を一意にする一連の文字。オブジェクトがパッケージの一部である場合、この中にはパッケージ名が含まれます。

生成されたコード内の接頭辞を使う名前

UML オブジェクトの生成された名前は次の順序の 4 つの部分からなります。

1. 文字 'z'
2. 名前を一意にする一連の文字。オブジェクトがパッケージの一部である場合、この中にはパッケージ名が含まれます。
3. 下線 '_'
4. C 識別子で許されない文字から UML 名を取り除いたもの

一連の文字

名前を一意にする一連の文字は、システムの宣言の構造体内の宣言の位置によって決まります。

- 特定のレベルの各宣言には番号 0、1、2、...、9、a、b、...、z が振られています。
- あるレベルの宣言の番号が 36 を超えるとシーケンスは 00、10、20,...、90、a0,...、z0、01、11、21,...、91、a1,...、z1,.....、0z、1z、2z,...、9z、az,...、zz となります。
- 宣言の数が 36 * 36 を超えると 3 文字のシーケンスが使われ、以下同様になります。

これで、名前を一意にする総シーケンスは、ユニットとその親の " 宣言番号 " で構成されています。つまり、"root" から初めてそれが定義されているユニットです。

例 **369** ———————————————————————————————

たとえば、あるソートが、インライン アクティブ クラスの 5 番目の宣言として定義されていて、システム内の 12 番目の宣言である部分を持つとします。すると、総シーケンスは b4 となります（2 つレベルのどこにも 36 以上の宣言がないとすると）。

———————————————————————————————

例 **370:** コード内の生成された名前 ———————————————————————

生成された名前の例 :

名前	宣言の位置	生成された名前
S1	システム内の 10 番目の宣言	S1_9 または z9_S1
Var2	アクティブ クラスの 3 番目の宣言、これはシステム内の 15 番目の宣言である部分を持つインライン アクティブ クラスの 5 番目の宣言	Var2_e42 または ze42_Var2

この他にも接頭辞を使用する生成された名前があります。たとえば、ソート MySort が za2c_MySort に変換されると、この型に関連する等値関数は yEq_za2c_MySort と呼ばれます（ある場合）。

28

C コード ジェネレータ ランタイム モデル

ランタイム モデルは C コード ジェネレータによって生成されるアプリケーションを管理し、そのアプリケーション内のアクティブ クラスのインスタンスのスケジューリング、シグナリングおよび実行を制御します。

この章では、アプリケーション コードのランタイム実行を制御する原理を理解するために役に立つ情報を提供します。C レベルでアプリケーションのデバッグを進める必要がある場合、この情報が役立ちます。

また本章では、システムに存在する各種オブジェクトを表すために使用される C のデータ構造体についても説明します。

シグナルとタイマー

シグナルとタイマーを表すデータ構造

シグナルは構造体型で表現されます。scttypes.h で定義される xSignalRec 構造体には、シグナル パラメータに由来するもの以外のシグナルについての全般情報が格納されています。scttypes.h には、シグナルに関する以下の情報が含まれます。

```
#ifdef XMSCE
#define GLOBALINSTID  int GlobalInstanceId;
#else
#define GLOBALINSTID
#endif

#if defined(XSIGPATH) && defined(XMSCE)
#define ENVCHANNEL  xChannelIdNode EnvChannel;
  /* Used if env split into connectord in Sequence
     Diagram trace */
#else
#define ENVCHANNEL
#endif

#ifdef XENV_CONFORM_2_3
#define XSIGNAL_VARP  void * VarP;
#else
#define XSIGNAL_VARP
#endif

define SIGNAL_VARS ¥
   xSignalNode   Pre; ¥
   xSignalNode   Suc; ¥
   int           Prio; ¥
   SDL_PId       Receiver; ¥
   SDL_PId       Sender; ¥
   xSignalIdNode NameNode; ¥
   GLOBALINSTID ¥
   ENVCHANNEL ¥
   XSIGNAL_VARP

typedef struct xSignalStruct  *xSignalNode;
typedef struct xSignalStruct {
   SIGNAL_VARS
}  xSignalRec;
```

したがって、xSignalNode 型は、xSignalRec 型の割り当てデータ領域を参照するために使われるポインタ型です。xSignalRec 構造体の各コンポーネントの機能は以下のとおりです。

- Pre および Suc。これらのポインタは、シグナルを受信インスタンスの入力ポートにリンクするために使用されます。

  入力ポートは二重にリンクされた リストとして実装されます。Suc は、シグナルを現在のシグナル型の使用可能メモリ リストにリンクするために使用されます。このリストは、このシグナル型を表す xSignalIdNode にあります。シグナルが使用可能メモリ リストにある場合、Pre は 0 となります。
- Prio。シグナルの優先度です。

- Receiver はシグナルの受信者を参照するために使用されます。受信者は signal sending 文で設定するか、あるいは算出します（直接アドレッシングなしの signal sending）。
- Sender はアクティブ クラスの送信インスタンスの Pid 値です。
- NameNode はシグナル型を表す xSignalIdNode への参照であり、このシグナル インスタンスのシグナル型を定義します。
- VarP は、C コード ジェネレータの旧バージョンとのシグナルの互換性を確保するために、マクロ XSIGNAL_VARP によって取り込まれるポインタです。通常の場合、このコンポーネントは使われないので現れません。
- EnvChannel はシーケンス図トレースの発信コネクタを特定するために使用されます。
- GlobalInstanceId は、シーケンス図トレースでシグナル インスタンスの一意の識別子として使用されます。

パラメータなしのシグナルは xSignalStruct によって表されますが、パラメータ付きのシグナルを表すために、ySignalPar_z_<package>_<number>_SignalName という名前の構造体型とこの構造体型を参照するポインタ型（ yPDP_ という接頭辞付）の 2 つが、生成されたコード内で定義されます。この構造体型は SIGNAL_VARS マクロで始まり、次に、シグナル パラメータの定義と同じ順序で、各シグナル パラメータごとに 1 つのコンポーネントが入ります。これらのコンポーネントには Param1、Param2 という名前が付けられます。

例 **371:** ——————————————————————————————

```
typedef struct {
    SIGNAL_VARS
    SDL_Integer  Param1;
    SDL_Boolean  Param2;
} ySignalPar_z_<package>_<number>_sig;
typedef ySignalPar_z_<package>_<number>_sig
*yPDP_z_<package>_<number>_sig;
```

これらの型はシグナル sig(Integer, Boolean) を表しています。

——————————————————————————————————————

すべてのシグナルは SIGNAL_VARS で定義されたコンポーネントから始まるので、SIGNAL_VARS 内のコンポーネントにのみアクセスする場合、ポインタを型キャストして xSignalNode 型に変換できます。

シグナルのデータ領域の割り当て

sctsdl.c には、xGetSignal と xReleaseSignal の 2 つの関数があり、これらの関数でシグナルのデータ領域が処理されます。

```
xSignalNode xGetSignal(
  xSignalIdNode  SType,
  SDL_PId        Receiver,
  SDL_PId        Sender )

void xReleaseSignal( xSignalNode  *S )
```

xGetSignal はシグナル型を識別する xSignalIdNode への参照と、2 個の Pid 値（アクティブ クラスの送信側と受信側のインスタンス）をパラメータとし、シグナル インスタンスを戻り値とします。xGetSignal はまずシグナル 型の使用可能メモリ リスト（シグナル型の SignalIdNode 内のコンポーネント AvailSignalList）を調べ、その中に利用可能なシグナルがあれば再利用します。使用可能メモリ リストが空の場合のみ、新しいメモリが割り当てられます。シグナルの型が xGetSignal 関数に不明な場合でも、このシグナル型の xSignalIdNode にあるコンポーネント VarSize が ySignalPar_SignalName を正しく割り当てるために必要な情報を提供します。

関数 xReleaseSignal は xSignalNode ポインタのアドレスを取得して、参照したシグナルをシグナル型の使用可能メモリ リストに返します。次に、xSignalNode ポインタが 0 に設定されます。

関数 xGetSignal は以下の場所で使用されます。

- 生成されたコード内（Output、TimerSet、TimerReset）
- 次のライブラリ内の複数の場所：
  SDL_Create
  SDL_SimpleReset
  SDL_Nextstate（トリガされた遷移のガードの処理のため）

関数 xReleaseSignal は以下のライブラリによって使用されます。

- SDL_Nextstate
- SDL_Stop。どちらの場合も、遷移を起動したシグナルを解放するため。

シグナル パラメータの詳細なレイアウト

シグナル パラメータについてのレイアウト情報は、「Set-SDL-Coder」が C コード ジェネレータ の Advanced options に指定されていれば、要求に基づいて取得できます。この情報は、C コード ジェネレータ : <basename>_cod.h と <basename>_cod.c によって生成された 2 つのファイルに格納されています。これらのファイルはコンパイルとリンクに必要です。

<basename>_cod.c ファイルには、シグナルの構造を記述した tSDLSignalInfoS 型と tSDLSignalParaInfoS 型の構造体の定義が含まれています。これらの型は、ランタイム ライブラリ ファイル sctpred.h で定義されます。

シグナルの受信と送信

このセクションでは、シグナルを取り扱う操作の概要を説明します。詳細については、アクティブ クラスのインスタンスを扱うセクションで説明します（978 ページの「シグナルの送受信」）。

シグナル インスタンスの送信には関数 SDL_Output が使用されます。この関数はシグナル インスタンスを取得し、それを受信側インスタンスの入力ポートに挿入します。

受信者がレディ キュー（遷移を実行可能だが、その実行がまだスケジュールされていないアクティブクラスのインスタンスが入っている「待ち行列」）内になく、現在のシグナルが即時遷移を引き起こせる場合、受信者のインスタンスはレディ キューに挿入されます。

受信者がレディ キューにすでにあるか、あるいは現在のシグナルを保存すべき状態にある場合、このシグナルは入力ポートに挿入されます。

シグナル インスタンスが遷移を起こせず、また保存する必要もない場合、シグナル インスタンスは即座に破棄されます（シグナル インスタンスのデータ領域は使用可能メモリ リストに返されます）。

入力ポートは nextstate 操作時にスキャンされ、遷移を起こせる入力ポート内の次のシグナルを探します。処理の後シグナル インスタンスは保存または破棄されます。

PAD 関数

特定の入力関数は存在せず、代わりに PAD 関数がランタイム ライブラリと生成されたコードの両方に提供されます。実行すべき次の遷移を引き起こすシグナル インスタンスは、メイン ループ（スケジューラ）の入力ポートから取り除かれ、直後に状態機械の PAD 関数が呼び出されます。**PAD 関数**は、C の関数でありアクティブ クラスの状態機械の動作を実装します。この関数は C コード ジェネレータによって生成されます。PAD 関数が最初に実行するのは、シグナル パラメータのローカル変数への割り当てです。

遷移を引き起こしたシグナル インスタンスは解放されて、現在の遷移を終了させる nextstate アクションまたは stop アクションの使用可能メモリ リストへ返されます。

シグナル番号ファイル

多くのリアルタイム オペレーティング システムでは、シグナルおよびメッセージを整数値で表すことが要求されます。各シグナルには整数値が割り当てられます。シグナル番号ファイルの名前は `<basename>.hs` です。

シグナル番号の使い方については、899 ページの「シグナルが多数ある場合の xOutEnv のパフォーマンス向上」も参照してください。

タイマーの操作

タイマーの表現

パラメータ付きタイマーは、シグナル定義の場合とまったく同様に、型定義によって表現されます（964 ページの「シグナルとタイマーを表すデータ構造」）。実行中は、設定されているタイマーの中でまだ切れていないすべてのタイマーは次の `xTimerRec` 構造体とシグナル インスタンスによって表現されます。

```
#define TIMER_VARS ¥
   xSignalNode   Pre; ¥
   xSignalNode   Suc; ¥
   int           Prio; ¥
```

```
    SDL_PId       Receiver; ¥
    SDL_PId       Sender; ¥
    xSignalIdNode NameNode; ¥
    GLOBALINSTID ¥
    ENVCHANNEL ¥
    SDL_Time      TimerTime;

typedef xTimerRec  *xTimerNode;

typedef struct xTimerStruct {
   TIMER_VARS
}  xTimerRec;
```

TIMER_VARS は、最後の TimerTime コンポーネントを除いて、SIGNAL_VARS マクロと一致しなければなりません。パラメータ付きタイマーは、シグナルの場合と同様に生成されたコードに、ySignalPar_timername 型と yPDP_timername 型を持っています（966 ページの「シグナルの受信と送信」）。ただし、SIGNAL_VARS は TIMER_VARS に置き換わります。

タイマー待ち行列

タイマーは、その生存期間中 2 つの異なる表示形態のいずれかをとります。1 つは、タイマー時間 が時間切れになるのを待っているタイマーです。この期間は、タイマーは xTimerQueue に挿入されます。タイマー時間が切れると、タイマーはシグナルとなり、ほかのシグナルと同様に受信側の入力ポートに挿入されます。xSignalRec と xTimerRec の typedef は同一なので、xTimerNode 型と xSignalNode 型との間の型 キャストは可能です。

タイマーをシグナルとして扱う際には、xTimerRec のコンポーネントは xSignalRec の場合と同じ方法で使用されます。タイマーがタイマー待ち行列にある間、コンポーネントは以下の方法で使用されます。

- Pre と Suc は、xTimerRec をアクティブ タイマーのタイマー待ち行列にリンクするために使われるポインタです。
- TimerTime は Set 操作で与えられる時間です。

タイマー待ち行列は変数 xSysD 内のコンポーネント xTimerQueue によって表現されます。

```
xTimerNode xTimerQueue;
```

この変数は sctsdl.c の関数 xInitKernel 内で初期化されます。xTimerQueue は初期化されると、タイマー待ち行列の先頭を参照します。

待ち行列の先頭はタイマーを表さない特別な要素です。待ち行列処理のアルゴリズムを簡単にするために導入されています。待ち行列の先頭にある TimerTime コンポーネントには非常に大きな時間値（xSysD.xMaxTime）が設定されます。

このようにタイマー待ち行列は、リストに先頭部分を持つ二重のリンクリストであり、タイマー時間に基づいてソートされているので、最短時間のタイマーが最初の位置に置かれています。

xTimerRec 構造体は、シグナルと同様に割り当てと再利用が行われます。

タイマーの取り扱い

タイマーは以下の場所で取り扱われます。

- タイマー定義
- タイマー出力
- TimerSet と TimerReset の操作

タイマー出力は、タイマーがタイムアウトしてタイマーシグナルが送出されたときに発生するイベントです。その後、タイマー シグナルは通常のシグナルとして扱われます。この操作は以下のように実装されます。

```
void SDL_Set(
  SDL_Time     T,
  xSignalNode  S )
```

この関数は `Set` 操作を表し、タイマーの `time` とシグナル インスタンスをパラメータとします。この関数はまずシグナル インスタンスを使用して暗黙のリセットを実行します。次に `S` の `TimerTime` コンポーネントを更新し、`S` をタイマー待ち行列の正しい位置に挿入します。

`SDL_Set` 操作は、生成されたコード内で、`SDL_Output` とほぼ同様に `xGetSignal` とともに使用されます。まずシグナル インスタンスが `xGetSignal` によって作成され、次にタイマー パラメータへの値の割り当てが行われ、最後に `SDL_Set` によって `Set` 操作が実行されます。

TimerReset には 2 つの関数が用意されています。`SDL_SimpleReset` はパラメータを持たないタイマー用、`SDL_Reset` はパラメータを持つタイマー用です。

```
void SDL_Reset( xSignalNode  *TimerS )

void SDL_SimpleReset(
  xPrsNode       P,
  xSignalIdNode  TimerId )
```

`SDL_Reset` は 2 つの関数 `xRemoveTimer` と `xRemoveTimerSignal` を使用して、タイマー待ち行列にあるタイマーとアクティブ クラスのインスタンスの入力ポートにあるシグナル インスタンスを削除します。次に、パラメータとして与えられているシグナル インスタンスを解放します。このシグナルは、TimerReset アクションで与えられるパラメータ値を運ぶためだけに使用されます。

関数 `SDL_SimpleReset` は `SDL_Reset` と同じ方法で実装されます。ただし、この関数は独自のシグナル インスタンス（パラメータなし）を生成します。

TimerReset アクションでは、タイマーがタイマー待ち行列から削除されて使用可能メモリ リストへ戻されます。入力ポート内のシグナル インスタンスは、入力ポートから削除され、現在のシグナル型の使用可能メモリ リストへ戻されます。

```
static void SDL_OutputTimerSignal( xTimerNode  T )
```

タイマー待ち行列にある最初のタイマーがタイムアウトしたとき、`SDL_OutputTimerSignal` がメイン ループから呼び出されます。次に対応するシグナル インスタンスが送出されます。

SDL_OutputTimerSignal は xTimerRec へのポインタをパラメータとして使用し、タイマー待ち行列からタイマーを削除し、SDL_Output を使用して通常のシグナル送信として送出します。

タイマーがアクティブかどうかを、関数 SDL_Active を呼び出すことによって調べることができます。この関数は生成されたコード内で使用され、active 操作を表します。

```
SDL_Boolean SDL_Active (
  xSignalIdNode TimerId,
  xPrsNode      P )
```

注記

テストできるのはパラメータを持たないタイマーだけです。これは C コード ジェネレータの制約です。

タイマーを取り扱う場所がもう 1 つあります。アクティブ クラスの インスタンスが stop アクションを実行すると、このインスタンスに接続されているタイマー待ち行列内のすべてのタイマーが削除されます。これは、関数 xRemoveTimer の最初のパラメータを 0 に設定して呼び出すことで実現しています。

アクティブ クラス

アクティブ クラスを表すデータ構造

アクティブ クラスのインスタンスは、2 つの構造体、すなわち、xLocalPIdRec と、アクティブ クラスの一般データとインスタンスのローカル変数および仮パラメータの両方を含む構造体（yVDef_ProcessName）によって表現されます。xLocalPIdRec と yVDef_ProcessName の両方を持つ理由は 974 ページの「create 操作と stop 操作」で説明します。

上の内容に対応する型定義は scttypes.h に以下のように定義されています。

```
#ifdef XPRSSENDER
#define XPRSSENDERCOMP   SDL_PId  Sender;
#else
#define XPRSSENDERCOMP
#endif

#ifdef XTRACE
#define XTRACEDEFAULTCOMP   int Trace_Default;
#else
#define XTRACEDEFAULTCOMP
#endif

#ifdef XGRTRACE
#define XGRTRACECOMP   int GRTrace;
#else
#define XGRTRACECOMP
#endif

#ifdef XMSCE
#define XMSCETRACECOMP   int  MSCETrace;
#else
```

```
#define XMSCETRACECOMP
#endif

#if defined(XMONITOR) || defined(XTRACE)
#define XINTRANSCOMP   xbool InTransition;
#else
#define XINTRANSCOMP
#endif

#ifdef XMONITOR
#define XCALL_ADDR   int  CallAddress;
#else
#define XCALL_ADDR
#endif

#define PROCESS_VARS ¥
   xPrsNode       Pre; ¥
   xPrsNode       Suc; ¥
   int            RestartAddress; ¥
   xPrdNode       ActivePrd; ¥
   void (*RestartPAD) (xPrsNode  VarP); ¥
   XCALL_ADDR ¥
   xPrsNode       NextPrs; ¥
   SDL_PId        Self; ¥
   xPrsIdNode     InstNameNode; ¥
   xPrsIdNode     TypeNameNode; ¥
   int            State; ¥
   xSignalNode    Signal; ¥
   xInputPortRec  InputPort; ¥
   SDL_PId        Parent; ¥
   SDL_PId        Offspring; ¥
   int            BlockInstNumber; ¥
   XSIGTYPE       pREPLY_Waited_For; ¥
   xSignalNode    pREPLY_Signal; ¥
   XPRSSENDERCOMP ¥
   XTRACEDEFAULTCOMP ¥
   XGRTRACECOMP ¥
   XMSCETRACECOMP ¥
   XINTRANSCOMP

typedef struct {
   xPrsNode      PrsP;
   int           InstNr;
   int           GlobalInstanceId;
}  xLocalPIdRec;

typedef xLocalPIdRec  *xLocalPIdNode;

typedef struct {
   int            GlobalNodeNr;
   xLocalPIdNode  LocalPId;
}  SDL_PId;

typedef struct xPrsStruct  *xPrsNode;

typedef struct xPrsStruct {
   PROCESS_VARS
} xPrsRec;
```

したがって、Pid 値は次の 2 つのコンポーネントを持つ構造体です。

- グローバル ノード番号
- xLocalPIdRec 構造体へのポインタ

xLocalPIdRec には次の 3 つのコンポーネントが含まれています。

- xPrsNode 型の PrsP ― アクティブ クラスのインスタンスの表現の一部である xPrsRec 構造体へのポインタ。
- int 型の InstNr ― アクティブ クラスの現在のインスタンスのインスタンス番号。モデル ベリファイヤ（Model Verifier）内のユーザーとの通信と動的エラー メッセージで使用される。
- GlobalInstanceId ― シーケンス図トレースで使用される、アクティブ クラスのインスタンスの一意の識別番号。

xPrsRec 構造体には以下のコンポーネントが含まれています。各 yVDef_ProcessName 構造体には PROCESS_VARS マクロが最初の項目として含まれているので、xPrsRec へのポインタと yVDef_ProcessName 構造体へのポインタとの間でポインタ値をキャストできます。

- xPrsNode 型の Pre と Suc ― レディ キューにあるアクティブ クラスのインスタンスをリンクします（973 ページの「レディ キュー」）。
- int 型の RestartAddress ― 実行の再開位置として適切なシンボルを検索します。
- xPrdNode 型の ActivePrd ― このインスタンスから呼び出されて現在実行中の操作を表す xPrdRec へのポインタです。呼び出されている操作が現在ない場合、ポインタは 0 です。
- RestartPAD ― 関数へのポインタです。このコンポーネントは、アクティブ クラスの動的な振る舞いを実現する状態機械を実装する PAD 関数を参照します。アクティブ クラス間の継承の処理には RestartPAD が使用されます。
- int 型の CallAddress ― このアクティブ クラスによって現在実行されている操作呼び出しのシンボル番号が含まれます。
- xPrsNode 型の NextPrs ― アクティブ リストまたは使用可能メモリ リストのどちらかにある、このアクティブ クラスのインスタンスをリンクするために使用します。これらの 2 つのリストの先頭に、現在のアクティブ クラスを表す IdNode 内のコンポーネント ActivePrsList と AvailPrsList があります。
- SDL_PId 型の Self ―アクティブ クラスの現在のインスタンスの Pid 値です。
- xPrsIdNode 型の InstNameNode および TypeNameNode ― 現在のアクティブ クラスまたはそのインスタンス化を表す PrsIdNode へのポインタです。
- int 型の State ―アクティブ クラスのインスタンスの現在の状態を表すために使用される int 値が含まれています。
- Signal of type xSignalNode ―シグナル インスタンスへのポインタ。参照されるシグナルは、アクティブ クラスの現在のインスタンスによる次の遷移を引き起こすシグナルか、またはアクティブ クラスのインスタンスによって現在実行されている遷移を引き起こしたシグナルです。
- xInputPortRec 型の InputPort ― アクティブ クラスのインスタンスの入力ポートを表す二重リンク リストの待ち行列の先頭です。シグナルは、xSignalRec 構造体の Pre および Suc コンポーネントを使用してこのリスト内でリンクされます。

- SDL_PId 型の Parent － アクティブ クラスの現在の親インスタンスの Pid 値です。アクティブ クラスの「静的」インスタンスは、値が NULL の親を持ちます。(静的とは、インスタンス数が固定されていて、すべてのインスタンスが起動時に作成されることを意味します)。
- SDL_PId 型の Offspring － アクティブ クラスのもっとも新しく作られたインスタンスの Pid 値です。インスタンスを作っていないアクティブ クラスの offspring 値は null です。
- int 型の BlockInstNumber －アクティブ クラスが、合成として定義された型属性として別のアクティブ クラスの一部となっている場合、どのインスタンスの一部になっているのかを見分けるために使用します。
- xSignalIdNode 型の pREPLY_Waited_For － アクティブ クラスが「リモートプロシージャコール」において pREPLY シグナルを暗黙の状態で待っているとき、予期される pREPLY シグナルの IdNode を格納します。
- xSignalIdNode 型の pREPLY_Signal －アクティブ クラスのインスタンスは pCALL シグナルを受信、すなわち、「リモートプロシージャコール」を受信すると、即座に戻り シグナルである pREPLY シグナルを作成します。この pREPLY シグナルが送出されるまでの間、シグナルを参照するために使用されます。
- SDL_PId 型の Sender － 送信側を表します。
- int 型の Trace_Default － アクティブ クラスのインスタンスについて定義されたトレースの現在値。
- int 型の GRTrace － アクティブ クラスのインスタンスについて定義されたグラフィカル トレースの現在値。
- int 型の MSCETrace － アクティブ クラスのインスタンスについての現在のシーケンス図トレース値。
- xbool 型の InTransition － アクティブ クラスが遷移を実行中は true となり、アクティブ クラスが待機の状態にある間は false となります。モデル ベリファイヤ（Model Verifier）のユーザー インターフェイスはこの情報を使用して、関連情報を表示します。

レディ キュー

レディ キューは先頭付きの二重のリンク リストです。ここには、即時遷移が可能だが遷移の完了を許可されなかったアクティブ クラスのインスタンスが含まれています。アクティブ クラスのインスタンスは signal sending 操作時と nextstate 操作時にレディ キューに挿入され、現在の遷移を停止する nextstate または stop 操作が実行されるとレディ キューから削除されます。レディ キュー内の先頭はインスタンスとしては使用されずキュー操作の簡便化のために使用されます。先頭は次の xSysD コンポーネントによって参照されます。

```
xPrsNode   xReadyQueue;
```

このコンポーネントは関数 xInitKernel 内で起動され、レディ キューを参照するためにランタイム ライブラリ全体で使用されます。

イベントのスケジューリングは、初期化実行後に main 関数から呼び出される関数 xMainLoop によって実行されます。

```
void xMainLoop()
```

この方法は、すべての対象の待ち行列（レディ キュー、タイマー待ち行列、入力ポート）をいつも正しい順序でソートさせるためにあります。この結果、ソートはオブジェクトが待ち行列に挿入されたときに実行されます。すなわち、スケジューリングは単純なタスクになります。タイマー待ち行列またはレディ キューからから最初のオブジェクトを選択し、そのオブジェクトを実行に移します。

コンパイル スイッチのさまざまな組み合わせに対して使用される関数 `xMainLoop` には、複数のバージョンのエンドレス ループの本体があります。遷移とタイマー出力のスケジューリングについては、概要は以下のとおりです。

```
while (1) {
  if ( xTimerQueue->Suc->TimerTime <= SDL_Now() )
    SDL_OutputTimerSignal( xTimerQueue->Suc );
  else if ( xReadyQueue->Suc != xReadyQueue) {
    xRemoveFromInputPort(xReadyQueue->Suc->Signal);
    xReadyQueue->Suc->Sender =
       xReadyQueue->Suc->Signal->Sender;
    (*xReadyQueue->Suc->RestartPAD)(xReadyQueue->Suc);
  }
}
```

または、記述表現では、

```
while (1) {
  if ( there is a timer that has expired )
    send the corresponding timer signal;
  else if ( there is an instance that can execute
            a transition ) {
    remove the signal causing the transition
      from input port;
    set up Sender in the instance to Sender of
      the signal;
    execute the PAD function for the state machine;
  }
}
```

メイン ループのさまざまなバージョンが、コンパイル スイッチの違った組み合わせを扱います。このメイン ループで必要な他のアクションはコンパイル スイッチによって決まります。かかるアクションの例を以下に示します。

- モデル ベリファイヤ（Model Verifier）の ユーザー インターフェイスのハンドリング
- `xInEnv` 関数の呼び出し
- 実時間またはシミュレートした時間のハンドリング
- スケジュールされた次のイベントまで実行を遅延
- 再計算が必要な、ガードまたはトリガされた遷移上のガードのハンドリング

create 操作と **stop** 操作

アクティブ クラスのインスタンスは、それがアクティブである間は、次の 2 つの構造体によって表現されます。

- `xLocalPIdRec`
- `yVDef_ProcessName` 構造体

これらの 2 つの構造体は動的に割り当てられます。`Pid` 値は 2 つのコンポーネント `GlobalNodeNr` と `LocalPId` を含む構造体（未割り当て）です。ここで、`LocalPId` は `xLocalPIdRec` へのポインタです。975 ページの図 242 に、アクティブ クラスのインスタンスを表す `xLocalPIdRec` と `yVDef_ProcessName` の構造体の接続方法が示されています。

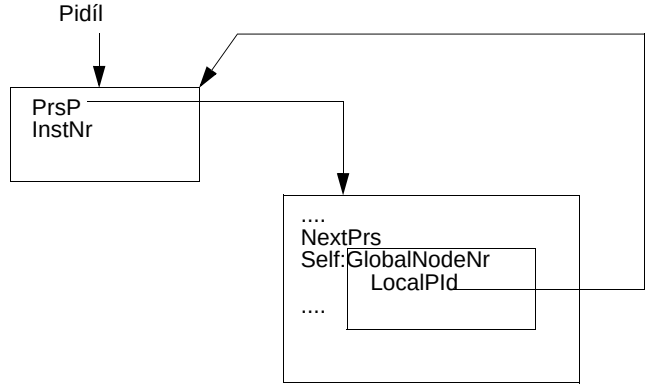


図 242: アクティブ クラスのインスタンスを表す xLocalPIdRec と yVDef_ProcessName

アクティブ クラスのインスタンスが stop アクションを実行したとき、そのインスタンスに使われているメモリを取り戻して、その後の create アクションで再利用できるようにしなければなりません。ただし、stop アクションの後、前の（無効な）`Pid` 値をアクティブ クラスの他のインスタンスの属性に保存できます。

シグナルがこのような前の `Pid` 値、すなわち、アクティブ クラスの停止したインスタンスへ送られた場合、該当するアクションを探して実行することが可能でなければなりません。アクティブ クラスのインスタンスの完全な表現を再利用される場合、これは可能ではありません。したがって、少しの情報とひいてはこれまで存在していたアクティブ クラスの各インスタンスのための多少のメモリが残っていなければなりません。これが `xLocalPIdRec` の目的です。これらの構造体の再利用は決してありません。その代わりに、アクティブ クラスのインスタンスが stop アクションを実行したとき、以下のことが発生します。

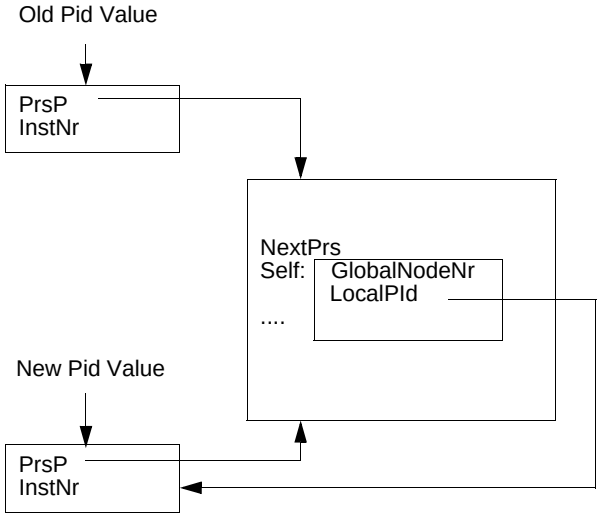


図 243: アクティブ クラスのインスタンスが stop アクション実行後のメモリ構造

- 新しい xLocalPIdRec が割り当てられ、その PrsP が yVDef_ProcessName を参照します（InstNr は 0）。
- yVDef_ProcessName の Self コンポーネントは新しい xLocalPIdRec を参照するように変更されます。
- それでも、前の xLocalPIdRec は yVDef_ProcessName を参照します。
- yVDef_ProcessName がアクティブ クラスの使用可能メモリ リストに入ります。

アクティブ クラスのインスタンスのデータ領域を create 操作で再利用するには、yVDef_ProcessName を使用可能メモリ リストから削除し、Self によって参照される xLocalPIdRec の InstNr コンポーネントを更新することだけが必要です。

アクティブ クラスのインスタンスを表す多少複雑な構造体を使用して、Pid 値がアクティブなインスタンスまたは停止したインスタンスを参照しているかどうかを知るための単純なテストを行うことができます。

P が Pid 変数の場合、次の式

```
P.LocalPId == P.LocalPId->PrsP->Self.LocalPId
```

は、アクティブ クラスのインスタンスがアクティブならば true、停止していれば false となります。

create および stop 操作の基本的な振る舞いは、関数 SDL_Create と SDL_Stop によって実行されます。

```
void SDL_Create(
  xSignalNode  StartUpSig,
```

```
    xPrsIdNode   PrsId )
void SDL_Stop( xPrsNode  PrsP )
```

アクティブ クラスのインスタンスを作成するには、以下の手順を生成されたコード内で実行します。

1. `xGetSignal` を呼び出して起動シグナルを取得します。
2. 状態機械のパラメータを起動シグナル パラメータに割り当てます。
3. 作成されるアクティブ クラスを表す `PrsIdNode` に加えて、起動シグナルをパラメータとして、`SDL_Create` を呼び出します。

`xGetProcess` では、アクティブ クラスのインスタンスはインスタンス セットの使用可能メモリ リスト（アクティブ クラスのインスタンスを表す `PrsIdNode` のコンポーネント `AvailPrsList`）から削除されるか、あるいは使用可能メモリ リストが空の場合は新しいメモリが割り当てられます。

アクティブ クラスのインスタンスはアクティブなインスタンスのリストにリンクされます（インスタンス セットを表す `PrsIdNode` のコンポーネント `ActivePrsList`）。使用可能メモリ リストとアクティブなリストの両方は、`yVDef_ProcessName` 構造体のコンポーネント `NextPrs` をリンクとして使用した単一リンクリスト（ヘッドなし）です。

開始遷移と他の遷移を平等に扱うには、start ステートを「start ステート」という名前の通常のステートとして実装します。start ステートは 0 で表されます。初期遷移を実行するために、「起動」シグナルがアクティブ クラスへ送られます。したがって、start ステートとは、起動シグナルを受信し、他のすべてのシグナルが保存するステートであると理解できます。この実装はモデル ベリファイヤ（Model Verifier）内では完全に透過的であり、起動シグナルは決して表示されません。

注記

仮パラメータの実際値が起動シグナルで受け渡されます。

シンボル テーブル ツリーの一部ではない 2 つの `IdNodes` が、start ステートと起動シグナルを表すために作成されます。

```
xStateIdNode   xStartStateId;
xSignalIdNode  xStartUpSignalId;
```

これらの `xSysD` コンポーネントは、`sctsdl.c`. の一部である関数 `xInitSymbolTable` 内で初期化されます。

stop 操作時には、関数 `SDL_Stop` が呼び出されます。この関数は、現在の遷移を発生したシグナルと入力ポートにある他のすべてのシグナルを解放します。また、この関数は、最初のパラメータを `0` に設定した `xRemoveTimer` を呼び出して、アクティブ クラスのこのインスタンスに接続されているタイマー待ち行列内のすべてのタイマーを削除します。次にこの関数は、stop 操作を実行するアクティブ クラスのインスタンスをレディ キューとアクティブ クラスのアクティブ リストから削除し、メモリを現在のインスタンス セットの使用可能メモリ リストに返します。

シグナルの送受信

一般原則

シグナルを送信するには生成されたコード内で 3 つのアクションが実行されます。

1. 最初に、`xGetSignal` を呼び出して、シグナル インスタンスを表すデータ領域を取得します。
2. 次に、シグナル パラメータをこれらの値に割り当てます。
3. 最後に、関数 `SDL_Output` を呼んで、シグナルを実際に送出します。

SDL_Output の詳細な操作

`SDL_Output` 関数では、最初に複数の動的テストを実施し、直接アドレッシング状態にある受信側が NULL でなく、停止していないかどうかを確認し、また受信側へのパスがあるかどうかを確認します。

シグナル送信に直接アドレッシングが含まれてなくて C コード ジェネレータ が受信側を判断できない場合は、`xFindReceiver` 関数が呼び出されて受信側を判断します。

次に、`SDL_Output` で、状況に基づいて、シグナルが処理されます。ここでは 3 つのケースを特定できます。

1. 最初に、環境関数 `xOutEnv` が呼び出されます。
2. 次に、環境を表すアクティブ クラスのインスタンスの入力ポートにシグナルが挿入されます（`xEnv`）。
3. 最後に、システムの内部シグナルが処理されます。ここで、3 つのケースを特定できます（この評価の方法についてはこのサブセクションの最後に説明します）。

- シグナルは受信側による即時遷移を発生できます。
- シグナルを保存しなければなりません。
- シグナルを即座に破棄しなければなりません。
    - シグナルが即時遷移を引き起こすことができる場合、シグナルは受信側の入力ポートに挿入され、アクティブ クラスの受信インスタンスがレディ キューに挿入されます。
    - シグナルを保存しなければならない場合、シグナルは受信側の入力ポートに挿入されます。
    - シグナルを破棄しなければならない場合、このシグナルのデータ領域を再利用するために、関数 `xReleaseSignal` が呼び出されます。

シグナルが、アクティブ クラスの現在のインスタンスによって次の遷移を発生する（signal sending または Nextstate 操作時）シグナルであると特定されたときは、アクティブ クラスの `yVDef_ProcessName` のコンポーネント `Signal` がそのシグナルを参照するように設定されます。シグナルはやはり入力ポート リストの一部です。

遷移を実行しなければならないとき、ステート マシーンの PAD 関数が呼び出される直前に、シグナルはメイン ループの入力ポートから削除されます。

最初に、PAD 関数で、input 文にしたがって、シグナルのパラメータがローカル変数にコピーされます。遷移の終わりの Nextstate または Stop 操作で、シグナル インスタンスは使用可能メモリ リストに返されます。

受信したシグナルの処理方法の評価

ランタイム ライブラリには、シグナルの処理方法（受信、送信、保存、破棄 ...）の評価が必要な場所が 2 箇所あります。

- アクティブ クラスの現在使われていないインスタンスへの signal sending 操作時。
- アクティブ クラスのインスタンスが入力ポートにシグナルを持っている Nextstate 操作時。

この計算はランタイム カーネル関数 `xFindInputAction` で実装されます。

```
typedef unsigned char xInputAction;
#define xDiscard         (xInputAction)0
#define xInput           (xInputAction)1
#define xSave            (xInputAction)2
#define xEnablCond       (xInputAction)3
#define xPrioInput       (xInputAction)4

static xInputAction xFindInputAction(
  xSignalNode  SignalId,
  xPrsNode     VarP,
  xbool        CheckPrioInput )
```

この関数のパラメータは次のとおりです。

- `SignalId`。これはシグナルへのポインタです。
- `VarP`。これはアクティブ クラスのインスタンスへのポインタです。

この関数の結果は以下のとおりです。

- この関数は、アクティブ クラス間、仮想または再定義の遷移間などの継承と同じように、アクティブ クラスのこのインスタンスに関するすべての情報を考慮に入れて、このシグナルに対して実行すべきアクション（受信、保存 ....）を返します。
- 関数の結果が `xInput` または `xPrioInput` の場合、`VarP` 構造体の `RestartPAD` および `RestartAddr` コンポーネントが、この入力の存在場所に関する情報で更新されます。

この最新の更新後に、スケジューラによって正しい遷移を開始できます。`RestartPAD` によって参照される関数を呼び出すことによって、実行される最初のアクションは `RestartAddr` の切り替えとシグナル受信シンボルの実行の開始です。

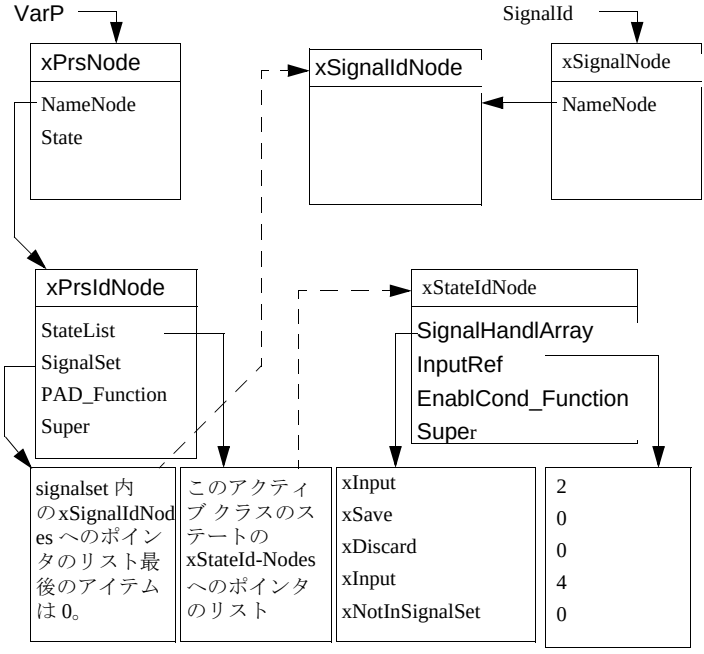


図 244: xFindInputAction の評価に使用されるデータ構造体

`InputAction`、`RestartAddr`、および `RestartPAD` を見つけるためのアルゴリズムは以下のとおりです。

1. `ProcessId` が `yVarP->NameNode` になるように、`StateId` が `ProcessId->StateList[yVarP->State]` になるようにします。
2. `ProcessId->SignalSet` で、`SignalId->NameNode` が見つかるインデックス `(Index)` を検索します。このシグナルが見つからなかった場合、このシグナルはアクティブ クラスのシグナル セットにはなく、アルゴリズムは結果 `xDiscard` を返して終了します。
3. `StateId->SignalHandlArray[Index]` は実行の対象となるアクションを与えます。この値が `xEnablCond` の場合、関数 `StateId->EnablCond_Function` が呼び出されます。この関数は `xInput` または `xSave` のどちらかを返します。

4. 先のテストの結果が `xInput` の場合、アルゴリズムはこの値を返して終了します。また `yVarP->RestartAddr` は更新されて `StateId->InputRef[Index]` となり、`yVarP->RestartPAD` は更新されて `ProcessId->PAD_Function` となります。
   - 先のテストの結果が `xSave` の場合、アルゴリズムはこの値を返して終了します。
   - 先のテストの結果が `xDiscard` であり、また `ProcessId->Super` が `NULL` の場合、アルゴリズムはこの値を返して終了します。
   - 先のテストの結果が `xDiscard` であり、また `ProcessId->Super` が `NULL` でない場合、これは別のアクティブ クラスから継承するアクティブ クラスです。次に、`ProcessId` に値 `ProcessId->Super` を割り当て、`StateId` に値 `StateId->Super` を割り当ててから、これらのテストを再度実行する必要があります。

Nextstate 操作

nextstate 操作は `SDL_Nextstate` 関数によって実装されます。この関数では以下のアクションが実行されます。

1. 現在の遷移を引き起こしたシグナル（`yVDef_ProcessName` のコンポーネント `Signal`）は解除され、ステート変数（`yVDef_ProcessName` のコンポーネント `State`）は新しいステートへ更新されます。
2. 次に、遷移を起こせるシグナルを見つけるためにアクティブ クラスの入力ポートがスキャンされます。このスキャン中、受信で指定したシグナルが見つかるまで、シグナルを保存するか、あるいは廃棄できます。
3. 遷移を引き起こせるシグナルが見つからなかった場合、連続したシグナルが遷移を引き起こせるかどうかの確認が行われます（982 ページの「ガードとトリガされた遷移上のガード」）。その後、アクティブ クラスのインスタンスがレディ キューから削除されます。
4. シグナル（またはトリガされた遷移上のガード）が遷移を引き起こせる場合、アクティブ クラスのインスタンスがレディ キューに再挿入されます。新しいステートにガードまたはトリガされた遷移上のガードが含まれ、アクティブ クラスのインスタンスがその新しいステートにある間にガードに含まれる式が値を変化できる場合、そのインスタンスはチェック リストに挿入されます。

参照

第 28 章「C コード ジェネレータ ランタイム モデル」の 982 ページ、「ガードとトリガされた遷移上のガード」

分岐操作とアクション操作

Decision 操作と Action 操作は生成されたコードに実装されます。ただし、`sctutil.c` および `sctda.c` ファイルで実装される Trace 関数と、インフォーマル分岐および `sctda.c` のサポート関数のいずれかを使用するすべての分岐を除きます。分岐は C の if 文として実装され、Action での代入は C の代入または関数呼び出しとして実装されます。

複合文

属性の宣言のない複合文は、複合文に含まれるアクションのシーケンスに変換され、属性の宣言のある複合文は操作（パラメータのない）と同じ方法で変換されます。

複合文内の new 文型は以下の規則にしたがって変換されます。

- 条件文は C の `if` 文に変換されます。
- 複合文の分岐は通常の分岐に変換されます。
- loop、continue、および break はすべて C の `goto` 文を使って変換されます。

ガードとトリガされた遷移上のガード

ガードまたはトリガされた遷移上のガードに関与する式は、`yCont_StateName` および `yEnab_StateName` という名前の関数内で生成されたコードに実装されます。これらの関数は、ガードおよびトリガされた遷移上のガードを含むステートごとに生成されます。これらの関数は、`StateIdNode` のコンポーネント `ContSig_Function` および `EnablCond_Function` を通してそのステートが参照されます。対応する関数が生成されなければ、これらのコンポーネントは `0` となります。

`EnablCond_Functions` は関数 `xFindInputAction` から呼び出され、この xFindInputAction は `SDL_Output` と `SDL_Nextstate` から呼び出されます。現在のシグナルのガードが true の場合は `xInput` が返され、true でない場合は `xSave` が返されます。次に、この情報はこの状態でシグナルをどのように処理するかを決定するために使用されます。

コンポーネント `ContSig_Function` が `0` でなく、即時遷移を引き起こせるシグナルが入力ポート スキャンで見つからなかった場合、`ContSig_Functions` が `SDL_Nextstate` から呼び出されます。`ContSig_Function` のプロトタイプを以下に示します。

```
void ContSig_Function_Name (
  void *, int *, xIdNode *, int *);
```

ここで、

- 最初のパラメータが `yVDef_ProcessName` へのポインタです。

- 残りのパラメータはすべて out パラメータです。
  - 2 番目のパラメータは、その値が >= 0 の場合、3 番目と 4 番目のパラメータが使用されることを指示します。
  - 3 番目のパラメータは、トリガされた遷移上の実際のガードを見つけることができるアクティブ クラスまたは操作のための `IdNode` です。
  - 4 番目は、このガードに接続されている `RestartAddress` です。

トリガされた遷移上のガードの式の値が true の場合、ガードを表すシグナル インスタンスが作成されて入力ポートに挿入され、その後、通常のシグナルとして扱われます。.. シグナル型は連続シグナルで、`xSignalIdNode` によって表されます（変数 `xContSigId` によって参照）。

チェックリストには、ガードおよびトリガされた遷移上のガードを繰り返し再計算する必要がある状態で待機しているアクティブ クラスのインスタンスを含んでいます。

以下の場合、アクティブ クラスのインスタンスがチェックリストに挿入されます。

- このインスタンスが、ガードまたはトリガされた遷移上のガードを含む状態になっています。
- シグナルまたはガードが即時遷移を引き起こすことができません。
- ガードまたはトリガされた遷移上のガードの 1 つまたは複数の式が値を、含まれている状態機械がある状態（view、import、now...）にある間に変更できます。

`StateIdNode` のコンポーネント `StateProperties` は、当該の式がその状態に存在するかどうかを反映します。

チェック リストが `xSysD` コンポーネントによって表されます。

```
xPrsNode  xCheckList;
```

ガードまたはトリガされた遷移上のガードの振る舞いのモデル化を、アクティブ クラスのインスタンスに自分自身に繰り返しシグナルを送信させ、その結果、現在の状態を繰り返させることによって実行させます。ここで選択した実装では、nextstate 操作は、PAD 関数の呼び出しが完了した直後、すなわち、遷移が終了した直後で、しかもタイマー出力の直後に、チェック リストにあるアクティブ クラスの全インスタンスの「背後」で実行されます。これは、プログラムのメイン ループで関数 `xCheckCheckList` を呼び出すことによって実行されます。

グローバル属性

グローバル属性については、2 つのコンポーネントが `yVDef_ProcessName` 構造体にあります。 1 つは属性の現在値のためのもので、もう 1 つは属性の現在「エクスポート中」の値のためのものです。また、グローバル属性ごとに、対応する `RemoteVarIdNode` 内のリストにリンク可能な構造体があります。次に、このリストを使用して、「インポート」アクションで属性の適切な「エクスポータ」を参照するときに、そのエクスポータを探します。

「インポート」アクションはより複雑です。「インポート」アクションには、ほとんどの場合、関数 `xGetExportAddr` の呼び出しが必要です。

```
void * xGetExportAddr (
```

```
  xRemoteVarIdNode RemoteVarNode,
  SDL_PId          P,
  xbool            IsDefP,
  xPrsNode         Importer )
```

- RemoteVarNode は、リモート属性を表す（暗黙または明示的）RemoteVarIdNode への参照です。
- P はインポート アクションで与えられる Pid 式です。
- IsDef は、Pid 式がインポート アクションで与えられるかどうかに基づいて 0 か 1 かが決まります。
- Importer はアクティブ クラスのインポートするインスタンスです。

xGetExportAddr は「インポート」アクションが適切かどうかを確認し、Pid 式が与えられなかった場合は、どのアクティブ クラスからインポートするかを計算します。

エラーがなかった場合、この関数は属性値の場所を示すアドレスを返します。次に、このアドレスが正しい型にキャストされてから（生成されたコードで）、値を取得します。「インポート」先となるアクティブ クラスが存在しない場合、xGetExportAddr 関数がゼロのみが入った属性のアドレスを返します。

操作

操作を表すデータ構造体

アクティブ クラスの操作は構造型によって表現されます。scttypes.h で定義される xPrdRec 構造体は操作に関する一般情報を格納している構造体であり、その操作のパラメータと属性はアクティブ クラスと同じ方法で生成されたコードで定義されます（970 ページの「アクティブ クラスを表すデータ構造」）。

scttypes.h には、操作に関する以下の型が含まれています。

```
#define PROCEDURE_VARS ¥
   xPrdIdNode   NameNode; ¥
   xPrdNode     StaticFather; ¥
   xPrdNode     DynamicFather; ¥
   int          RestartAddress; ¥
   XCALL_ADDR ¥
   void (*RestartPAD) (xPrsNode  VarP); ¥
   xSignalNode  pREPLY_Signal; ¥
   int          State;

typedef struct xPrdStruct  *xPrdNode;

typedef struct xPrdStruct {
   PROCEDURE_VARS
}  xPrdRec;
```

生成されたコードでは、yVDef_ProcedureName 構造体は以下のとおりに定義されます。

```
typedef struct {
   PROCEDURE_VARS
   components for FPAR and DCL
}  yVDef_ProcedureName;
```

xPrdRec のコンポーネントの使い方を以下に説明します。

- xPrdIdNode 型の NameNode ― 操作型を表す IdNode へのポインタです。
- xPrdNode 型の StaticFather ― 操作範囲の階層（および最上位のアクティブ クラス）を表すポインタで、操作が非ローカル属性を参照するときに使用されます。この例を 986 ページの図 245 に示します。StaticFather == 0 は static father がアクティブ クラスであることを意味しています。
- xPrdNode 型の DynamicFather ― この操作が参照される操作によって呼び出されることを表すポインタです。DynamicFather == 0 は、この操作がアクティブ クラスから呼び出される操作であることを意味します。また、このコンポーネントを使用して、操作型の使用可能メモリ リストにある xPrdRec にリンクします。
- int 型の RestartAddress ― このコンポーネントを使用して、実行の再開位置となる該当シンボルを探します。
- int 型の CallAddress ― この操作から実行される操作呼び出し（もしあれば）のシンボル番号が格納されます。
- RestartPRD ― 操作関数へのポインタです。このコンポーネントは、次のシンボルのシーケンスを実行する PRD 関数を参照します。操作間の継承の処理には RestartPRD が使用されます。
- xSignalIdNode 型の pREPLY_Signal ― アクティブ クラスのインスタンスは pCALL シグナルを受け取ると、すなわち、RPC を受け取ると、リターン シグナルの pREPLY シグナルを作成します。このコンポーネントは、この pREPLY シグナルが送信されるまでこのシグナルを参照するために使用されます。
- int 型の State ― 操作の現在のステートを表す値です。

986 ページの図 245 は、四重にネストされた操作呼び出しの実行後の構造体 yVDef_ProcedureName の例です。操作 Q はアクティブ クラス内で、操作 R と S は Q 内で、T は S 内で宣言されます。

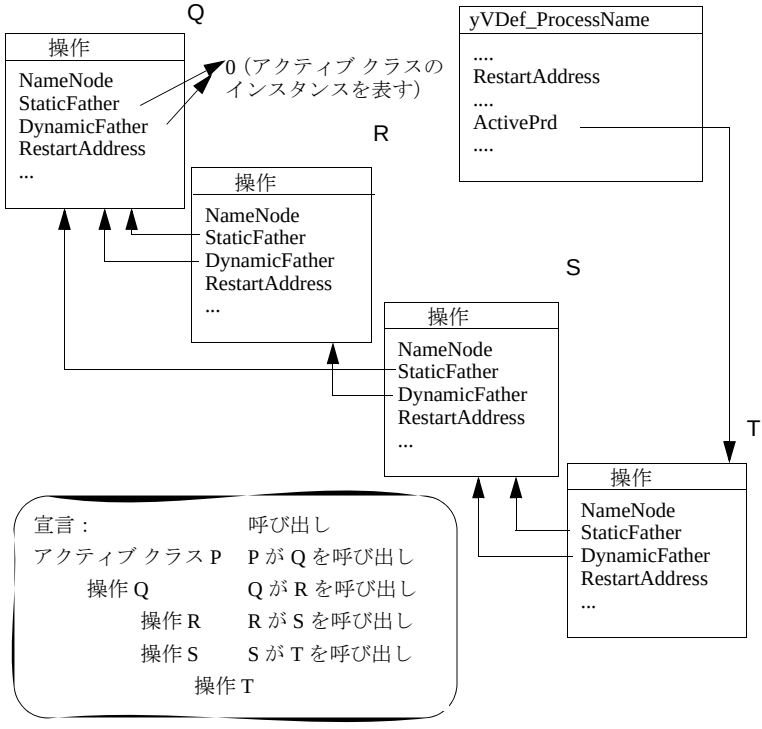


図 245: 4 重にネストされた操作呼び出し後の構造体 `yVDef_ProcedureName`

PRD 関数

操作は、C 関数と上記の構造体をそれぞれ一部使用して実装されます。各操作は **PRD 関数**によって表現されます。この関数は、操作で定義されるアクションを実行するために呼び出される C 関数です。この関数は、状態機械の PAD 関数に対応します。ただし、仮パラメータと属性を、生成されたコードで定義された構造体を使用して実装されます。ネストされた操作呼び出しの操作スタックは、コンポーネント `StaticFather` と `DynamicFather` を使用して実装され、C 関数スタックを使用しません。

操作の呼び出しと操作からの戻り

操作の呼び出しと操作からの戻りは 3 つの関数によって扱われます。

- 1 つの関数は操作のデータ領域の割り当てを扱います。

```
xPrdNode  xGetPrd( xPrdIdNode  PrdId )
```

- 2 つの関数は、操作の呼び出し時と操作からの戻り時に生成されたコードから呼び出されます。

```
void xAddPrdCall(
  xPrdNode  R,
  xPrsNode  VarP,
  int       StaticFatherLevel,
  int       RestartAddress )

void xReleasePrd (xPrsNode  VarP)
```

操作の呼び出しは以下のステップで表現される C で実行されます。

1. `xGetPrd` を呼び出して操作のデータ領域を取得します。
2. 操作パラメータをデータ領域に割り当てます。
3. `xAddPrdCall` を呼び出して静的および動的なチェーンに操作をリンクします。
4. 操作をモデル化する C 関数である `yProcedureName` 関数を呼び出します。

`xAddPrdCall` へのパラメータを以下に説明します。

- `R`: これは、`xGetPrd` の呼び出しから得られる `xPrdNode` への参照です。
- `VarP`: これは、アクティブ クラスの属性およびパラメータのデータ領域である `yVDef_ProcessName` への参照です（アクティブ クラスが操作呼び出しを実行した呼び出しの場合でも）。
- `StaticFatherLevel`: これは、呼び出し側と呼び出される操作との間の宣言のレベルの違いです。この情報は、`StaticFather` コンポーネントを正しく設定するために使用されます。
- `RestartAddress`: これは操作呼び出しの直後のシンボルのシンボル番号です。シンボル番号は、すべてのシンボルについて生成されるスイッチ ケース ラベルです。

`xGetPrd` は `xPrdRec` へのポインタを返します。xPrdRec は次に、操作のパラメータと属性を表すデータ領域のコンポーネントに対して値を直接割り当てるために使用できます。IN/OUT パラメータはこの構造体内でアドレスとして表現されます。

操作の戻りは、`xReleasePrd` の呼び出しとそれに続く戻り値 0 によって表現される生成されたコードで行われます。それによって、操作の振る舞いを表す関数が残ります。

操作の振る舞いを表す関数が次の 2 つの状態で戻されます。

- `Return` に至ったとき（関数が 0 を返します）。
- `Return` に至ったとき（関数が 1 を返します）。

0 が返された場合。操作呼び出し後に次のシンボルを使って実行が継続します。1 が返された場合、アクティブ クラスのインスタンスの実行が終了し、スケジューラ（メインループ）が主導権を握ります。このことは、複数のネストされた操作呼び出しが終了することを意味しています。

次のステート操作の終了後に操作を再開するときに、正しいシンボルで実行を継続するために、操作呼び出しを持つ状態機械に対して、以下のコードが PAD 関数に導入されます。

```
while ( yVarP->ActivePrd != (xPrdNode)0 )
  if ((*yVarP->ActivePrd->RestartPRD)(VarP))
    return;
```

このことは、未完了の操作が操作スタックの下の方から 1 つずつ、すべての操作が完了するまでか、あるいはこれらの 1 つが 1 を返すまで、すなわち、次のステートの操作を実行するまで、再開されることを意味します。この次のステート操作でアクティブ クラスのインスタンスがスケジューラにふたたびゆだねられます。

コネクタ

アクティブ クラスの受信インスタンスの検索

コネクタおよびポート用の `ChannelIdNodes` が関数 `xFindReceiver` と `xIsPath` で使用されます。これらの 2 つの関数は `SDL_Output` から呼び出され、signal sending 文に直接アドレッシングがないとき、アクティブ クラスの受信インスタンスを探します。この関数のそれぞれが、signal sending 文の直接アドレッシングの場合に受信側へのパスが存在するかを調べます。

どちらの場合も、アクティブ クラスの `IdNodes` の `ToId` コンポーネントを使用してパスが構築され、コネクタがそれに続きます。989 ページの図 246 はこれらのパスの構造を示しています。

システムの初期化時に、シンボル テーブルが構築されます。次に、システムで始まるシンボル テーブルの一部は、989 ページの「小規模システムと結果として生まれるシンボル テーブルの例」に略述された構造を持っています。この例からわかるように、システムでの宣言は `IdNodes`. によって直接反映されます。

注記

各コネクタは、各方向に 1 つで合計 2 つの `IdNodes` によって表現されます。これは単方向のコネクタについても当てはまります。この場合、使用しない方向のシグナル セットは空になります。

小規模システムと結果として生まれるシンボル テーブルの例

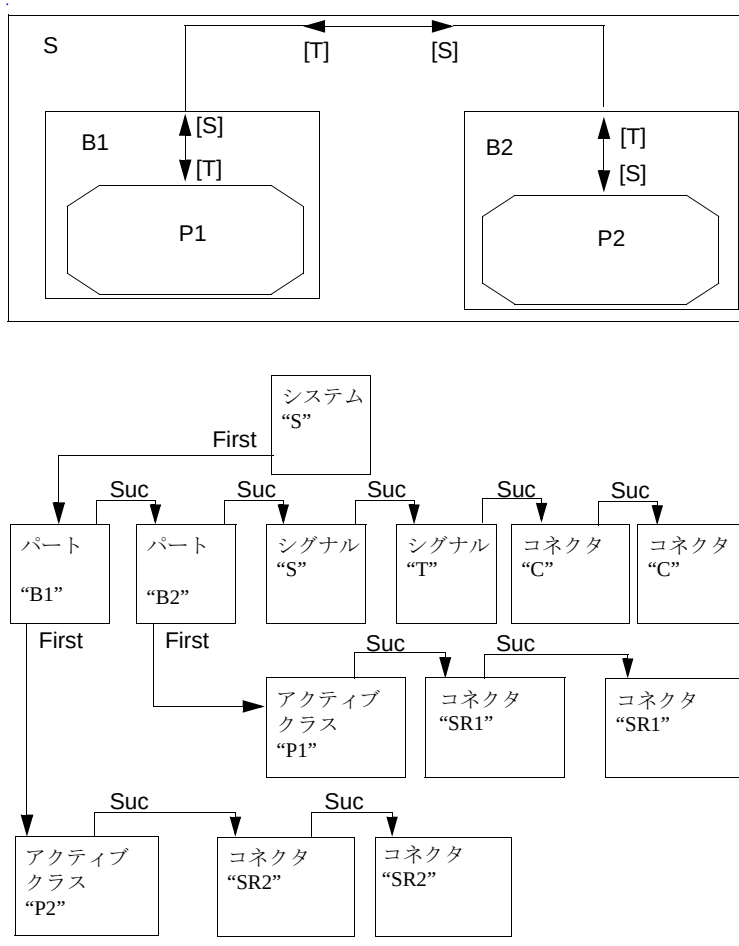


図 246: 上図のシステムのシンボル テーブル ツリー

各 `IdNode` はアクティブ クラスを表し、コネクタにはコンポーネント `ToId` が含まれます。`ToId` コンポーネントは、`IdNodes` に対する参照配列へのアドレスです。この配列のサイズはこのオブジェクトが接続されるアイテム数によって決まります。3 つの発信シグナル ルートを持つアクティブ クラスには、3 つのポインタに加えて終りの `0` ポインタを表現できる `ToId` 配列が含まれています。

989 ページの図 246 の例では、分岐が存在しないので、すべての `ToId` 配列のサイズは 2 つのポインタに必要な大きさです。

990 ページの図 247 は、アクティブ クラスのインスタンスの `IdNodes` で、コンポーネント `ToId` を使用してコネクタを接続して、パスを形成する方法を示しています。この場合、パスは単純なパスだけです（1 つは P1 から SR1、C、SR2 を経由して P2 へ、もう 1 つはその逆方向のパスです）。分岐を取り扱うこの構造体の汎化は単純です。

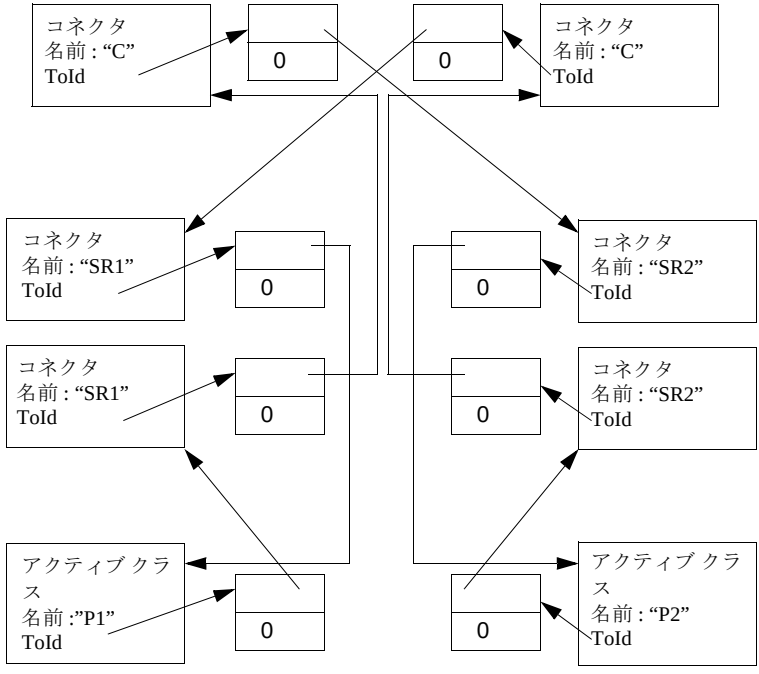


図 247: システムの `ToId` の接続

29

C コード ジェネレータ シンボル テーブル

本章は、C コード ジェネレータによって作成されるシンボル テーブルに関するリファレンス ガイドです。シンボル テーブルの用途は、パートに関する構造、コネクタ、アクティブ クラス向けの有効な入力シグナル セットといった、主としてアプリケーションの静的プロパティについての情報を格納することです。また、あるインスタンス セットのアクティブ クラスのすべてのアクティブ インスタンスのリストのような動的プロパティもシンボル テーブルに配置されます。

シンボル テーブルにあるノードは、宣言で初期化されたコンポーネントを持つ構造体です。アプリケーションの初期化時に、これらのノードからツリーがビルドされます。

シンボル テーブルの作成と構造

シンボル テーブルの作成

シンボル テーブルは次の 2 つのステップで作成されるツリーです。

1. シンボル テーブル ノードは、宣言で初期化されたコンポーネントを持つ構造体として宣言されます。
2. `yInit` 関数はノード内の一部のコンポーネントを更新し、ノードからツリーをビルドします。

シンボル テーブルの構造

以下の名前は、シンボル テーブルに常時存在するノードを参照するために使用できます。これらの名前は `scttypes.h` で定義されます。

```
xSymbolTableRoot
xEnvId
xSrtN_SDL_Bit
xSrtN_SDL_Bit_String
xSrtN_SDL_Boolean
xSrtN_SDL_Character
xSrtN_SDL_Charstring
xSrtN_SDL_Duration
xSrtN_SDL_IA5String
xSrtN_SDL_Integer
xSrtN_SDL_Natural
xSrtN_SDL_Null
xSrtN_SDL_NumericString
xSrtN_SDL_Object_Identifier
xSrtN_SDL_Octet
xSrtN_SDL_Octet_String
xSrtN_SDL_PId
xSrtN_SDL_PrintableString
xSrtN_SDL_Real
xSrtN_SDL_Time
xSrtN_SDL_VisibleString
```

`xSymbolTableRoot` はシンボル テーブル ツリー内のルート ノードです。このノードの下にシステム ノードが挿入されます。システム ノードの後に、システムの環境を表すノードが存在します（`xEnvId`）。次に、システムから参照される各パッケージに 1 つのノードが存在します。これは、定義済みのデータ型を含むパッケージについても当てはまります。定義済みのデータ型のノードは、パッケージ Predefined のノードの子であり、上記のリストに基づいて、名前 xSrtN_SDL_<type> によって直接参照できます。

シンボル テーブルのノード

シンボル テーブル内のノードは宣言の位置にしたがってツリーに配置されます。パートで宣言されたアイテムを表すノードはそのパート ノードの子ノードとなる、というように配置されます。このシンボル テーブル ツリーの階層は、パートおよびアクティブ クラス内の構造と宣言を直接反映します。

以下のノード型がツリー内に存在します。

ノード型	説明
xSystemEC	「ルート アクティブ クラス」のインスタンス、すなわち、システムまたはシステム インスタンスを表します。
xSystemTypeEC	「ルート アクティブ クラス」を表します。
xPackageEC	パッケージを表します。
xBlockEC	パートを表します。
xBlockTypeEC	パートを持つアクティブ クラスを表します。
xBlockSubstEC	パートの分解を表し、パート ノードの子として存在できます。
xProcessEC	アクティブ クラスとアクティブ クラスのインスタンスを表します。「環境アクティブ クラス」ノードはシンボル ノードの後に配置され、システムへの環境を表現するために使用されます。
xProcessTypeEC	アクティブ クラスを表します。
xProcedureEC	プロシージャを表します。
xOperatorEC	データ型またはパッシブ クラスでの操作を表します。
xCompoundStmtEC	属性宣言を含む複合文を表します。
xSignalEC xTimerEC	シグナルまたはタイマー型を表します。
xRPCSignalEC	「リモート」操作の呼び出しの実装に使用される暗黙のシグナル（pCALL、pREPLY）を表します。
xSignalParEC	シグナルまたはタイマー パラメータごとに 1 つのシグナル パラメータ ノード（シグナルおよびタイマーの子）が存在します。
xStartUpSignalEC	開始シグナル、すなわち、振る舞いを実装する状態機械の実パラメータが入っているアクティブ クラスの新しいインスタンスへ送信されるシグナルを表します。xStartUpSignalEC ノードは必ずそのアクティブ クラスのノードの直後に配置されます。
xSortEC xSyntypeEC	ニュータイプまたはシンタイプを表します。

ノード型	説明
Struct Component Node (xVariableEC)	1 つの構造体を表すソート ノードが、ソート定義内の各構造体コンポーネントについて、1 つの構造体コンポーネント ノードを子として持ちます。
xLiteralEC	enum 型に似たソート ノードが、リテラル リスト内の各リテラルについて、1 つのリテラル ノードを子として持ちます。
xStateEC	ステートを表し、アクティブ クラスとアクティブ クラス内の操作のためのノードの子として存在します。
xVariableEC xFormalParEC	状態機械への属性または仮パラメータを表し、アクティブ クラスとアクティブ クラス内の操作のためのノードの子として存在します。
xChannelEC xSignalRouteEC xGate	コネクタまたはポートを表します。
xRemoteVarEC	インターフェイス内の属性の定義を表します。
xRemotePrdEC	インターフェイス内の操作の定義を表します。
xSyntVariableEC	C コード ジェネレータによって導入される暗黙の変数またはコンポーネントを表します。
xSynonymEC	パッケージ内の属性を表します。使用されません。

シンボル テーブル内の名前付け

ノード（構造体変数）には、以下のとおりに、生成されたコード内で名前がつきます。

シンボル テーブル内の名前	対象
ySysR_SystemName	システム、システム型、システム インスタンス
yPacR_PackageName	パッケージ
yBloR_BlockName	パート、パートを持つアクティブ クラス、アクティブ クラスのインスタンス
yPrsR_ProcessName	インライン アクティブ クラス、アクティブ クラス、アクティブ クラスのインスタンス
yPrdR_ProcedureName	操作
ySigR_SignalName	シグナル、タイマー、起動シグナル、RPC シグナル
yChaR_ChannelName	コネクタ、ポート
yStaR_StateName	ステート
ySrtR_NewtypeName	ニュータイプ、シンタイプ
yLitR_LiteralName	リテラル

シンボル テーブル内の名前	対象
yVarR_VariableName	属性、仮パラメータ、シグナル パラメータ、構造体コンポーネント
yReVR_RemoteVariable	インターフェイス内の属性
yRePR_RemoteProcedure	インターフェイス内の操作

ノード参照

ほとんどの場合、ポインタを通してシンボル テーブル ノードを参照することが重要です。上記のテーブルに基づいて、属性のアドレスを使用することによって、すなわち、

```
& yPrsR_Process1
```

を使用して、当該の参照が得られます。また、将来の下位互換性の維持のために、以下の例に基づいたマクロが複数のエンティティ クラスについて生成されます。

```
#define yPrsN_ProcessName  (&yPrsR_ProcessName)
```

シンボル テーブル ノードを表す型

シンボル テーブルでの **xIdNode** 型の定義

以下の型定義はファイル `scttypes.h` で定義され、シンボル テーブルといっしょに使用されます。

```
typedef enum {
  xRemoteVarEC,
  xRemotePrdEC,
  xSignalrouteEC,
  xStateEC,
  xTimerEC,
  xFormalParEC,
  xLiteralEC,
  xVariableEC,
  xBlocksubstEC,
  xPackageEC,
  xProcedureEC,
  xOperatorEC,
  xProcessEC,
  xProcessTypeEC,
  xGateEC,
  xSignalEC,
  xSignalParEC,
  xStartUpSignalEC,
  xRPCSignalEC,
  xSortEC,
  xSyntypeEC,
  xSystemEC,
  xSystemTypeEC,
  xBlockEC,
  xBlockTypeEC,
  xChannelEC,
```

```
  xCompoundStmtEC,
  xSyntVariableEC
  xMonitorCommandEC
}   xEntityClassType;

typedef enum {
  xPredef, xUserdef, xEnum,
  xStruct, xArray, xGArray, xCArray,
  xOwn, xORef, xRef, xString,
  xPowerSet, xGPowerSet, xBag, xInherits, xSyntype,
  xUnionC, xChoice
}   xTypeOfSort;

typedef char  *xNameType;


typedef struct xIdStruct {
   xEntityClassType  EC;
#ifdef XSYMBTLINK
   xIdNode           First;
   xIdNode           Suc;
#endif
   xIdNode           Parent;
#ifdef XIDNAMES
   xNameType         Name;
#endif
}  xIdRec;

                              /*BLOCKSUBSTRUCTURE*/
typedef struct xBlockSubstIdStruct {
   xEntityClassType  EC;
#ifdef XSYMBTLINK
   xIdNode           First;
   xIdNode           Suc;
#endif
   xIdNode           Parent;
#ifdef XIDNAMES
   xNameType         Name;
#endif
}  xBlockSubstIdRec;

                                    /*LITERAL*/
typedef struct xLiteralIdStruct {
   xEntityClassType  EC;
#ifdef XSYMBTLINK
   xIdNode           First;
   xIdNode           Suc;
#endif
   xIdNode           Parent;
#ifdef XIDNAMES
   xNameType         Name;
#endif
   int               LiteralValue;
}  xLiteralIdRec;

                                    /*PACKAGE*/
typedef struct xPackageIdStruct {
   xEntityClassType  EC;
#ifdef XSYMBTLINK
   xIdNode           First;
```

```
   xIdNode           Suc;
#endif
   xIdNode           Parent;
#ifdef XIDNAMES
   xNameType         Name;
#endif
#ifdef XIDNAMES
   xNameType         ModuleName;
#endif
}  xPackageIdRec;

                                    /*SYSTEM*/
typedef struct xSystemIdStruct {
   xEntityClassType  EC;
#ifdef XSYMBTLINK
   xIdNode           First;
   xIdNode           Suc;
#endif
   xIdNode           Parent;
#ifdef XIDNAMES
   xNameType         Name;
#endif
   xIdNode          *Contents;
   xPrdIdNode       *VirtPrdList;
   xSystemIdNode     Super;
#ifdef XTRACE
   int               Trace_Default;
#endif
#ifdef XGRTRACE
   int               GRTrace;
#endif
#ifdef XMSCE
   int               MSCETrace;
#endif
}  xSystemIdRec;

                      /*CHANNEL,SIGNALROUTE,GATE*/
#ifndef XOPTCHAN
typedef struct xChannelIdStruct {
   xEntityClassType  EC;
#ifdef XSYMBTLINK
   xIdNode           First;
   xIdNode           Suc;
#endif
   xIdNode           Parent;
#ifdef XIDNAMES
   xNameType         Name;
#endif
   xSignalIdNode    *SignalSet; /*Array*/
   xIdNode          *ToId;      /*Array*/
   xChannelIdNode    Reverse;
}  xChannelIdRec;   /* And xSignalRouteEC.*/
#endif

                                    /*BLOCK*/
typedef struct xBlockIdStruct {
   xEntityClassType  EC;
#ifdef XSYMBTLINK
   xIdNode           First;
   xIdNode           Suc;
```

```
#endif
   xIdNode           Parent;
#ifdef XIDNAMES
   xNameType         Name;
#endif
   xBlockIdNode      Super;
   xIdNode          *Contents;
   xPrdIdNode       *VirtPrdList;
   xViewListRec     *ViewList;
   int               NumberOfInst;
#ifdef XTRACE
   int               Trace_Default;
#endif
#ifdef XGRTRACE
   int               GRTrace;
#endif
#ifdef XMSCE
   int               MSCETrace;
   int               GlobalInstanceId;
#endif
}  xBlockIdRec;

                                     /*PROCESS*/
typedef struct xPrsIdStruct {
   xEntityClassType  EC;
#ifdef XSYMBTLINK
   xIdNode           First;
   xIdNode           Suc;
#endif
   xIdNode           Parent;
#ifdef XIDNAMES
   xNameType         Name;
#endif
   xStateIdNode     *StateList;
   xSignalIdNode    *SignalSet;
#ifndef XOPTCHAN
   xIdNode          *ToId; /*Array*/
#endif
   int               MaxNoOfInst;
#ifdef XNRINST
   int               NextNr;
   int               NoOfStaticInst;
#endif
   xPrsNode         *ActivePrsList;
   xptrint           VarSize;
#if defined(XPRSPRIO) || defined(XSIGPRSPRIO) || ¥
    defined(XPRSSIGPRIO)
   int               Prio;
#endif
   xPrsNode         *AvailPrsList;
#ifdef XTRACE
   int               Trace_Default;
#endif
#ifdef XGRTRACE
   int               GRTrace;
#endif
#ifdef XBREAKBEFORE
   char   *(*GRrefFunc) (int, xSymbolType *);
   int               MaxSymbolNumber;
   int               SignalSetLength;
```

```
#endif
#ifdef XMSCE
   int                MSCETrace;
#endif
#ifdef XCOVERAGE
   long int          *CoverageArray;
   long int           NoOfStartTransitions;
   long int           MaxQueueLength;
#endif
   void             (*PAD_Function) (xPrsNode);
   void             (*Free_Vars) (void *);
   xPrsIdNode         Super;
   xPrdIdNode        *VirtPrdList;
   xBlockIdNode       InBlockInst;
#ifdef XBREAKBEFORE
   char              *RefToDefinition;
#endif
}  xPrsIdRec;

                                      /*PROCEDURE*/
typedef struct xPrdIdStruct {
   xEntityClassType  EC;
#ifdef XSYMBTLINK
   xIdNode            First;
   xIdNode            Suc;
#endif
   xIdNode            Parent;
#ifdef XIDNAMES
   xNameType          Name;
#endif
   xStateIdNode      *StateList;
   xSignalIdNode     *SignalSet;
   xbool            (*Assoc_Function) (xPrsNode);
   void             (*Free_Vars) (void *);
   xptrint            VarSize;
   xPrdNode          *AvailPrdList;
#ifdef XBREAKBEFORE
   char            *(*GRrefFunc) (int, xSymbolType*);
   int                MaxSymbolNumber;
   int                SignalSetLength;
#endif
#ifdef XCOVERAGE
   long int          *CoverageArray;
#endif
   xPrdIdNode         Super;
   xPrdIdNode        *VirtPrdList;
}  xPrdIdRec;


typedef struct xRemotePrdIdStruct {
   xEntityClassType    EC;
#ifdef XSYMBTLINK
   xIdNode             First;
   xIdNode             Suc;
#endif
   xIdNode             Parent;
#ifdef XIDNAMES
   xNameType           Name;
#endif
   xRemotePrdListNode  RemoteList;
```

```
}  xRemotePrdIdRec;

                                    /* SIGNAL, TIMER */
typedef struct xSignalIdStruct {
   xEntityClassType  EC;
#ifdef XSYMBTLINK
   xIdNode           First;
   xIdNode           Suc;
#endif
   xIdNode           Parent;
#ifdef XIDNAMES
   xNameType         Name;
#endif
   xptrint           VarSize;
   xSignalNode      *AvailSignalList;
   xbool           (*Equal_Timer) (void *, void *);
#ifdef XFREESIGNALFUNCS
   void            (*Free_Signal) (void *);
#endif
#ifdef XBREAKBEFORE
   char             *RefToDefinition;
#endif
#if defined(XSIGPRIO) || defined(XSIGPRSPRIO) ||
defined(XPRSSIGPRIO)
   int               Prio;
#endif
}  xSignalIdRec;  /* and xTimerEC, xStartUpSignalEC,
                     and xRPCSignalEC.*/

                                    /*STATE*/
typedef struct xStateIdStruct {
   xEntityClassType  EC;
#ifdef XSYMBTLINK
   xIdNode           First;
   xIdNode           Suc;
#endif
   xIdNode           Parent;
#ifdef XIDNAMES
   xNameType         Name;
#endif
   int               StateNumber;
   xInputAction     *SignalHandlArray;
   int              *InputRef;
   xInputAction    (*EnablCond_Function)
                          (XSIGTYPE, void *);
   void            (*ContSig_Function)
                  (void *, int *, xIdNode *, int *);
   int               StateProperties;
#ifdef XCOVERAGE
   long int         *CoverageArray;
#endif
   xStateIdNode      Super;
#ifdef XBREAKBEFORE
   char             *RefToDefinition;
#endif
}  xStateIdRec;

                                    /*SORT*/
typedef struct xSortIdStruct {
   xEntityClassType  EC;
```

```
#ifdef XSYMBTLINK
   xIdNode           First;
   xIdNode           Suc;
#endif
   xIdNode           Parent;
#ifdef XIDNAMES
   xNameType         Name;
#endif
#ifdef XFREEFUNCS
   void            (*Free_Function) (void **);
#endif
#ifdef XTESTF
   xbool           (*Test_Function) (void *);
#endif
   xptrint           SortSize;
   xTypeOfSort       SortType;
   xSortIdNode       CompOrFatherSort;
   xSortIdNode       IndexSort;
   long int          LowestValue;
   long int          HighestValue;
   long int          yrecIndexOffset;
   long int          typeDataOffset;
}  xSortIdRec;


                                    /*VARIABLE,...*/
typedef struct xVarIdStruct {
   xEntityClassType  EC;
#ifdef XSYMBTLINK
   xIdNode           First;
   xIdNode           Suc;
#endif
   xIdNode           Parent;
#ifdef XIDNAMES
   xNameType         Name;
#endif
   xSortIdNode       SortNode;
   xptrint           Offset;
   xptrint           Offset2;
   int               IsAddress;
}  xVarIdRec;    /* And xFormalParEC and
                       xSignalParEC.*/

typedef struct xRemoteVarIdStruct {
   xEntityClassType     EC;
#ifdef XSYMBTLINK
   xIdNode              First;
   xIdNode              Suc;
#endif
   xIdNode              Parent;
#ifdef XIDNAMES
   xNameType            Name;
#endif
   xptrint              SortSize;
   xRemoteVarListNode   RemoteList;
}  xRemoteVarIdRec;
```

すべてのテーブル ノードに共通のコンポーネント

型定義はシンボル テーブル ノードの内容を定義します。EC が適切な文字列に置き換わる各 `xECIdStruct` には、最初の 5 つのコンポーネントが共通して含まれます。これらのコンポーネントはシンボル テーブル ツリーのビルドに使用されます。これらのコンポーネントにアクセスするには、`xIdECNode` 型のいずれにもアクセスするポインタが必要です。シンボル テーブルでは、この目的のために、次の例に基づいて、`xECIdStruct` のそれぞれにアクセスするポインタ型が定義されます。

```
typedef XCONST struct xIdStruct  *xIdNode;
```

型 `xIdNode` は、ツリーを横断するときなど、一般的な型として使用されます。

すべての `xIdNode` にある 5 つのコンポーネントを以下に説明します。

- 型 `xEntityClassType` の **EC**。このコンポーネントは、ノードがどの種類の「オブジェクト」を表しているかを知るために使用されます。`xEntityClassType` はすべてのエンティティ クラスの要素を含んでいる列挙型です。
- `xIdNode` 型の **First**、**Suc**、および **Parent**。これらのコンポーネントはシンボル テーブル ツリーのビルドに使用されます。`First` は現在のノードの最初の子を参照します。`Suc` は次の弟を参照し、`Parent` は親（father）ノードを参照します。アプリケーションでは `Parent` だけが必要です。
- 型 `xNameType` の **Name**。これは `char *` として定義されます。このコンポーネントは、現在の「オブジェクト」の名前を文字列として表すために使用されます。モデル ベリファイヤ（Model Verifier）では、このコンポーネントはアプリケーションには必要ありません。

エンティティ クラス特有のコンポーネント

次に、テーブルには、表現されるエンティティ クラスによって決まるコンポーネントがあります。このセクションには、もう一方の `xECIdStruct` に含まれる共通しない要素を説明します。

パッケージ コンポーネント

- 型 `xNameType` の `ModuleName`。パッケージが ASN.1 から生成された場合、このコンポーネントは ASN.1 モジュールの名前を `char *` として保持します。

システム（ルート アクティブ クラス）コンポーネント

- 型 `xIdNode *` の `Content`。このコンポーネントには、システム レベル（モデル内の「ルート」アクティブ クラス）の全コネクタのリストが含まれています。
- `xPrdIdNode *` 型の `VirtPrdList`。これは、このシステム インスタンスのアクティブ クラスの全仮想操作のリストです。
- `xSystemIdNode` 型の `Super`。これは継承したシステム型への参照です。システムでは、このコンポーネントはヌルです。システム インスタンス内では、このコンポーネントは初期化されたシステム型への参照です。
- `int` 型の `Trace_Default` 。このコンポーネントには、システムに対して定義される現在のトレース値が含まれます。

- int 型の GRTrace。このコンポーネントには、システムに対して定義される現在のグラフィカル トレース値が含まれます。
- int 型の MSCETrace。このコンポーネントには、システムに対して定義される現在のシーケンス図トレース値が含まれます。

コネクタとポート コンポーネント

コネクタとポートについては、コネクタまたはポートの 2 方向を表す 2 つの連続した xChannelIdNodes が必ずシンボル テーブル内に存在します。これらのコンポーネントを以下に説明します。

- xIdNode * 型の SignalSet。このコンポーネントは、現在の方向のコネクタのシグナル セットを表します（単方向コネクタについては反対方向のシグナル セットは空です）。

  SignalSet は、シグナル セット内のシグナルを表す xSignalIdNodes を参照するコンポーネントを持つ配列です。配列の最後のコンポーネントは必ず NULL ポインタ（値 (xSignalIdNode)0 です）。
- xIdNode * 型の ToId。これは xIdNodes の配列で、この配列の各コンポーネントが、コネクタまたはポートの接続先の「オブジェクト」（シグナルの転送先のオブジェクト）を表すシンボル テーブルへのポインタとなります。

  ToId で参照できるオブジェクトはコネクタ、ポートおよびアクティブ クラスです。配列の最後のコンポーネントは必ず NULL ポインタ（値 (xIdNode)0）です。
- xChannelIdNode 型の Reverse。これは、同じコネクタまたはポートのもう一方の方向を表すシンボル テーブル ノードへの参照です。

パートとパートを持つアクティブ クラス

- xBlockIdNode 型の Super。パートでは、このコンポーネントはヌルです。パートを持つアクティブ クラスでは、このコンポーネントは、このコンポーネントの継承元のパートへの参照となります（継承がない場合はヌル）。アクティブ クラスのインスタンスでは、これはインスタンス化されるアクティブ クラスへの参照となります。
- xIdNode * 型の Contents。パートを持つアクティブ クラスのインスタンス化では、これらのコンポーネントには以下のリストが含まれています。
  - パート内のアクティブ クラスのインスタンス化
  - パート内のコネクタ
  - パートからの発信ポート
  - パート内のインライン アクティブ クラス
  - パート内のアクティブ クラスのインスタンス化で定義されるポート
- xPrdIdNode * 型の VirtPrdList。これは、アクティブ クラスのインスタンス内の全仮想操作のリストです。

- int 型の NumberOfInst。これはインスタンス セット内のインスタンス数です。したがって、このコンポーネントは、パートを持つアクティブ クラスのインスタンスの場合だけ関係します。
- int 型の Trace_Default 。このコンポーネントには、パートについて定義された実行トレースの現在値が入ります。
- int 型の GRTrace。このコンポーネントには、パートについて定義されたグラフィカル トレースの現在値が入ります。
- int 型の MSCETrace。このコンポーネントには、システムについて定義された現在のシーケンス図トレース値が含まれます。
- int 型の GlobalInstanceId。このコンポーネントは、シーケンス図トレースの実行時に必要な一意の ID を格納するために使用されます。

アクティブ クラス、インライン アクティブ クラス、インスタンス コンポーネント

- xStateIdNode * 型の StateList。これは、この（インライン）アクティブ クラスの xStateIdNodes への参照のリストです。アクティブ クラスの実行中のインスタンスのステート値を使用して、このリストを使って対応する xStateIdNode を見つけることができます。
- xIdNode * 型の SignalSet。これはアクティブ クラスの有効な入力シグナル セットを表します。

  SignalSet は、シグナル セットのパートとなるシグナルとタイマーを表す xSignalIdNodes を参照するコンポーネントを持つ配列です。配列の最後のコンポーネントは必ず NULL ポインタ（値 (xSignalIdNode)0）です。
- xIdNode * 型の **ToId**。これは xIdNode の配列です。これの各配列コンポーネントは、アクティブ クラスのこのインスタンスの接続先となるインスタンス、すなわち、シグナルの転送先となる「オブジェクト」を表す IdNode へのポインタです。

  ToId で参照できる「オブジェクト」はコネクタ、ポートおよびアクティブ クラスのインスタンスです。配列の最後のコンポーネントは必ず NULL ポインタ（値 (xIdNode)0）です。
- int 型の MaxNoOfInst。これは、現在のアクティブ クラスのための仕様に基づいて存在できるアクティブ クラスの並行インスタンスの最大数を表します。アクティブ クラスの無数の並行インスタンスは -1 によって表します。
- int 型の NextNo。これは、このインスタンス セットから作成される次のインスタンスに割り当てられるインスタンス番号です。
- int 型の NoOfStaticInst。このコンポーネントには、アクティブ クラスのインスタンス セットの開始時に存在する静的インスタンスの数が格納されます。
- xPrsNode * 型の ActivePrsList これは、現在のアクティブ クラスのアクティブ インスタンスの単一リンク リスト内の先頭位置を指すポインタのアドレスです。

  このリストは、アクティブ クラスのインスタンスを表すために使用される xPrsRec 構造体の NextPrs コンポーネントを使用して継続されます。リスト内の順序は、最初に作られたアクティブ インスタンスが最後に、最後に作られたアクティブ インスタンスが最初になります。

- `xptrint` 型の `VarSize`。アクティブ クラス（構造体 : `yVDef_ProcessName`）を表すために使用されるデータ領域のバイト サイズ。
- `Prio`。シグナルの優先度です。
- `xPrsNode` 型の **AvailPrsList**。これは、停止したアクティブ クラスのインスタンス用の有効リスト ポインタへのアドレスです。このデータ領域は、このクラスまたはそのインスタンス化に対する次の Create アクションで後で再利用されます。
- `int` 型の `Trace_Default`。このコンポーネントには、アクティブ クラスについて定義されたトレースの現在値が格納されます。
- `int` 型の `GRTrace`。このコンポーネントには、アクティブ クラスについて定義されたグラフィカル トレースの現在値が格納されます。
- `GRrefFunc`。シンボル番号（状態機械内のシンボルに割り当てられる番号）が与えられている場合、これはそのシンボルへのグラフィカル参照を格納した文字列を返します。
- `int` 型の `MaxSymbolNumber`。このコンポーネントは、現在のアクティブ クラスに含まれるシンボルの数です。
- `int` 型の `SignalSetLength`。このコンポーネントは、現在のアクティブ クラスのシグナル セットに含まれるシグナルの数です。
- `int` 型の `MSCETrace`。このコンポーネントには、アクティブ クラスに対して定義される現在のシーケンス図トレース値が含まれます。
- `long int *` 型の `CoverageArray`。このコンポーネントは、アクティブ クラス内の全シンボルの配列として使用されます。シンボルが実行されるたびに、対応する配列コンポーネントが 1 つずつ増加します。
- `long int` 型の `NoOfStartTransitions`。このコンポーネントは、現在のアクティブ クラスの開始遷移が実行される回数をカウントするために使用されます。この情報はカバレッジ テーブルに表示されます。
- `long int` 型の `MaxQueueLength`。このコンポーネントは、現在のアクティブ クラスのインスタンスの最大入力ポート長を登録するために使用されます。この情報はカバレッジ テーブルに表示されます。
- `PAD_Function`。これは関数へのポインタです。このポインタは、現在のアクティブ クラスの `yPAD_ProcessName` 関数を参照します。この関数は、この型のアクティブ クラスのインスタンスが遷移を実行するときに呼び出されます。`PAD_Functions` は、振る舞いを実装する状態機械で定義されるアクションを含むときに、生成されたコードの一部となります。
- `Free_Vars`。これは関数へのポインタです。このポインタは、現在のアクティブ クラスの `yFree_ProcessName` 関数を参照します。この関数は、アクティブ クラスのインスタンスが stop アクションを実行して、アクティブ クラス内のローカル属性が使用するメモリを解放するときに使用されます。
- `xPrsIdNode` 型の `Super`。インライン アクティブ クラスでは、このコンポーネントはヌルです。アクティブ クラスでは、このコンポーネントは、このコンポーネントの継承元のアクティブ クラスへの参照となります（継承がない場合はヌル）。
- `xPrdIdNode *` 型の `VirtPrdList`。これは、アクティブ クラスのインスタンス化に含まれる全仮想操作のリストです。

- `xBlockIdNode` 型の `InBlockInst`。このコンポーネントは、このアクティブ クラスを含むパート（もしあれば）への参照となります。
- `char *` 型の `RefToDefinition`。これはアクティブ クラスへのグラフィカル参照です。

操作、複合文コンポーネント

属性宣言を含む複合文宣言は操作として扱われます。ただし、たとえば、かかるオブジェクトにはステートを含むことはできません。

- `xStateIdNode *` 型の `StateList`。これは、このインライン アクティブ クラスの `xStateIdNodes` への参照のリストです。アクティブ クラスの実行中のインスタンスのステート値を使用して、このリストを使って対応する `xStateIdNode` を見つけることができます。
- `xIdNode *` 型の `SignalSet`。これは、インライン アクティブ クラスまたはアクティブ クラスの有効な入力シグナル セットを表します。

  `SignalSet` は、シグナル セットのパートとなるシグナルとタイマーを表す `xSignalIdNodes` を参照するコンポーネントを持つ配列です。配列の最後のコンポーネントは必ず値 `(xSignalIdNode)0` のヌル ポインタです。
- `Assoc_Function`。これは関数へのポインタです。このポインタは、現行のプロシージャの `yProcedureName` 関数を参照します。この関数はプロシージャの呼び出し時に呼び出され、該当するアクションを実行します。`yProcedureName` 関数は、プロシージャ グラフ内に定義されたアクションを含むとき、生成されたコードの一部となります。
- `Free_Vars`。これは関数へのポインタです。このポインタは、現行のプロシージャの `yFree_ProcedureName` 関数を参照します。この関数は、プロシージャが return アクションを実行して、プロシージャ内のローカル属性が使用するメモリを解放するときに使用されます。
- `xptrint` 型の `VarSize`。これは、プロシージャ（構造体 `yVDef_ProcedureName`）を表すために使用されるデータ領域のバイト サイズ。
- `xPrdNode *` 型の `AvailPrdList` これは、プロシージャ インスタンスを表すために使用されるデータ領域用の有効リスト ポインタのアドレスです。return アクションでデータ領域は有効リストに置かれ、このプロシージャ型の次の呼び出しで再利用できます。
- `GRrefFunc`。これは、シンボル番号（プロシージャ シンボルに割り当てられる番号）が与えられた場合、そのシンボルへのグラフィカル参照を格納した文字列を返す関数へのポインタとなります。
- `int` 型の `MaxSymbolNumber`。このコンポーネントは、現行のプロシージャに含まれるシンボルの数です。
- `int` 型の `SignalSetLength`。このコンポーネントは、現行のプロシージャのシグナル セットに含まれるシグナルの数です。
- `long int` 型の `CoverageArray`。このコンポーネントは、プロシージャ内の全シンボルの配列として使用されます。シンボルが実行されるたびに、対応する配列コンポーネントが 1 つずつ増加します。

- `xPrdIdNode` 型の `Super`。このコンポーネントは、このプロシージャの継承元のプロシージャへの参照となります（継承がない場合はヌル）。
- `xPrdIdNode *` 型の `VirtPrdList`。これは、アクティブ クラス内の全仮想操作のリストです。

リモート操作コンポーネント

- `xRemotePrdListNode` 型の `RemoteList`。このコンポーネントは、この操作を「エクスポート」するすべてのアクティブ クラスのリストの先頭になります。このリストは `xRemotePrdListStructs` のリンク リストです。この各ノードには「エクスポート」するアクティブ クラスへの参照が格納されています。

シグナル、タイマー、起動シグナル、**RPC** シグナルのコンポーネント

- `xptrint` 型の `VarSize`。これは、シグナル（構造体：`ySignalPar_SignalName`）を表すために使用されるデータ領域のバイト サイズです。
- `xSignalNode *` 型の `AvailSignalList`。これは、このシグナル型のシグナル インスタンスのための有効リスト ポインタへのアドレスです。
- `Equal_Timer`。これは関数へのポインタです。このポインタは、このノードを使用してパラメータを持つタイマーを表すときに、関数を参照するだけです。

  この場合、参照される関数を使用して、2 つのタイマーのパラメータが等しいか等しくないかを調べることができます。これは reset アクションで必要です。`Equal_Timer` 関数は生成されたコードの一部になります。これらの関数は、いずれも `sctsdl.c` で定義される関数 `xRemoveTimer` と `xRemoveTimerSignal` から呼び出されます。
- `Free_Signal`。この関数はシグナル参照を使用して、このシグナル パラメータから参照される動的データを利用可能なメモリのプールに戻します。
- `char *` 型の `RefToDefinition`。これはシグナルまたはタイマーの定義への参照です。
- `Prio`。シグナルの優先度です。

ステート コンポーネント

- `int` 型の `StateNumber`。このステートを表すために使用される `int` 値です。

- `xInputAction *` 型の `SignalHandlArray`。このコンポーネントは `xInputAction` の配列を参照します。`xInputAction` は、`xDiscard`、`xInput`、`xSave`、`xEnablCond`、`xPrioInput` の値を持つ列挙型です。

  この配列には、このステートが格納されている状態機械を表すノード内の `SignalSet` 配列と同じコンポーネント数が入ります。`SignalHandlArray` 内の各位置は、状態機械内の `SignalSet` 配列内の対応する位置にあるシグナルがこのステート内で処理される方法を表します。

  `SignalHandlArray` 内の最後のコンポーネントは `xDiscard` に一致します。これは、`SignalSet` 内の最後の値 `0` に対応しています。

  `SignalHandlArray` の所定のインデックスに `xInput`、`xSave`、または `xDiscard` の値が入っている場合、シグナルを処理する方法は明らかです。しかし、`SignalHandlArray` に値 `xEnablCond` が入っている場合、ガード式を計算して、シグナルがシグナル受信となるか、またはシグナルを保存しなければならないかを知る必要があります。
  この計算は以下に説明する `EnablCond_Function` の目的です。
- `int *` 型の `InputRef`。このコンポーネントは配列です。`SignalHandlArray` の特定のインデックスに `xInput`、`xPrioInput`、または `xEnablCond` が入っている場合、この `InputRef` にはグラフ内の対応するシグナル受信シンボルのシンボル番号が入ります。
- `EnablCond_Function` は `xInputAction` を返す関数です。ステートにガードが含まれている場合、このポインタは関数を参照します。そうでない場合、このポインタは `0` を参照します。`EnablCond_Function` は `xSignalIdNode`（シグナルを参照）と、アクティブ クラスのインスタンスを参照し、所定のインスタンスの現在の状態での現在のシグナルの受信のためのガードを計算します。

  この関数は `xInput` または `xSave` のどちらかを返します。
  `EnablCond_Functions` はガード式を含むとき、生成されたコードの一部になります。これらの関数は、ファイル `sctsdl.c` 内の関数 `xFindInputAction` から呼び出されます。`xFindInputAction` は関数 `SDL_Output` と `SDL_Nextstate` から使用されます。
- `ContSig_Function` は `int` を返す関数です。トリガされる遷移上のガードがステートに含まれる場合、このポインタは関数を参照します。そうでない場合、このポインタは `0` を参照します。
- `int` 型の `StateProperties`。このコンポーネントでは、アクティブ クラスのインスタンスがアクションを実行しない場合でも、最下位 3 ビットを使用して、ステート内のガードまたはトリガされる遷移上のガードの式に値を変更することがあるオブジェクトへの参照が含まれているどうかを示します。
- `long int` 型の `CoverageArray`。このコンポーネントは、アクティブ クラスのシグナル セット（+1）全体の配列として使用されます。入力操作が実行されるたびに、対応する配列コンポーネントが 1 つずつ増加します。シグナル セットの長さに等しいインデックスにある最後のコンポーネントは、ステート内で受信する、トリガされる遷移上のガードの数を記録するために使用されます。このコンポーネントに格納される情報はカバレッジ テーブルに与えられます。
- `xPrdIdNode` 型の `Super`。このコンポーネントは、このプロシージャの継承元のプロシージャへの参照となります（継承がない場合はヌル）。

- `char *` 型の `RefToDefinition`。これはステートの定義への参照を保有します（このステートが定義されるシンボルの 1 つ）。

ソートとシンタイプ コンポーネント

- `Free_Function`。この関数ポインタは、動的メモリを使って表示される型（Charstring、OctetString、String、Bag など）については非ゼロです。`Free_Functions` は動的メモリを動的メモリのプールに返すために使用されます。
- `Test_Function` は `xbool` を返す関数です。これは範囲条件を含むすべての型について非ゼロです。この関数ポインタはモデル ベリファイヤ（Model Verifier）によって使用され、値を属性に割り当てるときに値の妥当性を検査します。
- `xptrint` 型の `SortSize`。このコンポーネントは現在のソートの属性のバイトサイズを表します。

- xTypeOfSort 型の SortType。このコンポーネントはソート型を示します。. 取り得る値は、xPredef、xUserdef、xEnum、xStruct、xArray、xGArray、xCArray、xRef、xString、xPowerSet、xBag、xGPowerSet、xInherits、xSyntype、xUnion、および xChoice です。また、SortType の値に基づいて、以下のコンポーネントを持つことになります。

  SortType が xArray、xGArray、または xCArray の場合、
  - xSortIdNode 型の CompOrFatherSort。これは、コンポーネント ソートを表す SortIdNode へのポインタです。
  - xSortIdNode 型の IndexSort。これは、インデックス ソートを表す SortIdNode へのポインタです。xCArray では、インデックス ソートは必ず整数です。
  - xGArray では、LowestValue が xxx_ystruct 内の Data のオフセットとして使用されます。xArray と xCArray では、この値は 0 です。
  - xGArray では、HighestValue が xxx_ystruct のサイズとして使用されます。xArray では、この値は 0 です。xCArray では、これは最大のインデックス、すなわち、Length - 1 です。
  - xGArray では、yrecIndexOffset が xxx_ystruct 内の Index のオフセットとして使用されます。xArray と xCArray では、この値は 0 です。
  - xGArray では、yrecDataOffset が型（デフォルト値を表す値）内の Data のオフセットとして使用されます。xArray と xCArray では、この値は 0 です。

  SortType が xString、xGPowerSet または xBag の場合、
  - xSortIdNode 型の CompOrFatherSort。これは、コンポーネント ソートを表す SortIdNode へのポインタです。
  - LowestValue が xxx_ystruct 内の Data のオフセットとして使用されます。
  - HighestValue は xxx_ystruct のサイズとして使用されます。

  SortType は xPowerSet、xRef、xOwn、xORef です。
  - xSortIdNode 型の CompOrFatherSort。これは、コンポーネント ソートを表す SortIdNode へのポインタです。

  SortType が xInherits の場合、
  - xSortIdNode 型の CompOrFatherSort。これは、継承されたソートを表す SortIdNode へのポインタです。

  SortType が xSyntype の場合、
  - xSortIdNode 型の CompOrFatherSort。これは、親ソート（シンタイプのシンタイプである場合でも、シンタイプの元となるニュータイプ）を表す SortIdNode へのポインタです

- `xSortIdNode` 型の `IndexSort`。これは、親ソート（シンタイプの元となるニュータイプまたはシンタイプ）を表す `SortIdNode` へのポインタです。
- `long int` 型の `LowestValue`。シンタイプを配列（C 配列に変換）のインデックスとして使用できる場合、その値はシンタイプ範囲の最下位の値になります。それ以外の場合は 0 です。
- `long int` 型の `HighestValue`。シンタイプを配列（C 配列に変換）のインデックスとして使用できる場合、その値はシンタイプ範囲の最上位の値になります。それ以外の場合は 0 です。`LowestValue` と `HighestValue` は、モデル ベリファイヤ（Model Verifier）がこの型の配列をインデックス型として扱うときに使用します。

属性、仮パラメータ、シグナル パラメータ、構造体コンポーネント

- `xSortIdNode` 型の `SortNode`。これは、この属性またはパラメータを表す `SortIdNode` へのポインタです。
- `xptrint` 型の `Offset`。このコンポーネントは、アクティブ クラスまたは操作の属性、シグナル パラメータ、または構造体を表す構造体内のオフセットをバイトの単位で表現します。これは構造体内のこのコンポーネントの相対的な位置を示します。
- `xptrint` 型の `Offset2`。状態機械の仮パラメータの場合、このコンポーネントは `StartUpSignal` 内の仮パラメータのオフセットをバイトの単位で表します。アクティブ クラス内のグローバル属性の場合、このコンポーネントはこの属性の「エクスポート」された値のオフセットをバイトの単位で表します。
- `int` 型の `IsAddress`。このコンポーネントは操作の仮パラメータのために使用され、仮パラメータが IN パラメータか、IN/OUT パラメータか、または結果の属性かを示すために使用されます。

インターフェイス内の属性

- `xptrint` 型の `SortSize`。このコンポーネントはグローバル属性型のサイズを示します。
- `xRemoteVarListNode` 型の `RemoteList`。このコンポーネントは、このグローバル属性を「エクスポート」するすべてのアクティブ クラスのリストの先頭になります。このリストは `xRemoteVarListStruct` のリンク リストです。これの各ノードには、「エクスポートする」アクティブ クラスへの参照と、「エクスポートされた」値が入った `Offset` が格納されています。

型情報ノード（**Type Info Nodes**）

概要

型情報ノードは、操作の一般的なデータ型への実装を行うジェネリック関数によっておもにランタイムに使用されるデータ構造です。

型情報ノードのために使用される C 型 tSDLTypeInfo には、xSortIdNode 型と同じ情報が格納されています。xSortIdNode は、SDL ソートのみに使用される一般的な xIdNode の特殊ケースです。

型情報ノードは、xSortIdNode を代わりに使用できる生成されたコード内のほとんどの場所で使用されます。

注記

xSortIdNode 型はいくつかの状況では生成された C コードによってまだ使用されるので、依然として下位互換性が維持されています。この型は廃止となる候補に上がっているので、代わりに、tSDLTypeInfo 型を使用してください。

参照

シンボル テーブルでの xIdNode 型の定義

型情報ノードの型定義

型情報ノードを記述する型定義は sctpred.h ファイルに用意されています。

各型情報ノードは以下のものから構成された構造体です。

- すべての型情報ノードで利用できる型情報ノード内の一般的なコンポーネント
- 各型に特有の型特有の型情報ノードのコンポーネント

型情報ノードの最適化

型情報ノードは、ランタイム時にデータ型のプロパティを記述するデータ構造体です。この情報は、すべてのデータ型について割り当てを実行できる関数などのような、一般的な演算子の実装のために必要です。

変換されたシステム内のデータ型に使用される操作によっては、一部の型情報ノードが必要ないことがあります。ここで説明する最適化の目的はかかる型情報ノードを取り除くことにあります。

この第一歩が、下記の定義済みの整数型の例にしたがって、すべての型情報ノードを #ifdef 構造体で囲むことです。

```
#ifndef XTNOUSE_Integer
tSDLTypeInfo ySDL_SDL_Integer = {
  ....
  };
#endif
```

これは、整数についての型情報ノードは、XTNOUSE_Integer を定義することによって取り除けることを意味しています。

#ifndef 文内の定義済みのデータ型の名前を以下に記述します。

```
#ifndef XTNOUSE_Integer
#ifndef XTNOUSE_Real
#ifndef XTNOUSE_Natural
#ifndef XTNOUSE_Boolean
#ifndef XTNOUSE_Character
#ifndef XTNOUSE_Time
```

```
#ifndef XTNOUSE_Duration
#ifndef XTNOUSE_Pid
#ifndef XTNOUSE_Charstring
#ifndef XTNOUSE_Bit
#ifndef XTNOUSE_Bit_string
#ifndef XTNOUSE_Octet
#ifndef XTNOUSE_Octet_string
#ifndef XTNOUSE_IA5String
#ifndef XTNOUSE_NumericString
#ifndef XTNOUSE_PrintableString
#ifndef XTNOUSE_VisibleString
#ifndef XTNOUSE_NULL
#ifndef XTNOUSE_Object_identifier
```

ユーザー定義済みの型の場合、名前の選択は次のアルゴリズムに基づいて行われます。

1. 型名が一意であれば（C と同様に、大文字小文字は区別されます）、すなわち、この名前を持つシステム内のデータ型が 1 つだけであれば、`#ifndef` 内の名前は次のようになります。

   ```
   XTNOUSE_typename
   ```

2. 型名は一意ではないが、型名に型定義のスコープの名前を付加したものが一意の場合、`#ifndef` 内の名前は次のようになります。

   ```
   XTNOUSE_typename_scopename
   ```

3. ほかの場合には、`#ifndef` 内の名前は次のようになります。

   ```
   XTNOUSE_typename-with-prefix-or-suffix
   ```

アルゴリズムのこの部分だけを使用して、使用されていない型情報ノードを削除する適切な define 文を持つヘッダー（`.h`）ファイルを手で書くことはもちろん可能です。コンパイラまたはリンカーを使用すれば、未使用のデータを探すことが容易になります。

ただし、コード ジェネレータは型情報ノードの使用を予測します。この情報は次のファイルに格納されます。

```
auto_cfg.h
```

このファイルの最後のセクションに、型情報ノードを使用するための define 文が格納されます。以下に一例を示します。

例 **372:** 型情報ノード使用のための **define** 文

```
#ifdef XUSE_TYPEINFONODE_CFG
/* Type info node configuration */
#define XTNOUSE_Boolean
#define XTNOUSE_Character
#define XTNOUSE_Charstring
/* NOT #define XTNOUSE_Integer*/
....
....
/* NOT #define XTNOUSE_s*/
#endif
```

システム内のすべてのデータ型ごとに、型情報ノードを使用するかしないかを示す 1 行が存在します。また、コンパイル時に XUSE_TYPEINFONODE_CFG の定義が必要ですが、そうしなければ、型情報ノードの最適化の計算は自動的には行われません。

ファイル auto_cfg.h が自動的にインクルードされ、AgileC コードジェネレータによって使用されます。C コード ジェネレータでは、以下のコードが scttypes.h に含まれています。

```
#ifdef USER_CONFIG
#include "uml_cfg.h"
#else /* auto_cfg.h is always included because it
         contains information about the usage
        of 'long long' types */
#include "auto_cfg.h"
#endif
```

auto_cfg.h の構成設定の大半は、AUTOMATIC_CONFIG 定義で制御されます。auto_cfg.h の構造は以下のとおりです。

```
/* Program generated by <CCG name and version> */
/* CCG id define, for example: */
#define XSCT_CADVANCED

/* "auto_cfg.h" file generated for system <system name> */

#ifndef XUSE_GENERIC_FUNC
#define XUSE_GENERIC_FUNC
#endif

/* automatic defined for long long type, for example */

#ifndef XNOUSE_LONG_LONG
#define XNOUSE_LONG_LONG
#endif
#ifndef XUSE_TYPEINFONODE_CFG
#define XTNOUSE_long_long_int
#define XTNOUSE_unsigned_long_long_int
#endif

#ifdef AUTOMATIC_CONFIG

/* all other configuration settings: */

#ifndef XNOUSE_AUTOMATIC_OPERATOR_CFG
/* Predefined operator configuration */
#endif

#ifdef XUSE_TYPEINFONODE_CFG
/* Type info node configuration */
#endif

#endif /*AUTOMATIC_CONFIG*/
```

したがって、USER_CONFIG を定義して auto_cfg.h を uml_cfg.h にインクルードするか、または AUTOMATIC_CONFIG を定義すれば、最適化が行われます。

AUTOMATIC_CONFIG は、デフォルトで定義済み演算子の最適化を使用します。これをオフにするには、XNOUSE_AUTOMATIC_OPERATOR_CFG を定義します。

型情報ノードの最適化は、AUTOMATIC_CONFIG XUSE_TYPEINFONODE_CFG が定義されている場合のみ適用されます。

ときには、使用した型情報ノードに対する自動計算が機能しないことがあります。もっとも明らかな場合が、インライン C コード内での使用です。コード ジェネレータはかかるコードの解析をしないので、そのような使い方をまったく認識できる状態にありません。これらの状態に対処するために、ユーザーは特定の型情報ノードが使用されることと、その結果、特定のノードが依存するノードがあることを、コード ジェネレータに指示できます。もちろん、型情報ノード構成の後に #define と #undef を挿入することで、これらの状態をユーザーが手動で取り扱うことは可能です。しかし、型情報ノード間のすべての従属関係により、これが困難であるかもしれません。

ユーザーは、特定の型情報ノードが使われていることをファイルに明記することによって、コード ジェネレータに指示できます。コード ジェネレータは、以下の手順にしたがって、かかるファイルを探します。

環境変数 TAU_TYPEINFOCFG が定義されている場合、この変数の値はファイル名（パスを含む）として扱われ、そのファイルが読まれます。コード ジェネレータは、このファイルを開くことができない場合は、エラーを発行します。

その環境変数が定義されていない場合は、「typeinfo.cfg」という名前のファイルを検索します。コード ジェネレータは、中間ファイル（.pr）が作成されるディレクトリを調べます。「typeinfo.cfg」というファイルが見つかれば、そのファイルを読みます。該当するファイルが見つからない場合は、コード ジェネレータは、ユーザーが型情報構成ファイルを保有していないと見なします。

型情報構成ファイルの内容は以下の規則に従うものとします。

- 使用するものとして登録される各型をそれぞれの行に記載します。この場合、型名で始まります。
- 同じ名前を持つ複数のデータ型がシステムに存在するとき、型名の後にデータが定義されているスコープの名前を付けることができます。
- 型名とスコープ名は 1 つ以上のスペースまたはタブで区切ります。

例 **373:** 型名とスコープ名

```
typename1
typename2 scopename2
typename3
```

コード ジェネレータは、上記の基準（名前または名前とスコープ）に一致するデータ型を検索します。この検索は大文字と小文字を区別します。次に、以下の規則が適用されます。

- 基準（型名または型名とスコープ名）に一致するすべてのデータ型に含まれる型情報ノードが使用されるものとして登録されます。
- また、登録したノードが依存するすべての型情報ノードも使用されるものとして登録されます。
- 基準に一致するデータ型が存在しない場合は、エラー メッセージが提示されます。

大文字小文字を区別して検索を行う場合、以下の表にしたがって定義済みの型を与えるものとします。

```
Integer
Real
Natural
Boolean
Character
Time
Duration
Pid
Charstring
Bit
Bit_string
Octet
Octet_string
IA5String
NumericString
PrintableString
VisibleString
NULL
Object_identifier
```

これらの型のスコープ名は **Predefined** です。

型情報ノード内の一般的なコンポーネント

注記

このセクションで説明されるコンポーネントは、コンポーネントが表す型には依存せず、**すべての**型情報ノードに対して使用可能であり、型特有のコンポーネントに対して繰り返されることはありません。

```
/* --- General type information for types --- */

typedef T_CONST struct tSDLTypeInfoS {
  tSDLTypeClass   TypeClass;
  unsigned char   OpNeeds;
  xptrint         SortSize;
  struct tSDLFuncInfo *OpFuncs;
  T_SDL_EXTRA_COMP
#ifdef T_SDL_NAMES
  char           *Name;
#endif
#ifdef XREADANDWRITEF
  xIdNode         FatherScope;
  xSortIdNode     SortIdNode;
#endif
} tSDLTypeInfo;
```

- `TypeClass`: このコンポーネントは情報ノードがどの型を記述するかを定義します。利用可能な型とそれに対応する値を示すリストを以下の列挙型の定義に見つけることができます。

```
typedef enum
{
  /* standard types*/
  type_SDL_Integer=128,
  type_SDL_Real=129,
```

```
    type_SDL_Natural=130,
    type_SDL_Boolean=131,
    type_SDL_Character=132,
    type_SDL_Time=133,
    type_SDL_Duration=134,
    type_SDL_Pid=135,
    type_SDL_Charstring=136,
    type_SDL_Bit=137,
    type_SDL_Bit_string=138,
    type_SDL_Octet=139,
    type_SDL_Octet_string=140,
    type_SDL_IA5String=141,
    type_SDL_NumericString=142,
    type_SDL_PrintableString=143,
    type_SDL_VisibleString=144,
    type_SDL_NULL=145,
    type_SDL_Object_identifier=146,

    /* standard ctypes */
    type_SDL_ShortInt=150,
    type_SDL_LongInt=151,
    type_SDL_UnsignedShortInt=152,
    type_SDL_UnsignedInt=153,
    type_SDL_UnsignedLongInt=154,
    type_SDL_Float=155,
    type_SDL_Charstar=156,
    type_SDL_Voidstar=157,
    type_SDL_Voidstarstar=158,

    /* user defined types */
    type_SDL_Syntype=170,
    type_SDL_Inherits=171,
    type_SDL_Enum=172,
    type_SDL_Struct=173,
    type_SDL_Union=174, /* Not used */
    type_SDL_UnionC=175,
    type_SDL_Choice=176,
    type_SDL_ChoicePresent=177,
    type_SDL_Powerset=178,
    type_SDL_GPowerset=179,
    type_SDL_Bag=180,
    type_SDL_String=181,
    type_SDL_LString=182,
    type_SDL_Array=183,
    type_SDL_Carray=184,
    type_SDL_GArray=185,
    type_SDL_Own=186,
    type_SDL_Oref=187,
    type_SDL_Ref=188,
    type_SDL_Userdef=189,
    type_SDL_EmptyType=190,

    /* signals */
    type_SDL_Signal=200,
    type_SDL_SignalId=201

} tSDLTypeClass;
```

- `OpNeeds` : このコンポーネントには、割り当て、一致試験、フリー関数、および初期化に関する型のプロパティを与える 4 ビットが含まれています。
  - 最初のビットは、型が自動的に解放される必要があるポインタかどうかを指示します。最初のビットが設定されていれば、この型の値の内部にある解放すべきメモリを探す必要があります。
  - 2 番目のビットは、この型の 2 つの値が等しいか等しくないかを試験するために `memcmp` を使用できるかどうかを指示します。このビットが設定されていれば、特別な比較関数を与える必要があります。
  - 3 番目のビットは、この型の代入の実行に `memcpy` を使用できるかどうかを指示します。このビットが設定されていれば、特別な代入関数を与える必要があります。
  - 4 番目のビットは、この型を 0 以外に初期化する必要があるかどうかを指示します。

  以下のマクロを使用してこれらのプロパティを試験できます。

```
#define NEEDSFREE(P) ¥
  (((tSDLTypeInfo *)(P))->OpNeeds & (unsigned char)1)
#define NEEDSEQUAL(P) ¥
  (((tSDLTypeInfo *)(P))->OpNeeds & (unsigned char)2)
#define NEEDSASSIGN(P) ¥
  (((tSDLTypeInfo *)(P))->OpNeeds & (unsigned char)4)
#define NEEDSINIT(P) ¥
  (((tSDLTypeInfo *)(P))->OpNeeds & (unsigned char)8)
```

- `SortSize` : このコンポーネントは型のサイズを定義します。
- `OpFuncs` : これは固有の assign、equal、free、read、および write 関数への参照を格納した構造体へのポインタです。このコンポーネントは特別な場合に使用されます。assign、equal、free、read、または write 関数の実装に #ADT ディレクティブが使用された場合、これに関する情報は `OpFuncs` フィールドに格納されます。`OpFuncs` フィールドのデフォルト値は 0 ですが、これらの関数のどれでも用意していた場合、このフィールドは `tSDLFuncInfo` 構造体へのポインタとなります。この構造体は今度は用意されていた関数を参照します。

```
typedef struct tSDLFuncInfo {
  void *(*AssFunc) (void *, void *, int);
  SDL_Boolean (*EqFunc) (void *, void *);
  void (*FreeFunc) (void **);
#ifdef XREADANDWRITEF
  char *(*WriteFunc) (void *);
  int (*ReadFunc) (void *);
#endif
} tSDLFuncInfo;
```

- `Name` : これは文字列リテラルとしての型の名前です。
- `FatherScope` : これは、型が定義されるスコープのための `IdNode` へのポインタです。
- `SortIdNode` : これは、同じ型を記述する `xSortIdNode` へのポインタです。

型特有の型情報ノードのコンポーネント

次のセクションでは、型情報ノードを定義するコンポーネントを記載します。型特有のコンポーネントについてのみ説明します。一般的なコンポーネントについては前のセクションで取り上げて説明しています。

列挙型の型情報ノード コンポーネント

```
typedef T_CONST struct {
  int             LiteralValue;
  char           *LiteralName;
} tSDLEnumLiteralInfo;

typedef T_CONST struct tSDLEnumInfoS {
  tSDLTypeClass   TypeClass;
  unsigned char   OpNeeds;
  xptrint         SortSize;
  struct tSDLFuncInfo *OpFuncs;
  T_SDL_EXTRA_COMP
#ifdef T_SDL_NAMES
  char           *Name;
#endif
#ifdef XREADANDWRITEF
  xIdNode         FatherScope;
  xSortIdNode     SortIdNode;
#endif
#ifdef XREADANDWRITEF
  int             NoOfLiterals;
  tSDLEnumLiteralInfo *LiteralList;
#endif
} tSDLEnumInfo;
```

- `NoOfLiterals`：これは列挙型のリテラル数です。
- `LiteralList`：これは `tSDLEnumLiteralInfo` 要素の配列へのポインタです。このリストには、列挙値と文字列としてのリテラル名との間の変換テーブルが実装されています。

シンタイプ、継承を持つ型、および **Own**、**Oref** インスタンス化

```
typedef T_CONST struct tSDLGenInfoS {
  tSDLTypeClass   TypeClass;
  unsigned char   OpNeeds;
  xptrint         SortSize;
  struct tSDLFuncInfo *OpFuncs;
  T_SDL_EXTRA_COMP
#ifdef T_SDL_NAMES
  char           *Name;
#endif
#ifdef XREADANDWRITEF
  xIdNode         FatherScope;
  xSortIdNode     SortIdNode;
#endif
  tSDLTypeInfo   *CompOrFatherSort;
} tSDLGenInfo;
```

- `CompOrFatherSort` : これは親ソート（syntype, inherits）またはコンポーネント ソート（Own, ORef）の型情報ノードへの参照です。

PowerSet の型情報ノード コンポーネント（**[ ]** 内で **unsigned** として実装 **)**

```
typedef T_CONST struct tSDLPowersetInfoS {
  tSDLTypeClass   TypeClass;
  unsigned char   OpNeeds;
  xptrint         SortSize;
  struct tSDLFuncInfo *OpFuncs;
  T_SDL_EXTRA_COMP
#ifdef T_SDL_NAMES
  char           *Name;
#endif
#ifdef XREADANDWRITEF
  xIdNode         FatherScope;
  xSortIdNode     SortIdNode;
#endif
  tSDLTypeInfo   *CompSort;
  int             Length;
  int             LowestValue;
} tSDLPowersetInfo;
```

- `CompSort` : コンポーネント ソートの型情報ノードへの参照。
- `Length` : コンポーネント ソート内で予定される値の数。
- `LowestValue` : コンポーネント ソート内の最下位の値。

構造体の型情報ノード コンポーネント

```
typedef int (*tGetFunc) (void *);
typedef void (*tAssFunc) (void *, int);


typedef T_CONST struct {
  xptrint     OffsetPresent; /* 0 if not optional */
  void       *DefaultValue;
} tSDLFieldOptInfo;


typedef T_CONST struct {
  tGetFunc        GetTag;
  tAssFunc        AssTag;
} tSDLFieldBitFInfo;


typedef T_CONST struct {
  tSDLTypeInfo   *CompSort;
#ifdef T_SDL_NAMES
  char           *Name;
#endif
  xptrint         Offset;     /* ~0 for bitfield */
  tSDLFieldOptInfo *ExtraInfo;
} tSDLFieldInfo;
```

```
typedef T_CONST struct tSDLStructInfoS {
  tSDLTypeClass   TypeClass;
  unsigned char   OpNeeds;
  xptrint         SortSize;
  struct tSDLFuncInfo *OpFuncs;
  T_SDL_EXTRA_COMP
#ifdef T_SDL_NAMES
  char           *Name;
#endif
#ifdef XREADANDWRITEF
  xIdNode         FatherScope;
  xSortIdNode     SortIdNode;
#endif
  tSDLFieldInfo  *Components;
  int             NumOfComponents;
} tSDLStructInfo;
```

- `Components`：`tSDLFieldInfo` の配列。構造体の各フィールドの配列の 1 コンポーネント。
- `NumOfComponents`：構造体のフィールド数。
- `tSDLFieldInfo` 内の `CompSort` ：フィールド ソートの型情報ノードへの参照。
- `tSDLFieldInfo` 内の `Name`：文字列としてのフィールド名。
- `tSDLFieldInfo` 内の `Offset` ：UML 構造体を表す C 構造体のフィールドのオフセット。このコンポーネントは UML のビット フィールドについては ~0 です（ビット フィールドについてはオフセットを計算できません）。
- `tSDLFieldInfo` 内の `ExtraInfo`：このコンポーネントの変換は UML フィールドのプロパティによって決まります。
  - `Offset` が ~0 の場合、このフィールドはビット フィールドであり、`ExtraInfo` は、ビット フィールドの値を設定および取得する 2 つの関数を持つ `tSDLFieldBitFInfo` 構造体へのポインタとなります。
  - `Offset` が ~0 でなく、`ExtraInfo != 0` の場合、このフィールドはオプションであるか、デフォルト値を持つかのどちらかです。`ExtraInfo` は、`Present` フラグ（オプションでなければ 0）のためのオフセットを持つ `tSDLFieldOptInfo` 構造体へのポインタであり、またデフォルト値へのポインタです（デフォルト値がない場合は 0）。

choice の型情報ノード コンポーネント

```
typedef T_CONST struct {
  tSDLTypeInfo        *CompSort;
#ifdef T_SDL_NAMES
  char                *Name;
#endif
} tSDLChoiceFieldInfo;

typedef T_CONST struct tSDLChoiceInfoS {
  tSDLTypeClass   TypeClass;
  unsigned char   OpNeeds;
  xptrint         SortSize;
  struct tSDLFuncInfo *OpFuncs;
  T_SDL_EXTRA_COMP
```

```
#ifdef T_SDL_NAMES
  char           *Name;
#endif
#ifdef XREADANDWRITEF
  xIdNode         FatherScope;
  xSortIdNode     SortIdNode;
#endif
  tSDLChoiceFieldInfo *Components;
  int                  NumOfComponents;
  xptrint              OffsetToUnion;
  xptrint              TagSortSize;
#ifdef XREADANDWRITEF
  tSDLTypeInfo        *TagSort;
#endif
} tSDLChoiceInfo;
```

- `Components` : `tSDLChoiceFieldInfo` の配列。`choice` の各フィールドの配列の 1 コンポーネント。
- `NumOfComponents` : `choice` のフィールド数。
- `OffsetToUnion` : `choice` の表現の範囲内の、共用体の先頭へのオフセット。
- `TagSortSize` : タグ型のサイズ。
- `TagSort` : タグ ソートの型情報ノードへの参照。

`array` と `CArray` の型情報ノード コンポーネント

```
typedef T_CONST struct tSDLArrayInfoS {
  tSDLTypeClass   TypeClass;
  unsigned char   OpNeeds;
  xptrint         SortSize;
  struct tSDLFuncInfo *OpFuncs;
  T_SDL_EXTRA_COMP
#ifdef T_SDL_NAMES
  char           *Name;
#endif
#ifdef XREADANDWRITEF
  xIdNode         FatherScope;
  xSortIdNode     SortIdNode;
#endif
  tSDLTypeInfo   *CompSort;
  int             Length;
#ifdef XREADANDWRITEF
  tSDLTypeInfo   *IndexSort;
  int             LowestValue;
#endif
} tSDLArrayInfo;
```

- `CompSort` : コンポーネント ソートの型情報ノードへの参照。
- `Length` : 配列内のコンポーネント数。
- `IndexSort` : インデックス ソートの型情報ノードへの参照。
- `LowestValue` : インデックス範囲の先頭値（int として）。

一般的な配列の型情報ノード コンポーネント

一般的な配列は C のリンク リストとして表現される配列です。

```
typedef T_CONST struct tSDLGArrayInfoS {
  tSDLTypeClass   TypeClass;
  unsigned char   OpNeeds;
  xptrint         SortSize;
  struct tSDLFuncInfo *OpFuncs;
  T_SDL_EXTRA_COMP
#ifdef T_SDL_NAMES
  char           *Name;
#endif
#ifdef XREADANDWRITEF
  xIdNode         FatherScope;
  xSortIdNode     SortIdNode;
#endif
  tSDLTypeInfo   *IndexSort;
  tSDLTypeInfo   *CompSort;
  xptrint         yrecSize;
  xptrint         yrecIndexOffset;
  xptrint         yrecDataOffset;
  xptrint         arrayDataOffset;
} tSDLGArrayInfo;
```

- `IndexSort` : インデックス ソートの型情報ノードへの参照。
- `CompSort` : コンポーネント ソートの型情報ノードへの参照。
- `yrecSize` : `SDLType_yrec` 型のサイズ。
- `yrecIndexOffset` : `SDLType_yrec` 型にある `Index` のオフセット。
- `yrecDataOffset` : `SDLType_yrec` 型にある `Data` のオフセット。
- `arrayDataOffset` : `SDLType` 型にある `Data` のオフセット。`SDLType` は変換された配列型の C での名前です。

汎用の **PowerSet**、**Bag**、**String** および **Objectidentifier** の型情報ノード コンポーネント

```
typedef T_CONST struct tSDLGenListInfoS {
  tSDLTypeClass   TypeClass;
  unsigned char   OpNeeds;
  xptrint         SortSize;
  struct tSDLFuncInfo *OpFuncs;
  T_SDL_EXTRA_COMP
#ifdef T_SDL_NAMES
  char           *Name;
#endif
#ifdef XREADANDWRITEF
  xIdNode         FatherScope;
  xSortIdNode     SortIdNode;
#endif
  tSDLTypeInfo   *CompSort;
  xptrint         yrecSize;
  xptrint         yrecDataOffset;
} tSDLGenListInfo;
```

- `CompSort` : コンポーネント ソートの型情報ノードへの参照。
- `yrecSize` : `SDLType_yrec` 型のサイズ。
- `yrecDataOffset` : `SDLType_yrec` 型にある Data のオフセット。

制限された文字列の型情報ノード コンポーネント

制限された文字列は C の配列として実装される文字列です。

```
/* ------------------ LString ------------------- */
typedef T_CONST struct tSDLLStringInfoS {
  tSDLTypeClass   TypeClass;
  unsigned char   OpNeeds;
  xptrint         SortSize;
  struct tSDLFuncInfo *OpFuncs;
  T_SDL_EXTRA_COMP
#ifdef T_SDL_NAMES
  char           *Name;
#endif
#ifdef XREADANDWRITEF
  xIdNode         FatherScope;
  xSortIdNode     SortIdNode;
#endif
  tSDLTypeInfo   *CompSort;
  int             MaxLength;
  xptrint         DataOffset;
} tSDLLStringInfo;
```

- `CompSort` : コンポーネント ソートの型情報ノードへの参照。
- `MaxLength` : 文字列の最大長。
- `DataOffset` : `SDLType` 型にある `Data` のオフセット。`SDLType` は変換された文字列型の C での名前です。

シグナルの型情報ノード コンポーネント

シグナルは構造体と同じ方法で扱われます。

```
typedef T_CONST struct {
  tSDLTypeInfo       *ParaSort;
  xptrint             Offset;
} tSDLSignalParaInfo;

typedef T_CONST struct tSDLSignalInfoS {
  tSDLTypeClass   TypeClass;
  unsigned char   OpNeeds;
  xptrint         SortSize;
  struct tSDLFuncInfo *OpFuncs;
  T_SDL_EXTRA_COMP
#ifdef T_SIGNAL_SDL_NAMES
  char           *Name;
#endif
  tSDLSignalParaInfo *Param;
  int             NoOfPara;
} tSDLSignalInfo;
```

- `Param` : 各シグナル パラメータ型についての `tSDLSignalParaInfo` 型のコンポーネントを持つ配列。各パラメータについて、パラメータ ソートが型情報ノードへの参照として、またシグナルを表す構造体内のパラメータ値に対するオフセットとして与えられます。
- `NoOfPara` : シグナルに含まれるパラメータ数。

型情報ノードのユーティリティ マクロ

以下のユーティリティ マクロを使って、型情報ノードの構成、型情報ノードのコンパイラへの適用、またはインスタンスの追加やコンポーネントの削除が可能です。これは、C コード ジェネレータの通常のアプリケーションドメインで実行してはなりません。

```
#ifndef T_CONST
#define T_CONST const
#endif

#ifndef T_SDL_EXTRA_COMP
#define T_SDL_EXTRA_COMP
#define T_SDL_EXTRA_VALUE
#endif

#ifndef T_SDL_USERDEF_COMP
#define T_SDL_USERDEF_COMP
#endif

#if defined(XREADANDWRITEF) && !defined(T_SDL_NAMES)
#define T_SDL_NAMES
#endif

#ifdef T_SDL_NAMES
#define T_SDL_Names(P) , P
#else
#define T_SDL_Names(P)
#endif

#ifdef T_SIGNAL_SDL_NAMES
#define T_Signal_SDL_Names(P) , P
#else
#define T_Signal_SDL_Names(P)
#endif

#ifdef T_SDL_INFO
#define T_SDL_Info(P) , P
#else
#define T_SDL_Info(P)
#endif

#ifndef XNOUSE_OPFUNCS
#define T_SDL_OPFUNCS(P) , P
#else
#define T_SDL_OPFUNCS(P)
#endif

struct tSDLFuncInfo;
```

30

C コード ジェネレータマクロ

この章は、C プリプロセッサ マクロのリファレンスです。このマクロはランタイム ライブラリと C コード ジェネレータで生成したコードのプロパティを決定するのに使用します。

概要

この章はいくつかのセクションに分かれており、セクションごとに生成コードの主要側面を 1 つずつ取り上げます。各セクションで、マクロはアルファベット順に列挙されています。

ランタイム ライブラリ ソースとインクルード ファイルで、また生成 C コードでは、`#define/#ifdef/#ifndef` パターンを使用して、コンパイル時にコードのパートをインクルートしたりエクスクルードしたりします。

使用するマクロは以下の 3 つのグループに分類できます。

- 選択されたライブラリのプロパティを定義するマクロ
- プロパティの実装詳細を定義するマクロ
- コンパイラのプロパティを定義するマクロこのグループのマクロについては、第 25 章「C および AgileC ランタイム ライブラリ」の 920 ページ、「コンパイラへの適合」で説明します。

この章で取り上げる一部のマクロについては、記述を十分に理解するため、使用する基本データ構造、特に静的構造つまり `xIdNodes` の知識を習得することを強く推奨します。この情報は、991 ページの「C コード ジェネレータ シンボル テーブル」セクションにあります。

動的構造に使用するデータ型と、インスタンス、シグナル、タイマーなどのアプリケーションの振る舞いについてもよく理解しておくとよいでしょう。これらについては、963 ページの「C コード ジェネレータ ランタイム モデル」の章で説明しています。

C コード ジェネレータマクロ

以下のセクションに、C コード ジェネレータで使用するプリプロセッサ マクロの一覧を掲げます。

ライブラリ バージョン マクロ

SCTAPPLCLENV

これは、アプリケーションのライブラリを定義するマクロです。

SCTAPPLENV

これは、クロックのないアプリケーションのライブラリを定義するマクロです。

SCTDEB

これは、任意の環境のスタンドアロンモデル ベリファイヤ（Model Verifier）アプリケーションを定義するマクロです。

SCTDEBCL

これは、任意の環境のモデル ベリファイヤ（Model Verifier）アプリケーションをリアルタイムで定義するマクロです。

SCTDEBCLCOM

これは、ホスト デバッグ用にモデル ベリファイヤ（Model Verifier）アプリケーションをリアルタイムで定義するマクロです。

SCTDEBCLENV

これは、リアルタイム プロパティと環境関数で、任意の環境のスタンドアロン モデル ベリファイヤ（Model Verifier）アプリケーションを定義するマクロです。

SCTDEBCLENVCOM

これは、環境関数で、任意の環境のスタンドアロン モデル ベリファイヤ（Model Verifier）アプリケーションをリアルタイムで定義するマクロです。

SCTDEBCOM

これは、ホスト デバッグ用にモデル ベリファイヤ（Model Verifier）アプリケーションを定義するマクロです。

SCTOPT1APPLCLENV

これは、アプリケーションを最小限の所要メモリで定義するマクロで、実数は使用できません。コネクタに関する情報は生成されません。

SCTOPT2APPLCLENV

これは、アプリケーションを最小限の所要メモリで定義するマクロで、実数は使用できません。コネクタ情報は、`const` と宣言されます。

コンパイラ定義セクション マクロ

SCT_POSIX

これは、UNIX や POSIX のようなコンパイラやシステムを定義するマクロです。

SCT_WINDOWS

これは、Windows のコンパイラを定義するマクロです。

設定マクロ

COMMENT(P)

このマクロは次のように定義します。

```
#define COMMENT(P)
```

このマクロは、インクルードされた C コードにコメントを挿入するのに使用します。

GETINTRAND

これは、ランダム生成関数、通常 `rand()` または `random()` を定義するマクロです。

GETINTRAND_MAX

これは、GETINTRAND で記述された関数によって生成された最大の整数、通常 RAND_MAX または 2147483647（32 ビットの整数）を定義するマクロです。

SCT_VERSION_4_4

このマクロは、C コード ジェネレータ バージョン 4.4 が使用されていれば、生成コード中で定義されます。

XCAT(P1,P2)

これは、トークン P1 および P2 を連結する方法を定義するマクロです。オプションは以下のとおりです。

```
#define XCAT(P1,P2) P1##P2
```

または

```
#define XCAT(P1,P2) P1/**/P2
```

または

```
#define XCAT(P1,P2) XCAT2(P1)P2
#define XCAT2(P2) P2
```

X_LONG_INT

ソート整数は、C では `int` に変換されます。整数ソートを `long int` に変換するには、マクロ `X_LONG_INT` を定義します。

XMULTIBYTE_SUPPORT

マルチ バイト文字をサポートするコンパイラの場合は、このマクロを定義します。

XNOSELECT

このマクロは、UNIX オペレーティング システムで選択関数がサポートされていない場合に定義します。シミュレーション時に Enter キーを押して「ユーザー定義割り込み」を実装するのに使用するマクロです。

XNO_VERSION_CHECK

このマクロを定義すると、生成コードと `scttypes.h` ファイル間のバージョン チェックが行われません。

XSCT_CBASIC

このマクロは、CbasicC コード ジェネレータが使用されていれば、生成コード中で定義されます。アプリケーションのビルド時は、このマクロを定義してはなりません。

XSCT_CADVANCED

このマクロは、C コード ジェネレータが使用されていれば、生成コード中で定義されます。このマクロは定義する必要があります。

X_SCTTYPES_H

このマクロは `scttypes.h` で定義し、`scttypes.h` ファイルを問題なく複数回インクルードできるようにするために使用します。

X_XINT32_INT

`xint32` が `int` の場合はこのマクロを定義します。

X_XPTRINT_LONG

`xptrint` が `unsigned long` の場合はこのマクロを定義します。

一般プロパティ マクロ

ここで説明するプロパティ マクロは、ライブラリの調整に使用できます。特に記載のないプロパティの場合、すべての C コード、変数、構造体コンポーネントなどは、条件付きコンパイルを使用し、そのプロパティを使用するかどうかによってインクルードまたはエクスクルードします。

すなわち、たとえばモデル ベリファイヤ（Model Verifier）によって使用されるコマンドライン インタープリタのすべてのコードがアプリケーションで削除され、これでアプリケーションがより小さく高速になります。

プロパティ マクロは、以下で説明する関係の場合を除いて原則として独立で、組み合わせによって競合が発生しない限り、任意の組み合わせで使用できるはずです。

ただし、組み合わせの数が非常に多いため、すべての組み合わせをコンパイルし、正しく機能するかどうかテストすることもほぼ不可能です。機能しない組み合わせができてしまった場合は、モデル全体を Tau サポートまでお送りください。直ちにそのケースに対処します。

XASSERT

これは、無効なユーザー定義表明を検出し、報告するのに使用します。

XCALENDARCLOCK

これは、`sctos.c` のクロック関数（シミュレート時間でなく）の使用を指定するものです。時間は、クロック関数によって返されるものです。

XCLOCK

これは、`sctos.c` のクロック関数（シミュレート時間でなく）の使用を指定するものです。時間は、起動時はゼロです。

XCOVERAGE

これは、現在のコード カバレッジに関する情報を計算し、格納するコードでアプリケーションをコンパイルすることを指定するマクロです。`XMONITOR` とともに使用します。

XCTRACE

シミュレーション中に現在の C ライン番号を報告する能力を保持してコンパイルする場合は、このマクロを定義します。このマクロを定義すると、ソース C コードのどこで実行が停止しているかという情報をモニタに表示できます。この機能はモニタとともに使用するもので、これでモデル ベリファイヤ（Model Verifier）コマンド Show-C-Line-Number が実装できます。

XEALL

このマクロは以下をすべて定義するものです。XASSERT, XECREATE, XECHOICE, XECSOP, XEDECISION, XEERROR, XEEXPORT, XEFIXOF, XEINDEX, XEINTDIV, XEOPTIONAL, XEOUTPUT, XEOWN, XERANGE, XEREALDIV, XEREF

XECHOICE

このマクロを定義すると、`choice` 変数の非アクティブ コンポーネントへのアクセス試行が検出され、報告されます。

XECREATE

このマクロを定義すると、起動時に最大並行インスタンス数よりも多くの静的インスタンスが生成されているかどうかが検出され、報告されます。

XECSOP

このマクロを定義すると、ADT 操作のエラーが検出され、報告されます。

XEDECISION

このマクロを定義すると、考えられる `DecisionAction` からのパスがない場合、それが検出され、報告されます。

XEERROR

このマクロを定義すると、式のエラー項の使用が検出され、報告されます。

XEEXPORT

このマクロを定義すると、参照元のグローバル データのエラーが検出され、報告されます。

XEFIXOF

これは、実数が Fix 操作によって整数に変換された場合のオーバーフローを報告するマクロです。

XEINDEX

このマクロを定義すると、配列の境界外のインデックスが検出され、報告されます。

XEINTDIV

このマクロを定義すると、整数のゼロ除算が検出され、報告されます。

XENV

このコンパイル マクロを定義すると、環境関数 `xInitEnv`、`xCloseEnv`、`xInEnv`、`xOutEnv` が適切な場所で呼び出されます。

XEOPTIONAL

このマクロを定義すると、パッシブ クラスの非存在オプション属性へのアクセス試行が検出され、報告されます。

XEOUTPUT

このマクロを定義すると、シグナル送信（主としてシグナルが直ちに破棄されるシグナル送信）中の警告が検出され、報告されます。

XEOWN

このマクロを定義すると、Own ポインタと ORef ポインタの不正な使用が検出され、報告されます。

XERANGE

これは、範囲条件を含むソートの属性に値が割り当てられている場合に範囲エラーを報告するマクロです。

XEREALDIV

このマクロを定義すると、実数のゼロ除算が検出され、報告されます。

XEREF

このマクロを定義すると、ヌル ポインタ参照解除試行が検出され、報告されます。

XGRTRACE

このマクロにより、ソース UML ダイアグラムへのグラフィカル トレース バックが有効になります。この機能は、シミュレーションやデバッグ セッションおよび Show-Next-Symbol や Show-Previous-Symbol などのモデル ベリファイヤ（Model Verifier）コマンドのグラフィカル トレースを実装するのに使用します。グラフィカル トレースを

コマンドライン インタープリタなしで通常のトレースと同じ方法で使用できます（Trace_Default を GRTrace と置き換えます）。ただし、グラフィカル トレースは同期されるので、アプリケーションの速度が大幅に低下します。

XMAIN_NAME

このマクロを定義すると、sctsdl.c の main ( ) 関数の名前が、マクロによって付与された名前に変更されます。生成アプリケーションの統合時や、より大きい環境でのシミュレーション時に、main 関数が役立つ場合がありますが、main という名前は使用できません。この名前は、XMAIN_NAME マクロを定義することにより別の名前に変更できます。main 関数は、sctsdl.c ファイルにあります。

XMONITOR

このマクロを定義すると、モデル ベリファイヤ（Model Verifier）コマンドライン インタープリタがコンパイルされ、アプリケーションにリンクされます。このマクロにより、暗黙的に他のマクロがいくつか設定されます。

XMSCE

このマクロを定義すると、グラフィカル シーケンス図トレースが有効な状態でコードがコンパイルされます。

XNOMAIN

このマクロを定義すると、sctsdl.c の main 関数が削除されます。main 関数と xMainLoop 関数は、条件付コンパイルを使用して削除されます。この機能は、システムから生成されたコードを既存アプリケーションの一部にする場合、つまり、システムによって既存環境に新しい関数が実装される場合に使用することを意図したものです。アクションのスケジューリングを実装するために以下の関数が使用できます。

```
extern void xMainInit(
  void (*Init_System) (void)
#ifdef XCONNECTPM
 ,int argc,
  char *argv[]
#endif
  );

#ifdef XNOMAIN
extern void SDL_Execute (void);

extern int SDL_Transition_Prio (void);

extern void SDL_OutputTimer (void);

extern int SDL_Timer_Prio (void);

extern SDL_Time SDL_Timer_Time (void);
#endif
```

これらの関数の振る舞いは以下のとおりです。

xMainInit : この関数は、システムの初期化のため、ランタイム ライブラリの他の関数を呼び出す前に呼び出します。

SDL_Execute : この関数は、キューにある最初のアクティブ クラス インスタンスによって遷移を 1 つ実行します。この関数を呼び出す前に、アクティブ クラスのインスタンスが少なくとも 1 つキューにあることをチェックする必要があります。

SDL_OutputTimer : この関数はタイマー出力を 1 つ実行します。タイマー出力を実行できる状態のタイマーがある場合のみ呼び出します。

SDL_Timer_Time : この関数は、タイマー キューにある最初のタイマーの set 文にある時間を返します。タイマー キューが空の場合、最大可能時間値（xSysD.xMaxTime）が返されます。システムが既存環境でどのように統合されているかによって、モニタ システムの使用も可能な場合があります。その場合、モニタ コマンドを実行するのに xCheckMonitors 関数を呼び出す必要があります。

```
extern void xCheckMonitors (void);
```

上記の関数の使用方法を理解しやすくするため、ランタイム ライブラリの内部スケジューラの動作を表す例を示します。

例 **374**

```
  while (1) {
#ifdef XMONITOR
    xCheckMonitors();
#endif
    if ( SDL_Timer_Prio() >= 0 )
      SDL_OutputTimer();
    else if ( SDL_Transition_Prio() >= 0 )
      SDL_Execute();
  }
```

XSIGLOG

この機能により、システム内主要イベントの独自のログを実装できます。通常、このマクロは定義しません。このマクロを定義すると、シグナルの各送信、つまり SDL_Output 関数の各呼び出しが xSignalLog 関数の呼び出しになります。遷移が開始されるたびに、xProcessLog 関数が呼び出されます。

これらの関数のプロトタイプは以下のとおりです。

```
extern void xSignalLog
  (xSignalNode  Signal,
   int          NrOfReceivers,
   xIdNode    * Path,
   int          PathLength);

extern void xProcessLog
  (xPrsNode P);
```

これは、XSIGLOG が定義されていれば scttypes.h にインクルードされます。

Signal は、シグナル インスタンスを表すデータ領域のポインタとなります。

NrOfReceivers は、次の表に示すとおり signal sending の成功を示します。

Numbers Of Receivers の値	output 文内容
-1:	直接アドレッシング節がありますが、送信側と受信側の間にチャンネルとシグナル ルートのパスは見つかりませんでした。
0:	直接アドレッシング節はなく、受信側が受信側検索で見つかりませんでした。
1:	文に直接アドレッシングが含まれている場合、送信側と受信側の間にチャンネルとシグナル ルートのパスが見つかります。 文に直接アドレッシングが含まれていない場合、受信側検索で受信側が 1 つだけ見つかります。 signal sending は成功しました。エラーが考えられるのは、直接アドレッシングを含む signal sending が、停止したアクティブ クラスのインスタンスにダイレクトされている場合のみです。

3 つ目のパラメータ Path は IdNodes へのポインタの配列で、Path[0] はアクティブ クラスの送信インスタンスに IdNode を参照し、Path[1] は送信側と受信側の間のパスにある最初のシグナル ルート（またはチャンネル）を参照し、以下同様に続き、Path[PathLength] が受信インスタンスに IdNode を参照します。

xProcessLog 関数の P パラメータは、実行がちょうど開始されるアクティブ クラスのインスタンスを参照します。

4 つ目のパラメータ PathLength は、送信されたシグナルのパスを表すのに使用される Path 配列のコンポーネントの数を表します。環境とシグナルがやりとりされる場合、Path[0] または Path[PathLength] が、xEnvId、つまり「アクティブ クラス」環境の IdNode を参照します。

ユーザーが指定する xSignalLog 関数および xProcessLog 関数の実装では、パラメータによって付与される情報を、任意の適切な方法で自由に使用できます。ただし、シグナル インスタンスの内容を変更することはできません。これらの関数は、標準の入力や出力がない環境や、モニタ システムが遅すぎたり大きすぎて入らなかったりする環境に簡単なログ機能を実装できるようにするためのものです。適切な実装は、sctenv.c ファイルにあります。

XTENV

これは XENV と同じです（事実 XENV を定義します）が、xInEnv が次に呼び出されるときの時間値（タイプ SDL_Time の値）を返すという点が異なります。main ループは、指定時間経過後の最初の機会またはシステムがアイドルになった（シグナルを待っている）ときに xInEnv を呼び出します。

XTRACE

このマクロを定義すると、実行のトレースを標準出力できます。この機能は通常モデル ベリファイヤ（Model Verifier）とともに使用しますが、これはなくてもかまいません。出力のために `stdout` ファイルが使用できるようになっている必要があります。

モデル ベリファイヤ（Model Verifier）がない場合、ユーザー インターフェイスが使用できないため、インクルードされた C コードでトレース値を設定する必要があります。トレース コンポーネントは `Trace_Default` と呼ばれ、システムで定義されたさまざまなアクティブ クラスを表す `IdNodes`、およびアクティブ クラスのインスタンスを表すのに使用する構造体 `xPrsRec` にあります。これらのコンポーネントに格納される値は、モデル ベリファイヤ（Model Verifier）の Set-Trace コマンドによって付与される値です。値が指定されない場合は、-1 で表されます。

モデル ベリファイヤ（Model Verifier）がない場合は、トレース値 0 のシステム以外、すべてのトレース値が起動時に未定義となります。すなわち、起動時にアクティブなトレースはありません。

例 **375**

C コードでトレース値を設定するのに適切な文を以下に示します。

```
xSystemId->Trace_Default = value;
   /* System trace */
xPrsN_ProcessName->Trace_Default = value;
   /* Process type trace */
PId_Var.LocalPId->PrsP->NameNode->Trace_Default =
  value
   /* Process type trace */
PId_Var.LocalPId->PrsP->Trace_Default = value;
   /* Process instance trace */
```

`PId_Var` はタイプ `Pid` の変数と見なされます。

コード最適化 マクロ

XAVL_TIMER_QUEUE

このマクロを定義するとタイマーキューのデータ構造をソート済みリンクリストから AVL ツリーに変更します。大量のタイマーを使用しているモデルではパフォーマンスが改善されます。

XCONST

このマクロを定義すると、`xIdNode` 構造の大部分を、`XCONST` を `const` と定義することによって一定化できます。これはアプリケーションでのみ可能で、モデル ベリファイヤ（Model Verifier）ではできません。

XCONST_COMP

XCONST が const の場合は、通常このマクロを const と定義します。xIdNode 構造内のコンポーネント宣言に const を導入するのに使用します。

XCONST マクロおよび XCONST_COMP を使用し、IdStructs に使用されるメモリのほとんどを RAM から ROM に移動できます。もちろん、これはコンパイラとそのプロパティによります。

以下のマクロ定義が挿入できます。

```
#define XCONST const
#define XCONST_COMP const
```

これで、IdStructs のほとんどの宣言に const が導入されます。これで、C コンパイラが const を処理することになります。

XCONST_COMP マクロは、const を構造体定義内のコンポーネントに導入するのに使用します。これは、一部のコンパイラでは、構造体の const をそのものとして受け入れるのに必要です。

const の導入が成功すると、IdStructs 用のデータ領域の大部分が const になるため、多くの RAM メモリが節約できます。

XNOCONTSIGFUNC

このマクロは、トリガされた遷移上のガードの式を計算する関数をエクスクルードするために定義します。これで、ステートに対応する xIdNode の関数ポインタも 1 つ節約されます。このマクロを定義すると、トリガされた遷移上のガードは使用できません。

XNOENABCONDFUNC

このマクロは、ガードの式を計算する関数をエクスクルードするために定義します。これで、ステートに対応する xIdNode の関数ポインタも 1 つ節約されます。このマクロを定義すると、ガードは使用できません。

XNOEQTIMERFUNC

このマクロは、2 つのタイマーのパラメータを比較する関数をエクスクルードするために定義します。これで、シグナルに対応する xIdNode の関数ポインタも 1 つ節約されます。このマクロを定義すると、パラメータを持つタイマーは使用できません。

XNOREMOTEVARIDNODE

このマクロは、インターフェイスの属性の定義に対応する xIdNodes をインクルードしないために定義します。

XNOSIGNALIDNODE

このマクロは、シグナルやタイマーに対応する xSignalIdNodes をインクルードしないために定義します。

XNOSTARTUPIDNODE

このマクロは、起動シグナルに対応する xSignalIdNodes をインクルードしないために定義します。

XNOUSEOFEXPORT

このマクロを定義すると、グローバル属性を使用しないことが表明されます。

注記

XNOUSEOFEXPORT が定義されている場合にグローバル データを参照しようとすると、xGetExportAddr 関数が定義されていないため、コンパイル エラーとなります。

XNOUSEOFOBJECTIDENTIFIER

このマクロを定義すると、ObjectIdentifier タイプと、このタイプに対するすべての操作が削除されます。

XNOUSEOFOCTETBITSTRING

このマクロを定義すると、Bitstring タイプ、Octet タイプ、OctetString タイプとこれらのタイプに対するすべての操作が削除されます。

XNOUSEOFREAL

このマクロを定義すると、real タイプと、real に対するすべての操作が削除されます。このマクロを定義すると、Cfloat タイプおよび double タイプのすべてのオカレンスが削除され、たとえば、real タイプが使用できなくなります。

このマクロは、フローティング タイプの操作のライブラリ関数がロードされないよう、スペースを節約する必要がある場合に使用することを意図したものです。C コードにフローティング タイプの操作をインクルードする場合には使用できません。また、BasicCTypes.pr その他の ADT がシステムにインクルードされている場合も注意が必要です。この場合、real に依存するタイプがパッケージから削除されていなければなりません。

XOPT

このマクロは、完全最適化（XOPTCHAN を除く）をオンにします。つまり、以下のマクロを定義します。

XOPTSIGPARA	XOPTDCL
XOPTFPAR	XOPTSTRUCT
XOPTLIT	XOPTSORT

これらの XOPT マクロをモニタとともに使用してはなりません。

XOPTCHAN

このマクロは、システム中のコネクタのパスに関するすべての情報を削除するのに使用できます。以下のメモリ最適化が実行されます。

- 各コネクタとポートの 2 つの ChannelIdNodes が削除されます。
- アクティブ クラスのインスタンスを表す xPrsIdNodes の ToId コンポーネントが削除されます。
- ライブラリ（sctsdl.c）のいくつかの関数が不要になり、削除されます。

コネクタとポートに関する情報がなくなると、以下の計算が実行できなくなります。

1. 直接アドレッシングを含む signal sending 文での送信側と受信側の間にコネクタのパスがあるかどうかのチェック。これは、アプリケーションではおそらく実行されない単なるエラー テストであるため、問題ではありません。
2. コード生成時に C コード ジェネレータが計算を実行しなかった場合の、直接アドレッシングを含まない signal sending での受信側の計算。直接アドレッシングを含まない signal sending が使用されることになるため、こちらのほうは深刻です。

通常のシステムでは、システムのパート間の通信を起動するのに、直接アドレッシングを含まない signal sending を使用する必要があります。これは、直接アドレッシングを含む signal sending に必要な Pid 値を配布する方法が他にないためです。

この状況は、C コード ジェネレータが受信側を計算できれば解決されます。計算できない場合は、抽象データ型のライブラリの PIdList データ型が解決に使用できます。このデータ型を使用する場合、グローバル Pid リテラルを定数属性として実装し、導入できます。これで、これらのリテラルを使用して、直接アドレッシング節を含む signal sending 文を最初から利用できます。

注記

このコンパイル マクロを定義すると、すべてのシグナル送信をアクティブ クラスのインスタンスの直接アドレッシングで行うか、シグナルの受信側がコード生成時に計算可能でなければなりません。XOPTCHAN マクロを定義し、かつ直接アドレッシング節のない signal sending を使用すると（この場合 C コード ジェネレータによる最適化ができません）、xNotDefPId という名前が定義されていないという C コンパイル エラーとなります。

XOPTDCL

このマクロは、属性に対応する xIdNodes をインクルードしないために定義します。シンボル テーブル ツリーには、アクティブ クラスと操作で宣言された各属性に対応する VarIdNode があります。これらのノードはアプリケーションで使用されないため、XOPTDCL マクロを定義することによって削除してもかまいません。

XOPTFPAR

このマクロは、状態機械のパラメータに対応する xIdNodes をインクルードしないために定義します。シンボル テーブル ツリーには、アクティブ クラスと操作の各仮パラメータに対応する VarIdNode があります。これらのノードはアプリケーションで使用されないため、XOPTFPAR マクロを定義することによって削除してもかまいません。

XOPTLIT

このマクロは、リテラルに対応する xIdNodes をインクルードしないために定義します。enum タイプに変換される、パッシブ クラスの各リテラルに対し、リテラルを表す LitIdNode があります。これらのノードはアプリケーションで使用されないため、XOPTLIT マクロを定義することによって削除してもかまいません。

XOPTSIGPARA

このマクロは、シグナル パラメータに対応する xIdNodes をインクルードしないために定義します。シンボル テーブル ツリーには、シグナルの各パラメータに対するノードが 1 つあります。これらのノードはアプリケーションで必要でないため、XOPTSIGPARA マクロを定義することによって削除してもかまいません。

XOPTSORT

このマクロは、パッシブ クラスとシンタイプに対応する xIdNodes をインクルードしないために定義します。各パッシブ クラスとシンタイプは、SortIdNode によって表されます。これらのノードは、上記の他の XOPT (XOPTSIGPARA, XOPTDCL, XOPTFPAR, XOPTSTRUCT, XOPTLIT) がすべて定義されていればアプリケーションで使用されません。

XOPTSTRUCT

このマクロは、構造体コンポーネントに対応する xIdNodes をインクルードしないために定義します。SDL 構造体の各コンポーネントに対し、このコンポーネントのプロパティを定義する VarIdNode が 1 つあります。VarIdNode はアプリケーションで使用されないため、XOPTSTRUCT マクロを定義することによって削除してもかまいません。

XPATH_INFO_IN_ENV_FUNC

このコンパイル スイッチは、アプリケーションから環境に送信されるシグナルのシグナル パスをユーザーが使用できるようにする必要がある場合に設定します。

XPRSCOUNT, XPRSCOUNTHASH

あるアクティブクラスのインスタンス数を数えるアクションを最適化します。インスタンス数の計数は新しいインスタンス作成前に行われます。したがって、この機能によって、動的に大量のインスタンスを作成する際のパフォーマンスが改善されます。までに XPRSCOUNT マクロは事前に計算した値のインスタンス数のリンクリストであり、XPRSCOUNTHASH マクロは高速ルックアップのためにハッシュテーブルに数を格納することで PRSCOUNT を最適化しています。

XPRSHASH

アクティブクラスをトラッキングするデータ構造を 1 つのリンクリストから 1 つのハッシュテーブルに変更します。この機能は、アクティブクラスのインスタンス数が多いシステム向きです。

注記

複数多重度のパートとしてのアクティブクラスインスタンスがある場合、この最適化はパートコンテナ内のインスタンスの順序を変更します。インスタンスの挿入、追加の際にこの順序変更が発生します。コンテナのインデクス値に依存したコードは作成しないでください。インデクス値に依存した機能が必要な場合は、クラス参照についてのパラレルコンテナを作成してください。

XPRSOPT

このマクロを定義すると、アクティブ クラスのインスタンスのメモリ使用が最適化されます。すべてのメモリが再利用可能ですが、新しいアクティブ クラス インスタンスによってメモリが再利用された停止インスタンスへのシグナル送信は検出できません。この場合、新しいインスタンスがシグナルを受信します。

第 28 章「C コード ジェネレータ ランタイム モデル」の 974 ページ、「create 操作と stop 操作」セクションで、xLocalPIdRec 構造がアクティブ クラスの各生成インスタンスにどのように割り当てられるか、またこれらの構造が、stop アクション実行後もインスタンスを表すのにどのように使用されるかを説明しています。停止操作の実行されたアクティブ クラスのインスタンスにシグナルがいつ送信されたかを検出するには、この xLocalPIdRecs 処理メソッドが必要です。

動的に生成されたインスタンスを使用し、長期間実行されるアプリケーションでは、この方法で xLocalPIdRecs を処理すると、使用可能メモリがいずれなくなります。

XPRSOPT マクロを定義すると、xLocalPIdRecs のメモリが yVDef_ProcessName 構造とともに再利用されます。この結果、

- 動的に生成され終了したインスタンスの使用のために必要メモリが増加することはなくなります（所要メモリはアクティブ クラスの最大並行インスタンス数によるため）。

- シグナルが、stop アクションを実行したインスタンスにシグナルが送信される状況を常に見つけることができなくなります。

正確に言えば、アクティブ クラスのインスタンスを参照する Pid 属性が停止操作を実行する状況を考えます。その後、同じアクティブ クラスで、同じデータ領域を再利用して生成操作が実行されます。これで、Pid 属性は新しいインスタンスを参照するようになります。

これは、たとえば、古いインスタンスを意図したシグナルが新しいインスタンスに送信されるということです。XPRSOPT を定義しても、使用可能メモリ リストのインスタンスへのシグナル送信を検出することは可能です。

XUSE_SIGNAL_NUMBERS

このコンパイル スイッチは、環境関数が名前でなく（C コード ジェネレータによって割り当てられた）番号によってシグナルを検索する設計の環境にシグナルを実装するアプリケーションをコンパイルする場合に設定する必要があります。この機能の使用方法については、899 ページの「シグナルが多数ある場合の xOutEnv のパフォーマンス向上」を参照してください。

副次的機能を定義するマクロ

XBREAKBEFORE

このマクロは、主に MONITOR マクロまたは GRTRACE マクロが定義されている場合に定義します。このマクロを使用すると、参照用の関数や構造体コンポーネントが利用できるようになります。また、以下のマクロ

- XAT_FIRST_SYMBOL
- XBETWEEN_SYMBOLS
- XBETWEEN_SYMBOLS_PRD
- XBETWEEN_STMTS
- XBETWEEN_STMTS_PRD
- XAFTER_VALUE_RET_PRDCALL
- XAT_LAST_SYMBOL

を適切な関数呼び出しに展開するのにも使用できます。これらの関数は、デバッグ時にシンボル間の遷移を中断するのに使用します。

XCASEAFTERPRDLABELS

操作呼び出し直後のシンボルは、これらのシンボルのシンボル番号（C のケース ラベル）が、呼び出しフロー グラフの再開アドレスとして使用されるため、特別に処理しなければなりません。通常、このマクロは定義する必要があります。操作の呼び出しが適切な C 関数呼び出しに、Return が C return に変換され、操作の Nextstate が C return

に変換されなければ（つまり、操作を表す C 関数でアクティブ クラスが「保留」であれば）、XCASEAFTERPRDLABELS を定義する必要はありません。このマクロは XCASELABELS の関数と関連します。

XCASELABELS

状態機械または操作の振る舞いを実装する関数には、各シンボルのケース ラベルを含む大きな C スイッチ文が含まれます。このスイッチは、状態機械または操作の実行を任意のシンボルで再開できるようにするために使用します。アプリケーションでは、これらのラベルのほとんど（遷移を開始するシンボル、つまり開始、トリガ付き遷移、トリガされた遷移上のガードを除くすべて）を削除できます。すべてのシンボルのケース ラベルを導入するには、XCASELABELS マクロを定義する必要があります。すなわち、シミュレーションでは XCASELABELS を定義する必要がありますが、アプリケーションではその必要はありません。

XCOUNTRESETS

このマクロは、リセット操作時に削除されるタイマーの数をカウントするのに使用します。この情報は、情報を提供するテキスト トレース（XTRACE）によって使用されます。これは複数のタイマーが（暗黙的に）リセットされ得る場合、stop アクション時に有用です。アプリケーションでは、XCOUNTRESETS を定義してはなりません。

XENVSIGNALLIMIT

このマクロを定義すると、制限された数のシグナルのみが環境関数の入力ポートに格納されます。このマクロは、シミュレーション時、「環境アクティブ クラス」の入力ポートに保存する環境に送信されるシグナルの数を決定するために定義します。このようなシグナルは、シグナルを一覧表示するモデル ベリファイヤ（Model Verifier）コマンドで表示できます。このマクロはシミュレーションでのみ有用です。制限は `XENVSIGNALLIMIT` に定義された値と等しく、通常は 20 に設定されます。

XERRORSTATE

このマクロは、分岐から見つかったパスがない場合に使用できる「エラー」ステートを表すデータ構造を挿入するのに使用します。これは通常、XEDECISION が定義されていれば、定義する必要があります。

XFREESIGNALFUNCS

このマクロは、free 関数を含むタイプのパラメータを含む各シグナル、タイマー、起動シグナル用の free 関数を挿入するのに使用します。これらのシグナル free 関数は、シグナル内の割り当てデータを解放するのに使用できます。

XFREEVARS

このマクロは、stop アクションまたは return アクション直前に、free 関数を含むタイプのすべての属性用に free 関数呼び出しを挿入するのに使用します。すなわち、free アクションは、オブジェクトがなくなる前に、属性から参照される割り当てデータに対して実行されます。このマクロは定義する必要があります。

XIDNAMES

このマクロは、オブジェクトの名前を、オブジェクトに対応する `xIdNode` に格納する必要があるかどうかを判定するのに使用します。この文字列は、たとえばモデル ベリファイヤ（Model Verifier）で、ユーザーとの通信に使用します。通常、このマクロはアプリケーションでは使用してはなりません。

ターゲット デバッグで XIDNAMES を定義すると便利な場合があります。そうすれば、モデル ベリファイヤ（Model Verifier）から名前を出力するだけでオブジェクトを簡単に特定できるからです。この機能には、数パーセント余分のメモリが必要です。

XNRINST

アクティブ クラス インスタンス番号（アクティブ クラス インスタンス セットの個々のインスタンスに関連付けられている番号）を維持するには、このマクロを定義する必要があります。XNRINST は通常、モデル ベリファイヤ（Model Verifier）などのシミュレーション アプリケーションにのみ使用されます。

XOPERRORF

このマクロは、`sctsdl.c` に `xSDLOpError` 関数をインクルードするのに定義します。この関数は、ADT 操作のランタイム エラーを出力するのに使用します。

XPRSSENDER

このマクロは、`xPrsNode` にも送信側の値を格納するのに使用します。通常の場所は、最新の受信シグナルです。これは、シミュレーションでのみ必要となります。遷移が完了し、シグナルが使用可能メモリのプールに戻された後に送信側がモデル ベリファイヤ（Model Verifier）からアクセスされるためです。

XREADANDWRITEF

このマクロは、基本の Read および Write の関数をインクルードするのに使用します。これは主にシミュレーションで必要となります。

XREMOVETIMERSIG

このマクロは、非実行 Pid インスタンスのタイマーを削除できるようにするのに使用します。これは、タイマーの設定とリセットを行うモデル ベリファイヤ（Model Verifier）コマンドの実装にシミュレーションでのみ必要となります。

XSIGPATH

このマクロを定義すると、xIsPath 関数と xFindReceiver 関数が、送信側から受信側へのコネクタとポートのパスを out パラメータとして返します。これで、たとえばシグナル ログ生成に、この情報がモデル ベリファイヤ（Model Verifier）で使用できるようになります。通常、アプリケーションではこのマクロを定義してはなりません。

XSYMBTLINK

XSYMBTLINK マクロは、xIdNodes から完全なツリーをビルドする必要があるかどうかを判定するのに使用します。XSYMBTLINK を定義すると、すべての xIdNodes に Parent ポインタ、Suc ポインタ、First ポインタが含まれます。Parent ポインタの値は、直接 xIdNodes に生成されます。Suc ポインタと First ポインタは、xInsertIdNode 関数を呼び出すことにより、yInit 関数で計算されます。Suc ポインタと First ポインタはモデル ベリファイヤ（Model Verifier）で必要ですが、アプリケーションでは不要です。つまり、モデル ベリファイヤ（Model Verifier）では XSYMBTLINK を定義する必要がありますが、アプリケーションではその必要はありません。

XTESTF

このマクロは、シンタイプ（またはパッシブ クラス）の test 関数を範囲条件でインクルードまたは削除するのに使用します。yTest 関数は、配列の境界外インデックスと範囲をテストするためにモデル ベリファイヤ（Model Verifier）によって使用されます。すなわち、モニタが使用されている場合や、XERANGE または XEINDEX が定義されている場合に、XTESTF を定義する必要があります。

XTRACHANNELSTOENV

これは、環境へのリダイレクト コネクタを定義するマクロです。

アプリケーションのパーティションを使用する場合、環境へのコネクタの数は、コード生成時はわかりません。これは、接続に使用されるデータ領域のサイズがわからないということです。この状況は、以下の 2 つの方法で解決できます。

リダイレクトを処理する関数がより多くのメモリを割り当てるようにするか（これがデフォルトです）、リダイレクトされるコネクタの数を指定します（これは計算が困難な場合がありますが、所要メモリ量は少なくなります）。

例 **376**

前者（より多くのメモリを割り当てる）の場合、マクロ

```
#define XTRACHANNELSTOENV  0
#define XTRACHANNELLIST
```

を上記のように定義します。これは、scttypes.h の標準です。

コネクタの数を指定する場合は、マクロを以下のように定義します。

```
#define XTRACHANNELSTOENV  10
#define XTRACHANNELLIST    ,0,0,0,0,0,0,0,0,0,0
```

つまり、XTRACHANNELSTOENV がコネクタの数（上記の例では 10）で、XTRACHANNELLIST が同じ数のゼロのリスト（この場合はゼロ 10 個のシーケンス）です。

XTRACHANNELLIST

このマクロは XTRACHANNELSTOENV の関数と関連します。

静的データのマクロ、主に **xIdNode**

XBLO_EXTRAS

パートを含む、（場合によってはインラインの）アクティブ クラスのすべての生成構造体値には、構造体の最後にこのマクロが含まれます。このマクロを定義すると、新しいコンポーネントを挿入できます。xBlockIdStruct タイプも更新する必要があります。通常、このマクロは空にしておきます。

例 **377**

```
#define XBLO_EXTRAS  ,0
```

XBLS_EXTRAS

パート構造を含むアクティブ クラスのすべての生成構造体値には、構造体の最後にこのマクロが含まれます。このマクロを定義すると、新しいコンポーネントを挿入できます。xBlockSubstIdStruct タイプも更新する必要があります。通常、このマクロは空にしておきます。

例 **378**

```
#define XBLS_EXTRAS  ,0
```

XCOMMON_EXTRAS

xIdNode 構造のすべての生成構造体値には、共通コンポーネントの後にこのマクロが含まれます。すなわち、このマクロを定義することによって、すべての xIdNodes に新しいコンポーネントを挿入できます。通常、このマクロは空にしておきます。

例 **379**

新しい int コンポーネントを値 0 で挿入するには、次の定義が使用できます。

```
#define XCOMMON_EXTRAS  ,0
```

XLIT_EXTRAS

リテラル構造のすべての生成構造体値には、構造体の最後にこのマクロが含まれます。このマクロを定義すると、新しいコンポーネントを挿入できます。`xLiteralIdStruct` タイプも更新する必要があります。通常、このマクロは空にしておきます。

例 **380**

```
#define XLIT_EXTRAS  ,0
```

XPAC_EXTRAS

パッケージ構造のすべての生成構造体値には、構造体の最後にこのマクロが含まれます。このマクロを定義すると、新しいコンポーネントを挿入できます。`xPackageIdStruct` タイプも更新する必要があります。通常、このマクロは空にしておきます。

例 **381**

```
#define XSYS_EXTRAS  ,0
```

XPRD_EXTRAS

操作構造のすべての生成構造体値には、構造体の最後にこのマクロが含まれます。このマクロを定義すると、新しいコンポーネントを挿入できます。`xPrdIdStruct` タイプも更新する必要があります。通常、このマクロは空にしておきます。

例 **382**

```
#define XSYS_EXTRAS  ,0
```

XPRS_EXTRAS

```
(PREFIX_PROC_NAME)
```

アクティブ クラス、アクティブ クラスのインスタンス、インライン アクティブ クラス構造のすべての生成構造体値には、構造体の最後にこのマクロが含まれます。このマクロを定義すると、新しいコンポーネントを挿入できます。`xPrsIdStruct` タイプも更新する必要があります。

例 **383**

```
#define XPRS_EXTRAS(PREFIX_PROC_NAME) ¥
   ,XCAT(PREFIX_PROC_NAME,_STACKSIZE)
```

XSIG_EXTRAS

シグナル、タイマー、起動シグナル構造のすべての生成構造体値には、構造体の最後にこのマクロが含まれます。このマクロを定義すると、新しいコンポーネントを挿入できます。`xSignalIdStruct` タイプも更新する必要があります。

通常、このマクロは空にしておきます。

例 **384**

```
#define XSIG_EXTRAS  ,0
```

XSPA_EXTRAS

信号パラメータ構造のすべての生成構造体値には、構造体の最後にこのマクロが含まれます。このマクロを定義すると、新しいコンポーネントを挿入できます。`xVarIdStruct` タイプも更新する必要があります。

注記

属性、状態機械のパラメータ、シグナル パラメータ、構造体コンポーネントは、すべて `xVarIdStruct` で処理されます。

通常、このマクロは空にしておきます。

例 **385**

```
#define XSPA_EXTRAS  ,0
```

XSRT_EXTRAS

パッシブ クラスおよびシンタイプ構造のすべての生成構造体値には、構造体の最後にこのマクロが含まれます。このマクロを定義すると、新しいコンポーネントを挿入できます。`xSortIdStruct` タイプも更新する必要があります。通常、このマクロは空にしておきます。

例 **386**

```
#define XSRT_EXTRAS  ,0
```

XSTA_EXTRAS

ステート構造のすべての生成構造体値には、構造体の最後にこのマクロが含まれます。このマクロを定義すると、新しいコンポーネントを挿入できます。`xStateIdStruct` タイプも更新する必要があります。通常、このマクロは空にしておきます。

例 **387**

```
#define XSTA_EXTRAS  ,0
```

XSYS_EXTRAS

システム（「ルート」アクティブ クラスおよびそのインスタンス）のすべての生成構造体値には、このマクロが含まれます。このマクロを定義すると、構造体の最後に新しいコンポーネントを挿入できます。`xSystemIdStruct` タイプも更新する必要があります。通常、このマクロは空にしておきます。

例 **388**

```
#define XSYS_EXTRAS  ,0
```

XSYSTEMVARS

このマクロにより、「ルート」アクティブ クラスまたはメイン スレッドの実装を含む C ファイルの初めに宣言されたグローバル変数を導入できます。

XSYSTEMVARS_H

XSYSTEMVARS で宣言されたデータに `extern` 定義が必要な場合は、ここに導入します。これらの定義は、メイン スレッドの `.h` ファイルに入れられます（個別生成を使用した場合）。

XVAR_EXTRAS

属性、状態機械のパラメータ、構造体コンポーネントのすべての生成構造体値には、構造体の最後にこのマクロが含まれます。このマクロを定義すると、新しいコンポーネントを挿入できます。`xVarIdStruct` タイプも更新する必要があります（シグナル パラメータは `xVarIdStruct` タイプも使用します）。通常、このマクロは空にしておきます。

例 **389**

```
#define XVAR_EXTRAS  ,0
```

状態機械のデータとアクティブ クラスの操作

PROCEDURE_VARS

これは、ステートなど、アクティブ クラスの各操作に必要な構造体コンポーネントを定義するマクロです。

PROCESS_VARS

これは、状態機械の各インスタンスに必要な構造体コンポーネントを定義するマクロです。例 :

```
state, parent, offspring, self, sender, inputport
```

YGLOBALPRD_YVARP

このマクロは、アクティブ クラス外で定義された操作で、アクティブ クラスの yVDef 構造体を指し示す yVarP ポインタを宣言するのに使用します。グローバル操作ではアクティブ クラスにローカルなデータにアクセスできないため、PROCESS_VARS マクロで定義されたコンポーネントのみを含む構造体のポインタとして yVarP を使用するのが適切です。

YPAD_TEMP_VARS

状態機械の PAD 関数のローカル変数です。たとえば、signal sending や create アクションに必要な一時変数となります。

YPAD_YSVARP

受信シグナルの参照に使用する ySVarP ポインタの宣言です。通常、ySVarP は void * です。

YPAD_YVARP

(VDEF_TYPE)

このマクロは状態機械内で使用します。これは、状態機械中の属性にアクセスするのに使用するポインタ、yVarP の宣言に展開します。yVarP は VDEF_TYPE * タイプでなければなりません。VDEF_TYPE は状態機械の yVDef 構造体のタイプです。yVDef 構造体へのポインタが PAD 関数のパラメータとして渡されると、すでに宣言にある正しい値を yVarP に割り当てることができます。

YPRD_TEMP_VARS

これは、アクティブ クラスの操作の振る舞いを実装する関数のローカル変数を定義するマクロです。

YPRD_YVARP

パラメータ : (VDEF_TYPE)

このマクロは、アクティブ クラスの操作内で使用します。これは、アクティブ クラス中の属性にアクセスするのに使用するポインタ、yVarP の宣言に展開します。yVarP は VDEF_TYPE * タイプでなければなりません。VDEF_TYPE は状態機械の yVDef 構造体のタイプです。yVDef 構造体へのポインタが操作関数のパラメータとして渡されると、すでに宣言にある正しい値を yVarP に割り当てることができます。

PAD 関数で使用するマクロ

BEGIN_PAD

パラメータ：(VDEF_TYPE)

BEGIN_PAD は、PAD 関数の初めで実行されるコードを挿入するのに使用できます。VDEF_TYPE は、状態機械の yVDef type です。

BEGIN_START_TRANSITION

パラメータ：(STARTUP_PAR_TYPE)

このマクロは、開始遷移の初めで実行されるコードを挿入するのに使用できます。STARTUP_PAR_TYPE は、この状態機械の起動シグナルの構造体（接頭辞として ySignalPar の付いた）です。

CALL_SUPER_PAD_START

パラメータ：(PAD)

状態機械の開始遷移中、開始シンボルを含む、PAD 関数まで（PAD 関数を含む）のすべての継承 PAD 関数 を呼び出す必要があります。これは、状態機械で定義されたすべての属性を初期化するためです。このマクロは、継承 PAD 関数（マクロ パラメータ PAD）の呼び出しを実行するのに使用します。通常、このマクロは次のように展開します。

```
yVarP->RestartPAD = PAD; PAD(VarP);
```

この後に、実行モデルにより、return または goto NewTransition が続きます。

CALL_SUPER_PRD_START

パラメータ：(PRD, THISPRD)

このマクロの使用方法は CALL_SUPER_PAD_START と同様ですが、アクティブ クラスの操作の開始遷移に使用します。THISPRD が実行関数で、PRD が継承関数です。

LOOP_LABEL

LOOP_LABEL マクロは、OS タスクが遷移の終わりで return を実行しない場合（これは多くの OS に当てはまります）に、OS で必要な、nextstate 操作から次のシグナル受信操作までのループを形成するのに使用します。また、受信シグナルに対する free の処理や save queue の処理にも適しています。nextstate が Creturn を使用して実装される OS では、LOOP_LABEL マクロは通常空です。

LOOP_LABEL_PRD

このマクロは LOOP_LABEL と似ていますが、ステートを含むアクティブ クラスの操作に使用します。

LOOP_LABEL_PRD_NOSTATE

このマクロは LOOP_LABEL と似ていますが、ステートを含まないアクティブ クラスの操作に使用します。このマクロは通常、何にも展開しません。

SDL_OFFSPRING

子の値を返します。

SDL_PARENT

親の値を返します。

SDL_SELF

自分の値を返します。

SDL_SENDER

送信側の値を返します。

XEND_PRD

これは、アクティブ クラスの操作の振る舞いを表す関数の終わりに生成されるマクロです。何にも展開する必要はありません。

これを次のように定義すると

```
return (xbool)0;
```

値を返す関数の終わりに到達するというコンパイラ警告が削除される場合があります。

XPRSNODE

このマクロは通常、`xPrsNode` タイプに展開します。

XNAMENODE

このマクロは、PAD 関数から `xPrsIdNode` に到達します。これは通常 `yVarP->NameNode` です。

XNAMENODE_PRD

このマクロは、PRD 関数から `xPrdIdNode` に到達します。これは通常 `yPrdVarP->NameNode` です。

YPAD_FUNCTION

パラメータ：`(PAD)`

このマクロは、パラメータとして付与される PAD 関数の関数ヘッディングを示します。

YPAD_PROTOTYPE

パラメータ：(PAD)

このマクロは、パラメータとして付与される PAD 関数の関数プロトタイプを示します。

YPRD_FUNCTION

パラメータ：(PRD)

このマクロは、パラメータとして付与される PRD 関数の関数ヘッディングを示します。

YPRD_PROTOTYPE

パラメータ：(PRD)

このマクロは、パラメータとして付与される PRD 関数の関数プロトタイプを示します。

yInit 関数のマクロ

BEGIN_YINIT

このマクロは、yInit 関数の初めに置きます。変数宣言や初期化コードに展開できます。

XPROCESSDEF_C

パラメータ：(PROC_NAME, PROC_NAME_STRING, PREFIX_PROC_NAME, PAD_FUNCTION, VDEF_TYPE)

このマクロは、アクティブ クラスの各インスタンス セットのコードを導入するのに使用できます。

- PROC_NAME
  接頭辞のないアクティブ クラス名です。
- PROC_NAME_STRING
  文字列で表したアクティブ クラス名です。
- PREFIX_PROC_NAME
  接頭辞のあるアクティブ クラス名です。
- PAD_FUNCTION
  このアクティブ クラス インスタンス セットの PAD 関数です。
- VDEF_TYPE
  このアクティブ クラスの yVDef 構造体です。

XPROCESSDEF_H

パラメータ : (PROC_NAME, PROC_NAME_STRING, PREFIX_PROC_NAME, PAD_FUNCTION, VDEF_TYPE)

このマクロは、各アクティブ クラス インスタンス セットの extern 宣言 (適切な .h ファイルに置かれた) を導入するのに使用できます。

- PROC_NAME
  接頭辞のないアクティブ クラス名です。
- PROC_NAME_STRING
  文字列で表したアクティブ クラス名です。
- PREFIX_PROC_NAME
  接頭辞のあるアクティブ クラス名です。
- PAD_FUNCTION
  このアクティブ クラス インスタンス セットの PAD 関数です。
- VDEF_TYPE
  このアクティブ クラスの yVDef 構造体です。

xInsertIdNode

yInit 関数では、xInsertIdNode 関数が各 IdNode に対して呼び出されます。これは、アプリケーションでは不要で、xInsertIdNode マクロは次のように定義できます。

```
#define xInsertIdNode(Node)
```

xInsertIdNode 関数は、XSYMBTLINK、XCOVERAGE、XMONITOR のいずれかが定義されていれば必要です。

YINIT_TEMP_VARS

このマクロはすべての yInit 関数に置かれ、yInit 関数内で必要なローカル変数に展開できます。

シグナルとシグナル送信の実装

ALLOC_SIGNAL

このマクロは ALLOC_SIGNAL_PAR とともに使用します。

ALLOC_SIGNAL_PAR

パラメータ : (SIG_NAME, SIG_IDNODE, RECEIVER, SIG_PAR_TYPE)

このマクロは、ALLOC_SIGNAL とともに、送信するシグナルのためのデータ領域を割り当てるのに使用します。ALLOC_SIGNAL はシグナルにパラメータがない場合に、ALLOC_SIGNAL_PAR はシグナルにパラメータがある場合に使用します。割り当てられたデータ領域は、OUTSIGNAL_DATA_PTR マクロで記述された変数によって参照します。

- `SIG_NAME`
  接頭辞のないシグナル名です。
- `SIG_IDNODE`
  シグナルの `xSignalIdNode` です。
- `RECEIVER`
  これは、直接アドレッシング節で指定されたか、計算された受信側です。NO_TO signal sending では、`RECEIVER` は `xNotDefPId` です。
- `SIG_PAR_TYPE`
  これはシグナルのタイプ（接頭辞として `ySignalPar` の付いた）です。シグナルにパラメータがない場合は、このマクロ パラメータは XSIGNALHEADERTYPE です。

INSIGNAL_NAME

このマクロは、現在受信されているシグナルの識別子に展開します。複数のシグナルが同じ入力シンボルで列挙されている場合に、シグナル間の区別に使用します。

OUTSIGNAL_DATA_PTR

これは、シグナル送信時のシグナルのビルド中にシグナル データ領域を参照するポインタです。値を ALLOC_SIGNAL または ALLOC_SIGNAL_PAR で割り当てる必要があります。これで、シグナル パラメータの割り当て時や SDL_2OUTPUT マクロで使用できます。

SDL_2OUTPUT

このマクロは SDL_ALT2OUTPUT_COMPUTED_TO と関連します。

SDL_2OUTPUT_NO_TO

このマクロは SDL_ALT2OUTPUT_COMPUTED_TO と関連します。

SDL_2OUTPUT_COMPUTED_TO

このマクロは SDL_ALT2OUTPUT_COMPUTED_TO と関連します。

SDL_ALT2OUTPUT

このマクロは SDL_ALT2OUTPUT_COMPUTED_TO と関連します。

SDL_ALT2OUTPUT_NO_TO

このマクロは SDL_ALT2OUTPUT_COMPUTED_TO と関連します。

SDL_ALT2OUTPUT_COMPUTED_TO

パラメータ：(PRIO, VIA, SIG_NAME, SIG_IDNODE, RECEIVER, SIG_PAR_SIZE, SIG_NAME_STRING)

SDL_*OUTPUT* という名前の 6 つのマクロは、ALLOC_SIGNAL または ALLOC_SIGNAL_PAR で生成されたシグナルの送信に使用します。マクロの SDL_ALT バージョンは、ディレクティブ /*#ALT*/ が signal sending で指定されている場合に使用します。接尾辞のないバージョンは直接アドレッシングを含む signal sending で使用し、接尾辞 _COMPUTED_TO は、コード生成時に受信側を計算できる場合の、直接アドレッシングを含まない signal sending で使用します。接尾辞 _NO_TO は、受信側がコード生成時に計算できない場合の、直接アドレッシングを含まない signal sending を示します。

- PRIO
  現在は未使用です。
- VIA
  signal sending で指定される via リストです。
- SIG_NAME
  接頭辞のないシグナル名です。
- SIG_IDNODE
  シグナルの xSignalIdNode です。
- RECEIVER
  指定された（<SENDER>.<OUTPUT>）か、計算された受信側です。NO_TO signal sending では、RECEIVER は xNotDefPId です。
- SIG_PAR_SIZE
  シグナルの構造体のサイズ（接頭辞として ySignalPar の付いた）です。パラメータを持たないシグナルの場合は、SIG_PAR_SIZE は 0 です。
- SIG_NAME_STRING
  文字列で表したシグナル名です。

SDL_THIS

受信側が THIS の signal sending では、ALLOC_SIGNAL マクロと SDL_2OUTPUT マクロの RECEIVER パラメータは SDL_THIS となります。

SIGCODE

パラメータ：(P)

このマクロにより、シグナルの xSignalIdNode にシグナル コード（シグナル番号）を格納できます。マクロ パラメータ P は、接頭辞のないシグナル名です。

SIGNAL_ALLOC_ERROR

このマクロは ALLOC_SIGNAL マクロと、シグナルへのパラメータ値の割り当ての後に挿入します。alloc が成功したかどうかをテストするのに使用できます。

SIGNAL_ALLOC_ERROR_END

このマクロは SDL_2OUTPUT マクロの後に挿入します。

SIGNAL_NAME

パラメータ：(SIG_NAME, SIG_IDNODE)

このマクロは、パラメータとして付与されているシグナルの識別子に展開します。通常、識別子はシグナルの xSignalIdNode か int 値です。id が int 値の場合、#define signal_name number タイプの定義を挿入するのが適切です。

- SIG_NAME
  パラメータを持たないシグナル名です。
- SIG_IDNODE
  シグナルの xSignalIdNode です。

SIGNAL_VARS

各シグナル インスタンスに必要な構造体コンポーネントです。送信側、受信側、シグナル タイプはこのようなコンポーネントの例です。

TO_PROCESS

パラメータ：(PROC_NAME, PROC_IDNODE)

このマクロは、シグナルがアクティブ クラス インスタンス セットに送信される場合、ALLOC_SIGNAL マクロや SDL_2OUTPUT マクロの RECEIVER として使用します。

- PROC_NAME
  接頭辞のない受信側アクティブ クラス名です。
- PROC_IDNODE
  受信側アクティブ クラスの xPrsIdNode です。

TRANSFER_SIGNAL

このマクロは、下記の TRANSFER_SIGNAL_PAR と関連します。

TRANSFER_SIGNAL_PAR

パラメータ：(SIG_NAME, SIG_IDNODE, RECEIVER, SIG_PAR_TYPE)

これらのマクロは、ディレクティブ #TRANSFER が signal sending で指定されている場合に ALLOC_SIGNAL マクロの代わりに使用します。

- SIG_NAME
  接頭辞のないシグナル名です。
- SIG_IDNODE
  シグナルの xSignalIdNode です。

- RECEIVER
  これは、直接アドレッシング節で指定されたか、計算された受信側です。NO_TO signal sending では、RECEIVER は xNotDefPId です。
- SIG_PAR_TYPE
  これはシグナルのタイプ（接頭辞として ySignalPar の付いた）です。シグナルにパラメータがない場合は、このマクロ パラメータは XSIGNALHEADERTYPE です。

XNONE_SIGNAL

これは none シグナルの表現です。

XSIGNALHEADERTYPE

このマクロは、パラメータを持たないシグナルの構造体（接頭辞として ySignalPar の付いた）を示すのに使用します。このようなシグナルには生成構造体がありません。SIGNAL_VARS のコンポーネントのみを含む構造体の名前は XSIGNALHEADERTYPE としておくのが適切です。

XSIGTYPE

使用されているシグナル タイプの表現（xSignalIdNode または int）により、このマクロは、xSignalIdNode または int となります。

リモート操作の呼び出しの実装

ALLOC_REPLY_SIGNAL

このマクロは ALLOC_REPLY_SIGNAL_PRD_PAR と関連します。

ALLOC_REPLY_SIGNAL_PAR

このマクロは ALLOC_REPLY_SIGNAL_PRD_PAR と関連します。

ALLOC_REPLY_SIGNAL_PRD

このマクロは ALLOC_REPLY_SIGNAL_PRD_PAR と関連します。

ALLOC_REPLY_SIGNAL_PRD_PAR

パラメータ：(SIG_NAME, SIG_IDNODE, RECEIVER, SIG_PAR_TYPE)

これらのマクロは、RPC のシグナル交換の応答シグナルを割り当てるのに使用します。接尾辞 _PAR は、応答シグナルにパラメータが含まれている場合に使用します。接尾辞 _PRD は、暗黙的 RPC 遷移がアクティブ クラスの操作の一部である場合に使用します。

- SIG_NAME
  接頭辞のない応答シグナル名です。
- SIG_IDNODE
  応答シグナルの xSignalIdNode です。
- RECEIVER
  応答シグナルの受信側です。XRPC_SENDER_IN_ALLOC マクロと XRPC_SENDER_IN_ALLOC_PRD マクロは、実パラメータとして使用します。接尾辞 _PRD は、暗黙的 RPC 遷移がアクティブ クラスの操作の一部である場合に使用します。
- SIG_PAR_TYPE
  応答シグナルのタイプ（接頭辞として ySignalPar の付いた）です。応答シグナルにパラメータが含まれていない場合、マクロ名 XSIGNALHEADERTYPE が実パラメータとして生成されます。

REPLYSIGNAL_DATA_PTR

このマクロは REPLYSIGNAL_DATA_PTR_PRD と関連します。

REPLYSIGNAL_DATA_PTR_PRD

これは、ALLOC_REPLY_SIGNAL マクロで割り当てられた応答シグナルのデータ領域の参照となります。接尾辞 _PRD は、暗黙的 RPC 遷移がアクティブ クラスの操作の一部である場合に使用します。

SDL_RPCWAIT_NEXTSTATE

このマクロは SDL_RPCWAIT_NEXTSTATE_PRD と関連します。

SDL_RPCWAIT_NEXTSTATE_PRD

パラメータ： (PREPLY_IDNODE, PREPLY_NAME, RESTARTADDR)

これらのマクロは、RPC の呼び出し側に暗黙的 nextstate 操作を実装するのに使用します。接尾辞 _PRD は、暗黙的 RPC 遷移がアクティブ クラスの操作の一部である場合に使用します。

- PREPLY_IDNODE
  応答シグナルの xSignalIdNode です。
- PREPLY_NAME
  接頭辞のない応答シグナル名です。
- RESTARTADDR
  応答シグナルの暗黙的入力の再開アドレス（シンボル番号）です。

SDL_2OUTPUT_RPC_CALL

パラメータ： (PRIO, VIA, SIG_NAME, SIG_IDNODE, RECEIVER, SIG_PAR_SIZE, SIG_NAME_STRING)

これは RPC の呼び出しシグナルを送信するマクロです。

- PRIO
  現在は未使用です。
- VIA
  via リストで、この場合は常に (xIdNode *)0、つまり via リストはありません。
- SIG_NAME
  接頭辞のない RPC 呼び出しシグナル名です。
- SIG_IDNODE
  RPC 呼び出しシグナルの xSignalIdNode です。
- RECEIVER
  呼び出しシグナルの受信側です。これは通常の直接アドレッシング式として表現するか、マクロを使用して呼び出しで明示的受信側が指定されていない場合は、XGETEXPORTINGPRS となります。
- SIG_PAR_SIZE
  呼び出しシグナルの構造体のサイズ（接頭辞として ySignalPar の付いた）です。呼び出しシグナルにパラメータがない場合は、このパラメータは 0 です。
- SIG_NAME_STRING
  文字列で表した RPC 呼び出しシグナル名です。

SDL_2OUTPUT_RPC_REPLY

このマクロは SDL_2OUTPUT_RPC_REPLY_PRD と関連します。

SDL_2OUTPUT_RPC_REPLY_PRD

パラメータ : (PRIO, VIA, SIG_NAME, SIG_IDNODE, RECEIVER, SIG_PAR_SIZE, SIG_NAME_STRING)

これらのマクロは、RPC 応答シグナルを送信するのに使用します。接尾辞 _PRD は、暗黙的 RPC 遷移がアクティブ クラスの操作の一部である場合に使用します。

- PRIO
  現在は未使用です。
- VIA
  via リストで、この場合は常に (xIdNode *)0、つまり via リストはありません。
- SIG_NAME
  接頭辞のない RPC 応答シグナル名です。
- SIG_IDNODE
  RPC 応答シグナルの xSignalIdNode です。
- RECEIVER
  応答シグナルの受信側です。これは、XRPC_SENDER_IN_OUTPUT マクロまたは XRPC_SENDER_IN_OUTPUT_PRD マクロを使用して表現します。
- SIG_PAR_SIZE
  応答シグナルの構造体のサイズ（接頭辞として ySignalPar の付いた）です。応答シグナルにパラメータがない場合は、このパラメータは 0 です。
- SIG_NAME_STRING
  文字列で表した RPC 応答シグナル名です。

XGETEXPORTINGPRS

パラメータ：(REMOTENODE)

このマクロは、アクティブ 実マクロ パラメータとしてアクティブ クラスのリモート操作（正確に言えばリモート操作の IdNode）が指定されれば、このリモート操作の「プロバイダ」を 1 つ返す式に展開する必要があります。通常このマクロは、ライブラリ関数 xGetExportingPrs の呼び出しに展開されます。

XRPC_REPLY_INPUT

このマクロは XRPC_REPLY_INPUT_PRD と関連します。

XRPC_REPLY_INPUT_PRD

RPC 応答シグナルの受信に必要な特殊処理に使用できるマクロです。通常、何にも展開しません。

XRPC_SAVE_SENDER

このマクロは XRPC_SAVE_SENDER_PRD と関連します。

XRPC_SAVE_SENDER_PRD

このマクロは、応答シグナルを送信する際に利用できるよう、受信 RPC 呼び出しシグナルの送信側を保存するのに使用できます。接尾辞 _PRD は、暗黙的 RPC 遷移がアクティブ クラスの操作の一部である場合に使用します。

XRPC_SENDER_IN_ALLOC

このマクロは XRPC_SENDER_IN_ALLOC_PRD と関連します。

XRPC_SENDER_IN_ALLOC_PRD

このマクロは、ALLOC_REPLY_SIGNAL マクロで応答シグナルの受信側を（呼び出しシグナルの送信側から）取得するのに使用します。接尾辞 _PRD は、暗黙的 RPC 遷移がアクティブ クラスの操作の一部である場合に使用します。

XRPC_SENDER_IN_OUTPUT

このマクロは XRPC_SENDER_IN_OUTPUT_PRD と関連します。

XRPC_SENDER_IN_OUTPUT_PRD

このマクロは、SDL_2OUTPUT_RPC_REPLY マクロで応答シグナルの受信側を（呼び出しシグナルの送信側から）取得するのに使用します。接尾辞 _PRD は、暗黙的 RPC 遷移がアクティブ クラスの操作の一部である場合に使用します。

XRPC_WAIT_STATE

RPC wait ステートに使用するステート番号です。XRPC_WAIT_STATE は通常 -3 と定義します。

静的および動的 **create** と **stop** の実装

ALLOC_STARTUP

このマクロは ALLOC_STARTUP_PAR と関連します。

ALLOC_STARTUP_PAR

パラメータ : `(PROC_NAME, STARTUP_IDNODE, STARTUP_PAR_TYPE)`

このマクロは起動シグナルのデータ領域を割り当てるのに使用され、STARTUP_DATA_PTR マクロに記述されているポインタがこのデータ領域を参照するようにします。接尾辞 _PAR は、起動シグナルにパラメータが含まれている場合に使用します。

- `PROC_NAME`
  接頭辞のない生成アクティブ クラス名です。
- `STARTUP_IDNODE`
  生成アクティブ クラスの起動シグナルの `xSignalIdNode` です。
- `STARTUP_PAR_TYPE`
  生成アクティブ クラスの起動シグナルのタイプ（接頭辞として `ySignalPar` の付いた）です。

ALLOC_STARTUP_THIS

このマクロは起動シグナルのデータ領域を割り当てるのに使用され、STARTUP_DATA_PTR マクロに記述されているポインタがこのデータ領域を参照するようにします。このマクロは create THIS 操作で使用します。

INIT_PROCESS_TYPE

パラメータ : `(PROC_NAME, PREFIX_PROC_NAME, PROC_IDNODE, PROC_NAME_STRING, MAX_NO_OF_INST, STATIC_INST, VDEF_TYPE, PRIO, PAD_FUNCTION)`

このマクロは、`yInit` 関数でアクティブ クラス インスタンス セット 1 つごとに 1 回呼び出します。アクティブ クラス インスタンス セットのすべてのインスタンスに共通の機能を開始するのに使用します。

- `PROC_NAME`
  接頭辞のないアクティブ クラス インスタンス セット名です。
- `PREFIX_PROC_NAME`
  接頭辞のあるアクティブ クラス インスタンス セット名です。

- PROC_IDNODE
  アクティブ クラス インスタンス セットの xPrsIdNode です。
- PROC_NAME_STRING
  文字列で表したアクティブ クラス インスタンス セット名です。
- MAX_NO_OF_INST
  このアクティブ クラス インスタンス セットの最大インスタンス数です。
- STATIC_INST
  このアクティブ クラス インスタンス セットの静的インスタンス数です。
- VDEF_TYPE
  このアクティブ クラス インスタンス セットの yVDef タイプです。
- PRIO
  現在は未使用です。
- PAD_FUNCTION
  このアクティブ クラス インスタンス セットの PAD です。

SDL_CREATE

パラメータ： (PROC_NAME, PROC_IDNODE, PROC_NAME_STRING)

このマクロは、アクティブ クラスのインスタンスの生成（create アクション）に使用します。

- PROC_NAME
  接頭辞のないアクティブ クラス インスタンス セット名です。
- PROC_IDNODE
  アクティブ クラス インスタンス セットの xPrsIdNode です。
- PROC_NAME_STRING
  文字列で表したアクティブ クラス インスタンス セット名です。

SDL_CREATE_THIS

このマクロは THIS の生成を実装するのに使用します。

SDL_STATIC_CREATE

パラメータ： (PROC_NAME, PREFIX_PROC_NAME, PROC_IDNODE, PROC_NAME_STRING, STARTUP_IDNODE, STARTUP_PAR_TYPE, VDEF_TYPE, PRIO, PAD_FUNCTION, BLOCK_INST_NUMBER)

このマクロは、yInit 関数で、生成する必要のある、アクティブ クラス インスタンス セットの「静的な」アクティブ クラス インスタンス 1 つごとに 1 回呼び出します。

- PROC_NAME
  接頭辞のないアクティブ クラス インスタンス セット名です。
- PREFIX_PROC_NAME
  接頭辞のあるアクティブ クラス インスタンス セット名です。
- PROC_IDNODE
  アクティブ クラス インスタンス セットの xPrsIdNode です。

- PROC_NAME_STRING
  文字列で表したアクティブ クラス インスタンス セット名です。
- STARTUP_IDNODE
  アクティブ クラス インスタンス セットの起動シグナルの xSignalIdNode です。
- STARTUP_PAR_TYPE
  アクティブ クラス インスタンス セットの起動シグナルのタイプ（接頭辞として ySignalPar の付いた）です。
- VDEF_TYPE
  このアクティブ クラス インスタンス セットの yVDef タイプです。
- PRIO
  現在は未使用です。
- PAD_FUNCTION
  このアクティブ クラス インスタンス セットの PAD 関数です。
- BLOCK_INST_NUMBER
  このアクティブ クラス インスタンス セットが、コンポジションを含むアクティブ クラス インスタンス セットの一部である場合、このマクロは、それが属するアクティブ クラス インスタンス セットのインスタンス番号です。そうでない場合、パラメータは 1 です。

SDL_STOP

このマクロは、アクティブ クラスに対する Stop 操作を実装するのに使用します。

STARTUP_ALLOC_ERROR

このマクロは ALLOC_STARTUP マクロとシグナルへのパラメータ値の割り当ての後に挿入します。alloc が成功したかどうかをテストするのに使用できます。

STARTUP_ALLOC_ERROR_END

このマクロは SDL_CREATE マクロの後に挿入します。

STARTUP_DATA_PTR

このマクロは、起動シグナル データ領域の参照を格納するのに使用する一時変数に展開します。ALLOC_STARTUP マクロで割り当て、起動シグナルに実シグナル パラメータ（仮パラメータ値）を割り当てるのに使用します。

STARTUP_VARS

このマクロは、起動シグナルに追加一般コンポーネントを挿入するのに使用できます。すべての起動シグナル構造で SIGNAL_VARS の後には STARTUP_VARS が続きます。

タイマー、タイマー操作、**now** の実装

ALLOC_TIMER_SIGNAL_PAR

パラメータ：(TIMER_NAME, TIMER_IDNODE, TIMER_PAR_TYPE)

このマクロは、パラメータを持つタイマー シグナルのデータ領域を割り当てます。

- TIMER_NAME
  接頭辞のないタイマー名です。
- TIMER_IDNODE
  タイマーの xSignalIdNode です。
- TIMER_PAR_TYPE
  タイマーのタイプ（接頭辞として ySignalPar の付いた）です。

DEF_TIMER_VAR

このマクロは DEF_TIMER_VAR_PARA と関連します。

DEF_TIMER_VAR_PARA

パラメータ：(TIMER_VAR)

アクティブ クラスが含む各タイマー宣言に対し、アクティブ クラスの yVDef タイプにこのマクロのアプリケーションが 1 つあります。これらの宣言は、コンポーネント（タイマー変数）を yVDef 構造体に導入してタイマーをトラックするのに使用できます。パラメータ TIMER_VAR は、このような変数に適切な名前です。接尾辞 _PARA は、タイマーにパラメータがある場合に使用します。

INIT_TIMER_VAR

このマクロは INIT_TIMER_VAR_PARA と関連します。

INIT_TIMER_VAR_PARA

パラメータ：(TIMER_VAR)

このマクロは、アクティブ クラス属性の初期化中に開始遷移に挿入されます。これで、DEF_TIMER_VAR マクロに挿入されるタイマー変数の初期化が可能になります。パラメータ TIMER_VAR がこのような変数の名前です。接尾辞 _PARA は、タイマーにパラメータがある場合に使用します。

INPUT_TIMER_VAR

このマクロは INPUT_TIMER_VAR_PARA と関連します。

INPUT_TIMER_VAR_PARA

パラメータ：(TIMER_VAR)

このマクロは、タイマー シグナルの受信時に挿入されます。これで、DEF_TIMER_VAR マクロに挿入されるタイマー変数の更新が可能になります。パラメータ TIMER_VAR がこのような変数の名前です。接尾辞 _PARA は、タイマーにパラメータがある場合に使用します。

注記

タイマー シグナルが input * 文で受信されると、INPUT_TIMER_VAR はありません。

RELEASE_TIMER_VAR

このマクロは RELEASE_TIMER_VAR_PARA と関連します。

RELEASE_TIMER_VAR_PARA

パラメータ： (TIMER_VAR)

このマクロは停止時に挿入されます。これで、DEF_TIMER_VAR マクロに挿入されるタイマー変数のクリーンアップが実行可能になります。パラメータ TIMER_VAR がこのような変数の名前です。接尾辞 _PARA は、タイマーにパラメータがある場合に使用します。

SDL_ACTIVE

パラメータ： (TIMER_NAME, TIMER_IDNODE, TIMER_VAR)

このマクロは、タイマーの Active 操作を実装するのに使用します。パラメータを持つタイマーの Active 操作は、C コード ジェネレータでは実装されません。

- TIMER_NAME
  接頭辞のないタイマー名です。
- TIMER_IDNODE
  タイマーの xSignalIdNode です。
- TIMER_VAR
  DEF_TIMER_VAR マクロに挿入されるタイマー変数です。

SDL_NOW

Now の実装です。

SDL_RESET

パラメータ： (TIMER_NAME, TIMER_IDNODE, TIMER_VAR, TIMER_NAME_STRING)

このマクロは、パラメータを持たないタイマーの Reset 操作を実装するのに使用します。

- TIMER_NAME
  接頭辞のないタイマー名です。
- TIMER_IDNODE
  タイマーの xSignalIdNode です。

- TIMER_VAR
  DEF_TIMER_VAR マクロに挿入されるタイマー変数です。
- TIMER_NAME_STRING
  文字列で表したタイマー名です。

SDL_RESET_WITH_PARA

パラメータ： (EQ_FUNC, TIMER_VAR, TIMER_NAME_STRING)

このマクロは、パラメータを持つタイマーの Reset 操作を実装するのに使用します。

- EQ_FUNC
  2 つのタイマー インスタンスが等しいかどうかをテストできる生成 equal 関数の名前です。
- TIMER_VAR
  DEF_TIMER_VAR マクロに挿入されるタイマー変数です。
- TIMER_NAME_STRING
  文字列で表したタイマー名です。

SDL_SET

パラメータ： (TIME_EXPR, TIMER_NAME, TIMER_IDNODE, TIMER_VAR, TIMER_NAME_STRING)

このマクロは SDL_SET_TICKS_WITH_PARA と関連します。

SDL_SET_WITH_PARA

パラメータ： (TIME_EXPR, TIMER_NAME, TIMER_IDNODE, TIMER_PAR_TYPE, EQ_FUNC, TIMER_VAR, TIMER_NAME_STRING)

このマクロは SDL_SET_TICKS_WITH_PARA と関連します。

SDL_SET_DUR

パラメータ： (TIME_EXPR, DUR_EXPR, TIMER_NAME, TIMER_IDNODE, TIMER_VAR, TIMER_NAME_STRING)

このマクロは SDL_SET_TICKS_WITH_PARA と関連します。

SDL_SET_DUR_WITH_PARA

パラメータ： (TIME_EXPR, DUR_EXPR, TIMER_NAME, TIMER_IDNODE, TIMER_PAR_TYPE, EQ_FUNC, TIMER_VAR, TIMER_NAME_STRING)

このマクロは SDL_SET_TICKS_WITH_PARA と関連します。

SDL_SET_TICKS

パラメータ： (TIME_EXPR, DUR_EXPR, TIMER_NAME, TIMER_IDNODE, TIMER_VAR, TIMER_NAME_STRING)

このマクロは SDL_SET_TICKS_WITH_PARA と関連します。

SDL_SET_TICKS_WITH_PARA

パラメータ : `(TIME_EXPR, DUR_EXPR, TIMER_NAME, TIMER_IDNODE, TIMER_PAR_TYPE, EQ_FUNC, TIMER_VAR, TIMER_NAME_STRING)`

接頭辞 SDL_SET を付けたこのマクロは、タイマーの Set 操作を実装するのに使用します。

接尾辞 _WITH_PARA は、パラメータを持つタイマーのセットを示します。この場合、SDL_SET マクロの前に ALLOC_TIMER_SIGNAL_PAR マクロの呼び出し、およびタイマー パラメータの割り当てを付けます。

接尾辞 _DUR は、set 操作のタイマー値を (now + 式) と表現する場合に使用します。この場合、時間値と継続時間値（上記の式）は両方ともマクロ パラメータとして使用できます。

接尾辞 _TICKS は、set 操作のタイマー値を (now + TICKS(...)) と表現する場合に使用します。TICKS は継続時間値を返す操作です。この場合、時間値と継続時間値は両方ともマクロ パラメータとして使用できます。

- `TIME_EXPR`
  時間式です。
- `DUR_EXPR`
  継続時間式です（_DUR と _TICKS のみで使用します）。
- `TIMER_NAME`
  接頭辞のない応答タイマー名です。
- `TIMER_IDNODE`
  タイマーの `xSignalIdNode` です。
- `TIMER_PAR_TYPE`
  タイマーの構造体です（_WITH_PARA でのみ使用します）。
- `EQ_FUNC`
  2 つのタイマーのパラメータ値が同じかどうかをテストするのに使用できる関数です（_WITH_PARA でのみ使用します）。
- `TIMER_VAR`
  DEF_TIMER_VAR マクロに導入されるタイマー変数の名前です。
- `TIMER_NAME_STRING`
  文字列で表したタイマー名です。

TIMER_DATA_PTR

これは、タイマーのビルド中にタイマー データ領域を参照するポインタです。値を ALLOC_TIMER_SIGNAL_PAR で割り当てる必要があります。これで、シグナル パラメータの割り当て時や SDL_SET マクロで使用できます。

TIMER_SIGNAL_ALLOC_ERROR

このマクロは ALLOC_TIMER_SIGNAL_PAR マクロとタイマーへのパラメータ値の割り当ての後に挿入します。`alloc` が成功したかどうかをテストするのに使用できます。

TIMER_SIGNAL_ALLOC_ERROR_END

このマクロは SDL_SET マクロの後に挿入します。

TIMER_VARS

各タイマー インスタンスに必要な構造体コンポーネントです。送信側、受信側、タイマー タイプはこのようなコンポーネントの例です。

タイマーは、いったん送信されるとシグナルと見なされるため、TIMER_VARS は SIGNAL_VARS と同じでなければなりません。ただし、新しいコンポーネントを TIMER_VARS の後わり、共通のコンポーネントの後に追加してもかまいません。

XTIMERHEADERTYPE

このマクロは、パラメータを持たないタイマーの構造体を示すのに使用します。このようなタイマーには、パラメータを持つ生成構造体がありません。TIMER_VARS のコンポーネントのみを含む構造体の名前は XTIMERHEADERTYPE としておくのが適切です。

呼び出しとリターンの実装

ALLOC_PROCEDURE

パラメータ：`(PROC_NAME, PROC_IDNODE, VAR_SIZE)`

これは、アクティブ クラスで呼び出された操作のデータ領域（`yVDef`）を割り当てるマクロです。

- `PROC_NAME`
  接頭辞のある操作名です。
- `PROC_IDNODE`
  呼び出された操作の `xPrdIdNode` です。
- `VAR_SIZE`
  操作の `yVDef` 構造体のサイズです。

ALLOC_THIS_PROCEDURE

これは、call THIS が使用された場合に操作のデータ領域（`yVDef`）を割り当てるマクロです。

ALLOC_VIRT_PROCEDURE

(PROC_IDNODE)

これは、仮想操作の呼び出し時、アクティブ クラスで呼び出された操作のデータ領域（yVDef）を割り当てるマクロです。PROC_IDNODE パラメータは、呼び出された操作の xPrdIdNode です。

CALL_PROCEDURE

このマクロは CALL_PROCEDURE_IN_PRD と関連します。

CALL_PROCEDURE_IN_PRD

パラメータ：(PROC_NAME, PROC_IDNODE, LEVELS, RESTARTADDR)

このマクロは SDL に呼び出し操作を実装するのに使用します。yVDef 構造体は（ALLOC_PROCEDURE で）すでに割り当てられており、実パラメータはこの構造体のコンポーネントに割り当てられています。接尾辞 _IN_PRD は、操作の呼び出しがアクティブ クラスの操作で行われることを示します。

- PROC_NAME
  接頭辞のある操作名で、これは操作の振る舞いを表す C 関数の名前と同じです。
- PROC_IDNODE
  呼び出された操作の xPrdIdNode です。
- LEVELS
  呼び出し側と呼び出された操作の間の範囲レベルです。
- RESTARTADDR
  これは、操作呼び出し後のシンボルの再開アドレスです。

CALL_PROCEDURE_STARTUP

このマクロは CALL_PROCEDURE_STARTUP_SRV と関連します。

CALL_PROCEDURE_STARTUP_SRV

このマクロは、PAD 関数 が遷移の終わりのリターンによって残される場合のみ有用です。その場合、アクティブ クラスに未実行操作があれば、状態機械が再度「アクティブ」になったときに再開する必要があります。

CALL_THIS_PROCEDURE

パラメータ：(RESTARTADDR)

このマクロは THIS 操作の呼び出しを実装するのに使用します。RESTARTADDR は、操作呼び出し後のシンボルの再開アドレスです。

CALL_VIRT_PROCEDURE

このマクロは CALL_VIRT_PROCEDURE_IN_PRD と関連します。

CALL_VIRT_PROCEDURE_IN_PRD

パラメータ：(PROC_IDNODE, LEVELS, RESTARTADDR)

このマクロは仮想操作の呼び出し操作を実装するのに使用します。yVDef 構造体は（ALLOC_VIRT_PROCEDURE で）すでに割り当てられており、実パラメータはこの構造体のコンポーネントに割り当てられています。接尾辞 _IN_PRD は、操作呼び出しがアクティブ クラスの操作で行われることを示します。

- PROC_IDNODE
  呼び出された操作の xPrdIdNode です。
- LEVELS
  呼び出し側と呼び出された操作の間の範囲レベルです。
- RESTARTADDR
  これは、操作呼び出し後のシンボルの再開アドレスです。

PROCEDURE_ALLOC_ERROR

このマクロは ALLOC_PROCEDURE マクロと、操作パラメータへのパラメータ値の割り当ての後に挿入します。alloc が成功したかどうかをテストするのに使用できます。

PROCEDURE_ALLOC_ERROR_END

このマクロは CALL_PROCEDURE マクロの後に挿入します。

PROC_DATA_PTR

このマクロは、操作データ領域の参照を格納するのに使用する一時変数に展開します。ALLOC_PROCEDURE マクロで割り当て、実操作パラメータを割り当てるのに使用します。

SDL_RETURN

Return の実装です。

XNOPROCATSTARTUP

このマクロを定義すると、CALL_PROCEDURE_STARTUP マクロに関して述べたすべてのコード（上記）が削除されます。

ジョインの実装

join 文は、通常 C では goto として実装されますが、これよりも複雑な実装が必要な場合が 1 つあります。これは、join で記述されたラベルがスーパー タイプにある場合です。

XJOIN_SUPER_PRS

パラメータ：(RESTARTADDR,RESTARTPAD)

このマクロは、下記の XJOIN_SUPER_SRV と関連します。

XJOIN_SUPER_PRD

パラメータ：(RESTARTADDR,RESTARTPRD)

このマクロは、下記の XJOIN_SUPER_SRV と関連します。

XJOIN_SUPER_SRV

パラメータ：(RESTARTADDR,RESTARTSRV)

接頭辞 XJOIN_SUPER_SRV の付いたこのマクロは、アクティブ クラスと操作のスーパー タイプとのジョインを表します（この順序で）。

- RESTARTADDR
  スーパー タイプの再開アドレスです。
- RESTARTPAD, RESTARTPRD, RESTARTSRV
  スーパー タイプの PAD 関数です。

ステートと次のステートの実装

注記

RPC 呼び出しの暗黙的な nextstate 操作は RPC セクションで取り上げます。

ASTERISK_STATE

アスタリスク ステートのステート番号です。ASTERISK_STATE は通常 -1 と定義します。

ERROR_STATE

エラー ステートに使用するステート番号です。ERROR_STATE は通常 -2 と定義します。

START_STATE

開始ステートのステート番号です。START_STATE は 0 と定義します。

START_STATE_PRD

アクティブ クラスにある操作の開始ステートのステート番号です。
START_STATE_PRD は 0 と定義します。

SDL_NEXTSTATE

パラメータ：(STATE_NAME, PREFIX_STATE_NAME, STATE_NAME_STRING)

所定ステートの状態機械の nextstate 操作です。

- `STATE_NAME`
  接頭辞のないステート名です。
- `PREFIX_STATE_NAME`
  接頭辞のあるステート名です。この識別子は生成コードの適切なステート番号として定義し、通常はステートの表現として使用します。
- `STATE_NAME_STRING`
  文字列で表したステート名です。

SDL_DASH_NEXTSTATE

状態機械の破線 nextstate 操作です。

SDL_NEXTSTATE_PRD

パラメータ：`(STATE_NAME, PREFIX_STATE_NAME, STATE_NAME_STRING)`

所定ステートの（アクティブ クラスの操作の）nextstate 操作です。

- `STATE_NAME`
  ステートの表現です。
- `PREFIX_STATE_NAME`
  接頭辞のあるステート名です。この識別子は生成コードの適切なステート番号として定義し、通常はステートの表現として使用します。
- `STATE_NAME_STRING`
  文字列で表したステート名です。

SDL_DASH_NEXTSTATE_PRD

アクティブ クラスの操作の破線 nextstate 操作です。

ANY 分岐の実装

2 つのパスを持つ ANY 分岐（非決定性分岐）は、次の構造に従って生成されます。

```
BEGIN_ANY_DECISION(2)
DEF_ANY_PATH(1, 2)
DEF_ANY_PATH(2, 0)
END_DEFS_ANY_PATH(2)
BEGIN_FIRST_ANY_PATH(1)
  statements
END_ANY_PATH
BEGIN_ANY_PATH(2)
  statements
END_ANY_PATH
END_ANY_DECISION
```

BEGIN_ANY_DECISION

パラメータ：`(NO_OF_PATHS)`

ANY 分岐の始まりです。NO_OF_PATHS は、分岐のパスの数です。

BEGIN_ANY_PATH

パラメータ：(PATH_NO)

ANY 分岐の実装部分のパス（最初でない）です。PATH_NO はパス番号です。

BEGIN_FIRST_ANY_PATH

パラメータ：(PATH_NO)

ANY 分岐の実装部分の考えられる最初のパスです。PATH_NO はパス番号です。

DEF_ANY_PATH

パラメータ：(PATH_NO, SYMBOLNUMBER)

分岐のパスの定義です。

- PATH_NO
  パス番号です。
- SYMBOLNUMBER
  このパスの最初のシンボルのシンボル番号です。

END_ANY_DECISION

ANY 分岐の終わりです。

END_ANY_PATH

実装部分のパスの 1 つの終わりです。

END_DEFS_ANY_PATH

パラメータ：(NO_OF_PATHS)

ANY 分岐の定義部分の終わりです。NO_OF_PATHS は、分岐のパスの数です。

インフォーマル分岐の実装

インフォーマル分岐の実装は ANY 分岐と似ています。

BEGIN_FIRST_INFORMAL_PATH

パラメータ：(PATH_NO)

インフォーマル分岐の実装部分の考えられる最初のパスです。PATH_NO はパス番号です。

BEGIN_INFORMAL_DECISION

パラメータ：(NO_OF_PATHS, QUESTION)

インフォーマル分岐の始まりです。

- NO_OF_PATHS
  分岐のパスの数です。
- QUESTION
  出力される文字列定数です。

BEGIN_INFORMAL_ELSE_PATH

パラメータ：(PATH_NO)

インフォーマル分岐の実装部分の else パスです。PATH_NO はパス番号です。

BEGIN_INFORMAL_PATH

パラメータ：(PATH_NO)

インフォーマル分岐の実装部分のパスです。PATH_NO はパス番号です。

DEF_INFORMAL_PATH

パラメータ：(PATH_NO, ANSWER, SYMBOLNUMBER)

インフォーマル分岐のパスの定義です。

- PATH_NO
  パス番号です。
- ANSWER
  回答文字列です。
- SYMBOLNUMBER
  このパスの最初のシンボルのシンボル番号です。

DEF_INFORMAL_ELSE_PATH

パラメータ：(PATH_NO, SYMBOLNUMBER)

インフォーマル分岐の else パスの定義です。

- PATH_NO
  パス番号です。
- SYMBOLNUMBER
  このパスの最初のシンボルのシンボル番号です。

END_DEFS_INFORMAL_PATH

パラメータ：(NO_OF_PATHS)

インフォーマル分岐の定義部分の終わりです。NO_OF_PATHS は、分岐のパスの数です。

END_INFORMAL_ELSE_PATH

実装部分の else パスの終わりです。

END_INFORMAL_DECISION

インフォーマル分岐の終わりです。

END_INFORMAL_PATH

実装部分のパスの 1 つの終わりです。

コンポーネント選択テストのマクロ

このセクションのマクロは、たとえば choice 変数のコンポーネント選択の妥当性をテストするものです。構造のオプション コンポーネントやポインタの参照解除のテストについてもここで説明します。

XCHECK_CHOICE_USAGE

パラメータ：(TAG, VALUE, NEQTAG, COMPNAME, CURR_VALUE, TYPEINFO)

このマクロは XSET_CHOICE_TAG_FREE と関連します。

XSET_CHOICE_TAG

パラメータ：(TAG, VALUE, ASSTAG, NEQTAG, COMPNAME, CURR_VALUE, TYPEINFO)

このマクロは XSET_CHOICE_TAG_FREE と関連します。

XSET_CHOICE_TAG_FREE

パラメータ：(TAG, VALUE, ASSTAG, NEQTAG, FREEFUNC, COMPNAME, CURR_VALUE, TYPEINFO)

接頭辞 XSET_CHOICE の付いたこのマクロは、choice 変数の暗黙的タグのテストと設定に使用します。XSET_CHOICE_TAG と XSET_CHOICE_TAG_FREE は、choice のコンポーネントに値が割り当てられるとタグを設定します。このマクロの FREE バージョンは、free 関数を持つコンポーネントが choice に含まれている場合に使用します。XCHECK_CHOICE_USAGE は、アクセス対象のコンポーネントがアクティブかどうかをテストするのに使用します。

- TAG
  暗黙的タグ コンポーネントです。
- VALUE
  新しいタグ値または想定されるタグ値です。
- ASSTAG
  タグ タイプの割り当て関数です。

- NEQTAG
  タグ タイプの equal テスト関数です。
- FREEFUNC
  `choice` タイプの free 関数です。
- COMPNAME
  文字列で表した選択コンポーネント名です。
- CURR_VALUE
  タグ タイプの現在の値です。
- TYPEINFO
  タグ タイプのタイプ情報ノードです。

XCHECK_OPTIONAL_USAGE

パラメータ： `(PRESENT_VAR, COMPNAME)`

このマクロは、選択オプション コンポーネントがあるかどうかをチェックするのに使用します。PRESENT_VAR パラメータはこのコンポーネントの現変数で、COMPNAME は文字列で表した選択コンポーネント名です。

XCHECK_REF

このマクロは XCHECK_OREF と関連します。

XCHECK_OWN

このマクロは XCHECK_OREF と関連します。

XCHECK_OREF

パラメータ： `(VALUE, REF_TYPEINFO, REF_SORT)`

これらのマクロは、ヌル ポインタ（Own または ORef を使用する）が参照解除されていないかどうかをテストするのに使用します。これらのマクロは、Own または ORef ポインタ参照解除を含む各文の前に挿入します。ORef ポインタの場合、ORef が有効かどうか、つまり現在のアクティブ クラスが所有するオブジェクトを参照しているかどうかもチェックされます。

- VALUE
  これはポインタの値です。
- REF_TYPEINFO
  参照対象ソートのタイプ情報ノードです。
- REF_SORT
  参照対象インスタンス化パッシブ クラスに対応する C タイプです。

XCHECK_OREF2

パラメータ： `(VALUE)`

ORef ポインタが有効なポインタ、つまり NULL であるか、現在のアクティブ クラスが所有するオブジェクトを参照しているかをチェックします。

デバッグとシミュレーションのマクロ

XAFTER_VALUE_RET_PRDCALL

パラメータ：(SYMB_NO)

これは、値を返す操作呼び出しの実装（暗黙的呼び出しシンボル）と、値を返す操作の呼び出しを含むシンボルの間に生成されるマクロです。

SYMB_NO は、値を返す操作呼び出しを含むシンボルのシンボル番号です。

XAT_FIRST_SYMBOL

パラメータ：(SYMB_NO)

これは、シグナル受信または開始シンボルと、遷移の最初のシンボルの間に生成されるマクロです。SYMB_NO は、遷移の最初のシンボルのシンボル番号です。

XAT_LAST_SYMBOL

nextstate 操作または stop 操作の直前に生成されるマクロです。

XBETWEEN_STMTS

XBETWEEN_STMTS_PRD と関連します。

XBETWEEN_STMTS_PRD

パラメータ：(SYMB_NO, C_LINE_NO)

アクションの文と文の間に生成されるマクロです。接尾辞 _PRD は、これらの文がアクティブ クラスの操作の一部であることを示します。

- SYMB_NO
  次の文のシンボル番号です。
- C_LINE_NO
  この文の C のライン番号です。

XBETWEEN_SYMBOLS

XBETWEEN_SYMBOLS_PRD と関連します。

XBETWEEN_SYMBOLS_PRD

パラメータ：(SYMB_NO, C_LINE_NO)

遷移のシンボルとシンボルの間に生成されるマクロです。接尾辞 _PRD は、これらのシンボルがアクティブ クラスの操作の一部であることを示します。

- SYMB_NO
  次のシンボルのシンボル番号です。
- C_LINE_NO
  この文の C コードのライン番号です。

XDEBUG_LABEL

パラメータ： (LABEL_NAME)

このマクロにより、遷移の初めにラベルが挿入できます。このようなラベルはデバッグ時に便利です。LABEL_NAME パラメータは、ステート名とシグナル名の連結です。

例 **390**

以下の文では、「*」

「STATE *;」

「INPUT *;」

の代わりに、名前 ASTERISK が表示されます。

```
state State1; input Sig1;
state State2; input *;
state *; input Sig2;
```

これらの文の生成コードには、以下のマクロがあります。

```
XDEBUG_LABEL(State1_Sig1)
XDEBUG_LABEL(State2_ASTERISK)
XDEBUG_LABEL(ASTERISK_Sig2)
```

ラベルを導入するのに適切なマクロ定義は次のようになります。

```
#define XDEBUG_LABEL(L)  L: ;
```

XOS_TRACE_INPUT

パラメータ： (SIG_NAME_STRING)

このマクロは input 文として生成され、たとえば受信シグナルに関するトレース情報の生成に使用できます。SIG_NAME_STRING パラメータは、シグナル名です。

YPRSNAME_VAR

パラメータ： (PRS_NAME_STRING)

このマクロは、アクティブ クラスの PAD 関数中の変数の宣言間で生成されます。たとえば、アクティブ クラスの名前を含む Cchar * 変数の宣言に使用できます。このような変数はデバッグ時に便利です。PRS_NAME_STRING パラメータは、文字列で表したアクティブ クラス名です。

YPRDNAME_VAR

パラメータ：(PRD_NAME_STRING)

このマクロは、アクティブ クラスの操作に対する PRD 関数中の変数の宣言間で生成されます。たとえば、操作の名前を含む char* 変数の宣言に使用できます。このような変数はデバッグ時に便利です。PRD_NAME_STRING パラメータは、文字列で表した操作名です。

挿入されるユーティリティ マクロ

次のマクロのシーケンスを挿入します。このほとんどは、他のスイッチの組み合わせが使用されているために使用されない（IdNodes 中の）構造体コンポーネントの削除に関するものです。

```
#define NIL 0
#define XXFREE xFree
#define XSYSD xSysD.

#ifdef XTESTF
#define xTestF(p)  , p
#else
#define xTestF(p)
#endif

#ifdef XREADANDWRITEF
#define xRaWF(p)  , p
#else
#define xRaWF(p)
#endif

#ifdef XFREEFUNCS
#define xFreF(p)  , p
#else
#define xFreF(p)
#endif

#ifdef XFREESIGNALFUNCS
#define xFreS(p)  , p
#else
#define xFreS(p)
#endif

#define xAssF(p)
#define xEqF(p)



#ifdef XIDNAMES
#define xIdNames(p)  , p
#else
#define xIdNames(p)
#endif

#ifndef XOPTCHAN
#define xOptChan(p)  , p
#else
#define xOptChan(p)
```

```
#endif

#ifdef XBREAKBEFORE
#define xBreakB(p)  , p
#else
#define xBreakB(p)
#endif

#ifdef XGRTRACE
#define xGRTrace(p)  , p
#else
#define xGRTrace(p)
#endif

#ifdef XMSCE
#define xMSCETrace(p)  , p
#else
#define xMSCETrace(p)
#endif

#ifdef XTRACE
#define xTrace(p)  , p
#else
#define xTrace(p)
#endif

#ifdef XCOVERAGE
#define xCoverage(p)  , p
#else
#define xCoverage(p)
#endif

#ifdef XNRINST
#define xNrInst(p)  , p
#else
#define xNrInst(p)
#endif

#ifdef XSYMBTLINK
#define xSymbTLink(p1, p2) , p1, p2
#else
#define xSymbTLink(p1, p2)
#endif

#ifdef XCTRACE
#define xCTrace(p)  p,
#define xCTraceS(p)  p;
#else
#define xCTrace(p)
#define xCTraceS(p)
#endif

#if !defined(XPMCOMM) && !defined(XENV)
#define xGlobalNodeNumber() 1
#endif

#define xSizeOfPathStack 50

#ifndef xOffsetOf
#define xOffsetOf(type, field) ¥
```

```
        ((xptrint) &((type *) 0)->field)
#endif
#define xToLower(C) ¥
        ((C >= 'A' && C <= 'Z') ? ¥
        (char)((int)C - (int)'A' + (int)'a') : C)

#define xbool int
```

MAX_READ_LENGTH

これは、ソートの値を読み込むのに使用する `char *` バッファの長さを制御するマクロです。大きいデータ型が使用されている場合、バッファのサイズを、デフォルト サイズ（10,000 バイト）からより適切なサイズに再定義できます。

```
#ifndef MAX_READ_LENGTH
#define MAX_READ_LENGTH 5000
                    /* max length of input line */
#endif
```

SDL_NULL

Pid タイプのヌル値です。

xNotDefPId

これは SDL_2OUTPUT マクロで RECEIVER パラメータとして使用します。1056 ページの「シグナルとシグナル送信の実装」セクションに、このマクロの使用方法の例があります。

スレッド インテグレーションのマクロ

以下のマクロはスレッド インテグレーションのみに使用します。

THREADED

カーネルの詳細を定義するスレッド インテグレーション モデルのメイン マクロです。

THREADED_GLOBAL_VARS

グローバル変数定義です。

THREADED_GLOBAL_INIT

セマフォなどのグローバル変数の初期化です。

THREADED_THREAD_VARS

スレッド変数の定義です。

THREADED_THREAD_INIT

スレッド変数の初期化です。

THREADED_THREAD_BEGINNING

`xInitSem` のリリースを待ちます。

THREADED_LOCK_INPUTPORT

セマフォを取得することにより入力キューを保護します。

THREADED_UNLOCK_INPUTPORT

入力キューのためにセマフォをリリースします。

THREADED_WAIT_AND_UNLOCK_INPUTPORT

次のメッセージやシグナルが到着するか、次の内部タイマーが切れるのを待ちます。

THREADED_SIGNAL_AND_UNLOCK_INPUTPORT

シグナルを送信し、入力キューのためにセマフォをリリースします。

THREADED_LISTREAD_START

グローバル アクティブ リストおよび使用可能リストを、読み込む前にセマフォで保護します。

THREADED_LISTWRITE_START

グローバル アクティブ リストおよび使用可能リストを、書き込む前にセマフォで保護します。

THREADED_LISTACCESS_END

グローバル アクティブ リストまたは使用可能リストを保護するセマフォをリリースします。

THREADED_EXPORT_START

セマフォを取得することにより、グローバル データに対するアクセスとアクションを保護します。

THREADED_EXPORT_END

グローバル データに対するアクセスやアクションの後にセマフォをリリースします。

THREADED_START_THREAD

新しいスレッドを開始します。

THREADED_STOP_THREAD

スレッドを終了します。

THREADED_AFTER_THREAD_START

新たに生成されたスレッドの起動を同期させます。

31

AgileC コード ジェネレータ リファレンス

AgileC コード ジェネレータは、組込みシステムのアプリケーション開発での使用を目的としています。もちろん、他のタイプのアプリケーション向けに AgileC コード ジェネレータ を使用することはできますが、ジェネレータの機能は組込みシステム開発時に使用するように設計されています。

AgileC コード ジェネレータによって生成されたアプリケーションは、ベアとスレッドの 2 つのモードで実行できます。

ファイル構造

AgileC コード ジェネレータ によって生成される 1 つのアプリケーションは、複数のファイルから構成されます。

生成されたコードには、UML モデルによって記述されるシステムの内容と振る舞いが反映されます。コードは、ターゲット ディレクトリ（または、ターゲット ディレクトリのサブディレクトリ）に格納される、複数の `.c` ファイルと `.h` ファイルから構成されます。

生成されたコードのほかに、アプリケーションの構築をサポートする多数のファイルが存在します。このセクションでは、これらのファイルとその取扱いの方法について解説します。

必須ファイル

ターゲット ディレクトリに含まれるファイル

UML モデルから生成された C ファイルの格納先ディレクトリには、コード生成オプションに基づいて、以下の接尾辞と拡張子を持つファイルが入っています。

- `.m`

  UML モデルから生成された一組のファイルをコンパイルするための make ファイルです。
- `_env.tpm`

  他の `.c` ファイルをインクルードするために使用するテンプレート make ファイルです。AgileC コード ジェネレータは、アーティファクトの［Generate Environment Template Functions］の指定にしたがって、環境関数を含むファイルを生成すると同時にこのテンプレートを生成します。したがって、このテンプレートを修正して新しい名前で保存すれば、ジェネレータによって上書きされなくなります。コンパイル時にこの修正済みテンプレートファイルを使用するように指定できます。
- `.ifc` ファイル

  システム レベルのすべての重要な宣言を格納しているファイルです。このファイルは、環境関数を実装するファイルのような外部コードから、生成された C コード内のオブジェクトにアクセスするために、使用されます。
- `_env.c`

  このファイルには環境関数用のテンプレートが含まれています。環境関数は、UML モデルをその環境に接続するために使用します。多くの場合、環境関数は、多数のマクロ定義を含むユーザー指定ファイルによって実装されます。このケースでは、`_env.c` ファイルは、そのまま再生成して使用できます。一方、生成した環境テンプレートの構造を修正する必要がある場合は、環境テンプレートを新しい名前でコピーして、必要な修正を行います。このケースでは、`_env.tpm` ファイルも変更して、修正した環境テンプレートファイルをコンパイルする必要があります。AgileC コード ジェネレータは、アーティファクトの［Generate Environment Template Functions］の指定にしたがって、このファイルを生成します。

- `auto_cfg.h`

  AgileC コード ジェネレータによって生成される設定ファイルです。このファイルには、データやプログラムのサイズと、どの UML 構造体を使用しているかに関する情報があります。この情報は、一部のデータ構造のサイズを決定するため、および、アプリケーションでは不要なデータとコードを排除するために使用します。
- `uml_cfg.h`

  Application Builder によって生成される設定ファイルです。このファイルには、アプリケーション構築とコード生成時に使用する、ビルド オプションに関する情報が含まれています。
- `.o / .obj`

  コンパイル後、ターゲット ディレクトリにはオブジェクト ファイル（通常は、`.o` または `.obj`）と実行形式ファイル（通常は、`.sct`）も格納されます。

カーネル ディレクトリに含まれるファイル

多数のファイルがカーネル ディレクトリにあります。通常は次の場所です。
`<installation_dir>/addins/sdlkernels/agilec/kernel`

- `uml_kern.c` および `uml_kern.h`

  ランタイム カーネルの基本的な実装です。ファイル `uml_kern.h` は、すべての `.c` ファイルからインクルードされます。
- `sctpred.c` および `sctpred.h`

  定義済みのデータ型をサポートするための実装コードです。

RTOS ディレクトリに含まれるファイル

カーネル ディレクトリ下の RTOS サブディレクトリ内に、サポートされる RTOS（リアルタイム オペレーティング システム）インテグレーションコードがあります。インテグレーションごとに、個別のサブディレクトリがあります。

- `rtapidef.h` および `rtapidef.c`

各インテグレーションには `rtapidef.h` と `rtapidef.c` の 2 つのファイルが格納されています。現在、以下のインテグレーションがサポートされています。

- POSIX pthreads インテグレーション（サブディレクトリ POSIX）
- Win32 インテグレーション（サブディレクトリ Win32）

実行形式ファイルの構築に重要な 2 つの追加ファイルがあります。これらのファイルは comp.opt と、makeoptions または make.opt です。

- comp.opt

  ビルド アーティファクトのプロパティの［Target Kind］で値を選択すると、実際には、特定のディレクトリが選択されたことになります。AgileC コード ジェネレータは、そのディレクトリにある comp.opt ファイルを読み、その内容から「makefile」ファイルの生成方法を決定します。

  comp.opt には、コンパイラ、リンカーなどを呼び出すためのテンプレートも入っています。

- makeoptions または make.opt

  このファイルにはコンパイル スイッチと、カーネルのコンパイル方法に関する情報が含まれています。このファイルは、開発アプリケーション用に生成される「makefile」にインクルードされます。ファイル名とインクルードのための構文は、comp.opt の先頭に定義されています。

C ファイルのインクルード構造

uml_kern.h

最上位の定義は、uml_kern.h ファイルにあります。このファイルは、すべての .c ファイルからインクルードされます。そして、下記に示した複数の .h ファイルをインクルードします。

以下のファイルをインクルードします。

- string.h、stdlib.h、limits.h などの標準の C ヘッダー ファイル。
- uml_cfg.h
- auto_cfg.h
- comphdef.h（コンパイラ／ハードウェア インテグレーション）
- rtapidef.h（ランタイム API インテグレーション）
- sctpred.h

uml_kern.c

以下のファイルをインクルードします。したがって、最上位の要素である uml_kern.c のコンパイル時に、カーネル全体がコンパイルされます。

- uml_kern.h
- rtapidef.h（ランタイム API インテグレーション）（選択した場合）
- sctpred.c

環境関数

概要

実際のアプリケーションは、ソフトウェアまたはハードウェアなどの物理的な外部環境とともに動作します。UML モデルは、外部環境との相互作用環境（シグナルの送受信、またはリモート プロシージャ呼び出しなど）を高い抽象レベルで記述しますが、実行時に現実に発生する事象について記述しません。たとえば、実際には、UML システムから送信されるシグナルは、ハードウェア レジスタを設定したり、インターネットを通して TCP/IP パケットを送信したりします。こういった外部環境との相互作用は、**環境関数**によって実装されます。

相互作用の発生という事象を、実装から切り離すと、モデルの全体の振る舞いを理解しやすくなります。また、そうすることで、環境を完全に制御できる状態を作れば、モデルのシミュレーションや検証も楽になります。逆に、たとえば、ターゲット プラットフォーム上でシミュレーションを行おうとしても、ハードウェアへのアクセスができないために、最適なシミュレーションはできません。

以下の環境関数を利用できます。

```
extern void xInitEnv (void);
extern void xCloseEnv (void);
extern void xOutEnv (xSignal *);
extern void xInEnv (void);
```

- `xInitEnv` と `xCloseEnv` は、環境の初期化と環境との接続の終了時に呼び出されます。
- `xOutEnv` は、シグナルがシステムから環境へ送信されるときに、ランタイム カーネルによって呼び出されます。シグナルがシステムから送信されるときに必要なアクションを実装します。
- `xInEnv` は、環境内でのイベント発生によって、シグナルがシステムに送信されたときに、呼び出されます。動作の詳細は、以下に説明するように、状況によって異なります。

環境関数の用途は、どの環境関数を使用するかを Application Builder で指定することによって制御します。具体的には、アーティファクトのプロパティ設定にしたがって、以下のマクロ定義を、Application Builder から生成したファイル `uml_cfg.h` に組み込みます。

```
#define USER_CFG_USE_xInitEnv
#define USER_CFG_USE_xCloseEnv
#define USER_CFG_USE_xOutEnv
#define USER_CFG_USE_xInEnv
```

xInitEnv

この関数は、環境を初期化するために使用します。この関数は、アプリケーションの初期化時に、`uml_kern.c` 内の関数 `xInit` から 1 回だけ呼び出されます。

xCloseEnv

この関数は、環境を終了するために使用します。この関数は、uml_kern.c の main 関数から 1 回だけ呼び出されます。

注記

RTOS インテグレーションでは、この関数が、RTOS タスクを終了するのに適さない場合があります。OS スレッドを適切に終了する方法については、RTOS の技術ドキュメントを参照してください。

xOutEnv

xOutEnv 関数は、環境へ送信されるシグナルへのポインタを渡します。この関数は、uml_kern.c 内のシグナル送信関数（xOutput と xOutputSimple）から呼び出されます。この関数には、パラメータとして渡されるシグナルが環境へ送信されるときに必要なアクションを、すべて記述します。

この関数では、ポインタで実装されたすべてのシグナル パラメータのメモリを解放する必要があります。ただし、シグナル自身の取り扱いは、呼び出し側関数によって実行されます。xOutEnv 関数用に生成されたテンプレート（下記参照）には、メモリ管理のための適切なコードが自動的に挿入されます。xOutEnv の例は、テンプレート環境関数に関するセクションにもあります。

xInEnv

xInEnv 関数は、環境からアプリケーションへ送信されるシグナルを取り扱うための 1 つの方法です。この関数が最適かどうかは、実際のアプリケーションとインテグレーションのモードに依存します。uml_kern.c では、システムに送られたシグナルを処理する方法や場所と関係なく、以下の 2 つの関数を使用します。

```
extern xSignal *xGetSignal (xSignalId, int, xSignal **);
extern void xENVOutput (xSignal *, xuint8, SDL_Pid);
```

まず、xGetSignal はシグナル データ領域へのポインタを取得します。次に、シグナル パラメータに値が割り当てられ、最後に、xENVOutput 関数を使用してシグナルが送信されます。

xGetSignal パラメータ

- 第一パラメータ：シグナル ID（番号）
- 第二パラメータ：データ領域のサイズ。システムが、パラメータを持つシグナル、タイマー、およびパートが含まない場合、このパラメータはまったく使用されません。
- 第三パラメータ：常時 0 に設定された最適化パラメータです。

xENVOutput パラメータ

- 第一パラメータ：xGetSignal によって取得されるシグナルへの参照。
- 第二パラメータ：シグナルの優先順位。シグナル優先順位が使用可能でない場合、このパラメータは除外されます。
- 第三パラメータ：シグナルの受信側。

以下のセクションで、これらのパラメータの設定方法の詳細について説明します。

アプリケーションへ送信するシグナルの実装

アプリケーションへ送信するシグナルの実装方法には、主に 3 つの方法があります。

xInEnv を使用しない送信

まず最初は、xInEnv 関数を使用しないケースです。例として、**ベア** インテグレーション（RTOS がない）のケースを考えます。この場合、上述の関数を使用して、シグナルを割り込みルーチン内からシステムへ直接送信できます。シグナルとシグナルキューのデータ構造を保護するために、割り込み禁止と割り込み許可を切り替えるための関数やマクロを実装する必要があります。

ベア インテグレーションで **xInEnv** を使用して送信

2 番目の例も、**ベア**インテグレーションです（RTOS がない）ケースです。ただし、今度は、シグナルを割り込みルーチン内から直接には送信しません。代わりに、割り込みルーチンは、外部イベントに関する情報を記憶するグローバル データ構造を設定するために使用されます。次に、xInEnv 関数を使用してシグナルが実際に送信されます。xInEnv は、システムによって実行される遷移と遷移の間に、スケジューラ（uml_kern.c 内の関数 xMainLoop）から繰り返し呼び出されます。各呼び出しで、xInEnv 関数は外部イベントを調べ、適切なシグナルをシステムへ送信します。xInEnv 関数はできるだけ早くスケジューラへ戻ります。外部イベントの記憶に使用されるデータ構造の保護は、ユーザーの責任です。この場合、AgileC コード ジェネレータのカーネルのデータ構造を保護する必要はありません。

RTOS インテグレーションで **xInEnv** を使用してシグナルを送信

3 番目の例は RTOS インテグレーションのケースです。この場合は、xInEnv を独自のスレッドで実行します。定義済みインテグレーションでは、メイン スレッドは、他のスレッドを作成した後に、xInEnv を実行します。この場合の xInEnv 関数は以下のようになります。

```
while (1) {
  wait for an event;
  if (event corresponding to signal 1) {
    send signal 1;
  }
  if (event corresponding to signal 2) {
    send signal 2;
  }
  and so on;
}
```

「wait for an event」は、何も実行するものがないとき、このスレッドを保留します。それ以外の場合、このスレッドはいつでも実行できます。アプリケーションへのシグナルの送信を引き起こす事象が発生したとき、この事象を検出したコードはグローバル データ領域に情報を格納し、xInEnv 関数を再起動するコードを実行します。単純な場合には、セマフォを使用して xInEnv を取り扱うことができます。

「wait for an event」を適切な時間のスリープとして実装して、上記の構造でポーリングによる解決方法をとることもできます。各ポーリング動作時に、xInEnv は何らかの外部イベントが発生したかどうかをチェックし、対応するシグナルを送信します。

インターフェイス ヘッダー ファイル（.ifc）

.ifc ファイルは、システム内部で定義されたエンティティに関する情報を格納する、システム インターフェイス ヘッダー ファイルです。このようなエンティティについて、コードが生成されます。これらの定義の一部を、環境関数と同様に、外部コードで利用できます。あるシステムについて生成されたコードでは、C 言語上で一意の名前になるように、すべての UML 名に接頭辞または接尾辞が付きます。例外は、.ifc ファイルです。.ifc ファイルでは、UML 名には定義済みの接頭辞が付きます。

.ifc ファイルには、システムのパートの名前が含まれています。xInEnv 関数でシグナルを送信するとき、この名前を受信側として使用できます。

生成された環境関数

<systemname>_env.c ファイルの先頭に、以下の文があります。

```
#include “uml_kern.h”
#ifdef XENV_INC
#include XENV_INC
#endif
#include “exenv.ifc”
```

XENV_INC をファイル名として定義することによって、ユーザー定義ファイルのインクルードが可能になります。このマクロは、Application Builder が生成する uml_cfg.h ファイル内で定義するのが最適です。次のような行を挿入します。

```
#define XENV_INC “my_defines.h”
```

この方法を使うと、インクルードしたファイル上のマクロ定義にしたがって、生成された環境関数のスケルトンを埋めてゆくことができます。この方法を使えば、環境関数を再生成したときに、手動で追加したコードが上書きされる恐れはありません。

xInitEnv と xCloseEnv の構造

xInitEnv および xCloseEnv 関数の構造は以下のとおりです。

```
extern void xInitEnv(void)
{
  /* Code to initialize the environment may be
     inserted here */
  XENV_INIT
}

extern void xCloseEnv(void)
{
  /* Code to bring down the environment in a controlled
     manner may be inserted here. */
  XENV_CLOSE
}
```

ここで、XENV_INIT と XENV_CLOSE は、前もって定義していなければ、空のマクロです。通常は、これらのマクロの内容は、初期化と終了時に必要な一連のステートメントと関数呼び出しです。

関数 `xInitEnv` は下記のコードによって囲まれます。

```
#ifdef USER_CFG_USE_xInitEnv
#endif
```

このコードは、関数が必要であると指定された場合のみ、関数をコンパイルするためのものです。このマクロは、Application Builder が `uml_cfg.h` ファイルに設定します。

xOutEnv の構造

生成された `xOutEnv` 関数の構造は以下のとおりです。

```
extern void xOutEnv(xSignal *SignalOut)
{
  OUT_START_CODE

  /* Signal s1 */
  #ifndef OUT_SIGNAL_sig_s1
    #define OUT_SIGNAL_sig_s1
  #endif
  if (SignalOut->Sid == sig_s1) {
    OUT_SIGNAL_sig_s1
    return;
  }

  /* Signal s2 */
  #ifndef OUT_SIGNAL_sig_s2
    #define OUT_SIGNAL_sig_s2(P1, P2)
  #endif
if (SignalOut->Sid == sig_s2) {
    OUT_SIGNAL_sig_s2(
      ((ySignalPar_sig_s2 *)SignalOut)->Param1,
      ((ySignalPar_sig_s2 *)SignalOut)->Param2)
    xFreeSignalPara(SignalOut);
    return;
  }
}
```

ここで、OUT_START_CODE は、前もって定義していなければ、空のマクロです。上のように、コードは一連の **if** 文から構成されます。各 **if** 文が、環境に対して送信できるシグナル型ごとに処理を行います。各シグナルについて、**if** 条件式は、シグナル ID をテストしています。

対応する **if** 文が見つかると、マクロ `OUT_SIGNAL_signalname` によって定義されているステートメントが実行されます。このマクロはシグナル パラメータごとに 1 つのパラメータを持ちます。また、定義がなければマクロは空です。つまり、もし OUT_SIGNAL マクロが定義されていなければ、コンパイルはされても何も実行しないことになります。この仕組みによって、インクリメンタルな開発方法が可能になります。つまり、すべてのシグナルの処理を実装しなくても、実装済みのシグナル処理のテストを始めることができます。

上記の例のコードでは、.ifc ファイルで使用されるシグナルの名前が sig_%n であると前提にしています。ここで、%n は UML でのシグナル名です。シグナル s2 のセクションで xFreeSignalPara 関数呼び出しが必要になるのは、シグナル パラメータを動的メモリ割り当てを使用して実装したため、そのパラメータの free 操作が必要になる場合です。シグナル自身のデータ領域は、uml_kern.c 内の xOutEnv を呼び出す場所で取り扱われます。

xInEnv の構造

生成された xInEnv 関数の構造は、以下のとおりです。下記の例は、**ベア** インテグレーション向けの例です。

```
extern void xInEnv (void)
{
  xSignal *SignalIn;
  IN_START_CODE

  /* Signal s1 */
  #ifdef IN_SIGNAL_sig_s1
    if (IN_SIGNAL_sig_s1) {
      SignalIn = xGetSignal(sig_s1, 0, 0);
      xENVOutput(SignalIn, IN_RECEIVER_s1);
    }
  #endif

/* Signal s2 */
  #ifdef IN_SIGNAL_sig_s2
    if (IN_SIGNAL_sig_s2) {
      SignalIn = xGetSignal(sig_s2,
                        sizeof(ySignalPar_sig_s2), 0);
      ((ySignalPar_sig_s2 *)SignalOut)->Param1 =
        IN_PARA1_sig_s2;
      yAssF_s_7(
        ((ySignalPar_sig_s2 *)SignalOut)->Param2,
        IN_PARA2_sig_s2,
        XASS_MR_ASS_NF);
      xENVOutput(SignalIn, IN_RECEIVER_s2);
    }
  #endif

  IN_END_CODE
}
```

ここで、マクロ IN_START_CODE と IN_END_CODE は、ユーザーが定義していなかった場合は空となります。

各シグナルは 4 つのステップで処理されます。

- 処理を有効化するマクロ IN_SIGNAL_signalname。このマクロは、**if** 文の条件判断で使用され、システムへのシグナル送信を引き起こす外部イベントが発生したかどうかを検査します。
- SignalIn 変数に新しいシグナル データ領域が割り当てられる。
- シグナル パラメータがある場合は、それらに値を埋め込みます。各パラメータの値は、該当するマクロ IN_PARA1_signalname、IN_PARA2_signalname などを使用して定義します。

- シグナルは、xENVOutput を呼び出すことによって送信されます。ここで、シグナルの受信側をマクロ IN_RECEIVER_signalname によって与えなければなりません。適切な値は、.ifc ファイルに以下のように定義されています。

  ```
  #define xPartNo_Partname  <integer number>
  ```

上記の仕組みによって、インクリメンタル開発が可能になります。つまり、処理を有効化するマクロが定義されている場合にのみ、そのシグナルについての処理ロジックが存在するので、そのシグナルについては先行してテストを行えるからです。

スレッド アプリケーションについても、構造は同様です。下の例では、関数にはループが含まれます。

```
extern void xInEnv (void)
{
  xSignal *SignalIn;
  IN_START_CODE
  while (1) {
    IN_WAIT_FOR_ACTION
/* To avoid that the thread running xInEnv takes
       all resources it should wait on for example a
       semaphore until something occurs that should
       cause a signal to be sent into the system */

    /* Here the code for the signals is placed in the
       same way as in the previous example. */

  }

IN_END_CODE
}
```

注記

IN_WAIT_FOR_ACTION の実装は、上の「wait for an event」の説明を参照して行ってください。

アプリケーションのコンパイルとリンク

必須ファイル

AgileC コード ジェネレータのコンパイルとリンクに使用される必須ファイルを、以下に説明します。

- `comp.opt`：コンパイルやリンクなどのコマンドを定義します。
- `make.opt` または `makeoptions`：コンパイラ フラグとカーネルのコンパイルが含まれています。このファイルは、生成された make ファイルからインクルードされます。
- `<systemname>.m`：アプリケーションのために生成された make ファイルです。
- `<systemname>_env.tpm`：コード ジェネレータの管理下にないファイルのために生成された、テンプレート make ファイルです。このファイルは、テンプレート環境関数が作成される場合に生成されます。

comp.opt

`comp.opt` ファイルは、ビルド プロセス全体の管理を行うために使用します。ビルド プロセスがどの `comp.opt` ファイルを参照するかは、Application Builder で選択した［Target Kind］値にしたがいます。コード ジェネレータは、コード生成時にこのファイルを読み込み、指定された情報を使って、ビルドプロセスを制御します。

`comp.opt` ファイルは、5 つの重要な行とコメント行から構成されます。このファイルの構文の説明は、912 ページの「ライブラリ ファイル」にあります。完全な make プロセスの詳細な説明も同じ場所にあります。以下に、AgileC コード ジェネレータの概要を説明します。

- **第 1 行**：生成される make ファイルに `make.opt` をインクルードするための記述。コード ジェネレータは、この行を生成される make ファイルの先頭にコピーします。
- **第 2 行**：コンパイル コマンドのためのテンプレート。コード ジェネレータは、このテンプレートを使って、生成される make ファイルに適切なコンパイル コマンドを作成します。
- **第 3 行**：リンク コマンドのためのテンプレート。コード ジェネレータは、このテンプレートを使って、生成される make ファイルにリンク コマンドを作成します。
- **第 4 行**：make プロセス開始のために実行するコマンド。コード ジェネレータがコード生成完了に続いて実行するシェル コマンドです。
- **第 5 行**：ライブラリ構築のためのテンプレート。

<systemname>.m

コード ジェネレータは、コード生成プロセスを完了すると、`compt.opt` の第 4 行目に定義されたコマンドを実行します。通常は以下のとおりです。

```
make -f <systemname>.m sctdir=<a directory>
```

生成された make ファイル（.m）を使って make 機能が起動されることが分かります。

makeoptions (make.opt)

生成された make ファイルの先頭には、ファイル `makeoptions`（`make.opt`）をインクルードする include 文があります。コマンド ラインで make に渡される変数 `sctdir` は、この `makeoptions` が入っているディレクトリを指示するために使用します。

make プログラムは、`makeoptions` ファイルを処理します。このファイルには、コンパイラ、リンカ、カーネル ファイルのコンパイル コマンドのオプションなどについて、数多くの設定が確認します。次に、make プログラムは、生成された make ファイルから、生成ファイルのコンパイル コマンドとオブジェクト ファイルを実行形式ファイルにリンクするコマンドを検出します。

<system name>_env.tpm

テンプレート make ファイルは、コード ジェネレータが直接には管理していないファイルをコンパイルするために使用します。このようなファイルは、テンプレート環境関数を持つファイルが生成された場合に生成されます。ファイル名は `<system name>_env.tpm` であり、テンプレート環境関数を持つファイルのコンパイルを取り扱います。Application Builder によって、ユーザーは使用するテンプレート make ファイルを指定できます。これは、生成したファイル（ファイル名 * を使用）か、またはユーザー定義ファイルのいずれかです。指定したテンプレート make ファイルの内容は、生成された make ファイルに最後にコピーされます。

コンパイラの採用

クロス コンパイラなどを導入する必要がある場合は、`comp.opt` と `makeoptions`（または `make.opt`）ファイルを配置するディレクトリを作成してから、導入します。最も簡単な方法は、既存の `comp.opt` と `makeoptions`（または `make.opt`）を、そのコンパイラ用の新しいディレクトリにコピーして、そこで修正することです。こうすれば、Application Builder が、新しいディレクトリをカーネルとして選択できます。

コンパイラおよびオペレーティング システムとのインテグレーション

このセクションでは、コンパイラとターゲット プラットフォームを設定して、アプリケーションを実行する方法を解説します。

新しいコンパイラとのインテグレーション

新しいコンパイラを導入する場合、注意すべき点が 2 つあります。コンパイラ名とコンパイラスイッチ、および、必要なインクルード ファイルです。

コンパイラ名とコンパイラスイッチ

コンパイラ名と適切なコンパイラ スイッチの指定は、ビルド プロセスの一部です。これについては、すでに前のセクションで取り上げています（1098 ページの「アプリケーションのコンパイルとリンク」を参照）。

インクルード ファイル

もう 1 つの重要なポイントは、カーネル コードに必要なインクルード ファイルと生成コードに必要なインクルード ファイル、および、ユーザー固有コードで使用するインクルード ファイルです。インテグレーションが指定されていない場合、カーネルと生成コードに必要な .h ファイルをインクルードしているデフォルトのセクションが使用されます。この仕組みは、システム インクルード ファイルに関する ISO C 仕様に準拠しています。以下のコードがインクルードされます（カーネル ファイル uml_kern.h と比較してください）。

```
#if defined(USER_CFG_COMPHDEF)

  #include "comphdef.h"

#else

  /* Use default (ISO-C) */
  #include <string.h>
  #include <stdlib.h>
  #include <limits.h>
  #include <stdarg.h>
  #ifdef CFG_ADD_STDIO
    #include <stdio.h>
  #endif

  #if defined(__cplusplus) && defined(_MSC_VER)
    #include <cstddef> /* C++ and Microsoft compiler */
  #else
    #include <stddef.h>
  #endif

#endif
```

USER_CFG_COMPHDEF が定義されていない場合は、上記のとおりのインクルード ファイルがインクルードされます。

注記

`stdio.h` は、プリント文を使用する場合のみ必要です。

コンパイラ付属のコンパイラ／ランタイム ライブラリが、`string.h` に定義された関数を提供しない場合、カーネル（`uml_kern.c`）は、このファイルから使用される関数の実装をインクルードします。以下の関数がインクルードされます。

`memset`、`memcpy`、`strlen`、`strcpy`、`strncpy`、および `strcmp`

Application Builder が生成した `uml_cfg.h` ファイルに、以下の define 文を挿入すると、これらの関数のカーネル実装を使用できます。

```
#define USER_CFG_USE_memset
#define USER_CFG_USE_memcpy
#define USER_CFG_USE_strlen
#define USER_CFG_USE_strcpy
#define USER_CFG_USE_strncpy
#define USER_CFG_USE_strcmp
```

インクルードしたシステム ファイル（および、アプリケーションで必要な他の `.h` ファイル）のリストをカスタマイズするには、マクロ USER_CFG_COMPHDEF を定義する必要があります。

define 文は、ビルド アーティファクトの［**Extra code** / Head］フィールドで追加できます。この操作で、以下の文が `uml_cfg.h` ファイルにインクルードされます。

```
#define USER_CFG_COMPHDEF
```

定義済みカーネルで使用されている他のオプションは、コマンド行から -D オプションまたは同等のオプションを使用してコンパイラに与えた定義を、インクルードするためのものです。

この場合、ファイル `comphdef.h` がインクルードされます。さらに、コンパイラがインクルード ファイルを検索する先のディレクトリのリストの選択する必要があります。この選択によって、コンパイラは正しい `comphdef.h` を見つけることができます。通常、これはコンパイラへの -I オプションによって行います。

ランタイム システムとのインテグレーション

ハードウェアとソフトウェアによって提供されるランタイム システムとのインテグレーションには、以下のように多くの重要なポイントがあります。

- クロック関数
- 動的メモリ割り当てのメモリ管理
- 割り込み許可と割り込み禁止によるデータ保護（非スレッド インテグレーション）
- RTOS とのスレッド インテグレーション

コンパイラとのインテグレーションは、マクロ USER_CFG_RTAPIDEF の定義を `uml_cfg.h` にインクルードすることによって行います。

```
#define USER_CFG_RTAPIDEF
```

この場合、ファイル rtapidef.h が uml_kern.h にインクルードされ、ファイル rtapidef.c が uml_kern.c にインクルードされます。コンパイラが正しい rtapidef.* ファイルを見つけられるように、適切なオプションをコンパイラに与える必要があります。

If USER_CFG_RTAPIDEF が定義されていなければ、標準のインテグレーションが使用されます。この選択を行うアルゴリズムは以下のとおりです。

```
if (threading is used)
  if (Microsoft or Borland compiler is used)
    Use a Win32 threaded integration
  else
    Use a POSIX pthreads integration
  endif
else
  Use a non-threaded integration
endif
```

注記

POSIX pthreads インテグレーションと非スレッド インテグレーションは、ほとんどの UNIX システムおよびコンパイラで動作すると考えられますが、テストが行われたのは、製品がサポートしているシステムとコンパイラのみです。

クロック関数

UML でのタイマー概念をサポートするには、クロック関数が必要です。生成コードとカーネルは、現在の時間を返す xNow というクロック関数が存在することを前提としています。時刻の値は、SDL_Time 型の値によって表現されます。

クロック関数には、UNIX 系システム用と Windows 用の 2 つの標準実装があります。Windows では、システム クロックの読み出しに、標準関数 _ftime が使用されます。UNIX 系システムでは、標準関数 clock_gettime が使用されます。

クロック関数を実装するには、以下の詳細説明に従って、ユーザー独自の rtapidef.h ファイルと rtapidef.c ファイルをインクルードする必要があります。

タイマーを使用しておらず、UML や C 言語から明示的にはクロックにアクセスしていない場合は、クロックを実装する必要はありません。以下のマクロ定義を rtapidef.h ファイルに記述するだけです。

```
#define xInitSystime()
```

クロックの実装が必要な場合、以下の関数プロトタイプを rtapidef.h ファイルに記述します。

```
extern void xInitSystime(void);
extern SDL_Time xNow (void);
```

初期化関数が必要ない場合、xInitSystime 関数の代わりに次のマクロを使用します。

```
#define xInitSystime()
```

rtapidef.c ファイルには、これらの関数の実装が必要です。この実装は、アプリケーションが稼動するソフトウェアとハードウェアに大きく依存します。

メモリ管理

生成されたアプリケーションが、動的メモリを必要とする場合があります。これをサポートするために、「alloc」関数と「free」関数を用意する必要があります。ユーザーには以下の 3 つの選択肢があります。

- 最初の選択肢は、組み込み済みメモリ パッケージを使用することです。このオプションは、Application Builder で指定できます。この実装では、バイト配列を定義し、配列内のメモリを動的メモリとして使用します。メモリ管理に使用するアルゴリズムはカーネルが実装しており、基本的には、「最適な（best fit）」アルゴリズムが使われます。配列のサイズは、ユーザーが設定可能です。最小ブロック サイズの指定も可能です。この場合、2^n * min_block_size のサイズのブロックが割り当てられます。この仕組みによって、メモリの断片化を避けることができます。
- 2 番目の選択肢は、稼働環境から提供される calloc 関数と free 関数に依存することです。この選択肢がデフォルトです。calloc を使用できない場合は、CFG_NO_CALLOC_AVAILABLE を定義することで、代わりに malloc と memset を組み合わせて使用できます。
- 3 番目の選択肢は、メモリ管理を自分自身で実装することです。この場合、マクロ USER_CFG_USE_USER_MEMFUNC を定義する必要があります。以下の関数をを rtapidef.c ファイルに実装することが、前提になります。

```
extern void *xAlloc (unsigned int);
extern void xFree (void **);
```

プロトタイプについては、uml_kern.h に存在するので、rtapidef.h に挿入する必要はありません。

注記

アプリケーションが動的メモリを使用しない場合、これらの関数の実装は不要です。

xAlloc 関数は、パラメータとして与えられたサイズのメモリを確保し、そのメモリへの参照を返します。さらに、メモリは「0」に初期化されます。例を以下に示します。

```
void *xAlloc (unsigned int Size)
  {
    void * Ptr;
    Ptr = (void *)malloc(Size);
    if (Ptr)
      (void)memset(Ptr, 0, Size);
    return Ptr;
  }
```

xFree 関数はメモリへのポインタへのポインタを使用して、そのメモリを解放されたメモリのプールに戻します。この関数はメモリを解放し、そのポインタを「0」でクリアします。例を以下に示します。

```
void xFree (void ** Ptr)
  {
    if (*Ptr) {
      free(*Ptr);
      *Ptr = (void *)0;
    }
```

```
}
```

xAlloc 関数と xFree 関数はスレッド セーフでなければなりません。組み込み済みメモリ パッケージ内の関数の保護は、セマフォか、割り込み許可 / 割り込み禁止によって行います。2 番目の選択肢の場合、つまり OS の calloc と free を使用するときは、これらの関数がスレッド セーフであることを前提としています。

割り込み禁止と割り込み許可

非スレッド アプリケーションにおいて、割り込みルーチンからシグナルをシステムに送信できるようにしたい場合には、シグナル用の重要なデータ構造を同時アクセスから保護する必要があります。これを実現するには、カーネルが特定の操作を実行している間は、割り込みを禁止する必要があります。

割り込みの禁止と割り込みの許可を実装するには、下記の構造を持つ２つのマクロを rtapidef.h で定義する必要があります。

```
#define XBEGIN_CRITICAL_PATH ¥
        UserCodeToDisableInterrupts;

#define XEND_CRITICAL_PATH ¥
        UserCodeToEnableInterrupts;
```

ここで、UserCodeToDisableInterrupts と UserCodeToEnableInterrupts を、使用しているハードウェアとソフトウェアプラットフォーム上でのアクションを実行するコードに、置き換える必要があります。

スレッド インテグレーション

新しいインテグレーションの実装するには、このドキュメントの記載と合わせて、既存のインテグレーションのコードの解説ドキュメントも参照してください。リアルタイム OS とのインテグレーションを実装するには、以下の点を注意する必要があります。

- クロック関数を実装する必要がある。
- 複数のミューテックスやバイナリ セマフォを使用して、共有データを保護する必要がある。
- 適切なプロパティを持つスレッドを作成して同期するため、起動コードが必要になる。
- スレッドは、アイドルなときに自分の実行を中断できなければならない。そして、シグナルがスレッド内のパートへ送信されたときに、再起動して実行できなければならない。

インテグレーションの詳細を説明するため、例として POSIX インテグレーションを使用します。以下に示すコードの他に、rtapidef.h には、システム インクルード ファイルが必要です。

例 **391: POSIX** の **rtapidef.h** のインクルード

```
#include <pthread.h>
#include <sched.h>
#include <semaphore.h>
#include <time.h>
#include <sys/time.h>
```

例のように、RTOS の要請で `main()` 関数の名前を変更する必要がある場合は、XMAIN_NAME を以下のように定義して、`uml_kern.c` に含まれる `main()` 関数の名前を変更できます。例を以下に示します。

```
#define XMAIN_NAME agilec_main
```

ユーザーは、`agilec_main` 関数を呼び出す適切な main 関数を実装する必要があります。

クロック関数

クロック関数を、前のセクションの説明にしたがって実装する必要があります。

共有データの保護

利用可能なシグナルのリスト、利用可能なタイマーのリスト、解放されたパートのリスト（create アクションの場合）の保護のほかに、メモリ パッケージを使用している場合は、パッケージによって使用されるメモリの保護が必要です。

このためには、4 つのグローバル ミューテックスまたはバイナリ セマフォが必要です。これらの変数は、`rtapidef.h` で `extern` として定義され、`rtapidef.c` で宣言する必要があります。変数の名前は、以下に示す例と同じにします。

例 **392: rtapidef.h** ファイルの例

```
extern pthread_mutex_t xFreeSignalMutex;
extern pthread_mutex_t xFreeTimerMutex;
extern pthread_mutex_t xCreateMutex;
#ifdef USER_CFG_USE_MEMORY_PACK
  extern pthread_mutex_t xMemoryMutex;
#endif
```

例 **393: rtapidef.c** ファイル：

```
pthread_mutex_t xFreeSignalMutex;
pthread_mutex_t xFreeTimerMutex;
pthread_mutex_t xCreateMutex;
#ifdef USER_CFG_USE_MEMORY_PACK
  pthread_mutex_t xMemoryMutex;
#endif
```

これらの 4 つの変数は、アプリケーションの起動時に初期化されて、ロック解除状態にする必要があります。この初期化には、xThreadInit 関数が使用されます。

例 **394: xThreadInit**

```
void xThreadInit (void)
{
  (void)pthread_mutex_init(&xFreeSignalMutex, 0);
  (void)pthread_mutex_init(&xFreeTimerMutex, 0);
  (void)pthread_mutex_init(&xCreateMutex, 0);
  #ifdef USER_CFG_USE_MEMORY_PACK
    (void)pthread_mutex_init(&xMemoryMutex, 0);
  #endif
  ....
}
```

また、ミューテックスまたはバイナリ セマフォのために、ロック操作とロック解除操作を実装する必要があります。以下の 2 つの関数を実装します。

例 **395:** ロックおよびロック解除のための関数

rtapidef.h ファイル：

```
extern void xThreadLock (pthread_mutex_t *);
extern void xThreadUnlock (pthread_mutex_t *);
```

rtapidef.c ファイル：

```
void xThreadLock (pthread_mutex_t *M)
{
  (void)pthread_mutex_lock(M);
}

void xThreadUnlock (pthread_mutex_t *M)
{
  (void)pthread_mutex_unlock(M);
}
```

起動フェーズ - スレッドの作成

基本的な初期化の後で、AgileC コード ジェネレータカーネルは main() 関数内で指定スレッドを起動します。

スレッドごとに、xThreadInitOneThread 関数と xThreadStartThread 関数が呼び出されます。 xThreadInitOneThread はスレッド固有の初期化を行い、xThreadStartThread はスレッドを起動します。スレッドごとにカーネルで宣言された xMainLoop 関数が実行されます。これは、ラッパー関数 xThreadEntryFunc を使用して実行されます。このラッパー関数は、インテグレーションで定義され、スレッドで実際に開始される関数です。

すべてのスレッドが起動したら、main 関数で xThreadGo 関数が呼び出されます。これらの関数の詳細を次に説明します。

重要なのは、すべてのスレッドが作成され終わるまで、起動したスレッドは、UML 遷移を実行しない、ということです。したがって、xThreadEntryFunc は、最初のアクションとしてセマフォ上で待ち状態に入ります。すべてのスレッドが作成されると、xThreadGo 関数が、このセマフォを解放します。

グローバル データ構造 xSysD は、xSystemData 型のコンポーネントを保持する配列です。1 つのコンポーネントが 1 つのスレッドに対応します。xSysD は、スレッド内で現在起こっている事象についてのグローバル情報を保持しています。xSysD 内の情報の詳細については、1125 ページの「重要なデータ構造の概要」を参照してください。RTOS インテグレーションのコンテキストでは、2 つの点が重要です。まず、各スレッド（xMainLoop 関数）は、自分を表す xSysD のコンポーネントのアドレスを知っている必要があります。次に、各 xSysD のコンポーネントには、xThreadVars 型のフィールドがあり、このフィールドは、RTOS インテグレーションで定義する必要があります。

例は以下のとおりです。

rtapidef.h 内の xThreadVars 型

```
typedef struct {
  pthread_mutex_t SignalQueueMutex;
  pthread_cond_t  SignalQueueCond;
  pthread_t       ThreadId;
} xThreadVars;
```

最初の 2 つのフィールドについては、次のセクションで説明します。ThreadId は、起動時にスレッドの ID を格納するために使用されます。

このセクションで説明した動作のコード例は、以下のようになります。

例 **396:** ——————————————————————————————————

rtapidef.h ファイル：

```
extern sem_t xInitSem;
#if !defined(USER_CFG_USE_xInEnv) && !defined(XENV)
  extern sem_t xMainThreadSem;
#endif

extern void xThreadInitOneThread (
   struct _xSystemData *);
extern void xThreadStartThread (
   struct _xSystemData *,
   unsigned int, unsigned int,
   unsigned int, unsigned int);
```

rtapidef.c ファイル：

```
sem_t xInitSem;
#if !defined(USER_CFG_USE_xInEnv) && !defined(XENV)
  sem_t xMainThreadSem;
#endif

void xThreadInit (void)
{
  ....
```

```
  (void)sem_init(&xInitSem, 0, 0);
}

void xThreadInitOneThread(struct _xSystemData *xSysDP)
{
  (void)pthread_mutex_init(
    &xSysDP->ThreadVars.SignalQueueMutex, 0);
  (void)pthread_cond_init(
    &xSysDP->ThreadVars.SignalQueueCond, 0);
}


static void *xThreadEntryFunc (void *xSysDP)
{
  (void)sem_wait(&xInitSem);
  (void)sem_post(&xInitSem);
  xMainLoop((xSystemData *)xSysDP);
}


void xThreadStartThread(struct _xSystemData *xSysDP,
                        unsigned int StackSize,
                        unsigned int Prio,
                        unsigned int User1,
                        unsigned int User2)
{
  pthread_attr_t Attributes;
  ....
  (void)pthread_create(&xSysDP->ThreadVars.ThreadId,
                       &Attributes, xThreadEntryFunc,
                       (void *)xSysDP);
  ....
}

void xThreadGo(void)
{
  (void)sem_post(&xInitSem);

  #if defined(USER_CFG_USE_xInEnv)
    xInEnv();  /* AgileC */
  #elif defined(XENV)
    xInEnv(xNow());  /* Cadvanced */
  #else
    (void)sem_init(&xMainThreadSem, 0, 0);
    (void)sem_wait(&xMainThreadSem);
  #endif
}
```

`xInitSem` セマフォは、スレッドの同期のために使用します。`xThreadInit` の最初で「0」に初期化され、ブロック状態になります。その後、各スレッドで 1 回ずつ `xThreadStartThread` が起動します。ここで、`pthread_create` 関数が、3 番目のパラメータとして指定された関数（`xThreadEntryFunc`）を呼び出します。このとき、4 番目のパラメータとして指定された `(void *)xSysD` ポインタを使用します。また、`pthread_create` 関数は、1 番目のパラメータとして渡された変数 `ThreadVars` にスレッドの ID を保存します。2 番目のパラメータは、スレッドのプロパティですが、これについては後で説明します。

すべてのスレッドが作成完了前に、いずれかのスレッドが実行されてしまった場合、そのスレッドは、xThreadEntryFunc 関数内の sem_wait で中断し、メイン スレッドが xThreadGo 関数を呼び出してセマフォ xInitSem をポストするまで、待ち状態になります。このセマフォで待ち状態に入っているスレッドの 1 つが実行可能になり、直ちにまたセマフォがポストされます。すべてのスレッドが実行されるまで、これが続きます。

OS とアプリケーションのプロパティにより異なりますが、すべてのスレッドが実行された後、メイン スレッドは別の作業を実行できるようになります。推奨される作業は、xInEnv 関数を呼び出して、メインスレッドで実行することです。そうでなければ、上記のコードのようにセマフォ xMainThreadSem を使用して（xInEnv が使用されていない場合）、メイン スレッドを中断します。このセマフォのポストは任意の箇所で行うことができ、ポストされれば、メイン スレッドは実行を再開します。

メイン スレッドが xThreadStart 関数から戻った場合、プログラムは main() 関数での実行を継続し、exit を呼び出します。メイン スレッドが exit を実行したときのスレッド プログラムの振る舞いは、OS に依存します。POSIX pthreads では、すべてのスレッドが終了します。xThreadStart 関数の最後でメイン スレッドを保留することが重要なのは、このためです。

次に、スレッドのプロパティについて説明します。一般に RTOS では、スタック サイズや優先度などのプロパティは、個々のスレッドごとに設定できます。スレッド アーティファクトの使用での定義以外にも、以下の 4 つの整数値を指定できます。

- 1 番目の値 - スタック サイズ
- 2 番目の値 - 優先度
- 3 番目および 4 番目の値 -RTOS やユーザーが定義したプロパティのために使用 k 可能

定義済みのインテグレーションでは、1 番目の値と 2 番目の値のみが使用されています。これらの値はパラメータとして xTreadStartThread 関数に渡されます。

プロパティ設定方法の詳細は、RTOS に依存します。xThreadStartThread 関数など、利用可能なインテグレーションの記述を参照してください。

rtapidef.h ファイルで、4 つの xThreadData フィールドの適切なデフォルト値を設定します。スレッド 定義で値を指定しない場合は、これらのデフォルト値が使用されます。

例 **397**

```
#define DEFAULT_STACKSIZE     1024
#define DEFAULT_PRIO             0
#define DEFAULT_USER1            0
#define DEFAULT_USER2            0
```

スレッドの一時停止と再開

何も実行する作業がない場合、スレッドは自身を一時停止して、他のスレッドにプロセッサを解放します。タイマーが切れた場合、処理が必要になった場合、他のスレッド（xInEnv など）がこの停止中のスレッドで処理されるべきシグナルを送信した場合などに、スレッドは再開されます。

この機能を実装するには、条件変数と 1 つの mutex（バイナリ セマフォ）を使用します。タイムアウト付き、またはタイムアウトなしの条件付き wait や、一時停止中のスレッドを再開するシグナル送信の方法を実現する必要があります。この 2 つのエンティティ、つまり条件変数と 1 つの mutex は、スレッドごとに必要です。したがって、これらは、前に説明した xThreadVars 構造体に含まれています。

```
typedef struct {
  pthread_mutex_t SignalQueueMutex;
  pthread_cond_t  SignalQueueCond;
  pthread_t       ThreadId;
} xThreadVars;
```

SignalQueueMutex は、スレッドの外側から送信されたシグナルが挿入されるシグナル キューを保護します（xSysD 配列内の ExternSignalQueue）。
SignalQueueCond は、条件付き wait を可能にします。

SignalQueueMutex は、xThreadInitOneThread 関数で初期化する必要があります。SignalQueueCond を初期化する必要がある場合は、同じ場所で行われます。

例 **398**

```
void xThreadInitOneThread(struct _xSystemData *xSysDP)
{
 (void)pthread_mutex_init(&xSysDP->ThreadVars.SignalQueueMutex, 0);
 (void)pthread_cond_init(&xSysDP->ThreadVars.SignalQueueCond, 0);
}
```

SignalQueueMutex は、前に説明したように、xThreadLock 関数によってロックされます。ロック解除の方法には、以下の 3 つがあります。

- xThreadUnlock（前述）
- xThreadWaitUnlock
- xThreadSignalUnlock

xThreadWaitUnlock は、スレッドが自身を停止すべきだと判断した際に、スレッド自身から呼び出されます。xThreadSignalUnlock は、停止中のスレッドを再開したい別のスレッドから呼び出されます。どちらの関数にも、操作を実行するスレッドの xSysD ポインタが渡されます。

例 **399**

rtapidef.h ファイル：

```
extern void xThreadWaitUnlock (struct _xSystemData *);
extern void xThreadSignalUnlock (struct _xSystemData *);
```

In rtapidef.c ファイル：

```
void xThreadWaitUnlock (struct _xSystemData *xSysDP)
{
  #if defined(CFG_USED_TIMER) || defined(THREADED)
  #ifdef THREADED
      /* Cadvanced */
    if (xSysDP->xTimerQueue->Suc==xSysDP->xTimerQueue) {
  #else
      /* AgileC */
    if (! xSysDP->TimerQueue) {
  #endif
      (void)pthread_cond_wait(
        &xSysDP->ThreadVars.SignalQueueCond,
        &xSysDP->ThreadVars.SignalQueueMutex);
    } else {
      struct timespec timeout;
      #ifdef THREADED
          /* Cadvanced */
        timeout.tv_sec =
          ((xTimerNode)xSysDP->xTimerQueue->Suc)->
            TimerTime.s;
        timeout.tv_nsec =
          ((xTimerNode)xSysDP->xTimerQueue->Suc)->
            TimerTime.ns;
      #else
          /* AgileC */
        timeout.tv_sec = xSysDP->TimerQueue->Time.s;
        timeout.tv_nsec = xSysDP->TimerQueue->Time.ns;
      #endif
      (void)pthread_cond_timedwait(
        &xSysDP->ThreadVars.SignalQueueCond,
        &xSysDP->ThreadVars.SignalQueueMutex,
        &timeout);

    }
  #else
    (void)pthread_cond_wait(
      &xSysDP->ThreadVars.SignalQueueCond,
      &xSysDP->ThreadVars.SignalQueueMutex);
  #endif
  (void)pthread_mutex_unlock(
    &xSysDP->ThreadVars.SignalQueueMutex);
}

void xThreadSignalUnlock (struct _xSystemData *xSysDP)
{
  (void)pthread_cond_signal(
    &xSysDP->ThreadVars.SignalQueueCond);
  (void)pthread_mutex_unlock(
    &xSysDP->ThreadVars.SignalQueueMutex);
}
```

---

xThreadWaitUnlock 関数または xThreadSignalUnlock 関数が呼び出されると、SignalQueueMutex はロックされるので、両関数の最後に、そのロックを解除する必要があります。

xThreadWaitUnlock 関数では、スレッドは自身を一時停止しようとします。タイマーを使用していて、タイマー キューにアクティブなタイマーがある場合、そのタイマーは、タイマーが切れるか別のスレッドによって再開されるまで待機します。POSIX pthread では、pthread_cond_wait 関数によって、同様な動作が実行されます。タイマーを使用していないかアクティブなタイマーがない場合は、他の作業がそのスレッドを再開するまで、スレッドは停止されたままになります。POSIX pthreads では、pthread_cond_wait 関数によって、同様な動作が実行されます。

xThreadSignalUnlock 関数では、パラメータによって指定されたスレッドが再開されます。ここでは、pthread の関数 pthread_cond_signal を使用できます。

ここで説明するインテグレーションは、C コード ジェネレータでスレッド アプリケーションを生成するときにも使用されます。C コード ジェネレータのスレッド インテグレーションの実装には、いくつかの要件があります。まず、スレッドを停止できる関数が必要です。

rtapidef.h ファイル：

```
#if defined(THREADED) || defined(CFG_USED_DYNAMIC_THREADS)
extern void xThreadStopThread(struct _xSystemData *);
#endif
```

rtapidef.c ファイル：

```
#if defined(THREADED) || defined(CFG_USED_DYNAMIC_THREADS)
void xThreadStopThread(struct _xSystemData *xSysDP)
{
  pthread_mutex_destroy(&xSysDP->ThreadVars.SignalQueueMutex);
  pthread_cond_destroy(&xSysDP->ThreadVars.SignalQueueCond);
  pthread_exit(0);
}
#endif
```

THREADED は、C コード ジェネレータを使用するときに定義されますが、AgileC コード ジェネレータを使用するときは定義されません。xThreadStopThread 関数はスレッド固有のセマフォを除去し、スレッドを停止します。この関数を呼び出して、自分自身を停止します。

また、もう 1 つの違いは、2 つのコード ジェネレータのタイマーにアクセスする方法が異なることです。これは、xThreadWaitUnlock 関数の詳細に影響します。

C コード ジェネレータの場合、RTOS インテグレーションはマクロ層からアクセスされます。この階層のマクロは、C コード ジェネレータのカーネル ファイル内と生成されたコード内で使用されます。

例 **400: scttypes.h** の **#define** 文

以下の #define が該当します（scttypes.h から）。

```
#define THREADED_GLOBAL_VARS
#define THREADED_GLOBAL_INIT ¥
     xThreadInit();
#define THREADED_THREAD_VARS ¥
     xThreadVars ThreadVars;
#define THREADED_THREAD_INIT(SYSD) ¥
     xThreadInitOneThread(SYSD);
#define THREADED_THREAD_BEGINNING(SYSD)
#define THREADED_AFTER_THREAD_START ¥
```

```
     xThreadGo();
#define THREADED_START_THREAD(F, SYSD, STACKSIZE, PRIO, USER1,
USER2) ¥
xThreadStartThread(SYSD, STACKSIZE, PRIO, USER1, USER2);
#define THREADED_STOP_THREAD(SYSD) ¥
     xThreadStopThread(SYSD);
#define THREADED_LOCK_INPUTPORT(SYSD) ¥
     xThreadLock(&SYSD->ThreadVars.SignalQueueMutex);
#define THREADED_UNLOCK_INPUTPORT(SYSD) ¥
     xThreadUnlock(&SYSD->ThreadVars.SignalQueueMutex);
#define THREADED_WAIT_AND_UNLOCK_INPUTPORT(SYSD) ¥
     xThreadWaitUnlock(SYSD);
#define THREADED_SIGNAL_AND_UNLOCK_INPUTPORT(SYSD) ¥
     xThreadSignalUnlock(SYSD);
#define THREADED_LISTREAD_START  xThreadLock(&xFreeSignalMutex);
#define THREADED_LISTWRITE_START xThreadLock(&xFreeSignalMutex);
#define THREADED_LISTACCESS_END  xThreadUnlock(&xFreeSignalMutex);
#define THREADED_EXPORT_START    xThreadLock(&xCreateMutex);
#define THREADED_EXPORT_END      xThreadUnlock(&xCreateMutex);
```

MISRA コーディング ルール

AgileC コード ジェネレータランタイム カーネルのコードと AgileC コード ジェネレータによって生成されたコードは、その大半が、MISRA コーディング ルールに準拠しています。

AgileC コード ジェネレータコードは、UML 制限事項（C で goto が許されない場合、UML で goto を使用してはならない、など）を考慮すれば、141 の MISRA ルールのうちの 121 に準拠しています。UML 制限事項以外の他の制限事項も考慮に入れると、AgileC コード ジェネレータの生成コードは、さらに 6 つのルールに準拠します。残りの 6 つの必須ルールと 8 つの勧告ルールには準拠しません。

MISRA 準拠コードの生成を可能にするには、AgileC コード ジェネレータ オプションの［Generate MISRA compliant code］をチェックします。. さらに、［Code generation］プロパティの［Name mangling］オプションを［Prefix］に設定する必要があります（名前がその最初の 31 文字の範囲内で確実に一意となるようにするためールール 5.1）。

MISRA 準拠コードを有効にした場合、生成コードは MISRA に準拠するように変更されます。もっとも重要な変更は、stop アクションの変換方法です。1 つのダイアグラム内に複数の stop アクションが存在する場合、MISRA に準拠したコードは goto 文が使えないために、若干サイズが大きくなります。

MISRA 準拠を有効にした場合、MISRA チェックも有効となります。このチェックを有効にすると、生成 C コードが MISRA ルールに準拠しなくなるような問題が UML モデルにある場合に、警告が発せられます。

UML における明らかな制限事項

MISRA 準拠のコードを生成するためには、ユーザーが UML における制限事項を順守することが期待されます。たとえば、C の特定の概念について制限事項がある場合、明らかに、UML. の同様の概念にも同じ制限事項を適用しなければなりません。例を以下に示します。

- C では goto の使用が許されないので、ユーザーは UML に goto 文または結合フロー（グラフィカルな goto）を持ち込んではなりません。ルール 14.4.。
- 関数はただ 1 つの出口ポイントを持つので、ユーザーは UML にそのように操作を記述しなければなりません。ルール 14.7.。
- ルール 13.4、13.5、および 13.6 の for ループに対する要求は、UML の for ループにそのままマッピングされます。
- 繰り返しに含める break は 1 つだけです。このルールは、UML でも順守する必要があります。ルール 14.6.。
- ルール 16.2 では再帰呼び出しは許されていません。これは UML 操作（operation）にそのまま適用されます。
- ルール 18.4 によれば、共用体は使用してはなりません。つまり、UML での同等機能 ”`choice`” を使用してはなりません。

ユーザー記述の C コードを AgileC コード ジェネレータの生成コードに含めることができるので、この追加されたコードが MISRA ルールに違反していないことも前提となります。同じことは、このツールにインポートされる C コードについてもいえます。

UML における他の制限事項

下記の表は、一般的な AgileC コード ジェネレータアプリケーションで違反となるルールの一覧です。一部の拡張データ型を使用しなければ、生成コードはルールに準拠することになります。

ルール番号	ルール
11.1	「関数型」から別の「関数型」へのキャストを行なってはなりません。
12.10	カンマ演算子を使用してはなりません。
16.1	引数の個数を変えられる関数を使用してはなりません。
16.2	再帰呼び出しを使用してはなりません。
18.4	共用体を使用してはなりません。
20.4	動的メモリ割り当てを使用してはなりません。

一般的な **AgileC** コード ジェネレータ アプリケーションで違反となるルール

以下のルールを順守することにより、生成されたアプリケーションを準拠させることができます。

1. 下記のデータ型とテンプレートを使用してはなりません。
   - `Bit_string`、`Octet_string`、`Object_identifier`
   - `String`、`Bag`、`Own`, `Ref`
   - C の `Array` へのマッピングにインデックス型が妥当な型の場合、`array` だけを使用します（文字型と `Boolean` を含む列挙型、またはそれらの型に `integer` 型を加えた、範囲が限定された `syntype` など）。
   - `Powerset` は、上記の `Array` のインデックス型の記述にあったコンポーネント型とともに使用する必要があります。
2. アクティブ クラスとパッシブ クラスのすべての単純でない属性は「パート」でなければなりません。「パート」でない場合、オブジェクトに対する参照が行なわれます。
3. アクティブ クラスの属性の場合を除いて、属性に多重度を使用してはなりません。代わりに、`Array` か、または `CArray` を使用してください。
4. パッシブ クラスを返す操作、`Array`、`CArray` または `Powerset` を使用してはなりません。これには make 演算子が含まれます。
5. パッシブ クラスにデストラクタを使用してはなりません。

ルール 20.4（動的メモリ割り当てを使用しない）により、AgileC の構成に次の 2 つの要求が追加されます。

6. アクティブ クラスの最大インスタンス数を必ず指定しなければなりません。
7. すべてのシグナル型が適合するように、シグナル データのサイズを設定しなければなりません。これは `USER_CFG_MSG_BORDER_LEN` を構成します。

ルール 16.2（再帰呼び出しの禁止）`sctpred.h` には若干の再帰関数が存在しますが、上記の制限事項にしたがって、再帰呼び出しは行なわれません。

ルール違反

以下のセクションでは、コードジェネレータが違反しているルールとその理由について説明します。

ルール **14.7.** 関数は出口ポイントをその関数の終わりにただ **1** つ持つものとします。

このコーティング基準は AgileC の開発時には使用されてきませんでした。このようなコード記述方法には良い点と悪い点があります。出口が 1 つしかないという約束があると、コードを把握しやすくなる可能性があります。しかし一方、実際には関数の制御構造がより複雑になります。

また、このルールが実際に有効でない特別な例が、生成コードには存在します。生成コードの一部の関数は、状態機械を表します。1 つの状態機械が複数の異なる遷移を表し、各遷移にはそれ自身の論理的な終点が存在します。この論理的な終点は、関数では return として実装され、結果的に複数の出口ポイントとなります。

ルール **15.2** 無条件の **break**（実行の中断）は空でない各 **switch** 節を終了するものとします。

このルールの目的は、おそらく、switch 節間でのフォール スルーを回避することにあります。フォール スルーは AgileC コード ジェネレータでは使用されません。関数内で return の使用が許されているので、代わりに次のルールが使用されます。

無条件の break、または無条件の return は空でない各 switch 節を終了させます。

ルール **17.1** ポインタの算術演算は、配列または配列要素をアドレス指定するポインタに対してのみ適用します。
ルール **17.4** 配列のインデクシングがポインタの算術演算で唯一許される形式です。

アクティブ クラスの属性を効率よく処理するために、または同時に動的メモリを使用しないために、AgileC コード ジェネレータ カーネル内の複数の箇所でこれらのルールに違反しています。

ルール **19.4 C** マクロは、大かっこで囲まれたイニシャライザ、定数、かっこで囲まれた式、型修飾子、記憶域クラス指定、**do-while-zero** 構造体のみに展開されます。

このルールは違反されています。このルールに違反したマクロを使用してコード最適化を実装しているケースが多くあります。

ルール **19.6 #undef** は使用してはなりません。

undef 文は生成されるコード内において 2 つの状況で使用されます。`undef` 文を取り除くためのシンプルなソリューションはまだ見つかっていません。

サポートされていない勧告ルール

AgileC コードが準拠していない勧告ルールには、5.6、5.7、6.3、11.3、11.4、16.7、19.1、19.7 があります。

最適化と設定

最適化と設定は、主として 2 つの .h ファイル、auto_cfg.h と uml_cfg.h で行われます。ファイル auto_cfg.h は AgileC コード ジェネレータによって生成され、そのファイルには、コードの生成対象のシステム向けに自動で算出された最適化と設定が含んでいます。uml_cfg.h ファイルは、ユーザーが提供する設定情報から Application Builder が生成します。

auto_cfg.h

auto_cfg.h には、以下の内容が含まれています。ファイルの最初のセクションは、使用する配列のサイズと、特定のエンティティを表現するために必要なサイズ定数（8、16、または 32 ビット）の設定に主に使用されます。

```
#define CFG_NUMBER_PARTS 2
#define CFG_MAX_INSTANCES 1
#define CFG_NUMBER_TIMERS 0
#define CFG_MAX_TIMER_INSTS 0
#define CFG_NUMBER_SIGNALS 4
#define CFG_NUMBER_THREADS 0
#define CFG_MAX_ACTIONS 2
#define CFG_MAX_STATES 1
#define CFG_MAX_STATE_INDEX_ENTRY 2
```

2 番目のセクションは、コード生成の対象となるシステムで使用する概念に関する情報です。使用する概念について、#define 文が生成されます。一方、使用しない概念については、対応するマクロが定義されていないことを示すコメントが生成されます。このセクションの情報を使用して、コードのスケーリング、すなわち、必要のないデータ構造内のコードとフィールドの削除が行われます。

```
#define CFG_USED_UNLIMITED_INSTANCES
#define CFG_USED_TIMER
#define CFG_USED_CREATE
#define CFG_USED_STOP
#define CFG_USED_SAVE
#define CFG_USED_STATE_STAR
#define CFG_USED_INPUT_SAVE_STAR
#define CFG_USED_GUARD
#define CFG_USED_SIGNAL_WITH_PARAMS
#define CFG_USED_TIMER_WITH_PARAMS
#define CFG_USED_SIGNAL_WITH_DYN_PARAMS
#define CFG_USED_SENDER
#define CFG_USED_OFFSPRING
#define CFG_USED_PARENT
#define CFG_USED_SELF
#define CFG_USED_PROCEDURE
#define CFG_USED_RPC
#define CFG_USED_INITFUNC
```

3 番目のセクションには、生成したアプリケーションでは不要となる sctpred.c ファイル内のコードの削除に使用される、データ型に関係する項目についての情報が入っています。このセクションには、以下のような define 文があります。

```
#define XNOUSE_"at-lot-of-things-about-data-types"
```

まれに、auto_cfg.h ファイルに、作成されるアプリケーションと合わない情報が含まれることがあります。たとえば、特定の関数が使用されないという情報が 3 番目のセクションに入り、その結果 #ifdef によってその関数が除去されるが、実はこの関数が、環境関数内などユーザーが提供する C コード内で使用されるような場合です。この場合、uml_cfg.h の最後にマクロ定義を含めることによって、auto_cfg.h 内の情報を無効にできます。

例は以下のとおりです。auto_cfg.h に下記の行が含まれているものと仮定します。

```
#define XNOUSE_LENGTH_CHARSTRING
```

この行は、文字列の length 関数が使用されないことを示しています。この行を無効にするには、次のコードを uml_cfg.h に組み込みます。

```
#ifdef XNOUSE_LENGTH_CHARSTRING
  #undef XNOUSE_LENGTH_CHARSTRING
#endif
```

uml_cfg.h

このファイルは Application Builder によって生成され、ユーザーが選択したオプションに関する情報を保有しています。また、Application Builder では、テキスト フィールドに書かれたテキストを uml_cfg.h に挿入できます。この機能を使うと、ユーザー インターフェイスでは選択できないマクロ定義をコードに挿入できます。

uml_cfg.h は、auto_cfg.h と同じく、指定したオプションを使って AgileC コード ジェネレータアプリケーションを構築するために使用される定義が入るファイルです。uml_cfg.h ファイルは、ファイルの最初で auto_cfg.h をファイルをインクルードします。

以下に説明する機能は Application Builder のユーザー インターフェイスから選択が可能で、uml_cfg.h ファイルの内容に影響します。

Process properties（プロセス プロパティ）

- `#define USER_CFG_USE_NUMBER_FREE_INST <Integer>`

  インスタンス数が最大値を指定されていないパートについて、メモリが起動時に動的に割り当てられます。そのサイズは、初期インスタンス数分プラスここで指定する値になります。値は 0 より大きい必要があります。デフォルト値は 5 です。

Signal properties（シグナル プロパティ）

- `#define USER_CFG_MAX_SIGNAL_INSTS <Integer>`

  静的なシグナル キューの長さを定義します。動的なシグナルが使用されていない場合（USER_CFG_NOUSE_DYNAMIC_SIGNALS と比較）、シグナルを送信しようとしたときにシグナル キューが満杯になっていると、エラーが発生します。

- `#define USER_CFG_NOUSE_DYNAMIC_SIGNALS`

  シグナルに対する動的メモリの使用を停止します。小規模なシステムなど、動的メモリを使用しないシステムでは、この定義を設定します。この場合、USER_CFG_MAX_SIGNAL_INSTS を適切な値に設定することが重要です。

- `#define USER_CFG_SIGNALS_NEVER_DISCARDED`

  シグナル破棄の際にシグナル パラメータの領域をフリーするコードを入れたくない場合に設定します。この定義は、動的パラメータを持つシグナルを使用している場合のみ適用できます。XMK_USED_SIGNAL_WITH_DYN_PARAMS が auto_cfg.h で定義されている場合。USER_CFG_SIGNALS_NEVER_DISCARDED が定義されていて、かつ動的パラメータを持つシグナルが破棄されると、メモリ リークが発生します。
- `#define USER_CFG_USE_SIGNAL_PRIORITIES`

  シグナルについて、優先順位を使用します。シグナル キューは、まず優先順位でソートされ、続いて、到着順（同じ優先順位のとき）にソートされます。
- `#define USER_CFG_TIMER_PRIO <Integer>`

  すべてのタイマー シグナルに割り当てられる優先順位。デフォルトは 50 です。
- `#define USER_CFG_CREATE_PRIO <Integer>`

  すべての起動シグナルおよび生成シグナルに割り当てられる優先順位。デフォルトは 50 です。
- `#define USER_CFG_DEFAULT_PRIO <Integer>`

  明示的な優先順位を持たないすべてのシグナルに割り当てられる優先順位。デフォルトは 50 です。
- `#define USER_CFG_MSG_BORDER_LEN <Integer>`

  シグナル内のパラメータ用のデータ フィールドの長さを定義します。シグナル パラメータがこのメモリに入り切らない場合、動的メモリ割り当てが使用されます。デフォルト値は、パラメータを持つシグナルが使用されている（XMK_USED_SIGNAL_WITH_PARAMS が定義されている）場合は 4、どのシグナルもパラメータを持たない場合は 0 です。

Timer properties（タイマー プロパティ）

- `#define USER_CFG_MAX_TIMER_INSTS <Integer>`

  静的タイマー キューの長さ。デフォルトの長さは `auto_cfg.h` から算出した値 CFG_MAX_TIMER_INSTS です。パラメータを持たないタイマーの場合、デフォルトの長さはアクティブ タイマーの最大量です。パラメータを持つタイマーについては、この値を小さく設定します。通常、CFG_MAX_TIMER_INSTS よりも小さい値を使用します。動的なタイマーが使用されていない場合（USER_CFG_NOUSE_DYNAMIC_TIMERS と比較）、タイマーを設定しようとしたときにタイマー キューが満杯になっていると、エラーが発生します。
- `#define USER_CFG_NOUSE_DYNAMIC_TIMERS`

  タイマーに対する動的メモリの使用を停止します。この機能は、動的メモリを使用しない小規模なシステムでは通常オフにします。この場合、USER_CFG_MAX_TIMER_INSTS を適切な値に設定することが重要です。

- `#define USER_CFG_TIMER_SCALE <Integer>`

  この値によって、設定されたすべてのタイムアウト値のスケールを変更できます。初期段階のテストにおいて、実際には短時間で発生するタイムアウト事象を、検知可能なスケールに落として（たとえば、1 ミリ秒のタイムアウトに 1 秒かかるようにする、など）動作を検証する場合などに有効です。したがって、完成版のアプリケーションでは使用すべきではありません。処理速度のオーバヘッドとなるからです。

Dynamic memory allocation（動的なメモリ割り当て）

- `#define USER_CFG_USE_MEMORY_PACK`

  OS の関数 `malloc` と `free` の代わりに、ライブラリとして提供されているメモリ管理パッケージを使用するようにします。このパッケージを使用する場合は、USER_CFG_MEMORY_SIZE も設定する必要があります。デフォルトは定義されていません。
- `#define USER_CFG_USE_MEMORY_PACK <Integer>`

  メモリ パッケージが使用するバイト数を定義します。値は 16 の倍数で指定します。デフォルトは 8192 バイトです。
- `#define USER_CFG_MEMORY_MIN_BLOCKSIZE <Integer>`

  メモリ ブロックの最小サイズを定義します。この値を定義した場合、2^N * USER_CFG_MEMORY_MIN_BLOCKSIZE, N>=0 のサイズのブロックが使用されます。値は 16 の倍数で指定します。デフォルトは定義されていません。

Error detection（エラー検出）

- `#define USER_CFG_ERR_CHECK_BASIC`

  以下の基本的な状態機械のプロパティのチェックをオンにします。
  - シグナルの送信または作成のためのメモリがこれ以上ない。
  - シグナルのパラメータに割り当てるメモリがこれ以上ない。
  - タイマー インスタンス用のメモリがこれ以上ない。
  - `xOutEnv()` が存在せず、シグナルが環境へ送信される。
  - シグナルが破棄される。
  - 最大限度に到達したので、インスタンスをこれ以上作成できない。
  - パブリック属性のアクセス エラー。

  デフォルトは定義されていません。
- `#define USER_CFG_ERR_CHECK_INDEX`

  配列インデックスが範囲内にあることをテストします。デフォルトは定義されていません。
- `#define USER_CFG_ERR_CHECK_RANGE`

  シンタイプ値が範囲内にあることをテストします。デフォルトは定義されていません。

- `#define USER_CFG_ERR_CHECK_PREDEF_O`

  定義済み演算子のエラー状態をテストします。デフォルトは定義されていません。
- `#define USER_CFG_ERR_CHECK_DECISION`

  現在の分岐値のパスが存在することをテストします。デフォルトは定義されていません。
- `#define USER_CFG_ERR_CHECK_NULL_PTR`

  参照読み出しの前にポインタがヌルでないことをテストします。デフォルトは定義されていません。
- `#define USER_CFG_ERR_CHECK_MEMORY_PACK`

  メモリ パッケージのエラー状態をテストします。デフォルトは定義されていません。
- `#define USER_CFG_ERROR_MESS_STDOUT`

  エラー メッセージを `sdtout` に表示するかどうかを定義します。デフォルトは定義されていません。
- `#define USER_CFG_ERROR_MESS_STDERR`

  エラー メッセージを `stderr` に表示するかどうかを定義します。デフォルトは定義されていません。
- `#define USER_CFG_USE_ERR_MESS`

  エラー番号だけでなく、エラー メッセージも表示するかどうかを定義します。デフォルトは定義されていません。
- `#define USER_CFG_WARN_ACTION`

  この設定をすると、次のユーザー提供の関数

  ```
  void xUserWarnAction (unsigned char WarningNumber);
  ```

  が警告時に呼び出されます。デフォルトは定義されていません。
- `#define USER_CFG_ERR_ACTION`

  この設定をすると、次のユーザー提供の関数

  ```
  void xUserErrAction (unsigned char WarningNumber);
  ```

  がエラー時に呼び出されます。デフォルトは定義されていません。

その他

- `#define USER_CFG_USE_xInitEnv`
- `#define USER_CFG_USE_xCloseEnv`
- `#define USER_CFG_USE_xInEnv`
- `#define USER_CFG_USE_xOutEnv`

  上記を定義すると、対応する環境関数の呼び出しが組み込まれます。デフォルトは定義されていません。

- `#define USER_CFG_ADD_STDIO`

  `stdio.h` をインクルードするかどうかを定義します。これは、USER_CFG_UML_TRACE_STDOUT、USER_CFG_ERROR_MESS_STDOUT、USER_CFG_ERROR_MESS_STDERR のいずれかが定義された場合に、自動的に実行されます。デフォルトは定義されていません。

- `#define USER_CFG_UML_TRACE_STDOUT`

  UML レベルのトレースを `stdout` に表示するかどうかを定義します。デフォルトは定義されていません。

注記

AgileC コード ジェネレータの機能選択のためのユーザー インターフェイスには、`uml_cfg.h` ファイルへコピーされるフリー テキストを入れる機能もあります。この機能を使用して、ユーザー インターフェイスで直接選択できない他の定義を、組み込むことができます。

Application Builder での選択

Application Builder には、コード ジェネレータに渡されるが `uml_cfg.h` ファイルには影響を与えないオプションがあります。このオプションは、デフォルトでできるだけ読みやすいコードの生成を可能にするためのものです。読みやすさにかかわる特定の機能が必要な場合、その機能は個別に選択できます。

- 生成コードに含まれる**コメント**量

  状態機械について生成されるコード内で、コードに付加するコメントの量を選択できます。以下の選択肢があります。

  - **Sparse**：遷移を特定するために使用される一部のコメントだけを含みます。
  - **Structure**：レベル 1 に変換された各 UML シンボルのコメントが追加されます。
  - **Explanation**：レベル 2 にコードを説明するコメントが追加されます。

- 生成コードにランタイム テストのためのコードを含める

  この機能は、以下のいずれかのランタイム テストが選択された場合に選択する必要があります。
  USER_CFG_ERR_CHECK_INDEX、
  USER_CFG_ERR_CHECK_RANGE、
  USER_CFG_ERR_CHECK_DECISION、
  USER_CFG_ERR_CHECK_NULL_PTR

- 生成コードに UML ソースへの参照を C コメントとして含める

  この参照を使用して、コードから UML ソースまで、逆方向にナビゲートできます。

- 生成コードにテキスト実行トレースのためのコードを含める

  この機能を選択すると、生成されたコードにトレース関数への呼び出しを含めるように、コード ジェネレータに指示します。USER_CFG_UML_TRACE_STDOUT を有効にする場合は、これを有効にする必要があります。

- UML 名への**名前変換のための**接頭辞または接尾辞

  UML と C では名前スコープは異なります。したがって、生成した C コード内では、UML 名を直接は使用できません。デフォルトでは、コード ジェネレータが UML 名に接尾辞を追加し、それを C の識別子として使用して、その識別子が確実に一意となるようにします。比較的長い UML 名を使用していて、かつ、C コンパイラが名前の同一性を確認する際の検査文字数が少ない場合は、名前の衝突が発生する可能性があります。この事態を回避するには、接尾辞の代わりに接頭辞を使用します。

概念によるパフォーマンスの変化に関する情報

一部の UML 概念は、パフォーマンスが重視される小型システムで使用すると、問題を引き起こす可能性があります。生成アプリケーションのパフォーマンスを改善する方法の 1 つとして、このような言語概念を使わない、という方法があります。

もっとも注意すべきなのは、パートのインスタンス数が無制限な場合、つまり、パートのコンカレントインスタンスの最大数の上限を指定しない場合です。上限を設定した場合、コード ジェネレータはその値を使用して、このパートのインスタンスについて、インスタンス データの静的配列を宣言します。上限を設定しない場合、インスタンスの数に見合った動的メモリの割り当てを行う必要が生じ、さらに、メモリが不足した場合には、`realloc` を実行する必要が生じます。`realloc` の実行は、新しい大きなデータ領域が割り当てられて、旧領域のすべてのデータが新領域にコピーされる、ことを意味します。そして、インスタンス数の上限を持たないパートに対する create 操作も伴ないます。これら一連の動作には、非常に時間がかかる可能性があります。

「保存」概念の使用も、効率の低下を招く可能性があります。なぜならば、保存されたシグナルが頻繁にアクセスされるためです。「ガード」概念についても同じことが言えます。これは、「ガード」概念が「保存」概念を暗黙的に要求しているためです。ただし、これらの概念が、システムの振る舞いの実装に現実に役立つ場合は、まずこれらの概念を使用してシステムを記述すべきです。しかし、その後、オーバーヘッドを回避する必要が生じた場合は、これらの概念を削って代替表現ができないかどうかを検討して、システムを記述し直します。

USER_CFG_MSG_BORDER_LEN を設定する `uml_cfh.h` ファイルは、パラメータ付きのシグナル送信パフォーマンスに重要な意味を持つ可能性があります。パラメータがシグナルのデータ領域に適合しない場合には、新しいデータ領域の割り当てと使用後のその領域の開放が必要になります。これによって、実行速度は落ちるでしょう。一方 USER_CFG_MSG_BORDER_LEN の値を大きくすると、すべてのシグナルが大きくなり、メモリの無駄につながります。

重要なデータ構造の概要

このセクションで説明する重要な型は、すべて uml_kern.h に定義されています。このセクションを読む際には、いつでもこのファイルを参照できるようにしてください。

生成されたコードには、システムの各パートに関する情報が含まれています。パートごとに、以下のように、構造体変数型 xPartTable が生成されます。その例を以下に示します。

```
xPartTable xPartData_p_01 = {
  (xInstanceData *)xInstData_p_01,
  sizeof(yVDef_p_01),
  1,
  1,
  (xIniFunc)yIni_p_01,
  (xTransFunc)yPAD_p_01,
  xTransitionData_p_01,
  xStateIndexData_p_01
#ifdef CFG_USED_GUARD
,  0
#endif
};
```

生成されたコードには、システムのすべてのパートの xPartTable 構造体へのアドレスを格納した配列もあります。このグローバル変数は、パートに関する情報にアクセスするために、多くの場所で使用されます。その例を以下に示します。

```
xPartTable *xPartData[] =
{
  &xPartData_p_01,
  &xPartData_q_02
};
```

xPartTable 構造体の内容を調べると、以下のコンポーネントが含まれています。

コンポーネント	説明
InstanceData	パートの各インスタンスの 1 つの要素を持つ配列へのポインタ。これらのコンポーネントは、ステートおよびローカル変数値と同様に、さまざまな種類のインスタンス固有の値を格納するために使用される。
DataLength	上のコンポーネントで述べた各配列要素のサイズ。
MaxInstances	このパートのコンカレントインスタンスの最大数。
InitialInstances	システム起動時のインスタンス数。
yPAD_Function	パートの状態機械を実装する関数への参照。

コンポーネント	説明
TransitionTable および StateIndexTable	特定の状態の特定のシグナルの処理方法を決めるために使用するテーブル（配列）。
GuardFunc	状態機械で使用するガード式を算出できる関数への参照。
ThreadNumber	このパートが属するスレッド。スレッド インテグレーションでのみ使用されます。

他の重要なグローバル データ構造として、利用可能なシグナルとタイマーの配列があります。以下を参照してください。

```
/* Signal array */
static xSignal xSignalArray[CFG_STATIC_SIGNAL_INSTS];

/* Pointer to first element in list of free signals */
static xSignal *xFreeSignalList;

/* Timer array */
static xTimer xTimerArray[CFG_STATIC_TIMER_INSTS];

/* Pointer to first element in list of free timers */
static xTimer *xFreeTimerList;
```

これらの変数は uml_kern.c ファイル内にあり、利用可能なシグナルとタイマーのリストの開始ポインタのほかに、シグナルとタイマーの配列を定義しています。

最後の重要なグローバル データ構造として、xSysD と呼び出される xSystemData 型変数があります。非スレッド アプリケーションでは、xSysD は xSystemData 型の変数ですが、スレッド アプリケーションでは、xSysD は、スレッドごとに 1 つの xSystemData 型のコンポーネントを持つ配列です。

xSysD には、アプリケーションまたはスレッドで現在発生している事象についてのグローバル情報が含まれています。以下のコンポーネントがあります。

コンポーネント	説明
CurrentPid	アプリケーションまたはスレッドで現在実行しているインスタンスの Pid 値。
CurrentSymbolNr	状態機械を実装する関数内で実行を開始点となるシンボル番号。この関数の先頭にある switch 文で選択が行われます。
CurrentData	実行しているインスタンスのローカル変数へのポインタ。この参照は、パートの xPartTable の InstanceData コンポーネント内の該当する要素を参照します。
CurrentSignal	現在の実行の遷移を引き起こしたシグナルへのポインタ。

コンポーネント	説明
SignalQueue	アプリケーションまたはスレッドに対してグローバルな グローバルシグナル キューの先頭のシグナルへの参照。この待ち行列内のシグナルは、リンク リストで連結されています。
ExternSignalQueue	他のスレッドまたは他の環境から来るシグナルが格納されるシグナル待ち行列です。遷移間の特定の点で、シグナルは SignalQueue へ移されます。
OutputSignal	生成コードでシグナルを送信している間に使用される、一時変数。
TimerQueue	アプリケーションまたはスレッドに対してグローバルなグローバル タイマー キューの先頭のタイマーへの参照。
ThreadVars	スレッド アプリケーションの場合は、スレッド自身に関するデータを保持している構造体です。内容はインテグレーションによって異なります。

さらに詳細に説明する必要があるデータ構造として、Pid 値、およびシグナルとタイマーの内容があります。

Pid 値は実行しているインスタンスへの参照です。Pid 値は 2 つの部分から構成されます。パート番号（パートには 0,.1... と番号が付き、この番号がグローバルな xPartData 配列のインデックスとして使用される）とインスタンス番号（パートの xPartTable の InstanceData 配列のインデックスとして使用される）です。Pid 型は符号なしの型で、この 2 つの値を保持するのに適したサイズをもっています。

シグナルは xSignal 型によって定義され、以下のコンポーネントをもっています。

コンポーネント	説明
Next	リスト内のシグナルをリンクするためのポインタ。
Sid	シグナルの ID（シグナル型）
Receiver	受信側の Pid 値。
Sender	送信側の Pid 値。
SaveState	保存したシグナルの処理を高速化するために使用されます。
Prio	シグナルの優先順位（シグナルに［Use priorities］が設定されている場合）。
ParPtr	シグナルのパラメータへのポインタ。ParArea か、割り当てられたメモリのいずれかを参照します。
ParArea	シグナル パラメータのインライン領域で、このパラメータがこの領域に適合する場合に使用されます。

タイマーは xTimer 型によって定義され、以下のコンポーネントをもっています。

コンポーネント	説明
Next	リスト内のタイマーをリンクするためのポインタ。
Sid	タイマーの ID（タイマー型）
Owner	タイマーの所有者の Pid 値
Time	タイマーに設定される時間。
TimerParValue	タイマーのオプションの整数パラメータ。

パッシブ クラスの変換

UML のパッシブ クラスは、C のデータ型と操作に変換されます。AgileC コード ジェネレータについては、C コード ジェネレータ リファレンスの場合と同じです。生成された C コードの名前 に関するセクションは、AgileC コード ジェネレータに対しても有効です。

参照

第 20 章「アプリケーション ビルド リファレンス」の 816 ページ、「C アプリケーション ビルド時の UML サポートの制限事項」

32

C コンパイラ ドライバ

C コンパイラ ドライバ（CCD）は、生成した C コードのデバッグを簡単にするためのユーティリティです。C コンパイラ ドライバは、アプリケーション ビルダによって使用される make ファイルから呼び出すことができ、または OS のコマンド行から単独で起動することもできます。

CCD の適用エリア

C コンパイラ ドライバ（CCD）は、ユーザー定義のディレクトリに C の中間ファイルを生成することによって、C のデバッグを簡単にするユーティリティです。この C ファイルではすべての C マクロが展開されており、オプションで見映えを整えることもできます。使いやすさという点から、CCD を 1 つの C コンパイラ ドライバとして使用する、つまり C コード ジェネレータが生成する make ファイルから呼ばれる形にします。また必要であれば、ユーザー コマンドで C コードをコンパイルする際に、CCD をコマンド行オプションとしても使用できます。

CCD ユーティリティはすべてのコンパイラでサポートされる機能ではありません。ほとんどの作成済みカーネルで、作成済み CCD 設定ファイルは、通常のカーネル ファイルの一部ではありません。

CCD の利用方法

この機能を UML ツール セットで使用するためのもっとも簡単な方法は、使用するランタイム ライブラリに含まれる `makeoptions/make.opt` ファイルを修正することです。この機能をユーザ定義のランタイム ライブラリ内で有効にするには、ディレクトリとファイルの構造を Tau にあらかじめ定義された構造と同じにする必要があります。

必要な修正は次の行を、

```
sctCC = cc
/* or some other compiler executable name */
```

以下のように変更すること**だけ**です。

```
sctCC = sccd cc
/* or some other compiler executable name */
```

パスの変数に Tau 実行形式バイナリ ファイルのあるディレクトリを含む必要があります。たとえば Windows の場合以下のようになります。

```
C:¥Program Files¥IBM¥Rational¥TAU¥4.3¥¥bin
```

CCD のカスタマイズ

CCD の振る舞いをカスタマイズするには、設定ファイル `sccd_<your_compiler_type>..cfg` 内の変数を変更します。この変更の方法については、1132 ページの「CCD の振る舞い」を参照してください。

CCD ユーザー インターフェイス

CCD コマンド行インターフェイスの構文を以下に示します。

```
sccd [command] [option]
```

C ファイルのコンパイル

CCD を使用して C ファイルをコンパイルするには、次のコマンドを使用します。

```
sccd <C compiler command line>
```

パラメータは、通常の方法で C ファイルのコンパイルに使用されるコマンド行となります。このコマンドには、ユーザーの C コンパイラの名前、必要に応じたコンパイラ オプション、最後に C ファイルの名前が含まれます。

印刷の設定

設定ファイル `sccd.cfg` 内の変数の値を表示するには、オプションなしで次のコマンドを使用します。

```
sccd
```

ヘルプ情報を使って変数を表示するには、次のコマンドを実行します。

```
sccd -h
```

戻りコード

0: 成功。

1: `sccd` が「設定とヘルプ」を表示後の戻りコード。

2: `.c` 入力ファイルがありません。

3: `InFile.c` 入力ファイルを開けませんでした。

4: 書き込み対象の `InFile.i` を開けませんでした。

5: `sccdMOVE` または `sccdOUTFILEREDIR` を定義する必要があります。

6: 読み込み対象の `InFile.i` を開けませんでした。

7: `TmpDir/InFile.c` を開けませんでした。または作成できませんでした。

CCD によって実行されるアクション

CCD は以下の一連のアクションを実行します。

1. オプションのユーザ定義コマンドを実行する（`sccdUSER_CMD1`）。
2. 一時ファイルとプリプロセス `.c` ファイルのためのサブディレクトリを作成する。
3. オプションの 2 番目のユーザー定義コマンドを実行する（`sccdUSER_CMD2`）。
4. C プリプロセッサ パスを実行してすべての C マクロを展開する。
5. オプションの 3 番目のユーザー定義コマンドを実行する（`sccdUSER_CMD3`）。
6. そのファイルをプリティプリントする。
7. 選択によって、サブディレクトリをクリーンアップする。
8. 選択により、`.hs` ファイルをサブディレクトリにコピーする。

9. オプションの 4 番目のユーザ定義コマンドを実行する（sccdUSER_CMD4）。CCD から「indent」ユーティリティを呼び出したい場合、このコマンドを使用します（詳細については、1132 ページの「C ビューティファイア（C Beautifier）」を参照してください）。
10. 選択により、コンパイルする。すなわち、元のコマンドを実行する。
11. 選択により、サブディレクトリをクリーンアップする。ただし、デバッグ目的のプリプロセス .c ファイルは残します。

C ビューティファイア（C Beautifier）

C コード ジェネレータが生成した C コードをさらにフォーマットするために C ビューティファイアが必要な場合は、indent ユーティリティ（Joseph Arcaneaux 氏からの無償提供）を試みるとようでしょう。

indent 実行形式ファイルはユーザーのパスに置く必要があります。

indent を CCD から簡単に起動するには、該当する sccd.cfg ファイルの sccdUSER_CMD4 に以下の文を挿入します。ただし、sccd.cfg ファイルにその他の変更を加えていないことが前提です。

UNIX コンパイラ環境の場合：

```
sccdUSER_CMD4 = "indent -kr -l70 -i2 %p/%d/%f.c"
```

DOS のようなコンパイラ環境の場合：

```
sccdUSER_CMD4 = "indent -kr -l70 -i2 %p¥¥%d¥¥%f.c"
```

この設定によって、.c ソースファイルは、Kernighan & Ritchie の『The C Programming Language』で使われている規則に類似した規則に従ってフォーマットされます。また、このユーティリティは 1 行の文字数を 70 文字未満に整え、if/while/.. 文では 2 段のインデントを使用します。

例 **401**

やや手の込んだインデントの使い方の例を以下に示します。

```
sccdUSER_CMD4 = "indent -kr -l70 -br -nce -nlp -ci3 -i2
%p/%d/%f.c"
```

CCD 設定ファイル

CCD の振る舞い

CCD の振る舞いは、sccd_ で始まる変数を使用して定義します。これらの変数の定義は設定ファイル sccd_<your_compiler_type>..cfg で行います。

- 自分の C コンパイラに対応した設定ファイルを選択し、このファイルを `sccd.cfg` という名前でコピーします。UML ツール セット フレームワーク内から実行する場合、CCD は `$sctdir/sccd.cfg` を設定ファイルとして使用します。それ以外の場合、CCD はカレント ディレクトリ `$HOME`（**UNIX**）と `$SCCD` から `sccd.cfg` を探します。

注記

`sccd.cfg` が見つからなかった場合、GNU C コンパイラ（gcc）に最適なハードコードされたデフォルトが使用されます。

CCD の変数

CCD の振る舞いを制御する変数を以下に説明します。変数値に含まれるすべての文字は、スペースも含めて意味があります。

sccdNAME

```
sccdNAME = "Default"
```

`scttypes.h` に定義されるコンパイラ名

sccdINFILESUFFIX

```
sccdINFILESUFFIX = ".c"
```

インファイルのファイル名の予定の接尾辞。デフォルトは `"..c"`。

sccdCPP

```
sccdCPP = ""
```

C プリプロセッサの名前。これが空の場合、CPP が使用されます。デフォルトは `""` です。

sccdCPPFLAGS

```
sccdCPPFLAGS = ""
```

CPP を有効にし、コメントを削除しません。これは C コンパイラによって決まります。デフォルトは、gcc の場合は `"-P -E -C"`、cc の場合は `"-C -P"` です。

sccdMACROPREFIX

```
sccdMACROPREFIX = "-D"
```

CPP コマンド行の define MACRO の接頭辞。デフォルトは `"-D"` です。

sccdINCLUDE1

```
sccdINCLUDE1 = "-I"
```

CPP コマンド行インクルード パスの接頭辞。デフォルトは `"-I"` です。

sccdINCLUDE2

```
sccdINCLUDE2 = ""
```

CPP コマンド行インクルード パスの代替の接頭辞。デフォルトは "" です。

sccdOUTFILEREDIR

```
sccdOUTFILEREDIR = "-o "
```

CPP 出力ファイル名を制御する文字シーケンス。空の場合、代わりに sccdFMOVE を使用します。

sccdFMOVE

```
sccdFMOVE = ""
```

OS の強制的なファイル移動コマンドまたはファイル コピー コマンド。
sccdOUTFILEREDIR の代わりに使用されます。デフォルトは “” です。

sccdDELETE

```
sccdDELETE = "rm -f"
```

OS の強制的なファイル削除コマンド。デフォルトは "rm -f" です。

sccdCOPY

```
sccdCOPY = "cp"
```

OS の標準のコピーコマンド。デフォルトは "cp" です。

sccdCOMPILE

```
sccdCOMPILE = "ON"
```

最後のコンパイル パスを実行するかどうかを制御します。値は "OFF" デフォルトは "ON" です。

sccdDEBUG

```
sccdDEBUG = "OFF"
```

実行を有効にします。値は "ON" デフォルトは "OFF" です。

sccdPURGE

```
sccdPURGE = "ON"
```

一時ファイルを消去します。値は "OFF" デフォルトは "ON" です。

sccdUSE_HS

```
sccdUSE_HS = "OFF"
```

"ON" に設定すると、.hs ファイルは、コンパイルがパスするまでインクルードされません。値は "OFF" デフォルトは "ON" です。

sccdSILENT

```
sccdSILENT = "OFF"
```

トレースのプリントアウトを有効にします。値は “ON” デフォルトは “OFF” です。

sccdTMPDIR

```
sccdTMPDIR = "sccdtmp"
```

プリプロセスのための一時ディレクトリ。デフォルトは sccdtmp です。設定ファイルで sccdTMPDIR を "" または "." に設定すると、一時ディレクトリは生成されません。

sccdUSER_CMD1、sccdUSER_CMD2、sccdUSER_CMD3、sccdUSER_CMD4

```
sccdUSER_CMD1 = ""
sccdUSER_CMD2 = ""
sccdUSER_CMD3 = ""
sccdUSER_CMD4 = ""
```

ユーザー定義コマンド（1131 ページの「CCD によって実行されるアクション」）。一番目の変数（sccdUSER_CMD1）を除くすべての変数で、以下の擬似変数を使用できます。

- %f 拡張子のない入力ファイル名が展開されます。
- %p 入力ファイル パスが展開されます。
- %d sccdTMPDIR. の値が展開されます。

例 **402**

```
echo ¥"Pre-processed C-file = %p/%d/%f.c¥"
```

‘#’ を sccdUSER_CMDx と sccdTMPDIR で使用するには、¥# を入力します。

‘"’ を sccdUSER_CMDx と sccdTMPDIR で使用するには、¥" を入力します。

‘¥’ を sccdUSER_CMDx と sccdTMPDIR で使用するには、¥¥ を入力します。

UML と Java

「**UML と Java**」セクションの各章では、Java コードのエンジニアリングに UML ツール セットを使用する方法について説明しています。

Tau での Java サポートと Eclipse インテグレーションは、Windows OS でのみサポートされます。

33

Java サポート

このセクションでは、Java サポートの使用法について説明します。既存の Java アプリケーションのインポートと Java ラウンドトリップ サポート（コード生成とリバース エンジニアリング）によって、Tau 機能を拡張します。Java コードをコンパイルして実行したり、javadoc と JAR ファイルを生成するなど、さまざまな機能があります。

また、テキスト図やテキスト シンボルに Java 構文を使用して（UML 以外に）、設計言語を自由に選択することもできます。2 つの構文はいつでも自由に切り替えられます。

既存の Java ファイルは、UML モデルにインポートして簡単に再利用できます。また、既存の UML モデルから Java を生成することもできます。

Java プロジェクトの作成

Java プロジェクトを作成するには、以下の手順を行います。

1. ［ファイル］メニューから［新規 ...］を選択します。
2. プロジェクトの作成時にプロジェクトのタイプとして［UML(Java コード生成用)］を選択します。生成された Java コード用に Eclipse IDE を使用したい場合は、［UML(Java Eclipse プロジェクト用)］を選択します。
3. 使用したい Java 固有表現として、Java 1.4 または Java 5 を選択します。この選択は、デフォルト ライブラリなどに影響があります。

プロジェクトの作成方法については、プロジェクトの操作を参照してください。

既存のプロジェクト用に Java サポートをアクティブにするには、次の手順を行います。

1. ［ツール］メニューから［カスタマイズ］ダイアログを選択します。
2. ［アドイン］タブをクリックして、［JavaApplication］アドインにチェックを付けて選択します。
3. ［閉じる］をクリックします。

注記

Java サポートをアクティブにしてプロジェクトを作成またはロードすると、Java ランタイム ライブラリ全体がライブラリとしてロードされます。このライブラリは非常に大きいため、完全にロードされるまでに数分かかることがあります。ライブラリのロード中、Tau はユーザーによる対話操作に応答しません。

Java サポートでは、Sun の Web サイトからダウンロードできる Java Software Development Kit を使用します。

```
http://java.sun.com/javase
```

Java ビュー

Java サポートには、［Java View］というモデルの Java 中心のビューがあります。このビューでは、Java 言語の仕様にあるモデル要素のみを表示するため、［Standard View］とは要素の表示の仕方が異なります。UML 特有の構築子、たとえばシグナルや状態機械は、［Java View］には表示されません。Java サポートは、主に［Java View］で使用するように設計されています。

モデルが UML 固有の構築子を使っていない場合は、［Java View］を使うと便利です。

参照

モデル ビュー

［**Java View**］のアクティブ化

［Java View］をアクティブにするには、次の手順を行います。

1. ［表示］メニューで、［モデル ビューの再構成］を選択します。
2. ダイアログで［Java View］を選択して、［OK］をクリックします。

［モデル ビュー］は、［Standard View］と［Java View］間でいつでも切り替えられます。

［**Java View**］での作業

［Java View］の使用中は、Java 関連の要素のみを表示したり作成したりできます。他の要素は除外されるため、［Standard View］でアクセスする必要があります。

［Java View］で使用可能な要素は、以下のように Java 言語で自然に使用できる要素です。

- Java パッケージ、クラス、インターフェイス、属性、操作

および、これらの要素を完全に定義するために必要な以下のダイアグラム

- クラス図とテキスト図

さらに、このビューでは以下の Java ファイルを表すアーティファクトも使用できます。

- Java ファイル、JAR ファイル、Java ビルドアーティファクト

Java パッケージの作成など、操作の一部は［Java View］でのみ実行できます。Java ソース コードのインポートのように、任意のビューで実行できる操作もあります。また、最上位レベルのパッケージのエクスポートのように、一部の操作は、［Standard View］でのみ実行できます。パッケージは Java パッケージでなければ［Java View］では表示できず、エクスポートの目的はそのパッケージを Java パッケージにすることなので、これは当然のことです。

［Java View］には、Java モデルと Java ライブラリという、2 種類の最上位レベルがあります。Java モデル ノードは、ユーザー モデルの要素を表示する［Standard View］のモデル ノードに対応します。Java ライブラリ ノードには、定義済みのデータ型や java.lang などの定義済み Java ライブラリが含まれます。

［**Java View**］と［Standard View］の違い

［Java View］は、Java 要素をその言語とファイル システムに近い状態で自然に表せるように設計されていますが、［Standard View］は、UML に似た形式で要素を表示します。この 2 つのビューが本質的に異なる場合があります。これは、JAR ファイル アーティファクトを例として取り上げるとよくわかります。

モデル内で、JAR ファイルは以下の要素によって表されます。

- 物理 JAR ファイルを表すアーティファクト
- JAR に含まれるエンティティ

- アーティファクトからファイルのエンティティへの依存関係

アーティファクトと、JAR に含まれるエンティティは、モデル内で同じレベルにあります。依存関係はアーティファクトによって所有されます。これは、UML でのファイルの内容の指定方法です。

［Java View］では、JAR に要素が含まれている事実が強調されます。したがって、要素はアーティファクトによって所有されているかのように表示されます。JAR 内のエンティティはアーティファクトと同じレベル*ではなく*、アーティファクトの下のレベルで表示されます。［Java View］では、依存関係はまったく表示されません。その違いは以下の表で確認できます。

Standard View	**Java View**
アーティファクト 　　要素 1 への依存関係 　　要素 2 への依存関係 要素 1 要素 2	JAR ファイル アーティファクト 　　要素 1 　　要素 2

［Java View］と［Standard View］を切り替えるときはこれらの違いに注意してください。

Java ビルドアーティファクト

<<Java>> ステレオタイプの適用されたアーティファクトは、Java ビルドアーティファクトです。Java ビルドアーティファクトは、Tau で Java を使った作業を行う際に実行するさまざまなコマンドの、スコープと意味を定義します。

- Java ビルドアーティファクトは［Java］コンテキストメニューでコマンドを提供します。このコマンドを使って、Java コードの生成、コンパイル、生成されたソースコードの変更に伴うモデルの更新を行うことができます。
- Java ビルドアーティファクトは、<<manifest>> 依存を使用して、1 つまたは複数の UML 要素をマニフェストします。つまり、Java ビルドアーティファクト上で実行されるコマンドに影響される UML モデルの要素が何なのかを定義します。ビルドアーティファクトコマンドは、ビルドアーティファクトによって間接的にマニフェストされている要素にも適用されることに注意してください。たとえば、直接的にしろ間接的にしろ、” マニフェストされた要素に所有されている要素 ” などがこの例です。
- Java ビルドアーティファクトは、オプションと設定を <<Java>> ステレオタイプのタグ付き値の形で格納します。これらのオプションは、通常は、コンテキストメニューで起動したコマンドの実行結果に影響を与えます。たとえば、［Target Directory］オプションは Java ビルドアーティファクトに保存され、UML モデルのマニフェストされた要素と、ファイルシステム内の対応するソースコードとの間の関係付けを指定します。

ビルドアーティファクトの詳細については、ビルド アーティファクトとビルド アーティファクトの使用を参照してください。

Java ビルドアーティファクト コマンド

Java ビルドアーティファクト上のコンテキストメニュー［Java］から利用可能なコマンドは以下のとおりです。

Generate

このコマンドは、Java ビルドアーティファクトにマニフェストされたすべての要素について Java ソースコードを生成するために使用します。最後の生成以降修正された UML 要素だけが、翻訳の対象です。モデルに変更がない場合は、Java ソースファイルは生成されず、“up-to-date” メッセージが［ビルド］タブに出力されます。

［Java］メニューには［ソースコードの更新］コマンドがあります。このコマンドを使うと、［Generate］コマンドをモデル内のすべてのビルドアーティファクトについて実行します。

Generate (force)

このコマンドは、Generate と同様のはたらきをしますが、マニフェストされた要素について、対応する UML 要素が修正されたかどうかにかかわらず、すべての Java ソースファイルを強制的に生成します。

［Java］メニューには［ソースコードの強制更新］コマンドがあります。このコマンドを使うと、［Generate（Force)］コマンドをモデル内のすべてのビルドアーティファクトについて実行します。

Update

このコマンドは、生成済みの Java ソースファイルで変更した要素の変更内容を、モデルのマニフェストされた要素に反映するために使用します。このやり方は、一般に " ラウンドトリップエンジニアリング " と呼ばれます。生成後に変更された Java ファイルのみが処理の対象です、生成後に何の変更もくわえられなかった場合は、コマンドを実行しても何も起こらず、“up-to-date” メッセージが［ビルド］タブに出力されます。

［Java］メニューには［モデルの更新］コマンドがあります。このコマンドを使うと、［Update］コマンドをモデル内のすべてのビルドアーティファクトについて実行します。

Update (force)

このコマンドは、Update と同様のはたらきをしますが、マニフェストされた要素について、対応する Java ソースファイルが修正されたかどうかにかかわらず、ソースファイルから強制的に更新します。

［Java］メニューには［モデルの強制更新］コマンドがあります。このコマンドを使うと、［Update（force)］コマンドをモデル内のすべてのビルドアーティファクトについて実行します。

Build

このコマンドは、マニフェストされた要素に対応する Java ソースファイルをコンパイルするために使用します。コマンドは、まず Generate コマンドを実行してモデルから最新のソースファイルを生成して、その後コンパイルを実行します。このコマンドの詳細については、Java のコンパイルと実行を参照してください。

Java ビルドアーティファクト設定

Java ビルドアーティファクトは、設定内容を <<Java>> ステレオタイプのタグ付き値の形で格納します。この設定を表示、編集するには以下の手順で行います。

1. ［モデルビュー］で Java ビルドアーティファクトを選択します。
2. コンテキストメニューで［ビルド設定］を選択します。

この操作でプロパティ エディタが開きます。設定はこのプロパティエディタから行います。

一部の設定はすべてのビルドアーティファクトに共通です。これらの共通設定はビルド アーティファクトの使用の章で説明します。ここでは Java 特有の設定を説明します。

Perform transformations

デフォルトで、Java コードジェネレータは UML から Java への翻訳のために数多くの変換を実行します。対応するネイティブな Java 構築子のない UML 構築子については、正しい Java プログラムを取得するには、必ずこの変換を行う必要があります。内部的には、コード生成中に、ユーザーモデルの対象部分が新しく作成した一時モデルにコピーされています。この一時モデルを使用して、元のモデルに影響を与えないように変換が実行されます。

コード生成するモデルで、Java へのネイティブなマッピングのある UML 構築子（たとえばクラス、インターフェイス、パッケージなど）のみを使っている場合は、この一時モデルの作成をオフにすることで、コード生成がスピードアップします。ただし、これをオフにすると、Java コードジェネレータが実行する変換だけではなく、すべての種類の変換が行われなくなることに注意してください。つまり、生成コードをカスタマイズするためにアドインで追加したカスタム変換処理も行われなくなります。

Classpath

コンパイル時、実行時に使用される classpath 変数の値を手動で書き換えるために使用します。

Automatic source generation

この設定がオンになっていると、モデルを変更したときに Java ソースファイルは自動的に更新（再生成）されます。更新動作は、編集作業が終わった直後に行われます。デフォルトではこの設定はオフです。

Automatic model update

この設定がオンになっていると、Java ソースファイルを変更（外部エディタなどで保存する、など）したときにモデルは自動的に更新されます。デフォルトではこの設定はオフです。

Support roundtrip

生成した Java ソースファイルを変更して Update コマンドを使ってモデルに変更を反映するという作業方法を行う要請がない場合は、この設定はオフにしておけます。オフにしておくことで、Java コード生成は常に既存のファイル内容の削除と一からの再生成という形で行われます。

この設定をオンにするメリットは、コード生成のパフォーマンスの向上と生成されたファイル内のインデント設定が一貫する点です。

Apply <<informal>> stereotype to imported methods

この設定はデフォルトでオンです。この場合、インポートされた Java メソッドの操作本体の正当性を UML レベルでチェックする、というアクションは行われません。

Java ファイル

このセクションでは、既存の Java ファイルを使った作業、つまり UML モデルへの Java ファイルのインポートについて説明します。既存の UML モデルからの Java ファイルの生成方法についても説明します。

既存の Java アプリケーションのインポート

このセクションでは、既存の Java ファイルを UML モデルにインポートして作業する方法について説明します。

既存の Java アプリケーションを Tau にインポートするには、以下の手順を行います。

1. ［ファイル］メニューから［インポート］を選択します。
2. インポート ウィザードで［Import Java source code］を選択して、［OK］をクリックします。
3. ツリー表示からインポートしたいディレクトリを参照します。複数のディレクトリを選択して、同時にインポートすることもできます。

4. 同じページで、他にもいくつかオプションを選択できます。
   - ［Create diagrams］オプションを選択すると、インポート後の各パッケージに対してダイアグラムが作成されるように設定できます。このダイアグラムには、パッケージとその関係に含まれるすべてのクラスが表示されます。
5. ［Finish］をクリックしてインポート ウィザードを終了します。指定した各ディレクトリに対応する UML モデルに、1 つの Java パッケージが作成されます。

指定した各ディレクトリに対応する UML モデルは、個別のファイルに保存されます。

選択したフォルダ内のすべての Java ファイルが解析され、モデルに挿入されます。選択したフォルダ内のすべてのフォルダは、最上位レベルの Java パッケージとして挿入されます。選択したフォルダ自体に含まれているファイルは、名前のないパッケージで定義されたものとして扱われ、その場所に挿入されます。

インポートは繰り返し実行されるので、サブ フォルダはすべてサブ パッケージとしてモデルに挿入されます。Java ファイルとモデルのマッピングについては 1161 ページの「モデルからファイルへのマッピング」を参照してください。

インポートの結果のフォルダごとに、Java ビルドアーティファクト が追加されます。インポートされたフォルダのパスは Java ビルドアーティファクトの ‘Target Directory’ に格納されます。ディレクトリをインポートした場合は、同期ターゲットディレクトリ のリストに追加されます。モデルの Java パッケージとファイル システムのフォルダ間の関連付けは、パッケージのステレオタイプにフォルダパスを保存することで、保持できます。

Java ファイルがインポートされたら、Java メニューの［モデルの更新］コマンドと［ソースコードの更新］コマンドを使用して、モデルとソース コードの同期をとります。Java ビルドアーティファクトのコンテキストメニュー［Java］から、［Update］と［Generate］を使っても同じことができます。

ソース コードを編集するには、ファイルを表示する Java ファイル アーティファクトをダブルクリックするか、モデル要素、または、プレゼンテーション要素の［ソースコードの編集］コマンドを実行します。その結果、`.java` ファイルが組み込みテキスト エディタで開きます。

JAR ファイルのインポート

JAR ファイルの内容はモデルにインポートされるので、JAR ファイルの定義をモデル内で参照できます。定義のシグニチャのみがインポートされ、実装はインポートされません。

既存の JAR ファイルを Tau にインポートするには、以下の手順を行います。

1. ［ファイル］メニューから［インポート］を選択します。
2. インポート ウィザードで［Import JAR file］を選択して、［OK］をクリックします。
3. インポートしたい JAR ファイルを参照して選択します。

4. ［Open］をクリックしてインポート ウィザードを終了します。指定した各 JAR ファイルに対応する UML モデルに、1 つの Java パッケージが作成されます。パッケージにはクラス図が生成されて、インポートされたクラスとそれらの関係が示されます。

JAR ファイルは解析されてモデルに挿入されます。結果モデルのストラクチャは、JAR ファイルの内容に従います。通常、トップ レベル パッケージはひとつだけ作成されますが、JAR ファイルは複数の最上位レベルのパッケージを含むことができます。

JAR ファイルに対応する各パッケージは、個別の UML ファイルに保存されます。UML ファイルの名前は、JAR ファイルの名前に応じて付けられます。

また、`.jar` ファイルを表示するために、JAR ファイル アーティファクトが作成されます。アーティファクトからインポート後のパッケージへの依存関係が生成されます。

パッケージとパッケージの JAR ファイル間の関連付けは、アーティファクトのステレオタイプの JAR ファイルへのパスを保存することで、保持されます。

JAR ファイルのインポートにより作成されたパッケージは、静的ライブラリとして処理されるため、他のルート パッケージと比較して多くの制限があります。

- パッケージは JAR ファイルと自動的に同期をとりません。JAR ファイルが変更された場合、手動でインポートし直さなければなりません。
- JAR パッケージでは［モデルの更新］コマンドと［ソースコードの更新］コマンドを使用できません。

JAR ファイルの再インポートは以下の手順で行います。

1. ［モデルビュー］で JAR ファイルアーティファクトを選択します。
2. コンテキストメニューから［jar ファイルの再インポート］を選択します。

既存のモデルからの **Java** の生成

Java ファイルは、UML モデルから生成できます。モデルが当初どのように作成されたかは問いません。Java ファイルは以下の方法で生成できます。

- パッケージの Java ソース コードへのエクスポート
- JAR ファイルの生成
- javadoc の生成

パッケージの **Java** ソース コードへのエクスポート

モデル内のパッケージを Java ソース コードにエクスポートするには、次の手順を行います。

1. ［表示］メニューの ［モデルビューの再構成］を選択し、［Standard View］を選択して、［Standard View］をアクティブにします。
2. モデル内のパッケージを選択します。

3. ［Java］メニューの［エクスポート］サブ メニューを選択して、［パッケージ ...］をクリックします。
4. ファイル システム内のエクスポート先フォルダを選択して、［OK］をクリックします。

パッケージの内容全体が .java ファイルに生成され、モデルからファイルへのマッピングと Java コードジェネレータ リファレンスで説明しているルールに従って、ファイル システムに書き込まれます。さらに、エクスポートされた最上位の Java パッケージごとに、Java ビルドアーティファクト がモデルに追加されます。このビルドアーティファクトは、パッケージをマニフェストしており、フォルダのパスを <<Java>> ステレオタイプ内のターゲットディレクトリとして格納しています。

Java ファイルにエクスポートされると、モデルとソースコードの間の同期は、［Java］メニューにある［モデルの更新］コマンドと［ソースコードの更新］コマンドによって維持されます。Java ビルドアーティファクトのコンテキストメニュー［Java］から、［Update］と［Generate］を使っても同じことができます。

ソース コードを編集するには、ファイルを表示する Java ファイル アーティファクトをダブルクリックするか、モデル要素、または、プレゼンテーション要素の［ソースコードの編集］コマンドを実行します。その結果、.java ファイルが組み込みテキスト エディタで開きます。

以下のパッケージでは、エクスポート コマンドを使用できます。

1. 最上位レベルの UML パッケージ
2. Java パッケージ

初めてパッケージから Java を生成するときは、1. が使用されます。生成された Java ファイルをファイル システム内の新しい場所に移動するときは、2. が使用されます。2. の方法の代わりに、Java ビルドアーティファクトのターゲットディレクトリの指定値を変更して、［Generate（Force)］コマンドを使う方法もあります。

JAR ファイルの生成

JAR ファイルをモデル内のパッケージから生成するには、次の手順を行います。

1. モデル内のパッケージを選択します。
2. ［Java］メニューの［生成］サブ メニューを選択し、［JAR ファイル ...］をクリックします。
3. ファイル システム内の宛先フォルダを選択して、［OK］をクリックします。

その結果、JAR ファイルがパッケージと同じ名前の宛先フォルダ内に生成されます。JAR ファイルは、コンパイルされたクラスとインターフェイスのみから生成されます。つまり、パッケージと関連付けられたフォルダ内の .class ファイルです。JAR ファイルを生成する前に、Java ビルドアーティファクト上で必ず［生成］コマンドに続けて［コンパイル］コマンドを実行してください。コンパイルでエラーが報告されないようにする必要があります。

javadoc の生成

javadoc をモデル内のパッケージから生成するには、次の手順を行います。

1. モデル内のパッケージを選択します。
2. ［Java］メニューの［生成］サブ メニューを選択して、［Javadoc...］をクリックします。
3. ファイル システム内の宛先フォルダを選択して、［OK］をクリックします。

その結果、選択されたパッケージに対応する javadoc ファイルが宛先フォルダに生成されます。javadoc ファイルは、_doc が付加されてパッケージと同じ名前のサブ フォルダに配置されます。インデックス ページが組み込み Web ブラウザに表示されます。

Javadoc は Java ソース ファイルのみから生成されます。つまり、パッケージと関連付けられたフォルダ内の .java ファイルです。javadoc を生成する前に、必ず Java ビルドアーティファクトで［生成］コマンドを実行してください。また Javadoc はエラーのある Java ファイルでは動作しません。［コンパイル］を実行して生成コードが正しいことを確認してから、javadoc を実行してください。

UML モデルで Javadoc コメントを作成する方法については、コメント を参照してください。

Java 構文

UML ツール セットでのデフォルトの構文は、U2 と呼ばれます。これは UML テキスト構文です。Java サポートでは、Java 構文を使用してツールを拡張します。つまり、U2 または Java でモデルを編集できるようになります。どちらの構文を使用するかはユーザーが決定します。新しいモデルは、U2 構文または Java 構文で作成できます。また、既存のモデルを U2 構文または Java 構文で見ることができます。この 2 つの言語は、よく似ています。セマンティック レベルでは若干の違いはありますが、Java は基本的に U2 のサブセットです。Java にない U2 の構築子は、Java に変換されません。詳細については、Java コードジェネレータ リファレンスを参照してください。

Java 構文は以下で使用できます。

- テキスト図
- テキスト シンボル（クラス図とパッケージ図内）

Java 構文は状態機械図では使用されないことに注意してください。その理由は、Java 言語にはない U2 構築子を状態機械図では必要とするからです。このような構築子の例として、シグナル送信やタイマを使った作業があります。

Java 構文はパッケージ レベルのみに設定できます。通常は、最上位レベルのパッケージ上に設定されます。パッケージで Java 構文が有効になると、パッケージに含まれているすべてのテキスト図とテキスト シンボルに Java 構文が使用されます。また、Java 構文はパッケージによって所有されているすべての要素（パッケージを含む）に階層的に適用されます。

Java 構文は U2 構文よりも優先順位が高くなります。また、階層内では、高いレベルの方が低いレベルよりも優先順位が高くなります。つまり、最上位レベルのパッケージ上で Java 構文が有効になると、そのパッケージに含まれているすべての要素で Java 構文が使用されます。

モデルとソース コードの同期

モデルとソースコードは、モデルからソース コードへの更新、またはソースコードからモデルへの更新を行うことで、同期させることができます。ただし、Tau はあくまでモデル ベース のツールであり、モデルとソース コードを別の物として取り扱うように設計されています。つまり、Tau はモデルを重視します。したがって、更新の方向がモデルからコードなのか、コードからモデルなのかによって、振る舞いがやや異なります。また、更新を自動で行うか手動で行うかによっても異なります。詳細については、以下のセクションで説明します。

自動同期と手動同期

モデルとソース コードの更新は、2 通りの方法で行うことができます。ソース コードまたはモデルの変更時に自動的に行うか、専用の更新コマンドを使用して手動で行うかです。

同期モードを制御するために、以下の 2 つの設定があります。

- Automatic roundtrip
- Automatic code generation

これらの詳細については、Java 設定で説明しています。デフォルトでは、Automatic roundtrip と Automatic code generation は無効になっています。

手動による **Java** ソース コードの更新

Java ソース コードは、手動同期モードで、以下の 2 つの方法によってモデルから更新できます。

- アクティブ プロジェクト内のすべての Java パッケージを対象にする
- 選択した Java 要素を対象にする

アクティブ プロジェクト内のすべての Java パッケージの Java ソース コードを生成するには、次の手順を行います。

1. ［Java］メニューの［ソースコードの更新］を選択するか、キーボード ショートカットの **Ctrl + Alt+ S** キーを使用します。

選択した要素の Java ソース コードを生成するには、次の手順を行います。

1. ［モデル ビュー］またはダイアグラムで、要素を選択します。
2. 右クリックして、［ソースコードの更新］を選択します。

Java ソース コードの生成に［ソース コードの更新］コマンドを使用すると、モデルの変更によって影響を受ける Java ファイルのみが生成されます。したがって、ほとんどの場合、一部の Java ソース コード ファイルのみ再生成されます。モデル全体の Java ソース コード ファイルを更新するには、［ソースコード更新を強制］コマンドを選択します。

Java ソース コードは、複数モデル要素から生成できます（詳細については、Java ランタイム ライブラリを参照）。最も一般的には、クラスとインターフェイスがベースになります。

モデルからソース コードを生成するときに、以下のモデルからファイルへのマッピングで説明されているマッピング ルールを使用して、モデル内の選択されたパートがファイル システムに書き込まれます。

モデルとファイル システムは自動的に同期がとられますが、ソース コードの生成時にモデルのほうがファイル システムよりも優先順位が高くなります。ファイル システム内に表現がないモデル要素がある場合は、対応するファイル システム要素が自動的に作成されます。たとえば、最後のソース コード作成時以後にモデルに新しいクラスが追加された場合は、対応する `.java` ファイルが自動的に作成されます。

また、モデル内に表現がないファイル システム要素がある場合は（たとえば、モデル内に対応するパッケージがないフォルダなど）、検出後にファイル システム要素を削除できます。要素を削除するには、常に再確認を要求されます。

モデル要素の名前変更と移動

モデルと、ファイル システム内のファイルおよびフォルダ間の関連付けは、モデルからファイルへのマッピングで説明しているように、モデルからのソース コードのインポートまたはパッケージのエクスポート時に自動的に維持されます。ただし、モデル内のエンティティの名前変更または移動を行うと、関連付けは自動的に維持されません。したがって、ファイルまたはフォルダを表すモデル要素を移動する場合、十分注意して相互の関連付けを維持するようにしてください。これは、パッケージ、Java ファイル アーティファクト、JAR ファイル アーティファクトに当てはまります。

このルールには、次の場合に例外が 1 つあります。モデル内で名前変更される最上位レベルのパッケージ。最上位レベル パッケージの名前変更後にソース コードを更新すると、ファイル システムのディレクトリ名が変更されて、モデル内のすべてのサブパッケージと Java ファイル アーティファクトが更新され、新しい場所が反映されます。

要素の名前変更または移動を正しく行うと同時にファイルの関連付けを維持する方法について、概要を以下で説明します。

1. モデル要素の名前を変更し、適用可能な場合は、要素を表現する Java ファイル アーティファクトの名前を変更します。
2. Path タグ付き値を編集して要素のパスを更新し、新しい名前と場所を反映します。（アーティファクトと最上位レベル パッケージの場合のみ）。
3. ファイル システム内の対応するフォルダとファイルの移動か名前変更またはその両方を実行します。

これらの手順の 1 つでも正しく実行しないと、同期時に問題が発生する場合があります。たとえば、ファイル システム内のファイルを移動しないと、新しいファイルが新しい場所に生成されます。これらの新規ファイルには、古いファイルに基づく非 Javadoc コメントは含まれません。これは、モデルに格納されていないためです（詳細についてはコメントを参照してください）。

生成された **Java** ファイルの場所の変更

Java パッケージの生成されたすべての Java ファイルを移動するには、パッケージを目的の場所に再エクスポートします。詳細については、パッケージの Java ソース コードへのエクスポートを参照してください。

既存の `.java` ファイルは、新しい場所にコピーされます。すべての Java ファイル アーティファクトのパスは、自動的に更新されます。

1 つの Java ファイルの場所を変更することは、Java コンパイラによって強制されるパッケージとファイル システムの対応を損なう可能性があるため、推奨できません。ただし、上記のモデル要素の名前変更と移動で説明している手順で行うことができます。

既存ファイルへの変更

既存ファイルは必要な場合にのみ更新されます。つまり、対応するモデル要素が変更された場合にのみファイルが更新されます。モデルの変更がファイルにマージされて、ファイルの内容とフォーマットが最大限に維持されます。

手動による **Java** ソース コードからのモデルの更新

モデルは、手動同期モードで、以下の 2 つの方法によってソース コードから更新できます。

- アクティブ プロジェクト内のすべての Java パッケージを対象にする
- 選択した Java 要素を対象にする

アクティブ プロジェクト内のすべての Java パッケージをソース コードから更新するには、次の手順を行います。

1. ［Java］メニューの［モデルの更新］を選択するか、キーボード ショートカットの Ctrl + Alt + M キーを使用します。

選択した要素をソース コードから更新するには、次の手順を行います。

1. ［モデル ビュー］またはダイアグラムで、要素を選択します。
2. 右クリックして、［モデルの更新］を選択します。

モデルをソース コードから更新すると、ファイル システムがスキャンされ、モデルがファイル システム要素から更新されます。これには、以下で説明しているモデルからファイルへのマッピング ルールが使用されます。

Java ソース コードをベースとするモデルの更新に［モデルの更新］コマンドを使用すると、最後のモデル更新以降に変更された Java ファイルのみが更新対象となります。したがって、ほとんどの場合、一部の Java ソース コード ファイルが使用され、一部のモデルのみが更新されます。すべての Java ソース コード ファイルをベースとしてモデル全体を更新するには、［モデル更新を強制］コマンドを選択します。

ファイル システムとモデルは自動的に同期がとられますが、ファイル システムからのソース コードの生成時にファイル システムのほうがモデルよりも優先順位が高くなります。モデル内に表現がないファイル システム要素がある場合は、対応するモデル要素が自動的に作成されます。たとえば、最後の更新以後に新しい `.java` ファイルがファイル システムに追加された場合は、対応するクラスがモデル内に自動的に作成されます。

また、ファイル システム内に表現がないモデル要素がある場合は（たとえば、ファイル システム内に対応するフォルダがないパッケージなど）、検出後にパッケージがモデルから削除されます。要素を削除するには、常に再確認を要求されます。

同期ターゲットディレクトリ

Java ビルドアーティファクトのターゲットディレクトリは、ビルドアーティファクト上で［更新］または［強制更新］コマンドを実行すると、「同期」が取られます。同期とは、フォルダとモデルの内容を同一に維持することです。

同期フォルダの新しいサブ フォルダがファイル システム内で作成されると、検出後に逆にモデルに挿入されます。同様に、同期フォルダにマッピングされたモデル内のパッケージが削除されると、ファイル システム内のフォルダを削除するように要求されます。

モデルからソース コードへのナビゲート

Java コードをモデル要素用に生成した後は、要素から対応する `.java` ファイルにナビゲートできます。

Java ファイル アーティファクトのソース コードにナビゲートするには、単にアーティファクトをダブルクリックします。

モデルまたはプレゼンテーション要素からソース コードにナビゲートするには、次の手順を行います。

1. Java コードを確認する要素を選択します。要素は、［モデル ビュー］またはダイアグラムで選択できます。
2. 右クリックして、［ソースコードの編集］を選択します。
   または
   ［Java］メニューで、［ソースコードの編集］を選択します。

何らかの理由で対応するファイルが見つからない場合は、エラー メッセージが表示されます。その場合は、ファイルを再生成して、再度ナビゲートを試みます。

Java のコンパイルと実行

コンパイル

生成されたソース コード（.java ファイル）を .class ファイルにコンパイルするには、次の手順を行います。

1. コンパイルする要素を選択します。
2. ［Java］メニューで、［コンパイル］を選択します。

その結果、選択したすべての要素のソース ファイルが .class ファイルにコンパイルされます。パッケージを選択した場合は、選択したパッケージ内のすべてのクラスがコンパイルされます。

注記

コンパイルされるのは、既存のソース ファイルだけです。したがって、コンパイルを実行する前に、必ず［ソースコードの更新］コマンドを実行してください。

構文エラーなどによりコンパイルに失敗すると、コンパイル エラーが［スクリプト］出力タブに書き出されます。エラーにナビゲートするには、ファイル名と行番号のある行をダブルクリックします。ファイルが開き、エラーが発生している行にカーソルが配置されます。

エラーはモデルまたはソース コード内で解決できますが、エラーの修正時に必ずモデルとソース コードを同期することが重要です。

コンパイル時に使用される classpath は、classpath 変数の記述に従って計算されます。コンパイルは、常に source オプションを *1.4* に設定して（特に asserts を有効にするために）実行します。

クラスの実行

クラスを Java アプリケーションとして実行するには、次の手順を行います。

1. 実行するクラスを選択します。
2. ［Java］メニューで、［実行］を選択します。

クラスが Java アプリケーションとして実行され、実行結果が［スクリプト］タブに表示されます。クラスを実行するには、クラスが正しくコンパイルされ、main メソッドを含んでいる必要があります。.java ファイルが存在し、.class ファイルが存在しない場合は、.java ファイルが自動的にコンパイルされます。

実行時に使用される classpath は、classpath 変数の記述に従って計算されます。実行時に asserts を有効にするために、常に **esa** オプションを設定して実行します。

アプレットとしてのクラスの実行

このコマンドはクラスの実行の場合と同様ですが、クラスはアプレットにラップされ、その後に Web ブラウザで開かれます。このコマンドは、名前のないパッケージ内のクラスのみを対象にして動作します。これは、単純なクラスのクイック チェック用に設計されています。

1. アプレットとして実行するクラスを選択します。
2. ［Java］メニューで、［アプレットの実行］を選択します。

その結果、*<class name>.html* の形式でファイルが作成されて、クラスがアプレットとしてインスタンス化されます。その後 html ファイルが Web ブラウザに表示されます。

classpath 変数

Java コードのコンパイルおよび実行時に使用される classpath 変数は自動的に計算されます。通常、変更は必要ありません。

プロジェクト内のすべての Java パッケージと jar ファイルのディレクトリが classpath に追加されます。また、すべての同期フォルダ（詳細については、同期ターゲット ディレクトリを参照）が追加されます。

たとえば、プロジェクトに含まれていない外部ライブラリをインクルードするなど、上記のスキームで不十分な場合は、classpath を無効にできます。classpath を手動で無効にするには、次の手順を行います。

1. ［モデル ビュー］で **Java Model** ノード（［Standard View］の **Model** ノード）を選択します。
2. 右クリックして、［プロパティ ...］を選択します。
3. ［フィルタ］ドロップダウン メニューで、［Java Settings］を選択します。
4. 必要に応じて、［Classpath］リストにエントリを追加します。

注記

手動で classpath を無効にすると、classpath の自動計算が無効になり、手動でリストに追加したエントリのみが使用されます。新しい最上位レベル パッケージを作成した場合、classpath を必ず更新することが重要です。

Execution Tracing

The execution of Java programs can be traced in Tau using sequence diagrams. This feature can be used for visualizing communication between Java objects, to detect incorrect program flows, and in general to obtain an understanding of the run-time behavior of a Java program. Tracing is often combined with debugging in an IDE, such as Eclipse. By setting breakpoints around interesting sections of code a trace can be obtained for visualizing what the program does in those parts of the code.

In order to produce an execution trace the Java program must be **instrumented**. This is done by loading a trace agent into the Java virtual machine. The agent collects information about what happens in the program by responding to events sent by the virtual machine. This information is then sent to Tau, where it is presented in UML diagrams.

The Tau module which produces the diagram from the trace events is called a **tracer**. It is possible to implement custom tracers in order to present the trace information in some other way, for example in another kind of diagram, or to filter the information in a different way.

Tau ships with a standard tracer, the Instance Tracer, for producing sequence diagrams where each Java class instance (object) is represented by its own lifeline, and where the interaction between instances in the form of method calls are visualized.

ヒント

Since Java trace instrumentation is done without modifying the Java source code it is possible to trace the execution of any Java program, not only programs generated from Tau.

Start a New Trace Session

Perform the following steps in Tau to start a new trace session:

1. Select the command **Java / New Trace Session...**
2. In the dialog select which tracer to use for the trace session.
3. The selected tracer will by default be enabled initially, meaning that as soon as the dialog is closed it is ready to receive trace events. If you want to wait a little with enabling the tracer (for example to give time for running the Java program up to a breakpoint of interest first) then uncheck the **Enabled** checkbox.
4. Press **OK** to close the dialog.

When the dialog is closed a new top-level “trace” package will be created in the model. It is by default called “JavaTrace”. You may change this name to better describe the trace session.

The settings made in the dialog are stored on the trace package. You can change these settings at any time by opening the Properties Editor on the trace package with the filter `javaTrace` selected. The most common reason for doing this is to enable or disable the trace session in order to filter the trace to only cover interesting parts of the program execution.

Instrumenting the Java Program

To make the Java program instrumented you need to add an option to the Java virtual machine:

-agentlib:JavaInstrument

This will tell the Java virtual machine to load the instrumentation agent `JavaInstrument` before running the Java program. Note that you also must have your environment variable `PATH` (`LD_LIBRARY_PATH` on Unix) set to include the Tau installation `bin` directory so that the Java virtual machine can find the `JavaInstrument` agent. Note that for Linux RedHat 5 `LD_LIBRARY_PATH` must also contain the Tau installation `bin/.rh5` directory, and this directory must be listed before the `bin` directory.

The JavaInstrument agent takes several options which may be specified after an equal sign. Options are separated by commas, and each option consists of a name-value pair separated by a colon. For example:

-agentlib:JavaInstrument=host:localhost,port:57000

Available options are described below.

Option ‘host’

This option specifies the name of the host computer where the Tau instance to trace to is running. The default value is “localhost” meaning that Tau should be running on the same machine as the Java program. You may use this option to send the trace to a Tau running on a different machine in the network.

Option ‘port’

This option specifies the web server port of the Tau instance to trace to. The default port number is 57000, but if you have (or have had) multiple instances of Tau running on the machine, you may need to use a different port number. To find out which port number a particular instance of Tau is using follow the instructions inTau Web サーバーの使用法 .

Option ‘start_method’

By default instrumentation events start to be sent when the Java application reaches the main method. This option can be used for setting a different method to start the instrumentation in. Currently only the method name can be specified here; it is not possible to qualify the name or to specify a particular overload of a method.

The ‘start_method’ option is useful for applications which consists of a complex start-up phase, involving lots of calls to standard libraries. One example is applications with a user-interface. Typically it is uninteresting (and takes too long time) to trace what happens during this initialization phase. Tracing can then begin at the first method that gets called after the initialization phase is completed.

例 **403: Using the ‘start_method’ option to defer instrumentation start**

Consider the following Java program:

```
class MyClass {
  public static void main( String[] args) {
    MyDialog dlg = new MyDialog();
    start();
  }
}
```

Assuming that the `MyDialog` class is a Swing dialog, its instantiation will imply lots of calls to the Swing library. To avoid tracing these calls we use the ‘start_method’ option:

-agentlib:JavaInstrument=start_method:start

Tracing will now begin when the ‘start’ method is called, that is after the GUI has been initialized.

Option ‘skip’

This option specifies definitions that should not be instrumented. The value is a list of definitions using the same syntax as the Java import statement.

The default value of this option is “java.*;sun.*”, meaning that definitions in the ‘java’ and ‘sun’ packages will be excluded from tracing.

The ‘skip’ option allows you to reduce the generated trace to only involve those parts of the program you are interested in. For example, you may want to skip instrumenting all 3rd party libraries you are using.

Note that a call from method A to method B is only skipped from instrumentation if both A and B are part of a definition listed in the ‘skip’ list. If only one of these methods is in the ‘skip’ list the call will be traced. This can for example be used to see which library calls your code performs, or which 3rd party components that call your code.

例 **404: Using the ‘skip’ option to filter a trace**

The following JavaInstrument option:

-agentlib:JavaInstrument=skip:javax.*;MyPkg.MyClass

will exclude all definitions in the package `javax` and also the class `MyPkg.MyClass` from tracing.

Setting instrumentation options when using Eclipse

Here are the steps to perform in the Eclipse IDE in order to make the instrumentation settings described above. The instructions are for Eclipse 3.3; other versions may have a slightly different user interface.

1. Right-click the Eclipse Java project you want to instrument. Select **Debug As / Open Debug Dialog...**
2. Make sure the Run configuration for the project is selected, or create a new Run configuration for the project.
3. In the **Arguments** tab enter the instrumentation option in the **VM arguments** field. For example:
   `-agentlib:JavaInstrument=host:localhost,port:57000`
4. In the **Environment** tab create a new variable `PATH` (`LD_LIBRARY_PATH` on Unix) and set its value to the Tau installation `bin` directory. For example, `C:¥Program Files¥IBM¥Rational¥TAU¥4.3¥\bin`
5. Click **Debug** to close the dialog and start the debug session.

Of course you can do the same for a Run configuration in case you do not want to debug the Java program.

Instance Tracer

The Instance Tracer is a predefined tracer which visualizes application trace events as a UML sequence diagram. The diagram will contain one lifeline for each dynamic Java class instance that performs some actions during the trace session. The name of the lifeline is the address of the Java object.

For classes that contain static methods an additional lifeline will be created, called `static`. It represents all static parts of the class.

To improve trace performance certain low-level Java library objects are not traced. This includes objects for classes defined in the `java` and `sun` packages.

Called methods, including constructors, initializers etc. are visualized in the diagram as method calls. The creation of new Java objects is also visualized.

Example

Consider the Java program below:

```
package JavaTraceTest;
class C {
    public static void main( String[] args) {
        D d = new D();
        d.foo();
    }
}
class D {
    void foo() {
        return;
    }
}
```

The sequence diagram trace produced by the Instance Tracer for this program looks like shown in 図 248 on page 1160. The diagram shows the following events in the Java program:

1. The static `main` method in class `C` is called. The object making this call is a low-level library object which is not traced to the diagram.
2. A new instance of class `D` is created. The instance is located at address `0x1507fb2` in the Java virtual machine. The creation also implies the call of `D`'s constructor. The implementation of this constructor is auto-generated by the Java compiler.
3. Control is returned to `C.main` from `D`'s constructor.
4. A call to `D.foo` is made.
5. Control is returned to `C.main` from `D.foo`.
6. The `main` method is returned from.

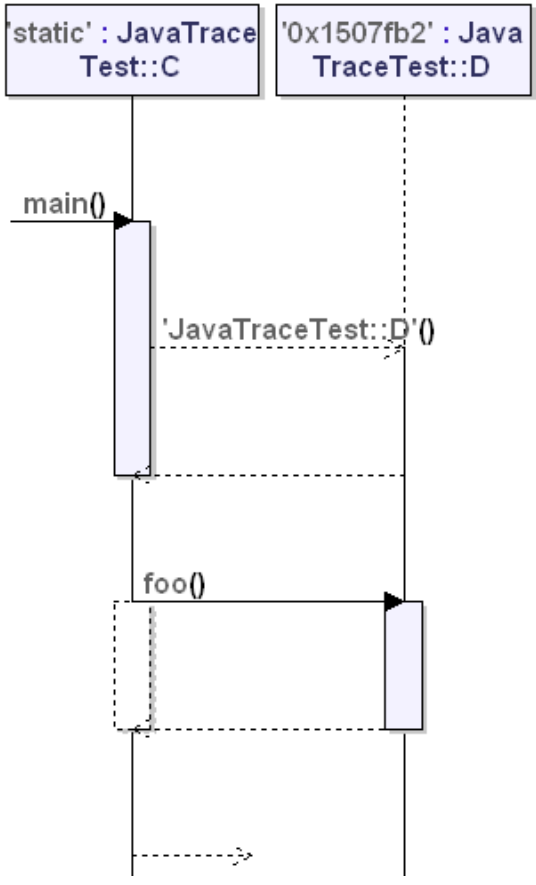


図 248: Sequence diagram trace produced by the Instance Tracer

Adding Custom Tracers

It is possible to add custom tracers in order to present the trace events received from the Java virtual machine in a custom way. This is done by defining a class that inherits from a `JavaTracer` class, and defining agent operations in the class that overload appropriate virtual operations from `JavaTracer`. The agents get called when trace events are received from the Java virtual machine. For more information about `JavaTracer` and its available operations see the package `TTDJavaModelCodeSync::Tracing` under the `Libraries` section of the model. There you can also find the definition of the `InstanceTracer`.

モデルからファイルへのマッピング

このセクションでは、モデル要素とファイル システム要素間のマッピング ルールについて説明します。以下の表に概要を示します。詳細については、以下のセクションを参照してください。

ファイル システム要素	モデル要素
.java ファイル	表す要素（クラスとインスタンス）とマニフェスト関係がある Java ファイル アーティファクト。
.jar ファイル	JAR の要素に対して依存関係がある JAR ファイル アーティファクト。
フォルダ	Java パッケージ

Java ファイル アーティファクト

Java ファイル アーティファクトを使用して、それぞれの .java ファイル、またはより具体的にはそれぞれの Java コンパイル ユニットを表します。Java ファイル アーティファクトは <<javaFile>> ステレオタイプが適用されたアーティファクトです。アーティファクトは、表す .java ファイルと同じ名前になります。

ファイルのパスは、javaFile ステレオタイプの File name プロパティに格納されます。このパスには、プロジェクトを基準にした相対パスまたは絶対パスを使用できます。パスは、インポートまたはエクスポート時に自動的に設定されます（可能な場合は、相対パスを使用）。マッピングは、必要に応じて手動で変更できます。

アーティファクト要素からモデル要素へのマニフェスト依存関係を使用して、ソース コードの更新時に .java ファイルに書き込む必要がある要素を指定します。表現された要素は、最上位レベル宣言としてファイルに書き込まれます。マニフェスト関係を作成、移動、または削除して、ファイルの内容を制御できます。

Java でのインポート宣言はファイルごと、またはコンパイル ユニットごとに定義されます。インポート宣言はインポート依存関係としてアーティファクトに格納されます。インポート依存関係を作成、移動、または削除して、ファイルに書き込むインポート宣言を制御できます。

アーティファクトを所有する UML パッケージ内の他の要素がこれらのインポート依存関係を認識できるように、アーティファクトはそれを所有するパッケージごとにインポートします。その結果、アーティファクトによってインポートされたすべての定義がパッケージ全体に適用されます。

アーティファクトは、ソース コードまたはモデルの更新時に、自動的に作成または削除されます。

注記

これらのアーティファクトを手動で作成、移動、削除することは推奨できません。

Java パッケージ

モデル内の Java パッケージは、ファイル システム内のフォルダにマッピングされます。ネストされたパッケージは、ファイル システム内のネストされたフォルダにマッピングされます。

Java パッケージは、<<javaPackage>> ステレオタイプが適用されたパッケージです。Java パッケージとファイル システム内のそのフォルダ間のマッピングは、ステレオタイプの Path タグ付き値によって制御されます。Path の値は、このパッケージのマッピング先であるファイル システム内のフォルダです。Path タグ付き値には、絶対パスまたはプロジェクトを基準とした相対パスを格納できます。

デフォルトにより Path タグ付き値は、最上位レベル パッケージにのみ設定されます。ネストされたパッケージのパスは、内包されるパッケージを基準にして自動的に計算されます。

既存の Java ファイルをインポートすると、マッピングが自動的に設定されます。通常、変更は必要ありません。ただし、マッピングはいつでも変更できます。

名前のないパッケージ

名前のないパッケージは、必要に応じて自動的に作成されます。通常は、インポート時に選択したフォルダの直下にある .java ファイルをインポートするときに作成されます。

名前のないパッケージは「default」と呼ばれ、ステレオタイプ **unnamedPackage** が適用されています。

名前のないパッケージ内の要素を同期しても、パッケージ宣言は生成されません。

名前のないパッケージは、プロジェクトごとに 1 つしか作成できません。

注記

名前のないパッケージは、Java 環境やコンパイラごとに異なる方法で扱われるため、コンパイルおよび実行時に問題が発生する可能性があります。したがって、できる限り**名前のないパッケージの使用は避けます**。

JAR ファイル アーティファクト

JAR ファイルは、.jar ファイルと同じ名前の JAR ファイル アーティファクトで表されます。アーティファクトには、JAR ファイルの要素に対する依存関係があります。

JAR ファイル アーティファクトは、JAR ファイルのインポート時に自動的に作成されます。JAR ファイル アーティファクトは設計目的で手動で作成できますが、JAR ファイルの生成方法の記述には使用できません。

JAR ファイル内にあるパッケージは、JAR ファイル内にあることを示す <<jarPackage>> ステレオタイプが適用された Java パッケージです。

JAR パッケージは静的であると見なされ、他のパッケージより制約的です。詳細については、JAR ファイルのインポートを参照してください。JAR ファイルをモデルから生成する方法の詳細については、JAR ファイルの生成を参照してください。

Java ランタイム ライブラリ

Java サポートには、最も一般的な Java ランタイム ライブラリが用意されています。内容は、［モデル ビュー］の **Java Libraries** ノード（［Standard View］使用時は **Libraries**）にあります。ランタイム ライブラリには、Java 配信の rt.jar ファイルの内容全体が含まれます。

ランタイム ライブラリのロード

デフォルトでは、Java プロジェクトのロード時にランタイム ライブラリが自動的にロードされます。すべての Java プログラムがこれらのランタイムライブラリを使うわけではないこと、サイズの大きいライブラリのロードに時間がかかることなどから、選択的に自動ロードをオフにすることも考慮すべきです。

起動時に rt.jar ライブラリをロードしないようにするには、［Load rt.jar on startup］チェック ボックスの選択を解除して、プロジェクトの Java 設定を変更します。

ランタイム ライブラリの使用

ランタイム ライブラリのいずれかの要素を、完全バインディングや名前の完成などで通常の UML 要素として使用できます。

それらの要素をダイアグラムで使用するには、要素を［モデル ビュー］からダイアグラムにドラッグします。シンボルまたはテキスト構文でタイプとして使用するには、要素名を使用します。要素は、正しくバインドするためにスコープ内で表示可能にする必要があります。詳細については、以下のパッケージと要素の可視性を参照してください。

パッケージと要素の可視性

Java 実装の要求に応じて、パッケージ java.lang の内容にいつでもアクセスできます。他のパッケージは、明示的にインポートするか完全修飾名によって参照してバインドする必要があります。U2 構文での例を以下に示します（Java 構文ではパッケージ全体を編集できません）。

```
package P1 <<import>> dependency to java::io {
  class A {
    Integer i;
    FileWriter fw;
    java::util::Vector v;
  }
}
```

上記の例では、属性 i のタイプ Integer は `java::lang::Integer` に自動的にバインドされます。`fw` 属性のタイプ FileWrite は、`java::io::FileWrite` にバインドされます。これは、パッケージ P1 によってパッケージ `java::io` がインポートされるためです。属性 v の Vector（ベクトル）には、完全修飾名を使用する必要があります。これは、このスコープではパッケージが表示されないためです。

注記

Java ファイルアーティファクトを使用して `.java` ファイルを表すため、アーティファクトによるインポートのみが、表されるファイルに書き込まれます。UML パッケージまたはクラスでのインポートは、パッケージから生成されたすべての `.java` ファイルに自動的に移行されません。

Tau オブジェクトランタイムライブラリ

状態機械やシグナルのように Java 言語に表現のない UML 構築子の翻訳をサポートするために、Tau Object Runtime (TOR) が使用されます。このライブラリは、主として Java コードジェネレータが生成したソースコードによって、Java ソースコードレベルで使用されます。しかし、一部のライブラリは、UML モデルレベルでアクセスすると有用です。したがって、TOR はライブラリとしてロードされます。

TOR ライブラリの詳細については、Java ランタイムフレームワークを参照してください。

その他のライブラリ

jsse.jar での暗号化サポートなど、他の Java ライブラリが必要な場合は、この JAR ファイルを手動でインポートできます。JAR ファイルのインポートを参照してください。

Java モデリングユーティリティ

Tau は、UML を使った Java プログラムのモデリングを簡単に行うためのユーティリティーを提供します。ユーティリティの一部は、Tau で通常の操作を行う際に自動的に適用されます。そうではないユーティリティは Java モデリング専用のものです。

アクティブクラス生成

Java クラスを ' アクティブ ' にするときに、Tau はそのクラスに自動的に 1 つの汎化を追加します。これは、そのクラスが TOR クラス DispatchableClass を継承できるようにするためです。‘init’ や ‘start’ などの重要なメソッドへの、UML レベルでのアクセスを可能にするためです。

Java クラスを ' パッシブ ' にした場合は、DispatchableClass への汎化は自動的に取り除かれます。

Main メソッドの追加

Java クラスに main メソッドを自動で追加できます。以下の手順で行います。

1. ［モデルビュー］でクラスを選択します。
2. コンテキストメニューで［ユーティリティ］>［Generate Main Method］コマンドを選択します。

main メソッドは、そのクラスのビルドを行う Java ビルドアーティファクトの情報から生成されます。ビルドアーティファクトに直接または間接的にマニフェストされた、ネストされていないアクティブクラスごとに、1 つのインスタンスが作成されます。そのインスタンスは Dispatcher に追加され、'init' や 'start' の呼び出しに備えます。最終的には 'run' が呼ばれます。

生成された mai メソッドは最上位のアクティブクラスのインスタンスを、単一スレッドのアプリケーションとして実行します。このデフォルトの実装は、たとえば、マルチスレッドプログラムを作成する ThreadedDispatcher を使用する、などの修正ができます。

Java 設定

それぞれの Java プロジェクト（またはモデル）には、Java 固有の設定があります。一部の設定は、特定の Java ビルドアーティファクトに固有であり、値も Java アーティファクトごとに異なります。こういった設定については Java ビルドアーティファクト設定で説明しています。

このセクションでは、モデル全体に共通である、他の種類の設定を説明します。これらのグローバルな設定は、プロジェクトファイルに格納されます。この設定の表示と編集には、以下の手順を実行します。

1. ［モデル ビュー］で **Java Model** ノード（[Standard View］の **Model** ノード）を選択します。
2. 右クリックして、［プロパティ ...］を選択します。
3. ［フィルタ］ドロップダウン メニューで、［Java Settings］を選択します。

以下の設定があります。

- Language version
  - 使用する Java バージョンを制御します。通常このオプションは［New］ウィザードで新しいプロジェクトを作成するときに指定した値をそのまま変更せずに使います。ただし、既存のプロジェクトで使用する Java のバージョンを上げたい場合などに、このオプションを使います。設定の変更後、Java プロジェクトを再ロードして正しいバージョンの Java ライブラリが使われていることを確認してください。
- Load rt.jar on startup
  - 起動時に Java ランタイム ライブラリ（rt.jar）の内容をロードするかどうかを指定します。of rt.jar のロードを無効にするとロード時間が大幅に短縮されますが、.jar 内の要素への参照はバインドされません。

既知の制限事項

以下のセクションでは、Java サポートに関連する既知のすべての制限事項について説明します。

アクティブコードジェネレータ

効率的にコードを生成するためには、各プロジェクト内のアクティブなコード ジェネレータの数を制限することを推奨します。たとえば、プロジェクト内で C++ アプリケーション ジェネレータ と Model Verifier コードジェネレータの両方がアクティブな場合、コード生成を終了するまでに長い時間が必要になり、多くのメモリ資源を消費します。

内部クラスでのモデル バインディング

内部クラスを定義する文は、バインドが正しく実行されない場合があります。その例を以下に示します。以下の例では、op1 の呼び出しに渡されたパラメータはバインドされません。

```
abstract class classA {
  abstract void run();
}
class classB {
  public void op1(classA p1) {}
  public void op2() {
    op1( new classA() {
        void run() {
        }
      }
    );
  }
}
```

UML パッケージに対応する **Java** 構文がない

UML パッケージ全体を Java テキスト構文で編集することはできません。UML と Java のパッケージ概念は表面的によく似ていますが、セマンティック上は異なります。UML と Java のパッケージ間に 1 対 1 のマッピングは確立されません。UML パッケージを編集できるのは、U2 テキスト構文のみです。

Java と **U2** の構文間の切り替えができない

Java と U2 の構文間で相互の切り替えができない場合があります。概念がサポートされていないかマッピングされていないために、いずれかの構文にエラーがあると、そのような状態になります。構文エラーがある場合は、構文を他の言語に変更できません。

その場合は、エラーを修正するか、テキスト シンボルまたはダイアグラムを削除して、最初から正しい構文で新規に作成する必要があります。

Java EE 5 アドインの使用法

The purpose of the Java EE 5 addin is to simplify the design of Java Enterprise Edition 5 applications using Tau. The JEE5 addin provides the following features:

- A Java EE 5 UML profile that gives access to Java EE 5 concepts in the UML model.
- Utilities to facilitate Java EE 5 model design directly from context menus in UML diagrams.

This document is structured into two parts:

- A simple introduction that step-by-step will take you through the design of a simple Java EE 5 enterprise component in Tau.
- A reference manual section that describes the commands and utilities available in the addin.

「Hello」サンプル

As a simple example let's consider creating a small Java EE model that will provide a greetings service to the world. We will start from scratch and end up with a complete Java EE component ready to be deployed on your application server of choice.

Java EE 5 アドインの入手

To work with Java EE 5 projects in Tau the JavaEE5 addin is needed. This can be downloaded from the Telelogic support pages at support.telelogic.com. Go to the SDK pages and download the Java EE 5 addin. To install the addin simply unzip it in the addins subdirectory in the Telelogic Tau installation directory.

Java EE 5 プロジェクトの作成

The next step is to create a new java project. This can be done using the New wizard in Tau:

1. - Select the File->New command and choose "UML for Java Code Generation" as the project type and 5 as the java version.
2. - Give it a suitable name, in the rest of this document we will refer to it as "hello".
3. - Select the "Location" to a suitable directory.

The name of the directory must not contain spaces to make sure all utilities described below will work. In the rest of this introduction we will assume the location is C:\TauProjects\hello.

4. - Accept the default options in the rest of the New wizard

   You have now created a standard Java 5 EE project.

JavaEE5 アドインの有効化

To use the Enterprise Edition concepts it is necessary to also switch on the JavaEE5 addin:

1. - Choose the command Tools->Customize.
2. - Open the Add-ins tab.

3. - Activate the JEE5 addin and click on the Close button.

   You are now ready to create Java EE 5 models.

EJB コンポーネントの作成 – セッション Bean

There are several component kinds in the Java EE 5 framework; stateless and stateful session beans, message driven beans and persistent entities. In this example we will start by creating a stateless session bean. To do this we essentially only need to do three things:

- define the business interface of the bean

- implement the business interface in a bean class

- package the bean in a JAR file

We will start by designing the component interface with its business methods and mark this as a remotely accessible bean interface by stereotyping it with the <<Remote>> stereotype:

- Start by creating a package to hold the EJB components called “ejb” inside the “hello” package.

- Double-click on the “ejb” package in the Model View and create a class diagram.

- Create a new interface in the diagram and call it “Hi”.

- Add an operation “greetings():String” to the interface

- Select the interface symbol, and choose “Apply <<Remote>>” from the context menu.

We now have completed the component interface definition and the next step is to create a stateless session bean that implements the interface:

- To get a kick-start in the implementation use the command “Create implementation class” available in the context menu for the interface. This will create a class that contains the correct operation and that realizes the interface.

- To mark this as a stateless session bean, choose the command “Apply <<Stateless>>” from the con text menu.

The next step is to provide the implementation of the greetings() operation. This can be done either in the UML model or by coding directly in the java file. In this example we’ll choose to edit the code directly. This is done as follows:

- To get java source code for the UML model select the “hello” package and choose the “Java->Export package” command.

- To edit the source code for the bean class, select the command “Edit source code” in the context menu of the “HiBean” class. This will open the java source code file in the built-in text editor.

- Fill in the implementation of the greetings method: public String greetings(){ return "Hello!"; }

- Save the file to update the UML model. Use for example the Ctrl-S keyboard shortcut in the source code editor.

The design and implementation of the EJB component is now finished and what remains is to build and package it. In general this is easiest to do using a build tool like Ant or using a Java IDE like Eclipse, but in this simple example we will use the standard java tools javac and jar, that are

available in the Java EE 5 SDK. This SDK can be downloaded e.g. from http://java.sun.com/javaee/downloads/index.jsp. Tau contains some simple commands in the Java menu that wraps javac and jar and makes them available from inside the tool. To use them there is however a need to set up the classpath for the java compiler to include both the all jar files used in the model and the directories where the source code is generated. The classpath is stored as a property of the root of the UML model:

- Select the “Java Model” node in the Model view and choose the “Properties…” command from the context menu.

- In properties dialog choose “Java Settings” as the “Filter” to see the java related settings.

- Add a value to the Classpath property that includes the javax.jar file. This should be something like “;/Sun/SDK/lib/javaee.jar” Note that the citation markers should not be part of the value. Also note the initial semicolon.

If you use an external Java IDE to compile the program you will instead need to add the javaee.jar file to the compilation settings of the IDE. The exact steps to do this depend on the details of the IDE, but if you are using Eclipse this is done in the Libraries tab of the properties for the java project.

We’re now ready to start compiling the two java files using the javac command:

- Select the “hello” package in the Model View.

- Choose the command Java->Compile. This will compile all files contained in this package.

The only remaining task is now to package the component into a jar file that can be deployed on an application server:

- Select the hello package and give the command “Java->Generate->Jar file…”

- When prompted: Select a suitable target directory.

- You can follow the progress of the jar command in the Script window. In our case this should show that the Hi.class and HiBean.class files are added to the jar file.

You now have a packaged component hello.jar that can be deployed on an application server. How to do this is specific for each application server but usually the application servers contain both a web based interface and a command line interface that can be used to deploy components. If you happen to be using the free Glassfish server the command line command to deploy our component would be “asadm deploy hello.jar”.

永続エンティティの作成

The next step is to create a few persistent entities, i.e. classes that in the end will be stored on a database on the application server.

We will create two classes that will give some personal flavor to the hello service we’re implementing by storing favorite greeting phrases for each person. The UML design consists of two classes, Person and Phrase as in the following figure:

After creating this small model (create them in the same ejb package as where you created the Hi interface and the HiBean class) we will mark them as persistent entities. This is done as follows for each class:

- Select the class

- Choose the command “Apply <<Entity>>” in the context menu. This will in addition to applying the correct stereotype also create a constructor that is mandatory for Java EE entity classes.

- Choose the command “Create Get/Set operations”. This will generate Java Bean compatible getters and setters that furthermore will be stereotyped according to the multiplicities and navigability of the attributes in the class.

The next step is to give each class a primary key. This is an attribute of the class that can be used to uniquely identify instances of the class.

- Select the “name” attribute in the Person class and choose the command “Select as primary key”. This will apply the stereotype <<Id> to the corresponding Get operation.

- Do the same for the “text” attribute of the Phrase class.

We now have two persistent entities in the model and the next step is to use these entities from the “hello” session bean. The idea is to add business methods to the bean to accomplish the following:

- It should be possible to add a new person with a favorite greeting phrase

- A new greetings method should be available that uses the personalized phrase.

To accomplish this do as follows:

- Add two operations to the Hi interface: greetings(name:String):String and addPerson(name:String,phrase:String)

- Select the interface and choose the command “Push operations to class” in the context menu.

- Open the source code editor for the HiBean class (for example by selecting the class symbol and choosing the command “Edit Source Code” in the context menu).

- Modify the source code according to the following example. The italic text is the text that you have to insert. It contains three parts, a reference to an EntityManager and the implementations of the two new operations:

```
package hello.ejb;
public class HiBean implements Hi {
@javax.persistence.PersistenceContext
javax.persistence.EntityManager em;
public String greetings(){ return "Hello!"; }
public String greetings( String name) {
Person p = em.find(Person.class,name);
if (p.getFavorite() != null) {
return p.getFavorite().getText();
} else {
return "Hello!";
}
}
public void addPerson( String name, String phrase) {
Person p = new Person();
p.setName(name);
Phrase ph = new Phrase();
ph.setText(phrase);
p.setFavorite(ph);
em.persist(p);
em.persist(ph);
}
}
```

Now we have completed the ejb part of the application and what remains is to build and package it. Do this in the same way as was described in section “Creating an EJB Component – Session Bean” above.

When deploying the new component to the application server you need to make sure that the application server contains a data base server and that the mapping from the Person and Phrase entities to tables in a data base is defined. By default there is assumed to be one table per entity class, with the same name as the class, and one column per attribute, also with the same name as the attributes. Most application servers contain a utility to create the tables automatically based on the entity classes. In many cases this is done by giving a deployment descriptor in the jar file that you package the EJB in. The deployment descriptor relevant for entities is a file called persistence.xml that is located in a directory called META-INF in the root of the jar file. Please consult your application server manuals for details.

Below is an example of what the persistence.jar file could look like if you are using the Glassfish application server. Notice the non-standard property “toplink.ddl-generation” that tells the application server to delete and recreate all necessary tables in the database associated with the “__default” jdbc data source.

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence
xmlns="http://java.sun.com/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/persis
tence
http://java.sun.com/xml/ns/persistence/persistence_1_
0.xsd" version="1.0">
<persistence-unit name="hello" transaction-
type="JTA">
<jta-data-source>jdbc/__default</jta-data-source>
<jar-file>hello.jar</jar-file>
<properties>
<property name="toplink.ddl-generation"
value="drop-and-create-tables"/>
</properties>
</persistence-unit>
</persistence>
```

参照マニュアル

インターフェイスで利用可能なコマンド

The following commands are available in the context menu for interfaces, both when selecting an interface symbol in a diagram and when selecting an interface in the Model View.

Apply <<Remote>>	The interface will be marked as a remotely accessible bean interface by adding the stereotype javax.ejb.Remote.
Apply <<Local>>	The interface will be marked as a remotely accessible bean interface by adding the stereotype javax.ejb.Local.
Create Implementation Class	A class will be generated that implements the interface. The class will realize the interface and will contain all the operations that were defined in the interface.
Push Operations to Class	The operations that are defined for the interface will be copied to any class that realizes the interface.

クラスで利用可能なコマンド

The following commands are available in the context menu for classes, both when selecting a class symbol in a diagram and when selecting a class in the Model View.

Apply <<Stateless>>	The class will be marked as a stateless session bean by adding the stereotype javax.ejb.Stateless.
Apply <<Stateful>>	The class will be marked as a stateful session bean by adding the stereotype javax.ejb.Stateful.
Apply <<Entity>>	The class will be marked as a persistent entity by adding the stereotype javax.persistence.Entity. The class will also automatically get a constructor generated to comply with the rules for persistent entities in Java EE 5.
Apply <<MessageDriven>>	The class will be marked as a message-driven bean by adding the stereotype javax.ejb.MessageDriven.
Create Interface Class	An interface will be generated that is realized by the class. The interface will contain all public operations defined in the class.
Create Get/Set Operations	Get/Set operations will be generated for all attributes in the class in a style compatible with the requirements for Java Beans. If both the class that owns the attributes, and the type of the attributes, are persistent entities (i.e. marked by the <<Entity>> stereotype) there will also be generated annotations on the Get operations (one of OneToMany OneToOne ManyToOne ManyToMany depending on the multiplicity and navigability).
Copy Operations from Interface	Operations in realized interfaces will be inserted in the class (if not already existing).

Commands Available in the Help menu

The following commands are available in the “Help” menu .

Help on Java EE 5 Profile	An introduction and reference to the Java EE 5 addin are displayed.

属性で利用可能なコマンド

The following commands are available in the context menu for attributes, both when selecting an attribute in a class symbol in a diagram and when selecting an attribute in the Model View.

Select as primary key	The attribute is marked as a primary key of the class. This is accomplished by stereotyping the corresponding get operation by javax::persistence::id.
Create Get/Set Operations	Get/Set operations will be generated for the attribute in the class in a style compatible with the requirements for Java Beans.

パッケージで利用可能なコマンド

The following command is available in the Edit menu when a package is selected.

Find EE5 Generated Elements	This command will create a tab in the Output window listing all elements that have been generated by the Java EE 5 addin utilities in the selected package.

永続エンティティ用ユーティリティ

The Java EE support for persistent entities (in addition to what has been described above) consists mainly in generating the Java EE stereotypes for relationships based on the multiplicities given in the UML model. Essentially this gives us a possibility to design the relationships using UML class diagrams and automatically generate the correct EJB annotations in the Java source code.

The generation of the multiplicity annotations is handled by the “Create Get/Set operations” command described above.

Java EE アプリケーションに適合したステレオタイプ

There are many stereotypes loaded by the Java EE addin as part of the UML model of the javaee.jar java library. You can check the stereotypes by examining the javax package in the Library folder of the Model View. Most of the interesting stereotypes are found in the javax.ejb package. This includes the <<Stateless>> and <<Stateful>> stereotypes indicating different kinds of session beans and the <<Remote>> and <<Local>> stereotypes used to define different kinds of bean interfaces.

It might also be worthwhile to investigate the javax.persistence package that contains, among others, the stereotype <<Entity>>, used to mark a class as a persistent entity to be stored in a data base on the server. Some other useful stereotypes in this package include:

- <<Table>>: Used to specify the mapping between an entity and a database table.

- <<Column>>: Used to specify the name of a database table column for an attribute in an entity.

34

Java コードジェネレータ リファレンス

この章は Java コードジェネレータのリファレンスガイドです。UML 言語から Java 言語への翻訳について説明しています。生成 Java コードのカスタマイズについても説明します。

概要

Java コード ジェネレータは、UML 言語構成要素 を Java 言語構成要素にマッピングします。このマッピングプロセスは、UML 言語構成要素を 2 つのカテゴリに区別して行われます。

1. Java のネイティブ表現を持つ UML 構成要素（クラス、インターフェイスなど）。
2. Java のネイティブ構成要素が存在しない UML 構成要素（状態機械、シグナルなど）。

第一のカテゴリに分類される UML 要素については、Java コード ジェネレータによる翻訳とは、UML 要素の Java 構文を生成先の `java` ファイルに出力することを意味します。つまり、このカテゴリの UML 要素は、直接 Java 言語に翻訳できます。

一方、第二のカテゴリに分類される UML 要素については、このような単純な方法では翻訳できません。Java コード ジェネレータは、第二のカテゴリの UML 要素を、まず、第一のカテゴリの UML 要素（つまり、直接 Java 言語に翻訳可能な要素）に変換する必要があります。この手法で、たとえば状態機械は、特定のライブラリ クラスを継承し、特定の属性とメソッドを持つ Java クラスに変換されます。

元の設計 モデルに手を入れずにこの変換を行うために、Java コード ジェネレータは、まず元のモデルの該当部分をコピーして、そのコピーに対して必要な変換を実行します。コード生成が完了すると、コピーは破棄されます。

本章では、主に、Java コード ジェネレータに組み込まれた変換機能が実装する、UML から Java への翻訳規則について説明します。また、本章の最後では、組み込み済みの変換機能以外の、カスタム変換機能を実装する方法も説明します。このカスタマイズの仕組みを使って、デフォルトの翻訳規則をカスタマイズしたり、Java コード ジェネレータが変換をサポートしていない UML 要素を Java 言語に変換するためのカスタム ジェネレータを実装できます。たとえば、ステレオタイプ化された UML 要素のコード生成をカスタマイズして、そのステレオタイプにコード レベルの意味を付与できます。

参照

生成 Java コードが使用する Java ランタイム ライブラリについては、Java ランタイム フレームワークを参照。

Java から UML への翻訳

モデルとソース コードの同期で説明しているように、Tau の Java サポートでは、**Java ソース コードから UML 要素への翻訳**もサポートしています。Java ソース コードから UML 要素への翻訳は、既存の Java コードを Tau にインポートするときや、Java ソース ファイルの生成後に加えた変更を反映してモデルを更新するとき（ラウンドトリップ エンジニアリング）に発生します。

通常、Java から UML への翻訳ルールは、UML から Java への翻訳ルールの正反対になります。ただし、上の第二のカテゴリに属する UML 要素、つまり、直接 Java に翻訳できない UML 要素（状態機械など）については、変換後に生成されたその要素の Java コードパターンから UML に翻訳し直すことは、基本的にできません。ラウンドトリップ エンジニアリング手法で、生成後のファイル内の変更を元のモデルに反映させ

たい場合は、この問題に注意する必要があります。一般的に、ラウンドトリップは、UML 構成要素に直接マッピングできる Java 構成要素についてのみサポートされます。当然ながら、この問題は、既存の Java コードをインポートするときには発生しません。既存 Java コードには、変換された UML 構成要素は含まれないからです。

本章の内容

以降の章では、Java コード ジェネレータで Java に変換できる UML サブセットを説明します。ここで説明しない UML 言語構成要素はサポートされていません。したがって、翻訳時に無視されます。

サポートされている各 UML 構成要素に対して翻訳ルールを示してゆきます。規則に例外がある場合はそれらも示します。

ほとんどの翻訳ルールについて、テキスト形式の UML と Java 構文を使用して例を示してゆきます。

注記

例は、翻訳ルールを説明することを主目的としており、生成コードの内容を詳細に説明するものではありません。また、理解しやすくするため、例にはコードの一部のみが示されています。エラー処理などの重要な部分も省略されていることがあります。したがって、コード例を自分のモデルにコピー / 貼り付けして使用する場合は、注意が必要です。

一般的な翻訳ルール

ここでは、翻訳対象の各種エンティティに適用される一般的な翻訳ルールを説明します。

定義の名前

Java 定義の名前は、その翻訳元の UML 定義の名前と同じです。

UML 名が正式な Java 識別子ではない場合（識別子に空白が含まれている、Java 予約語である、など）、名前変換は行われません。UML で使用している名前が Java でも有効であることを確認する必要があります。

また、一部の Tau コード ジェネレータでサポートされている **<<ansiName>> ステレオタイプは、Java コード ジェネレータではサポートされません。**日本語などを使用したモデルでは注意が必要です。

型付けされた定義の型

型付けされた定義（たとえば attribute、parameter などの型を持つ定義) の型参照は、Java の対応する型参照に翻訳されます。

集約（**Aggregation**）種別について

Java コード ジェネレータは、型付けされた定義の集約（aggregation）の種別を考慮しません。参照（reference）集約、共有（shared）集約、パート（part）集約は、生成される Java 定義では、すべて「参照」となります。

ただし、パート型の UML 属性（属性が集約種別「composition」である）で、その属性の多重度が 1 の場合は、対応する Java 属性は、Java 型の新しいインスタンスを作成する初期化コード（コンストラクタ呼び出し）となります。

例 **405:** 型の参照と集約（**aggregation**）種別の翻訳 ——————————————

UML

```
class AC {}

class Class1 {
  AC 'ref';
  shared AC sh;
  part AC prt;
}
```

Java

```
class AC {}

class Class1 {
    AC ref;
    AC sh;
    AC prt = new AC();
}
```

——————————————————————————————————————

定義済みのデータ型

定義済みのデータ型への参照は、Java でのデータ型への、同名の参照に翻訳されます。これは、Java と UML の両方の定義済みのデータ型の参照に当てはまります。

boolean 型や byte 型などの Java 組み込み型の UML 表記は、TTDJavaPredefined という名前のライブラリに含まれています。通常この型は、Java コード生成を目的としてモデルで使用される定義済みデータ型です。

UML の定義済みのデータ型（Charstring など）を使用する場合は、生成後のコードで使用するデータ型の Java 定義を、自分で行う必要があります。

例 **406:** 定義済みデータ型の翻訳 ——————————————

UML

```
class Class1 {
  boolean b;
  Charstring str;
}
```

Java

```
class Class1 {
    boolean b;
    Charstring str;
}
```

Charstring 型の Java 実装を提供しないと、‘str’ の定義に関するコンパイルエラーが発生することに注意してください。

コレクションと多重度

多重度 > 1（コレクション）を持つ属性（関連）は、直接には Java コードとして生成できません。この概念が Java 言語によってサポートされていないからです。

この属性のマッピングは、多重度が、非形式か形式的かによって異なります。

非形式多重度（**Informal Multiplicity**）

非形式多重度が使用されている場合（属性のプロパティが InformalMultiplicity == True）、多重度は無視され、属性の型がそのまま Java に翻訳されます。

したがって、データ型が List<B> の場合、属性の多重度に関係なく、この型が Java にマッピングされます。属性にデフォルトのコンテナ型と異なるコレクション型を指定する必要があり、かつ、分析の点から多重度を指定する必要がある場合などには、このやり方が適しています。

形式的多重度（**Formal Multiplicity**）

形式的な多重度が使用されている場合、多重度 > 1 の属性の型は、<<containerType>> ステレオタイプのインスタンスの有無によって決まる暗黙のコンテナ型となります。

どのデフォルト コンテナ型を使用するかを制御する方法については、<<containerType>> ステレオタイプを参照してください。

新しい Java プロジェクトの作成時には、デフォルトで <<containerType>> がモデル レベルに適用されています。このステレオタイプでどのデータ型が指定されるかは、使用する Java 処理系によって決まります。

- Java 5 以降では、デフォルトのコレクション型は java.util.Vector<Any> です。Any は属性の型です。
- Java 1.4 以前では、デフォルトのコレクション型は java.util.Vector です。

例 **407:** 形式的な複数多重度のデフォルト マッピング

UML

```
myAttribute : myType [*];
```

Java 5 以降

```
Vector<myType> myAttribute;
```

Java 1.4 以前

```
Vector myAttribute;
```

多重度を表現するために Vector が必要になる場合は、`java.util.Vector` へのインポート依存関係が自動的にモデルに追加されます。こうすることで、コード生成時に import 文が正しくソース コードに書き込まれます。

属性を保持している要素が Java ファイル アーティファクトによってマニフェストされている場合は、アーティファクトにインポートが追加されます。それ以外の場合は、インポートは要素自体に追加されます。

注記

`java.util.Vector` の import 文が追加されるのは、Vector が必要となる値に多重度を設定 / 変更した場合のみです。Java コードを古いモデルから生成する場合は、import 文を手動で追加する必要があります。デフォルトのコンテナ型を変更する場合、指定したコンテナ型にインポート依存関係を追加して、生成コードに正しい import 文が入るようにする必要があります。

定義の可視性

UML 定義の可視性は、対応する Java 定義と同じ可視性に変換されます。

注記

定義のデフォルトの可視性は Java と UML では異なります。したがって、UML の可視性を常に明示すること、つまり可視性を不明確にしないことが重要です。UML の可視性の詳細については、可視性を参照してください。

上記ルールの例外として、アクティブ クラスと状態機械のメンバー定義があります。これらのメンバーに対応する Java メンバーは、UML 上での可視性に関わらず、public 可視性になります。public 可視性は必須です。なぜならば、これらの Java メンバーは、UML モデルの合成状態やインライン 状態機械に対応した状態クラスと状態機械 クラスからアクセスされる可能性があるからです。例については、1220 ページの例 459 を参照してください。

修飾名

UML では、定義は、完全修飾名または相対修飾子を持つ名前によって参照されます。

Java は相対スコープ修飾子をサポートしません。Java では、名前は完全修飾されるか、修飾子をまったく持たないかのいずれかでなければなりません。修飾子をまったく持たない場合は、使用する定義に応じた import 文が必要となります。

UML の完全修飾名は、Java の完全修飾名に翻訳されます。

UML での完全修飾とは、グローバル スコープ修飾子（::）で始まる修飾子の指定と、グローバルスコープにある定義の参照で始まる修飾子の指定の両方を意味します。

相対修飾子を持つ UML 名は、修飾子のない単純な Java の名前に翻訳されます。必要となる import 文は、すでに生成されていない限り、ファイルの先頭に生成されます。

なお、別の方法として、常に Java の完全修飾名を生成する方法もありますが、コードの可読性が低いので使用されていません。

例 **408:** 修飾名の翻訳 ——————————————————————————

UML

```
// Relative qualifier to
com.telelogic.tau.tor.DispatchableClass
class C : tor::DispatchableClass {
  // Relative qualifier to java.util.concurrent.TimeUnit
  public TimeUnit tu;

  // Full qualifier to java.lang.Integer
  private ::java::lang::Integer i;
}
```

Java

```
import com.telelogic.tau.tor.*;
import java.util.concurrent.*;

class C extends DispatchableClass {
  public TimeUnit tu;
  private java.lang.Integer i;
}
```

——————————————————————————————————————

コメント

UML 定義に付加されたコメントは、Javadoc 型のコメント（/ ... */）に翻訳されます。**

Javadoc では、クラスやインターフェイスなど、特殊な定義にのみコメントが許されます。そのような定義以外の定義に UML 上でコメントが付加されている場合は、Java には翻訳されません。

通常の Java コメントは、Java ソース ファイルの任意の場所に追加できますが、同期を取ったときに UML モデルに反映されるのは、有効な Javadoc コメントのみです。それ以外のコメントは、ソース ファイルのみに存在します。

例 **409:** コメントの翻訳 ——————————————————————————

UML

```
class C comment "A JavaDoc comment" {}
```

Java

```
/**A JavaDoc comment*/
class C {
}
```

非名前ベース参照

参照が名前で行われない場合（GUID またはポインタで行われる）、その参照は名前ベースの参照に翻訳されます。参照が必ず前と同じターゲットと結び付けられるように、最小限の関連修飾子が追加されます。

現在この翻訳ルールは、参照の一部（エディタ内でグラフィカル構文を使用して編集される参照）に適用されます。

パッケージ（**Package**）

空でない UML パッケージは、Java パッケージに翻訳されます。

UML パッケージが <<javaPackage>> によってステレオタイプ化されていない場合は、このステレオタイプが適用されて、JavaView で表示できるようにします。

空の UML パッケージは Java に翻訳されません。空のパッケージのフォルダがファイル システムに作成されますが、UML パッケージが空である限りその中身は空です。

例 **410:** パッケージの翻訳

UML

```
package NJP {
  class Class1 {}
}

package Empty {}
```

Java

```
package NJP;

class Class1 {

}
```

依存関係（**Dependency**）

依存関係は、UML モデルでさまざまな形で使用でき、多くの場合、非形式と解釈されます。しかし、一部の特殊な依存関係は、Java コードに生成されます。ここでは、このような依存関係の使い方を説明します。以下のカテゴリに含まれない依存関係は、Java コードに翻訳されません。

インポート依存とアクセス依存

UML モデル内のインポート依存とアクセス依存の一部は、Java の import 宣言に自動マッピングされません。

`.java` ファイルを表すアーティファクトのインポート依存とアクセス依存のみが、対応するファイルに import 宣言として生成されます。パッケージから別のパッケージへの依存など、他の要素間のインポート依存は、`.java` ファイルに生成されません。

UML のアクセス依存は、Java の single-type-import 宣言にマッピングされます。UML のインポート依存は、Java の type-import-on-demand 文にマッピングされます。

例 **411:** アクセス依存とインポート依存の翻訳

UML

```
<<access>> dependency to java::util::Vector
<<import>> dependency to java::util
```

Java

```
import java.util.Vector;
import java.util.*;
```

注記

これらのマッピング ルールに従って構文的に正しい Java コードを得るには、パッケージのインポート時にはインポート依存を使用し、個々のモデル要素のインポート時にはアクセス依存を使用する必要があります。

また、Java コードに生成されるのがアーティファクトからの依存関係であることに注意してください。UML では、インポート依存とアクセス依存は、通常、パッケージ間またはクラス間の関係です。Java コード生成時の依存関係の扱いを簡単にするため、モデルをベースとしてソース コードを更新する際、アーティファクトを所有するパッケージと、アーティファクトによって表現されるクラスから、依存関係がアーティファクトに*コピー*されます。したがって、Java コードの生成時には、これらの依存関係がソース コードに生成されます。しかし、Java コード内で import 文が削除されても、UML モデル内のクラス間またはパッケージ間の対応する依存関係は、**削除されません**。これらの依存関係は、UML モデル内で手動で削除する必要があります。そうしないと、次にモデルからソース コードを更新したときに、import 文が再生成されてしまいます。

多くの場合、Java コード ジェネレータは必要な import 文を自動的に生成できます。たとえば、UML の相対修飾子（修飾名参照）を正しく処理するため、必要な import 文が自動的に生成されます。

クラス（Class）

UML のクラスは、Java のクラスに翻訳されます。

クラスに abstract のマークが付いている場合は、Java クラスは abstract となります。クラスに finalized のマークが付いている場合は、Java クラスは final と宣言されます。

例 **412:** クラスの翻訳 ——————————————————————————

UML

```
public abstract class A {}

public finalized class B {}

public class C {}
```

Java

```
public abstract class A {}

public final class B {}

public class C {}
```

——————————————————————————————————————

ネストされたクラス

ネストされた UML クラス（別のクラスのメンバーとして定義されたクラスなど）は、Java の静的なネストされたクラスに翻訳されます。

Java のネストされたクラスのセマンティクスは、UML のネストされたクラスとは異なります。Java では、ネストされたクラスは、内部クラスと静的なネストされたクラスの 2 種類があります。Java の静的なネストされたクラスは、UML のネストされたクラスとセマンティクスが似ています。

<<innerClass>> によってステレオタイプ化された、ネストされた UML クラスは、Java の内部クラスに翻訳されます。

例 **413:** ネストされたクラスの翻訳 ——————————————————————

UML

```
class Outer
{
    class Nested1 {}

    class <<innerClass>> Nested2 {}
}
```

Java

```
class Outer
{
    static class Nested1 {}
```

```
    class Nested2 {}
}
```

アクティブ クラス

コンストラクタ 状態機械（分類子の振る舞い状態機械）を含むアクティブ クラスは、TOR クラス DispatchableClass を拡張した Java クラスに翻訳されます。

生成されたクラスには次のメンバーが含まれます。

状態機械 クラス（分類子の振る舞い）型の属性「m_sm」

- メソッド「init」の再定義。その実装は、状態機械 クラスのインスタンスを作成してそれを「m_sm」属性に格納します。また、継承した実装の呼び出しも行います。UML クラスに「init」操作がすでにある場合、もう 1 つ生成されることはありません。
- 仮想関数メソッド「start」の再定義。その実装は、継承された実装を呼び出して、DispatchableClass クラスの状態機械を開始します。また、実装は所有側の DispatchableClass インスタンスの一部である各アクティブ クラス インスタンスを開始します。アクティブ クラスのインスタンスが開始されると、含まれるすべてのインスタンスも再帰的に開始されます。UML クラスに「start」操作がすでにある場合、もう 1 つ生成されることはありません。
- メソッド「receive」の再定義。その実装は、継承されるメソッドを呼び出します。
- メソッド「getClassifierBehavior」の再定義。その実装は「m_sm」属性を返します。

例 **414:** 分類子の振る舞い状態機械を持つアクティブ クラスの翻訳

UML

```
active class C
{
    statemachine initialize {}
}
```

Java

```
public class C extends DispatchableClass
{
    public void init()
    {
        m_sm = new C_initialize(this);
        super.init();
    }

    public void start()
    {
        super.start();
    }

    public boolean receive(Event e)
    {
        return super.receive(e);
```

```
    }

    public StateMachine getClassifierBehavior()
    {
        return m_sm;
    }

    C_initialize m_sm;
}
```

インターフェイス（Interface）

UML のインターフェイスは、Java のインターフェイスに翻訳されます。

例 **415:** インターフェイスの翻訳

UML

```
public interface Ifc {}
```

Java

```
public interface Ifc {};
```

シグナルをもつインターフェイス

UML インターフェイスが 1 つまたは複数のシグナルを含む場合、Java インターフェイスは TOR の EventReceiver インターフェイスを継承します。つまり、このインターフェイスを実装するクラスが、イベントを受信できる必要があることを意味します。

例 **416:** シグナルを含むインターフェイスの翻訳

UML

```
public interface Ifc {
  public signal sig;
}
```

Java

```
public interface Ifc extends EventReceiver {
  public static class sig2 extends Event {
    public sig2() {}
    public static boolean isTypeOf(Event e) {
      return e instanceof sig2;
    }
  }
};
```

UML のシグナルをもつインターフェイスを実装するクラスが、必ずしも状態機械を含んでいる必要がないことに注意してください。状態機械なしのクラスを使用すること、または、シグナルを受信したときに何が起こるのかを定義する目的のためだけに `EventReceiver::receive` を実装することが可能です。ただし、Java の EventReceiver インターフェイスは他のメソッドを含んでおり、これらの実装も提供する必要があることに注意してください。通常、これらのメソッドを実装する最も簡便な方法は、クラスを UML でアクティブにすることです。こうすることによって、対応する Java クラスは DispatchableClass を継承することになり、これらのメソッドの適切なデフォルト実装をもつことができます。

ステレオタイプ（**Stereotype**）

<<Metadata>> ステレオタイプが適用されている UML ステレオタイプは、Java の注釈（Annotation）に翻訳されます。

Java 5 以降では、特定の種類の定義に対して、注釈（Annotation）を付加できます。UML では、注釈は、<<Metadata>> ステレオタイプが適用されたステレオタイプで表されます。<<Metadata>> ステレオタイプ以外の UML ステレオタイプは、Java に翻訳されません。

例 **417:** ステレオタイプの翻訳

UML

```
<<Metadata>> stereotype Stereo extends Definition [0 .. 1] {}

<<Stereo>> class MyClass {}
```

Java

```
@interface Stereo {}

@Stereo class MyClass {}
```

既存の Java ソース コードや JAR ファイルのインポートによってモデルに挿入される注釈ステレオタイプは、TTDMetamodel::Definition を拡張します。これによって、これらのステレオタイプを全種類の定義に適用できます。

注釈ステレオタイプは、Java クラス図の「パレット」や［モデル ビュー］の［新規］メニューから手動でも作成できます。この場合、ステレオタイプをモデル内のある定義に適用するためには、TTDMetamodel::Definition の拡張を追加する必要があります。この作業は、［スタンダード ビュー］で行う必要があります。拡張は、Java View では表示されない UML の概念だからです。また、拡張の追加は、Java コードの生成後、Java 注釈を含む生成ファイルから再度モデルを更新する方法でも可能です。

ステレオタイプ属性

ステレオタイプの属性は、<<AnnotationElement>> によってステレオタイプ化されている場合は、Java の注釈要素に翻訳されます。

ステレオタイプ属性のデフォルト値は、注釈要素の対応するデフォルト値に翻訳されます。

例 **418:** ステレオタイプ属性の翻訳 ——————————————————————

UML

```
<<Metadata>> stereotype S extends Definition [0 .. 1] {

  <<AnnotationElement>> int id;
  <<AnnotationElement>> String desc = "[N/A]";
}

<<Stereo(.id = 14, desc = "foo".)>> class MyClass {}
```

Java

```
@interface S {
  int id();
  String desc() default "[N/A]";
}

@Stereo(id = 14, synopsis = "foo") class MyClass {

}
```

——————————————————————————————————————

属性（**Attribute**）

クラスに定義されている非定数 UML 属性は、対応する Java クラスの Java 属性に翻訳されます。

クラスまたはインターフェイスに定義されている定数 UML 属性は、対応する Java クラスの final Java 属性に翻訳されます。

アクセス修飾子またはデフォルト値のマッピングは、以下の例に示すように、簡単に行うことができます。

例 **419:** 属性の翻訳 ——————————————————————

UML

```
public class C {
  public int x = 14;
  public const int y = 15;
}
```

Java

```
public class C {
```

```
  public int x = 14;
  public final int y = 15;
}
```

ステレオタイプに定義されている属性は、ステレオタイプ属性 に記述されているルールに従って翻訳されます。

静的属性

静的な UML 属性は、静的な Java 属性に翻訳されます。UML 属性が静的定数の場合、Java 属性は「static final」属性となります。

例 **420:** 静的属性の翻訳

UML

```
public class C {
  public static int x = 14;
  public static const int y = 15;
}
```

Java

```
public class C {
  public static int x = 14;
  public static final int y = 15;
}
```

操作（**Operation**）

クラスまたはインターフェイス内の UML 操作は、対応する Java のクラスまたはインターフェイス内の Java メソッド定義に翻訳されます。

クラス内の静的な UML 操作は、対応する Java クラス内の静的な Java メソッド定義に翻訳されます。

例については、1189 ページの例 421 を参照してください。

操作本体（**Operation Body**）

UML の操作本体は、Java メソッドの本体に翻訳されます。

なお、UML では、操作本体はインライン定義形式（操作定義に操作本体がある）か、スタンドアロン形式（操作定義を参照する）のいずれかですが、Java では、すべてのメソッドはインライン定義形式になります。

例 **421:** 操作本体の翻訳

UML

```
class C {
  void foo(){ // Inline operation body
    return;
  }
  void bar();
  static void z(); // No defined body
}
void C::bar(){ // Stand-alone operation body
  return;
}
```

Java

```
class C {
  void foo(){
    return;
  }

  void bar(){
    return;
  }

  static void z() {}
}
```

UML 操作に操作本体がない場合は、対応する Java メソッドには、デフォルトの空の本体が生成されます。

状態機械実装のための操作

状態機械の実装用の操作は、Java メソッドに翻訳されます。

コード生成がサポートされる状態機械の実装は、開始遷移とローカル属性定義のみを持つものです。状態を含む状態機械の実装は、サポートされていません。開始遷移でのアクションは、対応する Java メソッドのステートメントに翻訳されます。

例 **422:** 状態機械の実装のための操作の翻訳 ———————————————————

UML

```
class C {
  public int m_op( int p1) statemachine {
    int i = 0;
    start {
        {
            {
                i = p1;
            }
        }
        return p1;
    }
  }
}
```

Java

```
class C {
```

```
  public int m_op( int p1) {
    int i = 0;
      {
        i = p1;
      }
      return p1;
  }
}
```

操作パラメータ

UML 操作のパラメータは、操作の翻訳である Java メソッドの仮パラメータに翻訳されます。

パラメータ方向が return である最初のパラメータの型が、翻訳された Java メソッドでの return 文で戻す型になります。return パラメータがない場合は、void メソッドが生成されます。

UML の他のパラメータ方向（inout、out など）は、Java に翻訳されません。

例 **423:** 操作パラメータの翻訳

UML

```
int foo(int p1, in int p2, inout int p3, out int p4);
void bar();
```

Java

```
int foo(int p1, int p2, int p3, int p4);
void bar();
```

型指定エンティティの一般ルール（型付けされた定義の型と比較してください）は、戻りパラメータにも当てはまります。

Java コード生成では、パラメータのデフォルト値設定はサポートされていません。

パラメータの多重度

パラメータに指定した多重度の影響は、他の任意の定義の場合と同じです（コレクションと多重度を参照してください）。ただし、パラメータの多重度は、パラメータがオプションであること（指定した多重度の範囲に 0 が含まれる）を示すためにも使用します。

コンストラクタ

UML のコンストラクタは、Java のコンストラクタに翻訳されます。

なお、UML のコンストラクタは、クラス操作「initialize」です。

コンストラクタ初期化子

UML のコンストラクタ初期化子は、Java の属性割り当て、または Java のコンストラクタ実装内の `super()` 呼び出しに翻訳されます。

UML の基底クラスの初期化には、ほかに 2 つの方法があります。1 つは「base」コンテキスト キーワードを使用する方法、もう 1 つは基底クラスの名前を参照する方法です。

アクションは以下の順序で生成されます。

1. `super()` の呼び出し。最大で 1 つです。
2. クラス内の定義順と同じ順序での属性の割り当て。

例 **424:** コンストラクタ初期化子の翻訳

UML

```
class D
{
    public D(long) {}
}

class C : D
{
    public C(boolean b) : m_c(true), m_b(b), base(5) {}

    private boolean m_b;
    private boolean m_c;
}
```

Java

```
class D
{
  public D(long) {}
}

class C extends D
{
  public C(boolean b)
  {
    super(5);
    m_b = b;
    m_c = true;
  }

  private boolean m_b;
  private boolean m_c;
}
```

デストラクタ

UML のデストラクタは、`finalize` メソッドのオーバロードに翻訳されます。

なお、UML のデストラクタは、クラス操作「finalize」です。

例 **425:** デストラクタの翻訳

UML

```
class D
{
    public ~D(){}
}
```

Java

```
class D
{
  @Override
  public finalize() {}
}
```

抽象操作

abstract（抽象）UML 操作は、Java の abstract 指定のメソッドに翻訳されます。

抽象操作を含む Java クラスは、抽象クラスとして扱われます。

例 **426:** 抽象操作の翻訳

UML

```
class S {
    abstract int f1();
}
class D : S {
    redefined int f1(); // Redefines S::f1
}
```

Java

```
abstruct class S {
  abstract int f1();
};
class D extends S {
  @Override
  int f1() {}
};
```

仮想、再定義、ファイナライズ操作

virtual（仮想）UML 操作は、Java の通常のメソッドに翻訳されます。

Java では、すべての非静的操作は暗黙的に仮想化されているためです。

redefined（再定義）UML 操作は、@Override 注釈付きの Java のメソッドに翻訳されます。

finalized UML 操作は、Java の final メソッドに翻訳されます。

例 **427: virtual** 操作、**redefined** 操作、**finalized** 操作の翻訳 ————————————

UML

```
class S {
    int f1();
    virtual int f2();
    virtual int f3();
    virtual int f4();
}
class D : S {
    int f1(); // Hides S::f1
    virtual int f2(); // Hides S::f2
    redefined int f3(); // Redefines S::f3
    finalized int f4(); // Redefines S::f4
}
```

Java

```
class S {
  int f1();
  int f2();
  int f3();
  int f4();
};
class D extends S {
  int f1();
  int f2();
  @Override
  int f3();
  @Override
  final int f4();
};
```

——

Java と UML のセマンティクスの違いに注意してください。上記の例では、`D::f1()` は `S::f1()` の実装を隠蔽しています。これに対して、Java では、`D.f1()` は `S.f1()` を「上書き」します。このように微妙な問題があるので、生成 Java コードを適切なオプションでコンパイルして、`@Override` 注釈が使われていないことを確認することを推奨します。

例外指定

UML 操作の例外指定（「throw」宣言）は、その操作の翻訳である Java メソッドの例外の指定に翻訳されます。

UML では例外指定はオプションですが、Java では、例外を発生する可能性をもつメソッドを呼び出すメソッド上への例外指定は、必須です。

例 **428:** 例外指定の翻訳 ————————————————————————————————

UML

```
void foo() throw Exc1, Exc2;
```

Java

```
void foo() throws Exc1, Exc2;
```

同期操作

<<synchronized>> によってステレオタイプ化された UML 操作は、Java の synchronized （同期）メソッドに翻訳されます。

例 **429:** 同期操作の翻訳

UML

```
class S {
  <<synchronized>> void foo(){}
}
```

Java

```
class S {
  synchronized void foo() {}
};
```

Main 操作

生成コードを実行可能とするために Java コードに「main」操作を生成するには、モデル内のクラスの 1 つに、「main」に対応する UML 操作を定義する必要があります。この UML 操作のシグニチャは以下のとおりです：

```
public static void main( Array<String> args);
```

「main」操作の追加を簡単に行うために、Tau は以下のような方法を提供しています：

1. ［モデルビュー］で操作を追加したいクラスを選択します。
2. 表示されるコンテキストメニューから［ユーティリティ］>［Generate Main Method］を選択します。

以下の規則にしたがって、「main」操作の定義が生成されます：

ビルド アーティファクトにマニフェストされているアクティブクラスごとに、1 つのインスタンスが作成される。その各インスタンスは、1 つの Dispatcher に追加され、初期化、開始される。最終的に、Dispatcher 上の run メソッドが呼び出される。

つまり、デフォルトの「main」操作は、完全に同期的（単一スレッド）なアプリケーションで実行されます。当然、このデフォルトの実装は、作成したいプログラムに合わせて変更可能です。

汎化（Generalization）

2 つの UML クラス間またはインターフェイス間の汎化は、対応する Java のクラス間またはインターフェイス間の継承（`extends`）に翻訳されます。

操作間の汎化は翻訳されません。

例 **430:** 汎化の翻訳 ———————————————————————————————

UML

```
class S {
}
class D : S {
}
```

Java

```
class S {
};
class D extends S {
};
```

———

関連（Association）

名前のない一方向関連は、UML モデルの属性として表記されます。したがって、このような属性の翻訳は、属性（Attribute）のルールに従います。

他の関連は、Java コード生成ではサポートされません。

データ型（Datatype）

UML のデータ型は、Java の列挙（enum）にマッピングされます。各リテラルは、1 つの enum 定数に翻訳されます。

コンストラクタ、属性、操作などのデータ型のメンバーは、対応する Java の列挙メンバーに翻訳されます。

例 **431:** データ型の翻訳 ———————————————————————————————

UML

```
public datatype Status {
  literals OK = new Status(“All is OK”);
  literals PROBLEM = new Status(“Problem occurred”);

  private const String description;
  private Status(String s) {
    this.description = s;
  }
}
```

Java

```
public enum Status {
  OK(“All is OK”), PROBLEM(“Problem occurred”);

  private final String description;
  private Status(String s) {
    this.description = s;
  }
}
```

enum リテラルの初期化時に、コンストラクタを呼び出すことができます。

式（Expression）

UML 式は、式の各部分ごとに Java に翻訳されます。定数式の評価は翻訳中には行われません。

UML 式の翻訳の大半は、以下の表に示すとおり、きわめて単純明快です。

UML 式	Java 式	UML 例	Java 例
かっこ付きの式	かっこ付きの式	`(a+b)`	`(a+b)`
単項式	単項式（Java 演算子は指定 UML 演算子による）	`not m_bOk ++var`	`!m_bOK ++var`
二項式	二項式（Java 演算子は指定 UML 演算子による）	`a + b`	`a + b`
this 式	'this' 式	`this`	`this`
呼び出し式	呼び出し式	`foo(3)`	`foo(3)`
フィールド式	メンバー アクセス式	`x.y`	`x.y`
インデックス式	添字指定演算子（'[]'）	`coll[4]`	`coll[4]`
create 式	'new' 演算子	`new C(1,2)`	`new C(1,2)`
条件式	条件式	`b ? x1 : y1`	`b ? x1 : y1`
実数値	浮動小数点型リテラル	`float a = 3.14;`	`float a = 3.14;`
整数値	整数型リテラル	`long a = 4;`	`long a = 4;`
charstring 値	string 型リテラル	`String s = "hola";`	`String s = "hola";`
文字値	文字型リテラル	`char c = ‘q’;`	`char c = ‘q’;`

その他の式の翻訳については、以降のセクションで説明します。

識別子

識別子は、それがバインドされている定義の名前と同じ方法で翻訳されます。

このルールは、識別子が式の一部である場合と、参照を表す場合の両方に適用されます（定義の名前と比較してください）。

UML または Java の定義済みデータ型の参照となる識別子の翻訳については、定義済みのデータ型で説明しています。他の定義済み UML 定義への参照については、非データ型 UML 定義済み定義への参照で説明しています。

基底クラスへの参照

アクションコードで UML クラスの基底クラス（上位クラス）への参照を使うには修飾子を付けた基底クラス名を使用します。こういった参照は、'super' キーワードを使った Java 参照に翻訳されます。

シンタイプを使った基底クラス参照でもこの翻訳規則が適用されます。

例 **432:** 基底クラス参照の翻訳

UML

```
class D
{
    public void foo() {}
}

class C : D
{
    syntype inherited = D;
    public void foo()
    {
      D::foo();
      inherited::foo();
    }
}
```

Java

```
class D {
    public void foo() {

    }
}

class C extends D {

    public void foo() {
        super.foo();
        super.foo();
    }
```

```
}
```

非データ型 UML 定義済み定義への参照

非データ型 UML 定義済み定義は、Java コードジェネレータの特別な処理で取り扱われます。定義済み UML データ型への参照の翻訳については定義済みのデータ型で説明しています。

下表は Java コードジェネレータがサポートする定義済み UML 定義をリストアップしています：

参照される UML 定義	Java 翻訳
is	instanceof
as	Type cast operator

例 **433: UML** 定義済み `is` と `as` の翻訳

UML

```
C var = new C();
if (is<D>(var))
{
    D d = as<D>(var);
}
```

Java

```
C var = new C();
if (var instanceof D)
{
    D d = (D)var;
}
```

非形式式

UML の非形式式は、何も翻訳されず、Java コード内に式のテキストをそのままコピーしたものになります。

したがって、Java コード ジェネレータでは非形式式を利用して、UML 構成要素からは直接生成できない Java コードを生成させることができます。また、UML モデルで表記できないレガシー Java コードとインターフェイスするためにも、非形式式が役立ちます。

非形式式が UML 定義の参照を含む場合、その式が生成された Java にコピーされる前に、識別子 の通常のルールに従って翻訳されます。

例 **434:** 非形式式の翻訳

UML

```
RemoteClass x = new RemoteClass();
long ext = [[##(x).remoteMethod()]];
```

Java

```
RemoteClass x = new RemoteClass();
long ext = x.remoteMethod();
```

TimerActive 式

UML の TimerActive 式は、Java のタイマー属性上の `isActive` メソッドの呼び出しに翻訳されます。

例 **435: TimerActive** 式の翻訳

1213 ページの例 454 に定義されているタイマーがアクティブかどうかを確認する例

UML

```
boolean b = Clock.isActive();
```

Java

```
Boolean b = timer_Clock.isActive();
```

注記

UML の TimerActive 式で使用される可能性のある実引数は、タイマーがアクティブかどうか調べるのに、どのバージョン（オーバーロードしている可能性がある）を照会すべきかを識別するためのものです。この実引数は Java 翻訳では無視されます。

now 式

UML の now 式は、TOR Time クラスの静的 now メソッドの呼び出しに翻訳されます。

この呼び出しで返される **Time** オブジェクトは、`to_double()` メソッドを呼び出して double 値に変換されます。これによって、そのオブジェクトは double 型の Java 式に対応できます。

例については、1211 ページの例 451 を参照してください。

テンプレート（**Template**）

UML のクラス テンプレートまたはインターフェイス テンプレートは、Java のクラスまたはインターフェイスの Generic 型に翻訳されます。

UML の操作テンプレートは、Generic 型の引数または戻り型を持つ Java のメソッドに翻訳されます。

例 **436:** テンプレート定義の翻訳 ——————————————————————

UML

```
template <type T>
class C
{
  T t;
  public T get_t()
  {
    return t;
  }
}
```

Java

```
class C<T> {
  T t;
  public T get_t()
  {
    return t;
  }
}
```

——————————————————————————————————————

atleast 制約

仮テンプレート パラメータに対する UML の atleast 制約は、対応する Java Generic 型パラメータに対する上限付きのワイルドカード（`extends`）に翻訳されます。

なお、Java の下限付きワイルドカード（`super`）は、UML では表記できません。

例 **437: atleast** 制約の翻訳 ——————————————————————

UML

```
class MyClass {}

template <type T atleast MyClass>
class C {}
```

Java

```
class MyClass {}

class C<T extends MyClass> {}
```

——————————————————————————————————————

注記

UML は、さらに高度な atleast 制約の使い方も数多くサポートしています。ただし、これらの構成要素は Java に対応する仕様がないため翻訳されません。

テンプレートのインスタンス化

UML テンプレートのインスタンス化は、UML テンプレートの翻訳である、Java Generic 型を使用する方法として翻訳されます。

実テンプレート型パラメータが通常の型参照の場合、対応する Java の型参照に翻訳されます。

注記

パラメータの定義の範囲外で使用される Generic 型パラメータのワイルドカードは、Java コードの生成時にはサポートされません。

例 **438:** テンプレート インスタンス化の翻訳

この例では、1201 ページの例 436 のテンプレート C をインスタンス化します。

UML

```
C<MyClass> v1 = new C<MyClass>();
MyClass mc = v1.get_t();
```

Java

```
C<MyClass> v1 = new C<MyClass>();
MyClass mc = v1.get_t();
```

アクション（Action）

UML アクションは Java 文に翻訳されます。

UML アクションの翻訳の大半は、以下の表に示すとおり、きわめて単純明快です。

UML アクション	Java ステートメント	UML 例	Java 例
複合アクション	複合文	`{ v = v + 1; }`	`{ v = v + 1; }`
Continue アクション	continue 文	`continue;`	`continue;`
Break アクション	break 文	`break;`	`break;`
If アクション	if 文	`if (b) { ...} else { }`	`if (b) { ...} else { }`

その他の UML アクションの翻訳については、以降のセクションで説明します。

定義アクション

関連定義が属性（Attribute）である UML の定義アクションは、Java のローカル変数定義に翻訳されます。

例 **439:** 定義アクションの翻訳 ——————————————————————

UML

```
class C {
  public void foo(){
    integer v = 4;
  }
}
```

Java

```
class C {
  public void foo(){
    integer v = 4;
  }
}
```

——

関連定義が属性ではない場合、定義アクションは翻訳されません。

式アクション

UML の式アクションの翻訳は、そのアクションに関連付けられている式の種類によって異なります。

関連する空式がある式アクションは、空の Java ステートメントに翻訳されます。

非形式式に関連する式アクションは、非形式テキスト コピーに翻訳されます。

Call 式に関連する式アクションは、呼び出し式にセミコロン（;）を追加して翻訳されます。

関連する Create 式がある式アクションは、create 式にセミコロン（;）を追加して翻訳されます。

例 **440:** 式アクションの翻訳 ——————————————————————

UML

```
void foo(){
    ;
    [[new C().doIt()]];
    open(true);
    new Integer();
}
```

Java

```
void foo(){
  ;
  new C().doIt();
  open(true);
  new Integer();
}
```

Try アクション

関連する catch 句がある UML Try アクションは、Java の catch 句を持つ try 文に翻訳されます。

例 **441: try** アクションと **throw** アクションの翻訳

UML

```
class C {
  void foo(boolean b) throws InternalError {
    if (b)
      throw InternalError();
  }

  void bar() {
    try {
      foo(false);
    }
    catch(InternalError e)
    {}
  }
}
```

Java

```
class C {
    void foo( boolean b) throws InternalError {
        if (b)
            throw InternalError();
    }
    void bar() {
        try
        {
            foo(false);
        }
        catch ( InternalError e)
        {

        }
    }
}
```

Throw アクション

UML の Throw アクションは、Java の throw 文に翻訳されます。

例については、1204 ページの例 441 を参照してください。

Loop アクション

UML の Loop アクションは、Java の while 文、do-while 文または for 文に翻訳されます。

この 3 つの文はすべて同じ構成要素の変形であり、どの文が使用されるかは UML モデルで使用されている構文の種類によって決まります。

例 **442: Loop** アクションの翻訳 ——————————————————————

UML

```
int a = 0;
for (int i = 0; i < 10; i++) {
    a = a + 1;
}
while (a > 0)
    a = a - 1;
do {
    a = a + 1;
} while (a < 10);
```

Java

```
int a = 0;
for (int i = 0; i < 10; i++) {
    a = a + 1;
}
while (a > 0)
  a = a - 1;
do {
a= a+ 1;
} while (a < 10);
```

——————————————————————————————————————

注記

Java 5 で導入された for 文のバリエーションである「foreach」は、Tau の UML モデルでは表記できません。

停止アクション

UML の停止アクションは、停止アクションが含まれる状態機械実装に対応する状態機械クラス上の「finish」メソッドの呼び出しに翻訳されます。

例 **443:** 停止アクションの翻訳 ——————————————————————

UML

```
stop;
```

Java

```
finish();
```

NextState アクション

通常の NextState アクション

「通常」の NextState アクション（状態を明示的に指定する NextState アクション）は、次の状態として指定された状態に対応する Java の属性上の「enter」メソッドの呼び出しに翻訳されます。

エントリ接続ポイントを指定する「via」句を持つ NextState アクションは、次の状態として指定された状態に対応する Java の属性上の「enter」メソッドの呼び出しに翻訳されます。

翻訳された接続ポイント属性が、「enter」呼び出しの引数として渡されます。

例 **444: NextState** アクションの翻訳

UML

```
nextstate Idle;
nextstate Idle via Cin;
```

Java

```
m_s_Idle.enter();
m_s_Idle.enter(m_s_Idle.m_sm.Cin);
```

履歴の NextState アクション

履歴の NextState アクションは、NextState アクションを含む UML 実装に対応する Java 状態機械 クラスの TopRegion 属性上の「enterHistory」メソッドの呼び出しに翻訳されます。

例 **445:** 履歴の **NextState** アクションの翻訳

UML

```
nextstate -;
```

Java

```
theTopRegion.enterHistory();
```

詳細な履歴の **NextState** アクション

詳細な履歴の NextState アクションは、NextState アクションを含む UML 実装に対応する Java 状態機械 クラスの TopRegion 属性上の「enterHistory」メソッドの呼び出しに翻訳されます。呼び出しの「deepHistory」フラグは「true」に設定されます。

例 **446:** 詳細な履歴の **NextState** アクションの翻訳

UML

```
nextstate ^-;
```

Java

```
theTopRegion.enterHistory(true /* deepHistory */);
```

シグナル送信アクション

シグナル送信アクションは、動的に生成されるシグナル インスタンスを引数として持つ TOR メソッド Utilities.sendTo() の呼び出しに翻訳されます。

シグナルの受信先を明示的に指定する必要があります。実行時に受信先の算出が必要なシグナル送信アクションはサポートされず、Java に翻訳されません。

例 **447:** シグナル送信アクションの翻訳

UML

```
output c.Ping(“hello”);
```

Java

```
Utilities.sendTo(new Ping1(“hello”), c);
```

シグナル送信アクションが複数のシグナルの送信を指定する場合、各送信シグナルに対して 1 つの「send」呼び出しがあります。

分岐アクション

UML の分岐アクションは、分岐回答ごとに 1 つの case ブランチを持つ Java の switch 文、または分岐回答ごとに 1 つの else ブランチを持つ if 文に翻訳されます。else ブランチへの翻訳の場合、break 文が内部にあると、空の switch 文内に if 文が配置される形になります。

Java の switch 文は、UML の分岐アクションと比べて制約があります。したがって、UML の分岐アクションは、その分岐アクションが以下のいずれかの型の属性を単純に参照する場合のみ、Java の switch 文に翻訳されます。

- プリミティブ データ型（byte、short、char、int）
- 上記のプリミティブ型をラップするクラス（Byte、Short、Character、Integer）
- 列挙型

また、各分岐回答式は単純な値であることが必要です。上記以外のすべての場合、分岐アクションは Java の if 文に翻訳されます。この if 文には分岐回答ごとに 1 つのブランチがあります。

生成された Java の if 文が break 文を含んでいると、if 文全体が空の swtich 文の default 分岐に配置されます。break 文で switch 構文から抜け出せるようにするためです。

例 **448:** 分岐アクションの翻訳 ——————————————————————————

最初の分岐アクション例では、Java の switch 文に翻訳されています。二番目の例では、if 文に翻訳されています。三番目の例では、switch 文内の if 文に翻訳されています。

UML

```
int e1, e2;
int i = 0;

switch (e1) {
  case 10 : {
    i = 1;
    break;
  }

  case 11 :
    break;

  default : {
    i = 3;
  }

}

const int x = 5;
int v;
switch (v) {
  case x : {return;}
  default : ;
}

switch (e2 > 5) {
```

```
      case true: {
        i = 1;
        break;
      }

      case false: {
        i = 0;
        break;
      }
    }
```

Java

```
    int e1;
    int e2;
    int i = 0;
    switch (e1) {
      case 10:
        i = 1;
        break;
      case 11:
        break;
      default:
        i = 3;
        break;
    }

    final int x = 5;
    int v;
    if (((v) == (x)))
    {
      return ;
    }
    else ;

    switch (e2 > 5) {
    default:
      if (((e2 > 5) == (true)))
      {
        i = 1;
        break;
      }
      else if (((e2 > 5) == (false)))
      {
        i = 0;
        break;
      }
    }
```

リターン アクション

UML の操作本体に含まれるリターン アクションは、Java の return 文に翻訳されます。

遷移に含まれているリターン アクションは、遷移を含む UML 状態機械実装の翻訳である Java の状態機械 クラスに属する TopRegion 属性上の、「finish」メソッドの呼び出しに翻訳されます。

例 **449:** 遷移内のリターン アクション

UML

```
start {
    return;
}
```

Java

```
public void initialTransition( ) {
    theTopRegion.finish();
}
```

リターン アクションが経由する接続ポイントを指定している場合、その接続ポイントの翻訳である属性が「finish」呼び出し時の引数として渡されます。

例 **450: Return via** 接続ポイント

終了接続ポイントを指定する遷移内のリターン アクション。

UML

```
start {
    return Cout;
}
```

Java

```
void initialTransition( ) {
    theTopRegion.finish(Cout);
}
```

タイマー設定アクション

タイマー設定アクションは、参照されるタイマーに対応する Java の timer 属性上の「set」メソッド呼び出しに翻訳されます。

第 1 引数が、指定するタイムアウト値を示す式です。タイムアウト式を明示指定しない場合にデフォルトのタイムアウト式が使われるように、タイマー定義をしておく必要があります。第 2 引数は、Java の timer クラス（TOR クラス TimerEvent を拡張するクラス）から動的に作成するインスタンスです。実タイマー パラメータは、このインスタンス作成時の実コンストラクタ パラメータになります。

例 **451:** タイマー設定アクションの翻訳

1213 ページの例 454 で定義されるタイマーの設定

UML

```
set Clock() = now + 4;
set Clock; // Using default timeout value (15)
```

Java

```
timer_Clock.set(new Time(Time.now().to_double() + 4), new
C.Clock());
timer_Clock.set(new Time(Time.now().to_double() + 15), new
C.Clock());
```

タイマー リセット アクション

タイマー リセット アクションは、参照されるタイマーに対応する Java の timer 属性上の「reset」メソッド呼び出しに翻訳されます。

例 **452:** タイマー リセット アクションの翻訳

1213 ページの例 454 で定義されるタイマーのリセット

UML

```
reset Clock();
```

Java

```
timer_Clock.reset();
```

注記

UML のタイマー リセット アクションで使用される可能性のある実引数は、リセットの対象となるタイマーのバージョン（オーバーロードしている可能性がある）を識別するためのものです。この実引数は Java 翻訳では無視されます。

シグナル（**Signal**）

UML シグナルは、TOR クラス Event（イベント）を継承する Java のクラスに翻訳されます。

このクラスには静的な「isTypeOf」メソッドがあります。このメソッドは、与えられたイベントがシグナルに対応するクラスの型かどうかを動的に検査するためのものです。このメソッドの実装には、Java の `instanceof` 演算子が使われます。

Java のイベント クラスは、シグナルの置かれた UML スコープに対応した Java スコープ内に置かれます。そのスコープが Java クラスの場合は、イベント クラスはネストした静的 Java クラスになります。そのスコープが Java パッケージの場合は、イベント クラスは通常の Java クラスになります。

シグナル パラメータ

シグナルにパラメータがある場合、Java イベント クラスには、各シグナル パラメータをパラメータとして持つコンストラクタが作成されます。さらに、シグナル パラメータごとに 1 つの public 属性が作成されます。コンストラクタ パラメータの名前、型、多重度、およびクラス属性は、生成元のシグナル パラメータと同一です。

シグナル パラメータが型のみで名前を持たない場合、対応するコンストラクタ パラメータ、および属性の名前は、「parX」になります。ここで X は、0 から始まるシグナル パラメータの索引値です。

例 **453:** パラメータを持つシグナルの翻訳

UML

```
class C
{
    signal Ping (String, long i = 4);
}
```

Java

```
public class C
{
    public static class Ping extends Event
    {
        public String par0;
        public long i = 4;

        public Ping(String par0, long i)
        {
            this.par0 = par0;
            this.i = i;
        }

        public Ping(String par0)
        {
            this.par0 = par0;
        }

        public static boolean isTypeOf(Event e)
        {
              return e instanceof Ping;
        }
    }
}
```

シグナル クラスの名前は、対応するシグナルの名前を翻訳したものです（定義の名前と比較してください）。ただし、シグナルはイベント クラスなので、同じ UML スコープ内に同じ名前を持つ複数のシグナルが存在する（オーバーロードされている）可能性があります。その場合、同じ名前を持つオーバーロードされたすべてのシグナルについて、名前が、シグナル名とシグナル パラメータを連結した形になります。

タイマー

UML のタイマーは、TOR クラス TimerEvent から継承する Java のクラスに翻訳されます。

これはシグナル（Signal）の翻訳とまったく同じ方法で行われます。さらに、Java クラスに 1 つの属性が生成されます。この属性は「timer_T」（T はタイマーの名前）と呼ばれ、TOR クラス TimerObject 型です。また、この属性は、this をコンストラクタパラメータとして初期化（インスタンス化）されます。

例 **454:** タイマーの翻訳 ——————————————————————————

UML

```
class C
{
    timer Clock() = 15;
}
```

Java

```
public class C
{
    public TimerObject timer_Clock = new TimerObject(this);

    public static class Clock extends TimerEvent
    {
            public static boolean isTypeOf(Event e)
        {
            return e instanceof Clock;
        }
    }
}
```

——————————————————————————————————————

注記

ここでは、デフォルトのタイムアウト値（1213 ページの例 454 の「15」）は、Java の翻訳結果には反映されません。ただし、タイムアウト値を指定しないタイマー設定アクションの翻訳では使用されます。

状態機械（State machine）

UML には 3 種類の状態機械があります。

1. **分類子の振る舞い状態機械。**アクティブ クラスによって所有されるコンストラクタ 状態機械（通常は「initialize」と呼ばれる）です。
2. **インライン 状態機械。**合成状態によって所有される状態機械です。
3. **スタンドアロン 状態機械。**1 つまたは複数の合成状態から参照できる状態機械です。

状態機械の翻訳ルールは、状態機械の種類によって異なります。以下の説明では、状態機械の種類による違いを、上の分類番号別に示します。

UML の状態機械は、TOR クラス StateMachine（状態機械）を拡張した Java のクラスに翻訳されます。

Java の状態機械 クラスの名前は以下のようになります（状態機械の種類によって異なります）。

1. 「C_SM」。C は所有クラスの名前、SM は状態機械の名前です。
2. 「C_S_SM」。S は合成状態の名前、C はその合成状態を含む状態機械実装の翻訳である状態機械 クラスの名前です。SM は状態機械の名前です。
3. 「SM」。SM は状態機械の名前です。

Java の状態機械 クラスには以下のメンバーが含まれます。

- 「m_owner」という名前の属性。この属性の型は状態機械の種類によって以下のように異なります。

1. 所有するアクティブ クラスの翻訳である Java クラス。
2. 所有する状態機械の翻訳である Java クラス。
3. TOR クラス DispatchableClass。

- 状態機械の種類が 2 または 3 の場合は、属性「m_ownerState」も生成されます。この属性の型は、所有側の状態クラス（2）または TOR クラス State（3）です。
- コンストラクタ。その実装は「m_owner」属性を初期化する。クラスに「m_ownerState」属性も含まれている場合、その属性も初期化されます。
- 「getDispatchableClass」メソッド。このメソッドの実装は、評価結果が、状態機械が属する dispatchable クラスとなる式を返します。この式は、状態機械の種類によって以下のようになります。

1. DispatchableClass としての「m_owner」属性。
2. 「m_owner.getDispatchableClass()」
3. 「m_owner」

- TOR クラス TopRegion 型の属性「theTopRegion」。これは初期化されて新しいインスタンス TopRegion となります。
- メソッド「init」の再定義。その実装は addRegion(theTopRegion) を呼び出します。

注記

単純な状態機械を通常の操作として実装できます。状態機械実装が状態を持たない場合は、通常の操作本体に翻訳されます（開始遷移のアクションが操作本体（Operation Body）に入る）。

例 **455:** 分類子の振る舞い状態機械の翻訳（**1**）

UML

```
active class C {
    statemachine initialize {
        ...
    }
}
```

Java

```
public class C_initialize extends StateMachine
{
    public TopRegion theTopRegion = new TopRegion(this);

    public C m_owner;

    public C_initialize(C owner)
    {
          m_owner = owner;
    }

    public void init()
    {
          addRegion(theTopRegion);
    }

    public DispatchableClass getDispatchableClass()
    {
          return m_owner;
    }
}
```

例 **456:** 合成状態 状態機械の翻訳（**2** と **3**）

UML

```
active class C {
    statemachine initialize {
        state S1 : statemachine initialize {}; // 3)
        state S2 : SM {}; // 2)
    }
}

statemachine SM {}
```

Java

```
public class C_initialize extends StateMachine
{
    // see 1214 ページの例 455 for the translation
}

public class C_S1_initialize extends StateMachine
{
    public TopRegion theTopRegion = new TopRegion(this);

    public C_initialize m_owner;
    public C_initialize.S1 m_ownerState;

    public C_S1_initialize(C_initialize owner,
C_initialize.S1 ownerState)
    {
          m_owner = owner;
        m_ownerState = ownerState;
    }
```

```
        public void init()
        {
            addRegion(theTopRegion);
        }

        public DispatchableClass getDispatchableClass()
        {
            return m_owner.getDispatchableClass();
        }
    }

    public class SM extends StateMachine
    {
        public TopRegion theTopRegion = new TopRegion(this);

        public DispatchableClass m_owner;
        public State m_ownerState;

        public SM(DispatchableClass owner, State ownerState)
        {
              m_owner = owner;
            m_ownerState = ownerState;
        }

        public void init()
        {
            addRegion(theTopRegion);
        }

        public DispatchableClass getDispatchableClass()
        {
            return m_owner;
        }
    }
```

この例には、合成状態の翻訳は示されていません。1218 ページの例 458 を参照してください。

状態機械実装からの非ローカル定義のアクセス

UML では、修飾子を使用せずに、定義を状態機械実装から参照できる場合があります。ある状態機械実装から、閉じたスコープ内に定義されているすべての定義にアクセスするような場合です。この定義は、以下のとおりです。

1. ローカル定義（同一状態機械実装内）
2. 包含している状態機械内の定義（合成状態のインライン 状態機械から参照する場合）
3. 包含するアクティブ クラス内の定義
4. グローバル定義

Java では、カテゴリ 2 とカテゴリ 3 の定義の参照は、1 つまたは複数の「m_owner」属性参照を使用するアクセス接頭辞を必要とします。この参照が Java の状態クラスに翻訳されるコンテキスト内にある場合、修飾子にもう 1 つ「m_owner」が追加されます（状態クラスには、所有する状態機械 クラスを参照する「m_owner」属性が含まれているからです）。例については、トリガ付き遷移を参照してください。

状態

UML の状態は、TOR クラス State（状態）を拡張した Java のクラスに翻訳されます。

状態クラスは、静的なネストされたクラスとして、Java 状態機械 クラス内に格納されます。これによって、状態名の名前変換を行う必要がなくなります（2 つの状態機械実装が同じ名前の状態を持つことができます）。

各状態クラスは、状態機械 クラスの「init」メソッドでインスタント化されます。各状態オブジェクトは、その状態専用の属性に格納されます。この属性は「m_s_S」と呼ばれます。S は状態の名前です。可視性は protected です。

Java の状態クラスには次のメンバーが含まれます。

- 所有する状態機械実装の翻訳である状態機械 クラス型の属性「m_owner」。
- 「m_owner」を初期化するコンストラクタ。
- 抽象メソッド「execute」のオーバーロード。状態から発生するすべてのトリガ付き遷移の if 文を含みます。つまり、状態クラスは、その状態から発生するすべてのトリガ付き遷移の実装を含みます。

例 **457:** 状態と開始遷移の翻訳

UML

```
active class C {
    statemachine initialize {
        start {
            nextstate Idle;
        }
        state Idle;
    }
}
```

Java

```
public class C extends DispatchableClass { … }

public class C_initialize extends StateMachine
{
    // … 上記の 状態機械（State machine）の翻訳を参照 …

    public void initialTransition()
    {
           m_s_Idle.enter();
    }

    public void init()
    {
           // …
```

```
            m_s_Idle = new Idle(theTopRegion, this);
               m_s_Idle.init();
        }

        public static class Idle extends State
        {
            public C_initialize m_owner;

            public Idle(Region region, C_initialize owner)
            {
             super(region);
             m_owner = owner;
            }

            public Dispatchable.EventAction execute(Event e)
            {
             // … Transition if-statements …
                 // ( 下記の トリガ付き遷移 を参照 )

             return Dispatchable.EventAction.NoMatch;
            }
        }

        protected Idle m_s_Idle;
    }
```

簡単にするため、この例では状態の翻訳に関係のないものを省略しています。

合成状態

状態機械を含んでいる状態や状態機械を参照する状態は、合成状態と呼ばれます。合成状態は、通常の状態に以下の要素を付加したものとして翻訳されます。

- この状態クラスは、参照先状態機械の翻訳である状態機械 クラスでタイプ指定された属性「m_sm」が与えられます。可視性は public です。
- オーバロードされるメソッド「init」の追加。このメソッドは、「m_sm」を状態機械 クラスの新しいインスタンスに初期化し、作成されるオブジェクトに対して「init」を呼び出します。
- オーバロードされる「getStateMachine」の追加。このメソッドは「m_sm」を返します。

例 **458:** 合成状態の翻訳

UML

1215 ページの例 456 を参照してください。

Java

```
    public static class S1 extends State
    {
        public C_initialize m_owner;

        public S1(Region region, C_initialize owner)
```

```
        {
            super(region);
            m_owner = owner;
        }

        public C_S1_initialize m_sm;

        public void init()
        {
            m_sm = new C_S1_initialize(m_owner, this);
            m_sm.init();
        }

        public StateMachine getStateMachine()
        {
            return m_sm;
        }
    }

    public static class S2 extends State
    {
        public C_initialize m_owner;

        public S2(Region region, C_initialize owner)
        {
            super(region);
            m_owner = owner;
        }

        public SM m_sm;

        public void init()
        {
            m_sm = new SM(m_owner.getDispatchableClass(), this);
            m_sm.init();
        }

        public StateMachine getStateMachine()
        {
            return m_sm;
        }
    }
```

スタンドアロン 状態機械（例内の SM）を参照する合成状態は、「init」メソッドを異なる方法で実装します（この合成状態は、「m_owner」上の `getDispatchableClass()` を呼び出し、dispatchable クラスに対する参照が与えられます）。

開始遷移

開始遷移は、開始遷移を含む状態機械実装の翻訳である Java 状態機械 クラスの「initialTransition」メソッドに翻訳されます。

「initialTransition」の実装は開始遷移のアクションの翻訳です。この翻訳は他のアクション（Action）とまったく同じルールに従います。

開始遷移の翻訳例については、1217 ページの例 457 を参照してください。

トリガ付き遷移

トリガ付き遷移は、Java 状態機械 クラスの 1 つの protected メソッドに翻訳されます。このメソッドは、トリガ付き遷移を含んだ状態機械実装を翻訳したものです。

遷移メソッドの名前は以下のようになります。

1. 遷移にガードもアスタリスク トリガもない場合、
   `trans_<StateNames>_<SignalClassNames>`
2. それ以外の場合、`trans_<StateNames>_<SignalClassNames>_<GUID>`

<StateNames> は、遷移をトリガできる状態の名前であり、<SignalClassNames> は、遷移をトリガするシグナルに対応した、すべてのイベント クラスの名前です。<GUID> は、トリガ付き遷移の GUID です。

それぞれのトリガ付き遷移について、遷移をトリガできる状態に応じて、各状態クラスの「execute」メソッドにも if 文が生成されます。この if 文は、「isTypeOf」メソッドを使用して、受け取ったシグナル イベントの動的な型が遷移のトリガとして指定された静的イベント型に一致することをテストします。一致した場合、以下のように処理されます。

1. イベントの実引数がある場合は、その値が参照属性に代入されます。Java では、イベント パラメータにアクセスするには、ジェネリック Event（イベント）型から特定のシグナル イベント型へのキャストが必要です。
2. 「leave」メソッドを呼び出すことによって、現在の状態から離れます。
3. 遷移メソッドが呼び出されます。
4. 最後に、「execute」メソッドは「Dispatchable.EventAction.Consumed」返して、イベントが消費されたことを示します。

例 **459:** トリガ付き遷移の翻訳 ——————————————————————————

UML

```
active class C {
    private long count;
    protected String strName;
    public C destination;

    statemachine initialize {
        String 'from';
        start {
            count = 0;
            ^ destination.sig(strName);
            nextstate Idle;
        }

        state Idle;
        for state Idle;
            input sig('from') {
```

```
                    count = count + 1;
                    ^ destination.sig('from');
                    nextstate Idle;
                }
            }
        }
    }
```

Java

```
    public class C extends DispatchableClass
    {
        //… 上記の アクティブ クラス の翻訳を参照 …

        public long count;
        public String strName;
        public C destination;
    }

    public class C_initialize extends StateMachine
    {
        // … 上記の 状態機械（State machine）の翻訳を参照 …

        protected String from;

        public void initialTransition()
        {
            m_owner.count = 0;
            sendTo(new sig(m_owner.strName),
    m_owner.destination);
               m_s_Idle.enter();
        }

        public void init()
        {
               // …
            m_s_Idle = new Idle(theTopRegion, this);
               m_s_Idle.init();
        }

        public static class Idle extends State
        {
            public C_initialize m_owner;

            public Idle(Region region, C_initialize owner)
            {
             super(region);
             m_owner = owner;
            }

            public Dispatchable.EventAction execute(Event e)
            {
             if (sig.isTypeOf(e))
                 {
                     m_owner.from = ((sig) e).sender;
                     leave();
                     m_owner.trans_Idle_sig(e);
                     return Dispatchable.EventAction.Consumed;
                 }
             return Dispatchable.EventAction.NoMatch;
```

```
        }
    }

    protected trans_Idle_sig(Event e)
    {
        m_owner.count = m_owner.count + 1;
        sendTo(new sig(from), m_owner.destination);
        m_s_Idle.enter();
    }

    protected Idle m_s_Idle;
}
```

簡単にするため、この例ではトリガ付き遷移の翻訳と関係のないものを、ほとんど省略しています。

この例の、状態機械属性「from」およびアクティブ クラス属性「count」、「strName」、「destination」の 可視性（定義の可視性参照）は、Java ではすべて public です。

遷移をトリガしたイベントは、自動生成された「遷移メソッド」のパラメータとして取得できます。このパラメータは、ターゲットの Java コードを使用することで、UML 遷移のアクションでアクセスできます。

注記

受信したシグナルの実引数は、UML シグナル送信アクションで参照される変数に代入されます。

複数トリガ

トリガ付き遷移が複数のトリガを持つ場合、対応する Java の if 文は、参照先の任意のイベントの受信時に true となる式を使用します。

受信したシグナルがパラメータを持つ場合、属性への実値の代入は if 文内で行い、実イベント型と予期していたシグナルが一致するようにします。

複数トリガの特殊なケースとして「アスタリスク入力」があります。これは、状態で受信可能なすべてのシグナルを表します。アスタリスク トリガを持つトリガ付き遷移の Java の翻訳は、単にイベントがヌルでないことを調べる if 文となります。

例 **460:** 複数のトリガを持つトリガ付き遷移の翻訳 ——————————

UML

```
state Idle;
for state Idle;
input sig1(a), sig2(b) {
    nextstate Idle;
}
input * {
   stop;
}
```

Java

```
public static class Idle extends State
```

```
{
    // … 上記の 状態 の翻訳を参照

    public Dispatchable.EventAction execute(Event e)
    {
        if (sig1.isTypeOf(e) || sig2.isTypeOf(e))
        {
            if (sig1.isTypeOf(e))
                m_owner.a = ((sig1) e).par0; // Assuming
unnamed param.
            if (sig2.isTypeOf(e))
                m_owner.b = ((sig2) e).par0; // Assuming
unnamed param.
            leave();
            m_owner.trans_Idle_sig1_sig2(e);
            return Dispatchable.EventAction.Consumed;
        }
        if (e)
        {
            leave();

m_owner.trans_Idle__GEN_68y_42UVUZ_42PLLeZR70IzmiZ8I(e);
            return Dispatchable.EventAction.Consumed;
        }
        return Dispatchable.EventAction.NoMatch;
    }

    protected trans_Idle_sig1_sig2(Event e)
    {
        m_s_Idle.enter();
    }

    protected
trans_Idle__GEN_68y_42UVUZ_42PLLeZR70IzmiZ8I(Event e)
    {
        finish();
    }
}
```

この場合、実シグナル引数を状態機械変数に移すコードは、イベント タイプの一致を調べる if 文内に入れられます。

遷移をトリガするすべてのイベントの名前で構成される遷移メソッドの名前付けに注意してください。アスタリスク入力の場合、遷移の GUID はその名前の一部となります。

ガード

トリガ付き遷移のガードは、「execute」メソッドでの状態遷移のために、if 文内の追加式に翻訳されます。この式は、トリガする遷移に対して true になる必要があります。

遷移がガードのみを持ち、トリガがない場合、イベントを受信せずに遷移をトリガする必要があります。したがって、この場合、遷移の if 文は、イベントがヌルであり、ガード条件を満たしていることを調べます。

例 **461:** 遷移のガード条件の翻訳

UML

```
state Idle;
for state Idle;
input sig1()[x == y] {...}  // Trigger and guard
[b == true] {...}           // Only a guard
```

Java

```
public static class Idle extends State
{
    // … 上記の 状態 の翻訳を参照

    public Dispatchable.EventAction execute(Event e)
    {
        if (sig1.isTypeOf(e) && (x == y))
        {
            ...
        }
        if (e == null && (b == true))
        {
            ...
        }
        return Dispatchable.EventAction.NoMatch;
    }
}
```

ラベル遷移

UML のラベル遷移は、ラベル遷移を含む状態機械実装の翻訳である Java の状態機械クラスの保護されたメソッドに翻訳されます。

「遷移メソッド」の名前は「trans_L」です。L は、ラベル遷移のラベルの名前です。

例 **462:** ラベル遷移の翻訳

UML

```
statemachine initialize {
    Lbl:
    {
        count = 1;
    }
}
```

Java

```
public class C_initialize extends StateMachine
{
    // … 上記の 状態機械（State machine） の翻訳を参照 …

    protected void trans_Lbl()
    {
        count = 1;
    }
```

```
}
```

接続ポイント

UML の接続ポイントは、接続ポイントが定義されている状態機械の翻訳である Java のクラスの属性に翻訳されます。

属性の名前は接続ポイントの名前であり、その型は以下のとおりです。

- TOR クラス EntryPoint（入場点）（接続ポイントがエントリ接続ポイントの場合）。
- TOR クラス ExitPoint（退場点）（接続ポイントが終了接続ポイントの場合）。

属性は、その型クラスの新しいインスタンスに初期化されます。

エントリ接続ポイントは状態機械のエントリ ポイント遷移から参照され、終了接続ポイントは状態機械を参照する合成状態の終了ポイント遷移から参照されます。これらの遷移は、以下の Java メソッドに翻訳されます。

- `entryPointTransitions()`（内側状態機械のクラス内）
- `exitPointTransitions()`（外側状態機械のクラス内）

これらのメソッドの実装は、接続ポイントを入力パラメータとして取得して、エントリ ポイントまたは終了ポイントの遷移ごとに 1 つの if 文を使用します。`exitPointTransitions()` の遷移メソッド（これは通常どおり翻訳される。トリガ付き遷移を参照）を呼び出す前に、現在の状態から移行するために、「theTopRegion」で「leave」メソッドを呼び出します。

例 **463:** 接続ポイントとエントリ／終了ポイント遷移の翻訳

UML

```
active class C : DispatchableClass {
    statemachine initialize {
    ...
        state Idle : statemachine initialize
            in Cin      // Entry connection point
            out Cout    // Exit connection point
        {
            start Cin {
                ...
            }

            start {
                ...
            }

            state WaitForSig;
            for state WaitForSig;
            input sig() {
                return Cout;
            }
        };

        for state Idle;
```

```
            [Cout] {
                count = 14;
                stop;
            }
        }
    }
```

Java

```
    public class C_initialize extends StateMachine
    {
        public static class Idle extends State
        {
            public C_initialize_Idle_initialize m_sm;

            ...
        }

        public Idle m_s_Idle;

        protected void trans_Idle_Cout()
        {
            m_owner.count = 14;
            finish();
        }

        public void exitPointTransitions(ExitPoint ep)
        {
            if (ep == m_s_Idle.m_sm.Cout)
            {
                theTopRegion.leave();
                trans_Idle_Cout();
            }
        }
    }

    class C_initialize_Idle_initialize extends StateMachine
    {
        EntryPoint Cin = new EntryPoint(this);
        ExitPoint Cout = new ExitPoint(this);

        protected void trans_Cin()
        {
            ...
        }

        public void initialTransition() { … }

        public void entryPointTransitions(EntryPoint ep)
        {
            if (ep == Cin)
            {
                trans_Cin();
            }
        }

        Idle m_ownerState;

        public static class WaitForSig extends State { ... }
    };
```

接続ポイントの翻訳に関係ない詳細は、例では省略されています。

接続ポイントによってトリガされる遷移の名前には、シグナル名の代わりに、接続ポイントの名前が使われます。

翻訳のカスタマイズ

Java コード ジェネレータの出力は、エージェントを使用してカスタマイズできます。この方法を使うと、生成されるコードを正確に制御できます。

生成されるコードのカスタマイズは、生成されるファイル内の特定の位置に任意のテキストを追加して行うことができます。これにより、生成される通常のコードとともに、追加のコード、コメントなどの生成が可能になります。

また、Java コード ジェネレータが直接処理できないモデル構成要素のカスタム変換や、Java コード ジェネレータによるデフォルトの翻訳の修正も可能です。

これらのすべてのカスタマイズ機能は、コード生成時に、これらのツール イベントによってトリガされるツール イベントとエージェントに基づいています。これらの概念の詳細については、エージェントを参照してください。

コード生成時にテキストを追加

コード生成時に、任意のテキストを追加できます。

- 生成後のファイルの先頭または最後（ツール イベント JavaPrintFile 参照）
- Java 定義の生成の直前または直後（ツール イベント JavaPrintDefinition 参照）

例 **464:** 生成される **Java** ファイルにカスタム ヘッダを出力

生成されるすべての Java ファイルに、コメント ブロックの形式で、カスタム ヘッダを表示するものとします。これは、<< 処理前 >> の JavaPrintFile ツール イベントでトリガされるエージェントを定義することによって行うことができます。たとえば、このエージェントの実装には以下の Tcl スクリプトを使用できます。

```
proc PrintHeader { triggeredBy timing context server
agentParameters } {
  upvar 1 $agentParameters ap
  set buffer [lindex $ap 0]
  u2::AddSourceBufferText $buffer "// This is a generated
file! Do not edit!\\n"
}
```

カスタム変換の実装

標準の Java 翻訳を利用できないモデル構成要素ついて Java コードを生成するために、ツール イベント Transformation を使用して、カスタム変換を実装できます。これは、標準の翻訳をカスタマイズする手段としても使用できます。

例 **465:** カスタム **Java** 変換の実装

Java コード生成時に、singleton デザイン パターンのサポートしたいとします。モデル レベルではこのデザイン パターンの実装を公開するつもりはなく、その代わり、<<singleton>> ステレオタイプでクラスをステレオタイプ化して、singleton クラスとして生成されることを示します。

この変換の実装は、<< 処理前 >> の Transformation ツール イベントでトリガされるエージェントを定義することによって行うことができます。たとえば、このエージェントの実装には以下の Tcl スクリプトを使用できます。

```
proc CustomTransformation { triggeredBy timing context server
agentParameters } {
  upvar 1 $agentParameters ap
  set messageList [lindex $ap 0]
  set roots [lindex $ap 2]
  set model [lindex $ap 3]
  foreach r $roots {
      if {[u2::HasAppliedStereotype $r "singleton"]} {
          u2::AddMessage $messageList "Transforming singleton
class" Information -subject $r

          ## Add attribute
          set type [u2::GetValue $r "Name"]
          set att [u2::Parse $model "private static part
$type m_inst;"]
          u2::SetEntity $r "OwnedMember" $att

          ## Add private constructor
          set constructor [u2::Parse $model "private void
$type ();"]
          u2::SetEntity $r "OwnedMember" $constructor

          ## Add instance method
          set method [u2::Parse $model "public static $type
instance(){return m_inst;}"]
          u2::SetEntity $r "OwnedMember" $method
      }
  }
}
```

このエージェントを使用して、新しい翻訳ルールを実装したことになります。その結果の例をここに示します。

UML

```
<<singleton>> class C{}
```

Java

```
public class C
{
    private static C m_inst = new C();

    private C() {}

    public static C instance()
    {
        return m_C;
    }
}
```

コード ジェネレータの通常の翻訳が実行される前にエージェントがトリガされるので、変換結果内で UML 構成要素を使用できます。たとえば、作成された属性は UML パートです。このパートは、集約（Aggregation）種別についてで説明した翻訳ルールに従って、「新しい」イニシャライザによって属性に翻訳されます。

35

Java ランタイムフレームワーク

この章では、Java ランタイム フレームワークの Tau Object Runtime（TOR）について説明します。フレームワーク内のすべてのクラスの目的とランタイム セマンティックについて説明します。

概要

この章は、Java で実装されたオブジェクト指向 UML ランタイム フレームワークである Tau オブジェクトランタイム (TOR) のリファレンス ガイドです。TOR は、構造と振る舞いの両面で、UML のランタイム セマンティックを実装します。

TOR は、明確に指定されたサービスを提供する一連のクラスとして実装されています。一部のクラスはフレームワーク自体の中で使用されます。その他のクラスは Java コードジェネレータで生成されたコードで使用され、モデルを実行可能な Java コードに変換します。

一部のクラスはモデリング時に使用されます。これらの “TOR クラス ” は TOR UML モデルとして使用できます。

フレームワークは、一連の Java ソースファイルとして提供されます。ファイルのリストは、ファイルのリスト のセクションを参照してください。

注記

TOR には C++ 実装もあります。この実装は、C++ アプリケーションジェネレータが生成するコードによって使用されています。C++ TOR のデザインと Java TOR のデザインはよく似ているので、Java と C++ の両方をターゲットとする UML モデルの作成が容易になります。TOR の C++ 実装については、C++ ランタイム フレームワークを参照してください。

TOR パッケージ

TOR のすべての定義は、com.telelogic.tau.tor Java パッケージにあります。このパッケージには、関連クラスをグループ化した複数のサブパッケージがあります。

TOR UML モデル

TOR の一部は、Java コードジェネレータがアクティブになると自動的にロードされるモデル ライブラリ「tor」として使用できます。このモデルには、ユーザー モデルで使用できるすべてのタイプ、クラスが、含まれます

このモデルのクラスについては、TOR クラス のセクションで説明しています。

注記

Java コード生成で使用される TOR UML モデルは、C++ コード生成で使用されるものと同一です。ただし、モデルの一部は Java TOR 用には不要なので、Java コードジェネレータではサポートされていません。

TOR のビルド

Tau が生成する Java コードをコンパイル、デプロイするために Eclipse インテグレーション を使用すると、TOR は通常は自動的にビルドされます。生成コードが TOR 構築子に依存していない場合は、TOR はビルドされません。

生成された Java コードを他の方法でビルドしている場合は、インストール済みの Tau の以下の場所にある TOR の Java ソースファイルを参照してください。

addins¥JavaApplication¥Etc¥TOR¥com¥telelogic¥tau¥tor

Java 1.5 またはそれ以降の Jav SDK が必要なことに注意してください。

便宜のためソースファイルは同じディレクトリにある tor.jar になっています。

TOR クラス

このセクションでは、TOR で定義されるクラスをアルファベット順に示します。各クラスの目的とランタイム セマンティックについて説明します。すべてのクラスは TOR パッケージ（またはそのサブパッケージ）で宣言されます。

- CompletedEvent
- Connector
- Dispatchable
- DispatchableClass
- Dispatcher
- DispatcherBehavior
- DispatcherData
- EntryPoint（入場点）
- Event（イベント）
- EventExecutor
- EventQueue
- EventReceiver
- ExitPoint（退場点）
- InstanceManager
- InternalEvent
- Port（ポート）
- Region（領域）
- RunInitialTransition
- State（状態）
- StateMachine（状態機械）
- Synthesized
- ThreadedDispatcher
- ThreadSafeEventQueue
- TimerEvent
- TimerObject
- TimerQueue
- TopRegion

CompletedEvent

ある State（状態）が終了したときフレームワークが生成する内部 Event（イベント）です。このイベントは、状態に渡され直ちに処理されます。このイベントは、トリガ イベントなしで任意の遷移をトリガするために使われます。こういった遷移は 「トリガーフリー遷移」と呼ばれます。

ガードのみがある遷移に使われるヌル イベントもありますが、これとは異なり CompletedEvent は、ガードなしの遷移、つまり、イベントもガードもない遷移をトリガします。

Connector

このクラスは、UML コネクタを現しており、モデル内で各コネクタごとに 1 回インスタンス化されます。インスタンスには、'connect' メソッドがあり 2 つのポートをコネクタで接続します。この仕組みによって接続されたポート間でのイベントの送信が可能になります。

Dispatchable

イベントの受信と実行の両方が可能なエンティティを現す抽象クラスです。イベントの受信は、EventReceiver インターフェイスの実装 (実現) で表現されます。イベントの実行は、EventExecutor インターフェイスの実装 (実現) で表現されます。

1 つの dispathcable は 1 つの Dispatcher と 1 つの EventQueue に関連付けられます。dispatcher が 1 つの Event（イベント）をキューから取り出して処理するときに、dispatcher は dispatchable に、このイベントをどのように処理するのかをたずねます。この結果、dispatchable はイベントの取り扱い方法を示した EventExecutor.EventAction を返します。

DispatchableClass

dispatchable クラスは、1 つ UML アクティブクラスを現します。このクラスは Dispatchable を継承して、自分に割り振られたイベントを受信して実行する能力を取得しています。

1 つの dispatchable クラスは、1 つの StateMachine（状態機械）と関連付けられている場合があります。この状態機械は、classifier behavior 状態機械と呼ばれます。

dispatchable クラスのあるインスタンスに送られる Event（イベント）は、そのインスタンスの StateMachine（状態機械）に渡されます。

クラスが「アクティブ」と指定されると、`tor::DispatchableClass` の継承は、自動的にモデルに追加されます。この仕組みによって、UML モデルで DispatchableClass の重要な操作をアクセスできるようになります。

dispatchable クラスのインスタンス化

dispatchable クラスのインスタンスは、作成されても、その振る舞いの実行を自動的には開始しません。まず初期化を行う必要があり、その後開始できるようになります。

初期化の中で、dispatchable クラスの classifier behavior 状態機械は初期化されて、実行の準備が行われます。初期化処理は、通常は、Java コードジェネレータによって main メソッドに生成されたコードによって自動的に行われます。dispatchable クラスを手動で初期化するには、以下のように init 操作を呼び出します :

```
public void init();
```

インスタンスを開始すると、初期遷移を実行する要求（RunInitialTransition）をポストすることによって、classifier behavior 状態機械が開始されます。いったん開始されれば、その状態機械はイベントを受信する用意が整います。開始処理は、通常は、Java コードジェネレータによって main メソッドに生成されたコードによって自動的に行われます。状態機械の呼び出しを手動で行うには、以下のように `start` 操作を呼び出します :

```
public void start();
```

すでに開始しているインスタンスを開始しても何も起こりません。

注記

「start」は UML テキスト構文の予約語であることに注意してください。したがって、上記の関数を参照する場合は、次のように単一引用符で囲む必要があります（`'Start'`）。

Dispatcher

dispatcher は EventQueue と関連付けられており、キューに入れられた Event（イベント）を処理します。イベントは、キューが空でない限り、キューから取り出されて 1 つずつ処理されるか、連続的に処理されます。

dispatcher は、現在アクティブなタイマーに対応する TimerObject が配置されている TimerQueue とも関連付けられています。

dispatcher を Dispatchable と対応付け、その後 dispatcher を開始するコードは、通常は Java コードジェネレータによって main メソッド内に自動的に生成されますが、以下に示すように手動で行うこともできます。

dispatchable を dispatcher に追加するには add 操作を呼び出します :

```
public void add(Dispatchable);
```

dispatchable を dispatcher から削除するには remove 操作を呼び出します :

```
public boolean remove(Dispatchable);
```

dispatcher のイベント キューからイベントの処理を開始するには、`run` 操作を呼び出します。この操作は、キューが空になるまでイベントをイベント キューから 1 つずつ取り出し、その後リターンします。

```
public void run();
```

キューからイベントを 1 つずつ処理するには step 操作を呼び出します。この操作は、キューにある次のイベントを取り出して処理します。

```
public void step();
```

DispatcherBehavior

ThreadedDispatcher の振る舞いをあらわすクラスで、イベントがイベントキューに到達するのを待ち受けて、それらのイベントを実行するために dispatcher を使用するスレッドを実現しています。DispatcherBehavior は ThreadedDispatcher の静的ネストクラスとして定義されています。

DispatcherData

A class containing the data needed by a ThreadedDispatcher に必要なデータを保持する 1 つのコンテナクラスで、ThreadedDispatcher の静的ネストクラスとして定義されます。データは DispatcherData にカプセル化されるので、ThreadedDispatcher によって起動されたスレッドと呼び出し側スレッドの両方から安全にアクセスできます。.

EntryPoint（入場点）

入場点は、合成 State（状態）または StateMachine（状態機械）の 1 つの Region（領域）に入場するために使用される名前付きの擬似状態です。これは、内部（たとえば実際のターゲット状態など）について何も公開しないで、合成状態やサブ状態機械に入場する方法を提供します。

入場点に到達すると、つまり、入場点をターゲットとする遷移が実行されると、その入場点を所有する領域に入り、入場点の出力遷移が実行されます。

1 つの入場点は任意の数の入力遷移を持つことができますが、出力遷移はただ 1 つのしか持てません。

Event（イベント）

Event クラスは、あるシステム内のあらゆる種類のイベントのタイプとオカレンスを現すために使用します。シグナル、非同期操作呼び出し、タイマー タイムアウトなどがその例です。Event のサブクラスがイベントのタイプを指定し、このクラスのインスタンスがイベントオカレンスを現します。

すべてのイベントには、特定のオブジェクト id によって現される受信者があります。InstanceManager は、そのオブジェクト id をイベント インスタンスの実際の受信者である Dispatchable への ポインタにマッピングする責任を負います。受信者はフレームワークが内部的に設定します。

EventExecutor

イベントを実行する能力を持つ抽象クラスです。このインターフェイスには、EventExecutor を実装するクラスで実装する必要のある、以下のメソッドがあります：

```
public EventAction execute(Event（イベント） event);
```

このメソッドの実装は、EventExecutor.EventAction 列挙の適切なリテラル値を返す必要があります。これは、EventExecutor がどのようにイベントを取り扱ったかを示すためです。

イベントは実行される前に受信されなければならない、ということに注意してください。つまり、通常は、EventExecutor を実装するクラスは EventReceiver をも実装することになります。

注記

イベントの受信と実行を明確に区別することが重要です。イベントが**受信**されるのは、フレームワークによって受信者に配信されたときです。イベントが**実行**されるのは、イベントの受信に関連付けられている振る舞い（通常は遷移）が実行されたときです。イベント受信はイベント実行に先行し、通常、受信と実行の間には一定の時間間隔があります。

EventExecutor.EventAction

受信者によってイベントがどのように取り扱われるかを示す列挙です。リテラル値とその意味は以下の表のとおりです。

リテラル	説明
`NoMatch`	受信者はイベントを取り扱わなかった。
`Defer`	受信者がイベントを保存した。
`Consumed`	受信者はイベントを消費した。

EventQueue

イベント キューは、Java の LinkedList によって実装された Event（イベント）の待ち行列です。

フレームワークは、イベントの挿入と削除など、イベント キューに関するあらゆる事象を自動で取り扱います。

EventReceiver

イベントを受信する能力を持つインターフェイスです。このインターフェイスには、EventReceiver を実装するクラスで実装する必要のある、以下のメソッドがあります :

```
public boolean receive(Event（イベント） e);
```

このメソッドの実装は、イベントが受信されれば true を返し、受信されなければ false を返す必要があります。

EventReceiver は、インターフェイスとして TOR UML プロファイルにあります。この仕組みにより、ユーザー独自のイベント受信者を宣言できます。sendTo ユーティリティ機能を使用して、イベントをイベント受信側へ送ることができます。一般的な使い方の 1 つとして考えられるのは、シグナルを受信する必要があるパッシブ クラスがある場合です。これは、クラスに EventReceiver インターフェイスを実装させて、`receive` メソッドを実装することで可能になります。

イベントの受信と実行の違いについては、EventExecutor の注記を参照してください。

ExitPoint（退場点）

退場点は、合成 State（状態）または StateMachine（状態機械）の Region（領域）から離れるために使用する名前付きの擬似状態です。退場点は、合成状態または下位状態機械から、それらをインスタンス化した際のコンテキスト（たとえば出力遷移のターゲット状態など）を知らずに離れる方法を提供します。

退場点に達すると、つまり、ターゲットとして退場点を持つ遷移が実行されると、その退場点を所有する領域を離れ、退場点の出力遷移が実行されます。

退場点は複数の入力遷移を持つことができますが、出力遷移は 1 つしか持つことができません。

InstanceManager

このクラスは、オブジェクト id と EventExecutor および EventReceiver の間のマッピングを維持する責任を負います。そのマッピングは、Dispatchable が作成されるタイミングとガーベジコレクタによって削除されるタイミングで最新状態に維持されます。インスタンスマネージャの主な用途は、インスタンスへの直接のポインタではなく、あるインスタンスのシンボリックな表現 (オブジェクト id) を得ることです。この間接的な表現は、複数のインスタンスが複数のスレッドから互いに独立に生成、削除されるようなマルチスレッドのシステムにおいては重要です。また、複数のアドレススペースをまたがるようなシステムでも必要です。

InternalEvent

このクラスは、TOR フレームワークによって内部的に送信されるすべてのイベント (モデル内にあるユーザ定義のイベントとは直接関連のないすべてのイベント) にとっての共通の基盤クラスです。

Port（ポート）

このクラスは、UML ポートを現しておインスタンスには、'connect' メソッドがあり 2 つのポートをコネクタで接続します。つまり、シグナルがポートから流入、流出する場合には、このクラスの型の属性が 2 つ生成されます。

各ポートは、Connector で接続された他のポートのリストを保持しています。

Region（領域）

領域抽象クラスは、1 つの State（状態）または 1 つの StateMachine（状態機械）の直交領域を現します。ある領域は、ある 1 組の状態を所有します。領域は、現在の状態と以前の状態を記憶しています。現在の状態とは、その領域の現在アクティブな状態のことです。以前の状態とは、最後に領域を離れたときにアクティブだった状態のことです。

領域への入退場は、遷移の結果にしたがって行われます。

Region に入る

領域に入るには複数の方法があります。初めて領域に入るときには、初期遷移が実行されます。以降その領域に入る際には、履歴情報を使用して前の状態に再入できます。領域には、EntryPoint（入場点）から入ることもできます。

現在の状態と以前の状態は、遷移の結果として領域内のある状態に入ったときに、更新されます。

Region を離れる

領域を離れるのは、他の領域のターゲット状態に向けた遷移がトリガされたときか、ExitPoint（退場点）をターゲットとする遷移がトリガされたときです。

現在の状態と以前の状態は、領域内のある状態を離れると更新されます。

Region の終了

領域は、その領域のある最終状態に到達すると完了します。最終状態は、フレームワークでは明示的に定義されていません。

領域が終了すると、内包している状態のトリガーフリー遷移が評価されます。

RunInitialTransition

DispatchableClass が開始されたときに、その DispatchableClass に送信される内部イベント。このイベントが実行されると、DispatchableClass クラスの初期遷移が実行されます。

RunInitialTransition は、TopRegion の静的ネストクラスとして定義されています。

State（状態）

ある StateMachine（状態機械）内の 1 つの状態を現す抽象クラス。ユーザーモデル内で状態に対応する生成されたクラスは、このクラスを継承します。

1 つの状態を所有する Region（領域）は 1 つです。一方、1 つの状態は、複数の領域や 1 つの状態機械を所有できます。このような状態は、合成状態と呼ばれます。

遷移とイベント処理

1 つの状態には一連の入力遷移と出力遷移があります。遷移は、フレームワーク内の個別のクラスで現されるのではなく、遷移を含む状態機械内の操作として現されます。

状態は、イベントを受信すると、そのイベントによってトリガされる出力遷移があるかどうかを調べます。その出力遷移があり、ガード条件が真であると、遷移が実行されます。遷移には順序がなく、フレームワークが最初に見つけた該当の遷移が実行されます。

該当する遷移がなければ、イベントはこの状態の任意の親状態、つまり、他の状態か状態機械へ渡されます。

状態に入る

ある状態に入るということは、その状態をターゲットとする遷移が実行されたということです。状態に入ると、その状態の任意の入場アクションが実行されます。そして、その状態が合成状態の場合は、所有している領域または状態機械にも入ります。

ある状態に入ると、その所有者である領域の現在と前の状態が更新されます。

状態を離れる

ある状態を離れるということは、その状態の任意の出力遷移がトリガされたか、その状態の親状態の出力遷移がトリガされたということです。状態を離れると、その状態の任意の退場アクションが実行されます。

ある状態を離れると、その所有者である領域の現在と前の状態が更新されます。

StateMachine（状態機械）

1 つの状態機械とその実装を現す抽象クラス。ユーザーモデル内で状態機械に対応する生成されたクラスは、このクラスを継承します。

注記

1 つの状態機械は、最上位領域がリストされたものです。現行では、このリストには、常に、StateMachine の生成済みサブクラスで定義された、ただ 1 つの TopRegion が含まれています。

1 つの状態機械を、クラスの classifier behavior として 1 つの DispatchableClass に関連付けることができます。dispatchable クラスに送信された Event（イベント）は状態機械に渡されて、そこで処理されます。

1 つの State（状態）は、1 つの状態機械を所有できます。このような状態機械は、下位状態機械と呼ばれます。この状態に送信されたイベントは、下位状態機械へ渡されて処理されます。

状態機械がイベントを処理できるようにするには、その状態機械を初期化して開始する必要があります。この作業は、関連付けられた DispatchableClass または State（状態）に対して対応するアクションが行われたときに、フレームワークによって内部的に行われます。

Synthesized

Java コード生成時に合成（Synthezied）された定義にマーク付けするための使用される注釈表記です。生成 Java コードを変更して UML モデルを更新する際には、合成された定義については更新の対象になりません。

Synthesized 注釈は生成コードを読みやすくするためにも使用できます。UML モデルエンティティから生成されたコードの部分とコードジェネレータが追加した部分を明確に分けるからです。

ThreadedDispatcher

スレッド版の Dispatcher クラス。ThreadedDispatcher は、この Dispatcher に関連付けられたすべての Dispatchable にイベントを割り振ることができる 1 つのスレッドを実現します。この仕組みによって、モデルにおける柔軟なスレッド配置が可能になります。以下に例を示します :

- アクティブクラスのインスタンス 1 つに 1 つのスレッド
  この場合、各アクティブクラスはただ 1 つの ThreadedDispatcher に関連付けられます。モデリングの例で言うと、各アクティブクラスが 1 つの ThreadedDispatcher を 1 つのパートとして含むような形です。そのクラスのコンストラクタで、新たに作成されたインスタンスは ThreadedDispatcher に追加されて、起動されます。起動されたスレッドの生存期間は ThreadedDispatcher の生存期間と同じなので、アクティブクラスのインスタンスが削除されると、スレッドは終了します。
- アクティブクラスに 1 つのスレッド
  クラスに静的な ThreadedDispatcher 属性を持たせる形です。そのクラスのコンストラクタで、新たに作成されたインスタンスは ThreadedDispatcher に追加されます。
- 任意のインスタンスのセットに 1 つのスレッド
  あるシングルトンクラスに ThreadedDispatcher 属性を 1 つ持たせて、任意のインスタンスを割り振り対象として追加する形です。

1 つの ThreadedDispatcher は、起動されたスレッドと呼び出し側スレッドの両方からアクセスする必要のあるすべてのデータを保持する DispatcherData を 1 つ所有します。たとえば、DispatcherData は Dispatcher と ThreadSafeEventQueue を保持します。このようなデータを DispatcherData インスタンスにカプセル化することによって、データをスレッドセーフなやり方でアクセスできるようになります。

Dispatcher とイベントキュー は自動的にインスタンス化され、関連付けられます。Dispatcher はスレッドセーフなやり方でイベントをキューから取り出して処理します。

ThreadedDispatcher を呼び出す方法は以下のとおりです :

```
public void launch();
```

ThreadedDispatcher を停止する方法は以下のとおりです

```
public boolean endThread();
```

このメソッドは、ThreadedDispatcher を可能な限り早急に終了します。ただし、現在実行中のイベントからトリガされた振る舞いの中断はしません。このメソッドは同期的に処理されます。つまり、現在のイベントからトリガされた振る舞いが完了するまで、このメソッドの処理は待たされます。このメソッドのオーバーロード版では、タイムアウト値の指定ができます。この指定を使うと、スレッド終了まで長時間待たされずに済みます。スレッドが指定時間の経過後に終了していない場合は、このメソッドは false を返します。

ThreadSafeEventQueue

ThreadedDispatcher が使用する、EventQueue のスレッドセーフなサブクラス。. キュー上で実行されるすべての操作は遂次化され、1 つの Semaphore を使用してキューに到着する新しいイベントを待ち受けます。

TimerEvent

タイマーイベントは、タイマー時間切れのイベントを現す特殊な Event（イベント）です。タイマーイベントは TimerObject と関連付けられます。この TimerObject 上で set や reset のようなタイマーアクションが実行されます。

TimerObject

タイマー オブジェクトは Dispatchable 内での 1 つのタイマーの宣言を現します。タイマー オブジェクトは、タイマーのセットとリセットを行うメソッドと現在アクティブになっているかどうかを照会するメソッドを提供します。これらのメソッドの実装には、関連する TimerEvent と TimerQueue が使われており、タイマーのセマンティクスを実現しています。

タイマーオブジェクトは、タイマーがアクティブな場合は、タイムアウト値を保持しています。

TimerQueue

タイマーキューは、現在アクティブなタイマーに対応する TimerObject の優先度付き待ち行列です。タイマー オブジェクトは、そのタイムアウト時間の順に並んでいます。

各 Dispatcher には 1 つのタイマー キューがあり、この待ち行列で、dispatcher が管理している Dispatchable のアクティブタイマーに対応するタイマーオブジェクトが、管理されています。

TopRegion

Top region は、ある StateMachine（状態機械）によって所有される Region（領域）です。所有者である状態機械は、Top region から取得できます。

１つの Top region が完了して、他のすべての Top region も終了していれば、所有者である状態機械も終了します。

TopRegion は Region（領域）のサブクラスです。

ユーティリティ

TOR フレームワークと生成された Java コードが、内部的に頻繁に使用するユーティリティメソッドがあります。これらのメソッドは、TOR UML ライブラリとして定義されているので、ユーザーが使用することもできます。すべてのユーティリティは、Utilities 抽象クラスの静的メソッドとして宣言されています。

sendTo

受信者に Event（イベント）を送信するために使われるメソッドです。受信者は EventReceiver として指定できます。

```
public static boolean sendTo(Event e, EventReceiver r);
```

このメソッドには、イベントをポートに送信できるオーバーロードされたバージョンがあります。ランタイムがコネクタ構造から受信者を特定できるので受信者を明示指定する必要がない場合です。

```
public static boolean sendTo(Event e, Port p);
```

イベントは処理のために受信者に送信されます。イベントが受信者に受け取られたか、または失われたかを示すブール値が返されます。

いずれの場合も、イベント取り扱いの責任は受信側に渡ります。イベントは、sendTo 関数に渡された後は、アクセスも削除もできません。

setTimeUnit

TOR が使用する時間単位を設定するためのメソッドです。

```
public static void setTimeUnit(double seconds);
```

時間単位は秒の値で指定します。たとえば、時間単位を 1 ミリ秒に設定するには setTimeUnit(0.001) を呼び出します。

重要！

現時点では時間単位はアプリケーション全体で共有されています。タイマーがすでにアクティブになっている場合は別のスレッドから時間単位を変更しないでください。予期しない結果が生じる可能性があります。

オペレーティング システム抽象化レイヤ

TOR は、オペレーティングシステムが通常提供する機能とインターフェイスするため、別個の抽象化レイヤを設けています。Java プラットフォームの場合は、このレイヤを Java ライブラリの機能として利用できます。このセクションではこのレイヤで定義されているクラスを説明します。

OS レイヤのすべてのクラスは、TOR パッケージに含まれるサブパッケージであるパッケージ os に定義されています。パッケージの完全修飾名は `com.telelogic.tau.tor.os` です。

OS レイヤ内のクラスの一部は TOR UML モデルでも利用可能です。たとえば、セマフォのような TOR のスレッド同期プリミティブを利用したいケースは頻繁にあります。

Semaphore

セマフォは、複数のスレッドからアクセスされる資源を、順番にアクセスするために使用されます。セマフォは取得と開放ができます。セマフォの取得には 2 通りの方法があります。開放されるまで待機する方法と、待ち時間を決めて開放されるのを待って時間が経過した場合は先に進む方法のいずれかです。

Time

時間（絶対時間と相対時間）を表現するクラス。このクラスには、現在時刻の取得や時間値の足し算と引き算を行うなどの各種の関数があります。Java による実装は Date オブジェクトです。

Thread

`com.telelogic.tau.tor.os` パッケージのサブパッケージ。生成された Java アプリケーションを複数のスレッドで動作させるためのクラスとユーティリティを含みます。

Behaviour

1 つのスレッドは `java.lang.Thread` を継承する Behaviour クラスによって現されます。ThreadedDispatcher クラスは `Behaviour` サブクラス (DispatcherBehavior) を使用して、その内部で dispatcher を実行します。

`com.telelogic.tau.tor.os` パッケージと同じく、スレッド関連のユーティリティを含む静的な `Utilities` クラスがあります。

suspend

```
public static void suspend(Time timeout);
```

現在のスレッドを指定した時刻まで一時停止します。ここで ‘timeout’ は絶対時間の値です。

getCurrentThreadId

```
public static long getCurrentThreadId();
```

現在実行中のスレッドの ID を返します。

ファイルのリスト

次のセクションでは、Tau インストールの一部として提供される TOR のファイルを説明します。

Java ソースファイル

TOR ソースコードはすべて Tau インストールに含まれており、インストールディレクトリの以下のフォルダ内にあります:

```
addins¥JavaApplication¥Etc¥TOR¥com¥telelogic¥tau¥tor
```

これらのファイルは、TOR 依存の生成された Java アプリケーションをビルドする際に使用されます。TOR のビルドについては TOR のビルドを参照してください。

以下の表にすべてのファイルとファイルに含まれる TOR 宣言を示します。

ファイル	TOR 宣言
CompletedEvent.java	CompletedEvent
Dispatchable.java	Dispatchable
DispatchableClass.java	DispatchableClass
Dispatcher.java	Dispatcher
EntryPoint.java	EntryPoint（入場点）
Event.java	Event（イベント）
EventExecutor.java	EventExecutor
EventQueue.java	EventQueue
EventReceiver.java	EventReceiver
ExitPoint.java	ExitPoint（退場点）
InstanceManager.java	InstanceManager
InternalEvent.java	InternalEvent
Region.java	Region（領域）
State.java	State（状態）
StateMachine.java	StateMachine（状態機械）
ThreadedDispatcher.java	ThreadedDispatcher, DispatcherBehavior, DispatcherData
ThreadSafeEventQueue.java	ThreadSafeEventQueue
TimerEvent.java	TimerEvent
TimerObject.java	TimerObject

ファイル	TOR 宣言
TimerQueue.java	TimerQueue
TopRegion.java	TopRegion
Utilities.java	sendTo, setTimeUnit
os/Semaphore.java	Semaphore
os/Status.java	OS 関連メソッドの戻り値として使用する状態の列挙
os/Time.java	Time
os/thread/Behaviour.java	Behaviour
os/thread/Utilities.java	suspend, getCurrentThreadId

36

Eclipse インテグレーション

Tau Eclipse インテグレーションにより、UML ツール セットを拡張して、Java に対応させ、Eclipse IDE（統合開発環境）と Java IDE との統合を可能にします。Eclipse インテグレーションは、UML ツール セットと Eclipse に追加された複数のコマンドで構成され、Java 開発環境におけるシームレスなラウンド トリップ エンジニアリングを円滑に行います。

Eclipse プロジェクトを既存の UML プロジェクトから作成することも、Eclipse のプロジェクトを UML にインポートすることもできます。両方のツールにプロジェクトがあれば、同期をとることができます。この統合により、ツール間の移動も円滑になります。

この統合は Java サポート上に構築されます。

Eclipse インテグレーションのインストール

Eclipse インテグレーションのコンポーネント

Eclipse インテグレーションは 2 つのコンポーネントで構成されます。

- EclipseIntegration アドイン
- TauG2Integration Eclipse プラグイン

適切な統合を行うために、この 2 つのコンポーネントは正しくインストールされていなければなりません。EclipseIntegration アドインは、Tau インストーラで自動インストールされますが、Eclipse プラグインは自動インストールされません。

Eclipse インテグレーション プラグイン

TauG2Integration プラグインを Eclipse にインストールするには：

1. Eclipse が適切にインストールされ、動作していないことを確認します。
2. プラグイン インストール ファイルを検索します。

   ```
   C:¥Program Files¥IBM¥Rational¥TAU¥4.3¥integrations¥Eclipse
   com.telelogic.taug2integration_4.1.0.jar
   ```

3. `jar` ファイルを Eclipse のインストール先ディレクトリにある plugins フォルダにコピーします。
4. プラグインは、次回 Eclipse を開始したときには起動しています。

注記

EclipseIntegration アドインと **JavaApplication** アドインは同時に使用できるように設計されていません。同じコマンドもありますが、どちらかのアドインにしかないコマンドもあります。また、実装が若干異なるコマンドもあります。混乱を避けるために、プロジェクトで有効にするアドインは、どちらか 1 つのみとします。

Tau でのインテグレーションの有効化

Eclipse インテグレーションを使用するすべてのプロジェクトで EclipseIntegration アドインを有効にしなければなりません。有効にするには次の手順に従います。

1. ［ツール］メニューから［カスタマイズ］ダイアログを選択します。
2. ［アドイン］タブをクリックして、［EclipseIntegration］アドインにチェックを付けて選択します。
3. ［OK］をクリックします。

注記

アドインを有効にする場合、またはアドインが有効になっているプロジェクトをロードする場合、Java ランタイム ライブラリ 全体が 1 つのプロファイルとしてロードされます（Load rt.jar on startup オプションの設定に依存）。このプロファイルの容量が大きいため、操作に時間が掛かることもあります。プロファイルのローディング中は、Tau からの応答はありません。

Eclipse の操作

ワークフロー シナリオ

Eclipse インテグレーションは、主として 2 つの場面で使用されることを想定しています。2 つの場面とは、java コードを出発点とする場合と UML モデルを出発点とする場合です。

- UML を開始点とする場合 : Eclipse 内で New ウィザードを使用して UML モデルを作成します。(1252 ページの「Eclipse での Java プロジェクトの作成」) ここで UML モデリング構築子を使用して、モデルの分析と設計ができます。十分に設計が行われて安定したと判断した場合は、モデルから設計に対応する Java コードを生成し、Java コードとモデルの同期を維持しながら作業を続行します。(1252 ページの「Eclipse での Java プロジェクトの作成」)
- Java を開始点とする場合 : 既存の Eclipse Java プロジェクトから対応する UML モデルを作成してアプリケーションの静的構造を可視化、分析します。これは 1252 ページの「Tau プロジェクトの作成」で説明しています。作成された UML モデルと Java コードの同期については、モデルとコードの同期で説明しています。

最初の場合のバリエーションとして、UML モデルを Eclipse には保存しない方法もあります。この方法もサポートされます。

Eclipse での UML プロジェクトの作成

Eclipse で新規の UML プロジェクトを生成するには以下の手順を行います。

1. Eclipse で［File ］>［New］>［Project］を選択します。
2. Tau フォルダにある「UML Project」をプロジェクトタイプとして、選択して［Next］をクリックします。
3. ウィザードの次のページでは［Java Code Generation］または他の適切な UML プロジェクトタイプを選択します。

4. ウィザードの 3 番目のページでは、以下のようなプロジェクトの詳細を設定できます。ほとんどの場合、設定値を変更する必要はありません。最後に［Finish］を押します。
   - プロジェクトに追加する UML ファイルを選択する
   - UML ファイルに名前をつける
   - プロジェクトの位置を選択する
   - 最上位に名前のない UML パッケージを作成する

プロジェクトが作成されると、［Package Explorer］内の ttp ファイルをダブルクリックして UML の編集を始められます。

Eclipse での Java プロジェクトの作成

既存の UML モデルから Java コードとこれに対応する Eclipse プロジェクトを生成するには、以下の手順を行います。

1. Tau でプロジェクトを開き、このプロジェクトで EclipseIntegration アドインを有効にします。

注記

Java ランタイム ライブラリのロードには、数分かかる可能性があります。

2. ［Eclipse］メニューから［Eclipse ディレクトリの生成］を選択します。

このコマンドにより、プロジェクト全体から Java コードが生成され、Eclipse が開いて接続し、対応する Eclipse プロジェクトが作成され、プロジェクト間のリンクが設定されます。これで、プロジェクトがシームレスなラウンド トリップ エンジニアリングを行うように設定されます。使用可能な Eclipse 関連コマンドについては、1254 ページの「Tau から Eclipse へ」を参照してください。

Eclipse が動作している場合、UML ツール セットから［ソースコードの更新］コマンドを実行するたびに、Eclipse プロジェクトとの同期がとられます。［Eclipse との同期］コマンドによってソース コードを更新できますが、Eclipse が稼動していない場合は対応する Eclipse プロジェクトと一緒に Eclipse を起動しようとします。同期オプションの詳細については、モデルとコードの同期を参照してください。

モデル要素から Eclipse の Java ソース コードに移動するには、［Eclipse 内を検索］コマンドを使用します。

Tau プロジェクトの作成

Eclipse の Java コードから UML モデルを生成するには、以下の手順を行います。

1. Eclipse のプロジェクトを開いて、［Package Explorer］を選択します。
2. ［Tau ］メニューから［Create corresponding Tau project］を選択します。表示されるダイアログで、必要に応じて Tau プロジェクトの場所を選択できます (デフォルトの位置は Eclipse ワークスペースのプロジェクトディレクトリです)。作成した UML から自動でダイアグラムを生成することもできます。

このコマンドにより、Tau が起動し、新しい空のプロジェクトが作成され、Java ソース コードから UML モデルへのリバース エンジニアリングが行われます。また、プロジェクト間のリンクが設定されます。

これで、プロジェクトがシームレスなラウンド トリップ エンジニアリングを行うように設定されます。使用可能な Tau 関連コマンドについては、1256 ページの「Eclipse から Tau へ」を参照してください。

注記

Java ランタイム ライブラリが Tau にロードされるため、新しい Tau プロジェクトのロードには数分かかることがあります。

Eclipse の Java コードの変更を同期させるには、［Synchronize with Tau］コマンドを使用します。新たに作成されたクラスとファイルがすべて検出されて UML モデルに追加され、変更箇所はマージされます。同期オプションの詳細については、モデルとコードの同期を参照してください。

Java ソース コードから対応するモデル要素に移動するには、［Locate in Tau］コマンドを使用します。

モデルとコードの同期

java ソースコードと UML は同期させておくことができます。UML モデルの java 設定を変更することで、どのタイミングで同期を行うかを制御できます。詳細については モデルとソース コードの同期を参照してください。

Tau から Eclipse へ

通信

Eclipse と通信を行うコマンドは、EclipseIntegration アドインに Eclipse 実行形式ファイルの場所が格納されている場合のみ使用できます。コマンドを最初に実行するとき、Eclipse 実行形式ファイルの場所を指定するよう指示されます。この情報は Eclipse オプションに Eclipse location として保存されます。この場所は、［Eclipse ディレクトリの変更］コマンドで変更できます。

注記

Tau と Eclipse の間に通信が確立するときに、一時ファイルの生成が必要になる場合があります。Unix 環境では、TEMP 環境変数を正しく設定してください。

Eclipse との接続

Tau はソケットを介して Eclipse と通信します。この際、ほとんどのコマンドは Eclipse が実行されていないと実行できません。Eclipse が実行していない場合、Tau は自動的に Eclipse を起動しようとします。起動に失敗すると、タイムアウト後にダイアログが表示され、接続の再試行を選択するか、コマンド実行を中止できます。

ヒント

接続に失敗した場合、TauG2Integration プラグインが正しくインストールされているかどうか確認してください。

Tau のコマンド

JAR ファイルのインポート

インポートウィザードを使って JAR ファイルを UML モデルにインポートできます。詳細は JAR ファイルのインポートを参照してください。

エクスポート

Java ファイルは、最初にどのように作成されていても、UML ツール セットの UML モデルから生成できます。パッケージを Java ソース コードにエクスポートするとき、Java ファイルが生成されます。［Eclipse］メニューの［Export］を選択してサブメニューの［Package］をクリックします。

モデルの更新

Java ファイルのエクスポート後、［モデルの更新］コマンドを使用してモデルと同期をとることができます。プロジェクトが Eclipse プロジェクトに接続していれば、［Eclipse 内を検索］コマンドを実行して、Eclipse の `.java` ファイルを開くことができます。

このコマンドによって java ファイルの変更に基づいてインクリメンタルなモデル更新を行えます。

モデルの強制更新

このコマンドを使用するとすべての java ファイルが再読み込みされ、必要に応じてモデルが更新されます。

ソースコードの更新

Java ファイルのエクスポート後、［ソースコードの更新］コマンドを使用してソースコードの同期をとることができます。プロジェクトが Eclipse プロジェクトに接続していれば、［Eclipse 内を検索］コマンドを実行して、Eclipse の `.java` ファイルを開くことができます。

- ［Eclipse］メニューから［ソースコードの更新］を選択します。

これでモデル内にあるすべての Java パッケージの Java ファイルが更新されます。

ソースコードの強制更新

このコマンドを使用するとすべての java ファイルが再読み込みされ、ソースコードが更新されます。

Eclipse ディレクトリの変更

Tau は Eclipse 実行形式ファイルを実行して、Eclipse に接続します。実行形式ファイルが移動されたり、Eclipse が複数インストールされている場合、以下の方法で必要な実行形式ファイルを選択できます。

- ［Eclipse］メニューから［Eclipse ディレクトリの変更］を選択します。

Eclipse ディレクトリの生成

既存の UML モデルから Eclipse プロジェクトを作成するには、以下の手順を行います。

1. ［Eclipse］メニューから［Eclipse プロジェクトの生成］を選択します。

これで、Eclipse が起動して、Tau のアクティブ プロジェクトと同じ名前のプロジェクトが作成されます。プロジェクトが作成される場所は、現在の Eclipse ワークスペースの場所に依存します。

Java ソース コードは、アクティブ プロジェクトのすべてのパッケージに生成されます。Eclipse のデフォルトで設定されたソース コードの場所が使用されます（デフォルトでは Eclipse プロジェクトと同じフォルダ）。Java 構文もすべてのパッケージで有効です。

ソース コードの場所は［同期ターゲットディレクトリ］のリストに追加されます。これで、ソース コードからモデルを更新したとき、ファイル システムの変更が必ず自動でモデルに反映されます。

Eclipse との同期

Eclipse にエクスポートされたプロジェクト、または Eclipse で作成されたプロジェクトと同期をとるには、以下の手順を行います。

- ［Eclipse］メニューから［Eclipse との同期］を選択します。

これで、モデル内にあるすべての Java パッケージの Java ファイルとこれに対応する Eclipse プロジェクトを更新します。

このコマンドは、Tau プロジェクトと Eclipse プロジェクトが接続されている場合のみ、使用できます。Eclipse インテグレーションのコンポーネントを参照してください。

Eclipse 内を検索

Eclipse の対応するソース コードでモデル要素を検索するには、以下の手順を行います。

1. ［モデル ビュー］またはダイアグラムで 1 つの要素を選択します。
2. ［Eclipse］メニューから［Eclipse 内を検索］を選択します。

これで、Eclipse の要素が選択されます。要素がパッケージの場合、このパッケージは［Package Explorer］で選択します。この方法をとらないと、［Package Explore］と［Outline View］のほか、対応する Java ソース ファイルが開いてエディタ内でも要素が選択されてしまいます。

Use Java Syntax

このコマンドを選択すると、Java 構文により、ツールの Java サポートが拡張されます。

Eclipse から Tau へ

どのコマンドを使用するにも、Eclipse ワークスペースに Tau インストールを保存しておく必要があります。コマンドを最初に実行するとき、Tau インストールの場所を指定するよう指示されます。この情報は Eclipse に保存されるため、再度指定する必要がありません。何らかの理由で変更する必要がある場合は、［Set Tau Location］を実行して変更できます。

注記

Tau と Eclipse の間に通信が確立するときに、一時ファイルの生成が必要になる場合があります。Unix 環境では、TEMP 環境変数を正しく設定してください。

Eclipse のコマンド リスト

Set Tau Location

Tau インストールの場所を設定または変更するには、以下の手順を行います。

1. ［Tau］メニューから［Set Tau Location...］を選択します。
2. Tau インストール フォルダを選択します。通常は以下のフォルダです。

   ```
   C:¥Program Files¥IBM¥Rational¥TAU¥4.3¥
   ```

3. ［OK］をクリックします。

Create corresponding Tau project

Eclipse のプロジェクトから Tau プロジェクトを作成するには、以下の手順を行います。

1. ［Tau］メニューから［Create corresponding Tau project...］を選択します。
2. Tau プロジェクト ファイルのフォルダを選択します。
3. ［OK］をクリックします。

新しいプロジェクトが作成され、Tau にロードされます。`.u2` ファイルが 1 つ作成され、Eclipse プロジェクトのすべてのソース ファイルがモデルと同期されます。

ソース コードのロケーションは Tau の［同期ターゲットディレクトリ］のリストに追加されます。これで、ソース コードからモデルを更新したとき、ファイル システムの変更が必ず自動的にモデルに反映されるようになります。

Synchronize with Tau

Tau にエクスポートされたプロジェクト、または Tau で作成されて Eclipse にエクスポートされたプロジェクトを同期させるには、以下の手順を行います。

1. ［Tau］メニューから［Synchronize with Tau］を選択します。

これで、プロジェクトのすべての Java ファイルに対応するモデルが更新されます。Eclipse で追加されたクラスや Java ファイルなどの新しいパッケージは、モデルに追加されます。既存の要素に対する変更が検出され、モデルにマージされます。削除された要素は、モデルから削除されます。

ファイルからモデルへのマッピングについては、1150 ページの「モデルとソース コードの同期」を参照してください。

Locate in Tau

Eclipse から Tau のモデル要素を検索するには、以下の手順を行います。

1. ［Package Explorer］、［Outline View］または、Java エディタで要素を選択します。
2. ［Tau］メニューから［Locate in Tau］を選択します。

これで Tau が起動し、モデル ナビゲータによって対応するモデル要素に移動します。このモデル要素に対応するプレゼンテーション要素が 1 つだけある場合、このプレゼンテーション要素を含むダイアグラムが表示されます。

Eclipse オプション

Eclipse 関連のオプションは以下のファイルにテキスト形式で保存されます。

```
eclipseOptions.ini
```

通常、`EclipseIntegration` アドイン フォルダにあります。

```
C:¥Program
Files¥IBM¥Rational¥TAU¥4.3¥¥addins¥EclipseIntegration
```

このファイルは、Eclipse インテグレーションによる最初のオプション設定時に必要に応じて作成されます。オプションごとに、キーと値（key/value）のペアが「=」記号で区切って保存されます。オプションは、以下の例のように、復帰改行文字で区切られます。

```
key1=value1
key2=value2
```

オプションを変更するには、テキスト エディタでファイルを開いて、値を変更します。キーがまだない場合、キーと値で復帰改行文字を追加します。

注記

手作業で編集する前に、必ず、オプション ファイルのバックアップ コピーをとります。

現状では 2 つのオプションがあります。

- Eclipse location
- Platform options

Eclipse location

Eclipse 実行形式ファイルの場所を指定します。このオプションは、Tau から最初に Eclipse を起動したとき、設定されます。

通常、`ExecutablePath` キーの下に保存され、Eclipse の実行形式ファイルを示します。

```
ExecutablePath=C:¥Program Files¥eclipse¥eclipse
```

Platform options

Eclipse の起動時に使用するプラットフォーム オプションを指定します。通常、Eclipse インテグレーションにより、Eclipse に渡されるプラットフォーム オプションはありません。デフォルト以外のワークスペースを使用する場合など、プラットフォーム オプションを指定して Eclipse を起動する場合、このオプションを設定する必要があります。

これは、以下の例のように、`PlatformOptions` キーに格納されています。

```
PlatformOptions=-data c:¥temp
```

この値は、変更されずにプラットフォーム オプションとして Eclipse に渡されます。したがって、Eclipse を手動で起動する場合、この値を指定する必要があります。

注記

PlatformOptions 値でパスを指定する場合は、パスにスラッシュを使用する必要があります。

UML と C#

「UML と C#」セクションの各章では、UML モデルから C# アプリケーションを生成する方法、および既存の C# コードを Tau にインポートする方法について説明します。

Tau での C# サポートは、Windows OS でのみサポートされます。

37

C# サポート

このセクションでは、Tau での C# サポートの使用方法について説明します。C# サポートは Tau の機能を拡張して、既存の C# アプリケーションを UML へインポートし、UML モデルからの C# アプリケーションを生成できるようにします。また、モデルまたはコードが変更された場合に、モデルとコードを同期させるためのサポートも含まれています。

ここでは、生成された C# アプリケーションのコンパイル、デバッグ、実行には、別の開発環境を使用することを想定しています。Tau には Visual Studio .NET とのインテグレーション機能もありますが（Visual Studio .NET インテグレーションを参照）、この章で説明する汎用の C# サポートはどの C# 開発環境でも使用できます。

C# サポートの使用

C# プロジェクトの作成

Tau の C# プロジェクトは、新規プロジェクトとして最初から作成するか、あるいは既存の Tau プロジェクトを C# プロジェクトに変換できます。

新規 C# プロジェクトを作成するには、以下の手順を行います。

1. ［ファイル］メニューから［新規］を選択します。
2. プロジェクトの作成時にプロジェクトのタイプとして［UML (C# コード生成用)］を選択します。
3. 必要な設定をすべて行ってウィザードを終了します。

プロジェクトの作成方法の詳細については、第 1 章「Tau 4.3 の紹介」の 20 ページ、「プロジェクトの操作」を参照してください。

既存のプロジェクト用に C# サポートを有効にするには、次の手順を行います。

1. ［ツール］メニューから［カスタマイズ］ダイアログを選択します。
2. ［アドイン］タブをクリックして、［CSharpApplication］アドインにチェックを付けて選択します。
3. ［閉じる］をクリックします。

C# プロジェクトの作成方法に関係なく、［モデル ビュー］のライブラリ フォルダに新しい C# 固有のライブラリが表示されます。また、メニューバーに新しい C# メニューが追加されます。

C# 固有のライブラリ

C# プロジェクトでは以下の C# 固有のライブラリを利用できます。

TTDCSharp

このライブラリには、C# サポートで使用されるすべてのステレオタイプが格納されています。ステレオタイプのタグ付き値は、C# ツールに対するさまざまなオプションとなります。また、このライブラリは、エージェントを使用して、API を C# 関連ツールに公開します。公開 API からこれらのエージェントを使用して、C# 関連コマンドをプログラムで呼び出すことができます。

TTDCSharpPredefined

このライブラリには、標準の UML 構成要素で表現できない C# 固有の言語構成要素を表現するために使用する、複数のステレオタイプが含まれています。

たとえば、標準の UML には部分クラスの概念は存在しません。クラスは、ただ 1 つの場所で完全に定義する必要があります。しかし、C# では、クラスは、`partial` キーワードを使用して複数の場所で部分定義できます。partial タイプを表現するために、TTDCSharpPredefined ライブラリには <<partial>> ステレオタイプが用意されています。

また、このライブラリには、string、int、bool などの内蔵 C# 型の UML 表現も含まれています。

TTDCSharpRuntime

このライブラリは、「mscorlib」として知られている Microsoft.NET のメイン アセンブリの UML 表現です。これは、「System」と「Microsoft」という名前の 2 つのライブラリ パッケージとして表示されます。このライブラリにはほとんどの C# プログラムで幅広く使用される多くの基本定義が含まれており、インポート済みライブラリとして利用できます。その他のアセンブリは、C# アプリケーションから使用したい場合は、すべて .NET アセンブリ インポータを使用してあらかじめインポートしておく必要があります。

C# メニュー

C# メニューには、UML モデルから C# コードを生成するためのコマンドがあります。また、これらのコマンドにより、UML モデルと生成された C# コードのどちらかに変更があった場合に、モデルとコードの同期を維持できます。

このメニューのすべてのコマンドは、ユーザー モデルに存在するすべてのものに対して（以下は［モデル ビュー］内のモデル フォルダ）、グローバルに動作します。これらのコマンドを選択的に使用するには、選択した要素のショートカット メニューから対応するコマンドを使用します。

モデルの更新

生成した C# ファイル内の変更を反映して、UML モデルを更新します。このコマンドは、UML モデルから C# ソース ファイルが生成されている場合にのみ意味があります（詳細については、C# コードの生成を参照してください）。UML モデルから生成された後で変更されたファイルのみが処理の対象となります。

このコマンドは、UML モデルから生成されていない既存の C# コードのインポートには使用できません。このようなコードのインポートには、既存の C# コードのインポートで説明されているように、C# インポータを使用します。

モデルの強制更新

このコマンドは［モデルの更新］とほとんど同じですが、最後に UML モデルから生成された後で変更されていない C# ソース ファイルも含め、生成されたすべての C# ソース ファイルが処理の対象となります。タイムスタンプを修正せずに C# ファイルが変更された場合、［モデルの更新］コマンドではこれらの C# ファイルが変更されたことを検出できません。このような場合には、［モデルの強制更新］コマンドが有用です。

ソースコードの更新

このコマンドは、UML モデルの変更を反映して、生成された C# ファイルを更新します。C# ファイルが生成されていない場合、C# コードの生成に説明されているように、C# ファイルが生成されます。このコマンドでは、C# ファイルが更新または生成された後で UML モデルに対して加えられた変更をベースとして、更新が必要なファイルのみが更新されます。

ソースコードの強制更新

このコマンドは、［ソースコードの更新］とほとんど同じですが、すべての C# ソースファイルが更新または生成処理の対象となります。このコマンドは、タイムスタンプを修正せずに、UML モデルの要素が変更された場合に有用です。Tau のモデル変更検出機能をカバーします。

C# コードの生成

C# ソース コードは、［ソースコードの更新］コマンドまたは［ソースコードの強制更新］コマンドを使用して UML モデルから生成されます。これらのコマンドは、モデルからファイルへのマッピング仕様に基づいて、C# ソース ファイルを生成します。

モデルからファイルへのマッピング

モデルからファイルへのマッピング仕様は、モデル要素を C# ソース ファイルにマッピングする方法を指定します。この仕様は、C# ファイル アーティファクト（生成する C# ファイルを指定）と、これらのアーティファクトからモデル内の定義への <<manifest>> 依存（各ファイルに生成する定義を指定）から構成されます。

モデルからファイルへのマッピング仕様はモデル内の別のパッケージに格納し、そのパッケージを独自のファイルに保存することを推奨します。これによって、C# アプリケーションの論理と動作を定義するモデル自身と、アプリケーションを構成する C# ファイルを指定するモデルからファイルへのマッピングとを、明確に分離できます。

上記の推奨に従った結果、UML モデルの構造は以下のようになります。

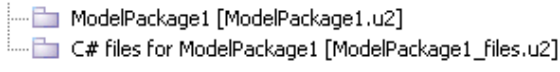


図 249: C# コード生成用の UML モデルの構造

UML モデルでは、最上位のモデル パッケージが複数存在することがあります。各最上位モデル パッケージは、アセンブリや実行形式ファイルなど、C# ソフトウェアのコンポーネントに対応します。最上位モデル パッケージごとに、そのモデル パッケージ用のモデルからファイルへのマッピング仕様を含む 1 つのパッケージが存在します。

注記

C# コード ジェネレータは、コード生成時に、モデルからファイルへのマッピング パッケージに情報を追加します。たとえば、C# コード ジェネレータは C# ファイル アーティファクトのタイムスタンプを更新し、モデルと生成された C# コード間のナビゲートを可能にする情報を追加します。したがって、モデルからファイルへのマッピング パッケージが書き込み可能であることが要求されます。モデルからファイルへのマッピング パッケージを独自のファイルに保存することにより、モデル パッケージが入ったファイルを修正することなく、コードの生成が可能になります。

モデルからファイルへのデフォルトのマッピング

ユーザー定義のモデルからファイルへのマッピングが存在しない場合、C# コード ジェネレータは、モデルからファイルへのデフォルトのマッピングを作成してから、コードの生成を行います。このデフォルトのマッピングは、対応するモデル パッケージの隣の別パッケージに格納されます。モデルからファイルへのマッピングは、以下の規則に従って生成されます。

1. パッケージに格納される各定義は、デリゲート以外は、独自の C# ファイル内に生成される。
2. C# ファイルには、そのファイルに格納される定義の完全修飾名が付けられる。
3. デリゲートは、各定義と同じ C# ファイルに生成される。

例 **466:** モデルからファイルへのデフォルトのマッピング

UML

```
package P {
class Class1 {
    void Operation1( long p1);
    delegate void Delegate1( long a);
}
enum DataType1 {
    Literal1
}
}
```

C#（"P.Class1.cs" ファイル内）

```
namespace P {
class Class1
{
    void Operation1(long p1)
    {
    }

    delegate void Delegate1(long a);
}
}
```

C#（"P.DataType1.cs" ファイル内）

```
namespace P {
enum DataType1
{
```

```
    Literal1
}
}
```

モデルからファイルへの部分的なマッピングが存在しており、モデルに後で定義が追加されている場合、既存のモデルからファイルへのマッピング パッケージは、不足している定義に対応したファイル マッピングによって更新されます。

生成後のファイルの格納場所

モデルからファイルへのマッピングは、生成後のファイルのファイル システム内での格納場所も定義します。デフォルトでは、これらのファイルは、対応する UML 定義が位置する .u2 ファイルと同じフォルダに生成されます。

特定のファイルを別の場所に生成する場合は、ファイル アーティクラフトのタグ付き値 `path` を変更できます。

全ファイルを別の場所に生成するには、ステレオタイプ `<<cSharpCodePath>>`（C# コード パス）を適用し、タグ付き値 `path` にフォルダを指定します。このステレオタイプは、モデルからファイルへのマッピング パッケージか、あるいはモデル ノードに対して適用できます。前者の場合、モデルからファイルへのマッピング パッケージに格納されるファイル アーティファクトのすべての相対パスは、C# ファイルの生成時に、そのパスに相対して変換されます。後者の場合、モデルからファイルへのすべてのマッピング パッケージのいずれにも `<<cSharpCodePath>>` が適用されていなければ、すべてのマッピング パッケージにオプションが適用されます。

生成済み C# ファイルとの間のナビゲート

生成された C# ファイルを開くには、モデル内でそのファイルを表すファイル アーティファクトをダブルクリックします。また、生成された定義のショートカット メニューから［ソースに移動］コマンドを選択して、生成された C# ファイル内の定義の格納場所（または複数の格納場所ー場合によっては、1 つの UML 定義が複数の C# ファイルに分かれることがあります）まで直接ナビゲートできます。

生成された C# ファイルまでナビゲートすると、Tau に組み込まれているテキスト エディタ（または、Visual Studio .NET インテグレーションがインストールされ、起動している場合は Visual Studio）でそのファイルが開かれます。

また、生成された C# コードから対応する UML モデル エンティティへと、逆方向にナビゲートすることも可能です。これは、Visual Studio .NET インテグレーションの特定のコマンドを使用するか、あるいは Tau に組み込まれたテキスト エディタのショートカット メニューから［ソースに移動］コマンドを選択して行います。

翻訳ルール

UML モデルから C# コードへの生成時に、各種 UML エンティティから C# コードへのマッピングを定義する翻訳ルールが使用されます。これらの翻訳ルールは、UML から C# へのマッピングで説明しています。UML エンティティには、C# へのマッピングが定義されていないものもあります。現時点では、C# コード ジェネレータはそのようなエンティティを検出した場合、C# に翻訳せずに無視します。

生成済み C# コードのコンパイル、実行、デバッグ

ここでは、生成された C# アプリケーションのコンパイル、デバッグ、実行には、別の開発環境を使用することを想定しています。どの C# 開発環境でも使用できますが、C# サポートは主として Microsoft Visual Studio .NET で使用されるように設計されています。この開発環境との特別なインテグレーションを利用できます。このインテグレーションにより、生成された C# コードの開発は大幅に促進されます（Visual Studio .NET インテグレーションを参照）。

既存の C# コードのインポート

このセクションでは、既存の C# ソース コードを UML モデルにインポートする方法について説明します。インポートしたい C# コードがアセンブリとしてビルドされていた場合、ソース コードではなくアセンブリを Tau にインポートすることが可能です。アセンブリをインポートすると、インポートした C# モジュールは UML モデルからはブラック ボックスとして表示されます。これを視覚化してアクセスできます。詳細については、.NET アセンブリ インポータを参照してください。ただし、インポートの目的が UML 内の C# モジュール全体のリバースエンジニアリング、または解析である場合、そして C# メソッドの実装もインポートしたい場合、このセクションで説明する C# コード インポータを使用する必要があります。

C# インポート ウィザードの使用

C# インポート ウィザードを使用して、既存の C# コードを Tau にインポートします。以下のステップを実行します。

1. ［モデル ビュー］でモデル ノードを選択し、［ファイル］メニューから［インポート］を選択します。
2. インポート ダイアログで［C# のインポート］を選択して、［OK］をクリックします。
3. Visual Studio プロジェクト（.csproj ファイル）に基づいてインポートを実行するか、あるいは個々の C# ファイル（または C# ファイルが入っているフォルダ）を指定するかを選択します。Visual Studio プロジェクトをインポートする場合、プロジェクトに複数の構成が存在するときは、使用するプロジェクト構成も指定する必要があります。その結果、選択した構成に含まれるすべての C# ファイルが、その構成に対応した .csproj ファイルに指定されている設定に基づいてインポートされます。

4. 次のウィザード ページで、インポートの追加オプションを指定できます。
   - ［ダイアグラムの生成］オプションを使用すると、インポータはインポートした各 C# ファイルについてダイアグラムを作成します。このダイアグラムには、C# ファイルに含まれるすべてのパッケージ、クラス、インターフェイスなど、それら関係が表示されます。
   - インポートを実行する前に追加の詳細オプション（プリプロッサ設定など）を指定したい場合、［いますぐインポート］オプションの選択を解除できます。詳細については、詳細インポート オプションを参照してください。
5. ［完了］をクリックしてインポート ウィザードを終了します。

詳細インポート オプション

C# インポート ウィザードでは、C# インポータに対して設定できる共通オプション数は限られています。より詳細なオプションを指定するには、以下のステップを実行します。

1. インポート ウィザードによって作成された最上位のインポート パッケージを選択します。
2. プロパティ エディタを開き、フィルタとして `<<cSharpImportSpecification>>` ステレオタイプを選択します。このオプションは、Visual Studio プロジェクトファイルからのインポートを行った場合は利用できません。この場合には、インポートのためのすべてのオプションがプロジェクトファイルから取得されます。
3. すでに適切なインポート オプションを指定している場合は、インポート パッケージのショートカット メニューから［モデルの更新］コマンドを選択してインポートを実行します。

C# のインポート結果

インポートの結果は、インポート ウィザードによって作成される最上位のインポート パッケージの下に格納されます。この結果には、グローバル名前空間で定義されるすべての C# 定義に対応する UML 定義が含まれます。作成されるパッケージには、インポートされた C# プロジェクト ファイルと同じ名前が付けられます。個々の C# ファイルがインポートされた場合、パッケージは `ImportedDefinitions_X` と呼ばれます。X は名前を一意にするために付加される数字です。

C# コードを UML に翻訳するとき、複数の翻訳ルールが使用されます。これらのルールは、C# コードの生成時に使用されるルールと同じですが、逆方向に適用されます。詳細については、UML から C# へのマッピングを参照してください。

C# インポータは、インポート パッケージの他に、モデルからファイルへのマッピング仕様を格納するパッケージも作成します。このパッケージの目的は、C# コード ジェネレータを使用して、インポートされた C# コードを再生成できるようにすることです。したがって、モデルからファイルへのパッケージの構造は、C# コード ジェネレータが使用する構造と同じです（モデルからファイルへのマッピングを参照）。

インポート済み C# ファイルとの間のナビゲート

インポートされた C# ファイルを開くには、モデル内でそのファイルを表すファイル アーティファクトをダブルクリックします。また、インポートされた定義のショートカット メニューから［ソースに移動］コマンドを選択して、インポートされた C# ファイル内の定義の格納場所（または複数の格納場所－場合によっては、1 つの UML 定義が複数の C# ファイルに由来することがあります）まで直接ナビゲートできます。

インポートされた C# ファイルまでナビゲートすると、Tau に組み込まれているテキスト エディタ（または、Visual Studio .NET インテグレーションがインストールされ、起動されている場合は Visual Studio）でそのファイルが開かれます。

また、インポートされた C# コードから対応する UML モデル エンティティへ、と逆方向にナビゲートすることも可能です。これは、Visual Studio .NET インテグレーションの特定のコマンドを使用するか、あるいは Tau に組み込まれたテキスト エディタのショートカット メニューから［ソースに移動］コマンドを選択して行います。

モデルとソース コードの同期

C# ソース コードがモデルから生成された場合、あるいは C# ソース コードがモデルにインポートされた場合、Tau は、モデルからファイルへのマッピングを使用して、コードとモデルの関係を追跡します。モデルからコード、または逆方向への完全なナビゲート機能により、あるいはモデルからファイルへのマッピングにより、モデルとコードのいずれかに変更があったとき、それらの間の同期をとることができます。したがって、UML モデルから、あるいは C# コード内で、C# アプリケーションに対する変更を行うことができます。

自動同期と手動同期

モデルとソース コードの更新は、2 通りの方法で行うことができます。ソース コードまたはモデルの変更時に自動的に行うか、専用の更新コマンドを使用して手動で行うかです。

自動同期

デフォルトでは、自動同期はオンになっています。オフになっている場合は、自動同期の適用方法を指定する次のオプションのどちらかまたは両方をオンにしてオンにできます。

- Automatic model update
- Automatic source generation

これらの詳細については、C# の設定で説明します。

手動同期

モデルと一致するように C# ソース コードを手動で更新するには、［ソースコードの更新］コマンドまたは［ソースコードの強制更新］コマンドを使用します。

C# ソース コードと一致するようにモデルを手動で更新するには、［モデルの更新］コマンドまたは［モデルの強制更新］コマンドを使用します。

UML から C# へのマッピング

このセクションでは、UML と C# 言語間のマッピングについて説明します。このマッピングは、UML 構成要素から C# 構成要素への翻訳ルールの形で示されます。これらの変換ルールは C# コードの生成時に使用されます。これらの翻訳ルールは、C# 構成要素から UML 構成要素への変換を定義するために逆方向に適用することもできます。逆方向への適用では、既存の C# コードのインポート時にこれらのルールが使用されます。

モデリング言語である UML は、C# よりも構成要素の多い言語です。下記の翻訳ルールに明示的に記載されていない UML 構成要素は、コード生成時に C# に翻訳されません。

一般的な翻訳ルール

ここでは、各種の変換されたエンティティに適用される一般的な翻訳ルールを説明します。

定義の名前

生成される C# 定義の名前は、対応する UML 定義の名前と同じです。

生成される C# 名が有効な C# 識別子となることを保証するために、名前変換は行われません。特に、モデルに以下のものを含まないようにします。

- 操作パラメータなど、名前を持たない定義
- C# のキーワードと同じ名前を持つ UML 定義

UML と C# の定義済みの型

UML の定義済みの型には C# の対応する型にマッピングされるものもありますが、C# 用のモデリングの場合には、C# の定義済みの型のみを使用することを推奨します。C# の定義済みの型に対応する UML 表現は、`TTDCSharpPredefined` プロファイルに記載されています。

`TTDCSharpPredefined` プロファイルには、C# 配列構成要素の UML 表現も含まれます。SArray 型は一次元の C# 配列を表し、`MArray` 型は多次元の C# 配列を表します。これらの型は、C# 配列で利用可能な操作に対応した操作を持っています。

例 **467:** ——————————————————————————————

UML

```
class X {
  Charstring v;
  SArray<int> x;
  MArray<int,2> y;
```

```
}
```

C#

```
class X
{
    string v;
    int[] x;
    int[,] y;
}
```

ポインタ型 (非管理タイプへのポインタ) の表記は、TTDCppPredefined ライブラリのサブセットを使用しています。これは、定義済み C/C++ 型の表記に使用されるライブラリです。このライブラリは、C# におけるポインタ型表記で使用される "void*" 型と CPtr テンプレートを含んでいます。

コメント

UML 定義に添付されたモデル コメントは、C# ドキュメント形式のコメント（3 個のスラッシュ /// で始まり、XML タグで囲まれる）に翻訳されます。定義の種類によっては、C# コンパイラはドキュメント形式のコメントのみをサポートします（詳細については、C# コンパイラのドキュメントを参照してください)。他の種類の定義に存在するモデル コメントは、C# に翻訳されません。

翻訳でサポートされる XML タグを以下に示します。

- <param> : 操作パラメータまたはデリゲート パラメータの場合。
- <summary> : 他の定義の場合。

UML テキスト構文で /* */ または // を使用して定義されている通常の注釈コメントは、C# に翻訳されません。ただし、モデル コメントが // または /* で始まる場合、このコメントは、ドキュメント形式のコメントとしてではなく、通常の C# コメントとして生成されます。

例 **468:** コメントの翻訳 ———————————————————————————

次のパラメータ 's' では、モデルで定義されたコメント「Param comment」を持ちます (ただし、テキスト構文では表示されません)。

UML

```
class X comment "//Class comment" {
  void foo(string s) comment "Summary comment";
}
```

C#

```
//Class comment
class X
{
    ///<summary>
    ///Summary comment
    ///</summary>
    ///<param name="s">Param comment</param>
    void foo(string s) {}
```

```
}
```

パッケージ（**Package**）

パッケージは名前空間に翻訳されます。

パッケージに **<<TTDCppPredefined::globalNamespace>>** ステレオタイプが適用されている場合、このパッケージは C# のグローバル名前空間を表します。したがって、このパッケージの名前空間は生成されません。

注記

このステレオタイプがモデル内の複数のパッケージに適用されている場合、これらすべてのパッケージ内の名前を必ず一意にしなければなりません。したがって、グローバル名前空間を表すために使用するパッケージは最大 1 つまでとすることを推奨します。

例 **469:**

UML

```
package P {}
```

C#

```
namespace P {}
```

クラスとインターフェイス（**Class** と **Interface**）

UML のクラスは、C# のクラスに翻訳されます。

UML のクラスが抽象と定義されている場合、C# のクラスも抽象となります。UML のクラスに <<sealed>> ステレオタイプが適用されている場合、C# のクラスも sealed と宣言されます。

UML のインターフェイスは、C# のインターフェイスに翻訳されます。

例 **470:** クラスとインターフェイスの翻訳

UML

```
class C {}
<<sealed>> class FinalImpl {}
interface Ifc {}
```

C#

```
class C {}
sealed class FinalImpl {}
interface Ifc {}
```

継承

UML の汎化関係（継承または実現化）は、C# の（クラスまたはインターフェイスの）継承に翻訳されます。

例 **471:**

UML

```
class C : BaseClass, Ifc1, Ifc2 {}
```

C#

```
class C : BaseClass, Ifc1, Ifc2 {}
```

partial 型

UML のクラスまたはインターフェイス（およびデータ型（Datatype））に <<partial>> ステレオタイプを適用されている場合があります。この場合、対応する C# のクラス、インターフェイスまたは構造体は、**partial** キーワードによって定義されます。これにより、定義を複数のソース ファイルに分割できます。クラス、インターフェイスまたは構造体の各メンバーに対して使用するソース ファイルは、ソース ファイルを表す C# のファイル アーティクラフトからメンバー定義への <<manifest>> 依存を使用して、モデルからファイルへのマッピング で定義されます。

データ型（**Datatype**）

データ型にリテラルが最低 1 つ存在する場合、データ型は C# の enum に翻訳されます。データ型にリテラルが 1 つも存在しない場合、データ型は C# の構造体に翻訳されます。

例 **472:**

UML

```
datatype D1 {}
enum D2 { L1, L2 }
```

C#

```
struct D1 {}
enum D2 { L1, L2 }
```

ステレオタイプ（**Stereotype**）

`TTDCSharpPredfined::CSAttribute` を継承するステレオタイプは、`System.Attribute` を継承するクラスに翻訳されます。

例 **473:**

UML

```
public stereotype S : CSAttribute
{}
```

C#

```
public class S : System.Attribute
{}
```

シンタイプ（**Syntype**）

パッケージまたはクラスで定義されるシンタイプは、C# の "using alias" ディレクティブに翻訳されます。

例 **474:**

UML

```
syntype MyString = System::String;
```

C#

```
using MyString = System.String;
```

依存（**Dependency**）

2 つのパッケージ間の <<access>> 依存は、その依存を所有するパッケージの翻訳である名前空間で、using 宣言に翻訳されます。

例 **475:**

UML

```
package P1 <<access>> dependency to P2 {

}
package P2 {

}
```

C#

```
namespace P1
{
  using P2;
}

namespace P2 {}
```

C# ファイル アーティファクトからパッケージへの <<access>> 依存は、対応する C# ファイルの先頭で using 宣言に翻訳されます。

操作（**Operation**）

UML の操作は、C# のメソッドに翻訳されます。UML の操作本体は、C# の対応するメソッド本体に翻訳されます。UML の操作に指定された本体がない場合、C# の空のメソッド本体が生成されます。

「get」または「set」アクセサリが指定された、'[]' という名前の派生型操作は、C# のインデクサに翻訳されます。

例 **476:**

UML

```
class X {
  void Operation1() { }
  public <<Derived="true">> string '[]'(int index)
    get {return "First";}
    set {}
}
```

C#

```
class X {
  void Operation1() { }
  public string this[int index]
  {
    get {return "First";}
    set {}
  }
}
```

パラメータ

UML のパラメータは、C# のパラメータに翻訳されます。UML のパラメータの方向は、下表のように翻訳されます。

UML の方向	**C# パラメータの種類**
in（または方向指定なし）	通常のパラメータ
inout	**ref** パラメータ
out	**out** パラメータ
操作の戻り	操作の戻り

例 **477**

UML

```
bool op(bool p1, in bool p2, inout bool p3, out bool p4);
```

C#

```
bool op(bool p1, bool p2, ref bool p3, out bool p4)
{}
```

Virtual、redefined、finalized 操作

virtual 操作は、virtual メソッドに翻訳されます。

redefined 操作は、override メソッドに翻訳されます。

finalized 操作は、sealed メソッドに翻訳されます。

例 **478:**

UML

```
virtual void op1();
redefined void op2();
finalized void op3();
```

C#

```
virtual void op1() {}
override void op2() {}
sealed void op3() {}
```

型変換演算子

<<implicitTypeConvOperator>> でステレオタイプされる操作は、単一の暗示的な C# 型変換演算子に変換されます。

<<explicitTypeConvOperator>> でステレオタイプされる操作は 、単一の明示的な C# 型変換演算子に変換されます。

いずれの場合も操作の戻り値の型は、変換演算子のターゲット型を示しています。

例 **479:**

UML

```
class C {
    static <<implicitTypeConvOperator>> int implicit(C x);
    static <<explicitTypeConvOperator>> bool explicit(C x);
}
```

C#

```
class C
{
     public static implicit operator int(C x) {}
     public static explicit operator bool(C x) {}
```

```
}
```

デリゲート（**Delegate**）

UML のデリゲートは、C# のデリゲートに翻訳されます。デリゲートのパラメータは、操作のパラメータと同じように翻訳されます（「パラメータ」参照）。

例 **480**

UML

```
class B {
  delegate void Delegate1(bool p1,string p2);
}
```

C#

```
class B
{
  delegate void Delegate1(bool p1, string p2);
}
```

属性（**Attribute**）

非派生型の UML 属性は、C# のフィールドに翻訳されます。

「get」または「set」アクセサリが指定された派生型の UML 属性は、C# のプロパティに翻訳されます。

例 **481: UML** 属性の翻訳

UML

```
class Class1 {
  string a;
  string / b
  get
  {
    return "foo";
  }
  set
  {
    a = value;
  };
}
```

C#

```
class Class1
{
  string a;
  string b
  {
    get
```

```
      {
        return "foo";
      }
      set
      {
        a = value;
      }
    }
  }
```

UML 属性が TTDCSharpPredefined から Event 型によって型指定されている場合、その UML 属性は C# のイベントに翻訳されます。

例 **482:**

UML

```
public delegate void D(object 'sender', EventArgs e);
public class SampleEventSource {
  public Event<D> SampleEvent;
}
```

C#

```
public delegate void D(object sender, EventArgs e);
public class SampleEventSource
{
  public event D SampleEvent;
}
```

定数属性

UML の定数属性は、C# の定数に翻訳されます。

例 **483:**

UML

```
const long x = 4;
```

C#

```
const long x = 4;
```

複数の多重度を持つ属性

UML 属性が形式的な複数の多重度を持つ場合、UML の属性型の翻訳である型によって型指定された、C# のフィールドに翻訳されます。形式的な多重度に関する情報は C# の翻訳には含まれませんが、コメントの形で生成されたコードに組み込まれます。

複数の多重度を持つ属性は、非形式と指定するほうが有用な場合がよくあります。この場合、適切なコンテナ型で属性を型指定できます。このコンテナ型は C# のフィールドでも型としても使用されます。

例 **484:**

この例では、`a1` は形式的な多重度を持ち、`a2` は非形式な多重度を持っています。どちらの多重度も複数です。

UML

```
B [*] a1;
System::Collections::Generic::List<B> {[*]} a2;
```

C#

```
B a1 /*{{Multiplicity='*'}}*/;
System.Collections.Generic.List<B> a2;
```

関連（**Association**）

名前のない一方向の関連は、UML モデルの属性として示されます。このような属性の翻訳は、属性（Attribute）のルールに従います。

テンプレート（**Template**）

UML のテンプレート定義は、ジェネリック パラメータを持つ対応する C# の定義に翻訳されます。UML のテンプレート パラメータに対する制約は、C# のジェネリック パラメータに対する対応する制約に翻訳されます。

例 **485:**

UML

```
template <class T atleast B >
class Class1 {
  T a;
}
```

C#

```
class Class1<T> where T : B
{
  T a;
}
```

式（**Expression**）

UML の式には、C# の式へのマッピングが定義されているものがあります。しかし、UML の式の型は定義済みの UML 型であり、C# の式の型は定義済みの C# 型である場合があることに注意してください。定義済みの UML の型は、定義済みの C# の型と必

ずしも完全に互換性があるとは限りません。定義済みの C# の型を定義済みの UML の型から作成できるなど、ある程度の互換性はあります。ただし、定義済みの型で使用可能な演算子は、UML 言語と C# 言語では同じではありません。

定義済みの型の間に型の互換性がないことでモデルにセマンティック エラーが発生した場合、非形式な UML 式を使用できます。このような非形式な式では C# の式のインライン化が可能です。式のテキストは、コード ジェネレータが生成後の C# ファイルにそのままコピーします。非形式な UML の式はすべての型と互換性があり、オーバーロード操作の呼び出し時には、曖昧性を避けるために、タイプキャストの使用（または中間属性の宣言）が必要になる場合があります。

（非形式ではない）UML の式は、下表のように翻訳されます。

UML の式	**C# の式**	**UML の例**	**C# の例**
かっこ付きの式	かっこ付きの式	`(a + b)`	`(a + b)`
単項式	単項式	`-a`	`-a`
this 式	this 式	`this.x`	`this.x`
二項式	二項式	`a = b`	`a = b`
インデックス式	インデクサ アクセス	`obj[10]`	`obj[10]`
生成式	new 演算子の使用	`new C()`	`new C()`
条件式	条件（**?:**）演算子の使用	`a ? b : c`	`a ? b : c`
実数値	ダブル型リテラル	`3.14`	`3.14`
整数値	int 型リテラル	`8`	`8`
charstring 値	string 型リテラル	`"Tau"`	`"Tau"`
call 式	呼び出し式	`foo()`	`foo()`
フィールド式	メンバー アクセス式	`a.b`	`a.b`
インスタンス式	属性	`S(. .)`	`[S()]`
識別子	単純名	`x`	`x`
基本式	base	`base`	`base`
値式	value	`value`	`value`
リスト式	配列初期化子	`{1,2}`	`{1,2}`
デリゲート実装式	匿名メソッド式	`delegate() {return;}`	`delegate(){return;}`

C# 言語には、上記の表に記載した以外の式が存在します。これらの式は、UML では非形式な式として表現されます。

特別な呼び出し式

特定の UML 操作の呼び出しは、下表に示すとおり、特別な方法で翻訳されます。

呼び出される UML の操作	C# の式	UML の例	C# の例
Predefined::cast	キャスト式	`cast<C>(x)`	`(C) x`
TTDCSharpPredefined::typeof	typeof 演算子の使用	`typeof(long)`	`typeof(long)`
Predefined::is	is 演算子の使用	`is<MyC>(x)`	`x is MyC`
Predefined::as	as 演算子の使用	`as<MyC>(x)`	`x as MyC`
TTDCSharpPredefined::Default	デフォルト値式	`Default<C>()`	`default(C)`
TTDCppPredefined::GetValue	ポインタ メンバー アクセス	`p.GetValue().x`	`p->x`

アクション（**Action**）

UML のアクションには、C# のアクション コードへのマッピングが定義されているものがあります。ただし、UML 言語と C# 言語のアクションのセマンティックは同じではありません。現時点では、C# コード ジェネレータは、セマンティックの違いを調整するための変換は実行しません。したがって、C# への翻訳対象となる操作の振る舞いは、非形式な UML のアクションを使用して指定するとよいでしょう。このような非形式なアクションでは、C# コードのインライン化が可能です。インライン化された C# コードは、コード ジェネレータが生成後の C# ファイルにそのままコピーします。

C# インポータは、すべての C# アクションコードを非形式的な UML アクションとしてインポートします。

例 **486:** 非形式な **UML** のアクションとインライン化された **C#** コードの使用

UML

```
void hi() {
[[System.Console.WriteLine("Hi there!");]]
}
```

C#

```
void hi()
{
System.Console.WriteLine("Hi there!");
}
```

（非形式ではない）UML のアクションのは、下表のように翻訳されます。翻訳ルールの例は、UML と C# の構文に違いがある場合、あるいは参照先のアクションの種類が明確ではない場合にのみ規定されています。

UML のアクション	C# 文	UML の例	C# の例
複合アクション	複合文		
continue アクション	continue 文		
break アクション	break 文		
if アクション	if 文		
ループ アクション	ループ アクションの種類に応じて、for 文、while 文、または do 文		
式アクション	式文	`foo(3);` `x = 10;`	`foo(3);` `x = 10;`

定義アクション	宣言文	`{`
		`string s;`
		`}`
			`{`
			`string s;`
			`}`

try アクション	try 文		
throw アクション	throw 文		
分岐アクション	switch 文		
リターン アクション	return 文		
join アクション	goto 文	`join L;`	`goto L;`

上記の表に記載されていない UML のアクションは、C# には翻訳されません。

C# の設定

C# の各モデルについて、C# 固有の設定がプロジェクト ファイルに格納されています。これらの設定を表示・編集するには、次の手順を行います。

1. ［モデル ビュー］でモデル ノードを選択します。
2. 右クリックして、［プロパティ］を選択します。
3. プロパティ エディタの［ステレオタイプ］ボタンをクリックし、<<C# Settings>> ステレオタイプを適用します。
4. ［フィルタ］ドロップダウン メニューで、［C# Settings］を選択します。

以下の設定を使用できます。

- Automatic model update
  - この設定を有効にした場合、たとえば C# ソース ファイルを外部テキスト エディタで変更して保存したときなど、C# ソース ファイルが変更されるとモデルが自動更新されます。デフォルトでは、［automatic model update］は無効になっています。
- Automatic source generation
  - この設定を有効にした場合、モデルが変更されると C# ソース ファイルが自動更新されます。この更新は、モデルの編集を終了してから少し時間をおいて行われます。デフォルトでは、［Automatic source generation］は無効になっています。
- Support roundtrip
  - デフォルトで、ラウンドトリップのサポートは有効になっています。生成された C# コードへの変更をモデルに戻す予定がない場合は、このスイッチをオフにします。オフにすると、コードを生成するたびに、C# ソースコードは完全に再生成されます。

38

C# 言語向け Visual Studio .NET インテグレーション

Tau Visual Studio .NET インテグレーションにより、生成された C# アプリエーションのビルドとデバッグを行うことができます。詳細については、Visual Studio .NET インテグレーションの説明を参照してください。

UML と C++

「UML と C++」セクションの各章では、UML プロジェクトの C++ アプリケーションへの変換方法について説明しています。

39

Tau での C++ サポート

このセクションでは Tau での C++ サポートの概要と、さまざまな C++ 活用シナリオにおけるツールの使用方法について説明します。

- 既存の C++ コードの可視化
- UML - C++ ラウンドトリップ エンジニアリング
- 高度な UML 概念を使ったアプリケーションの生成
- 既存の C++ アプリケーションの Tau への移行
- C++ 開発環境での Tau 管理下の C++ コード の使用
- Tau UML 環境から C++ API にアクセスする
- Tau で生成されたアプリケーションの実行のトレース

概要

主な機能

Tau での C++ サポートの基本機能を 1292 ページの図 250 に示します。

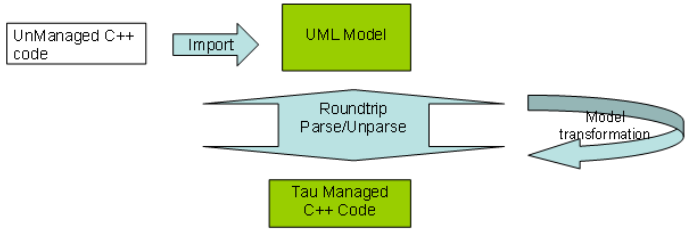


図 250: C++ ラウンドトリップ

C++ コードのラウンドトリップ解析／逆構文解析とモデル変換が主な機能です。ラウンドトリップ解析／逆構文解析で、C++ 構文と UML モデルの特定の概念との間の 1 対 1 のマッピングが可能になります。この機能を使って、C++ ファイルと UML モデルとの同期を取ります。モデル変換機能により、より詳細な UML 概念が使用可能になります。つまり、C++ 構文と 1 対 1 のマッピングがない場合でも、C++ コードを生成するような場合です。代表的な例は、UML 状態機械定義からの C++ コード生成です。

外部 C++ とラウンドトリップ

また、Tau には、C++ API を UML モデルにインポートするための汎用 C/C++ のインポート機能があります。この機能とラウンドトリップ解析／逆構文解析とでは目的が異なります。C/C++ のインポートは、プリプロセスのディレクティブやマクロのような C++ 言語をサポートするものです。このインポートでは、その処理中に、マクロの拡張を実行するなど、インポート中になんらかの変換を行います。C/C++ のインポートは、UML 開発者が C++ ライブラリの API を利用できるようにすることを主眼としています。これに対して、ラウンドトリップ C++ 解析／逆構文解析は、C++ コードと UML モデル間で厳密に同期をとることを主な目的としています。このため、C++ のサブセットを定義して、C++ 構文と UML モデル間の 1 対 1 のマッピングが維持されるようにしています。このサブセットの概念のみがラウンドトリップ解析／逆構文解析に用いられます。

Tau での C++ の使用

上で述べた基本機能を利用するさまざまなシナリオが考えられます。

- 既存の C++ コードの可視化
- UML - C++ ラウンドトリップ エンジニアリング

- 高度な UML 概念を使ったアプリケーションの生成（UML 状態機械および合成構造の定義に基づく概念など）

さらに、以下のような、特別な考慮が必要なケースがあります。

- 既存の C++ アプリケーションの Tau への移行
- C++ 開発環境での Tau 管理下の C++ コード の使用
- Tau UML 環境から C++ API にアクセスする
- Tau で生成されたアプリケーションの実行のトレース

C++ の使用シナリオ

既存の C++ コードの可視化

これは Tau がどのように C++ の開発者を支援するかを示す一番単純な例です。このシナリオでは、C/C++ のインポートと Tau のダイアグラムを自動的にレイアウトする機能を使用しています。

また、複数の C++ クラス定義を含む C++ ヘッダー ファイルについて考えます。このクラス定義を UML ダイアグラムで可視化するには、以下の手順を行います。

- インポート ウィザードを使用して、C++ ファイルを UML モデルにインポートします。まず、[ファイル] メニューの [インポート] コマンドを選択します。[C/C++ のインポート] を選択し、インポートするファイルを選択します。単純なケースでは、そのまま直接ファイルをインポートでき、インポートしたファイル内の定義を含む **ImportedDefinitions** という名前のパッケージがルート レベル に作成されます。
- インポートの過程で、インクルード ファイルのパスなどの追加の情報を与える必要が生じる場合があります。この場合は、ウィザードの [今すぐインポート] の選択を解除して、[ImportedDefinitions] パッケージのプロパティに表示されるインポートの設定を変更します。次に、[ImportedDefinitions] パッケージを右クリックして、ショートカット メニューの [C/C++ のインポート] を選択し、インポートを完了します。
- インポートしたクラスをダイアグラムで可視化するには、ダイアグラムのショートカット メニューから [要素の表示] コマンドを使用して、表示するクラスを選択します。または、[プレゼンテーションの作成] ダイアログを使用して、要素とダイアグラムを作成します。

モデルからソース コードへのナビゲート

C++ コード ジェネレータによって、モデルからコードへナビゲートできます。コード生成の結果としてヘッダー ファイル、または実装ファイルが作成されている場合、[モデル ビュー] で UML 要素を右クリックし、[ソースに移動] を選択すると、UML 要素から対応するコードの行にナビゲートできます。

注記

Visual Studio .NET インテグレーションでは、双方向のナビゲーションも可能です。C++ ソース コードから Locate in Tau も実行できます。

UML - C++ ラウンドトリップ エンジニアリング

Tau のラウンドトリップ機能について、以下の例を使って説明します。この例では、Tau を使用して、単純な「Hello World...」スタイルのアプリケーションをモデリングします。

プロジェクトの作成

最初に、メニューの［ファイル］->［新規］を選択し、［プロジェクト］タブで［UML (C++ コード生成用)］プロジェクト タイプを選択して、Tau に新規プロジェクトを作成します。

次に、1295 ページの図 251 のようにクラス「Hello」と 1 つの操作「PrintIt」を持つ単純な UML モデルを作成します。

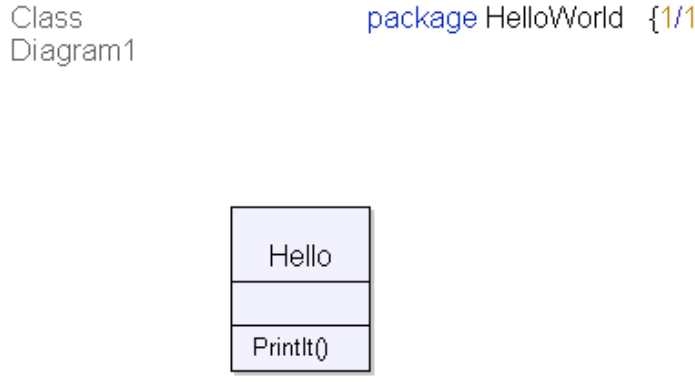


図 251: クラス図の例

C++ コードの生成

C++ コード生成するには、［ビルド］メニューから［構成の生成］コマンドを選択します。ビルド ウィザード が開いて入力を求められた場合は、作成したパッケージをビルド ルートとして指定します。この操作で、パッケージ内のすべての要素が生成後の C++ コードに含まれます。

ウィザードを閉じると、UML モデルに対応する C++ ファイルが生成されます。生成されたファイルは、パッケージ「Result of C++ Code Generation」に入ります。このコード ジェネレータで使用されるファイル マッピングでは、クラスごとにヘッダー ファイルとソース ファイルを 1 つずつ生成します。この例では、2 つのファイル「Hello.h」と「Hello.cpp」が作成されます。

［モデル ビュー］の［Hello.h］ファイル アーティファクトをダブルクリックすると、ファイルがテキストエディタで開かれます。クラス定義が期待したとおりに生成されているかを確認できます。

```
#ifndef GEN_IzuOjIq0WdSL_45GMYmEjyT3hE
#define GEN_IzuOjIq0WdSL_45GMYmEjyT3hE
#include "torAnnotations.h"
```

```
namespace HelloWorld {
    class Hello {
        void PrintIt();
    };
}

#endif/*<GENERATED> GEN_IzuOjIq0WdSL_45GMYmEjyT3hE */
```

生成されたヘッダー ファイルには以下の行が含まれます。

```
#include "torAnnotations.h"
```

torAnnotations.h ファイルには、マクロが含まれます。このマクロは特定の UML 構造体でラウンドトリップを行う場合に使用されます。構造体の例としては、これがないとインターフェイスではなくクラスに変換されてしまうインターフェイスなどがあります。このファイルは UML へのマッピングの精度を上げるために使用されます。たとえばユーザーが手動でインターフェイスを追加してモデルに挿入することを可能にするため、これは必ず生成されます。

生成後の C++ コードの変更

このファイルを修正し、変更を UML モデルに反映させることができます。

クラスに「i」という属性を追加してみましょう。

```
#ifndef GEN_IzuOjIq0WdSL_45GMYmEjyT3hE
#define GEN_IzuOjIq0WdSL_45GMYmEjyT3hE
#include "torAnnotations.h"

namespace HelloWorld {
    class Hello {
        int i;
        void PrintIt();
    };
}

#endif/*<GENERATED> GEN_IzuOjIq0WdSL_45GMYmEjyT3hE */
```

変更をモデルに反映

変更を UML モデルに反映させるには、以下の手順を行います。

Hello.h ファイルを保存して、［ビルド］メニューの［構成の更新］コマンドを実行します。

［モデル ビュー］で Hello クラスに属性”i ”が追加されたことを確認できます。

ラウンドトリップ機能を使って C++ コードの保守を行う場合は、次のことに注意してください。

- ラウンドトリップ機能では、C++ 構文のサブセットのみが使用できます。このサブセットは、一般的に使用されるほとんどの C++ 言語フィーチャをカバーします。詳細は C++ テキスト構文で説明しています。

- サポートされていない構成要素を使用する場合、「ユーザー セクション」を作成できます。詳細は C++ テキスト構文ドキュメントの「コメント」のセクションを参照してください。

デフォルトでは、［構成の更新］操作の対象となるのは最後の更新またはコード生成以降に変更されたファイルのみです。すべてのファイルを対照として更新を行うには、［Full Update］操作を行います。

モデルとコードの自動同期

上で説明したシナリオでは、モデルとコードまたはコードとモデルの間の同期を取るために手動のコマンドを使用しました。この同期を自動で行うことができます。つまり、モデルが変更されて保存されるとコードが生成され、生成されたファイルを変更して保存するとモデルが更新されるようにできます。この機能は、デフォルトでは無効になっていますが、以下の手順で有効にできます。

1. ［モデルビュー］でビルドアーティファクトを選択します。
2. コンテキストメニューで［プロパティ］を選択します。
3. プロパティエディタのフィルタリストで、'C++ Application Generator' を選択します。
4. コードの自動同期化を有効にするには［Automatic code generation］をチェックします。
5. モデルの自動同期化を有効にするには［Automatic model update］をチェックします。

注記

［Automatic model update］オプションを有効にした場合は、［Support roundtrip］オプションも有効化する必要があります。

高度な **UML** 概念を使ったアプリケーションの生成

本章で述べたように、Tau の UML - C++ ラウンドトリップ エンジニアリング機能を使用して、クラス、操作、データ型といった UML と C++ 表現の間で 1 対 1 の対応をもつ一般的な概念をすべて扱うことができます。ただし、より高度な UML 概念を使用する場合は、Tau の C++ サポートの一部として提供されるモデル変換および関連するランタイム フレームワークを利用します。

モデル変換

モデルの編集中に対話的に変換が実行される場合があります。たとえば、クラスが「**Active**」と設定されている場合です。「Active」と設定されたクラスは、自動的に「DispatchableClass」（Tau で提供されるランタイム フレームワークの一部）を継承します。通常ほとんどの変換はコード生成中に UML モデル上で実行されるため、UML モデル内では見えません。

ランタイム フレームワーク

ランタイム フレームワークは、1 つの UML モデル（フレームワークの、UML の観点から見て適切な全側面を含む）および C++ ソース コードとして提供されます。UML モデルは［モデル ビュー］のセクションの **tor**（Tau Object Run-Time の短縮形）下にあります。

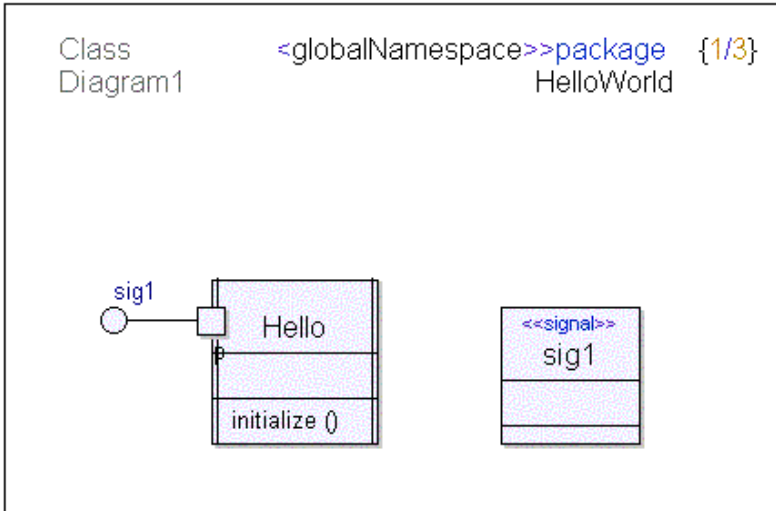


図 252: クラス図内のシグナル定義

前の例の「Hello」クラスを「Active」に設定しましょう。このモデルにシグナル「sig1」を追加して（1298 ページの図 252）、「Hello」クラスに状態機械を追加します（1299 ページの図 253）。

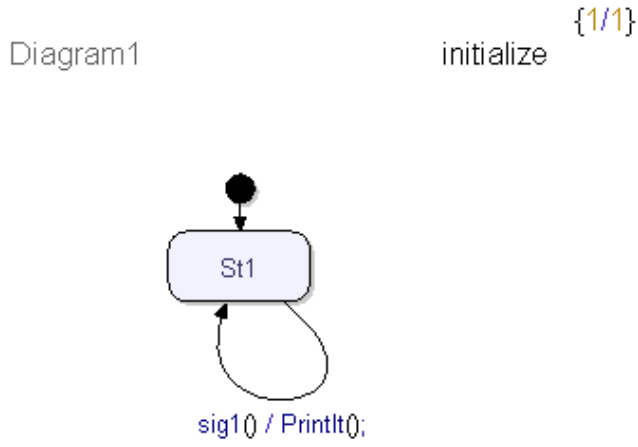


図 253: 状態機械の例

C++ コードの生成

このモデルからコードを生成する場合、モデル変換により作成される C++ コードの量は、上記のラウンドトリップの例よりも多くなります。`Hello.cpp` ファイルを開いて（**Result of C++ Code Generation** パッケージのファイル アーティファクトをダブルクリック）、生成後のコードを確認しましょう。

このファイルの特徴は、状態機械の C++ 表現である `Hello_initialize` クラスと、作成した単純な状態機械の遷移に対応する C++ コードを含む関数 `HelloWorld::Hello_initialize::trans_St1_sig1` です。

このプログラムを完成させるには、「PrintIt」操作を実装する必要があります。`main()` 関数は自動的に生成されます。より複雑なケースでは、`main()` 関数を自分で実装することもあり得ます。

「PrintIt」操作は以下のように定義できます。

```
void PrintIt() {
    std::cout << "Hello World¥n";
}
```

この関数の特徴は、`std::cout` を使用する点です。`std::cout` は `CppStdLibrary` アドインを有効にすると UML でも利用できます。C++ で使用する場合は `<iostream>` をインクルードします。

モデルから外部 C++ ヘッダーにアクセスするには、2 つの方法があります。

- Tau UML 環境から C++ API にアクセスする方法で説明されているように、API をインポートする。
- UML 環境でチェックされないターゲット コード 文（[[...]] で囲まれた文など）内で関数を使用する。

実環境で API を使用する場合はインポートによる手法がより望ましいのですが、単純なケースでは、ターゲット コードによるアプローチの方が簡単です。インライン コード アプローチの場合、make の過程でインクルードするファイルを定義しなければなりません。このためには、モデル内でファイルを表す UML アーティファクト間に << include >> 依存性を定義します。これで、インポートされたヘッダーに include 文が自動的に挿入されます。

この例では、1300 ページの図 254 に示すようにします。

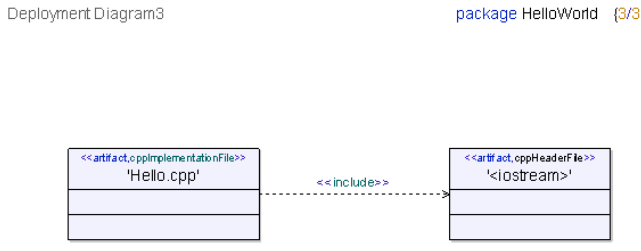


図 254:<< include >> 依存性の例

この段階で、アプリケーションのコードを再生成できます。［ビルド］->［構成のビルド］コマンドを選択して再生成を実行すると、プログラムのコンパイルおよびリンクも起動されます。この操作の結果、実行形式ファイルが作成されます。この実行形式ファイルは、C++ アプリケーション ジェネレータで生成されるファイル用に指定される［ターゲット ディレクトリ］で指定されたディレクトリにあります。

［ターゲット ディレクトリ］を指定していない場合、プロジェクト ファイルのあるディレクトリ下の、ビルドアーティファクトの名前の付いたディレクトリにファイルが生成されます。コンソール ウィンドウからアプリケーションを起動すると、おなじみの「Hello World!」の文字が画面に出力されます。

参照

ランタイム モデルの詳細については、第 43 章「C++ ランタイム フレームワーク」を参照してください。

C++ コード生成および C++ 構成要素へのマッピング時にサポートされる UML 概念の詳細については、第 41 章「C++ アプリケーション ジェネレータ リファレンス」を参照してください。

Tau UML 環境から C++ API にアクセスする

アプリケーションの一部を Tau で開発し、他の部分を他のツールを使用して開発するやり方は、一般的です。このような状況では、アプリケーション全体の定義を UML モデル内でアクセスできるようにする必要があります。Tau では、これを C/C++ のインポートを使用して実現しています。

C++ 開発環境での Tau 管理下の C++ コードの使用

Tau とともに Visual Studio .NET を使用する

Tau には Visual Studio .NET インテグレーションが含まれます。これにより、Tau から Visual Studio .NET プロジェクトを直接作成でき、Visual Studio .NET のソース コードと UML モデル間のナビゲートや、その他の機能が利用できるようになります。

注記

生成後の C++ コードは Visual Studio .NET 固有のものではありません。どの C++ コンパイラも Tau とともに使用できます。

ワークフロー

Tau と C++ 開発環境を一緒に使用した基本的なワークフローは以下のとおりです。

- Tau の UML を使用した、アプリケーションの分析フェーズを実行します。前段階として、IBM Rational DOORS による要求分析があるケースが主です。
- 分析モデルを UML の設計モデル向けに洗練させます。
- モデルから C++ コードを生成します。
- Tau で生成されたファイルを、使用したい C++ 開発環境に持ち込みます。
- ラウンドトリップ機能とモデル変換機能を使用して、C++ ファイルを更新します。
- C++ 開発環境の make およびデバッグ機能を使用して、アプリケーションを完成してデプロイします。

既存の C++ アプリケーションの Tau への移行

Tau は、既存の C++ アプリケーションから UML への移行をサポートしています。移行は、C/C++ のインポートの機能に基づいて行われます。つまり、C++ ファイルをインポートし、ラウンドトリップ機能とコード ジェネレータを使用して新しい C++ ファイルを作成します。

外部アプリケーションのヘッダー ファイルをインポートできます。ヘッダーファイル内の定義と振る舞いコードの両方をインポートできます。ただし、C++ インプリメンテーション ファイルはインポートできません。

C++ ファイルのインポート方法

基本的なワークフローは以下に示すとおりです。

1. ［ファイル］->［インポート］を選択し、インポート ウィザードを起動します。
2. インポートするファイルを選択します。
3. 必ず、［Import action code］オプションと［Generate artifacts］オプションの両方を有効にします。
4. 単純なケースでは、［今すぐインポート］を選択して、インポートを実行し、ウィザードを終了します。この場合はステップ 7 に進みます。インポート ウィザードで指定可能なオプション以外にもオプションを設定したい場合は、［今すぐインポート］を選択せずに、ウィザードを終了します。この場合は、モデルのルート レベルに「ImportedDefinitions」というパッケージが作成されます。
5. （オプション）「ImportedDefinitions」パッケージのプロパティ ページを開いて、［Add source file references to enable navigation from the UML model to the C++ source］チェックボックスの選択を解除します。これにより、インポート元のファイルにナビゲートできなくなりますが、モデルをベースに新しいファイルを再生成した際の混乱は少なくなります。このステップを実施しなければ、コード生成後も、元のファイルと生成ファイルへのソース参照が維持されます。
6. ［モデル ビュー］の［ImportedDefinitions］上のショートカット メニューから［C/C++ のインポート］を選択し、インポートを完了します。
7. この段階で、インポートされたファイルの定義を、インポータによって作成された「ImportedDefinitions」パッケージ内で確認できます。このパッケージには、インポートされたファイルを表すファイル アーティファクトと、C++ アプリケーションジェネレータを使用してファイルを再生成するためのビルドアーティファクトもあります。
8. （オプション）インポート後の定義をさまざまなパッケージに配置し直し、異なるファイルにマニフェストします。
9. 生成されたビルドアーティファクト上で［ビルド］->［生成］コマンドを使用して、インポートしたファイルを元に C++ コードを再生成します。デフォルトでは生成されたファイルは元のファイルと別の場所に配置されることに注意してください。これはビルド アーティファクトの［Target directory］オプションの設定によって変更できます。また、コード生成前に、プリプロセッサ設定など、ビルドに適した設定を行うことができます。

制限事項

移行のサポートには考慮すべき重要な制限事項がいくつかあります。このため、インポート前に C++ コードを修正したり、UML モデルから C++ コードを再度生成する前に UML モデルに変更を加えなければならない場合があります。

- ヘッダーファイルのみがインポートの対象です。インプリメンテーションファイルのインポートは、動作する場合もありますが、多くの場合 UML 中に重複した定義を発生させます。これは、異なるインプリメンテーションファイルが同じヘッダーファイルを含んでいるためです。インポータオプション［Do not import definitions from included header files］を使用してこの状況を回避できます。しかし、あらゆるケースにおいて、インプリメンテーションファイルのインポートは、ヘッダーファイルのインポートを正しく行った後で別の手順として行うべきです。
- インポートされるファイルは、Preprocessor（プリプロセッサ）で処理されます。したがって UML にインポートされるのは処理結果の C++ ファイルです。このため、マクロ、条件付きコンパイル、プリプロセッサ ディレクティブは処理後の UML モデルにはありません。
- C++ コードのコメントは、上の理由で、UML にインポートされません。

Tau で生成されたアプリケーションの実行のトレース

C++ コード生成時にトレース機能を追加できます。これを「コードに機能を備える」と言い、アプリケーションの実行時にメタ データを作成する行を生成します。このデータは、シーケンス図の実行を可視化するために必要な情報を Tau に提供するか、ログ ファイルに渡されます。

ワークフロー

1. ビルド アーティファクトの［ C++ Application Generator］プロパティで機能の装備を有効にする。
2. コードを生成する。
3. コードに行を追加してトレース機能を有効にする。以下の例のように、main 関数の最初の方に追加します。

```
int main() {
    //<GENERATED>
TOR_INSTRUMENTATION(TOR_GEN_Mxon1Vw89tiLYt_459WVk8GBGI);
    //</GENERATED>
tor::meta::EventManager* e =
tor::meta::EventManager::GetInstance();
e->createNewHostTracer();
e->createNewLogFile("C:/log");

    //Your code...
}
```

4. アプリケーションをビルドして実行する。Tau シーケンス図の実行をトレースする。

注記

トレース機能は独自のスレッドで実行されます。このためには、マルチスレッド アプリケーションをビルドしていなければなりません。

5. Visual Studio .NET インテグレーションで、ランタイム時の Tau Trace 機能 を制御します。

注記

Visual Studio .NET ランタイム コントロールを使用する場合、コードを手書きは不要です（ステップ 3）。

機能を備えた **API**

`EventManager` クラスのシングルトン オブジェクトによって、機能を備えた（Instrumented）API が提供されます。このオブジェクトにアクセスするには以下の呼び出しを行います。

```
tor::meta::EventManager* eventManager =
tor::meta::EventManager::GetInstance();
```

シーケンス図トレースの場合、ホスト トレーサを作成または削除します。

```
eventManager->createNewHostTracer();
eventManager->deleteAllHostTracers();
```

ログファイルの作成と削除

```
eventManager->createNewLogFile("C:/log");
eventManager->deleteAllLogFiles();
```

機能の装備の有効 / 無効化

```
eventManager->setActive(true);
eventManager->setActive(false);
```

Tau での **C++** サポートを使ってみる

C++ サポートを体験するには、Tau インストール ディレクトリにある C++ のサンプルを利用するのが一番の早道です。サンプルを用いて開発を始めるには、［ファイル］->［新規］を選択して開く、［新規］ダイアログの［テンプレート］タブを使用します。

このテンプレートには、多数の C++ のサンプルがあります。サンプルの名前には、言語およびコードジェネレータステレオタイプの接頭辞が含まれており、それぞれ区別できるようになっています。必要に応じてサンプルを選択し、ニーズに合わせて修正拡張可能な UML ／ C++ アプリケーションを実行してください。

40

C++ テキスト構文

C++ アプリケーション ジェネレータのラウンドトリップ機能とともに使用可能な C++ テキスト構文について説明します。

テキスト構文の一覧はオンライン ヘルプでのみ閲覧可能です。

41

C++ アプリケーション ジェネレータリファレンス

この章は、C++ アプリケーション ジェネレータのリファレンス ガイドです。

概要

C++ アプリケーション ジェネレータは、以下の機能を持っています。

- コードジェネレータと相互作用してスケジュールされるサービスを提供します。
- 必要に応じて元のモデルを拡張します。
- 過去にコード生成された C++ ファイルを解析して、ユーザーコーディングされた、または C++ アプリケーション ジェネレータによって生成されたコードをマージします。
- コードの構造を構築します。
- 最後にファイルにコードを書き出します。

C++ アプリケーション ジェネレータ アドイン

C++ アプリケーション ジェネレータからコード生成するためのモデルを作成するには、以下で説明するアドインを有効にする必要があります（これは［ツール］メニューから［カスタマイズ］ダイアログを選択して行うことができます）。これらのアドインは、新しい“C++ コード生成用 UML”プロジェクトを作成するとき自動的に有効になりますが、それ以外の場合は手動で有効にする必要があります。

CppGen

このアドインはプロファイル `TTDCppAppGen` をロードします。このプロファイルには、モデル変換と変換の間の依存関係の形で記述された C++ アプリケーション ジェネレータの UML 仕様があります。このプロファイルには、コード ジェネレータのオプションを記述するステレオタイプも含まれています。

通常、ユーザーはこのプロファイルを意識する必要はありませんが、上級ユーザーはこれを使用して UML モデルを C++ コードに翻訳する方法をカスタマイズできます。

CppTypes

このアドインはプロファイル `TTDCppPredefined` をロードします。このプロファイルには C++ の基本データ型（`int`、`bool`、`char` など）の UML 表現があります。モデル内で基本 C++ データ型を使用するために、これらのデータ型を参照できます。このプロファイルには、UML ではそのまま表現できない C++ 構造要素を表すために使われるステレオタイプもあります。たとえば、操作に適用し、インライン宣言 C++ 関数として生成することを指定できるステレオタイプ `<<inline>>` があります。

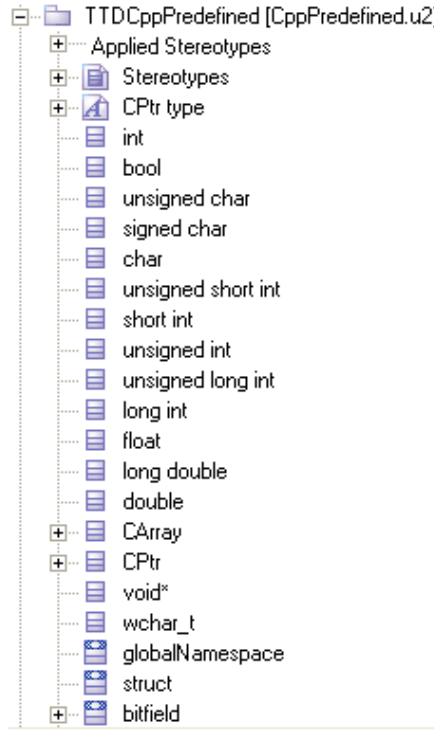


図 255 TTDCppPredefined プロファイルのパート

C++ アプリケーション ジェネレータの基本原理

C++ アプリケーション ジェネレータは、モデル変換およびその後の出力ステップ（逆構文解析とも呼ばれる）の実行によるコード生成という原理に基いています。モデル変換は、すべての non-trivialUML 構成要素を C++ 逆構文解析で扱える trivial 構成要素に翻訳することを目的としています。モデル変換は、各翻訳の後に新しい結果モデルを構築するのではなく「その場」で行われます。

C++ アプリケーション ジェネレータは、アプリケーション ビルダによって、通常の手順に従って個別の実行形式として開始されます。C++ アプリケーション ジェネレータは、入力としてアプリケーション ビルダから以下を受け取ります。

1. <<build>> アーティファクト。このアーティファクトには、通常のアプリケーション ビルダステレオタイプや C++ アプリケーション ジェネレータステレオタイプ（すべての翻訳オプションを含む）のタグ付き値（ターゲット ディレクトリなど）があります。
2. ビルドするエンティティのリスト。これらはユーザーが選択するエンティティ（“build selection”）かビルド アーティファクトが表現するエンティティです（“ビルドアーティファクト”または“構成のビルド”）。C++ アプリケーション ジェネレータはこれらのエンティティを繰り返し処理し、各エンティティに対応する <<<file>> アーティファクト（1310 ページの「モデルからファイルへのマッピング」を参照）を検索します。結果として得られた一連の <<file>> アーティファクトが、生成（または更新）される一連のターゲット C++ を表します。

ドキュメント構造

次の章では、C++ に翻訳できる UML サブセットを説明します。ここで説明されない UML 言語構成要素は翻訳時に無視されます。

サポートされている各 UML 構成要素に対して翻訳規則を示します。規則に例外がある場合はそれらも示します。

サポートされている各 UML 構成要素は、trivial か non-trivial に分類できます。trivial 構成要素が C++ 逆構文解析で直接 C++ にマッピングできるのに対して、non-trivial 構成要素は C++ 逆構文解析で C++ を生成する前に trivial 構成要素に翻訳する必要があります。ラウンドトリップは trivial 構成要素のみサポートしています。

ほとんどの翻訳規則に対して、具体的な UML と C++ 構文を使用して例を示します。それぞれの例は、単に翻訳規則を説明することを目的としており、生成されるコードを詳しく示すものではありません。

モデルからファイルへのマッピング

UML モデルには、生成された C++ ファイルにモデル要素をマッピングする方法の仕様を含むことができます。そのような仕様は、ソース ファイル（ヘッダーと実装ファイル）を表現するアーティファクト（Artifact）と、アーティファクトからソース ファイルに生成されることになるモデル要素への依存関係（Dependency）から、構成されます。依存関係は、定義済み「manifest」または「manifest implementation」ステレオタイプでステレオタイプ化されています。アーティファクトは、ファイルの名前と場所を指定する「path」属性を含む「file」ステレオタイプの派生形でステレオタイプ化されています。

明確にするために、アーティファクトは他のモデル要素とは別に、包含するパッケージのフォルダ内に示されます。

C++ コード生成の文脈においては、「file」ステレオタイプの 2 つの派生形が考えられます。

cppHeaderFile

C++ ヘッダー ファイルを示します。

cppImplementationFile

C++ 実装ファイルを示します。

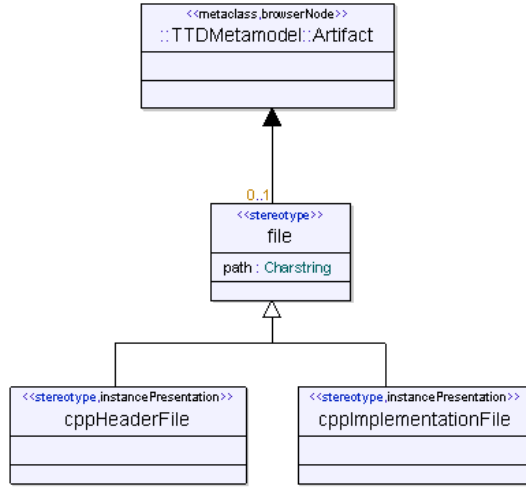


図 256 C++ 翻訳における当該ファイル アーティファクト

上記のとおりアーティファクトとステレオタイプを使用して、モデルからファイルへのマッピングを明示的に指定し、かつ、モデルからファイルへのデフォルトのマッピングを生成するオプションがオフならば、生成されるソース ファイルは指定されたマッピングで記述されているものだけになります。

各ヘッダー ファイルは、そのファイルで manifested と指定されたすべてのエンティティを含みます。

各実装ファイルは、そのファイルで manifested と指定されたすべてのエンティティと、mainfest implementation 関係を通して、そのファイルで manifested と指定されたエンティティの実装を含みます。

例 **487:** ——————————————————————————————

UML

```
<<cppHeaderFile (. path = "MyClass.h".)>> artifact
MyClassHeader
<<manifest>> dependency to MyClass;
```

```
    <<cppImplementationFile (. path = "MyClass.cpp".)>> artifact
    MyclassImpl <<'manifest implementation'>> dependency to
    MyClass;
    class MyClass {
        int foo(){
            return 14;
        }
    }
```

C++（ファイル "MyClass.h" 内）

```
    class MyClass {
      int foo();
    };
```

C++（ファイル "MyClass.cpp" 内）

```
    #include "MyClass.h"
    int MyClass::foo(){
      return 14;
    }
```

MyClass::foo は、UML で「インライン定義済み」ですが、その C++ 実装はヘッダー ファイルではなく実装ファイルに入ります。<<inline>> ステレオタイプは関数をインラインにする必要があることを指定するために使用でき、その場合その実装はヘッダー ファイルに入れられます。

ユーザーがビルドするために選択したエンティティが、C++ ヘッダーまたは実装ファイルに「manifest」されるよう指定されていない場合、それは翻訳されません。しかし、C++ アプリケーション ジェネレータにはモデルからファイルへのデフォルトのマッピングを作成するオプションがあります。このオプションはデフォルトでオンになっているので、ユーザーが明示的にモデルからファイルへのマッピングを作成する場合は、このオプションをオフにする必要があります。

モデルからファイルへのデフォルトのマッピングは以下のとおり生成されます。

1. 各パッケージに対して、ヘッダー ファイルと実装ファイルが生成されます。
2. パッケージに含まれる各構造化分類子（クラス、選択（Choice）、またインターフェイス）に対して、ヘッダー ファイルと実装ファイルが生成されます。
3. パッケージが所有する状態機械（スタンドアロン 状態機械）に対して、ヘッダー ファイルが生成されます。状態機械の実装を持てば、実装ファイルも生成されます。
4. モデルからファイルへのデフォルトのマッピング内の合成アーティファクトの名前は、ファイルが生成されるエンティティの名前と同じです。

注記

エンティティが既にファイル内で「manifest」されている（直接的に、または合成によって間接的に）場合、モデルからファイルへのデフォルトのマッピングはそのエンティティに対して別のファイル マッピングを作成しません。

モデル内のエンティティは、以下のようにファイルに翻訳されます。

1. ヘッダー ファイル アーティファクトが作られたエンティティは、ヘッダー ファイル に翻訳されます。これは、ヘッダー ファイル アーティファクトからエンティティに対して <<manifest>> 依存性を追加することによって行われます。
2. 実装ファイル アーティファクトが作られたエンティティは、実装ファイルに翻訳されます。これは、実装ファイル アーティファクトからエンティティに対する <<manifest implementation>> 依存性を追加することによって行われます。
3. 他のエンティティは、合成における所有者エンティティと同じファイルに翻訳されます。つまり、合成に関しては翻訳は再帰的に行われます。

生成されたファイルの名前は、一般的に `file::path` のタグ付き値で明示的に指定されます。指定されていない場合（たとえば、モデルからファイルへのデフォルトのマッピング）は、代わりにファイルを示すアーティファクトの名前が使用されます。

例 **488:** モデルからファイルへのデフォルトのマッピング

UML

```
package test {
    class S {
        Integer Operation1() {
            return 5;
        }
    }
    Integer var;
    extern int var2;
}
```

C++（ファイル "test.h" 内）

```
namespace test {
  extern tor::Integer var;
  extern int var2;
};
```

C++（ファイル "test.cpp" 内）

```
#include "test.h"
namespace test {
  tor::Integer var;
};
```

C++（ファイル "S.h" 内）

```
namespace test {
  class S {
    tor::Integer Operation1();
  };
};
```

C++（ファイル "S.cpp" 内）

```
#include "S.h"
tor::Integer test::S::Operation1() {
  return 5;
```

```
}
```

インクルード保護

生成された各 C++ ヘッダー ファイルは、条件付きコンパイル マクロの生成により、二重インクルードされないようになっています。

このマクロのデフォルトの名前は、ヘッダー ファイルを示すアーティファクトの GUID をベースにした一意な識別子になります。

例 **489:**

UML

```
class X {
}
```

C++ （ファイル "X.h" 内）

```
#ifndef GEN_Wr5SkVKjOm5LrSHaaV45sx9V
#define GEN_Wr5SkVKjOm5LrSHaaV45sx9V
class X {
};
#endif //GEN_Wr5SkVKjOm5LrSHaaV45sx9V
```

ヘッダーファイルを表すファイル アーティファクトのプロパティ「includeProtectionBegin」と「includeProtectionEnd」を設定することで、インクルード保護の形式を修正できます。プロパティ エディタでは、これらのプロパティはそれぞれ "Include Protection First String" および "Include Protection Last String" と表示されます。これらのプロパティはビルド アーティファクトにも設定が可能で、生成されるすべてのヘッダー ファイル（これらのプロパティを明示的に指定することによって設定を無効にするヘッダー ファイルを除く）のインクルード保護のフォーマットを指定します。

インクルード保護プロパティは、以下に示すコードの代替テキスト文字列です

コード	置換後の文字列
%%	1 つの「%」文字。
%f	ファイルのベース名（初期パスを削除）。「.」はアンダースコア「_」に変換。たとえば、「my/path/Xyzzy.h」は「Xyzzy_h」となる。
%F	「%f」と同様だが、小文字は大文字に変換される。たとえば、「my/path/Xyzzy.h」は「XYZZY_H」となる。
%g	ヘッダー ファイルを表すアーティファクトの GUID をベースとした文字列。

インクルード保護プロパティのデフォルト値：

「includeProtectionBegin」は「#ifndef GEN_%g¥n#define GEN_%g¥n」
「includeProtectionEnd」は「#endif // GEN_%g」

注記

"includeProtectionBegin" プロパティを指定する文字列には、最後に必ず復帰改行文字（¥n）を入れます。復帰改行文字がないと、生成されたコードが #define に付加され、コンパイル エラーとなります。

一般的な翻訳ルール

この章では、翻訳対象の各種エンティティに適用される一般的な翻訳ルールを説明します。

定義の名前

C++ 定義の名前は、その翻訳元の UML 定義の名前と同じです。

例外は、ansiName ステレオタイプによって、UML 定義に ANSI 名が指定されている場合で、その場合、C++ 定義には指定された ANSI 名が付けられます。

ANSI 名が指定されていないか、実際には ANSI ではない ANSI 名が指定されている場合、C++ 名は ANSI 名となるように調整されます。

上記の翻訳ルールから得られた C++ 名が有効な C++ 識別子ではない場合（キーワードの場合や、数字で始まる場合など）、デフォルトが“Name_”のユーザー設定接頭辞が付けられます。また、C++ 識別子では許されない文字（たとえば空白など）を含むために無効である場合、許されない文字は、その ASCII 文字列表現で置き換えられます。

例 **490:** 名前の翻訳

UML

```
class volatile {
}
<<ansiName(.name = "XYZ".)>> class '1åäö'{
}
<<ansiName(.name = "1åäö".)>> class C {
}
<<ansiName(.name = "1par".)>> class D {
}
part D 'an attribute';
```

C++

```
class Name_volatile {
};
class XYZ {
};
class tau_003100e500e400f6_tau {
};
class Name_1par {
};
D an_32_attribute;
```

型付けされた定義のタイプ

集約（**Aggregation**）種別の影響

型付けされた定義（たとえば Attribute、Parameters、Syntype などの型を持つ定義）の型参照は、C++ の対応する型参照に翻訳されます。

以下にその規則を示します。

参照型または共用型の型付けされた UML エンティティ（つまり、型付けされたエンティティの集約の種類が「reference」または「aggregation」の場合）は、以下のような C++ 定義に翻訳されます。

- エンティティの型がデータ型（datatype）（またはデータ型のシンタイプ（syntype））でなければ、C++ ポインタ型指定子を持つ。
- エンティティの型がデータ型（またはデータ型のシンタイプ）ならば、型指定子を持たない。

パート型の型付けされた UML エンティティ（つまり、型付けされたエンティティの集約の種類が“composition”の場合）は C++ では型指定子に翻訳されません（この定義は“value”型となります）。

この規則の**例外**は、パートの多重度がオプションか非シングルの場合です（多重度と集約種別を合わせたときの影響と比較してください）。

`TTDCppPredefined::CPtr` テンプレートのインスタンス化によって型指定される UML エンティティは、そのテンプレートのネストされているインスタンス化ごとに 1 つのポインタ型指定子を持つ C++ 定義に翻訳されます。

例 **491:** 型参照の翻訳 ———————————————————————————————

UML

```
class D {
    E a1;
    shared E a2;
    part E a3;
    CPtr<E> a4;
    CPtr<CPtr<E> > a5;
}
```

C++

```
class D {
  E* a1;
  E* a2;
  E a3;
  E* a4;
  E** a5;
};
```

———————————————————————————————————————

定義済みのデータ型

UML 定義済みデータ型の参照は、以下の表に従って C++ データ型の参照に翻訳されます。

UML 定義済みデータ型	C++ データ型	コメント
`Boolean`	`tor::Boolean`	「`bool`」と定義されます。
`Character`	`tor::Character`	ASCII コンパイルでは「unsigned char」、Unicode コンパイルでは「`wchar_t`」と定義されます。
`Integer`	`tor::Integer`	デフォルトで「int」と定義されますが、示す整数の大きさによって「`long int`」、「`long long int`」、または整数クラスとして定義することもできます。
`Natural`	`tor::Natural`	デフォルトで「unsigned int」と定義されますが、示す整数の大きさによって「`unsigned long int`」、「`unsigned long long int`」、または自然数クラスとして定義することもできます。
`Real`	`tor::Real`	デフォルトで「double」と定義されますが、示す実数の大きさによって「float」、「long double」、「long float」、または実数クラスとして定義することもできます。
`Charstring`	`tor::Charstring`	STL クラス `std::string`（Unicode 構成では `wstring`）を継承し、標準コレクション インターフェイス（以下で説明）を実装するクラスとして定義されます。
`String`	`tor::Charstring`	String は、UML では単に Charstring のシンタイプです。
`String`	`tor::String`	コレクションと多重度の影響を参照してください。

ターゲット C++ データ型（tor::…）は、TOR ヘッダー torTypes.h で定義されています。

基本的な C++ データ型はすべて「TTDCppPredefined」プロファイル パッケージを通して UML レベルで使用できます。これらのデータ型の使用は、対応する C++ 基本データ型に翻訳されます。

例 **492:** 定義済みデータ型の翻訳

UML

```
Boolean b;
```

```
Any a;
TTDCppPredefined::float f;
```

C++

```
tor::Boolean b;
tor::Any a;
float f;
```

コレクションと多重度の影響

定義済みの UML コレクション 型は String 型です。これへの参照は、「tor::String」型への参照に翻訳されます。「tor::String」型は、STL（デフォルトで std::vector）からコンテナ クラスを継承するクラスとして定義されています。

tor::String は直接定義済みの UML String 型のインターフェイスにほぼ対応するインターフェイスを持っています。このインターフェイスの実装は、ベース クラス コンテナから継承した関数を呼び出します。

例 **493:** 多重度 ---

1 より大きい多重度 を持つエンティティは、String 型で型指定されるエンティティと同じように翻訳されます。

UML

```
Character str[*];
String<Integer> istr;
class C {
  C [10] lst;
  C [1, 5..10] lst2;
}
```

C++

```
tor::String<tor::Character> str;
tor::String<tor::Integer> istr;
class C {
  tor::String<C*> lst;
  tor::String<C*> lst2;
};
```

注記

TTDCppPredefined パッケージには、直接 C/C++ に組み込まれた配列構成要素に対応する CArray データ型があります。パフォーマンス上の理由などから、こちらを String の代わりに使用することもできます。

多重度と集約種別を合わせたときの影響

上記のように、集約の種別が“composite”（パート）の属性は、C++ の value として表されます。したがって、パートとその所有者との生存期間の関係は、C++ 言語の規約が自動的にあてはめられます（所有者クラスのインスタンスが削除されると、そのクラスが所有するパートも削除されます）。

ただし、上記の振る舞いがあてはまるのは、パートが、非オプションの多重度 1 の場合（多重度が正確に 1）だけです。1 以外の多重度またはオプション多重度の場合、生存期間の関係について C++ 言語の規約が適用されません。したがって、この場合 C++ アプリケーション ジェネレータは生存期間が正しくなるように追加のコードを生成します。

注記

UML セマンティックでは、1 以外の多重度を持つパートは、コレクションに含まれるエンティティ（コレクション自身ではない）であり、そのコレクションは所有側のインスタンスと生存期間について関連を持つ必要があります。

この問題は、次の翻訳ルールで解決されます。

1 以外の多重度を持つパートは、`tor::String` 型で型付けされる属性に翻訳されます。ここで、要素型は翻訳された属性型へのポインタです。属性がデータ型（またはデータ型のシンタイプ）で型付けされている場合、型の翻訳はそのまま行われ、ポインタ指定子は追加されません。要素型が定義済みの value テンプレートのインスタンスである場合も同じことが起こります。

さらに、所有者側クラスのデストラクタにも、コードが追加されます。このコードはパート用に割り振られたスタック変数から構成されます。この変数は `tor::MultiDeleter` で型付けされます。このクラスは、デストラクタの中ですべての `String` 要素を削除します。このようにして、実行終了時にデストラクタを通過すると、String 要素が自動的に削除されることを保証しています。

注記

多重度が非形式（非形式多重度とカスタム コンテナ型を参照）と指定される場合、パート セマンティックを持つコンテナを実装するコードは追加されません。

例 **494:** ——————————————————————————————

UML

```
class C {
  part C [*] c_list;
  part String<C> c_list2;
}
```

C++

```
class C {
  tor::String<C*> c_list;
  tor::String<C*> c_list2;

  ~C() {
    tor::MultiDeleter<C> deleter_c_list(c_list);
    tor::MultiDeleter<C> deleter_c_list2(c_list2);
  }
```

```
};
```

オプションで多重度 1 を持つパートはポインタ型の属性に翻訳されます。属性がデータ型（またはデータ型のシンタイプ）で型指定されている場合、型の翻訳はポインタ指定子を追加せずにそのまま使用されます。

所有者側クラスのコンストラクタにもポインタを 0 に初期化するコードが追加され、デストラクタには、削除時に必ず 1 以外の多重度を持つパートに対応するポインタも削除するコードが追加されます。このデストラクタ コードは 1 以外の多重度を持つパートの場合と同じように生成されます。ただし、デストラクタの変数 はクラス tor::SingleDeleter で型付けされます。

例 **495:**

UML

```
class C {
  part D [0..1] opt_d;
}
```

C++

```
class C {
  D* opt_d;

  C() : opt_d(0) { }

  ~C() {
    tor::SingleDeleter<D> deleter_opt_d(opt_d);
  }
};
```

以下の表に集約の種別、型、および多重度との各種の組み合わせによる属性の翻訳の概要を示します。

多重度（Multiplicity）	パート（クラスで型指定）	参照（クラスで型指定）	データ型で型指定
0..1	ポインタ変数 コンストラクタで 0 に初期化 デストラクタで削除	ポインタ変数	値変数
1	Value 変数		
* 0..n n..m ここで、 m, n > 1	ポインタ変数のコンテナ デストラクタですべてのコンテナ要素を削除	ポインタ変数のコンテナ	値変数のコンテナ

初期インスタンス

属性がインスタンスの初期数を指定する場合、以下の表に従って初期インスタンスを作成するコードが所有者側クラスのコンストラクタに追加されます。

多重度／開始基数	パート（クラスで型指定）	参照（クラスで型指定）	データ型で型指定
`0..1 / 1`	属性を最初に作成されたインスタンスに設定します。	属性を NULL に設定します。	N/A

`m..n / m`
 	
ここで、
`m, n > 1`
	コンテナに m 個の初期作成インスタンスを挿入します。	コンテナに m 個の NULL を挿入します。	コンテナに m 個のデータ型のデフォルト リテラルを挿入します。

ユーザー定義コンストラクタがない場合、デフォルトのコンストラクタが作成されます。インスタンスの初期数が指定されていない場合でも多重度が閉じた範囲（下限と上限が等しい、つまり多重度として 1 つの値が指定されているケースも含め）を指定した場合には同じ翻訳ルールが適用されます。

例 **496:** 初期インスタンスの翻訳

UML

```
class C {
  part D [0..1]/1 opt_d;
  part D [4]/2 d2;
  D [1] d3;
  D [8] d4;
  Integer [7] i1;
}
```

C++

```
class C {
  D* opt_d;
  tor::String<D*> d2;
  D* d3;
  tor::String<D*> d4;
  tor::String<tor::Integer> i1;

  C() : opt_d(new D), d2(true), d3(0), d4(8), i1(7) {
    tor::initString<D>(d2, 2);
  }
  ~C() {
    if (opt_d)
      delete opt_d;
  }
};
```

注記

多重度が非形式（詳細な情報については非形式多重度とカスタム コンテナ型を参照）と指定される場合、コンテナに初期要素を追加するコンストラクタ コードは追加されません。

非形式多重度とカスタム コンテナ型

構造体型の多重度を非形式と指定することにより、デフォルトの String コンテナとは異なるコンテナ型を使用できます。

構造体型が非形式な多重度を持つ場合、C++ アプリケーション ジェネレータは、指定コンテナ型のセマンティックが不明と仮定します。したがって、コンテナをインスタンスの初期数で初期化するコードを生成しません（インスタンスの初期数が指定されている場合）。また、コンテナのすべてのインスタンスを削除するコードも生成しません（コンテナがパートである場合）。

例 **497:** カスタム コンテナ型を持つ属性の翻訳 ————————————————

UML

```
class A {
    public MyContainer<B> {[*]} m_b;
    public part MyContainer<B> {[*]} m_p / 2;
    public MyContainer<B> {[*]} m_r / 4;
}
```

C++

```
class A {
public:
    MyContainer<B*> m_b;
    MyContainer<B*> m_p;
    MyContainer<B*> m_r;
};
```

——

注記

A には m_p を B の 2 つの初期インスタンスで初期化するコンストラクタと、m_r を 4 つの NULL エントリで初期化するコンストラクタがありません。また、m_p のすべてのインスタンスを削除するデストラクタはありません。

コメント

コメントは、添付先のモデル要素が翻訳される直前に <MODEL> タグを追記の上で、C++ のソース コード コメントに翻訳されます。

モデル要素に複数のコメントが添付されている場合、対応する C++ コメントは空白行で区切られて生成されます。

例 **498:** ————————————————————————————————

UML

```
package P1 comment "This is package P1" comment "ver. 1" {
```

```
    class C {

      }
  }
```

C++

```
  /*<MODEL> This is package P1 */

  /*<MODEL> ver. 1 */
  namespace P1 {
    class C { ... };
  };
```

トランスレータ オプション Support roundtrip がオフになっている場合、<MODEL> マーカーは印刷されません。

外部定義

外部定義は、対応する C++ の「extern」宣言に翻訳されます。

したがって、「external」の意味は UML と C++ で同じです。エンティティは、モデル内で使用されているのに、外部で定義されている（たとえばライブラリで）場合、UML モデルで external として宣言する必要があります。

例 **499:** ---

UML

```
  DD extern var;
```

C++

```
  extern DD var;
```

注記

C++ アプリケーション ジェネレータは、対応する UML 定義に外部とマークされていない場合にも、C++ の「extern」定義を生成することがあります。たとえば、非定数属性は C++ 変数に翻訳され、マニフェストされるヘッダー ファイルで「extern」と宣言されます。詳細については、属性（Attribute）を参照してください。

非名前ベース参照

参照が名前で行われない場合（GUID またはポインタで行われる）、その参照は非名前ベースの参照に翻訳されます。参照が必ず前と同じターゲットと結び付けられるよう最小限の関連修飾子が追加されます。

この翻訳ルールは現在限定された一連の参照（エディタ内で一般にグラフィカル構文を使用して編集される参照）に適用されます。

合成されたエンティティのマーカー

生成されたファイルで、UML から C++ へ翻訳する際に合成されるエンティティに対応するすべての部分は、<GENERATED> マーカーで囲まれます。

例 **500:** 合成されたエンティティ

UML

```
class X {
  const Integer i = 14;
}
```

C++

```
class X {
  const tor::Integer i;
// <GENERATED>
  public:
  X() : i(14) { }
// </GENERATED>
};
```

<GENERATED> マーカーは、UML モデル内に直接対応するものがないにもかかわらず、翻訳時に合成されるコードがファイルのどの部分なのかを示しています。コードフォーマットの修正またはコメントの追加を除いて、このセクションのコードの変更は避けてください。

注記

見やすくするため、本書ではすべてのマーカーを例から除外しています。

トランスレータ オプション Support roundtrip がオフになっている場合、<GENERATED> マーカーは出力されません。

パッケージ（**Package**）

パッケージは名前空間に翻訳されます。

パッケージにステレオタイプ `TTDCppPredefined::globalNamespace` が適用されている場合、それは C++ のグローバル名前空間を示し、名前空間が生成されることはありません。

注記

このステレオタイプがモデル内の複数のパッケージに適用されている場合、これらすべてのパッケージ内の名前が一意であることを保証するのはユーザーの責任です。したがって、グローバル名前空間を示すには最大で 1 つのパッケージを使用することを推奨します。

例 **501:**

UML

```
package P1 {
  class C {

    }
}
<<globalNamespace>> package P2 {
    class C {

    }
}
```

C++

```
namespace P1 {
  class C {  };
};
class C { };
```

依存（**Dependency**）

依存は、UML モデルでいろいろな方法で使用でき、多くの場合非形式と解釈されます。しかし、生成された C++ コードに表示される依存には特殊な使い方があります。本章では、こういった依存の使い方を説明します。以下に示すカテゴリに含まれない依存は C++ に翻訳されません。

インクルード依存

定義済み <<include>> ステレオタイプでステレオタイプ化された依存は、依存の供給元がファイルを表すアーティファクトである場合、そのファイルの #include に翻訳されます。

依存のクライアントがファイルを表す別のアーティファクトである場合、そのファイルの先頭で #include ディレクティブが生成されます。

例 **502:**

UML

```
<<cppHeaderfile (. path = "sif.h".)>> artifact
SomeIncludeFile;

<<cppHeaderfile (. path = "..¥..¥include¥generic.h".)>>
artifact GenericTypes <<include>> dependency to
SomeIncludeFile
<<manifest>> dependency to X;

class X {
}

<<cppHeaderfile (. path = "IRunnable_interface.h".)>>
artifact IRun_ifc
<<manifest>> dependency to IRunnable;

interface IRunnable <<include>> dependency to GenericTypes {
```

```
}
```

C++ （ファイル “..¥..¥include¥generic.h” 内）

```
#include "sif.h"

class X {
};
```

C++ （ファイル “IRunnable_interface.h” 内）

```
#include "..¥..¥include¥generic.h"
UML_INTERFACE IRunnable {
};
```

必要な各 #include に対して手動で依存を指定する必要はありません。C++ アプリケーション ジェネレータは、どのファイルに 1 コンパイル ユニットで使用される定義が生成されるかを分析して自動的に必要な最低限の #include を計算します。しかし、次のような場合には、手動で #include 依存を指定する仕組みが必要です。

1. インライン C++ コードでヘッダー ファイルからの定義が使われている場合（このようなコードは UML レベルで解釈されないため）。
2. C++ 実装に外部ソース ファイルをインクルードするため。
3. C++ 実装ファイルにコンパイル済みヘッダーの使用を指定するため。

実装ファイルにコンパイル済みヘッダーを使用する方法を 1326 ページの例 503 に示します。

例 **503:** ———————————————————————————————

UML

```
<<cppHeaderfile (. path = "StdAfx.h", precompiled = true .)>>
artifact PrecompiledHeader;

<<cppImplementationFile (. path = "foo.cpp" .)>>
artifact foo_impl <<manifest>> dependency to foo,
<<include>> dependency to PrecompiledHeader;

bool foo(){
  return false;
}
```

C++ （ファイル “foo.cpp” 内）

```
#include "StdAfx.h"

bool foo(){
  return false;
}
```

アクセス依存

定義済み <<access>> ステレオタイプでステレオタイプ化された依存は、依存の供給元が <<'global namespace'>> ステレオタイプでステレオタイプ化されていないパッケージ P である場合、そのパッケージの「using namespace N」（ここで N は P の翻訳）に翻訳されます。依存の供給元が別の種類のエンティティである場合、「using」宣言が生成されます。

例 **504:**

UML

```
package P1 <<access>> dependency to P2
<<access>> dependency to P3::E {
  C var;
}

package P2 {
  class C { }
  class D { }
}

package P3 {
  class E { }
}
```

C++

```
namespace P1 {
  using namespace P2;
  using P3::E;

  C var;
};

namespace P2 {
  class C {};
  class D {};
};

namespace P3 {
  class E {};
};
```

インポート依存

定義済み <<import>> ステレオタイプでステレオタイプ化された依存は、依存の供給元が <<'global namespace'>> ステレオタイプでステレオタイプ化されていないパッケージである場合、供給元の パッケージに対応する名前空間に対して「using namespace」に翻訳されます。

注記

C++ アプリケーション ジェネレータでは、インポート依存は、アクセス依存の場合とまったく同じように扱われます。

例 **505:** ———————————————————————————————

UML

```
package P1 <<import>> dependency to P2 {
}
```

C++

```
namespace P1 {
  using namespace P2;
};
```

———————————————————————————————————————

フレンド依存

TTDCppPredefined の <<friend>> ステレオタイプでステレオタイプ化された依存は、依存のクライアントが 1 つのクラスか Choice(選択) である場合は、フレンド宣言に翻訳されます。

フレンド宣言は、C++ に翻訳されたクラスまたはユニオン内に配置されます。

例 **506:** ———————————————————————————————

UML

```
class Class1 <<friend>> dependency to foo(Integer) {}
void foo( Integer);
```

C++

```
void foo(tor::Integer par0);
class Class1 {
    friend void ::foo(tor::Integer par0);
};
```

———————————————————————————————————————

構造化分類子（**StructuredClassfier**）

構造化分類子は次のルールに従って翻訳されます。

Class はクラスに翻訳されるか、ステレオタイプ `TTDCppPredefined::struct` が適用されていれば構造体に翻訳されます。

インターフェイスは、抽象クラスに翻訳されます。つまり、そのすべてのメンバー関数が純仮想なクラスです。

`choice` は、共用体に翻訳されます。

他の構造化分類子は翻訳されません。

例 **507:** ———————————————————————————————

UML

```
class C { }
<<struct>> class S { }
interface I {
  int foo( char);
}
choice U1 {}
```

C++

```
class C { };
struct S {};
UML_INTERFACE I {
  int foo(char) = 0;
}
union U1 {};
```

UML_INTERFACE は、TOR ヘッダー torAnnotations.h で定義されているマクロです。これは class に定義されています。トランスレータ オプション Support roundtrip がオフの場合は、このマクロの代わりに class キーワードが使用されます。

属性（**Attribute**）

非定数属性は、次のルールに従って翻訳されます。定数属性については、定数属性を参照してください。

構造化分類子が所有する属性は、構造化分類子（StructuredClassfier）の翻訳であるクラス、構造体、または共用体のメンバー変数に翻訳されます。

パッケージが所有する属性は、パッケージの翻訳である名前空間がスコープとなる変数に翻訳されます。

パッケージ内の属性が「external」と宣言されていない限り、それは変数宣言（パッケージ用に生成されたヘッダー ファイル内）と変数定義（パッケージ用に生成されたソース ファイル内）の両方を返します。ただし、変数が C++ 実装ファイルで表現されている必要があります。ヘッダー ファイル内の変数宣言は、ヘッダー ファイルを複数の異なるコンパイル ユニットからリンク エラーを起こさずにインクルードできるよう常に external になります。

例 **508:** ———————————————————————————————————

UML

```
package test {
    class S {
        Boolean m_b;
    }
    Integer var;
}
```

C++（ファイル“test.h”内）

```
namespace test {
  class S {
    tor::Boolean m_b;
```

```
  };
  extern tor::Integer var;
};
```

C++ （ファイル “test.cpp” 内）

```
tor::Integer test::var;
```

属性デフォルト値

構造化分類子（StructuredClassfier）が所有する非静的属性（Attribute）のデフォルト値は、すべてのコンストラクタの初期化リストで対応するメンバー変数の初期化に翻訳されます。

パッケージが所有する属性（Attribute）または静的属性のデフォルト値は、対応する変数定義の初期化に翻訳されます。

例 **509:** ———————————————————————————————

UML

```
package test {
    class S {
        Boolean m_b = true;
        public S() {
           // Some initialization
        }
    }
    Integer var = 14;
}
```

C++ （ファイル test.h 内）

```
namespace test {
  external tor::Integer var;
};
```

C++ （ファイル test.cpp 内）

```
tor::Integer test::var = 14;
```

C++ （S.h 内）

```
namespace test {
  class S {
    tor::Boolean m_b;
    public:
    S();
  };
};
```

C++ （ファイル S.cpp 内）

```
test::S::S() :m_b(true) {
```

```
}
```

注記

構造化分類子の属性がデフォルト値を持つが、構造化分類子にコンストラクタが宣言されていない場合、メンバー変数の初期化を示すため、C++ クラス用に可視性が public の明示的なパラメータなしコンストラクタが作られます。

属性にデフォルト値が指定されていると代入（Assignment）を意味するので、この場合は代入の翻訳ルールも適用されます。

属性の可視性

構造化分類子（StructuredClassfier）が所有する属性の可視性は、対応するメンバー変数の対応する可視性に翻訳されます。

可視性が public、protected、および private の構文は、UML と C++ で同じです。ただし、UML には、package 可視性という追加の可視性があります。

C++ 翻訳では、package 可視性は public 可視性に翻訳されます。

UML で可視性が指定されていないと、デフォルトで public または private 可視性になります（どちらになるかはコンテキストに依存します）。いずれにしても翻訳では、可視性は C++ のルールで要求されていれば明示的に指定されます（C++ でもデフォルトの可視性はコンテキストに依存します）。たとえば、可視性が指定されていないクラス内で定義されている属性は C++ では private となります。

UML と C++ とのわずかな違いとして、UML ではすべての属性に可視性を指定するのに対して、C++ では次のように一度にすべてのメンバー変数に可視性を指定できます。後者のやり方の方がコードの可読性が良いと見なされているので、同じ可視性を持つ複数の連続した属性は、対応する C++ メンバー変数に対して「可視性宣言」を 1 つだけ与えます。

例 **510:**

UML

```
class S {
    public Boolean m_b;
    public Character m_c;
    protected int m_i;
    private bool m_bo;
    package char m_ch;
}
```

C++

```
class S {
public:
  tor::Boolean m_b;
  tor::Character m_c;
protected:
  int m_i;
private:
  bool m_bo;
```

```
public:
  char m_ch;
};
```

静的属性

構造化分類子（StructuredClassfier）が所有する静的属性は、対応する静的メンバー変数に翻訳されます。

例 **511:**

UML

```
class S {
    Boolean m_b1;
    static Boolean m_b2;
}
```

C++（S.h 内）

```
class S {
  tor::Boolean m_b1;
  static tor::Boolean m_b2;
};
```

C++（S.cpp 内）

```
tor::Boolean S::m_b2;
```

定数属性

構造化分類子（StructuredClassfier）が所有する定数属性は、メンバー定数に翻訳されます。

パッケージが所有する定数属性は、非メンバー定数に翻訳されます。

変数とは異なり、C++ 定数はデフォルトで内部リンクを持ちます。定数属性が外部リンクを持つ C++ 定数に翻訳されるよう指定するには、UML 属性で「external」プロパティを true に設定しておく必要があります。

例 **512:**

UML

```
class S {
    const Boolean m_b1 = true;
}
const double e = 2.78; // Internal constant
const Integer extern b = 6; // External constant
```

C++（ヘッダー ファイル内）

```
class S {
  const tor::Boolean m_b1;
  S() : m_b1(true) {}
};
const double e = 2.78;
extern const tor::Integer b;
```

C++ （実装ファイル内）

```
const tor::Integer b = 6;
```

注記

この翻訳ルールは、静的属性のルールと組み合わせて C++ で定数メンバーを与えることができます。

ビットフィールド

TTDCppPredefined::bitfield ステレオタイプが適用されている属性（Attribute）は、ビットフィールドに翻訳されます。

例 **513:**

UML

```
<<struct>> class S {
    int <<bitfield(. 2 .)>> m1;
    bool <<bitfield(. 1 .) m2;
}
```

C++

```
struct S {
  int m1 : 2;
  bool m2 : 1;
};
```

ビットフィールドのサイズはタグ付き値によって指定されます。

操作（**Operation**）

構造化分類子が所有する操作は、構造化分類子（StructuredClassfier）の翻訳であるクラス、構造体、または共用体のメンバー関数に翻訳されます。

パッケージが所有する属性は、パッケージの翻訳である名前空間がスコープとなる関数に翻訳されます。

注記

コンストラクタとデストラクタは UML と C++ の両方で特殊な操作です。したがって、これらは通常の操作と同じルールに従って翻訳されます（UML で「initialize」というコンストラクタは、C++ のルールに従って所有者クラス、構造体、または共用体と同じ名前になります）。

例 **514:** ———————————————————————————————

UML

```
package test {
    class S {
        int foo();
        S(bool); // Constructor
        initialize(); // Constructor
        finalize(); // Destructor
    }
    Integer open();
}
```

C++

```
namespace test {
  class S {
    int foo();
    S(bool);
    S();
    ~S();
  };
  tor::Integer open();
};
```

———————————————————————————————————————

操作パラメータ

操作のパラメータは、操作の翻訳である関数の仮パラメータに翻訳されます。

パラメータ方向「in/out」または「out」は、参照型のパラメータに翻訳されます。

パラメータ方向「return」の最初のパラメータの型（最大で 1 つ）は、操作の翻訳である関数の戻り型に翻訳されます。戻りパラメータがない場合、void 関数が生成されます。

例 **515:** ———————————————————————————————

UML

```
int foo(int p1, in int p2, inout int p3, out int p4);
```

C++

```
int foo(int p1, int p2, int& p3, int& p4);
```

型指定エンティティの一般ルール（1316 ページの「型付けされた定義のタイプ」と比較してください）はリターン パラメータにも当てはまります。

———————————————————————————————————————

パラメータのデフォルト値

操作パラメータのデフォルト値は、パラメータの翻訳である関数パラメータの対応するデフォルト値に翻訳されます。

注記

デフォルト値を持つパラメータがトレーリング パラメータではない場合、C++ ルールに従って対応する C++ パラメータにデフォルト値を与えることができません。

例 **516:**

UML

```
int foo(int p1 = 4, int p2, int p3 = 5);
MyClass bar();
part MyClass func();
```

C++

```
int foo(int p1, int p2, int p3 = 5);
MyClass* bar();
MyClass func();
```

デフォルト値は任意の式（Expression）です。

パラメータにデフォルト値が指定されていると代入（Assignment）を意味するので、この場合は代入の翻訳ルールも適用されます。

パラメータの多重度

パラメータに指定した多重度の影響は、一般的に任意の定義の場合と同じです（1318 ページの「コレクションと多重度の影響」を参照してください）。ただし、パラメータの多重度は、パラメータがオプション（その場合指定された多重度範囲に 0 が含まれる）であることを指定するためにも使用されます。

注記

オプション パラメータが最後のパラメータではない場合、C++ ルールに従って対応する C++ パラメータにデフォルト値を与えることができません。

例 **517:**

UML

```
void f(int p1[0..1], int p2, MyClass p3[0..1], MyClass
p4[0..*]);
```

C++

```
void f(int p1, int p2, MyClass* p3 = 0,
       String<MyClass*>* p4 = 0);
```

抽象操作

構造化分類子（StructuredClassfier）が所有する抽象操作は、構造化分類子の翻訳であるクラス、構造体、または共用体の純粋仮想メンバー関数に翻訳されます。

他の抽象操作は、通常の操作に翻訳されます。

例 **518:** ——————————————————————————————

UML

```
class S {
    abstract int f1();
}
class D : S {
    redefined int f1(); // Redefines S::f1
}
```

C++

```
class S {
  virtual int f1() = 0;
};
class D : public S {
  virtual int f1();
};
```

——————————————————————————————————————

仮想、再定義またはファイナライズした操作

構造化分類子（StructuredClassfier）が所有する仮想（virtual）、再定義（redefined）、またはファイナライズ（finalized）した操作は、構造化分類子の翻訳であるクラス、構造体、または共用体の仮想メンバー関数に翻訳されます。

したがって、操作が再定義またはファイナライズされたという指定は、C++ への翻訳では表示されません。

例 **519:** ——————————————————————————————

UML

```
class S {
    int f1();
    virtual int f2();
    virtual int f3();
    virtual int f4();
}
class D : S {
    int f1(); // Hides S::f1
    virtual int f2(); // Hides S::f2
    redefined int f3(); // Redefines S::f3
    finalized int f4(); // Redefines S::f4
}
```

C++

```
class S {
  int f1();
  virtual int f2();
  virtual int f3();
  virtual int f4();
};
class D : public S {
  int f1();
  virtual int f2();
  virtual int f3();
  virtual int f4();
};
```

例外指定

操作の例外指定（「throw」宣言）は、その操作の翻訳である関数の例外指定に翻訳されます。

例 **520:**

UML

```
void foo() throw Exc1, Exc2;
void <<TTDCppPredefined::noException>> bar();
```

C++

```
void foo() throw (Exc1, Exc2);
void bar() throw ();
```

1337 ページの例 520 に示すように操作に例外がないことを示すために使用できる `TTDCppPredefined::NoException` という特殊なステレオタイプがあります。

操作参照

操作参照を示すインターフェイスは、インターフェイスの「call」操作のシグニチャに対応する関数ポインタ型の typedef に翻訳されます。

「call」操作の呼び出しは、対応する関数ポインタの呼び出しに翻訳されます。

例 **521:**

UML

```
<<operationReference>> interface IFoo {
    Integer call(Boolean);
}

Integer foo(Boolean);

Integer main() {
    IFoo i = operation foo(Boolean);
    return i.call(true);
```

```
    }
```

C++

```
    typedef tor::Integer (*IFoo)(tor::Boolean);
    tor::Integer foo(Boolean);

    tor::Integer main() {
        IFoo i = foo;
        return i(true);
    }
```

汎化（Generalization）

2 つの構造化分類子（StructuredClassfier）間の汎化は、構造化分類子の翻訳であるクラス、構造体、または共用体間の継承に翻訳されます。

注記

`EventClasses`（たとえば Signals）には汎化の特殊翻訳ルールが適用されます。

操作間の汎化は翻訳されません。

デフォルトで継承は public で非仮想となりますが、ステレオタイプ `TTDCppPredefined::inheritanceVisibility` および／または `TTDCppPredefined::virtualInheritance` を適用することにより、private または protected 継承と virtual 継承を指定することもできます。

例 **522:**

UML

```
    class S {
    }
    class D : S {
    }
```

C++

```
    class S {
    };
    class D : public S {
    };
```

関連（Association）

名前のない一方向関連は、UML モデルの属性として示されます。したがってこのような属性の翻訳は属性（Attribute）のルールに従います。

例 **523:**

UML

```
class T {
    U b1;
}
class U {

}
association A {
    from T a2;
    from U a1;
    to a1;
    to a2;
}
association B {
    from T b2;
    to b1;
    to b2;
}
```

C++

```
class T {
  U* b1;
};
class U {
};
class A {
  T* a2;
  U* a1;
};
class B {
  T* b2;
};
```

シンタイプ（**Syntype**）

シンタイプは C++ の型定義に翻訳されます。

シンタイプのデータ制約は C++ に翻訳されません。

例 **524:**

UML

```
syntype X = Integer;
```

C++

```
typedef tor::Integer X;
```

データ型（**Datatype**）

データ型はプレーンな列挙型にマッピングされます。つまり、データ型に操作がある場合、それらは C++ に翻訳されません。

非形式定義（Informal Definition）

非形式定義は、定義テキストの逐語コピーに翻訳されます。

非形式定義が UML 定義の参照を含む場合、それは式が生成された C++ にコピーされる前に識別子の通常のルールに従って翻訳されます。

例 **525:** ———————————————————————————————

UML

```
[[int status = reinterpret_cast<#(Integer)>(STATUS);]];
```

C++

```
/*<TARGET>*/
int status = reinterpret_cast<tor::Integer>(STATUS);
/*</TARGET>*/
```

———————————————————————————————————————

式（Expression）

UML 式は、式の各部分を個別に扱うことによって C++ に翻訳されます。定数式は、翻訳時に評価されません（ほとんどの場合は可能ですが）。

ほとんどの UML 式の翻訳は次の表に示すとおり簡単です。

UML 式	C++ 式	UML 例	C++ 例
かっこ付きの式	かっこ付きの式	`(a+b)`	`(a+b)`
単項式	使用する C++ 演算子が指定された UML 演算子で決まる単項式（1342 ページの「定義済み UML 定義の参照」を参照）。	`not m_bOk` `++var`	`!m_bOK` `++var`
二項式	使用する C++ 演算子が指定された UML 演算子で決まる二項式（1342 ページの「定義済み UML 定義の参照」を参照）。		
this 式	'this' 式		
インデックス式	添字指定演算子（'[]'）	`coll[4]`	`coll[4]`
create 式	'new' 演算子	`new C(1,2)`	`new C(1,2)`

UML 式	C++ 式	UML 例	C++ 例
条件式	条件式	`b ? x1 : y1`	`b ? x1 : y1`
実数値	実数リテラル	`Real a = 3.14;`	`tor::Real a = 3.14;`
整数値	整数リテラル	`Integer a = 4;`	`tor::Integer a = 4;`

残りの式の翻訳は本章の後半で説明します。

識別子

識別子は、それがバインドされている定義の名前と同じ方法で翻訳されます。

このルールは、識別子が式の一部である場合と参照を表す場合の両方に適用されます（定義の名前と比較してください）。

例 **526:**

UML

```
<<ansiName(.name = "XYZ".)>> class '1åäö' {
    public void g();
}
'1åäö' volatile = new '1åäö'();
void foo(){
    volatile.g();
}
```

C++

```
class XYZ {
};
XYZ* Name_volatile = new XYZ();
void foo(){
  Name_volatile->g();
}
```

UML 識別子がスコープ修飾子を持つ場合、翻訳された識別子はスコープ修飾子を含みます。

注記

多少異なるスコープ ルールの結果、UML では必要ありませんが、C++ でスコープ修飾子が追加されることもあります。この逆のことが起きる場合あります。`<<globalNamespace>>` でステレオタイプ化したパッケージがスコープ修飾子内で参照されている場合、その参照は削除されます。

定義済み **UML** 定義の参照

上記のルールの例外は、識別子が定義済み UML エンティティ（定義済みパッケージに含まれるエンティティ）を示す場合です。

その理由は、定義済みパッケージが C++ に翻訳されないからです。参照されている定義が型の場合、翻訳は「定義済み型」のルールに従います。参照されている定義が操作の場合、翻訳は以下の表に従って行われます。

定義済み UML 操作	**C++ 演算子**
reinterpret_cast	reinterpret_cast
ASSERT	ASSERT It is assumed that this function (or macro) is defined in a file included by the user.
Boolean::'='	operator=(tor::Boolean, tor::Boolean)
Boolean::not	operator!(tor::Boolean)
Boolean::and Boolean::'&&'	operator&&(tor::Boolean, tor::Boolean)
Boolean::or Boolean::'||'	operator||(tor::Boolean, tor::Boolean)
Boolean::equal Boolean::'=='	operator==(tor::Boolean, tor::Boolean)
Boolean::'!='	operator!=(tor::Boolean ,tor::Boolean)
Boolean::xor	operator^(tor::Boolean, tor::Boolean)
Boolean::'=>'	tor::implies(tor::Boolean, tor::Boolean)
Integer::'='	operator=(tor::Integer, tor::Integer)
Integer::'-'	operator-(tor::Integer)
Integer::'-'	operator-(tor::Integer, tor::Integer)
Integer::'+'	operator+(tor::Integer)
Integer::'+'	operator+(tor::Integer, tor::Integer)
Integer::'++'	operator++(tor::Integer)
Integer::'--'	operator--(tor::Integer)
Integer::'*'	operator*(tor::Integer, tor::Integer)
Integer::'/'	operator/(tor::Integer, tor::Integer)
Integer::mod	tor::mod(tor::Integer, tor::Integer)
Integer::rem	operator%(tor::Integer, tor::Integer)
Integer::power	tor::power(tor::Integer, tor::Integer)

定義済み UML 操作	C++ 演算子
Integer::equal Integer::'=='	operator==(tor::Integer, tor::Integer)
Integer::'!='	operator!=(tor::Integer, tor::Integer)
Integer::'<'	operator<(tor::Integer, tor::Integer)
Integer::'>'	operator>(tor::Integer, tor::Integer)
Integer::'>='	operator>=(tor::Integer, tor::Integer)
Integer::'<='	operator<=(tor::Integer, tor::Integer)
Character::'='	operator=(char, char)
Character::'=='	operator==(tor::Character, tor::Character)
Character::'!='	operator!=(tor::Character, tor::Character)
Character::'<'	operator<(tor::Character, tor::Character)
Character::'>'	operator>(tor::Character, tor::Character)
Character::'<='	operator<=(tor::Character, tor::Character)
Character::'>='	operator>=(tor::Character, tor::Character)
Real::'='	operator=(tor::Real)
Real::'-'	operator-(tor::Real, tor::Real)
Real::'+'	operator+(tor::Real, tor::Real)
Real::'*'	operator*(tor::Real, tor::Real)
Real::'/'	operator/(tor::Real, tor::Real)
Real::'=='	operator==(tor::Real, tor::Real)
Real::'!='	operator!=(tor::Real, tor::Real)
Real::'<'	operator<(tor::Real, tor::Real)
Real::'>'	operator>(tor::Real, tor::Real)
Real::'<='	operator<=(tor::Real, tor::Real)
Real::'>='	operator>=(tor::Real, tor::Real)
is	tor::is(ARG arg)
as	tor::as(ARG arg)

注記

tor::is と tor::as の実装は、C++ の dynamic_cast 演算子をベースにしていることに注意してください。したがって、ARG は多相（polymorphic）型（つまり、少なくとも 1 つの仮想演算子を持つ）でなければなりません。現在、非多相（non-polymorphic）型に対するこれらの操作の使用を確認する構文チェックは行っていません。引数タイプが非多相（non-polymorphic）型の場合、空の仮想操作を追加して多相（polymorphic）型にできます。

参照されている定義が定数の場合、翻訳は以下の表に従って行われます。

定義済み UML 定数	C++ 定数
Real PLUS_INFINITY	PLUS_INFINITY この定数（またはマクロ）は、ユーザーがインクルードしたファイルで定義されていると仮定します。
Real MINUS_INFINITY	MINUS_INFINITY この定数（またはマクロ）は、ユーザーがインクルードしたファイルで定義されていると仮定します。

注記

参照が、定義済みパッケージの明示的な参照を含むスコープ修飾子を持つ場合（たとえば Predefined::Integer）、その参照は C++ 翻訳から削除されます。

定義済み C++ 定義の参照

上記の識別子翻訳ルールのもう 1 つの例外は、識別子が定義済み C++ エンティティ（定義済み TTDCppPredefined パッケージに含まれるエンティティ）を示す場合です。

TTDCppPredefined のほとんどの定義には通常の翻訳ルールが適用されます（ほとんどは C++ の対応する内蔵定義と同じ名前を持つため）。

例外は以下の表に示す一部の操作です。

TTDCppPredefined 操作	C++ 演算子	UML 例	C++ 例
CPtr::'new[]'	new[] operator	CPtr<int> p = CPtr<int>::'new[]'(9);	int* p = new int[9];
CPtr::'delete[]'	delete[] operator	CPtr<int>::'delete[]'(p);	delete p[];

これらの操作に加え、CPtr および CArray テンプレートの参照は、C++ ポインタと配列型指定子に翻訳されます（1341 ページの例 526 および 1316 ページの「集約（Aggregation）種別の影響」と比較してください）。

注記

参照が、TTDCppPredefined パッケージの明示的な参照を含むスコープ修飾子を持つ場合（たとえば TTDCppPredefined::int）、その参照は C++ 翻訳から削除されます。

外部 C/C++ 定義の参照

識別子翻訳ルールのもう 1 つの例外は、識別子が UML ではターゲット コードと異なる名前を持つ外部 C/C++ エンティティの参照を示す場合です（たとえば C/C++ のインポートによって UML モデルにインポートされている）。

UML は識別子に任意の名前を許すので、この例外の効果は名前がアポストロフィで囲まれている場合にそれを削除するだけです。

非形式式

非形式式は、式テキストの逐語コピーに翻訳されます。

非形式式が UML 定義の参照を含む場合、それは式が生成された C++ にコピーされる前に識別子（Identifier）の通常のルールに従って翻訳されます。

例 **527:** ———————————————————————————————

UML

```
int ptrSize = [[sizeof(void*)]];
int p2 = [[#(ptrSize) * 8 - 4]];
```

C++

```
int ptrSize = /*<TARGET>*/ sizeof(void*) /*</TARGET*>/;
int p2 = /*<TARGET>*/ ptrSize * 8 - 4 /*</TARGET*>/;
```

———————————————————————————————————————

呼び出し式

UML 呼び出し式は C++ 関数呼び出しに翻訳されます。

呼び出し式内の実引数は一般的に任意の式（Expression）です。

例 **528:** ———————————————————————————————

UML

```
void f(int p2, int p3, MyClass p4[0..1]);
void foo(){
    f(4, 3, new MyClass());
}
```

C++

```
void f(int p2, int p3,
       MyClass* p4 = 0);
```

```
void foo(){
  f(4, 3, new MyClass());
}
```

操作呼び出しは、実引数から仮操作パラメータへの明示的な代入を発生させます。したがって、この場合も代入（Assignment）の翻訳ルールが適用されます。

フィールド式

UML フィールド式は、C++ メンバー アクセス演算子「.」（オペランドのタイプが C++ の参照か値型の場合）または「->」（オペランドのタイプが C++ のポインタ型の場合）に翻訳されます。

代入

代入は二項式（Binary expression）として示され、そのように翻訳されます。ただし、代入には以下に示す追加翻訳ルールが適用されます。

UML 代入の左辺の翻訳が C++ の値型で、左辺の翻訳が C++ のポインタ型の場合、対応する C++ 代入の右辺にはアドレス参照演算子（&）が適用されます。同じように UML 代入の左辺の翻訳が C++ のポインタ型で、左辺の翻訳が C++ の値型の場合、対応する C++ 代入の右辺には間接演算子（*）が適用されます。

この翻訳ルールは、UML で認められるパートと参照の間の代入（あるいはその逆）が C++ でも認められることを保証します。

例 **529:**

UML

```
C c = new C();
part C cv;
cv = c;
C cref;
cref = cv;
```

C++

```
C* c = new C();
C cv;
cv = *c;
C* cref;
cref = &(cv);
```

この翻訳ルールは、暗黙的な代入にも適用されます。

1. デフォルト値の属性またはパラメータへの代入。
2. 実操作引数の、操作呼び出し内の仮操作パラメータへの代入。これは、実際のコンストラクタ引数の仮コンストラクタ パラメータへの代入にも適用されます。
3. 戻り値のリターン アクション内のパラメータへの代入。

Charstring と文字値

Charstring または文字値はデフォルトで _T マクロで囲まれています。このマクロは TOR ヘッダー「torTypes.h」で定義されており、ASCII 構成では何にも展開されず、Unicode などの wide-character 構成では L に展開されます。この目的は、生成されたコードが ASCII 構成と非 ASCII 構成の両方でコンパイルできるようにするためです。

翻訳オプション Enable non-ASCII compilation をオフにすると（デフォルトはオフ）Charstring および文字値に _T マクロを追加しないようにできます。このオプションがオフでも、wchar_t 型を使用してマルチ バイト文字列を使用できます。

例 **530:** ———————————————————————————————

UML

```
Charstring s = "zenith";
wchar_t[*] str = "angst";
Character c = 'x';
wchar_t cc = 'y';
```

C++

```
tor::Charstring s = tor::Charstring(_T("zenith"));
wchar_t str[] = L"angst";
tor::Character = _T('x');
wchar_t cc = L'y';
```

———————————————————————————————————————

1347 ページの例 530 のように、Charstring 値は `tor::Charstring` オブジェクト（_T マクロで囲んだ後）の構造に翻訳されます。その理由は、UML Charstring リテラルの型が、他の場合のように `const TCHAR*` ではなく、`tor::Charstring` である必要があるからです。

上記の翻訳ルールがないと、右辺と左辺が C++ で定数 TCHAR* で型指定されているバイナリ演算子の使用は、間違った演算子が適用されることになります。たとえば、比較は UML Charstring 変数値の内容間で行われずにポインタ値間で行われます。各 UML Charstring リテラルに対する `tor::Charstring` オブジェクトの明示的な構築でこの問題を解決できます。

TimerActive 式

TimerActive 式は、式が参照しているタイマーに対応するタイマー変数の `isActive` 関数の呼び出しに翻訳されます。

例 **531:** ———————————————————————————————

UML

```
timer T1 (Integer i);

void foo() {
  if (active(T1 (2))) {
    ...
  }
```

```
}
```

C++

```
class T1 : public virtual tor::TimerEvent {
public:
  typedef tor::TimerEvent theTimerEvent;
  tor::Integer i;
  T1(tor::Integer i) : i(i) {}

  static inline bool isTypeOf(tor::Event* e) {
    return tor::cast<T1*>(e) != 0;
  }

};

tor::TimerObject timer_T1(*this);

void foo() {
  if (timer_T1.isActive()) {
    ...
  }
}
```

注記

UML の TimerActive 式で使われている実引数の唯一の目的は、タイマーがアクティブかどうか調べるのにそのどの（オーバーロードしている可能性がある）バージョンを照会するかを確認することです。したがって、これら実引数は C++ 翻訳では表示されません。

テンプレート（**Template**）

UML 構造化分類子（StructuredClassfier）テンプレートは、対応する C++ クラス、構造体、または共用体テンプレートに翻訳されます。

例 **532: : ----------**

UML

```
template <class T, const int x >
class CSL
{
    T mv = x;
    T op1(int p1 = x);
}
```

C++

```
template <class T, int x >
class CSL {
  T mv;
  CSL() : mv(x) {}
  T op1(int p1 = x);
```

```
};
```

UML 操作テンプレートは C++ 関数テンプレートに翻訳されます。

例 **533:**

UML

```
template <class T, T x >
void foo(T p1 = x);
```

C++

```
template <class T, T x >
void foo(T p1 = x);
```

テンプレート定義内には、通常の翻訳ルールが適用されます。ただし、仮テンプレート パラメータへの参照を翻訳する場合は、テンプレート インスタンス化内の対応する実テンプレート パラメータのプロパティに関する知識に依存しているような翻訳ルールは、適用されません。これは、テンプレート定義からテンプレートがどのようにインスタンス化されるかを知ることができないためです。たとえば、UML 参照にポインタ宣言子を追加する翻訳ルールは、仮テンプレート パラメータへの参照の翻訳には適用されません（1316 ページの「集約（Aggregation）種別の影響」を参照）。仮テンプレート パラメータの ‘atleast’ 制約を指定して、対応する実テンプレート パラメータに制約を設定することにより、仮テンプレート パラメータへの参照をより正確に翻訳できます。

テンプレートのインスタンス化

UML テンプレートのインスタンス化は、UML テンプレートの翻訳である、C++ テンプレートのインスタンス化に翻訳されます。

実テンプレート型 パラメータはデータ型への通常参照です。従って、「集約（Aggregation）種別の影響」にあるポインタ宣言子を追加するルールが、このケースでも適用されます。参照ではなく "value" を使用して、参照型を実テンプレート パラメータとして渡すには、定義済みの Value テンプレートを使用できます。

例 **534:**

この例では 1348 ページの例 532 のテンプレート CSL をインスタンス化します。

UML

```
CSL<MyClass, 4> v1;
CSL<Value<MyClass>, 4> v2;
```

C++

```
CSL<MyClass*, 4> v1;
CSL<MyClass, 4> v2;
```

実テンプレート型パラメータのポインタ宣言子が追加されるのを避けるもう 1 つの方法として、対応する仮テンプレート パラメータの定義内で、これを参照型であると指定することです。これには、Atleast 制約を使用します。

Atleast 制約

仮テンプレート パラメータに 'atleast T' 制約がある場合（ここで T はデータ型）、テンプレート定義内のこの仮テンプレート パラメータへの参照は、T への参照と同じように処理されます。

例 **535:** ——————————————————————————————

仮テンプレート パラメータに 'atleast Any' 制約を指定すると、対応する実テンプレート パラメータは参照型の制約が課されます。これにより、C++ アプリケーション ジェネレータは、このような仮テンプレート パラメータへの参照を、参照型への参照の翻訳（つまり、ポインタ宣言子の追加）の場合と同じように翻訳できます。

UML

```
template <type Type atleast Any>
class buffer {
    public Type [*] variable;
    public <<inline>> buffer(Natural n) {
        for (Natural i = 0; i < n; i++) {
            Type t = new Type();
            variable.append(t);
        }
    }
}
```

C++

```
template <class Type>
class buffer {
    public:
    tor::String<Type *> variable;
    inline buffer(tor::Natural n) {
        for (tor::Natural i = 0; i < n; i++) {
            Type *t = new Type;
            variable.append(t);
        }
    }
};
```

`Type` に `'atleast Any'` 制約がないので、テンプレート定義内の `Type` への参照にポインタ宣言子は追加されません。

——————————————————————————————————————

‘atleast’ 制約はテンプレートのインスタンス化の翻訳に影響を与えることもあります。

注記

UML は、そのテンプレート構成要素に、ここで説明したものの他にもいくつかの機能があります。たとえば、テンプレート引数にプロトタイプを指定できます。これらの構成要素は C++ に対応するものがないので翻訳されません。

アクション（Action）

UML アクションは C++ 文に翻訳されます。

ほとんどの UML アクションの翻訳は次の表に示すとおり簡単です。

UML アクション	C++ 文	UML 例	C++ 例
削除アクション	delete 文	`delete T;`	`delete T;`
複合アクション	複合文	`{ v = v + 1; }`	`{ v = v + 1; }`
Continue アクション	continue 文		
ブレーク アクション	break 文		
If アクション	if 文		

アクションにラベルが添付されている場合、それはアクションの翻訳である文の直前で C++ ラベルに翻訳されます。

例 **536:**

UML

```
void foo(){
    LABEL: Integer v;
    goto LABEL;
}
```

C++

```
void foo(){
  LABEL:
  tor::Integer v;
  goto LABEL;
}
```

定義アクション

関連する定義が属性（Attribute）である定義アクションは、ローカル変数定義に翻訳されます。

例 **537:**

UML

```
void foo(){
    Integer v = 4;
}
```

C++

```
void foo(){
  tor::Integer v = 4;
}
```

関連する定義が属性ではない場合、定義アクションは翻訳されません。この場合、C++ アプリケーション ジェネレータは警告を出します。

```
Warning: Local definitions other than local variables are not
supported. The local definition 'name' will not be translated
to C++.
```

式アクション

式アクションの翻訳は、そのアクションに関連している式の種類によって異なります。

関連する空式がある式アクションは、空の C++ 文に翻訳されます。

非形式式に関連する式アクションは、非形式テキストに翻訳されます。

呼び出し式に関連する式アクションは、呼び出し式にセミコロン（;）を追加して翻訳されます。

関連する Create 式がある式アクションは、create 式にセミコロン（;）を追加して翻訳されます。

例 **538:** ---

UML

```
void foo(){
    ;
    [[##ifdef _WIN32]];
    open(true);
    new Integer;
    [[##endif]];
}
```

C++

```
void foo(){
  ;
#ifdef _WIN32
  open(true);
  new tor::Integer;
#endif
}
```

1352 ページの例 538 で示すように、「#」の後に「(」以外の文字が続く場合、インデントの通常の規則は無視されます（その場合、非形式テキストがプリプロセッサ ディレクティブを指定するため）。

Try アクション

関連する catch 句を持つ Try アクションは、catch 句を持つ try 文に翻訳されます。

Throw アクション

Throw アクションは throw 文に翻訳されます。

Throw アクションの式は、どの例外を発生させるかを指定します。以下のいずれか発生させます。

- 発生させる新しい例外を指定する
- すでに発生して検出した「再発生させる」例外を指定する。

例 **539:**

UML

```
void foo(){
    try {
        compute();
    }
    catch (tor::EMultiplicityOutOfRange e) {
        throw e;
    }
    catch (Any a) {
        throw a;
    }
    throw MyClass(14);
}
```

C++

```
void foo(){
  try {
    compute();
  }
  catch (tor::EMultiplicityOutOfRange e){
    throw e;
  }
  catch (...) {
    throw;
  }
  throw MyClass(14);
}
```

Loop アクション

Loop アクションは、while 文、do-while 文、または for 文に翻訳されます。

この 3 つの文はすべて同じ構成要素の変形でどの文が使用されるかは UML モデルで使用されている構文の種類によって決まります。

例 **540:** ———————————————————————————————

UML

```
Integer a = 0;
for (Integer i = 0; i < 10; i = i + 1) {
    a = a + 1;
}
while (a > 0)
    a = a - 1;
do {
    a = a + 1;
} while (a < 10);
```

C++

```
tor::Integer a = 0;
for (tor::Integer i = 0; i < 10; i = i + 1) {
    a = a + 1;
}
while (a > 0)
  a = a - 1;
do {
a= a+ 1;
} while (a < 10);
```

———————————————————————————————————————

停止アクション

停止アクションは、停止アクションが含まれる状態機械実装に対応する状態機械 クラスの「finish」関数の呼び出しに翻訳されます。

例 **541:** ———————————————————————————————

UML

```
stop;
```

C++

```
finish();
```

———————————————————————————————————————

NextState アクション

通常の **NextState** アクション

「通常」の NextState アクション（状態を指定する nextstate アクション）は、次の状態と指定された状態に対応する状態変数の「enter」関数の呼び出しに翻訳されます。

例 **542:** ———————————————————————————————

UML

```
   nextstate idle;
```

C++

```
   m_s_idle->enter();
```

---

nextstate アクションが「via」エントリ接続ポイントを指定する場合、エントリ接続ポイントの翻訳であるメンバー変数のアドレスが「enter」呼び出しの引数として渡されます。

例 **543:** ———————————————————————————————

UML

```
   nextstate idle via Cin;
```

C++

```
   m_s_idle -> enter(&m_s_Idle->m_sm->Cin);
```

History NextState アクション

履歴の NextState アクションは、nextstate アクションを含む状態機械実装のトップ リージョンの「`enterHistory`」関数の呼び出しに翻訳されます。

例 **544:** ———————————————————————————————

UML

```
   nextstate -;
```

C++

```
   theTopRegion::enterHistory();
```

Deep History NextState アクション

詳細な履歴の NextState アクションは、「`deepHistory`」フラグが true に設定され、nextstate アクションを含む状態機械実装のトップ リージョンの「`enterHistory`」関数の呼び出しに翻訳されます。

例 **545:** ———————————————————————————————

UML

```
   nextstate ^-;
```

C++

```
theTopRegion::enterHistory(true /* deepHistory */);
```

シグナル送信アクション

シグナル送信アクションは、動的に作成されたシグナル インスタンスを引数として「send」または「sendTo」関数の呼び出しに翻訳されます。

例 **546:** 明示的な宛先へのシグナル送信と宛先持たないシグナル送信

UML

```
output X.sig1;
output sig2(4);
```

C++

```
tor::sendTo(new sig1(), X);
m_owner->send(new sig2(4));
```

例 **547:** ポート経由のシグナル送信

UML

```
   port port1 out with PingSignal;
...
   output PingSignal via port1;
...
```

C++

```
...
   sendTo( new PingSignal(), &(m_owner->port1) );
...
```

シグナル送信アクションが、複数のシグナルの送信を指定する場合、送信される各シグナルに対して 1 つの「send」または「sendTo」呼び出しがあります。

分岐アクション

分岐アクションは、分岐回答ごとに 1 つの枝を持つ if 文に翻訳されます。

switch 文ではなく if 文に翻訳する理由は、C++ の switch 文は UML 分岐アクションより制限が多いからです。

例 **548:**

UML

```
switch (e) {
  case <10 : {
     i = 1;
```

```
        break;
      }
      case >=10 :
        break;
      default : {
        i = 3;
      }
    }
    switch (b < 8) {
      case true: {
        i = 1;
      }
      case false: {
        i = 0;
        break;
      }
    }
```

C++

```
    if (e < 10)
    {
      i = 1;
      goto GEN_9AnS2IFnpu0Lxq66AVx4TFzV;
    }
    else if (e >= 10)
      goto GEN_9AnS2IFnpu0Lxq66AVx4TFzV;
    else
    {
      i = 3;
    }
    GEN_9AnS2IFnpu0Lxq66AVx4TFzV:
    if ((b < 8) == true)
    {
      i = 1;
    }
    else if ((b < 8) == false)
    {
      i = 0;
      goto GEN_kpCCZVljYHVLaWD9DIlWN0DE;
    }
    GEN_kpCCZVljYHVLaWD9DIlWN0DE:;
```

「case」句内の break アクションは、制御を分岐アクションでの判断に続くアクションに移すために goto 文に置き換えられます。分岐アクションでの判断に続くアクションがない場合は、空のアクションを定義して挿入し、そのラベルへの goto 文を挿入します。

UML と C++ で case 文の処理内容に違いがあることに注意してください。C++ では特定のアクション後に break 文を挿入して、他の条件判断後の処理に制御が移らないようにする必要があります。しかし、UML では、そのような配慮は必要ありません。それでも、処理の後に break 文を入れておくことは、誤解を招かないという意味で有用ではあります。

リターン アクション

OperationBody に含まれるリターン アクションは、return 文に翻訳されます。

遷移に含まれるリターン アクションは、遷移を含む状態機械実装のトップ リージョンの「finish」関数の呼び出しに翻訳されます。

例 **549:** 遷移内のリターン アクション

UML

```
start {
    return;
}
```

C++

```
void initialTransition( ) {
    theTopRegion::finish();
}
```

リターン アクションが「via」終了接続ポイントを指定する場合、終了接続ポイントの翻訳であるメンバー変数のアドレスが「finish」呼び出しの引数として渡されます。

例 **550: Return via** 接続ポイント

終了接続ポイントを指定する遷移内のリターン アクション。

UML

```
start {
    return Cout;
}
```

C++

```
void initialTransition( ) {
    theTopRegion::finish(&Cout);
}
```

ジョイン アクション

ジョイン アクションは goto 文に翻訳されます。

1351 ページの例 536 のジョイン アクションが伴う簡単な例と比較してください。

ジョイン アクションが遷移に含まれていてその遷移内で定義されていないラベルを参照する場合は、上記ルールの例外です。異なる遷移は異なる C++ 関数に翻訳されるので、デフォルトの翻訳ルールはそのようなジョインを非ローカル goto に翻訳します。それは C++ では不正です。

遷移に含まれ、別の遷移に定義されているラベルを参照するジョイン アクションは、ラベルを含む遷移の翻訳である遷移関数の呼び出しに翻訳されます。呼び出しの直後に return 文が続きます。

非ローカル ジョインは、参照しているラベルがラベル遷移のラベルでない限りサポートされません。

例 **551:**

UML

```
void foo(Integer i) {
X:
  if (i == 1) {
    i = i + 1;
    join X;
  }
}
```

C++

```
void foo(tor::Integer i) {
X:
  if (i == 1) {
    i = i + 1;
    goto X;
  }
}
```

タイマー設定アクション

タイマー設定アクションは、設定しているタイマーに対応するタイマー変数の「set」関数の呼び出しに翻訳されます。

呼び出しの最初の引数が指定されているタイムアウト式です。タイマー設定アクションでタイムアウト式が指定されていない場合、タイマー定義で代わりに使用するデフォルトのタイムアウト式を指定する必要があます。呼び出しの 2 番目の引数は、タイマー定義の翻訳である `tor::TimerEvent` 派生クラスの動的に作成されたインスタンスです。実際のタイマー パラメータは、このインスタンスの作成で実際のコンストラクタ パラメータに翻訳されます。

例 **552:**

UML

```
timer T1 (Integer i);
timer T2 () = 15;

void foo() {
  set T1 (5) = now + 12;
  set T2;
}
```

C++

```
class T1 : public virtual tor::TimerEvent {
public:
  tor::Integer i;
  T1(tor::Integer i) : i(i) {}

  static inline bool isTypeOf(tor::Event* e) {
    return tor::cast<T1*>(e) != 0;
  }
};

class T2 : public virtual tor::TimerEvent {
public:
  static inline bool isTypeOf(tor::Event* e) {
    return tor::cast<T2*>(e) != 0;
  }
};

tor::TimerObject timer_T1;
tor::TimerObject timer_T2;

void foo() {
  timer_T1.set(tor::os::Time(tor::os::Time::now().to_double()
+ 12), new T1(5));
  timer_T2.set(tor::os::Time(tor::os::Time::now().to_double()
+ 15, true), new T2());
}
```

タイマー リセット アクション

タイマー リセット アクションは、リセットするタイマーに対応するタイマー変数の「reset」関数の呼び出しに翻訳されます。

例 **553:**

UML

```
timer T1 (Integer i);

void foo() {
  reset T1 (2);
}
```

C++

```
class T1 : public virtual tor::TimerEvent {
public:
  tor::Integer i;
  T1(tor::Integer i) : i(i) {}

  static inline bool isTypeOf(tor::Event* e) {
    return tor::cast<T1*>(e) != 0;
  }
};

tor::TimerObject timer_T1;

void foo() {
```

```
  timer_T1.reset();
}
```

注記

UML のタイマー リセット アクションで使われている実引数の唯一の目的は、タイマーのどの（オーバーロードしている可能性がある）バージョンをリセットするかを確認することです。したがって、これら実引数は C++ 翻訳では表示されません。

インターナル（**Internal**）

インターナル構成要素は標準 UML の一部ではなく、コンポーネントの簡便な設計をサポートするために導入された IBM Rational 固有の拡張です。これには、Bridge デザイン パターンとの類似点がいくつかありますが、まったく同じではありません。C++ 生成ではインターナルは無視されます。インターナルの目的は、UML による設計で Bridge パターンを使用することで実現できます（その後これを C++ アプリケーション ジェネレータが処理します）。

操作本体（**Operation Body**）

操作本体は C++ 関数本体に翻訳されます。

操作本体は、UML では操作が直接所有しますが、C++ ではそうではありません。なぜなら、C++ では通常、ヘッダー ファイルに実装を入れることが望ましくないためです（インライン操作を除く）。

例 **554:**

UML

```
class C {
  void foo(){
    return;
  }
  void bar();
  void <<inline>> inl() {
    return;
  }
}
void C::bar(){
  return;
}
```

C++（C.h ファイル内）

```
class C {
  void foo();
  void bar();
  inline void inl() {
    return;
  }
}
```

C++（C.cpp ファイル内）

```
void C::foo(){
  return;
}
void C::bar(){
  return;
}
```

シグナル（Signal）

シグナルは、ランタイム クラス `tor::Event` から継承する C++ クラスに翻訳されます。

例 **555:**

UML

```
signal sig_Ping;
```

C++

```
class sig_Ping : public virtual tor::Event {
  static inline bool isTypeOf(tor::Event* e) {
    return (tor::cast<sig_Ping*>(e) != 0);
  }
};
```

シグナル クラスの名前は、対応するシグナルの名前の翻訳です（1315 ページの「定義の名前」と比較してください）。しかし、シグナルはイベント クラスなので、同じ UML スコープ内に同じ名前を持つ複数のシグナルが存在する可能性があります（オーバーロード）。その場合、同じ名前を持つオーバーロードしているすべてのシグナルは、その名前の後にシグナル パラメータの型が追加されます。1363 ページの例 556 を参照してください。

注記

静的な「`isTypeOf`」関数は、イベントがクラスに対応するシグナルで動的に型指定されているかを決定します。この関数の実装は `tor::cast` 関数（これはデフォルトで `dynamic_cast` を使用）をベースにしています。

シグナル パラメータ

シグナルにパラメータがある場合、シグナルの翻訳であるクラスにはシグナル パラメータごとに 1 つのパラメータを持つコンストラクタが与えられます。さらに、シグナル パラメータごとの 1 つの public 属性も与えられます。コンストラクタ パラメータの名前、型、多重度およびクラス属性は、それらの生成元のシグナル パラメータと同一です。

シグナル パラメータが名前を持たない場合、コンストラクタ パラメータとクラス属性には名前「`parX`」が与えられます。ここで X は、シグナル パラメータのゼロ基準インデックスです。

例 **556:**

UML

```
signal sig_Ping(Charstring, Boolean b, Integer i = 3);
signal sig_Pong(Integer k);
signal sig_Pong(Boolean b);
```

C++

```
class sig_Ping : public virtual tor::Event {
public:
  tor::Charstring par0;
  tor::Boolean b;
  tor::Integer i;
  sig_Ping(tor::Charstring par0, tor::Boolean b, tor::Integer
i = 3) :
    par0 (par0), b (b), i (i) { }
};
class sig_Pong_Integer : public virtual tor::Event {
public:
  tor::Integer k;
  sig_Pong_Integer(tor::Integer k) :
    k (k) { }
};
class sig_Pong_Boolean : public virtual tor::Event {
public:
  tor::Boolean b;
  sig_Pong_Boolean(tor::Boolean k) :
    b (b) { }
};
```

注記

生成されたコンストラクタの初期化では、対応するクラス属性に各コンストラクタ パラメータの値が割り当てられます。

タイマー（**Timer**）

タイマーは、ランタイム クラス `tor::TimerEvent` から継承する C++ クラスに翻訳されます。

これはシグナル（Signal）の翻訳とまったく同じ方法で行われます。ただし、その他にタイマーを所有する定義の翻訳で `tor::TimerObject` で型指定されたタイマー変数の定義があります。

タイマー変数の名前はタイマーの名前と同じですが、接頭辞「timer_」が追加されます。

例 **557:**

UML

```
timer T (Integer a) = 15;
```

C++

```
class T : public virtual tor::TimerEvent {
public:
  tor::Integer a;
  T(tor::Integer a) :
    a (a) { }

  static inline bool isTypeOf(tor::Event* e) {
    return tor::cast<T*>(e) != 0;
  }

};

tor::TimerObject timer_T;
```

注記

デフォルト タイムアウト値（1363 ページの例 557 では 15）は、C++ 翻訳結果には直接表示されませんが、タイムアウト値を指定しないタイマー設定アクションの翻訳に使用されます。

状態機械（State Machine）

UML 状態機械は、特別な種類の操作としてモデル化されていますが、特殊な方法で翻訳されます。

ここで注目するのは状態機械がクラスに含まれる一般的なケースです。状態機械はクラスの外の別個の定義として（たとえばパッケージ内で）、あるいは合成状態のインライン 状態機械として宣言することもできます。これらの状態機械の翻訳方法に関しては、1374 ページの「合成状態を定義する状態機械」と比較対照してください。

UML モデルでは、状態機械（シグニチャ）と状態機械の実装（実装）を明確に区別しています。C++ 翻訳ではこの区別はそれほど明確ではありません。実際、状態機械のほとんどの興味深いプロパティはその実装の一部です。したがって、この章では状態機械実装の翻訳も取り上げます。

UML 状態機械は、ランタイム クラス「tor::StateMachine」とランタイム クラス「tor::TopRegion」から継承する C++ クラスに翻訳されます。

状態機械 クラスの名前は「C_SM」で、C は所有クラスの名前、SM 状態機械の名前です。

注記

単純な状態機械を通常の操作の実装として使用できます。そのような状態機械実装が状態を持たない場合、それは通常の操作本体として翻訳されます（開始遷移のアクションが操作本体（Operation Body）に入れられる）。

UML クラスに少なくとも 1 つのコンストラクタである状態機械が含まれる場合、対応する C++ クラスはランタイム クラス「tor::DispatchableClass」から継承します。

生成された Dispatchable クラスには次のメンバーが含まれます。

- 状態機械 クラス（分類子の振る舞い）で型指定されたメンバー変数「m_sm」。
- 仮想関数「tor::DispatchableClass::init」の再定義。その実装は状態機械 クラスのインスタンスを作成し、それを「m_sm」メンバー変数に格納します。また、継承した実装の呼び出しも行います。
  UML クラスがすでに「init」操作を持っている場合、もう 1 つ生成されることはありません。C++ アプリケーション ジェネレータは、既存の「init」操作が独自の方法で、Dispatchable クラスを初期化すると仮定します。

注記

「init」関数は静的構造体の初期化にも使用されます。

- 仮想関数「tor::DispatchableClass::start」の再定義。その実装は継承された実装を呼び出して、Dispatchable クラスの状態機械を開始します。また、実装は所有側の Dispatchable クラス インスタンスの一部である各アクティブ クラス インスタンスを開始します。つまり、アクティブ クラスのインスタンスが開始すると、すべての内包インスタンスも再帰的に開始します。
  UML クラスがすでに「start」操作を持っている場合、もう 1 つ生成されることはありません。C++ アプリケーション ジェネレータは、既存の「start」操作が独自の方法で状態機械（および内包インスタンス）を開始すると仮定します。
- 仮想関数「tor::DispatchableClass::receive」の再定義。その実装は単に継承した関数を呼び出します。
- 仮想関数「tor::DispatchableClass::getClassifierBehavior」の再定義。その実装は「m_sm」メンバー変数を返します。
- 「theDispatchableClass」と呼び出される「tor::DispatchableClass」の typedef。
- 状態機械 クラスと状態機械 クラスに含まれるすべての状態クラスのフレンド宣言。

生成された状態機械 クラスには次のメンバーが含まれます。

- 対応する Dispatchable クラスで型指定された「m_owner」というメンバー変数。
- コンストラクタ。このコンストラクタの実装は、「m_owner」メンバーを初期化します。
- 仮想デストラクタ。このデストラクタの実装は、「init」関数で作られたインスタンスを削除します（詳細については、1366 ページの例 558 を参照してください）。
- 「getDispatchableClass」関数。その実装は所有者変数を tor::DispatchableClass として返します。
- 継承した tor::StateMachine と tor::TopRegion の 2 つの型定義。これらの型定義は、「theStateMachine」および「theTopRegion」といいます。
- 仮想関数「tor::StateMachine::init」の再定義。その実装は引数「this」を使って継承された関数「addRegion」を呼び出します。

- 含まれるすべての状態クラスのフレンド宣言。

例 **558:**

UML

```
active class C {
    statemachine initialize {
       ...
    }
}
```

C++

```
class C : public virtual tor::DispatchableClass {
public:
  typedef tor::DispatchableClass theDispatchableClass;

  virtual void init() {
    m_sm = new C_initialize(this);
    theDispatchableClass::init();
  }
  virtual void start() {
    theDispatchableClass::start();
  }
  virtual bool receive(tor::Event* e) {
    return theDispatchableClass::receive(e);
  }
  virtual tor::StateMachine* getClassifierBehavior {
    return m_sm;
  }
  friend class C_initialize;
protected:
  tor::StateMachine* m_sm;
};
class C_initialize : public virtual tor::StateMachine,
                     public tor::TopRegion {
public:
  typedef tor::StateMachine theStateMachine;
  typedef tor::TopRegion theTopRegion;
  C_initialize(C* owner) : m_owner(owner) {}
  virtual ~C_initialize() {}
  virtual void init() {
    theStateMachine::addRegion(this);
  }
  virtual tor::DispatchableClass* getDispatchableClass() {
    return m_owner;
  }
protected:
  C* m_owner;
};
```

状態

状態はクラスに翻訳されます。クラスの名前は状態名の翻訳（1315 ページの「定義の名前」と比較してください）で、クラスはランタイム クラス「tor::State」から継承します。

状態クラスは、その状態から発生するすべてのトリガ付き遷移の実装を含みます。

各状態クラスは状態機械 クラスの「init」関数でインスタンス化され、得られた各状態インスタンスはその状態の対応するメンバー変数に格納されます。デストラクタには、割り当てられた状態を解放するため、対応する delete 文が入ります。

状態クラスのフレンド宣言が状態機械 クラスと所有者（Dispatchable）クラスの両方に追加されます。

これは、状態クラスの実装が状態機械 クラスの非公開データ、あるいは所有者（Dispatchable）クラスの非公開データをもアクセスしなければならない可能性があるためです（たとえば、シグナルデータを所有者クラスまたは状態機械実装内で定義されている変数に転送する場合、など）。

例 **559:** ———————————————————————————————————

UML

```
active class C {
    statemachine initialize {
        start {
            {
            }
            nextstate Idle;
        }
        state Idle;
    }
}
```

C++

```
class C : public virtual tor::DispatchableClass {
public:
  friend class C_initialize;
  friend class Idle;
  ...
};

class C_initialize : public virtual tor::StateMachine,
                     public tor::TopRegion {
public:
  ...

  virtual ~C_initialize() {
    ...
    if (m_s_Idle)
      delete m_s_Idle;
  }

  virtual void init() {
    ...
```

```
      m_s_Idle = new Idle(this, this);
    m_s_Idle->init();
    ...
  }

  class Idle : public tor::State {
  public:
    Idle(tor::Region* region, C_Initialize* owner ) :
      tor::State(region),
      m_owner(owner) {}
    tor::Dispatchable::EventAction execute(tor::Event* e);
  protected:
    C_Initialize* m_owner;
  };

protected:
...
  Idle* m_s_Idle;
public:
  friend class Idle;
};
```

簡単にするため、例では状態の翻訳に関係のないほとんどのものを省略しています。

状態は、合成状態になるため状態機械を含むか参照できます。そのような状態の翻訳については、1374 ページの「合成状態を定義する状態機械」で説明しています。

開始遷移

開始遷移は、開始遷移を含む状態機械実装の翻訳である状態機械 クラスの「`initialTransition`」関数に翻訳されます。

「`initialTransition`」の実装は開始遷移のアクションの翻訳です。この翻訳は他のアクション（Action）とまったく同じルールに従います。

開始遷移の翻訳の例は 1368 ページの「トリガ付き遷移」と比較してください。

トリガ付き遷移

開始遷移は、開始遷移を含む状態機械実装の翻訳である状態機械 クラスの「initialTransition」関数に翻訳されます。また、これは遷移をトリガできる状態に対応する各状態クラスの「execute」関数で if 文を与えます。

「遷移関数」の名前は以下のようになります。

1. 遷移がガードもアスタリスク トリガもない場合、

   ```
   trans_<StateNames>_<SignalClassNames>
   ```

2. それ以外の場合、

   ```
   trans_<StateNames>_< SignalClassNames >_<GUID>
   ```

<StateNames> は、遷移をトリガできるすべての状態の名前（下線で区切る）で、<SignalClassNames> は、遷移をトリガできる任意のシグナル（Signal）に対応するすべてのクラスの名前です。

<GUID> は、トリガ付き遷移の GUID です。

「execute」関数内の if 文は「isTypeOf」関数を使用してシグナル受信イベントの動的タイプが遷移のトリガで指定されている静的イベント タイプと一致するかテストします。一致する場合は現在の状態が残され（「leave」関数を呼び出して）遷移関数が呼び出されます。最後にイベントが削除され、「execute」関数は「Consumed」を返してイベントが消費されたことを示します。

例 **560:**

UML

```
statemachine initialize {
  Charstring 'from';
    start {
    {
      {
        count = 0;
      }
    }
    ^ destination.sig(strName);
    nextstate Idle;
  }
  state Idle;
  for state Idle;
    input sig('from') {
      {
        {
          Charstring name = strName;
          count = count + 1;
        }
      }
      ^ destination.sig(strName);
      nextstate Idle;
    }
}
```

C++

```
class C_initialize : public virtual tor::StateMachine,
                     public tor::TopRegion {
    protected:
    C* m_owner;

    class Idle : public tor::State {
        protected:
        C_initialize* m_owner;
        public:
        Idle(tor::Region* region, C_initialize* owner )
        : tor::State(region), m_owner(owner){

        }
        tor::Dispatchable::EventAction execute( tor::Event* e
) {
```

```
                if (sig::isTypeOf(e))
                    {
                    m_owner->from = tor::cast<sig*>(e)->sender;
                        {
                            leave();
                            m_owner->trans_Idle_sig();
                            delete e;
                            return tor::Dispatchable::Consumed;
                        }
                    }
                return tor::Dispatchable::NoMatch;
            }
        };

        private:
            tor::Charstring from;
        protected:
        Idle* m_s_Idle;
        public:
        virtual void initialTransition( ) {
            {
                {
                    m_owner->count = 0;
                }
            }
             tor::sendTo(new sig(m_owner->strName), m_owner-
    >destination);
          m_s_Idle->enter();
        }

        void trans_Idle_sig(tor::Event* theEvent) {
            {
                {
                    tor::Charstring name = m_owner->strName;
                    m_owner->count = m_owner->count + 1;
                }
            }
            m_s_Idle->enter();
        }
    };
```

簡単にするため、例ではトリガ付き遷移の翻訳に関係のないほとんどのものを省略しています。

遷移をトリガするイベントは、生成された「遷移関数」のパラメータとして得られます。このパラメータは「theEventand」で、ターゲット コードを使用して遷移のアクションからアクセスできます。

注記

受信シグナルの実引数は、UML シグナル送信アクションで参照されている変数に代入されます。

複数トリガ

トリガ付き遷移が複数のトリガを持つ場合、その遷移はこれらのトリガが指定する任意のイベントでトリガできます。

したがって、「execute」関数内の遷移の if 文条件は、イベント タイプをすべてのトリガのイベント タイプと比較し、どれかが一致すると true になるよう拡張されます。

例 **561:**

UML

```
     state Idle;
     for state Idle;
       input sig('from'), sig2(x) {
         {
...
         }
         nextstate Idle;
       }
   }
```

C++

```
   class Idle : public tor::State {
       protected:
       C_initialize* m_owner;
       public:
       Idle(tor::Region* region, C_initialize* owner )
       : tor::State(region), m_owner(owner){

       }
       tor::Dispatchable::EventAction execute( tor::Event* e ) {
           if (sig::isTypeOf(e) || sig2::isTypeOf(e)) {
               if (sig::isTypeOf(e)) {
                   m_owner->from = tor::cast<sig*>(e)->sender;
               }
               if (sig2::isTypeOf(e)) {
                   m_owner->x = tor::cast<sig2*>(e)->
               }
               leave();
               m_owner->trans_Idle_sig(e);
               delete e;
               return tor::Dispatchable::Consumed;
           }
           return tor::Dispatchable::NoMatch;
       }
   };
```

この場合、実際のシグナル引数を状態機械変数に移すコードは、イベント タイプの一致を調べる if 文内に入れられます。

複数トリガの特殊なケースとして「アスタリスク入力」があります。これは、受信可能なすべてのシグナルを表します。

アスタリスク トリガ（*）を持つトリガ付き遷移は、任意の種類の受信イベントでトリガされます。したがって、そのような遷移の if 文条件は単にイベント（e）が NULL ではないことを調べます（条件の指定トリガから派生した部分）。

アスタリスク トリガの if 文は、execute 関数の最後の if 文して挿入されます。

例 **562:** ———————————————————————————————

UML

```
...
  for state Idle;
    input * {
...
    }
}
```

C++

```
class Idle : public tor::State {
...
    tor::Dispatchable::EventAction execute(tor::Event* e ) {
        if (e)
        {
...
        }
        return tor::Dispatchable::NoMatch;
    }
};
```

———————————————————————————————————————

ガード

トリガ付き遷移のガードは、「execute」関数内の遷移の if 文内で追加の式に翻訳されます。この式は、トリガする遷移に対して true になる必要があります。

例 **563:** ———————————————————————————————

UML

```
...
  for state Idle;
    input sig('from') [myGuard == true]{
...
    }
}
```

C++

```
tor::Dispatchable::EventAction execute( tor::Event* e ) {
    if (sig::isTypeOf(e) && (myGuard == true))
        {
...
        }
```

———————————————————————————————————————

遷移がガードのみ持ちトリガがない場合、ガード式が true になり次第イベントを受信せずに遷移をトリガする必要があります。したがって、この場合、遷移の if 文はイベント変数（e）が NULL であり、ガード式が true であることのみを調べます。

例 **564:**

UML

```
...
  for state Idle;
    [myGuard == (x > 14)]{
...
    }
}
```

C++

```
tor::Dispatchable::EventAction execute( tor::Event* e ) {
    if (!e && (x > 14))
        {
...
        }
```

ラベル遷移

ラベル遷移は、ラベル遷移を含む状態機械実装の翻訳である状態機械 クラスの関数に翻訳されます。

「遷移関数」の名前は「trans_<LabelName>」で、<LabelName> は、ラベル遷移のラベルの名前です。

例 **565:**

UML

```
statemachine initialize {
  L:
  {
    count = 1;
  }
  ...
}
```

C++

```
class C_initialize : public virtual tor::StateMachine,
                     public tor::TopRegion {
  ...
  void trans_L() {
    count = 1;
  }
}
```

合成状態を定義する状態機械

クラスの「分類子の振る舞い」として定義されていない状態機械は、合成状態のサブ状態機械の定義に使用できます。そのような状態機械は、異なる合成状態から単独で参照できるスタンドアロン 状態機械として定義できます。あるいは、1 つの特定合成状態のインライン サブ状態機械として定義できます。

「合成状態」状態機械は、次の変更を加えた「分類子の振る舞い」状態機械（State Machine）と同じ方法で翻訳されます。

1. この場合、翻訳する Dispatchable クラスがありません。
2. 状態機械 クラスの名前は、インライン 状態機械が合成状態なら C_S_SM で、状態機械がスタンドアロン 状態機械なら SM です。S は合成状態の名前で、C はその状態を含む状態機械実装の翻訳である状態機械 クラスの名前です。SM は状態機械の名前です。
3. 「`m_owner`」属性は、所有元状態機械の翻訳であるクラスか（インライン状態機械の場合）、属性「`tor::DispatchableClass`」で（スタンドアローン状態機械の場合）型指定されます。
4. メンバーシップ属性「`m_ownerState`」が追加されます。この属性の型は、所有元ステートの翻訳であるクラスか（インライン状態機械の場合）、属性「`tor::State`」で（スタンドアローン状態機械の場合）クラスです。
5. インライン 状態機械の場合、「`getDispatchableClass`」関数の実装は以下のように変更されています。

   `"m_owner->getDispatchableClass()",`
   これは繰り返し Dispatchable クラスの所有者を得るためです。

合成状態の状態クラスはサブ状態機械 クラスをインスタンス化する「`init`」関数を含み、得られた状態機械 インスタンスは合成状態のクラス内のメンバー変数「`m_sm`」に格納されます。状態クラスのデストラクタには、割り当てられた状態機械を解放するため、対応する delete 文が入れられます。

状態クラスの「`init`」関数は、状態を所有する状態機械実装の状態機械 クラスの「`init`」関数内で呼び出されます。

例 **566:** インライン 状態機械を持つ合成状態

UML

```
statemachine initialize {
...
  state Idle : statemachine initialize {
  ...
  };
}
```

C++

```
class C_initialize_Idle_initialize : public virtual
tor::StateMachine,
                                     public tor::TopRegion {
```

```
  class Idle : public tor::State {
      protected:
      C_initialize* m_owner;
      public:
      Idle(tor::Region* region, C_initialize* owner);
      tor::Dispatchable::EventAction execute(tor::Event* e);

      virtual void init() {
         m_sm = new C_initialize_Idle_initialize(m_owner,
this);
         m_sm -> init();
      }

      C_initialize_Idle_initialize* m_sm;
      virtual tor::StateMachine* getStateMachine() {
        return m_sm;
      }
  };

...
};
```

例 **567:** 非インライン 状態機械を持つ合成状態

UML

```
statemachine initialize {
...
  state Idle : SM {
  ...
  };
}
statemachine SM {
...
}
```

C++

```
class Idle : public tor::State {
    protected:
    C_initialize* m_owner;
    public:
    Idle(tor::Region* region, C_initialize* owner);
    tor::Dispatchable::EventAction execute(tor::Event* e);

    virtual void init() {
       m_sm = new SM(m_owner->getDispatchableClass(), this);
       m_sm -> init();
    }

    SM* m_sm;
    virtual tor::StateMachine* getStateMachine() {
      return s_sm;
    }
};

class SM : public virtual tor::StateMachine,
```

```
            public tor::TopRegion {
...
}
```

これから、インライン 状態機械を持つ合成状態と非インライン 状態機械を持つ合成状態との唯一の違いは「init」関数の実装の小さな違いだけであることがわかります。

接続ポイント

接続ポイントは、接続ポイントを含む状態機械の翻訳である状態機械 クラスのメンバー変数に翻訳されます。

メンバー変数の名前は接続ポイント名が翻訳されたものです。接続ポイントの型は、エントリ接続ポイントの場合は tor::EntryPoint で、終了接続ポイントの場合は tor::ExitPoint です。

接続ポイント メンバー変数は、状態機械 クラスのコンストラクタで初期化されます。

例 **568:** 接続ポイント ——————————————————————————

UML

```
active class C : DispatchableClass {
    statemachine initialize {
...
        state Idle : statemachine initialize
        in Cin
        out Cout {
            start Cin {
...
            }
            start {
...
            }
            state WaitForSig;
            for state WaitForSig;
                input sig() {
                    return Cout;
                }
        };
        for state Idle;
            [Cout] {
                {
                    count = 14;
                }
                stop;
            }
    }
}
```

C++

```
class C_initialize : public virtual tor::StateMachine,
                     public tor::TopRegion {
    public:
    void trans_Idle_Cout() {
```

```
        m_owner->count = 14;
        finish();
      }
      virtual void exitPointTransitions( tor::ExitPoint* ep) {
        if(ep == &m_s_Idle -> m_sm -> Cout) {
         if (m_current)
            m_current->leave();
         trans_Idle_Cout();
         return;
      }
      protected:
      Idle* m_s_Idle;
};
class C_initialize_Idle_initialize : public virtual
tor::StateMachine,
                                     public tor::TopRegion {
    friend class WaitForSig;

    public:
    void trans_Cin() {
    ...
    }
    virtual void entryPointTransitions( tor::EntryPoint* ep)
{
      if (ep == &Cin) {
        trans_Cin();
        return;
      }
    }
    Idle* m_ownerState;
    public:
    tor::EntryPoint Cin;
    tor::ExitPoint Cout;
};
```

注記

接続ポイントの翻訳に関係ない詳細は、例では省略されています。

アーキテクチャ（**Architecture**）

クラスのアーキテクチャ（または内部構造）は、クラスの属性、クラスのポート、およびそれらをリンクするコネクタで構成されています。ここでは、これらの概念がどのように C++ コードに翻訳されるかを説明します。

属性

アクティブ クラスで型指定され、多重度＞ 1 である各属性に対して、属性を含むクラス内に protected 属性のユーティリティ メソッドが生成されます。このユーティリティ メソッドは、自動的に属性にインスタンスを追加し、そのインスタンスをオーナーと同じディスパッチャに追加し、初期化し、周囲のコネクタに基づく他の適切なインスタンスに接続し、最後に開始します。

例 **569:** ———————————————————————————————————

UML

```
active class Sys {
...
part Ping[*] p1;
...
part Pong[*] p2;
connector c1 from p1.pingP to p2.pingP;
...
}
```

C++

```
...
void Sys::p1_Insert( Ping * p ) {
  .p1.append( p );
  .addToCurrentDispatcher(p);
  .p->init();
  .for (int i = 1; i <= p2.length(); ++i) {
    c1.connect(&(p->pingP), &(p2[i]->pongP));
  .}
  .p->start();
}
...
```

———

アクティブ インスタンスを含む属性に使われるコレクション型の変更はサポートされていません。このため、Insert 操作が使用する繰り返し演算子を実装するデータ型が必要です。

定義済みテンプレート操作 `tor::insert<Any p>(Any obj)` の呼び出しは、前のルールで定義されている適切なユーティリティ メソッドの呼び出しに翻訳されます。

例 **570:** ———————————————————————————————————

UML

```
   part Ping[*] p1;
...
   tor::insert<p1>( new Ping() );
...
```

C++

```
...
   this->p1_Insert( new Ping() );
```

```
...
```

各アクティブ クラスに対してクラス内に、現在接続しているすべてのコネクタから自動的に切り離す、protected 属性のユーティリティ メソッドが生成されます。

例 **571:**

UML

```
active class Ping {
...
   port p1 out with PongSignal;
...
}
```

C++

```
...
void Ping::disconnect() {
      p1.disconnect();
}
...
```

コネクタ

各コネクタは、クラス内の `tor::Connector` で型指定される属性に翻訳されます。作成された属性の名前は、コネクタ名を翻訳したものです。

例 **572:**

UML

```
...
  connector c1 from part1.port1 with sig1 to part2.port2 with
sig2;
...
    }
}
```

C++

```
...
  tor::Connector c1;
...
```

コネクタは、ポートを接続する基盤として使用され、異なるオブジェクトの 2 つのポートを接続できる「`connect`」メソッドをサポートします。

注記

暗黙のコネクタ（非接続ポートのマッチングに基づいて作成されたコネクタ）はサポートしていません。

コネクタは、自らに関連する、伝送されるシグナル / サポートされているインターフェイスなどに対して制約を適用できます。これらの制約は、ツールが静的にチェックしますが、ランタイム時に強制はされません。ほとんどの場合、唯一の影響は、実行が速くなることです。しかし同時に、シグナルの異なるサブセットに基づく「コネクタ分割」もできなくなります。ポートに複数のコネクタが接続されている場合、シグナルは常に接続されている最初のコネクタに従います。

通常、送信されたシグナルの配信場所の曖昧性は排除したほうがよいでしょう。シグナル伝送をコネクタに依存しない場合、(シグナルが明示的に指定された受信者に送信される場合など)、曖昧性の問題は発生しません。コネクタを使用する場合は、'via <port>' 句を使用して、送信側で使用する出力ポートを指定します。これにより、複数出力ポートが存在した場合の曖昧性を減らすことができます。明示的な受信者も出力ポートも指定しないでシグナルを出力する場合、実行時の受信者に対するポートとコネクタ パスが正確に 1 つ存在しなければなりません。それ以外の場合、シグナルが失われます（または予想外の受信者に配信されます)。

ポート

各ポートは、クラス内の、ポートがサポートする各通信方向に対して `tor::Port` で型指定される属性に翻訳されます。作成される属性の名前は `<portName>_in` と `<portName>` です。ここで、`<portName>` はポート名が翻訳されたものです。

例 **573:** ———————————————————————————————

UML

```
...
  port p1 in with sig1 out with sig2;
...
```

C++

```
...
  tor::Port p1_in;
  tor::Port p1;
...
```

———————————————————————————————————————

振る舞いポートである各ポートに対して対応するクラスの `init` メソッドでポートの「`targetBehavior`」属性を `true` に初期化するコードが生成されます。

例 **574:** ———————————————————————————————

UML

```
active class Pong ... {
...
  bport p1 in with sig1 out with sig2;
...
}
```

C++

```
...
```

```
void Pong::init() {
    ...
    p1.targetBehavior = this;
    ...
}
...
```

静的構造体の初期化

静的パート（多重度で定義される）を含むクラスのインスタンスを動的に作成する場合、パートのインスタンスは自動的に作成されて初期化されます。つまり、クラスの `init` メソッド内にコードが生成されます。

静的多重度を持つアクティブ クラスで型指定される各属性に対して、静的インスタンスを初期化するコードが、`init` メソッド内に生成されます。

例 **575:**

UML

```
active class Top {
...
part Ping[1] ping;
part Pong[1] pong;
connector c1 from ping.port1 with sig1 to pong.port2 with
sig2
...
}
```

C++

```
...
class Top ... {
...
Ping ping;
Pong pong;
...
}
...
void Top::init() {
    m_sm = new Top_initialize(this);
    theDispatchableClass::init();
    addToCurrentDispatcher(&ping);
    addToCurrentDispatcher(&pong);
    ping.init();
    pong.init();
    c1.connect(&(ping.pingP),&(pong.pongP));
}
...
```

すでに UML クラスが「`init`」操作を持っている場合、もう 1 つ生成されることはありません。C++ アプリケーション ジェネレータは、既存の「`init`」操作が独自の方法で Dispatcher クラスを初期化すると仮定します。

注記

「init」関数は状態機械（State Machine）の初期化にも使用されます。

定義済み開始基数を持つアクティブ クラスで型指定される各属性に対して、開始インスタンスを初期化するコードが init メソッド内に生成されます。

例 **576:** ———————————————————————————————

UML

```
active class Top {
...
part Ping[*]/2 ping;
part Pong[*]/3 pong;
connector c1 from ping.port1 with sig1 to pong.port2 with
sig2
...
}
```

C++

```
...
class Top ... {
...
tor::String<Ping> ping;
tor::String<Pong> pong;
...
}
...
void Top::init() {
    m_sm = new Top_initialize(this);
    theDispatchableClass::init();
    tor::addToDispatcher<Ping>(this, ping, 2);
    tor::addToDispatcher<Pong>(this, pong, 2);
    tor::init(ping, 2);
    tor::init(pong, 2);
    for (int i2 = 1; i2 <= 2; i2++) {
      for (int i3 = 1; i3 <= 3; i3++) {
        c1.connect( &(ping[i2]->pingP), &(pong[i3]->pongP) );
      }
    }
...
```

———————————————————————————————————————

`tor::addToDispatcher`, `tor::init` と `tor::start` は、TOR ランタイム ライブラリのユーティリティ関数です。これらは、読み易さを向上するため for-loop の代わりに使用されます。

注記

静的初期化（開始基数 > 0 または定数多重度 > 1）は、作成されたクラスがパラメータなしのコンストラクタを持つ場合にのみサポートされます。

インスタンスの切断

ポートを持つアクティブ インスタンスを削除すると、それはアーキテクチャから自動的に削除されます。たとえば、接続されているコネクタから切断されます。これはランタイム フレームワークが処理します。

パッケージ TTDCppPredefined

TTDCppPredefined パッケージには、C++ 言語構成要素と定義済みタイプの UML 表現があります。このパッケージは、C/C++ のインポートと C++ アプリケーション ジェネレータが使用しますが、これらのツールが使われない場合にユーザーが使用することもできます。

定義済みのタイプ

TTDCppPredefined パッケージは、C++ データ型を示す以下の UML データ型を含みます。

UML データ型	定義済み UML データ型	C/C++ 基本型
int	Integer	signed int, int
unsigned int	Integer	unsigned int, unsigned
long int	Integer	signed long int, signed long, long int, long
unsigned long int	Integer	unsigned long int, unsigned long
short int	Integer	signed short int, signed short, short int, short
unsigned short int	Integer	unsigned short int, unsigned short
long long int	Integer	signed long long int, signed long long, long long int, long long
unsigned long long int	Integer	unsigned long long int, unsigned long long
char	Character	char
signed char	Character	signed char
unsigned char	Character	unsigned char
wchar_t	Character	wchar_t

UML データ型	定義済み UML データ型	C/C++ 基本型
float	Real	float
double	Real	double long double
bool	Boolean	bool

ステレオタイプ

TTDCppPredefined パッケージは、簡素な UML では直接表現できない C++ 構成要素を示す次の UML ステレオタイプを含みます。

'globalNamespace' extends Package

パッケージに対する明示的な名前空間を生成してはならないことを指定します。代わりにパッケージは C++ の暗黙グローバル名前空間を示します。

'struct' extends Class

Class はクラスではなく構造体クラスに翻訳する必要があることを指定します。

'inheritanceVisibility' extends Generalization

Generalization は、public 以外（private または protected）の継承に翻訳する必要があることを指定する属性を含みます。

'virtualInheritance' extends Generalization

Generalization は、仮想継承に翻訳する必要があることを指定する属性を含みます。

'inline' extends Operation

関数がインラインとして宣言されていることを指定します。

'bitfield' extends Attribute

ビットフィールドに使用するビット数を指定する整数属性を含みます。

'CppReference' extends StructuralFeature

属性またはパラメータが C++ 参照 (&) であることを指定します。このステレオタイプは、主として、操作の戻り値が参照を返す C++ 関数にマップされる必要があることを指定するために使用されます。

'auto' extends StructuralFeature

属性またはパラメータには C++ の `auto` 記憶指定子が必要であることを指定します。

'register' extends StructuralFeature

属性またはパラメータには C++ の `register` 記憶指定子が必要であることを指定します。

'mutable' extends Attribute

属性を `mutable` として宣言する必要があることを指定します。

'volatile' extends StructuralFeature, Operation

C++ の `volatile` 指定子を表現します。

'explicit' extends Operation

構築子 (コンストラクタ) を C++ で `explicit` としてマークする必要があることを指定します。

'export' extends Signature

テンプレート定義を、`export` キーワードを使用して C++ にエクスポートされたとマークする必要があることを指定します。

'friend' extends Dependency

このステレオタイプは、UML において C++ フレンド宣言を表現するのに使用します。詳細はフレンド依存 を参照してください。

'__declspec' extends Definition

一部の C/C++ コンパイラで言語仕様の拡張としてサポートされる `__declspe` キーワードを表現します。

'manifest implementation' inherits manifest

このステレオタイプは、C++ 実装ファイルを示すファイル アーティファクトがサプライヤ定義のすべての実装側面を表現することを指定するために通常の 'manifest' ステレオタイプの代わりに使用できます。したがって、クラスの各操作本体に対してそれらを同じ C++ 実装に入れる必要があることを指定する代わりに、ファイル アーティファクトからクラスへこのステレオタイプでステレオタイプ化された 1 つの依存性だけでそれを指定できます。

翻訳オプション

C++ アプリケーション ジェネレータのオプションは、C++ アプリケーション ジェネレータプロファイル パッケージに含まれる <<C++ Application Generator>> ステレオタイプの属性のタグ付き値として示されます。以下のセクションにオプションの概要を示します。オプションごとに、対応するステレオタイプ属性を示します。

名前変換オプション

UML 名を生成済み C++ コードで使用できない場合に、コード ジェネレータが行う名前変換に関するオプションです。

Name prefix

`nameManglingOptions = NameManglingOptions (. namePrefix .) (string)`

名前を C++ 命名規則に従って有効にするために定義にどの接頭辞を与えるかを指定します。デフォルトは ”`Name_`” です。

Enable non-ASCII compilation

`enableUnicode (boolean)`

これを true に設定すると、すべての文字と Charstring 値は _T マクロで囲まれます。これにより、生成されたコードを ASCII および非 ASCII（たとえば Unicode）構成でコンパイルできます。

モデルからファイルへのデフォルトのマッピング オプション

コード ジェネレータが使用するデフォルトのモデルからファイルへのマッピングに関するオプションです。

Use default model-to-file mapping

`generateDefaultFileMapping (boolean)`

これを true に設定すると C++ アプリケーション ジェネレータは、デフォルトでモデルからファイルへのマッピングに使用する UML 仕様を生成します。

Use precompiled header

`fileMappingOptions = FileMappingOptions (. precompiledHeader .) (string)`

モデルからファイルへのデフォルトのマッピングで生成されるすべての実装ファイルに使用する、コンパイル済みヘッダー ファイルを指定します。

コード フォーマット オプション

生成済みコードのフォーマットに関するオプションです。

Indentation

```
codeFormattingOptions = CodeFormattingOptions (. indentSize .)
(natural)
```

各インデント レベルに使用する空白の数を指定します。

コード編成オプション

このオプションは生成コードの編成に関連しています。

Sorting of bodies

```
sortingOptions = CodeOrganizationOptions (. sortOperationBodies
.) (SortOperationBodiesKind)
```

このオプションは生成コード内の操作本体の編成を制御します。選択肢は、アルファベット順のソート、またはモデル内の操作の定義の順です。後者の場合、合成操作は、非合成操作の後に配置されます。

インラインの操作本体には影響しません。

Grouping of members

```
sortingOptions = CodeOrganizationOptions (. sortMembers .)
(SortMembersKind)
```

このオプションは、メンバー定義のグループ化について制御します。選択肢は、モデル内の順序か、可視性に従った順序です。モデル内の順序の場合、合成メンバーは、非合成メンバーの後に配置されます。可視性に従った順序の場合は、public メンバー、protected メンバー、private メンバーの順で配置されます。

Group transition if-statements according to trigger definition scope

```
sortingOptions = CodeOrganizationOptions (.
sortTransitionIfStatements .) (Boolean)
```

このオプションを有効にすると、遷移 if 文が遷移とリガの定義スコープにしたがってグループ化されます。遷移に対してよりローカルなスコープがより先に配置されます。

Enable COM agents

```
enableCOMAgents (boolean)
```

このオプションをオンにすると、COM オブジェクトとして実装されたエージェントはコード生成時に有効になります。COM の使用にはパフォーマンス上のオーバーヘッドがあるため、COM エージェントをコード ジェネレータのカスタマイズに使用しないならこのオプションはオンにしないことを推奨します。

Support roundtrip
`supportRoundtrip (boolean)`

これを true に設定すると、生成されたファイルにはファイル内の変更を「ラウンド トリップ」して UML モデルに戻すことができるいくつかの注釈（コメントとマクロ）を含みます。このオプションは、生成されたファイルで変更を行う予定がなければオフにできます。オフにすると、コード ジェネレータの効率がよくなり（既存ファイルの解析が不要になり単純に上書きできる）、生成済みコードに注釈が少なくなります。

Time unit
`timeUnit (TimeUnit)`

生成されるアプリケーションで使用する時間の単位を指定します。時間単位を 秒、ミリ秒、マイクロ秒、およびナノ秒から選択します。別の時間単位を使用するか実行時に動的に時間単位を変更するには 1423 ページの「setTimeUnit」を参照してください。

Link with TOR
`torLibrary (TORLibraryLinkKind)`

生成済みコードと TOR ライブラリとのリンク方法を指定します。デフォルトでは、TOR と静的ライブラリとしてリンクされますが、動的（共有）ライブラリとして TOR とリンクすることもできます。このトピックの詳細については、TOR のビルドを参照してください。

生成済みコードが TOR ライブラリに依存しない場合は、この設定は無視されます。

装備に関するオプション

生成済みコードの機能の装備に関するオプションです。

Enable instrumentation
```
instrumentationOptions = InstrumentationOptions (. enable .)
(boolean)
```

このオプションをオンにすると C++ アプリケーション ジェネレータは機能を備えたコードを生成します。これは一般的に 2 つの追加ファイル `torInstrumentation.h` と `torInstrumentation.cpp` およびその他の生成ファイルに対する多少の追加を含みます。機能の装備は、生成されたアプリケーションがイベント（アプリケーション レベルのイベントと区別するため「メタ イベント」と呼ばれます）を送ることによってそれが何を行っているかを通信できるようにすることを目的としています。これらのメタ イベントは、アプリケーション内で何が起きているかを示すシーケンス図トレースを作るためにログ ファイルに書き込むかホスト Tau IDE へ送ることができます。UML レベルのデバッグを行うためには、機能の装備を有効にしておく必要があります。

Synchronous instrumentation events
```
instrumentationOptions = InstrumentationOptions (.
syncMetaEvents .) (boolean)
```

このオプションをオンにすると、メタ イベント（つまり、アプリケーションが何をしているかを示すイベント）の処理が同期的に行われます。これは、たとえば生成済みアプリケーションを、ホスト Tau IDE でシーケンス ダイアグラムを作成しながら外部デバッガでデバッグする場合に便利です。その場合、ダイアグラムは、デバッグしているアプリケーションをステップしながら徐々に更新されます。

デバッグ オプション

C++ アプリケーション ジェネレータによって C++ コードに翻訳される、UML レベルでのモデルのデバッグに関するオプションです。

Executable for debug session

```
debugSettings = DebugOptions (. executable .) (string)
```

Tau から Launch コマンドを使用してデバッグを開始する場合、デフォルトではビルド アーティファクトから生成されたアプリケーションが起動され、デバッグに割り当てられます。このオプションにより、別の実行形式ファイルの起動を指定できます。これは、ビルド アーティファクトから動的（共有）ライブラリが生成されており、ライブラリのホストとして別のアプリケーションが使用されている場合などに有用です。

Command line arguments

```
debugSettings = DebugOptions (. commandLineArgs .) (string)
```

Tau から Launch コマンドを使用してデバッグ セッションを開始するときに起動する実行形式ファイルに、コマンドライン引数を指定します。

Host name

```
debugSettings = DebugOptions (. host .) (string)
```

実行中の実行形式ファイルに割り当てて開始するデバッグ セッションに、TCP/IP ホストの名前を指定します。

TCP/IP port

```
debugSettings = debugOptions (. port.) (natural)
```

実行中の実行形式ファイルに割り当てて開始するデバッグ セッションに、TCP/IP ホストのポート番号を指定します。

インクルード保護オプション

インクルード保護オプションは、各ヘッダ ファイルについて生成されるインクルード保護のフォーマットを指定します。詳細については、インクルード保護を参照してください。

Include Protection First String

```
includeProtSettings = IncludeProtectionSettings (.
includeProtectionBegin .) (string)
```

インクルード保護の開始のフォーマットを指定します（ヘッダー ファイルの先頭で生成）。

Include Protection Last String

```
includeProtSettings = IncludeProtectionSettings (.
includeProtectionEnd .) (string)
```

インクルード保護の終了のフォーマットを指定します（ヘッダー ファイルの最後で生成）。

Automatic model update

```
autoUpdate (boolean)
```

このオプションを true に設定すると、Tau が生成済み C++ ソースファイルの変更を検知したタイミング（たとえばテキストエディタでの保存など）で、モデル更新が自動で行われます。このオプションを使用するには、Support roundtrip オプションを有効にする必要があります。

Automatic code generation

```
autoGenerate (boolean)
```

このオプションを true に設定すると、UML モデルを変更して保存したタイミングで、C++ コード生成が自動で実行されます。

Automatically add operation bodies for operations

```
autoGenerateOperationBodies (boolean)
```

このオプションは、モデルに操作本体が定義されていない場合に C++ アプリケーション ジェネレータが空の操作本体を追加 するかどうかを制御します。

翻訳のカスタマイズ

C++ アプリケーション ジェネレータの出力は、エージェントを使用してカスタマイズできます。これにより、生成されるコードを正確に制御できます。

生成されるコードは次の 2 通りの方法でカスタマイズできます。

1. 生成されるファイル内の特定の位置に任意のテキストを追加。この方法は、生成される通常のコードとともに、追加のコード、コメントまたはプリプロセッサ ディレクティブの生成を可能にします。
2. C++ 言語の特定構成要素の生成を任意のテキストに置換。この方法により、マクロ ライブラリの使用などの非 C++ 構成要素を UML エンティティとして表現できます。また、UML エンティティの C++ への翻訳方法を定義することが可能となります。

ここでは、C++ アプリケーション ジェネレータをカスタマイズするこれらの 2 つの方法について説明します。どちらの方法も、カスタマイズ エージェントの定義に基づいています（エージェントの詳細については、第 57 章「エージェント」を参照してください）。

コード生成時にテキストを追加

この種のカスタマイズは、コード生成時に、これらのツール イベントによってトリガされるツール イベントとエージェントに基づいています。可能なカスタマイズは以下の 2 通りの方法で、任意のテキストを追加することです。

- C++ 定義の生成の直前または直後（ツール イベント Print C++ Source File 参照）
- 生成後のファイルの先頭または最後に任意のテキストを追加（ツール イベント Print C++ Definition 参照）

umlCppAgentCustomization の例は、Print C++ Definition ツール イベントに対処するために、C++ API を用いてカスタマイズ エージェントを記述する方法の例です。

コード生成時にテキストを置換

この種のカスタマイズは、ユーザー定義ステレオタイプを使用してモデリングされます。これらのユーザー定義のステレオタイプは、操作見出し、操作定義、Typedef など、カスタマイズの対象となる C++ 構成要素（文法規則）を指示する一組の組み込みのステレオタイプを特化したものです。ユーザーによる特化が必要な内蔵ステレオタイプの定義形式は次のとおりです。

```
stereotype customCppGen<GRAMMAR RULE> extends
TTDMetamodel::<METACLASS> [0 .. 1] {
    abstract void Unparse(out Charstring result);
}
```

<GRAMMAR RULE> は、そのステレオタイプによってカスタマイズ可能な C++ 文法規則の名前です。<METACLASS> は、その文法規則に一致する UML のメタ クラスです。

"Unparse" 操作はベース ステレオタイプで抽象として定義されるので、ユーザー定義のステレオタイプはこの操作を実装しなければなりません。これは、同じシグニチャを使って "Unparse" 操作をユーザー定義のステレオタイプに追加し、その操作に <<agent>> ステレオタイプを適用し、実行する実装を定義します。エージェントの実装は、"result" 出力パラメータへの文字列値の割り当てに関与します。この結果、本来ならその特定の文法規則に対して生成されるテキストの代わりに、C++ アプリケーション ジェネレータによってこの文字列が表示されます。

注記

翻訳のカスタマイズは、ラウンドトリップ エンジニアリングと一緒に使用しないことを推奨します。ユーザー定義のカスタマイズ エージェントをアクティブにして生成されたファイルをどうしてもラウンドトリップしたい場合、これらのエージェントによって追加されるコードは、ラウンドトリップ時に無視されるように、<GENERATED> タグ付きコメントで囲む必要があります。

例 **577**

こういったカスタマイズ可能性を以下の C++ の特別な適用例で説明します。System-C は、パフォーマンス分析とハードウェア統合を意図した C++ ベースのハードウェア モデリング言語です。System-C は、その内容は、ハードウェアを C++ で記述する仕組みを提供する C++ のクラスとユーティリティのライブラリです。System C の中心的な概念は「モジュール」です。これは C++ のクラスですが、ソース コードでは、SC_MODULE マクロを使用して定義されます。小さい System-C モジュール（NAND ゲートを定義）は、次のように記述されます。

```
SC_MODULE(nand){
sc_in<bool> A, B;
sc_out<bool> O;
}
```

これは標準の C++ クラスですが、マクロで定義されます。このマクロは通常の C++ クラスの定義の代わりに使用されるため、標準の C++ コードの生成ではこのコードは生成できません。ただし、カスタマイズを行うことで、この生成を準備できます。

実現したいことは、エンド ユーザーが UML モデルでクラスを簡単に定義し、そのクラスに <<Module>> ステレオタイプを適用し、System-C のモジュールであることを示すことです。C++ コード ジェネレータを実行すると、通常のクラス ヘッダーの代わりに、上記の例にあるような SC_MODULE マクロが生成されます。

これを行うには、ステレオタイプ定義 <<Module>> を使用して、System-C 用の UML モジュールを作成します。このステレオタイプが C++ のコード生成に作用するようにするには、そのステレオタイプを内蔵ステレオタイプ customCppGenClassHeading から継承させます。さらに、標準の C++ コード生成を無効にするエージェントとなるように定義されている "Unparse" という操作を追加します。エージェントはさまざまな方法で実装できますが、この場合には、別の DLL で提供される C++ の実装を選択します。テキスト構文では、<<Module>> ステレオタイプの定義は次のように記述されます。

```
stereotype Module : customCppGenClassHeading {
<<agent(.implKind = CPP,
implementation="some/path/myLib#Module.dll".)>>
```

```
    void Unparse(out Charstring result);
    }
```

System-C プロファイルがロードされると、C++ コード ジェネレータは、ステレオタイプ <<Module>> のタグが付いたクラスを検出すると、クラス見出しのコードを発行する代わりに、Unparse エージェントを呼び出します。

エージェント定義から参照される動的ライブラリ Module.dll には、"result" 出力パラメータのテキスト文字列を割り当てる関数の C++ で表現した実装が含まれています。この結果、このテキスト文字列は通常のクラス見出しの代わりに使用されます。この場合、このエージェントを、以下のように C++ ソース コードで定義できます。

```
AGENT_IMPL( Module )
{
    tstring strReturn = _T("SC_MODULE( ");
    tstring strName;
    pContext->GetValue( _T("Name"), strName );
    strReturn += strName;
    strReturn += _T( " )" );

    u2::AgentParameter* apReturn = agentParameters.front();
    apReturn->Set(strReturn);
}
```

エージェントの実行をトリガする要素（この場合は、<<Module>> クラス）を特定する暗黙のパラメータ "pContext" に注目してください。生成されるコードの一部となるテキスト文字列は、"agentParameters" 出力パラメータを使用して返されます。

カスタマイズのポイント

このセクションでは、特定の文脈に対して出力されるテキストを置き換えることによって、C++ コードジェネレータの出力をカスタマイズできる文脈を定義しています。C++ コードのどの部分が置き換えられるかについては、サポートされる C++ 文法を使用して定義しています。第 40 章「C++ テキスト構文」を参照してください。

Namespace Heading

ステレオタイプ名 : customCppGenNameSpaceHeading

拡張 : Package

文法規則 :

<namespace definition> ::=

‘namespace’ [<identifier>] <braced declarations>

置き換えられるパート : ‘namespace’ [<identifier>]

Namespace

ステレオタイプ名 : customCppGenNameSpace

拡張 : Package

文法規則 :

<namespace definition> ::=

‘namespace’ [<identifier>] <braced declarations>

置き換えられるパート : Entire rule

Class Heading

ステレオタイプ名 : customCppGenClassHeading

拡張 : Class

文法規則 :

<class definition> ::=

<class key> [<identifier>] [<base clause>] <class body>

置き換えられるパート : <class key> [<identifier>] [<base clause>]

Class

ステレオタイプ名 : customCppGenClass

拡張 : Class

文法規則 :

<class definition> ::=

<class key> [<identifier>] [<base clause>] <class body>

置き換えられるパート : Entire rule

Interface Heading

ステレオタイプ名 : customCppGenInterfaceHeading

拡張 : Interface

文法規則 :

<interface definition> ::=

'UML_INTERFACE' <identifier> [<base clause>] <interface body>

置き換えられるパート : 'UML_INTERFACE' <identifier> [<base clause>]

Interface

ステレオタイプ名 : customCppGenInterface

拡張 : Interface

文法規則 :

<interface definition> ::=

'UML_INTERFACE' <identifier> [<base clause>] <interface body>

置き換えられるパート : Entire rule

TypeDef

ステレオタイプ名 : customCppGenTypedef

拡張 : Syntype

文法規則 :

<typedef declaration> ::=

‘typedef’ <decl specifier seq> <declarator> ‘;’

置き換えられるパート : Entire rule

Attribute

ステレオタイプ名 : customCppGenAttribute

拡張 : Attribute

文法規則 :

<member declaration> ::=

[<decl specifier seq>] [<member declarator>] ‘;’

| <function definition> [‘;’]

| <using declaration>

| <template declaration>

| <friend declaration>

| <typedef declaration>

置き換えられるパート : [<decl specifier seq>] [<member declarator>] ‘;’

Enumeration Heading

ステレオタイプ名 : customCppGenEnumerationHeading

拡張 : Datatype

文法規則 :

<enum definition> ::=

‘enum’ [<identifier>] <enumerator body>

置き換えられるパート : ‘enum’ [<identifier>]

Enumeration

ステレオタイプ名 : customCppGenEnumeration

拡張 : Datatype

文法規則 :

<enum definition> ::=

‘enum’ [<identifier>] <enumerator body>

置き換えられるパート : Entire rule

Union Heading

ステレオタイプ名 : customCppGenUnionHeading

拡張 : Choice

文法規則 :

<class definition> ::=

<class key> [<identifier>] [<base clause>] <class body>

置き換えられるパート : <class key> [<identifier>] [<base clause>]

Union

ステレオタイプ名 : customCppGenUnion

拡張 : Choice

文法規則 :

<class definition> ::=

<class key> [<identifier>] [<base clause>] <class body>

置き換えられるパート : Entire rule

Operation Definition Heading

ステレオタイプ名 : customCppGenOperationHeading

拡張 : Operation

文法規則 :

<function definition> ::=

[<simple decl specifier seq>] [<type specifier>] <declarator> <function body>

| '#if' <expression> [<simple decl specifier seq>] [<type specifier>]

<declarator> <function body> '#endif'

置き換えられるパート : [<simple decl specifier seq>] [<type specifier>] <declarator>

Operation Definition

ステレオタイプ名 : customCppGenOperation

拡張 : Operation

文法規則 :

<function definition> ::=

[<simple decl specifier seq>] [<type specifier>] <declarator> <function body>

| '#if' <expression> [<simple decl specifier seq>] [<type specifier>]

<declarator> <function body> '#endif'

置き換えられるパート : [<simple decl specifier seq>] [<type specifier>] <declarator> <function body>

Operation Declaration Heading

ステレオタイプ名 : customCppGenOperationDeclarationHeading

拡張 : Operation

文法規則 :

<function definition> ::=

[<simple decl specifier seq>] [<type specifier>] <declarator> <function body>

| '#if' <expression> [<simple decl specifier seq>] [<type specifier>]

<declarator> <function body> '#endif'

置き換えられるパート : [<simple decl specifier seq>] [<type specifier>] <declarator>

Operation Declaration

ステレオタイプ名 : customCppGenOperationDeclaration

拡張 : Operation

文法規則 :

<function definition> ::=

[<simple decl specifier seq>] [<type specifier>] <declarator> <function body>

| '#if' <expression> [<simple decl specifier seq>] [<type specifier>]

<declarator> <function body> '#endif'

置き換えられるパート : [<simple decl specifier seq>] [<type specifier>] <declarator> <function body>

Operation Body

ステレオタイプ名 : customCppGenOperationBody

拡張 : Operation

文法規則 :

<function body> ::=

[<ctor initializer>] <compound statement>

置き換えられるパート : Entire rule

Generalization

ステレオタイプ名 : customCppGenGeneralization

拡張 : Generalization

文法規則 :

<base specifier> ::=

    [<access to base>] <identifier>

置き換えられるパート : Entire rule

その他

ここでは、どの UML 構成要素とも直接関連しないが、C++ 言語の本質に関連する UML から C++ への翻訳の側面をいくつか説明します。

宣言の順序と前方宣言

C++ ソース ファイルに生成される宣言の順序は、次のとおりです。

1. 定義（インライン操作本体を含む）
2. 操作本体（非インライン）

定義が同じファイル内の別の定義に依存する場合、参照と定義の順序のために必要ならばサプライヤ定義の前方定義が生成されます。前方宣言は、最初の参照エンティティのできるだけ近くに置かれます。しかし、C++ のスコープ ルールに従う必要があります。つまり、前方宣言は対応する定義と同じスコープ内に入れる必要があります。されに、メンバー定義の場合には前方宣言と対応する定義でアクセス可視性（public、private など）が同じでなければなりません。

例 **578:** ———————————————————————————————————

UML

```
class C {
  public D x;
  class D {
  }
  E y;
}
class E {
}
```

C++

```
class E; // placed in global scope
class C {
public:
  class D; // public and placed in C
  D* x;
  class D {
  };
  E* y;
};
class E {
};
```

———

注記

スコープ S 内の定義 A が同じスコープ S 内の別の定義 B に順序が依存している場合、A は B より先に生成されます。つまり、1 つのスコープ内の定義は、順序依存性によって必要ならソートされています。順序依存性は、ターゲットの完全な定義（前方宣言では不十分）を必要とする参照です。参照依存性は遷移的です。つまり、A が B に依存し、B が C に依存する場合、A も C に依存します。

前方宣言は、可能な場合は #include:s ではなくヘッダー ファイルにも生成できます。これにより、生成済みファイル間での環状のインクルード依存の発生を抑えることができます。

main 関数

実行形式のアプリケーションのエントリ ポイントを定義するために、すべての C++ プログラムでグローバル スコープに 'main' 関数が必要です。UML モデルから C++ アプリケーションを生成する際、以下の 3 通りの方法で 'main' 関数を取得できます。

1. UML モデルで 'main' 操作を指定し、<<‘global namespace’>> パッケージに格納します。操作の実装をマニフェストするため、実装ファイルに <<‘manifest implementation’>> 依存を追加します。'main' 操作の実装は、UML で全体を指定するか、あるいは、必要に応じてターゲット コードの一部を含めることもできます。
2. 上記と同様に UML モデルで 'main' 操作を指定します。ただし、UML に 'main' 操作を実装しません。その代わり、別ファイルに 'main' 操作の実装を手書きしてからコンパイルし、生成されたアプリケーションにリンクさせます。
3. UML モデルで 'main' 関数をまったく指定せず、完全に手動で定義します。

注記

UML モデルからライブラリを生成する場合は、'main' 操作は必要ありません。

デフォルトの 'main' 関数の実装は、以下のルールに従って、C++ アプリケーション ジェネレータによって生成されます。

UML モデルにおける main 操作の生成

Tau は UML モデルに ’main’ 操作を追加するためのコマンドを提供します。この ’main’ 操作は、C++ アプリケーション ジェネレータによって ’main’ 関数に翻訳されます。このコマンドを起動するには以下の手順を行ってください。

1. ［モデルビュー］でパッケージを選択します。パッケージは <<global namespace>> ステレオタイプで型付けされたパッケージです。
2. コンテキストメニューから、［ユーティリティ］>［Generate Main Function］を選択します。

以下の規則にしたがって、デフォルトの ’main’ 関数の実装が生成されます。

ビルド アーティファクトによりマニフェストされるアクティブ クラスごとにインスタンスが 1 つ生成されます。1 つのインスタンスは 1 つの Dispatcher に追加され、初期化されて、開始されます。最後に Dispatcher で 'run' 関数が呼び出されます。

つまり、デフォルトの 'main' 関数は、完全に同期型のシングルスレッドのアプリケーションになります。もちろん、このデフォルト実装は、使用形態に合わせて変更できます。

例 **579:** デフォルト **'main'** 関数の生成

UML

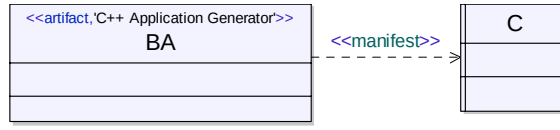


図 257: ビルド アーティファクトによってマニフェストされたアクティブ クラス

C++

```
int main(int argc, char** argv) {
  tor::Dispatcher* dispatcher = new tor::Dispatcher(new
tor::EventQueue);
  C* v1 = new C;
  dispatcher -> add(v1);
  v1 -> init();
  v1 -> start();
  dispatcher -> run();
  return 0;
}
```

モデルには、main 操作が生成されるパッケージをマニフェストする C++ ビルドアーティファクトを含んでいる必要があります。C++ ビルドアーティファクトが見つからない場合は、空の main 操作が生成され、警告メッセージが［メッセージ］タブに表示されます。

42

C++ アプリケーションの環境

このセクションでは、C++ アプリケーション ジェネレータを使って生成したアプリケーション（またはアプリケーションの一部）とその環境との間のインターフェイスをとる方法について説明します。特定のモデルにどの手法を選択するかは、環境の特性、モデルで使用するスレッド デプロイの方法、設計者個人の好みなど、複数の要因によって決まります。したがって、ここで提示する情報は、有効な選択肢をすべて示した完全なリストではなく一般的な代表例を示すものとして考えてください。

概要

アプリケーションの環境を構成するものが何なのかについて、従来次のような定義がなされていました。つまり、それはアプリケーションの中心的な責務の領域の外部と考えられるアプリケーションの部分の総体である、という定義です。ここで「外部」という用語の意味は漠然としています。なぜなら多くの場合、アプリケーションとその外部環境との間にはっきりした境界線を引くことは難しいからです。このような境界線を引くための基準はたくさんあります。その一部を以下に示します。

- ハードウェア コンポーネントに対して、ソフトウェア コンポーネント
- ユーザ インターフェイス コンポーネントに対して、その他のコンポーネント
- ある技術で開発されたコンポーネントに対して、別の技術で開発されたコンポーネント
- 従来のコンポーネントに対して、新しく開発されたコンポーネント

このセクションでは、「環境」という用語は、C++ アプリケーション ジェネレータによって生成されたのではないアプリケーションの部分を意味します。たとえば、手書きのソフトウェア コンポーネント、サードパーティのライブラリ、またはハードウェア コンポーネントなどです。

環境のモデリング

環境の表現を UML モデルそのものに含めることが有用である場合が多々あります。こういった表現は 1 つまたは複数の UML 定義（通常は、クラス、属性、アクター、サブジェクトなど）で構成されるでしょう。システム分析時または初期設計段階でこれを行っていることも多いはずです。

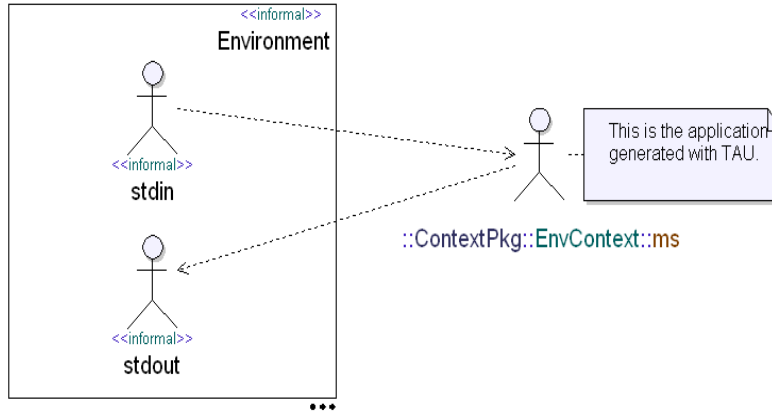


図 258: `stdin` からの読み込み `stdout` へ書き出すアプリケーションの環境のモデリング

環境とのインターフェイス

詳細設計に進む段階で、環境内のコンポーネントとのインターフェイスを形式化する必要があります。この形式化には少なくとも以下の 3 つの方法があります。

- 環境コンポーネントが C または C++ モジュールの場合、C/C++ のインポートを利用して、そのインターフェイスの詳細な UML 表現を自動生成できます。したがって、その環境コンポーネントを、C++ アプリケーション ジェネレータを使って生成される UML モデルから直接使用できます。
- 環境コンポーネントがハードウェア コンポーネントの場合、モデルにおいてそのコンポーネント用のインターフェイス アダプタを作成することが得策となることがあります。このようなインターフェイス アダプタは多くの場合クラスタイプの属性としてモデル化され、ハードウェア コンポーネントとやりとりする責任を負います。また、UML モデルの他の部分にとっても、ハードウェア コンポーネントを抽象化したものとして機能させることができます。
- 環境コンポーネントが外部コード（GUI など）で構成されている場合、その環境コンポーネントに `tor::EventReceiver` インターフェイスを実現させて、UML モデルからの環境コンポーネントへのシグナルの送信を行うことができます。もちろん、他のカスタムインターフェイスを実現させて、簡易な双方向の関数呼び出しの目的に使用することもできます。これらのインターフェイスは UML モデルに含まれている必要があり、C++ アプリケーション ジェネレータによって C++ ヘッダーとして生成される必要があります。そして、生成されたヘッダーは外部コードにインクルードされて使用されます。外部モジュールがシグナルを UML モデルに返す必要がある場合は、外部モジュールは TOR ライブラリとリンクされる必要があります。

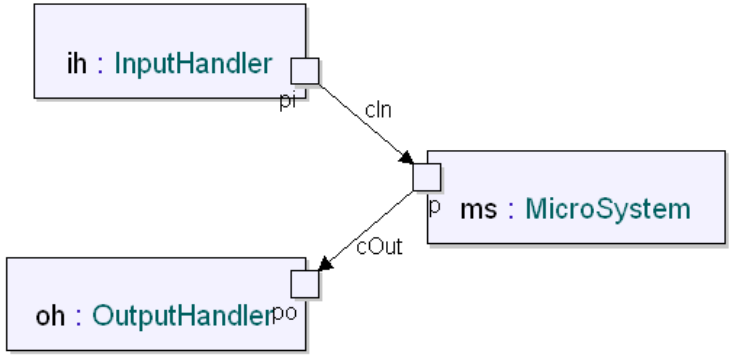


図 259: インターフェイス アダプタによる環境コンポーネントの表現

一般的にいって、環境コンポーネントとインターフェイスする際には以下の仕組みが有効です。

1. モデルから外部の C または C++ コードにアクセスする機能。これは、C/C++ のインポート ツールを使用して簡単に実行できますが、インライン ターゲット コード（[[...]]）などの他の方法も使用できます。外部コードをモデルと並行して開発する場合、Tau から外部クラスのヘッダーを生成し、実装ファイルを手動で作成する方法も有効です。
2. 環境コンポーネントによって実現されるインターフェイスを通して操作を呼び出す機能。このインターフェイスは、外部で定義してから UML にインポートするか、または UML モデル内で定義してから C++ を生成するか、のいずれかです。
3. シグナルを外部 C++ クラスに送信する機能（`tor::EventReceiver` インターフェイスを使用）。外部クラスがシグナルを直ちに実行する（同期実行）場合と、後で実行する（非同期実行）場合があります。
4. 外部コードからモデルへシグナルを送信する機能。これは、C++ ランタイム フレームワークの `torUtilities.h` ヘッダーで利用可能な `tor::sendTo` ユーティリティを使用して実行されます。
5. 外部コードから UML 内で定義された操作を呼び出してデータにアクセスする機能。モデルから生成された該当のヘッダー ファイルをインクルードして行います。

マルチスレッド アプリケーション

モデルをマルチスレッド プログラムとしてデプロイした場合に重要なのは、モデル内のエンティティから行われる操作呼び出しやシグナルが、異なるスレッドから実行される可能性があることを理解することです。インターフェイスする外部コンポーネント自体がスレッドセーフでない場合、UML モデル内のコンポーネントに対応したイン

ターフェイス アダプタを作成できます（1406 ページの図 259 参照）。このアダプタは、通常はアクティブクラスのインスタンスであり、それ独自のスレッドかまたはモデル内の他のスレッドの 1 つにディスパッチされる必要があります。いずれの場合も、このアダプタは、モデルから外部コンポートへのすべての通信の順序付けを行なうことで、環境コンポーネント内のスレッド問題に対する「プロテクタ」として機能します。

外部のリソースからのデータ読み出しの責任を負うインターフェイス アダプタは、ブロックとなることがよくあります。このようなインターフェイス アダプタにはそれ独自の実行スレッドを与えて、外部コンポーネントからのデータ読み出しを待っている間にモデルの他の部分をブロックすることを避ける必要があります。

43

C++ ランタイム フレームワーク

この章では、C++ ランタイム フレームワークの Tau Object Runtime（TOR）について説明します。フレームワーク内のすべてのクラスの目的とランタイム セマンティックについて説明します。

参照

C++ アプリケーション ジェネレータで生成されたコードにおける TOR に使用方法の詳細については第 41 章「C++ アプリケーション ジェネレータリファレンス」を参照してください。

概要

この章は、C++ で実装されたオブジェクト指向 UML ランタイム フレームワークである Tau オブジェクト ランタイム（TOR）のリファレンス ガイドです。TOR は、構造と振る舞いの両面で、UML のランタイム セマンティックを実装します。

TOR は、それぞれ明確に指定されたサービスを提供する一連のクラスとして実装されています。一部のクラスはフレームワーク自体の中で使われます。その他のクラスは C++ アプリケーション ジェネレータで生成されたコードで使われて、モデルを実行可能な C++ コードに変換します。

一部のクラスはモデリング時に使われます。これらの TOR クラスは TOR UML モデルとして使用できます。

フレームワークは、一連のヘッダーとソース ファイルとして提供されます。ファイルのリストは、ファイルのリストのセクションを参照してください。

TOR 名前空間

TOR のすべての宣言は `tor` という名前空間で行われます。これにより、ユーザー定義クラスとの衝突を避けながら State のような短い記述的なクラス名を使用できます。また、グローバル スコープ内の定義が多くなりすぎるのも防ぎます。

TOR UML モデル

TOR の一部は、C++ アプリケーション ジェネレータがアクティブになると自動的にロードされるモデル ライブラリ「tor」として使用できます。このモデルには、ユーザー モデルで使用できるすべてのタイプ、クラス、操作、および関数が、含まれます。

このモデルのクラスと操作については、TOR クラスのセクションで説明しています。

注記

TOR フレームワーク内のエンティティおよびその中でも特にメモリ割りあて、Mutex、セマフォなど同期機構を使用するものは、明示的であれ暗黙的であれ、割り込みルーチンやシグナルハンドラのような機能の中では使用すべきではありません。オペレーティングシステムにもよりますが、このような使用はアプリケーションを誤動作させる可能性があります。

TOR のビルド

TOR は、C++ アプリケーション ジェネレータで生成されたアプリケーションを初めてビルドするときに、自動的にビルドされます。生成済みアプリケーションが TOR へのリンクを必要とする TOR 定義に依存していない場合は、（たとえば、ヘッダー ファイルに定義されたテンプレートのみを使用している場合）、TOR はビルドされません。

ビルド アーティファクトに適用されている C++ Application Generator ステレオタイプにより、TOR を静的ライブラリまたは動的（共有）ライブラリのどちらとしてビルドすべきかを指定できます。デフォルトでは、静的ライブラリとしてビルドするよう指定されています。すべてが TOR に依存する複数のバイナリによってアプリケーションが

構成される場合、TOR を動的（共有）ライブラリとしてビルドするように変更する必要があります。たとえば、アプリケーションが複数の動的（共有）ライブラリで構成される場合、または複数の動的（共有）ライブラリとともに実行される場合などは、これに当てはまります。この場合、動的リンクを使用することが重要です。これは、TOR がアプリケーション全体の中で 1 つのコピーにのみ存在するよう設計されているからです。アプリケーション内で複数コピーの TOR を持つと（複数のバイナリ リンクが静的に設定されている場合）、アプリケーションの異なるバイナリ モジュール間で送信されたシグナルが失われ、予期せぬランタイム エラーが発生することがあります。

TOR を動的（共有）ライブラリとしてビルドすると、生成済みコードと TOR ライブラリ自体のコンパイル時に、`TOR_DLL` マクロが定義されます。アプリケーションにユーザーが作成したコードが含まれており、TOR に依存している場合、定義されたマクロによってそれもコンパイルする必要があります。

TOR のビルド方法のカスタマイズについては、ビルドを参照してください。

TOR の初期化とファイナライズ

TOR ライブラリを使用するためには、まずそれを初期化する必要があります。ライブラリのロード時に、TOR と静的にリンクされている実行形式ファイルの開始の一部として、あるいは動的（共有）ライブラリとしてビルドされた TOR のロード時に、ライブラリの初期化が自動的に行われます。

アプリケーションで TOR の使用を停止する場合は、TOR をファイナライズして、初期化時に割り当てられたリソース（スレッド、メモリなど）を解放する必要があります。ライブラリをアンロードする際、TOR と静的にリンクされている実行形式ファイルの終了の一部として、あるいは動的（共有）ライブラリとしてビルドされた TOR のアンロード時に、ライブラリのファイナライズが自動的に行われます。

TOR の自動的な初期化およびファイナライズでは不十分な場合があります。たとえば、アプリケーションがその実行の一部としてのみ TOR を使用している場合、不要になった場合は明示的に TOR をファイナライズしてリソース（メモリとスレッド）を保存できます。再度ライブラリが必要になったときは TOR を明示的に初期化できます。

TOR の明示的な初期化およびファイナライズは、関数 initializeLibrary および finalizeLibrary を使用して実行できます。

注記

OS によっては（Windows など）、プログラムの通常の終了手順によって（main 関数からのリターンなどにより）、動的にリンクされたライブラリがアンロードされる前に、すべてのアクティブ スレッドが暗黙的に終了します。したがって、そのような OS で実行されているアプリケーションと TOR が動的にリンクされている場合は、アプリケーションの終了前に、finalizeLibrary を呼び出して TOR を明示的にファイナライズする必要があります。そうしないと、TOR が使用するスレッドが OS によって完全に終了されるため、TOR のアンロード時に行われる自動ファイナライズによって、プログラムがハングアップします。

常に必要とは限りませんが、初期化とファイナライズは明示的に行ったほうがよい場合があります。たとえば、initializeLibrary の呼び出しを main 関数の最初に置くか、グローバル オブジェクトのコンストラクタ内に入れることができます。同様に、finalizeLibrary の呼び出しを main 関数の最後に置くか、グローバル オブジェクトのデストラクタ内に入れることができます。

重要！

finalizeLibrary 呼び出しによって TOR を明示的にファイナライズした場合、それ以降 TOR を使用しないことが重要です。特に注意が必要なケースとして、アプリケーションが（ThreadedDispatcher を使用して）UML モデルを複数スレッドにデプロイする場合があります。このようなスレッドは、TOR をファイナライズする前にすべてファイナライズする必要があります。そうしないと、ファイナライズされた後で、これらのスレッドが TOR にアクセスし、クラッシュが発生することがあります。finalizeLibrary を呼び出す前に、必ず ThreadedDispatcher のすべてのインスタンスを削除してください。

TOR クラス

このセクションでは、TOR で定義されるクラスをアルファベット順に示します。各クラスの目的とランタイム セマンティックについて説明します。すべてのクラスは TOR 名前空間で宣言されます。OS 関連クラスについてはオペレーティング システム抽象化レイヤを参照してください。また、メタデータ関連のクラスについては、メタデータ表現を参照してください。

- CompletedEvent
- Connector
- Dispatchable
- DispatchableClass
- Dispatcher
- DispatcherData
- EntryPoint
- Event
- EventExecutor
- EventQueue
- EventReceiver
- ExitPoint
- InstanceManager
- InternalEvent
- Port
- Region
- RunInitialTransition
- State
- StateMachine
- DispatcherData
- ThreadSafeEventQueue
- TimerEvent
- TimerObject
- TimerQueue
- TopRegion

CompletedEvent

ある State が終了したときフレームワークが生成する内部 Event です。これは、状態に渡され直ちに処理されます。トリガ イベントなしで任意の遷移をトリガするために使われ、「トリガー フリー遷移」ともいいます。

ガード付き遷移にのみ使われるヌル イベントもありますが、これとは異なり CompletedEvent は、ガードなしの遷移、つまり、イベントもガードもない遷移をトリガします。

Connector

コネクタは、一般的に DispatchableClass のサブクラスであるポートを所有するクラス間で Event を送るために使用する 2 つの Port 間の接続を表します。

コネクタは、2 つのポートを接続できます。2 つのポートが接続されていると、一方のポートに送られたイベントはもう一方のポートに渡されます。2 つのポートを接続するには connect 操作を使用します。

```
void connect(Port* from, Port* to);
```

Port の接続は、ポート自身に対する操作を使用して処理することもできます。

Dispatchable

イベントの受信と実行の両方が可能なエンティティを現す抽象クラスです。イベントの受信は、EventReceiver インターフェイスの実装 (実現) で表現されます。イベントの実行は、EventExecutor インターフェイスの実装 (実現) で表現されます。

1 つの dispathcable は 1 つの Dispatcher と 1 つの EventQueue に関連付けられます。dispatcher が 1 つの Event をキューから取り出して処理するときには、dispatcher は dispatchable にそのイベント処理のために特定されるべきイベントをたずねます。その結果、dispatchable はイベントの取り扱い方法を示した Dispatchable::EventAction を返します。

Dispatchable::EventAction

受信側がイベントを処理する方法を示すために使用する列挙型です。以下の表にリテラルとその意味を示します。

リテラル	説明
NoMatch	受信側はイベントを処理しない。
Defer	受信側はイベントを保存する。
Consumed	受信側がイベントを消費する。

DispatchableClass

dispatchable クラスは、1 つ UML アクティブクラスを現します。このクラスは Dispatchable を継承して、自分に割り振られたイベントを受信して実行する能力を取得しています。

dispatchable クラスの例としては状態機械および / または内部構造があります。第 4 章「UML 言語ガイド」の 247 ページ、「アーキテクチャ モデリング」と比較してください。

dispatchable クラスのインスタンスに送られる Event は、そのインスタンスの StateMachine に渡されます。

クラスが「アクティブ」と指定されると、`tor::DispatchableClass` の継承は、自動的にモデルに追加されます。したがって、アクティブ プロパティが dispatchable クラスの情報を保持します。

dispatchable クラスのインスタンス化

dispatchable クラスのインスタンスは、作成されてもその振る舞いの実行を自動的には開始しません。まず初期化を行う必要があり、その後開始できるようになります。

初期化の最中、すべての内部構造と状態機械の実行の準備が行われます。通常、C++ アプリケーション ジェネレータで生成されたコードが自動的に初期化を行います。dispatchable クラスを手動で初期化するには、`init` 操作を呼び出します。

```
virtual void init();
```

インスタンスを開始すると、初期遷移を実行する要求（RunInitialTransition）をポストすることによって、その状態機械が開始されます。いったん開始されるとインスタンスはイベントを受信できます。開始は、C++ アプリケーション ジェネレータで生成されたコードが自動的に行います。たとえば、生成された main 関数はマニフェストされたクラスのインスタンスを開始し、あるコンテナの開始はデフォルトではそれが含む要素を開始します。ただし、最大限の柔軟性を確保するには、これらをカスタマイズするべきです。開始操作をオーバーロードしてインスタンスの開始をカスタマイズでき、`start` 操作を呼び出していつでも手動で開始できます。

```
virtual void start();
```

すでに開始しているインスタンスを開始しても何も起こりません。

注記

「start」は UML テキスト構文の予約語であることに注意してください。したがって、上記の関数を参照する場合は、次のように単一引用符で囲む必要があります（`'Start'`）。

Dispatcher

dispatcher は EventQueue と関連付けられており、キューに入れられた Event を処理します。イベントは、キューが空でない限り、キューから取り出されて 1 つずつ処理されるか、連続的に処理されます。

dispatcher は、現在アクティブなタイマーに対応する TimerObject が配置されている TimerQueue とも関連付けられています。

dispatcher を Dispatchable と対応付け、その後 dispatcher を開始するコードは、通常 C++ アプリケーション ジェネレータで生成されたコードが追加しますが、以下に示すように手動で行うこともできます。

dispatchable を dispatcher に追加するには `add` 操作を呼び出します。

```
void add(Dispatchable*);
```

dispatchable を dispatcher から削除するには `remove` 操作を呼び出します。

```
bool remove(Dispatchable*) const;
```

dispatcher のイベント キューからイベントの処理を開始するには、run 操作を呼び出します。この操作は、キューが空になるまでイベントをイベント キューから 1 つずつ取り出し、その後リターンします。

```
void run();
```

キューからイベントを 1 つずつ処理するには step 操作を呼び出します。この操作は、キューにある次のイベントを取り出して処理します。

```
void step();
```

DispatcherData

ThreadedDispatcher に必要なデータを保持する 1 つのコンテナクラスです。データは、ThreadedDispatcher によって開始されたスレッドと呼び出し側スレッドの両方から、安全にアクセスできるようにカプセル化されます。

EntryPoint

入場点は、合成 State または StateMachine の 1 つの Region に入場するために使用される名前付きの擬似状態です。これは、その内容に関して何も（たとえば実際のターゲット状態）明かさずに合成状態または下位状態機械に入る方法を提供します。

入場点に到達すると（つまり、ターゲットとして入場点を持つ遷移が実行されると）、その入場点を所有する領域に入り、入場点の出力遷移が実行されます。

入場点は複数の入力遷移を持つことができますが、出力遷移は 1 つしか持つことができません。

Event

Event クラスは、あるシステム内のあらゆる種類のイベントのタイプとオカレンスを現すために使用します。シグナル、非同期操作呼び出し、タイマー タイムアウトなどがその例です。Event のサブクラスがイベントのタイプを指定し、このクラスのインスタンスがイベントオカレンスを現します。

すべてのイベントには、特定のオブジェクト id によって現される受信者があります。InstanceManager は、そのオブジェクト id をイベント インスタンスの実際の受信者である Dispatchable への ポインタにマッピングする責任を負います。受信者はフレームワークが内部的に設定します。

EventExecutor

イベントを実行する能力を持つ抽象クラスです。このクラスは、以下のような、EventExecutor の実装によって実装されなければならない 1 つの純粋仮想関数を保持します。

```
virtual EventAction execute(Event* event) = 0;
```

この関数の実装は Dispatchable::EventAction 列挙の適切なリテラルを返して、EventExecutor によるイベントの取り扱い方法を示す必要があります。

イベントは実行される前に受信されなければならない、ということに注意してください。つまり、通常は、EventExecutor を実装するクラスは EventReceiver をも実装することになります。

注記

イベントの受信と実行を明確に区別することが重要です。イベントが**受信**されるのは、フレームワークによって受信者に配信されたときです。イベントが**実行**されるのは、イベントの受信に関連付けられている振る舞い（通常は遷移）が実行されたときです。イベント受信はイベント実行に先行し、通常、受信と実行の間には一定の時間間隔があります。

EventQueue

イベント キューは Event の FIFO キューです。キューに入れられるイベントは、キューに関連する Dispatcher が処理します。

フレームワークは、イベントの挿入と削除など、イベント キューに関するあらゆる事象を自動で取り扱います。

EventReceiver

イベントを受信する能力を持つ抽象クラスです。このクラスには、EventReceiver の実装クラスによって実装しなければならない 1 つの純粋仮想関数があります。

```
virtual bool receive(Event* e) = 0;
```

この関数の実装は、イベントが受信されれば true を返し、受信されなければ false を返す必要があります。

EventReceiver もインターフェイスとして TOR UML プロファイルにあります。これにより、ユーザー独自のイベント受信者を宣言できます。sendTo ユーティリティ機能を使用して、イベントをイベント受信者へ送ることができます。一般的な使い方の 1 つとして考えられるのは、シグナルを受信する必要があるパッシブ クラスがある場合です。これは、クラスに EventReceiver インターフェイスを継承させ、`receive` 関数を実装することにより可能になります。

イベントの受信と実行の違いについては、EventExecutor の注記を参照してください。

ExitPoint

退場点は、合成 State または StateMachine の Region から離れるために使用する名前付きの擬似状態です。退場点は、合成状態または下位状態機械から、それらをインスタンス化した際のコンテキスト（たとえば出力遷移のターゲット状態など）を知らなくても、離れる方法を提供します。

退場点に達すると（つまり、ターゲットとして退場点を持つ遷移が実行されると）、その退場点を所有する Region を抜け、退場点の発信遷移が実行されます。

退場点は複数の入力遷移を持つことができますが、出力遷移は 1 つしか持つことができません。

InstanceManager

このクラスはオブジェクト id と EventExecutor および EventReceiver 間のマッピングを維持する責任を負います。そのマッピングは、イベント受信者とイベント実行者が作成されたり削除された場合も最新状態を維持します。インスタンスマネージャの主な用途は、インスタンスへの直接のポインタではなく、あるインスタンスのシンボリックな表現 (オブジェクト id) を得ることです。この間接的な表現は、複数のインスタンスが複数のスレッドから互いに独立に生成、削除されるようなマルチスレッドのシステムにおいては重要です。また、複数のアドレススペースをまたがるようなシステムでも必要です。

InternalEvent

このクラスは、フレームワークによって内部的に送信されるあらゆるイベント (モデル内にあるユーザ定義のイベントとは直接関連のないすべてのイベント) にとっての共通の基盤クラスです。

Port

ポートは、DispatchableClass が Event を送受信できる接続ポイントを示します。

ポートは、そのポートを所有する dispatchable クラスに関連付けられます。この関連は、通常、dispatchable クラスを初期化するとき、生成された C++ コードにより自動的に設定されます。

Port の接続

Connector を使用してポートを別のポートに接続するには、`attach` 操作を呼び出します。

```
void attach(Port * port, Connector * connector);
```

イベントの送受信

イベントがポートへ送られると、それは 2 つの方法で処理できます。

1. ポートが Connector によって他のいくつかのポートに接続されている場合、イベントはフレームワークが見つけた最初の接続ポートへ送られます。
2. それ以外の場合、関連する dispatchable クラスが分類子の振る舞い（StateMachine）をする場合、イベントはその状態機械へ渡されます。

イベントが受信側によって処理されたか否かを示すブール値が返されます。ポートを通してイベントを送るには `send` 操作を呼び出します。

```
bool send(Event * event, bool deleteEvent = true);
```

Region

領域は、State または StateMachine をまたがる場所を表現します。領域は 1 組の状態を所有します。領域は現在の状態と以前の状態を把握しています。現在の状態とは、領域の現在アクティブな状態のことです。以前の状態とは、最後に領域を離れたときにアクティブだった状態のことです。

領域への入退場は、遷移の結果にしたがって行われます。

Region に入る

領域に入るには複数の方法があります。初めて領域に入るときには、初期遷移が実行されます。以降領域に入る際には、履歴情報を使用して前の状態に再入することもできます。領域は、EntryPoint を通して入ることもできます。

現在の状態と以前の状態は、遷移の結果として領域内のある状態に入ったときに、更新されます。

Region を離れる

領域を離れるのは、他の領域のターゲット状態に向けた遷移がトリガされたときか、ExitPoint をターゲットとする遷移がトリガされたときです。

現在の状態と以前の状態は、領域内のある状態を離れると更新されます。

Region の終了

領域は、その領域のある最終状態に到達すると完了します。最終状態は、フレームワークでは明示的に定義されていません。

領域が終了すると、内包している状態のトリガーフリー遷移が評価されます。

RunInitialTransition

DispatchableClass が開始されたときに、DispatchableClass に送信される内部イベント。このイベントが実行されると、DispatchableClass クラスの初期遷移が実行されます。

State

StateMachine の状態を示します。1 つの状態を所有する Region は 1 つです。一方、1 つの状態は、複数の領域や 1 つの状態機械を所有できます。このような状態は、合成状態と呼ばれます。

遷移とイベント処理

1 つの状態には一連の入力遷移と出力遷移があります。遷移は、フレームワーク内の個別のクラスで現されるのではなく、遷移を含む状態機械内の操作として現されます。

状態は、イベントを受信すると、そのイベントによってトリガされる出力遷移があるかどうかを調べます。一致するものがあり、ガード式が true ならば遷移が実行されます。遷移には順序がなく、フレームワークが最初に見つけた該当の遷移が実行されます。

該当する遷移がなければ、イベントはこの状態の任意の親状態、つまり、他の状態か状態機械へ渡されます。

状態に入る

状態をターゲットとする遷移が実行されるとその状態に入ります。状態に入るとその状態の入場アクションが実行されます。そしてその状態が合成状態の場合は、所有している領域または状態機械にも入ります。

ある状態に入ると、その所有者である領域の現在と前の状態が更新されます。

状態を離れる

ある状態を離れるということは、その状態の任意の出力遷移がトリガされたか、その状態の親状態の出力遷移がトリガされたということです。状態を離れると、その状態の任意の退場アクションが実行されます。

ある状態を離れると、その所有者である領域の現在と前の状態が更新されます。

StateMachine

状態機械とその実装を示します。

注記

StateMachine は正確に 1 つの TopRegion を持っています。

1 つの状態機械を、クラスの classifier behavior として 1 つの DispatchableClass に関連付けることができます。dispatchable クラスに送信された Event は、状態機械に渡され、そこで処理されます。

1 つの State は、1 つの状態機械を所有できます。このような状態機械は、下位状態機械と呼ばれます。この状態に送信されたイベントは、下位状態機械へ渡されて処理されます。

状態機械がイベントを処理できるようにするには、その状態機械を初期化して開始する必要があります。この作業は、関連付けられた dispatchable クラスに対して対応するアクションが行われたときに、フレームワークによって内部的に行われます。

ThreadedDispatcher

このクラスは、以前は DispatcherThread という名称で知られていましたが、Dispatcher のスレッド版です。ThreadedDispatcher はある 1 つのスレッドを実装します。ThreadedDispatcher は、この Dispatcher に関連付けられたすべての Dispatchable にイベントを割り振ることができる 1 つのスレッドです。この仕組みによって、モデルにおける柔軟なスレッド配置が可能になります。以下に例を示します。

- アクティブクラスのインスタンス 1 つに 1 つのスレッド
  この場合、各アクティブクラスはただ 1 つの ThreadedDispatcher に関連付けられます。モデリングの例で言うと、各アクティブクラスが 1 つの ThreadedDispatcher を 1 つのパートとして含むような形です。そのクラスのコンストラクタで、新たに作成されたインスタンスは ThreadedDispatcher に追加されて、起動されます。起動されたスレッドの生存期間は ThreadedDispatcher の生存期間と同じなので、アクティブクラスのインスタンスが削除されると、スレッドは終了します。
- アクティブクラスに 1 つのスレッド
  クラスに静的な ThreadedDispatcher 属性を持たせる形です。そのクラスのコンストラクタで、新たに作成されたインスタンスは ThreadedDispatcher に追加されます。
- インスタンスのセットに対して 1 つのスレッド
  パッケージスコープの ThreadedDispatcher 属性を 1 つ持たせて、複数のインスタンスを割り振り対象として追加するモデルです。

1 つの ThreadedDispatcher は、起動されたスレッドと呼び出し側スレッドの両方からアクセスする必要のあるすべてのデータを保持する DispatcherData を 1 つ所有します。たとえば、DispatcherData は Dispatcher と ThreadSafeEventQueue を保持しています。このようなデータを DispatcherData インスタンスにカプセル化することによって、データをスレッドセーフなやり方でアクセスできるようになります。

ディスパッチャとイベントキュー は自動的にインスタンス化され、関連付けられます。ディスパッチャはスレッドセーフなやり方でイベントをキューから取り出して処理します。

ThreadedDispatcher を呼び出す方法は以下のとおりです。

```
void launch();
```

ThreadedDispatcher を停止する (開始されたスレッドは終了します) には単純に削除します。または、以下の関数の明示的な呼び出しでもスレッドを停止できます。

```
void endThread();
```

この関数は、ThreadedDispatcher を可能な限り早急に終了します。ただし、現在実行中のイベントからトリガされた振る舞いの中断はしません。この関数は同期的に処理されます。つまり、現在のイベントからトリガされた振る舞いが完了するまで、この関数の処理は待たされます。この関数のオーバーロード版では、タイムアウト値の指定ができます。この指定を使うと、スレッド終了まで長時間待たされずに済みます。スレッドが指定時間の経過後に終了していない場合は、この関数は false を返します。

ThreadSafeEventQueue

ThreadedDispatcher によって使用される、EventQueue のスレッドセーフなサブクラスです。キュー上で実行される操作はすべて Mutex や Semaphore で保護されます。

TimerEvent

タイマー イベントは、タイマー タイムアウト イベントを示す特殊な Event です。これは、`set` や `reset` などのタイマー アクションを行える TimerObject に関連付けられています。

TimerObject

タイマー オブジェクトは、Dispatchable 内のタイマーの宣言を現し、タイマーをセット / リセットする関数、およびタイマーが現在アクティブか否かを照会する関数を提供します。これらの関数の実装は、関連する TimerEvent と TimerQueue を使用してタイマー のセマンティックを実現します。

タイマーオブジェクトは、タイマーがアクティブな場合は、タイムアウト値を保持しています。

TimerQueue

タイマー キューは、現在アクティブなタイマーに対応する TimerObject の優先キューです。タイマー オブジェクトは、そのタイムアウト時間の順に並んでいます。

各 Dispatcher には 1 つのタイマー キューがあり、この待ち行列で、dispatcher が管理している Dispatchable のアクティブ タイマーに対応するタイマー オブジェクトが、管理されています。

TopRegion

トップ リージョンは、StateMachine が所有する Region です。所有者である状態機械をトップ リージョンから取り出すことができます。

1 つのトップ リージョンが完了して、他のすべてのトップ リージョンも終了していれば、所有者である状態機械も終了します。

TopRegion は、Region のサブクラスです。

ユーティリティ

フレームワークと生成された C++ コードが内部的に頻繁に使用するユーティリティ関数があります。これらは、ユーザーが TOR UML ライブラリで使用することもできます。すべてのユーティリティは TOR 名前空間で宣言されています。

sendTo

受信側に Event を送るために使用する関数です。この関数には 3 つのオーバーロード版があります。受信側として EventReceiver、オブジェクト id、または Port を指定できます。

```
bool sendTo(Event* e, EventReceiver* c);
bool sendTo(Event *event, InstanceManager::ObjectId
receiverId);
bool sendTo(Event* e, Port* pp);
```

イベントが処理のため受信側に送られます。イベントが受信側によって受信されたか失われたかを示すブール値が返されます。

いずれの場合も、イベント取り扱いの責任は受信側に渡ります。イベントは、`sendTo` 関数に渡された後は、アクセスも削除もできません。

受信側が既知の場合は、受信側として EventReceiver やオブジェクト id を使用するタイプの関数を使う必要があります。Port タイプの関数は、送信側から見て受信側が不明である場合に内部構造中のイベントを送るために使用します。イベントを送り出せるポートだけは明らかである必要があります。当然ながらイベント送信については、受信側を直接指定して送る場合に比べて、ポート指定で送る場合の方がオーバーヘッドはあります。したがって、Port タイプの `sendTo` 関数を使用するのは、イベントを搬送する接続構造に信頼性がある場合に限ることをお勧めします。

cast

これは、Event のランタイム タイプチェックを行うために使用する関数です。これはイベントの動的キャストを行ってテンプレート タイプと一致するか調べます。

```
template <class T> T cast(Event* e) {
    return dynamic_cast<T>(e);
}
```

この関数には 2 つのバージョンがあります。1 つは const 宣言済みイベント ポインタに作用し、もう 1 つは non-const ポインタに作用します。

setTimeUnit

TOR が使用する時間単位を設定するための関数です。

```
void setTimeUnit(double seconds);
```

時間単位を秒の値で指定します。たとえば、時間単位を 1 ミリ秒に設定するには tor::setTimeUnit(0.001) を呼び出します。

C++ アプリケーション ジェネレータの Time unit を秒、ミリ秒、マイクロ秒、またはナノ秒に設定するオプションがあります。別の時間単位（たとえば分）を使用したいか、実行時に動的に時間単位を変更したい場合は、`setTimeUnit` 関数を呼び出すことができます。

重要！

現時点では時間単位はアプリケーション全体で共有されています。タイマーがすでにアクティブになっている場合は別のスレッドから時間単位を変更しないでください。予期しない結果が生じる可能性があります。

initializeModel

これは、TOR UML ライブラリ内の操作としてのみ使用できます。その理由は、この操作の実装が C++ アプリケーション ジェネレータによって生成されているためです。機能的な観点からは常に UML モデルで `initializeModel` を呼び出す必要はありませんが、たとえば `main` 操作の先頭やグローバルオブジェクトのコンストラクタなどで常に実行することを推奨します。

initializeLibrary

TOR ライブラリの明示的な初期化を実行します。通常、初期化は自動的に行われます（詳細については TOR の初期化とファイナライズを参照してください）。したがって、この関数は、finalizeLibrary を使用して TOR を明示的にファイナライズして、その後再度初期化する場合のみ必要です。

TOR がすでに初期化されている場合は、この関数呼び出しは無効です。

finalizeLibrary

TOR ライブラリの明示的なファイナライズを実行します。通常、ファイナライズはアプリケーションの終了時に自動的に行われます（詳細については TOR の初期化とファイナライズを参照してください）。ただし、自動終了が行われるより前に、TOR の明示的なファイナライズが必要な場合があります。その場合は、この関数を使用します。

TOR がすでにファイナライズされている場合は、この関数呼び出しは無効です。

定義済みのデータ型

TOR では、UML の定義済みのデータ型から C++ データ型（定義済み）へのマッピングを行う複数のデータ型が定義されています。すべてのデータ型は、TOR 名前空間で定義されます。

単純なデータ型

以下の表は、単純な UML データ型から C++ 基本データ型へのマッピングを示します。

UML **データ型**	**C++ データ型**
Boolean（ブール値）	bool
Character（文字）	char（Unicode では wchar_t）
Integer（整数）	int
Natural（自然数）	unsigned int
Real（実数）	double

演算子

equal

```
Boolean equal(Integer i1, Integer i2)
Boolean equal(Natural n1, Natural n2)
Boolean equal(Boolean b1, Boolean b2)
```

implies

```
Boolean implies(Boolean b1, Boolean b2)
```

mod

```
Integer mod(Integer i1, Integer i2)
```

power

```
Integer power(Integer base, Integer exponent}
```

is

```
template <class T, class ARG> Boolean is(ARG arg)
```

as

```
template <class T, class ARG> T as(ARG arg)
```

Any クラス

Any タイプは、Any という空のクラスにマップされます。

```
class Any {};
```

Charstring クラス

Charstring タイプは `std::string` または `std::wstring`（Unicode）テンプレートをベースにした、同じ名前のクラスにマッピングされます。

コンテナ

TOR では、U2 定義済みの定義済みパッケージで定義されている一般コンテナが実装されています。すべてのコンテナは、TOR 名前空間で定義されます。

注記

String コンテナのみサポートしています。

String

TOR の String コンテナは、U2 コンテナ `Predefined::String`. の C++ 実装です。

String は、`std::vector` をベースにしたテンプレート クラスとして定義されています。

参照

第 41 章「C++ アプリケーション ジェネレータリファレンス」の 1318 ページ、「コレクションと多重度の影響」

オペレーティング システム抽象化レイヤ

TOR は、基礎となるオペレーティング システムとのインターフェイスをとるため個別の抽象化レイヤを使用します。このセクションではこのレイヤで定義されているクラスを説明します。

OS レイヤ内のすべてのクラスは、`os` という名前空間で定義されています。この名前空間は、TOR 名前空間 `tor` に含まれています。したがって、完全修飾名は `tor::os.` です。

OS レイヤ内のクラスの一部は TOR UML モデルでも利用可能です。たとえば、ミューテクスやセマフォのような TOR のスレッド同期プリミティブを利用したいケースは頻繁にあります。

Mutex

mutex（バイナリ セマフォ）は、複数の Thread がアクセスするリソースをロックするために使われます。mutex は、ロックすること、およびアンロックができます。これらの操作を行うとそれが成功したかどうかの結果状況が返されます。mutex をアンロックできるのはロックしたスレッドだけです。

以前から使用されているユーティリティクラス auto_lock があります。これによって、コードのブロックを compound ステートメントで囲って同期化できます。このクラスをコンストラクトすると mutex がロックされ、デストラクトするとアンロックされます。複数の出口を持つ関数 (例外をスローする関数も含める) を同期化するのに特に有用です。こういう関数の場合、いくつかの戻り場所で mutex をアンロックし忘れることは容易に想定できるからです。

RWLock

データをスレッド セーフな方法でアクセスするために使用する読み / 書き ロックです。このロックは、「読み」のロック / アンロックと「書き」のロック / アンロックを別々に設定できます。読み / 書き ロックの典型的な使用方法の例は、「書き込み」をロックして複数の人の「読み取り」を許可する場合です。

Semaphore

セマフォは、複数の Thread からアクセスされる資源を、順番にアクセスするために使用されます。セマフォは取得と開放ができます。セマフォを取得する方法には、セマフォが開放されるまで待つ方法と指定時間を待ってから先へ進む方法の 2 つがあります。

Gate

ゲートは 1 つの同期化プリミティブであり、これにより 1 つの Thread が他のスレッドの実行を制御できるようなります。一時点でゲートを通過できるのは 1 つのスレッドのみであり、ゲートは独立的にロック、アンロックできます。ゲートがロックされている場合はスレッドはそこに入れません。

Thread

OS のスレッドです。これは、OS レベルでの実行スレッドを示します。スレッドは、通常、フレームワークが自動的に作成して起動します。使用方法は、ThreadedDispatcher を参照してください。

Time

時間（絶対時間と相対時間の両方）の OS 表現です。このクラスには、現在時刻の取得、追加、および時間値の引き算を行う各種の関数があります。

Process

OS プロセスを表現します。このクラスはプロセスの開始や終了を行う関数やプロセス ID(pid) を取得する関数を含みます。

メタデータ表現

TOR には、C++ アプリケーション ジェネレータで生成されるアプリケーションに関するメタデータの記述に使用する、複数のクラスがあります。メタデータには、アプリケーションの構造に関する情報（使用可能な定義の「ツリー」や定義名など）と、アプリケーションの実行内容に関する情報（呼び出す操作、送信するシグナル、など）の両方が含まれます。後者は「メタイベント」として知られています。このセクションではこれらのメタデータ クラスについて説明します。

メタデータ レイヤのすべてのクラスは、TOR 名前空間 「tor」に内包される meta という名前空間で定義されます。したがって、完全修飾名は「tor::meta」となります。

メタデータは、機能を備えたアプリケーションでのみ使用できます。C++ アプリケーション ジェネレータには、機能を備えたアプリケーションを生成するオプション（Enable instrumentation）があります。このオプションを有効にすると、生成されるソースファイルに機能が追加されます。さらに、torInstrumentation.h ファイルと torInstrumentation.cpp ファイルが生成されます。これらのファイルには、TOR のメタデータ レイヤのクラスを使用する定義が含まれます。TOR ライブラリは、機能を備えたアプリケーションのビルド時に設定されるマクロ TOR_USE_INSTRUMENTATION とともにコンパイルされます。

Application

アプリケーションはアプリケーション全体を表現するオブジェクトです。アプリケーションには必ずこのクラスのインスタンスが 1 つ含まれます。アプリケーションは Definition を継承し、名前と GUID は、そのアプリケーションの生成時に使用された ビルド アーティファクトから取得します。

Call

この Event は、操作の呼び出し時に送信されます。呼び出し元のコンテキストから送信されます。

Called

この Event は、操作の呼び出し時に送信されます。呼び出された操作のコンテキストから送信されます。

Coder

すべての Coder クラスで実現化されるべきインターフェイスです。Coder はアプリケーション データを CoderBuffer にエンコードし、CoderBuffer をアプリケーション データにデコードします。

CoderBuffer

Coder クラスがアプリケーションからエンコードしたデータを格納するために使用する、抽象バッファへのインターフェイスです。

Create

この Event は、データ型の新しいインスタンスの作成時に、作成者のコンテキストから送信されます。

Definition

アプリケーションの全般定義を表現します。C++ 定義の生成元である UML 定義との関係を作成するために、定義ごとに名前と GUID が格納されます。また、メタ定義は所有するメタ定義ともリンクを持ち、C++ 定義が定義されているメタ定義を表現します。この結果、アプリケーションの構造を記述するメタ定義のツリーが形成されます。

Delete

この Event は、データ型のインスタンスの削除時に送信されます。削除者のコンテキストから送信されます。

Event

このクラスはすべてのメタイベントの共通基盤クラスです。Event は InternalEvent のサブクラスです。Event には純粋仮想エンコード操作が定義され、具体的なメタイベントを表現するすべてのサブクラスによって実装されます。アプリケーションと 1 つの EventMedia 間の通信が発生するのは、アプリケーション内で現在起こっている事象を表す適切な Event インスタンスを送信した場合です。

EventManager

機能を備えたアプリケーションは、このクラスのインスタンスが 1 つ（1 つだけ）持ちます。それはアクティブクラスであり、イベント マネージャ インスタンスはそれを独自のスレッドで実行します。イベント マネージャは、EventMedia のリストを管理し、リストの登録と登録解除の機能を持ちます。イベント マネージャはアクティブまたは非アクティブです。アクティブの場合、メタイベント（アプリケーションの現在の実行内容に関する記述）の「処理」要求に応答します。この処理には、メタイベントを文字列にエンコードしたり、エンコードされた文字列を現在登録されているすべての EventMedia に文字列に発行したりする処理が含まれます。また、イベント マネージャは、アプリケーションのスレッドごとに現在の呼び出しスタックの表現を保持する役割を持ちます。

アクティブクラスであるため、イベントマネージャは状態機械（ちなみにこの状態機械は C++ Application Generator によって UML モデルから生成されます）を保持します。この状態機械はイベントマネージャの現在の状態を表現します。アプリケーションが

ブレークポイントに到達するか、Tau ホスト デバッガによってブレークされると特殊な状態に入ります。その状態においては、イベントマネージャは Step や Go のようなデバッグ用コマンドをすべて取り扱うことができます。

EventMedia

エンコードされたメタイベントの発行先となるメディアとのインターフェイス。メディアという用語はごく一般的な意味で使用されています。つまり、イベント メディアはエンコードされたメタ イベントを取り込んで、その処理を行うオブジェクトのことです。

EventReceived

この Event は、アプリケーションのイベント（InternalEvent 以外）の受信時に送信されます。

EventSent

この Event は、アプリケーションのイベント（InternalEvent 以外）の送信時に送信されます。

Exit

この Event は、アプリケーションの終了直前に送信されます。EventMedia と同期をとって、アプリケーションの終了前に未処理のメタイベントをすべて処理するため、内部的に使用されます。

LogFile

この EventMedia は、エンコードされた各メタイベントをログファイルに記述します。

Operation

アプリケーションの操作を表現します。操作には constructor、destructor と ordinary の 3 種類あります。操作は Definition を継承します。

Return

この Event は、操作が返される時点で、呼び出された操作のコンテキストから送信されます。

Returned

この Event は、操作が返される時点で、リターンの実行先の操作のコンテキストから送信されます。

StackContext

スタック フレームの表現です。スタック フレームは呼び出しスタックに属し、アプリケーションのスレッドごとに 1 つの呼び出しスタックがあります。

StringCoderBuffer

このクラスは CoderBuffer インターフェイスを実現化します。文字列メンバーとしてエンコードされた結果を格納します。これは、エンコードされたデータを格納するために TOR アプリケーションで使用される標準クラスです。

StructuralFeature

属性または操作パラメータを表現する構造的機能です。Definition を継承します。

TauHostDebuggerProxy

TOR アプリケーションにおける Tau ホストデバッガを表現するプロキシクラスです。EventManager はこのプロキシを使用して、TCP/IP 接続を通して Tau ホストデバッガと通信します。

TauHostTracer

この EventMedia は、エンコードされたメタイベントを実行中の Tau アプリケーションに送信します。Tau アプリケーションはシーケンス図の形式でトレースを生成し、アプリケーションの実行内容を表示します。

U2P_Coder

このクラスは Coder インターフェイスを実現化します。これは、U2P（UML テキスト構文）を使用してエンコード結果を表示する、標準コーダ です。

ファイルのリスト

次のセクションでは、Tau インストールの一部として提供される TOR のファイルを説明します。

ソースとヘッダー ファイル

完全な TOR ソース コードは Tau のインストールに含まれ、インストール ディレクトリ内の次のフォルダ内にあります。

```
addins¥CppGen¥Etc¥TOR¥CPP
```

これらのファイルは、C++ アプリケーションを作成するとき使われます。

以下の表にすべてのファイルとそれらに含まれる TOR 宣言を示します。

ファイル	TOR 宣言
tor.h	通信と状態機械に関連する全ファイルをインクルードする共通インクルードファイル。主に C++ アプリケーション ジェネレータによって生成されたコード向け
torAnnotations.h	モデルをコード変更で更新するために使用するいくつかのマクロ（a.k.a. ラウンドトリップ）
torChoice.h	UML choice の表現。
torCoders.h torCoders.cpp	Coder, U2P_Coder, CoderBuffer, StringCoderBuffer
torCommunication.h torCommunication.cpp	TCP/IP 通信に使用される様々なクラス。TCP/IP を介して通信を行う分散型アプリケーションのビルドに使用できます
torCompletedEvent.h	CompletedEvent
torConnector.h torConnector.cpp	Connector
torContainers.h	String
torDispatchable.h torDispatchable.cpp	Dispatchable, EventExecutor, EventReceiver
torDispatchableClass.h torDispatchableClass.cpp	DispatchableClass
torDispatcher.h torDispatcher.cpp	Dispatcher
torDispatcherThread.h torDispatcherThread.cpp	ThreadedDispatcher, DispatcherData
torEntryPoint.h	EntryPoint

ファイル	TOR 宣言
torEvent.h torEvent.cpp	Event
torEventManager.h torEventManager.cpp	EventManager
torEventManagerImpl.h torEventManagerImpl.cpp	生成されたイベント マネージャの状態機械実装
torEventMedia.h torEventMedia.cpp	EventMedia, LogFile, TauHostTracer
torEventQueue.h torEventQueue.cpp	EventQueue
torExceptions.h	例外タイプの定義
torExitPoint.h	ExitPoint
torMeta.h	メータデータに関連する全ファイルをインクルードする共通インクルード ファイル
torMetaData.h torMetaData.cpp	Definition, StructuralFeature, Operation, Application
torMetaEvents.h torMetaEvents.cpp	Event, Call, Called, Create, Delete, EventReceived, , EventSent, Exit, Return, Returned さらにデバッグコマンドごとに 1 つのメタイベント
torMetaStackContext.h torMetaStackContext.cpp	StackContext
torOperators.h	equal, implies, mod, power、および UML 演算子を表す他の関数
torOs.h torOs.cpp	すべての OS 抽象レイヤファイル用の共通インクルードファイル
torOs_X.h torOs_X.cpp	オペレーティング システム抽象化レイヤの実装。X はオペレーティング システム名を示す。 torOs_any.h と torOs_any.cpp の 2 つが存在し、新規 OS 用の OS 抽象レイヤ実装の際にテンプレートとして使用可能
torOSRep.h torOSRep.cpp	TGTOS マクロ設定によってどちらの実 OS ファイルをすべきかのスイッチ
torPlatform.h	TOR ライブラリのためのプラットフォーム固有のビルド設定。このファイルは TOR ライブラリのすべての実装ファイルにインクルードされます

ファイル	TOR 宣言
torPort.h torPort.cpp	Port
torProcess.h torProcess.cpp	Process
torRegion.h torRegion.cpp	Region
torState.h torState.cpp	State
torStateMachine.h torStateMachine.cpp	StateMachine
torSync.h torSync.cpp	Mutex, RWLock, Semaphore, Gate
torThread.h torThread.cpp	Thread
torThreadSafeEventQueue.h torThreadSafeEventQueue.cpp	ThreadSafeEventQueue
torTime.h torTime.cpp	Time
torTimer.h torTimer.cpp	TimerEvent, TimerObject, TimerQueue
torTopRegion.h torTopRegion.cpp	TopRegion
torTypes.h	単純なデータ型 , Any クラス , Charstring クラス
torUtilities.h torUtilities.cpp	sendTo, cast, setTimeUnit, initializeModel

TOR インテグレーション ガイド

本ガイドでは、C++ ランタイムライブラリを新しいオペレーティングシステムに適合させる方法を説明します。

Tau には、オペレーティングシステムとのインテグレーションのサンプルが同梱されており、Tau とともにインストールされます。初めてインテグレーションを行う場合はこのサンプルからスタートするのがよいでしょう。

ＯＳ プリミティブ

このセクションでは、C++ ランタイムライブラリが使用する「低レベル」のオペレーティングシステムプリミティブについて説明します。各プリミティブは C++ ランタイムライブラリによってほぼ完全に「抽象化」され、生成コードは下層であるオペレーティングシステムに依存しないようになっています。

これらのプリミティブの属する名前空間は “tor::os” です。ほとんどの場合、各プリミティブは特定のクラスで抽象化されています。これらのクラスが下層のオペレーティングシステムにおけるプリミティブの実装を使用しています。

これらの 抽象化されたプリミティブ は、すべて “torOs_xxx.h” および “tor_Os_xxx.cpp” (“xxx” はオペレーティングシステムの名前) という名前のファイルに存在します。2 つのファイル名 “torOs_user.h” および “torOs_user.cpp” は、特定の既存オペレーティングシステムではなく、ユーザー定義による実装として使うことを意図して確保されています。つまり、新しいオペレーティングシステム とのインテグレーション作業を開始するには、既存の “torOs_xxx.cpp” および “torOs_xxx.h” を “torOs_user.cpp” および “torOs_user.h” へとコピーして、ベースとして使い始めるのがよいでしょう。

“torOsRep.h” ファイルは、プリプロセッサマクロ “TGTOS” の設定値に対応した “torOs_xxx.h” ファイルをインクルードします。このマクロは以下の表に示した値をとることができます。この値は “torOsRep.h” に定義されています。

名称	値	説明
OS_USER	0x0100	ユーザー定義用。
OS_WIN32	0x0200	Microsoft Windows オペレーティングシステムの 32 ビット版用。
OS_LINUX	0x0300	汎用 Linux インテグレーション用。
OS_SOLARIS	0x0400	Solaris インテグレーション用。

マクロ “TGTOS” が他で明示的に定義されていない場合は、“torOsRep.h” ファイル内のプリプロセッサ ステートメントでホストオペレーティングシステムに対応した値を選択するようになっています（たとえば、Linux システム上では、値 ”OS_LINUX” が選択されます）。

Time

Time は抽象化クラス “tor::os::Time” で定義され、“tor::os::TimeRep” で実装されます。Time クラスは、絶対時間（オペレーティング システム上での）と相対時間（期間）の両方を表現します。サンプルのインテグレーションにおいては、OS の時間表現に対応させてあり、この時間を使用してセマフォでのタイマー Wait やスレッドのタイマー中断を行います。これにより、一般的な表現と特定 OS 上での表現の間の不必要な変換を行わなくてすみます。

“TimeRep” クラスは以下のインターフェィスを実装する必要があります（ただし時刻の値を保持する属性は割愛してあります）。

```
class TimeRep {
public:
    TimeRep();
    TimeRep(long s, long ns);
    TimeRep(const TimeRep &a, const TimeRep &b);
    TimeRep(const TimeRep &a);          //  = a
    ~TimeRep();

    void norm();
    void zero();
    void setNow();

    void set(const TimeRep &x);
    void set(long s, long ns);
    void set(double d);

    void add(const TimeRep &x);
    void add(long s, long ns);

    void sub(const TimeRep &x);
    void sub(long s, long ns);
    void sub(const TimeRep &a, const TimeRep &b);

    int cmp(const TimeRep &x) const;

    double to_double() const;
    long to_ms() const;

    std::ostream &print(std::ostream &os) const; // optional

    static double setUnit(double u);

};
```

デフォルトコンストラクタ “TimeRep()” は、各値を論理ゼロ (0x00) に初期化する必要があります。これは以下の処理を行えるようにするためです。

```
TimeRep a, b; b.setNow(); a.add(b); if (a.cmp(b) == 0) { OK;
}
```

“zero()” 操作も結果は論理ゼロになる必要があります。

コンストラクタ “TimeRep(long s, long ns)” は、与えられた秒数およびナノ秒数 (10-9s) を使用して変数を初期化します。“set(long s, long ns)” 操作も同様に値を設定する必要があります。

コンストラクタ “TimeRep(const TimeRep &a, const TimeRep &b)” は、論理差 “a - b” で変数を初期化します。“sub(const TimeRep &a, const TimeRep &b)” 操作も同じ値を設定します。

“norm()” 操作は、ヘルパー機能です。算術演算の後に値を正規化します。

“setNow()” 操作は現在の時刻を割り当てます。

“int cmp(const TimeRep &x) const” 操作は、オブジェクトが別のオブジェクト x よりも「小さい」場合は、「-1」を返します。2 つのオブジェクトが「等しい」場合は、「0」を返します。オブジェクトが別のオブジェクト x よりも「大きい」場合は、「+1」を返します。

“double”(倍精度浮動小数点数) への変換、または “double” からの変換には、内部の単位乗数が使用されます。この単位乗数を設定するには “setUnit()” 操作を行います。

時刻の実表記は POSIX では “struct timespec” として、Win32 では “long m_s, m_ns;” のペアとして実装されます。

Mutex

排他ロックを抽象的に表現したのが “tor::os::Mutex” クラスです。このクラスは “tor::os::MutexRep” クラスによって実装されます。“Mutex” は最も基本的な排他制御である無条件ロックと無条件アンロック をサポートします。

“MutexRep” クラスは次のインターフェィスを実装する必要があります (ただし相互排他ロックの値を保持する属性は割愛してあります):

```
class MutexRep  {
public:
  MutexRep();
  ~MutexRep();
  Status init(bool initialOwner);
  Status destroy();
  Status lock();
  Status unlock();
};
```

注記

相互排他ロックの初期化と終了処理を、それぞれ、コンストラクタおよびデストラクタから切り離していることに注意してください (“Mutex“ クラスにおいても同等です)。これにより、相互排他ロックのグローバルなインスタンスとしての再初期化が可能になります。こういった役割が必要でない場合は、初期化と終了処理をコンストラクタおよびデストラクタに実装することも可能です (この場合には、init() と destroy() は空になるでしょう)。

“lock()” 操作および “unlock()” 操作は無条件に行われます。操作の成否によって、それぞれ “StatusOK” または “StatusFAIL” を返す必要があります。

相互排他ロックを保持する属性は 、POSIX の場合 “pthread_mutex_t m_mutex” として定義され、Win32 の場合 “HANDLE m_handle” として定義されます。Win32 では、ハンドルは “CreateMutex()” 操作を使用して初期化されます。

Semaphore

セマフォを抽象的に表現したのが “tor::os::Semaphore” クラスです。このクラスは “tor::os::SemaphoreRep” クラスによって実装されます。“Semaphore” は無条件ロック / アンロックとタイマベースのロック（呼び出しスレッドはタイムアウト値到達までセマフォ上で停止）を提供します。

“SemaphoreRep” クラスは以下のインターフェィスを実装する必要があります（ただしセマフォの値を保持する属性は割愛してあります）。

```
class SemaphoreRep {
public:
  SemaphoreRep();
  ~SemaphoreRep();

  Status init(unsigned int initialCount, unsigned int
maxCount);
  Status destroy();

  Status put(unsigned int count);

  Status get();
  Status get(Time timeout);
};
```

Mutex と同様に、相互排他ロックの初期化と終了処理を、それぞれ、コンストラクタおよびデストラクタから切り離しています。タイムアウト状況取得の “get(Time timeout)” 操作のタイムアウト値が絶対時間であることに注意してください。

POSIX では、次の 3 つの属性がセマフォの実装に使用されています :

```
pthread_mutex_t m_mutex;
pthread_cond_t m_cond;
unsigned m_count;
```

Win32 では、ハンドルの初期化に “CreateSemaphore()” を使用します。

Thread

Thread は名前空間 “tor::os::Thread” の 3 つの操作によって実装されます :

```
bool createThread(ThreadFunc func, void *arg);
void suspend(Time timeout);
unsigned long id();
```

“createThread()” 操作は、1 つの引数を持つ関数である “ThreadFunc” を持つ新しいスレッドを作成します。POSIX では、スレッドは「切り離された状態 (detached state)」で作成されます。

“suspend()” 操作は絶対時間を使用します。POSIX では、“nanosleep()” を使用して実装されます。Win32 では “Sleep()” を使用します。

“id()” 操作はスレッドごとに固有の値となる id を返します。

Process

Processe は通常のインテグレーションでは使用されません。使用する場合は以下のインターフェイスの実装を提供する必要があります。

```
class ProcessRep {
public:
    typedef Charstring::inherited string;
    ProcessRep();
    ~ProcessRep();
    bool launch(const string &cmd, const std::vector<string>
&argv, const string &dir);
    bool terminate();
    PID getPID() const { return m_pid; }
}
```

コンストラクタとデストラクタは空にします。“launch()” 操作が外部のプロセスを起動します。“terminate()” 操作は外部のプロセスを停止し、完了後にリターンします。

ビルド

このセクションでは、ユーザー定義ツールを使用してアプリケーションをビルドをするために必要な設定を簡単に説明します。本セクションを読むにあたっては、Makefile ジェネレータのセクションも参照してください。

まず必要なのは、モデルとそれに関連するビルドアーティファクトです。また、前のセクションで説明した変更事項を 2 つのファイル、“torOs_user.cpp” と “torOs_user.h” に反映してあることを前提とします。コンパイラ、リンカ、make その他ビルドに必要なツールのパスが正しく設定されており使用できることも前提とします。

最初に <<MakefileGenerator Settings>> ステレオタイプをビルドアーティファクトに付加して、Makefile ジェネレータの振る舞いを変更できるようにします。このステレオタイプは ”MakefileGen” アドインで定義されており、［ツール］>［カスタマイズ］>［アドイン］タブからアクティブにできます。続いてビルドアーティファクト上で右クリックして［ステレオタイプ］を選択するか、［プロパティ］から［プロパティの編集］ダイアログを開いて、“MakefileGen::Makefile Generator Settings” をビルドアーティファクトに適用します （注記 : 類似のステレオタイプ “Makefile Generator” がありますが、ここでは使用しません）。

［プロパティの編集］ダイアログでステレオタイプの設定を変更します。“Target Kind” の設定値としてターゲットシステムにもっとも近いものを選びます。ここで、ビルドアーティファクトを右クリックして、［ビルド (C++ Application Generator)］>［生成］からコード生成を行うと、コードとともに Makefile が生成されます。ほとんどの場合、コンパイラなど向けの設定値を変更する必要があります。つまり、ビルドアーティファクトの “Makefile Generator Settings” ステレオタイプの “User Code” セクションにエントリを追加を行います。このセクションの各行は、Makefile 中の対応する make 変数に指定された値を上書きします。

プリプロセッサマクロ “TGTOS” は “OS_USER” に設定されている必要があります。これは、前のステップで生成された Makefile 中のプリプロセッサ定義を設定している行をコピーして行います。つまり、“Target Kind = Linux - g++” の場合、“User Code” セクション は以下のようになります :

```
DEFINES=$(TAUDEFINES) -D_REENTRANT -DTGTOS=OS_USER
```

“Target Kind = Win32 - cl” の場合は以下のようになります :

```
DEFINES=$(TAUDEFINES) /D "WIN32" /D "NDEBUG" $(SUBSYSDEF)
/DTGTOS=OS_USER
```

いずれの場合も、この 1 行のみを記述します。

同様に、C++ コンパイラ設定、コンパイラフラグ、リンカ設定、make 対象プログラムなどを変更する必要がある場合は、“User Code” セクションに追加してゆきます。

make 変数および Makefile ジェネレータの設定の一覧は、Makefile ジェネレータのセクションを参照してください。

44

C++ アプリケーションのデバッグ

UML Debugger により、UML モデルの振る舞いをデバッグして、実装が正しいことを確認できます。

UML Debugger によるデバッグ機能を備えたアプリケーションをビルドするには、C++ アプリケーション ジェネレータでアプリケーションをビルドする場合と同様な手順を実行します。このセクションでは、基本的なビルド機能と UML Debugger の使用方法について説明します。

参照

第 6 章「アプリケーションのベリファイ」

UML Debugger の概要

UML Debugger を使って、UML モデルの振る舞いをデバッグして、実装が正しいことを確認できます。アプリケーション ビルダを使用して、モデルから C++ コード実行形式プログラム、つまりデバッグ機能を備えたアプリケーションを生成し、シミュレーション用にカスタマイズされた定義済みランタイム ライブラリにリンクします。デバッグ機能を備えた C++ ライブラリをビルドする場合にも、UML Debugger も使用できます。モデルをシミュレートするとは、さまざまなコマンドとブレークポイントを使用して実行形式プログラムを実行するという意味です。シミュレーションを自動的に実行することも、手動で遷移を段階的に追跡することもできます。

ユーザインタフェースから UML Debugger を制御するか、または Visual Studio .NET デバッグセッションに移行できます。アプリケーションと環境との間でやりとりがある場合、この振る舞いをターゲットの C++ デバッガ（Visual Studio .NET）によって追跡可能であり、UML Debugger で実行を制御し、Visual Studio .NET デバッガで値の検査および設定、スレッドと呼び出しスタックの観察を行なうことができます。

セッションの実行は、状態機械図、クラス図およびテキスト図でグラフィカルにトレースできます。

デバッグ機能を備えたアプリケーションの生成

注記

実行形式の UML Debugger アプリケーションを生成するには、C/C++ コンパイラをインストールしておく必要があります。

UML Debugger にはデバッグ機能を備えたアプリケーションの生成が必要です。デバッグ機能を備えるには以下を実行します。

1. 生成時に追加のデバッグコードを記述する。
2. デバッグ機能付き TOR ライブラリを使用してアプリケーションにリンクする。

デバッグ機能の配備はアプリケーションの振る舞いには影響を及ぼしませんが、実行中に UML Debugger による制御を受けます。

ビルド設定

生成する各実行形式ファイルについて、ビルド アーティファクトが存在しなければなりません。デバッグ可能な実行形式ファイルを生成するには、このアーティファクトにデバッグ機能を実装する必要があります。これは、C++ アプリケーション ジェネレータのステレオタイプの［プロパティの編集］ダイアログで設定します（ステレオタイプのプロパティは［フィルタ］メニュー フィールドで切り替えます）。

以下の設定をよく考慮し、ニーズに合った設定を行ってください。

- Link with TOR
- 装備に関するオプション
- デバッグ オプション

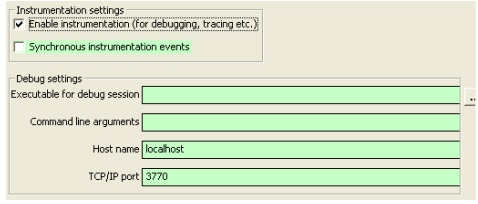


図 260: 機能実装とデバッグのためのオプション

デバッグ機能をもつアプリケーションはビルドアーティファクトを作成することで生成できます。詳細は、ビルド アーティファクトを使用したビルドを参照してください。

ビルド設定が複数のビルド アーティファクトのために同時に使用される場合は（ビルド設定の使用参照）、そのうちの 1 つのみがビルドアーティファクトが UML Debugger ビルド アーティファクトである必要があります。これは同時に実行できる UML Debugger のインスタンスが 1 つに限られるためです。

参照

第 20 章「アプリケーション ビルド リファレンス」

UML Debugger の実行

UML Debugger をエラーなしで正しくビルドできたら、UML Debugger を起動できます。

UML Debugger の起動

UML Debugger を起動するには、実行中のアプリケーションに接続するか、またはビルド アーティファクトから UML Debugger のビルドと起動を実行する［起動］を使用します。［ベリファイ］ツールバー上の適用可能なコマンドがアクティブになります。

実行中のアプリケーションへの接続

すでに実行中の実行形式ファイルに接続するとき、この実行形式ファイルが、モデルで使用した同じビルド アーティファクトでビルドされていなければなりません。

1. たとえば、Visual Studio .NET を使用する、またはコマンドプロンプトから直接入力するなどして、アプリケーションを起動します。
2. ［ベリファイ］メニューの［実行中のプログラムにアタッチ］コマンドで実行中のアプリケーションを UML Debugger に接続します。

アタッチの手順の一部として、Tau はアタッチされるプロセスが UML Debugger に対応するようにデバッグ機能をもっているかを簡単にチェックします。

5 秒以内にデバッグ対象のアプリケーションを検知できないと、Tau はそれ以上待つか、それともアタッチを中止するかをたずねてきます。

アタッチ前に終了していたなど、アプリケーションにアタッチできない場合があります。そのような場合には、アプリケーションの開始前に環境変数 TTD_TOR_WAIT_FOR_ATTACH を設定します。この環境変数が設定されていると、アプリケーションは、最初のステートメント実行前に待ち状態に入ります。これによって Tau がアプリケーションに接続する時間を作ることができます。

もう 1 つの環境変数 TTD_TOR_HOST_DBG を設定することもできます。この変数は “hostname#portnumber” という文字列形式で設定が可能であり、デバッグセッションで使用する TCP/IP 設定を指定します。この変数を設定するのは、デフォルトの TCP/IP ホストやポートの値を変更したい場合です。

ビルド アーティファクトを使用したビルド後の起動

この方法はすでに UML Debugger のビルド アーティファクトを作成している場合に便利です。

1. ［モデル ビュー］で、起動する UML Debugger を表すビルド アーティファクトを右クリックします。
2. ショートカット メニューから［**ビルド (**C++ Application Generator**)**］を選択して、メニュー項目［起動］を選択します。

UML Debugger がビルドされます。ビルドに成功した場合は、新規にビルドされた UML Debugger が起動されます。

構成を使用したビルド後の起動

この方法は、複数のビルド アーティファクトをビルドし、そのうちの 1 つを UML Debugger とし、新規にビルドした UML Debugger を起動する場合に便利です。1 つの特殊な状況として、構成に UML Debugger ビルド アーティファクトが 1 つのみ含まれている場合があります。

- ［プロジェクト］ツール バーの［構成の実行］ボタンをクリックします。アクティブな構成に含まれているすべてのビルド アーティファクトのビルドが行われ、新規にビルドされた UML Debugger が起動されます。

UML Debugger の終了

UML Debugger を終了するには、以下のいずれかの手順を実行します。

- ［ベリファイ］メニューから［停止］を選択します。
- デバッグ対象のアプリケーションを停止します。たとえばコンソール ウィンドウを閉じます。

実行のトレース

UML Debugger を実行しているときは、簡単に実行トレース情報を入手できます。したがって、アプリケーション内の各遷移とイベントを追跡できます。以下のトレース方法から選択できます。

- UML モデルの追跡
- Visual Studio .NET インテグレーションによるシーケンス図トレース

UML モデルの追跡

この追跡方法では、UML モデル内の各アクションを追跡できます。追跡を開始すると、選択されたアクションのダイアグラムが表示されます。追跡を続行するに従って、ダイアグラム内の次の文が順次にハイライト表示されます。

実行直前のシンボルまたはシンボル内の文の横に緑の三角形が挿入されます。

シーケンス図トレース

シーケンス図トレースは、Visual Studio .NET インテグレーションを使用して有効にします。トレースのワークフローは以下のとおりです。

- トレースするアプリケーションを含むワークスペースを開く。
- Enable instrumentation オプションを有効にしてコードを生成していることを確認する。
- Visual Studio .NET インテグレーションを使用して、Visual Studio .NET プロジェクトを作成して開く。
- Visual Studio .NET でデバッグ セッションをビルドして実行する。
- コードの先頭にブレークポイントを設定します。コードがブレークポイントで停止したら、（Tau ツールバーから）Tau Trace コマンドを使用してトレースを起動します。［Sequence diagram visualization］を選択します。

Tau のシーケンス図の形式でトレースが開始されます。

参照

Visual Studio インテグレーションを使用しないシーケンス図トレースを取得する方法は、Tau で生成されたアプリケーションの実行のトレース を参照してください。

アプリケーションの実行

実行を開始するために使用できる複数のコマンドがあります。どちらのコマンドを使用するかは UML Debugger のモードに依存します。以下の 2 つのモードがあります。

1. ***Run mode***
   このモードはアプリケーションが現在実行されていることを意味します。ブレークポイントの設定と解除、および実行の中断（Break Mode）などのコマンドが利用できます。詳細は Run mode コマンド と Breakpoint コマンド を参照してください。
2. ***Break mode***
   このモードはアプリケーションが現在実行されておらず、デバッグコマンド入力待ちで中断していることを意味します。ブレークポイントの設定と解除、および実行の再開（Run mode）などのコマンドが利用できます。詳細は Break mode のコマンド と Breakpoint コマンド を参照してください。

Break mode のコマンド

UML Debugger が Break mode にある場合、実行状態への復帰は、実行を制御するコマンド（[ベリファイ] メニューおよびツールバー）によって行われます。さらに Breakpoint コマンド も使用できます。

Go

アプリケーションの実行を再開します。ブレークポイントに遭遇するか、Break Execution コマンドが入力されるか、アプリケーションの終了まで、実行は継続します。

Step Over

アプリケーション実行を次のアクションへと進めます。そのアクションが操作の呼び出しである場合は、その操作の実装には入らず、操作の終了後へと制御が移ります (ステップオーバー)。

Step Into

アプリケーション実行を次のアクションへと進めます。そのアクションが操作の呼び出しである場合は、その操作の実装に入り、実装の 1 行目で中断します。

Step Out

アプリケーション実行を、現在実行している操作実装から抜けるまで進めます。

Run execution up to this point

このコマンドは、ダイアグラム中のアクション上でクリックしたときに、コンテキストメニューで使用できます。アプリケーション実行は選択したアクションまで進められます。

Run mode コマンド

UML Debugger が Run mode にある場合、その実行を実行中断コマンド（[ベリファイ] メニュー）を使用して中断できます。さらに Breakpoint コマンド も使用できます。

Break Execution

アプリケーション実行は即時に中断し、Break Mode に入ります。

Breakpoint コマンド

ブレークポイントを設定して、任意の箇所で実行を中断できます。

ブレークポイントの挿入

ブレークポイントを挿入するには、以下の手順を行います。

1. ブレークポイントの挿入先としたいシンボルが含まれている状態機械図を開きます。
2. シンボルを右クリックして、ショートカット メニューから［ブレークポイントの挿入 / 削除］をクリックします。ブレークポイントが挿入されたことを示す赤いドットがシンボル フレームまたはシンボル フレーム内の文の隣に追加されます。

［ベリファイ］ツール バーから［ベリファイ］メニューのコマンドを使用して、ブレークポイントを挿入することもできます。

ブレークポイントの正確な位置付け

以下の例は、ブレークポイントを設定して目的の位置を正確に特定する方法を示しています。

- ステートメント : アクション シンボルに複数の文が含まれている場合、ブレークポイントを特定の文に設定するには、カーソルを文の上に置きます。これはテキスト図のブレークポイントの設定にも適用されます。
- ネクストステート : ネクストステートアクションにブレークポイントを挿入するには、ステートシンボルに向かうコネクタライン上に挿入します。
- アクションシンボルとライン（Output、Decision、Stop、Return、nextstate への Flow-line、History、Task、ジョインを現す Junction シンボル）: ブレークポイントは対応するアクションに設定されます。

- 遷移シンボルとライン（Start、DecisionAnswer、Input、ラベル遷移を表す Junction シンボル、遷移ライン、Connect 遷移（つまり、合成状態の後のガード シンボル））：ブレークポイントは遷移の複合アクションに設定されます（つまり、遷移が発生した場合に起動する最初のアクションに設定されます）。
- ステート（テキストまたはブラウザ内で選択されたもの）：ブレークポイントはステートを参照するすべての nextstate アクション上に設定されます（つまり、このブレークポイントはステートに入った場合は常にヒットします）。
- ステートシンボル：ブレークポイントはステートシンボルに含まれるあらゆるステートを参照するすべての nextstate アクション上に設定されます。アスタリスク ステート上や排他的なステートをもつステート上にはブレークポイントを設定できません。
- 操作本体シンボル（テキストまたはブラウザ内で選択されたもの）：ブレークポイントは本体の複合アクション（つまり、本体が実行された場合に最初に実行されるアクション）に設定されます。
- 操作（OperationSymbol、クラスシンボル中の OperationLabel またはテキスト内で選択されたもの）：ブレークポイントは本体の複合アクションに設定されます。操作が本体を持たない場合（抽象操作またはインターフェース操作）は、ブレークポイントはデバッグされるアプリケーション内では設定できません。
- 状態機械実装（モデルビューで選択されたもの）：ブレークポイントはあらゆる開始遷移における最初のアクションに設定されます。

ブレークポイントの削除

ブレークポイントを削除するには、以下の手順を行います。

1. ブレークポイントを削除するアクションが表示されているダイアグラムを開きます。
2. シンボルまたはテキストを右クリックして、ショートカット メニューから［ブレークポイントの挿入 / 削除］をクリックします。

ブレークポイントのリストの表示

モデルビューの Model Verifier Session パッケージの欄に設定されたすべてのブレークポイントが表示されます。これらのブレークポイントをダブルクリックすれば、対応するアクションに移動します。

コンパイルされたブレークポイント

モデルそのものに永続的にブレークポイントを設定することが望ましい場合があります。そのようなブレークポイントはモデル内では、`tor` ライブラリパッケージ内の `tor::debugBreak` への呼び出しとして表現されます。この呼び出しが実行されるとアプリケーションはブレークモードに入ります。

コンパイルされたブレークポイントの挿入はモデルを修正するため、再コンパイルが必要になります。

ソースへのステップイン

ステップ全体にわたってアプリケーションを実行する場合、選択は［制御をターゲット デバッガに移す］コマンドの実行結果に影響を及ぼします。

［制御をターゲット デバッガに移す］を選択すると、現在の実行ポイントについてコードがチェックされます。このポイントは、デバッグセッションを停止中に存在します。現在の実行ポイントにソース コード参照が存在する場合、これが使用されます(予想される結果です)。明示的なソース コード参照を持たないエンティティがある場合、コマンド ボタンが無効になります。このような場合、実行直前のコード内で選択を行います。この現在の選択が、［制御をターゲット デバッガに移す］コマンドの参照として使用されます。

エラー処理

UML の動的ルールに違反すると、シミュレーションの実行中に動的エラーが発生します。動的エラーの例としては、ヌル参照アクセス、索引範囲外のコンテナ操作、メモリアクセスエラー、キャッチされない例外、などが考えられます。

動的エラーが検出された後は、デバッグセッションは終了します。

45

C++ 言語向け Visual Studio .NET インテグレーション

Tau Visual Studio .NET インテグレーションにより、生成された C++ アプリエーションのビルドとデバッグを行うことができます。詳細については、Visual Studio .NET インテグレーションの説明を参照してください。

UML と要求

本章では、UML において要求をどのようにモデリングするのか、DOORS に定義された要求とどのように協調して動作するのかを説明します。

46

要求のモデリング

本章では要求を UML でモデル化する方法を説明します。

はじめに

Tau による要求のモデル化を始めるには以下の手順を行います。

- Tau を開始して新規プロジェクトを作成するか既存プロジェクトを開きます。
- [Requirements] アドインをアクティブにします。Requirements アドインの活動化 を参照してください。
- 要求ビュー に切り替えます。（オプション）
- 要求図 を作成してモデリングを開始します。

参照

DOORS との協調

要求のインポート

リンクを使った作業

Requirements アドイン

Requirements アドインは Tau における要求モデリングをサポートします。

主要な機能は以下のとおりです。

- 要求 プロファイル を実装
- 要求中心のモデルビュー 要求ビューを提供
- 要求中心のプロパティビュー 要求プロパティビュー を提供
- 多彩な 要求レポート

アドインを活動化する方法は、Requirements アドインの活動化 を参照してください。

Requirements アドインの活動化

要求モデリング機能はアドインとして提供されます。このアドインは使用するプロジェクトごとに手動にて活動化する必要があります。アドインを活動化する手順は以下のとおりです。

1. [ツール] メニューから [カスタマイズ ...] を選択します。
2. ［アドイン］タブをクリックし、[Requirements] アドインをチェックします。
3. ［OK］をクリックします。

要求ビュー

［要求ビュー］はモデルについて要求中心のビューを提供します。このビューでは、要求とその間の関連の表示についてフォーカスします。

要求ビューを活動化するには以下の手順を行います。

1. ［表示］メニューで［モデルビューの再構成 ...］を選択します。
2. ダイアログで［Requirement View］を選択し、［OK］をクリックします。

［Standard View］と［Requirement View］は任意のタイミングで切り替え可能です。

参照

第 1 章「Tau 4.3 の紹介」の 6 ページ、「モデル ビュー」

第 72 章「ダイアログ ヘルプ」の 2158 ページ、「デフォルトのモデル ビュー」

要求プロパティビュー

［要求プロパティビュー］はある要素のプロパティについて、要求中心の表示を提供します。

要求プロパティビューを活動化するには以下の手順を行います。

1. 特定の要素を選択して［プロパティエデイタ］を開きます。
2. ［オプション ...］をクリックします。
3. プロパティビューを［Requirement Property View］に変更します。
4. ［OK］をクリックします。

［Standard Property View］と［Requirement Property View］は任意のタイミングで切り替え可能です。

参照

第 2 章「モデルの操作」の 66 ページ、「プロパティ エディタのオプション」

要求レポート

要求とその関連について数多くのレポートが利用できます。

以下のレポートが要求を一覧表示します。

- 要求レポート
- 未達成要求レポート
- 未検証要求レポート

以下のレポートが要求間の関連を一覧表示します：

- 要求関係レポート

要求をリストするレポートとして 基本レポート または 詳細レポート が使用できます。

基本レポート

このレポートは常に利用可能であり、選択した要素の基準に合致したすべての要求をリストします。

各 要求 について以下の情報が表示されます。

- 名称
- 識別子
- テキスト
- 見出し

詳細レポート

Requirement アドインと DOORS フォーマル モジュール をともに使用すると、基本レポートとともに詳細レポートも利用できます。

詳細レポートはフォーマルモジュールを表現するパッケージかフォーマルモジュール内の単一の要求に対してのみ利用可能です。それは、属性のセットはモジュールごとに異なる可能性があるためです。

基本レポートの 3 つのカラムに加えて、詳細レポートでは、対応する DOORS モジュールで属性によって指定されて表示されている各属性カラムにつき 1 つのカラムが表示されます。

注記

DXL レイアウトカラムは Tau では表現されないため、レポートにも現れません。

要求レポート

単純にすべての要求をリストします。

未達成要求レポート

選択された要素についての未達成の要求をすべてリストします。ある要求が未達成であると見なされるのは、内向きの satisfy 関係 がない場合です。

未検証要求レポート

選択された要素についての未検証の要求をすべてリストします。ある要求が未検証であると見なされるのは、内向きの verify 関係 がない場合です。

要求関係レポート

選択された要素についての要求の関連をすべてリストします。すべての要求の関係 が考慮対象です。

各関連について以下の情報が表示されます。

- 関連の種別
- ソース要素の名称
- ソース要素の種別
- ターゲット要素の名称
- ターゲット要素の種別

要求 プロファイル

要求プロファイルは、要求モデリングのコンセプトに基づいて UML を拡張します。要求プロファイルは OMG において SysML の一部として標準化されています。

基本事項

要求プロファイルは、要求を表現する <<requirement>> ステレオタイプと要求と他の UML モデル要素間の関係を指定する一連のステレオタイプを定義します。これらのコンセプトを統合して要求仕様とトレーサビリティ（追跡可能性）に対するサポートを提供します。

また、要求と要求間の関係を編集するための新しいダイアグラムタイプ、要求図（Requirements diagram）が導入されます。

以下のセクションでは、要求の指定に有用なあらゆるコンセプトがリストアップされて説明されます。ほとんどのコンセプトは要求プロファイルにて実現されます。他の一部は標準の UML や UML Testing Profile にて実現されます。ここでは一覧を完全にするためにそれらもリストします。

要求

<<requirement>> はステレオタイプ化された Class であり要求を表現します。要求には以下のプロパティがあります。

- テキスト：要求のテキスト
- 見出し：要求の見出しテキスト
- 識別子：要求の識別子

要求は任意の深さに入れ子にして、要求の階層を現すことができます。

要求の関係

このプロファイルでは、依存としてステレオタイプ化される一連の要求の関係を指定します。各関係は以下のセクションで説明します。

trace 関係

<<trace>> 関係は、標準の UML 概念であり、モデル要素間の任意の種類の変更を追跡するために使用されます。要求トレーサビリティにも適用可能です。

<<trace>> は Requirements プロファイルには定義されていませんが、要求とともに使用されることが多いためにここに挙げています。

copy 関係

2 つの要求間の <<copy>> 関係は、クライアント側の要求のテキストが供給側のテキストの読み取り専用のコピーであることを指定します。

コピー関係は、マスター / スレーブ関係の一種であり、要求の再利用の指定を意図したものです。

注記

クライアント側のテキストは、自動では、供給側のテキストとは同期状態を維持できません。手動で同期する必要があります。

DeriveReqt 関係

<<deriveReqt>> 関係は、要求が他の要求に由来することを指定するために使用されます。たとえば、SRD のシステム関係は通常 URD のユーザー要求に由来します。

Refine 関係

<<refine>> 関係は、標準的な UML 概念です。これは、モデル要素（クライアント）が他のモデル要素（サプライヤ）を精微化するために使用されます。

<<refine>> は要求プロファイルでは定義されませんが、要求と一緒に使用されることが多いため、ここに記載しました。

satisfy 関係

<<satisfy>> 関係は、モデル要素が、ある要求を満足するか、または実装するかを指定するために使用されます。

verify 関係

<<verify>> 関係は、テストケースが、ある要求を検証しているか、テストしているかを指定するために使用されます。

テストケース

テストケースは 1 つ以上のモデル要素のテストの振る舞いを指定する操作です。テストケースは UML テスト プロファイル で定義されます。

注記

テストケースは、UML テストプロファイルサポートがアクティブになっている場合のみ使用可能です。詳細は テスト プロファイル サポートの有効化 を確認してください。

要求図

要求とそれらの関係を表示、編集するためのダイアグラムです。

要求図には、要求、要求の関係、および他の関係要素が含まれます。

ツールバーを使用して作成できるのは、要求、パッケージ、および要求の関係です。他の種類の要素を表示するには、それらをモデルビューからダイアグラム上にドラッグします。

注記

要求ビューではすべての要素が表示されているわけではないことに注意してください。したがって、[モデルビュー] から要素をドラッグした場合は、標準ビューなどに切り換えないと表示されないことがあります。

47

DOORS との協調

本章では、DOORS からの要求（要件）のインポート方法と Tau での要求を使った作業について説明します。要求管理に関する Tau の主要な機能は以下のとおりです。

- DOORS 要求を UML モデル内でビジュアルに表示する
- 要求と UML モデル要素の間のリンクを確立する

DOORS 要求は SysML の要求プロファイルにしたがって UML でモデル化されます。このプロファイルの詳細については、要求のモデリング を参照してください。

UML モデル / 要素の DOORS での表現方法については、DOORS 内の UML 要素を参照してください。

要求のインポート

要求は、DOORS インポートウィザード を使用して DOORS からインポート されます。

インポートウィザードを開始するには、以下の手順を行います。

- ［モデル ビュー］でモデルノードまたは任意の他のモデル要素を選択。
- ［ファイル］メニューから［インポート］を選択。
- ［DOORS からインポート］を選択。
- ［OK］をクリック。

DOORS インポートウィザード が起動します。

DOORS インポートウィザード

インポートウィザードは、まず DOORS に接続して、DOORS データベースのツリービューを表示します。DOORS が起動していない場合は、起動されてログインを要求されます。

いったん接続すると、データベースを表すツリービューが表示され、すべてのプロジェクト、フォルダ、フォーマルモジュールが表示されます。ツリービュー内の各ノードにはチェックボックスがあり、ここをチェックすることでインポート対象となります。

- インポートしたいフォーマルモジュール（複数可）を選択。
- 各モジュールに対して、オプションでインポート設定 を変更可能。
- [Import >] をクリック。

インポート処理が開始されます。進捗バーが表示されて処理の進捗を表示し、リストにメッセージが表示されます。

- ［完了］をクリックしてウィザードを終了。

インポートの結果に関する情報は インポートの結果 を参照してください。インポートされた要求に対して実行できる修正については、インポートした要求の修正を参照してください。

インポート設定

各フォーマルモジュールについて、以下のようなインポート設定を指定できます。

- ベースライン
  インポート中に使用するベースラインを指定する。デフォルトでは、最新のベースラインが使用される。
- ビュー
  インポート中に使用するビューを指定する。デフォルトは「Standard View」。
- ファイル
  インポートされたフォーマルモジュールを保管する .u2 ファイルの名前を指定する。デフォルトのファイル名はフォーマルモジュール名。

ベースラインとビュー (およびビューに適用されるフィルタ) はインポートの結果に影響を与えます。詳細は インポートの結果 を参照してください。

インポートの結果

次のセクションではフォーマルモジュールのインポートによって作成される要素を列挙して説明します。インポートされた要求に対して実行できる修正については、インポートした要求の修正を参照してください。

フォーマル モジュール

フォーマルモジュールは、要求のインポート で説明している表記を使用してインポートされます。フォーマルモジュールパッケージはモデルの最上位レベルに挿入され、インポート設定で指定したファイルに配置されます。

オブジェクト

各オブジェクトは要求のインポート で説明している表記を使用してインポートされます。

インポートで使用されるビューで確認できるオブジェクト（インポート設定で指定）だけがインポート可能です。

オブジェクト属性値は、そのオブジェクト上の <<requirementAttributes>> ステレオタイプのインスタンスにインポートされます。ステレオタイプの定義の詳細については、下の 属性 を参照してください。

属性

属性は、要求のインポート で説明している表記を使用してインポートされます。インポートされる属性はインポートで使用するビューのカラム (インポート設定 にて指定) に対応します。

<<requirementAttributes>> ステレオタイプが作成され、フォーマル モジュールパッケージに挿入されます。

注記 **1**

DXL レイアウト属性は Tau にインポートされません。

注記 **2**

<<requirementAttributes>> ステレオタイプは手動では変更しないでください。

リンク

リンクは常に trace 関係としてインポートされます。必要に応じて、いつでも Tau で関連タイプを手動で変更できます。

リンクのターゲットはシンボリック参照で指定されます。シンボリック参照と一致する要素がロードされると、リンクはその要素にバインドされます。それ以外の場合はリンクが保持されます。新しいモジュールをインポートすると、可能な場合はリンクが解決されます。

インポートした要求の修正

インポートされた要求は、Tau 内で修正し、その変更を DOORS に反映できます。Tau から DOORS への変更のコミットを参照してください。

フォーマルモジュール中の変更だけではなく、フォーマルモジュールパッケージそのものの変更や移動も行わないでください。フォーマルモジュールの他のスコープへの移動はすべてのリンクを無効にします。

DOORS 要素の UML 表記

このセクションでは DOORS 要素の UML 表記について説明します。内容は、要求 プロファイル に準拠しますが、若干の拡張が加えられています。

フォーマル モジュール

DOORS フォーマルモジュールは <<formalModule>> ステレオタイプを適用したパッケージとして表現されます。フォーマルモジュールのプロパティは以下の表のとおりです。

プロパティ	説明
Id	対応するフォーマルモジュールの ID
Baseline	モジュールをインポートする際に選択するベースライン
View	モジュールをインポートする際に選択するビュー。空の値は、最新バージョンが使用されていることを示します。
Link Module	リンクを DOORS にコミットするときにインテグレーションによって使用されるリンク モジュールへのフル パス。この値が空の場合、デフォルトのリンク モジュールが使用されます。
Last Synchronized	最新の同期化の日付と時間

注記

フォーマルモジュールプロパティの値はインポートおよび同期化の際に自動的に設定されます。手動では変更しないでください。
変更可能な値は Link Module のみです。

オブジェクト

DOORS オブジェクトは標準の 要求 表記を用いて表現されます。

要求プロパティと DOORS オブジェクト属性の間の対応付けは、下記の表のとおりです。

プロパティ	DOORS オブジェクト
Name	Object Heading
Id	Absolute Number
Text	Object Text
Heading	Object Heading

注記

要求の id はインポートおよび同期化の際に自動的に設定されます。手動では変更しないでください。

表

表は構造化された方法で表現されます。表自体と各行はプレースホルダ要素によって表現され、セルは実際の要求オブジェクトです。

```
Table
        Row1
                Cell1
                Cell2
                Cell3
        Row2
                Cell4
                Cell5
                Cell6
```

表の要素は、<<table>> でステレオタイプ化されたクラスとして表現され、行は <<row>> でステレオタイプ化されたクラスです。

属性

DOORS 属性は、UML では <<requirementAttributes>> ステレオタイプを使用して表現されます。各 フォーマル モジュール はそのモジュールで使用できる属性を表現する、このステレオタイプを含んでいます。このステレオタイプはフォーマルモジュールのインポート時に自動的に作成されます。

<<requirementAttributes>> ステレオタイプのインスタンスは 1 つのフォーマルモジュール内のすべてのオブジェクトに適用されます。このインスタンスは実際の属性値を保持します。

注記 **1**

DOORS のビューに加えられた変更は、Tau にインポートする前に保存しておく必要があります。たとえば、DOORS でモジュールに属性を追加した場合、先に DOORS でビューを保存しておく必要があります。

注記 **2**

<<requirementAttributes>> ステレオタイプはフォーマルモジュールのインポート中に自動的に作成されます。手動で変更しないでください。

リンク

DOORS リンクは標準の 要求の関係 表記、つまりステレオタイプ化された依存、として表現されます。

DOORS での要素の検索

UML モデル要素から対応する DOORS オブジェクトへとナビゲートするには、以下の手順を行います。

- ダイアグラムまたは［モデルビュー］で要素を右クリック
- コンテキストメニューから［DOORS で検索］を選択

この操作によって、DOORS がまだ起動していない場合は、起動され、要素が検索されます。インポートの際に使用したビューとベースラインが検索の対象となります。ビューがすでに使用不可になっている場合、またはオブジェクトがビューにおいて可視状態でない場合は、DOORS の［Standard view］が使用されます。

このコマンドは、同期化した UML 要素、インポートされたフォーマルモジュールを表現しているパッケージ、インポートされた要求を現す UML<<requirements>> クラスに対して使用できます。

このコマンドは、DOORS ツールバーまたは［Doors］メニューからもアクセスできます。

Tau から DOORS への変更のコミット

DOORS からインポートされたフォーマル モジュールに変更を加えることができます。サポートされている変更の種別については、サポートされる変更を参照してください。

Tau で行った変更を元のフォーマル モジュールにコミットするには、以下の手順を行います。

- ［モデルビュー］でフォーマルモジュール（に相当するパッケージ）を選択する
- 右クリックして［DOORS にコミット］を選択する

注記

旧バージョンの Tau でそのフォーマル モジュールを使用していた場合は、必ず旧バージョンの Tau からの移行に記載されている指示に従ってください。

特定のベースラインからインポートされたフォーマル モジュールの内容は変更できません。このようなモジュールについては、内向きのリンクのみ作成できます。

サポートされる変更

UML 要素に対する変更のうち DOORS へのコミットがサポートされるのは、下表に示した変更です。

注記

フォーマル モジュールの編集には注意が必要です。サポートされていない変更が、すべてユーザー インターフェイスによって禁止されているわけではありません。これは、DOORS インテグレーションが使用されない場合の要求モデリングにおける制限を減らすための意図的な仕様です。

要素	サポートされる変更	制限／コメント
要求	作成、削除、移動 名前と属性値の変更	Id プロパティは変更してはなりません。 要求をフォーマル モジュール間で移動してはなりません。 表内の要求作成、削除または移動できません。 表内の要求が子要求を持つことはできません。
要求関係（リンク）	作成、削除、移動、ターゲットの変更	リンクを作成するインテグレーションによって使用されるリンク モジュールは、フォーマル モジュールのプロパティとして指定されます。
要求図	すべての変更をサポート	-

要素	サポートされる変更	制限／コメント
フォーマル モジュール	-	属性定義（<<requirementAttributes>> ステレオタイプ） インポートしたフォーマル モジュールを移動したり、その名前を変更すると、「DOORS から Tau を更新」に説明するように、DOORS からモジュールを更新するときに問題が発生します。
表	-	表に対する変更は、含まれている要求のプロパティを変更することのみサポートされます。
行	-	行に対する変更は、含まれている要求のプロパティを変更することのみサポートされます。
他の UML 要素	すべての変更をサポート	-

表に明示的に記載されていない変更はサポートされません。たとえば、以下のような変更はサポートされません。

- 表の行の作成と削除
- 表の行への要求の追加

DOORS から Tau を更新

DOORS での変更を反映して Tau のモデルを更新する方法は、以下の 2 通りがあります。

- 最後の同期以降の変更を反映して更新
- 完全更新

DOORS での変更を反映して Tau のモデルを更新するには、以下の手順を行います。

- ［モデル ビュー］でフォーマル モジュールを選択する
- 右クリックして［DOORS から更新］を選択するか、必要な場合は［DOORS から完全更新］を選択する

最後の同期以降の変更を反映して更新

フォーマルモジュールを表す Tau のモデルを、最後の同期以降に行われた DOORS での変更を反映して更新できます。これは、パフォーマンスを最適化するための更新方法ですが、以下のような制限があります。

注記 **1**

特定のベースラインからインポートされたフォーマル モジュールの内容は変更できません。このようなモジュールについては、内向きのリンクのみ作成できます。

注記 **2**

DOORS で削除されたオブジェクトは常に Tau でも削除されます。これは DOORS のビューで可視であっても変わりません。削除されたオブジェクトを Tau で削除するには、パージされる前に同期する必要があります。

サポートされる変更

DOORS モジュールでの以下の変更がサポートされます。

- オブジェクトの作成と削除
- オブジェクトの移動
- オブジェクト属性値の変更
  - インポートの際に表示された属性のみが更新されます
- リンクの作成と削除

他の変更は現在のところサポートされていないため、操作は予期しない結果に終わる可能性があります。サポートされない変更の例は以下のとおりです。

- ビュー定義の変更
- フィルタの変更と適用
- 属性定義の変更

これらの変更が必要な場合は、完全更新によってモジュールを更新するか、モジュールを再インポートする必要があります。要求のインポートを参照してください。

完全更新

フォーマル モジュールは、DOORS 側で行われたすべての変更を反映して完全に更新されます。完全更新が終了すると、Tau のモジュールは DOORS バージョンと完全に同期されます。

完全更新を使用するタイミング

完全更新は、DOORS で以下のいずれかの変更が行われた場合に必要です。

- ビューの定義が変更された
  - たとえば、属性の変更または追加によって
- フィルタが変更された、または適用された
- Tau のフォーマル モジュールを更新せずに、オブジェクトが削除または消去された

ビューまたはベースラインの変更

インポートしたフォーマル モジュールのビューまたはベースラインを変更するには、以下の手順を行います。

- ［モデル ビュー］でフォーマル モジュールを選択。
- 右クリックして、［ビュー / ベースラインの変更］を選択。
- ダイアログで、使用したいビューとベースラインを選択し、［OK］をクリック。

ビューまたはベースラインを変更すると、モジュールの完全更新が実行されます。

リンクの作成

「リンクの管理」の説明にあるとおり、リンクの作成にはいくつかの方法があります。DOORS インテグレーションはリンク作成のための別の機構、つまり、DOORS 内の要求と Tau 内の UML 要素の間でのドラッグアンドドロップを提供します。

DOORS 要求から UML 要素にリンクを作成するには、DOORS から要求をドラッグして、Tau の［モデルビュー］内の要素にドロップします。DOORS からのドロップはターゲットのモデル要素が DOORS 表現を持っている場合に動作します。つまり、その要素は DOORS からインポートしたものであるか、または DOORS にエクスポートしたものです。リンクは DOORS 内にのみ自動で作成されます。Tau でリンクを表示するには、その UML モデルを DOORS から更新する必要があります。

UML 要素から DOORS 要求にリンクを作成するには、［モデルビュー］から UML 要素を DOORS にドラッグして、要求にドロップします。結果は、UML モデルに <<trace>> 依存関係が作成されます。この場合は、UML 要素が DOORS 表現を持っている必要はありません。UML モデルの変更を手動で DOORS にコミットするまでは、DOORS 内にはリンクは作成されません。

DOORS への要求のエクスポート

DOORS フォーマルモジュールを作成する時に使う最も一般的なシナリオは、DOORS でフォーマルモジュールを作成して、その後 Tau にインポートするというものです。ただし、ある状況においては、フォーマルモジュールの定義を Tau で始めることが有効な場合もあります。このセクションではこの機構について説明します。要求を含むパッケージ（TTDDoors::formalModule ステレオタイプ適用）は、DOORS にフォーマルモジュールとしてエクスポートできます。パッケージのエクスポートの手順は以下のとおりです。

1. ［モデルビュー］で DOORS フォーマルモジュールのルートとしたい項目を選択する。
2. その項目を右クリックして、ショートカットメニューから［DOORS にエクスポート］を選択する。
3. DOORS がまだ起動していない場合は、ここで起動される。DOORS にログインした後、エクスポートの操作は継続する。
4. ［モジュールへのエクスポート］ダイアログで代理モジュールのパラメータを設定する。
5. DOORS データベース内の［場所］を選択する。
6. オプションで［オブジェクト識別子］データ、つまり開始番号や接頭辞またはデフォルト値などを指定する。
7. ［エクスポート形態］で［要求パッケージ］を選択する。
8. ［OK］を押す。

新しいフォーマルモジュールが DOORS に作成されます。名前は、エクスポートされたパッケージの名前です。この名前は変更できません。

パッケージ内のすべての要求は DOORS オブジェクトとしてエクスポートされます。他のすべての要素は無視されます。また、エクスポートされた要求の名前は無視され、エクスポート後に、対応する DOORS オブジェクトのオブジェクト識別子（object identifier）に置き換えられます。

注記 **1**

パッケージをエクスポートできるのは一度だけです。フォーマルモジュールの Id プロパティが設定されていると、エクスポートはオフになります。

Supported changes

モジュールがエクスポートされたら、変更について、インポートされたモジュールと同じ制約が適用されます。完全なリストは「サポートされる変更」を参照してください。

要求属性の追加

要求属性は、Tau では手動で指定したり変更したりできません。したがってそれらはエクスポートの対象ではありません。要求属性や値をエクスポートしたモジュールに追加するには、以下の手順を実行します。

- DOORS で属性や値を指定して、ビューを保存する。
- Tau で新たなビューに切り替える。手順は、「ビューまたはベースラインの変更」を参照。

この操作で、モジュールは正しい属性定義と値で更新されます。これで、属性が Tau で使用できます。

参照

トレーサビリティの管理

DOORS ツールバー

Tau の DOORS ツールバー ボタンを使って、今まで説明してきた操作の一部を実行できます。

- **DOORS の開始** : DOORS がまだ起動されていなければ、DOORS を起動します。ログイン情報を入力すると、Tau が DOORS に接続されます。
- **DOORS フォーマル モジュールのインポート** : DOORS インポート フォーマル モジュール ウィザードを起動します。
- **DOORS 内の要素を検索** : DOORS を起動し、特定のオブジェクトを DOORS で検索します。
- **DOORS にエクスポート** : 選択した要素（およびその要素に含まれる要素）を DOORS にエクスポートします。
- **DOORS のデータで更新 :** 現在選択している要素を DOORS での変更で更新します。
- **変更を DOORS にコミット** : 選択している要素へのすべての変更を DOORS にコミットします。

旧バージョンの **Tau** からの移行

インポートされた要求をバージョンの異なる Tau に移行する場合、Tau と DOORS の情報を保持するためには以下に記載されている指示に従う必要があります。

重要！

指示に従わないと、Tau と DOORS の情報が失われることがあります。インテグレーションでは両方のツールのデータが自動的に保存されてしまうので、失われたデータは、前回保存された状態またはベースライン バージョンに復帰する以外、簡単に復活させる方法はありません。

UML の要求

この説明は、エクスポートした UML モデルとともに、要求の UML 表記を使用している場合を対象としています（DOORS で UML 要素を使うを参照）。

旧バージョンの Tau で以下のことを行います。

- エクスポートされたすべての UML モデル（代理モジュール）に対して、［DOORS にコミット］を実行する。

新バージョンの Tau で以下のことを行います。

- すべてのフォーマル モジュール パッケージに対して、［DOORS から全体を更新］を実行する。
- エクスポートされたすべての UML モデルに対して、［DOORS から更新］を実行する。
- ［ファイル］メニューから［すべて保存］を選択する。

.dim ファイル内の要求

この説明は、エクスポートした UML モデルとともに、要求の古い表記（.dim ファイル）を使用している場合を対象としています（DOORS で UML 要素を使うを参照）。

旧バージョンの Tau で以下のことを行います。

- エクスポートされたすべての UML モデル（代理モジュール）に対して、［DOORS にコミット］を実行する。
- インポートされたすべてのフォーマル モジュールに対して、［リンクを DOORS にコミット］を実行する。

新バージョンの Tau で以下のことを行います。

- Tau プロジェクトから .dim ファイルを削除する（任意でファイル システムからファイルを削除する）。
- 新しいインテグレーションを使用してフォーマル モジュールを再度インポートする（要求のインポートを参照）。
- すべての代理モジュールに対して、［DOORS から更新］を実行する。
- ［ファイル］メニューから［すべて保存］を選択する。

注記

.dim ファイルと UML モデルなど、異なる表記を混同することは、サポートされません。混同により、データが失われることがあります。

Tau3.1.1 またはそれ以前を使ってエクスポートされた代理モジュール

このセクションの説明は、Tau3.1.1 以前を使ってエクスポートした代理モジュールを使用するケースに適用されます。Tau 4.0 では、モジュールのエクスポートスキームを単純化し、従来の［DOORS］タブを使って中間的表現を用いる方法を取り止めました。

この結果、過去にエクスポートしたモジュールは、すべて再エクスポートする必要があります。

Tau の旧バージョンでは：

- ［リンクを DOORS から更新 **...**］を、DOORS にエクスポートされたすべての UML モデル（代理モジュール）について実行して、すべてのリンクが UML モデルで使用できるようにする。

DOORS では：

- 代理モジュールを削除または名前変更する。

Tau の新バージョンでは：

- Tau プロジェクトから .dim ファイルを取り除く。
- Tau の UML モデルを、「DOORS で UML 要素を使う」を参照して、新しいインテグレーションで再エクスポートする。
- ［DOORS から更新］をすべてのエクスポートしたパッケージついて実行する。
- ［ファイル］>［すべてを保存］を選択する。

UML モデルのテスト

この章では Tau のテスト プロファイルの使用方法について説明します。

48

UML テスト プロファイル

このセクションでは UML テスト プロファイルを使用した UML モデルのテスト方法について説明します。

UML テスト プロファイルは、テスト仕様の概念を付加した、UML の拡張機能です。

テスト プロファイルによって、多様なシステムのテストを行うことができますが、その主な目的はアクティブ クラスのブラックボックス テストです。主な機能には、テストの指定、実行、ログ記録があります。

必要な作業を理解するためには、以下のセクションをご覧ください。

- テスト プロファイル サポートの有効化
- テスト モデルの作成
- テスト アプリケーションのビルドと実行

テスト プロファイル サポートの有効化

UML テスト プロファイル サポートはアドインとして実装されますが、使用するプロジェクトごとに手動で有効にする必要があります。アドインを有効にするには、以下の手順を行います。

1. ［ツール］メニューから［カスタマイズ］ダイアログを選択します。
2. ［アドイン］タブをクリックして、［TestingProfile］アドインにチェックを付けて選択します。
3. ［OK］をクリックします。

UML テスト プロファイル

UML テスト プロファイルは、OMG によって標準化された UML プロファイルの 1 つであり、UML 2 にテスト モデリング機能を加えた拡張です。テスト ケース、テスト コンポーネント、判定などのテストの概念を用いて UML を拡張しています。

次のセクションでは、テスト プロファイルの最も重要な概念について簡単に説明します。テスト プロファイルの詳細については、プロファイル仕様（OMG からダウンロードできます）をお読みください。

このプロファイルは、ライブラリ「`TTDTestingProfile`」として実装されます。このライブラリはテスト プロファイル サポートを有効にするとロードされます。プロファイル仕様からの定義はすべてこのライブラリに含まれています。

定義

アービター（**Arbiter**）

テスト アービトレーションを実行します。つまり、テスト ケースの判定を維持保存します。あるテスト ケースの現在の判定を保存して、テスト コンポーネントによる判定の設定と取得を可能にします。各テスト コンテキストにアービターが 1 つあります。

アービター インターフェイスは以下の 2 つの操作を持っています。

```
getVerdict() :Verdict
```

現在のテスト ケースの判定を返します。

```
setVerdict(v :Verdict)
```

現在のテスト ケースの判定を設定します。

スケジューラ

テスト コンテキスト内でテストの実行を制御します。テスト コンテキスト内でテスト コンポーネントを管理し、テスト ケースを実行します。各テスト コンテキストにスケジューラが 1 つあります。

テスト ケース

テストの振る舞いを指定する、ステレオタイプ化された操作です。SUT に送信する一連の「刺激」と、その予測応答を指定します。実応答と予測応答間の一致に基づいて、判定がくだされます。

テスト コンテキスト

テスト ケースのグループ化機構として機能するクラスです。通常、あるクラスのすべてのテストは 1 つの同じテスト コンテキストに含まれます。テスト コンテキストには、一連のテスト ケース、テスト ケース内の複数のテスト コンポーネント、アービター（Arbiter） 1 つとスケジューラ 1 つがあります。

また、テスト構成（テスト コンポーネント、アービター、スケジューラによる初期構成）は、テスト コンテキストで指定します。

テスト コンポーネント

テスト ケースの振る舞いにおいて実行されるクラスです。テスト ケースの振る舞いに従い SUT と通信して、アービター（Arbiter）に判定を報告します。

テストの目的

テスト ケースの目的は、テストの対象が何であるかを非形式的な形で記述することです。テストの目的は、テスト コンテキストまたはテスト ケースからテスト対象の要素への依存関係を用いて指定します。

SUT

SUT（System Under Test）は、テスト対象のシステムまたはクラスのインスタンスです。各テスト コンテキストに SUT が 1 つ含まれます。

判定

判定とはテスト ケースの結果です。以下の表に示す 4 つの値があります。

判定値	意味
pass	テストの実行が成功して、SUT がテスト ケースで指定されたとおりに応答した。
inconclusive	テストを実行した結果からは、SUT が正しく振る舞ったかどうかは判定できなかった。
fail	SUT がテスト ケースで指定されたとおりに動作しなかったため、テストの実行に失敗した。
error	テスト システムにエラー（テスト ケースが無限ループに陥った場合など）がある。

テスト モデルの作成

ある特定のクラスをテストするためのテスト モデルを作成するには、以下の手順を行います。

1. SUT としての対象クラスを含むテスト コンテキストを作成します。［テスト コンテキストの作成］ダイアログを使用すると最も簡単ですが、テスト コンテキストを手動で作成することもできます。
2. テスト コンテキストでテスト ケースを作成します。テストケースの作成方法にはさまざまなものがあります。テスト ケースの作成を参照してください。また、［テスト コンテキストの作成］ダイアログで、空のテスト ケースを自動作成することもできます。
3. 新しいテスト ケースに対してテスト ケースの振る舞いの指定を行います。テスト コンテキスト内のテスト ケースにはシーケンス図を使用する必要があります。テスト コンポーネントのテストケースを指定するには、状態機械図を使用できます。
4. 手順 2 と 3 を繰り返して、テストを定義するために必要な数のテスト ケースを追加します。

それぞれ異なるテスト ケースを持つ複数のテスト コンテキストを作成できます。

テスト モデルを完成したら、テスト アプリケーションを作成して実行できます。

参照

テスト アプリケーションのビルドと実行

テスト コンテキストの作成

テスト コンテキストを作成するには、アクティブ クラスを選択してから右クリックして、[テスト コンテキストの作成 ...] を選択します。

[テスト コンテキストの作成] ダイアログが開き、デフォルトで SUT のタイプとしてアクティブ クラスが初期設定されます。

[テスト コンテキストの作成] ダイアログ

このダイアログは、テスト コンテキストの作成に使用します。ダイアログに入力する値の一覧表を以下に示します。

値	説明
オーナー	テスト コンテキストの所有者
名前	テスト コンテキストの名前
SUT：名前	テスト コンテキストの SUT パートの名前
SUT ：タイプ	テスト コンテキストの SUT パートのタイプ
テスト コンポーネント：名前	テスト コンテキストのテスト コンポーネント パートの名前
テスト コンポーネント：タイプ名	テスト コンテキストのテスト コンポーネント パートのタイプ名
テスト構成の生成	テスト構成図を生成するかどうかを指定します。
テスト ケースの生成	空のテスト ケースを生成するかどうかを指定します。
ビルド アーティファクトの作成	テスト コンテキストをマニフェストするビルド アーティファクトを作成するかどうかを指定します。
新規ファイルに保存	テスト コンテキストを新規ファイルに保存するかどうかを指定します。

すべての値を入力して［OK］をクリックすると、ダイアログの情報に従ってテスト コンテキストが作成されます。

テスト コンテキスト「Name」が「Owner」の中に作成され、「Owner」と同じファイルに保存されます。所有者として「Model」を指定すると、このテスト コンテキストはモデルの最上位レベルに作成されます。

テストの目的の依存関係が、テスト コンテキストから SUT タイプに付加されます。

テスト コンテキストには、さまざまな要素が作成されます。各要素の詳細を以下に説明します。

- SUT パート
- テスト コンポーネント パート
- いくつかのコネクタ

- いくつかの依存関係
- テスト構成図
- 初期テスト ケース

［ビルド アーティファクトの作成］を選択すると、テスト コンテキストをマニフェストするビルド アーティファクトが自動的に作成されます。

［新規ファイルに保存］を選択すると、テスト コンテキスト（および作成された場合はビルド アーティファクト）がファイルの保存ダイアログで指定した新規ファイルに保存されます。

SUT パート

SUT パートは、合成属性であり、ダイアログで指定した名前と型をもつ <<SUT>> ステレオタイプを保持しています。

テスト コンポーネント パート

テスト コンポーネント パートの名前はダイアログで指定されます。新しいアクティブ クラスがテスト コンテキスト内に作成され、テスト コンポーネント タイプとして設定されます。クラスには、<<testComponent>> ステレオタイプが適用されます。

SUT タイプの各ポートに対応してテスト コンポーネント クラスのポートが 1 つあります。ポートの実現化インターフェイスと要求インターフェイスが互いに交換されて、SUT とテスト コンポーネント間の通信が可能になります。つまり、SUT ポートの実現化インターフェイスがテスト コンポーネント ポートの要求インターフェイスになったり、テスト コンポーネント ポートの実現化インターフェイスが SUT ポートの要求インターフェイスになります。

コネクタ

SUT パートとテスト コンポーネント パートのポートの組ごとに、1 つのコネクタが生成されてポートに接続されます。

依存関係

SUT タイプで可視であるすべての定義がテスト コンテキストにおいても可視であるようにするために、テスト コンテキストから必要な定義すべてに対して、インポート依存関係とアクセス依存関係 (またはその一方) が自動作成されます。

テスト構成図

オプションで、テスト コンテキスト内にテスト構成を記述する合成構造図を作成できます。このダイアグラムには、SUT 用とテスト コンポーネント用の 2 つのパート シンボルがあります。SUT とテスト コンポーネントの各ポートが表示され、これらのポートがコネクタに接続されます。

初期テスト ケース

オプションで、テスト コンテキストに初期テスト ケースを作成できます。テスト ケースは <<testCase>> ステレオタイプが適用された操作です。このテスト ケースには、2 本のライフライン（SUT 用とテスト コンポーネント パート用）をもつシーケンス図が 1 つ含まれます。

テスト ケースの作成

テスト ケースは <<testCase>> ステレオタイプが適用された操作です。テストケースを所有できるのは、テスト コンテキストまたはテスト コンポーネントのみです。

新しい空のテスト ケースの作成方法は、以下のセクションで説明しています。

- テスト コンテキストへの空テスト ケースの追加
- テスト コンポーネントへの空テスト ケースの追加

既存のシーケンス図を利用してテストケースを作成する方法は、既存ダイアグラムからのテスト ケースの作成で説明しています。

テスト コンテキストへの空テスト ケースの追加

テスト コンテキストにテスト ケースを追加するには、［モデル ビュー］でそのテスト コンテキストを右クリックして、［テスト ケースの作成］を選択します。

新しいテスト ケースがテスト コンテキストに追加されます。テスト ケースには、テスト コンテキストの各パートごとにライフラインを 1 本もつシーケンス図が 1 つ含まれます。

ライフライン間にメッセージを追加するか、テスト コンテキスト内で他のテスト ケースに対する参照を追加して、このテスト ケースの振る舞いを手動で指定できます。詳細については、1495 ページの「シーケンス図」を参照してください。

テスト コンポーネントへの空テスト ケースの追加

テスト コンポーネントにテスト ケースを追加するには、［モデル ビュー］でそのテスト コンポーネントを右クリックして、［テスト ケースの作成］を選択します。

新しいテスト ケースがテスト コンポーネントに追加されます。テスト ケースには空の状態機械図があります。

これで状態機械図によってテストケースの振る舞いを指定できます。

既存ダイアグラムからのテスト ケースの作成

モデル ベリファイヤ（Model Verifier）からトレース図を作成する場合のように、既存のシーケンス図からテスト ケースを作成するには、［モデル ビュー］でそのダイアグラムを右クリックして、［テスト コンテキストへの追加］を選択します。

サブメニューが開き、モデル内のすべてのテスト コンテキストが一覧表示されます。新しいテスト ケースの作成先テスト コンテキストをクリックすると、［テスト コンテキストへの追加］ダイアログが表示されます。ダイアログで［OK］をクリックすると、テスト ケースが作成されます。

テスト ケースのシーケンス図でサポートされる要素はメッセージ、タイミング制約、および参照シンボルなので、これらの要素のみがテスト ケース図にコピーされます。

テスト コンテキストへの追加

［テスト コンテキストへの追加］ダイアログには、4 つのリスト ボックスがあります。ダイアログには、リスト ボックス間でライフラインを移動するためのボタンがあります。このダイアログの目的は、ライフラインを各カテゴリーに分類して、テスト ケースにおける解釈の方法を決定することです。

ダイアログが開くと、左側の［ライフライン］リスト ボックスに、選択したダイアグラムのライフラインがすべて表示されます（1 つのライフラインが SUT ライフラインとしてあらかじめ右上のリストボックスに選択されている場合があります）。ダイアログを閉じる時点でこの［ライフライン］リスト ボックスに残っているライフラインは、新しいテスト ケースにコピーされません。

右側の［**SUT**］リスト ボックスには、新しいテスト ケースで SUT を表現するライフラインが表示されます。元のダイアグラムのライフラインで、選択されたテスト コンテキスト内の SUT と同じタイプのものがある場合、このライフラインは［ライフライン］リスト ボックスではなく、［SUT］リスト ボックスに表示されます。SUT ライフラインは 1 つだけ指定できます。

右側の［テスト コンポーネント］リスト ボックスには、このテスト コンテキストにおけるテスト コンポーネントを表現するライフラインが表示されます。新しいダイアグラムを作成すると、このリスト ボックスのライフラインは、テスト コンポーネントを表現する **1 本のライフラインにマージ**されます。

右側の［テスト ケース属性］リスト ボックスには、テスト ケースにおいて属性を表現するライフラインが表示されます。このライフラインは、テスト ケースのダイアグラムにコピーされます。属性ライフラインの型ごとに、同じ型の属性がテスト ケース下に作成されます。すでに属性がある場合は、ライフラインはその属性に関係付けられます。

［テスト コンポーネント］リスト内のライフライン間のメッセージと［テスト ケース属性］リスト内のライフライン間のメッセージは、新しく作成されるテスト用のシーケンス図にコピーされません。

ライフラインを目的のリスト ボックスに移動したら、［OK］をクリックしてダイアログを閉じます。新しいテスト ケースが、選択したテスト コンテキストに作成されます。前述のように、ライフラインの分類に従って、新しいテスト ケースには元のダイアグラムに基づくシーケンス図が含まれます。

ダイアログを閉じると、新しいシーケンス図が自動作成されます。メッセージと参照の順序は元のダイアグラムに従います。作成後、表示して編集できます。

テスト ケースの振る舞いの指定

テスト ケースの振る舞いは、シーケンス図と状態機械図の 2 タイプのダイアグラムで指定できます。シーケンス図はテスト コンテキストが所有するテスト ケースに使用し、状態機械図はテスト コンポーネントが所有するテスト ケースに使用します。テスト ケースは、1 つのタイプのダイアグラムでのみ指定できます。つまり、テスト コンテキスト内でテスト ケースをシーケンス図で指定した場合、テスト コンポーネント内で同じテストケースを状態機械図で指定することはできません。

テスト ケースでは、「刺激」の送信時に SUT からの**予測応答**を指定します。したがって、SUT がテスト ケースの指定どおりに動作する場合、テスト ケースは必ず `pass`（合格）すると考えられます。判定の合格は暗黙的であり、テスト ケースで指定する必要はありません。`pass`（合格）以外の判定を必要とする場合のみ、テスト ケースで判定を設定する必要があります。

一般的にはシーケンス図を使用してテスト ケースを指定します。シーケンス図は多様なテストで使用されます。状態機械図は、シーケンス図では表現力が不足するような下位レベルのテスト ケースに使用します。状態機械図を使用するのは、シーケンス図では目的の振る舞いを表現できない場合に限る方がよいでしょう。

テスト コンテキスト内でシーケンス図で指定されたテスト ケースは、中間テスト モデルの作成時にテスト コンポーネント内の状態機械に変換されます。一方、テスト コンポーネント内で状態機械図で指定されたテスト ケースは、変換されずそのまま残ります。

シーケンス図

シーケンス図で指定したテスト ケースには、最低 2 本のライフライン（SUT 用に 1 本とテスト コンポーネント用に 1 本）があります。テスト ケースの振る舞いは、これらのライフライン間の相互作用として指定します。

テスト ケースでは、以下のシンボルを使用できます。

- ライフライン
- メッセージ ライン
- アクション シンボル（テスト コンポーネントのライフラインでのみ使用可能）
- 参照シンボル
- タイマー仕様ライン

単純のテスト ケースの作成方法を、1495 ページの例 580 に示します。

例 **580:** 単純なテスト ケース ----------

このテスト コンポーネントは、整数パラメータ値「3」を持つ `Ping` シグナルを SUT に送信します。SUT は応答として、同じパラメータ値を持つ `Pong` シグナルをテスト コンポーネントに送信します。

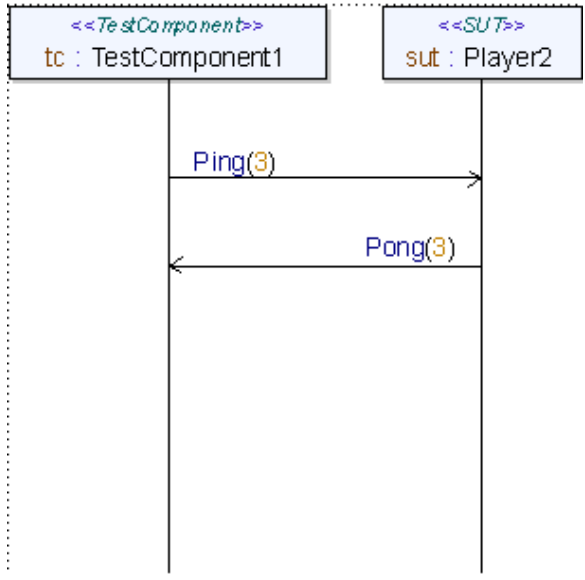


図 261: 単純なテスト ケースの例

パラメータ値「3」を持つ Pong シグナルは、SUT からの予測応答です。これは、テスト ケースを合格にできる、唯一の応答です。これ以外のシグナルを受信した場合、つまりパラメータが「3」以外だった場合、テスト ケースは不合格となります。

この場合、判定を明示的に設定する必要はありません。予測応答を受信した場合テストケースは合格し、それ以外の場合は不合格となります。

アクション シンボルの使用

アクション シンボルを使用して、テスト ケースの下位レベルのアクションを実行できます。この方法は、値をテストする場合、および pass（合格）以外の判定を設定する場合に必要となります。

- 判定を明示的に設定するには、`setVerdict` 操作を呼び出して目的の判定値を指定します。
- パラメータまたは変数値など、条件をテストする場合、通常 `assertTrue` 操作を使用します。

アクション シンボルは、テスト コンポーネントのライフラインにのみ使用できます。テスト ケースにおける SUT 内部の振る舞いに対しては何ら仮説を立てられないためです。

例 **581:** 条件のテスト

1497 ページの図 262 に、ITTDArbiter の `assertTrue` 操作を使用して、SUT からテスト コンポーネントに送信されたパラメータの値をテストする例を示します。

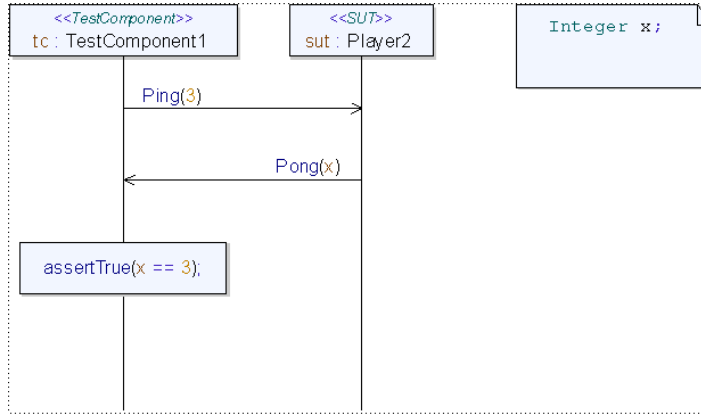


図 262: `assertTrue` 操作を使用して条件をテストする例

パラメータ値が「3」であれば、判定は `pass`（合格）と設定され、それ以外の値の場合は `fail`（不合格）となります。

他のテスト ケースまたは操作の参照

あるテスト ケースまたはテスト ケースのある操作を参照するには、シーケンス図で参照シンボルを使用し、参照先のテスト ケースの名前を入力します。

別のテスト ケースとして実行したい場合はテスト ケースを使用し、他のテスト ケースの一部として実行したい場合は操作を使用します。

注記

参照先のテスト ケースまたは操作は、同じテスト コンテキストによって所有され、同じライフラインを持っている必要があります。

タイミング制約の指定

タイミング制約は、メッセージの送受信間隔の予測時間に関する制約を表現するのに使用します。

タイミング制約は、テスト コンポーネントのライフラインにのみ使用できます。テスト ケースにおける SUT の内部の振る舞いに対しては何ら仮説を立てられないためです。

タイミング制約を指定するには、タイマー仕様ラインを使用し、両端をテストコンポーネントに接続し、指定する制約をテキストラベルに入力します。以下の 3 種類のタイミング制約を指定できます。

- 指定した時間値以下：< x、<= x

  メッセージの送信と受信との間隔が x 秒より短い場合テストは合格し、それ以外の場合は不合格となります。
- 指定した時間値以上：> x、>= x

  メッセージの送信と受信との間隔が x 秒より長い場合テストは合格し、それ以外の場合は不合格となります。
- 指定した時間値の範囲内：`{x..y}`

  メッセージの送信と受信との間隔が指定した時間の範囲内の場合テストは合格し、それ以外の場合は不合格となります。

`{== x}` または `{x},` など、固定の絶対値を用いた制約は、意味がないのでサポートされません。

例 **582:** タイミング制約 ----------

以下の例は、テスト ケースでのタイミング制約の使用法を示します。テスト コンポーネントは `Ping` シグナルを SUT に送信し、3 秒以内に `Pong` シグナルが返されると予測します。

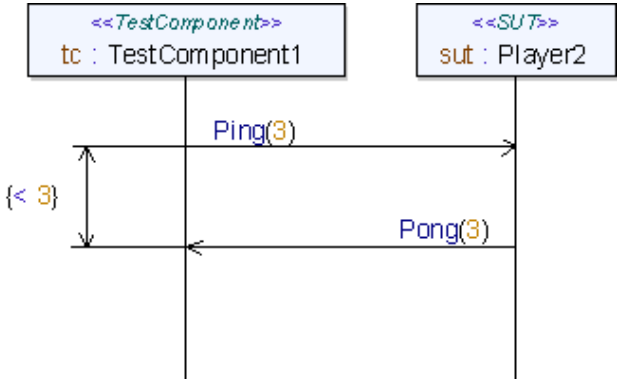


図 263: タイミング制約の例

3 秒以上経過してから `Pong` シグナルを受信した場合、テストは不合格となります。また、Pong 以外のシグナルを受信した場合やパラメータが「3」以外だった場合は、返信を受信したタイミングには関係なく、テスト ケースは不合格となります。

その他のライフライン

SUT 用のライフラインとテスト コンポーネント用のライフラインのほか、**テスト ケースの属性**を表現するライフラインも使用できます。これは、属性がアクティブ クラスのインスタンスに対する参照である場合のみ有用です。たとえば、SUT がインスタンスをシグナル パラメータとして渡し、テスト コンポーネントが SUT に加えてそのインスタンスとも直接やりとりする必要がある場合などです。

注記

シーケンス図で現在使用できるのは、3 種類のライフラインのみです。SUT を表現する 1 本のライフライン、テスト コンポーネントを表現する 1 本のライフライン、テスト ケース自体の属性を表現する任意数のライフラインです。

状態機械図

状態機械図を使用して指定されたテスト ケースには、要件あるいは制約はありません。明示的に判定が異なる値に設定されていない限り、テスト ケースは合格します。

明示的に判定を設定するには、アクション ボックスを使用して `setVerdict` 操作を呼び出します。条件をテストするには、`assertTrue` 操作を呼び出します。

例 **583:** 状態機械で指定されたテスト ケース**----------**

以下の例は、状態機械を使用して 1496 ページの図 261 のテスト ケースを指定する方法を示しています。この場合のテスト ケースの所有者は、テスト コンテキストではなくテスト コンポーネントです。

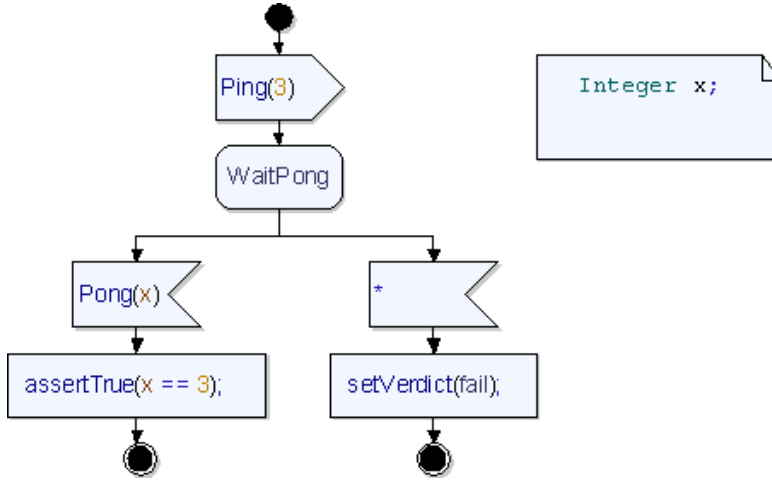


図 264: 状態機械で指定されたテスト ケース

テスト フレームワーク

テスト プロファイルの実装は、あるテスト フレームワークを使用します。このフレームワークは、テスト プロファイルで指定されたセマンティックを実装し、テストの実行とログ記録を制御します。このフレームワークは `TTDTestFramework` というライブラリで定義されます。

`TTDTestFramework` には、テスト プロファイルとその他の要素のインターフェイスの実装があります。ここでは、フレームワークの主な概念について説明します。

インターフェイス

ITTDArbiter

これは、アービター（Arbiter）インターフェイスを特化するインターフェイスです。以下の 2 つの新しい操作が追加されています。

```
assertTrue(b:Boolean)
```

これは、条件をテストし、それに従って判定を設定するための操作です。ブール式 `b` が true のとき判定は `pass`（合格）に設定され、`b` が false の場合は `fail`（不合格）に設定されます。

```
getFinalVerdict() :Verdict
```

これは、テスト ケースが終了した時点で、最終判定を返します。テスト ケース終了時にスケジューラによって呼び出されます。

ITTDScheduler

これは、スケジューラインターフェイスとよく似たインターフェイスですが、フレームワークに合わせて調整されています。以下の 2 つのシグナルを定義します。

```
finishTestCase(ITTDTestComponent)
```

テスト コンポーネントが、テスト ケースの実行を終了したことをスケジューラに通知するために使用します。

```
reportNewTestComponent(ITTDTestComponent)
```

テスト コンポーネントが、その存在をスケジューラに通知するために使用します。

ITTDTestComponent

以下のように、1 つのシグナルのみを持つテスト コンポーネント インターフェイスです。

```
startTestCase(Charstring)
```

テスト コンポーネントにテスト ケースを開始するよう指示します。これは、スケジューラが使用します。

実装

TTDArbiter

これは、ITTDArbiter インターフェイスの実装です。テスト ケースの総合判定を属性に保存します。

TTDScheduler

これは、ITTDScheduler インターフェイスの実装です。テスト ケースを取り込んで実行し、対象テスト コンポーネントを追跡して、テストを実行します。

テスト アプリケーションのビルドと実行

このセクションでは、テスト コンテキストからテスト アプリケーションをビルドして実行する方法について説明します。

テスト アプリケーションのビルド

テスト コンテキストからテスト アプリケーションをビルドするには、以下の手順を行います。

テスト コンテキストからテスト アプリケーションをビルドするには、テスト コンテキストをマニフェストするビルド アーティファクトが必要です。通常、ビルド アーティファクトは［テスト コンテキストの作成］ダイアログで自動的に作成されますが、これを手動で作成することもできます。

1. テスト コンテキストを右クリックして［Test Generation］-> ［新しいアーティファクト］を選択します。

   ビルド アーティファクトが作成され、テスト生成ステレオタイプが適用されます。
2. 必要に応じて、テスト生成ステレオタイプのデフォルト値を変更します。
3. コード ジェネレータのステレオタイプを適用して、ビルドするテスト アプリケーションのタイプを指定します。現在対応しているのは、モデル ベリファイヤ（Model Verifier）のみです。

ビルド アーティファクトをビルドするには、［ビルド］コマンド（ビルド ショートカット メニュー）を使用します。

ビルドに成功すると、結果として作成されたテスト アプリケーションをテスト ドライバに挿入して実行できます。

テスト アプリケーションの実行方法については、1507 ページの「テスト アプリケーションの実行」を参照してください。ビルドのステップはスキップされることが多いです。これは、テスト アプリケーションを実行すればビルドは自動的に実行されるからです。

テスト アプリケーションのビルド プロセス

ビルドプロセスの中で、元となるテスト モデルは中間テスト モデルに変換され、この中間モデルからコードが生成されてテスト アプリケーションがビルドされます。1503 ページの図 265 にこのプロセスの概要を図示します。

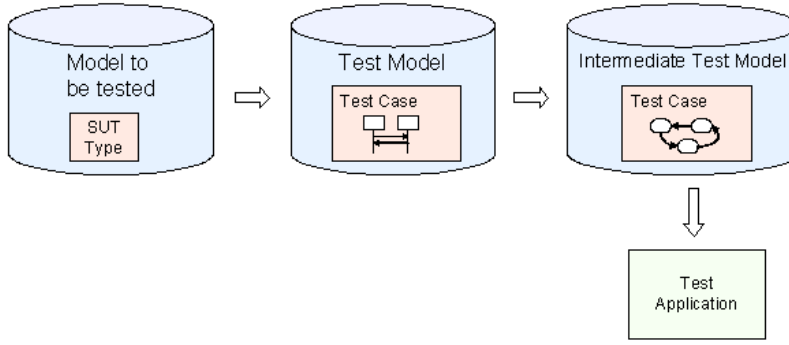


図 265: ビルド プロセスの詳細

中間モデルは、ビルド時にプロジェクトに挿入されます。ビルドごとに新しい中間モデルが生成され、古いモデルは破棄されますが、中間モデルを保存して個別にビルドすることもできます。通常、中間モデルは変更できません。どうしても変更が必要な場合は、その変更がテスト フレームワークに適合しており、モデルがビルドできることを確認する必要があります。

中間テスト モデル

テスト モデルは、ビルド プロセスで中間テスト モデルに変換されます。複数回の変換を経た後にテスト モデルから実行テスト モデルになります。

たとえば、「シーケンス図」で説明したように、シーケンス図で指定されている各テスト ケースの振る舞いは、選択したコード ジェネレータでコードを生成できる形式に変換する必要があります。

また、この変換によって、テスト モデルがテスト フレームワークと正しく相互作用することを確実にします。

変換

以下の表に、テスト モデルから中間モデルへの解釈中に行われるすべての変換を示します。

テスト モデル	中間モデル
テスト ケース	同じ名前のテストコンポーネント内の操作
テスト ケースの振る舞い (シーケンス図)	テスト コンテキスト内の対応する操作の状態機械 (ダイアグラムなし)
テスト ケースの属性	対応する状態機械内の同一属性
テスト コンポーネントから送信されたメッセージ	同じシグナルとパラメータを持つシグナル送信アクション
テスト コンポーネントが受信したメッセージ	1504 ページの「メッセージの受信」を参照。
アクション シンボル (`CompoundActionOccurrence`)	同じ内容の複合アクション
参照シンボル (`InteractionOccurrence`)	参照先のシグニチャの呼び出し。具体的には、参照先のシグニチャとパラメータを含む呼び出し式を持つ式アクション。
タイマー仕様ライン	1505 ページの「タイマー仕様ライン」を参照。

メッセージの受信

テスト コンポーネントが受信したメッセージは、テスト ケースに対応する状態機械において、状態 (ステート) とトリガされた遷移に変換されます。

まず状態機械は、特定のシグナルを受信するまで、ある状態で待機します。他のシグナルを受信してもエラーとして処理されます (操作は判定を不合格と設定し、リターンします)。指定されたシグナルを受信すると、シグナル パラメータが処理され、次のアクションが処理されます。シグナル パラメータは、シグナル パラメータの式に従って、2 つの方法のいずれかで処理されます。シグナル パラメータが属性の場合、シグナル パラメータの値が属性に割り当てられます。シグナル パラメータがそれ以外 (一般式) の場合、受信したシグナル パラメータ値は、この値 (属性値) との照合でテストされます。2 つの値が異なれば、エラーとして処理されます (判定は不合格と設定され操作はリターンします)。

例 **584:** ----------

メッセージの受信 :

```
sig(a, 1);
```

シーケンス図では以下のように変換されます (u2p 構文) :

```
      ..... /* previous action */
      nextstate Wait;
  }
state Wait;
  input sig(a, tmp) {
```

```
    if (tmp != 1) {
      arbiter.setVerdict(fail);
      return;
    }
    ..... /* action following the signal reception */
  }
  input * {
    arbiter.setVerdict(fail);
    return;
  }
```

タイマー仕様ライン

タイマー仕様ラインは、指定時間値にしたがっていくつかの異なる方法で変換されます。すべてのマッピングにおいて、ある 1 つのタイマーを共有して、タイミング制約を実装します。

< 時間値、または <= 時間値

「< 指定時間値」、または「<= 指定時間値」が指定されたタイマー仕様は、タイマー定義と、タイマー仕様の開始位置のタイマー セット アクションに変換されます。

指定時間の終点に、タイマーのリセット アクションが挿入されます。これは、その点でタイマーがまだアクティブであるためです。また、どのような状態においてもタイマー シグナルが受信されるとエラーになります。これはタイマーが切れたことを意味するからです。したがって、すべての状態（state *）に対して、タイマー シグナルを受信する場合を考慮し、これにより判定を不合格に設定する遷移と操作からのリターンへと進むようにします。

例 **585:** 指定値に満たないタイマー仕様 ----------

タイマー宣言：

```
timer t = 3;
```

タイマー仕様の開始：

```
set(t);
```

タイマー仕様の終了：

```
reset(t);
```

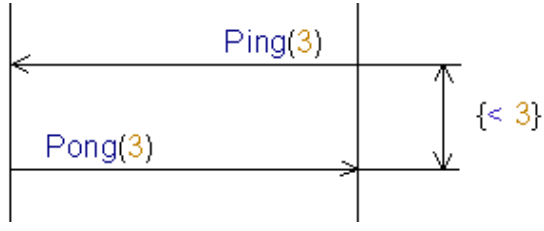


図 266: 「< 指定時間値」または「<= 指定時間値」が設定されたタイマー仕様

すべての状態でのエラー :

```
state *;
  input t {
    arbiter.setVerdict(fail);
    return;
  }
```

> 時間値、または、>= 時間値

「> 指定時間値」または「>= 指定時間値」が指定されたタイマー仕様は、タイマー定義と、タイマー仕様の開始位置のタイマー セット アクションに変換されます。

指定時間の終点で、タイマーがまだアクティブがどうかのテストが実行されます。アクティブな場合、結果はエラーとして処理されます（判定は不合格と設定されて、操作はリターンします)。どの状態でタイマー シグナルを受信しても、OK であり、同じ状態に再入します。

例 **586:** 指定値を超えるタイマー仕様 **----------**

タイマー宣言 :

```
timer t = time-value;
```

タイマー仕様の開始 :

```
set(t);
```

タイマー仕様の終了 :

```
if (active(t)) {
  arbiter.setVerdict(fail);
  return;
}
```

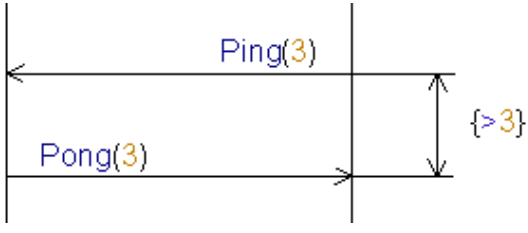


図 267: 「> 指定時間値」または「>= 指定時間値」が設定されたタイマー仕様

すべての状態 :

```
state *;
  input t {
    nextstate -;
  }
```

時間範囲

時間範囲が設定されているタイマー仕様は、指定時間値より大きい時間値を持つタイマー仕様 1 つと指定時間値より小さい時間値を持つタイマー仕様 1 つに変換されます。

テスト アプリケーションの実行

テスト アプリケーションを実行するには、ビルドに使用したビルド アーティファクトを右クリックし、[ビルド] メニューから [テストの実行] を選択します。テスト アプリケーションが実行され、内蔵の Web ブラウザに結果が表示されます。

生成されたテスト入力ファイルを使用してテストが実行され、結果はテスト実行時にテスト ドライバによって作成されたテスト ログ ファイルに保存されます。

モデル ベリファイヤ（Model Verifier）を使用してテスト アプリケーションをビルドした場合、テストの実行も対話形式で制御できます。モデル ベリファイヤ（Model Verifier）でのテスト アプリケーションの実行を参照してください。

テスト ドライバ実行形式ファイルを使用して、Tau の外部でバッチ モードでテスト アプリケーションを実行することもできます。

モデル ベリファイヤ（**Model Verifier**）でのテスト アプリケーションの実行

モデル ベリファイヤ（Model Verifier）でテスト アプリケーションを実行するには、ビルド アーティファクトを右クリックし、[ビルド（Model Verifier）] メニューから [起動] を選択します。これで、テストの実行を手動で制御できます。

1. [実行] をクリックし、Model Verifier コンソール ウィンドウにテスト ケースを入力するよう指示されるまで待ちます。
2. 実行するテスト ケースの名前を入力します。たとえば、「`'TestCase1'`」と入力して「`TestCase1`」を実行します。

テスト ケースの結果は、モデル ベリファイヤ（Model Verifier）の出力ウィンドウに書き込まれます。

注記

次のテスト ケースを実行するには、モデル ベリファイヤ（Model Verifier）を再起動します。

テストの実行結果

Tau からテスト アプリケーションを実行すると、テストの実行結果はテスト ログ ファイルに書き込まれ、テスト終了時に表示されます。

テストの実行とログ記録

このセクションでは、テスト アプリケーションをビルドした後のテストの実施方法について説明します。1508 ページの図 268 に、その構造を示します。

テスト ドライバという実行形式ファイルによってテスト アプリケーションを起動し、テスト ケースを実行します。テストの実行について以下の手順を追って説明します。

1. テスト ドライバがテスト入力ファイルを読み出して、実行するテスト ケースを決定する。
2. テスト ケースを実行するテスト アプリケーションを起動する。
3. テスト ケースが実行されると、テスト アプリケーションがテスト ドライバに判定を報告する。
4. テスト ドライバが判定結果をテスト ログ ファイルに書き込む。
5. 入力ファイルのテスト ケースごとに、ステップ 1 から 4 が繰り返し実行される。

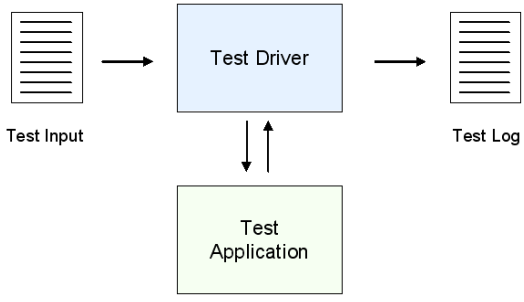


図 268: テスト実行の構造

以下のセクションでは、テストの実行過程の各コンポーネントについて詳しく説明します。

テスト ドライバ

テスト ドライバは、テスト アプリケーションを駆動する実行形式ファイルです。つまり、テスト アプリケーションで一連のテスト ケースを逐次実行します。

このテスト ドライバ実行形式ファイルは `TestDriver.exe` という名前で、インストール フォルダの「bin」フォルダに保存されています。

```
TestDriver -type c testApplication inputFile [logFile]
```

`-type` オプションは、テスト アプリケーションのタイプを指定するために使用します。現時点でサポートされているのは C アプリケーションのみなので、指定できる値は「c」のみです。

`testApplication` パラメータは、生成されたテスト アプリケーションのフルパス名を指定します。

`inputFile` パラメータは、テストの実行中に使用するテスト入力ファイルのフル パス名を指定します。

オプションの `logFile` パラメータは、テストの実行中にテスト ドライバが作成するテスト ログ ファイルのフル パス名を指定します。値を指定しないと、テスト ドライバ コンテキストの標準出力にテスト ログが書き込まれます。

テスト入力ファイル

テスト入力ファイルはテスト ドライバが読み込むファイルです。このファイルには、実行するテスト ケース、実行順序、タイムアウト値が指定されています。

テスト入力ファイルは、以下のフォーマットをもつ XML 形式のテキスト ファイルです。

```
<testinput>
  <testcase>
    <name>testcaseName       </name>
    <timeout>timeout value    </timeout>
  </testcase>
  ...
</testinput>
```

`testcaseName` は実行するテスト ケースの名前で、`timeout value` は、タイムアウトを指定する整数値（ミリ秒）です。このタイムアウト値までに最終判定が生成されなかった場合、テスト ケースの実行が中止されます。タイムアウト要素はオプションです。指定しなかった場合タイムアウトは適用されません。

Tau でテスト モデルをビルドすると、テスト アプリケーションのターゲット ディレクトリに、テスト入力ファイルが自動生成されます。この入力ファイルには、テスト コンテキストのすべてのテスト ケースが含まれます。テスト ケースごとにテスト生成ステレオタイプで指定されたタイムアウトがあります。

テスト ケースを削除する場合、1 つのテスト ケースを複数回実行する場合、あるいはタイムアウトを変更する場合、入力ファイルを手動で変更する必要があります。

テスト ログ ファイル

テスト ログにはテスト実施の結果が書き込まれます。実行されたすべてのテスト ケースは、判定およびログ ファイルからモデルへとナビゲーションするための情報とともに一覧表示されます。テストログは、テスト実施中にテスト ドライバによってテスト生成ステレオタイプで指定した場所に作成されます。

テスト ログは XML 形式で記述され、XSL スタイル シートを使用してログ ファイルをブラウザに表示します。ログ ファイルの作成時に、`ttdTestLog.xsl` というスタイル シート ファイルがテスト ログと同じ場所にコピーされます。スタイル シートの内

容を変更するには、このファイルを同じ名前の別のファイルで置き換えます。ログの作成時にその場所にすでにスタイル シート ファイルが存在する場合、そのファイルは上書きされません。

スタイルシートを置き換えると、テスト ログの表示の仕方が変更されます。

テスト生成ステレオタイプ

テスト生成ステレオタイプは、コード ジェネレータ固有のステレオタイプとともにビルド アーティファクトに適用され、テスト モデルをテスト コンテキストからビルドするよう指定します。

テスト生成ステレオタイプには以下のタグ付き値があります。

- Test Log

  ログ ファイルへのフルパス名。デフォルト値は空です。ログ ファイルはビルド アーティファクトのターゲット ディレクトリに作成されます。

- Timeout

  単一のテスト ケース実行のためのタイムアウト値を指定する整数値。デフォルトでは、すべてのテスト ケースに同じタイムアウト値が設定されています。テスト ケースごとに異なる値を使用するには、コード生成後にテスト入力ファイルを手動で修正する必要があります。

テスト生成ステレオタイプは、ビルド ルートとしてテスト コンテキストを持つビルド アーティファクトにのみ適用できます。

既知の制限事項

このセクションでは、テスト プロファイルに関する既知の制限事項について説明します。

ビルド アーティファクトのタイプ

サポートされるビルド タイプはモデル ベリファイヤ（Model Verifier）のみです。対話形式で実行しているか、テスト ドライバを使用してバッチ形式で実行しているかに関わらず、すべての C コード テスト アプリケーションに使用できます。

テスト ケースの振る舞い

テストの振る舞いは、シーケンス図と状態機械図によってのみ指定できます。

テスト コンテキストが所有するテスト ケースは、シーケンス図で指定し、テスト コンポーネントが所有するテスト ケースは状態機械で指定する必要があります。

テスト ケース

テスト ケースにパラメータは指定できません。

テストケースに含めることができるシーケンス図は 1 つだけです（テスト コンテキストが所有する場合）。

テスト ケースの振る舞い（シーケンス図）

テスト ケースのシーケンス図には、以下のダイアグラム要素のみを含むことができます。

- ライフライン シンボル
- アクション シンボル

  アクション シンボルは、テスト コンポーネントのライフラインでのみ使用可能です。
- 参照シンボル
- メッセージ ライン
- タイマー 仕様ライン

  タイマー仕様ラインの両端は、テスト コンポーネントのライフラインに接続する必要があります。

  `{== x}` または `{x},` など、絶対時間値は意味がないので、タイマー仕様ラインには使用できません。
- コメント シンボル
- テキスト シンボル

テスト コンポーネント

1 つのテスト コンテキストあたりテスト コンポーネントは 1 つのみです。

テスト プロファイル

テスト プロファイルの以下の概念は、テスト生成および実行の目的ではサポートされません。

- デフォルトおよびデフォルト アプリケーション
- Validation アクション、Log アクション、Finish アクション
- `InteractionOperator`
- テスト ログおよび `Testlog` アプリケーション
- テストデータ：ワイルド カードおよびコーディング ルール
- タイム ゾーン

タイマー

タイマーは、状態機械で指定したテスト ケースでのみ使用できます。

DOORS を使った UML モデリング

この章では、以下のセクションで DOORS と Tau を使った UML モデリングについて説明します。

- DOORS へのモデルの格納
- DOORS 内の UML 要素
- トレーサビリティの管理

50

DOORS へのモデルの格納

Tau プロジェクトとモデルを、ファイル・システムに加えて、DOORS に格納できます。既存の DOORS インフラを利用して、モデルの表示、編集、共有が可能になります。

DOORS に格納されたモデルでは、コード生成からモデル・ベリフィケーションまで Tau のすべての機能が利用できます。

この章では DOORS に格納されたモデルの使い方を説明します。

参照

1532 ページの「DOORS で UML 要素を使う」

DOORS への格納の概要

このセクションでは、プロジェクトとモデルを DOORS に格納する方法を説明します。

情報がどのように DOORS で整理されるかを理解するには、Tau ワークスペース、プロジェクト、.u2 ファイルについての基本的な知識が必要です。これらの基礎については、プロジェクトとモデルの基礎で説明しています。詳細については、プロジェクトの操作を参照してください。

すでにこの知識や概念を理解している場合は、以下の各項目に直接進んでください。

- DOOR でのプロジェクト
- DOORS 内のモデル
- デフォルト・プロジェクト
- ファイル・システム
- 同期
- 格納場所と同期

プロジェクトとモデルの基礎

モデルとは、互いに関連し、ともに保存される一連の UML 要素です。ファイル・システムに保存する場合、モデルは .u2 ファイルに保管されます。モデルをロードするには、そのモデルがある特定のプロジェクトに入っている必要があります。

プロジェクトとは、関連する 1 つまたは複数のモデルへの参照を保持するコンテナです。プロジェクトをロードすると、関連するすべてのモデルがロードされます。モデルへの参照のほかに、特定の設定も含まれています。ファイル・システムに保存する場合、プロジェクトは .ttp ファイルに保管されます。プロジェクトの名前は、ファイル名に由来します。

通常、プロジェクトと UML モデルは、複数のユーザーによって共有されます。

ワークスペースとは、一連のプロジェクトへの参照を保持するユーザー固有のコンテナです。通常、ワークスペースは複数ユーザーによって共有されません（共有は可能です）。

参照

20 ページの「プロジェクトの操作」
18 ページの「ワークスペースの操作」

DOOR でのプロジェクト

DOORS 内の各フォーマル・モジュールは 1 つの UML プロジェクトを収納できます。

DOORS に格納されたプロジェクトは、ファイル・システムに格納されたプロジェクトと同様に動作します。DOORS に格納されたプロジェクトは、DOORS に格納されたモデルとファイルシステムに格納されたモデルの両方を参照できます。1 つのプロジェクトをもつ 1 つのフォーマル・モジュールは複数のモデルを収納できます。

DOORS に格納されたプロジェクトの名前は、モジュールに由来しており、手動で変更できません。

DOORS 内のモデル

DOORS 内の各フォーマル・モジュールは、1 つまたは複数のモデルを収納できます。モデルは、モジュール・レベルでもオブジェクト・レベルでも作成できます。モジュール・レベルでは、作成できるモデルは 1 つです。オブジェクト・レベルでは、複数のモデルが作成できます。各モデルは、ある 1 つのプロジェクトへの 1 つの参照、デフォルト・プロジェクトを保持しています。

デフォルト・プロジェクト

あるモジュールまたはオブジェクトののデフォルト・プロジェクトは、ダイアグラムを右クリックするか、または［Open in Tau］コマンドを使って Tau を開いたときに、どのプロジェクトをロードするかを制御します。［Set Default Project］コマンドを使って任意の時点でデフォルト・プロジェクトを変更できます。モデルのデフォルト・プロジェクトを設定するを参照してください。

デフォルト・プロジェクトは、モデルの作成時またはモデルをプロジェクトに初めて追加したときに自動的に設定されます。1 つのモデルは複数のプロジェクトで使用される可能性があるため、後でデフォルト・プロジェクトを変更できるようになっています。

ファイル・システム

DOORS に格納されたプロジェクトとモデルを使う場合でも、いくつかのファイルをファイル・システムに保存する必要があります。たとえば、ワークスペース・ファイル、プロジェクトとダイアグラムに関するユーザー設定ファイル、コード生成の結果の生成物などです。

各プロジェクトには、これらの出力を格納するためのデフォルトの出力場所があります。各プロジェクトについて、名前としてプロジェクト・モジュール id の付いたフォルダが、Tau のデフォルト・プロジェクト配置場所に作成されます。

通常の完全パス名は以下のとおりです。

```
C:\Documents and Settings\User Name\My Documents\My Projects\Module Id
```

デフォルトのプロジェクト配置場所は、［オプション］ダイアログで変更できます。

同期

ダイアグラムの画像を含んだモデルの内容は、モデルとモジュールを同期することでモジュールに再表示できます。詳細は、「DOORS 内の UML 要素」を参照してください。

格納場所と同期

プロジェクトとモデルは、UML 要素の表示や UML 要素とオブジェクトの同期を行わずに格納できます。言い換えると、格納と同期は完全に独立です。フォーマル・モジュールがモデルを収納しているということが、そのままそのモジュールが代理モジュールである、ということにはなりません。

格納場所にかかわらず、任意の UML モデルを DOORS 内のフォーマル・モジュールと同期できます。

DOORS 内のプロジェクトを使う

このセクションでは、DOORS 内のプロジェクトの使用法について説明します。以下のトピックをカバーします。

- DOOR でプロジェクトを作成する
- DOORS からプロジェクトをロードする
- ［Select UML Project］ダイアログ

参照

DOORS 内のモデルを使う

DOOR でプロジェクトを作成する

UML プロジェクトを作成するには、以下の手順を実行します。

- フォーマル・モジュールを作成して、排他編集モードで開きます。
  モジュールについて特有の要件はありません。すでにプロジェクトを保持していないモジュールであれば、どのモジュールでもかまいません。
- ［Tau］メニューから［Add UML Model...］コマンドを実行します。
- ［Select UML Project］ダイアログで［Create new project］オプションを選択します。
- 必要なプロジェクト・タイプを選択して［OK］をクリックします。
  プロジェクト・タイプはモデルの［モデルビュー］での外観と同期処理に影響します。
- 必要な［データ同期設定］設定を設定して［OK］をクリックします。

DOORS からプロジェクトをロードする

UML プロジェクトは 2 つの異なる方法でロードできます。明示的にプロジェクトを指定する方法と、モデルを指定してプロジェクトは暗黙的に指定する方法です。

UML をプロジェクトをロードするには、以下の手順を実行します。

- メイン エクスプローラ ウィンドウから Tau メニューの［Load UML Project...］コマンドを実行します。
- 必要なプロジェクトを含むモジュールを確認して［OK］をクリックします。

Tau が起動して、プロジェクトとそのプロジェクトが参照するモデルをすべてロードします。

モデルをロードすることでプロジェクトを暗黙的にロードする方法については、「モデルを Tau で開く」のセクションで説明しています。

［Select UML Project］ダイアログ

このダイアログを使用して、既存のプロジェクトの選択と新規プロジェクトの作成ができます。このダイアログは複数の異なるコマンドから使用されます。たとえば、［Add UML Model...］や［Add UML Model to Project...］などです。

このダイアログはプロジェクトの指定方法として以下のやり方を提供しています。

- **Create new project**

  Tau の［Developer ウィザード］を使用して新規 UML プロジェクトを作成します。
- **Add to active Tau project**

  現在アクティブな Tau のプロジェクトに UML モデルを追加します。
- **Add to existing project**

  DOORS に収納されている既存のプロジェクトを表示し、モデルをそのプロジェクトに追加します。

既存のモデルをあるプロジェクトに追加する際に、［Use as default project］チェックボックスを使用して、プロジェクトをモデルのデフォルト・プロジェクトにするかどうかを選択できます。

プロジェクトの作成や変更を行うには、ユーザーはそのプロジェクトを収納しているモジュールに対してモジュール・レベルの書き込みアクセス権限が必要です。

DOORS 内のモデルを使う

このセクションでは、DOORS 内の UML モデルの使用法について説明します。

- 新規モデルを作成する
- モデルを Tau で開く
- 複数のプロジェクトに属するモデルを使う
- モデルのデフォルト・プロジェクトを設定する
- モデルを分割する
- モデルを削除する

参照

モジュール・レベルとオブジェクト・レベル

DOORS 内のプロジェクトを使う

新規モデルを作成する

モデルは二つの異なるレベル、つまり、モジュール・レベルとオブジェクト・レベルで作成できます。詳細は、モジュール・レベルとオブジェクト・レベルを参照してください。

モジュール・レベル

新規モデルをモジュール・レベルで作成するには、以下の手順を実行します。

1. フォーマル・モジュールを作成して、排他編集モードで開きます。

   モジュールについて特有の要件はありません。すでにモジュールレベルでモデルを保持していないモジュールであれば、どのモジュールでもかまいません。
2. ［Tau］メニューから［Add UML Model..］コマンドを実行します。
3. ［Select UML Project］ダイアログを使用して、新規モデル用にプロジェクトを作成、選択します。

デフォルトで新規モデルの同期はデフォルト同期設定を使って行われます。この設定の変更方法については、「同期設定の変更 」を参照してください。

オブジェクト・レベル

新規モデルをオブジェクト・レベルで作成するには、以下の手順を実行します。

1. フォーマル・モジュール内のオブジェクトを選択します。オブジェクトへの書き込みアクセス権限があることを確認してください。
2. 右クリックして、コンテキスト・メニューから［Add UML Model...］コマンドを実行します。
3. ［Select UML Project］ダイアログを使用して、新規モデル用にプロジェクトを作成、選択します。

デフォルトで新規モデルの同期はデフォルト同期設定を使って行われます。この設定の変更方法については、「同期設定の変更」を参照してください。

注記

モデルを収納しているモジュールと同じモジュールにプロジェクトを作成する、または同じモジュールのプロジェクトを修正するには、モジュール・レベルの書き込みアクセス権限が必要です。モデルを同じモジュール内のプロジェクトに追加するには、排他編集モードであることを確認してください。

モデルを **Tau** で開く

UML モデルを Tau で開くには、以下のいくつかのやり方があります。

- メイン・エクスプローラ・ウィンドウから、［Tau］メニューの［Load UML Project...］コマンドを使用する。「DOORS からプロジェクトをロードする」を参照してください。
- フォーマル・モジュール内で、［Tau］メニューの［Open in Tau］コマンドを使用する。
- ダイアグラムの画像をダブル・クリックする。

モデルを Tau で開く際に、モデルのデフォルト・プロジェクトによってどのプロジェクトがロードされるかが決定されます。

プロジェクトとそのプロジェクトが参照しているモデルがロードされると、Tau でのモデルの編集モードは、対応する DOORS モジュールの現在の編集モードに依存するようになります。DOOR でモジュールが読み取り専用モードで開かれていると、Tau でもモデルは読み取り専用になります。読み取り専用のモデルを編集可能にするには［Make Editable］コマンドを使用します。詳細は、「読み取り専用のモデルを編集可能にする」を参照してください。

複数のプロジェクトに属するモデルを使う

ある 1 つのモデルは一時点で任意の数のプロジェクトに収納され得ます。

［Add UML Model to Project］コマンドを使用すると、既存のモデルをあるプロジェクトに追加できます。

モジュール・レベル

モジュール・レベルのモデルをあるプロジェクトに追加するには、以下の手順を実行します。

1. モデルを収納しているフォーマル・モジュールを開きます。

   任意の編集モードが使用できますが、新しいプロジェクトをモデルのデフォルト・プロジェクトにするには書き込みアクセス権限が必要です。
2. フォーマル・モジュール内で、［Tau］メニューの［Add UML Model to Project］コマンドを実行します。

3. ［Select UML Project］ダイアログを使用して、モデルの追加先であるプロジェクトを選択します。

   ［Use as default project］チェックボックスを使用すると、このプロジェクトをデフォルト・プロジェクトにするかどうかを制御できます。デフォルト・プロジェクトは後で変更できます。「モデルのデフォルト・プロジェクトを設定する」を参照してください。

オブジェクト・レベル

オブジェクト・レベルのモデルをあるプロジェクトに追加するには、以下の手順を実行します。

1. フォーマル・モジュールを開いてモデルを収納しているオブジェクトを選択します。

   任意の編集モードが使用できますが、新しいプロジェクトをモデルのデフォルト・プロジェクトにするには書き込みアクセス権限が必要です。
2. 右クリックして、コンテキスト・メニューから［Add UML Model to Project］コマンドを実行します。
3. ［Select UML Project］ダイアログを使用して、モデルの追加先であるプロジェクトを選択します。

   ［Use as default project］チェックボックスを使用すると、このプロジェクトをデフォルト・プロジェクトにするかどうかを制御できます。デフォルト・プロジェクトは後で変更できます。「モデルのデフォルト・プロジェクトを設定する」を参照してください。

注記

モデルを収納しているモジュールと同じモジュールにプロジェクトを作成する、または同じモジュールのプロジェクトを修正するには、モジュール・レベルの書き込みアクセス権限が必要です。モデルを同じモジュール内のプロジェクトに追加するには、排他編集モードであることを確認してください。

モデルのデフォルト・プロジェクトを設定する

あるモデルのデフォルト・プロジェクトを設定するには、以下の手順を実行します。

モジュール・レベル

1. モデルを収納しているフォーマル・モジュールを排他編集モードで開きます。
2. ［Tau］メニューから［Set Default Project］コマンドを実行します。
3. 必要なプロジェクト・タイプを選択して［OK］をクリックします。

オブジェクト・レベル

1. モデルを収納しているフォーマル・モジュールを排他編集モードまたは共有編集モードで開きます。
2. モデルを収納しているオブジェクトがロックされていることを確認します。

3. オブジェクトを右クリックして、コンテキスト・メニューから［Set Default Project］コマンドを実行します。
4. 必要なプロジェクト・タイプを選択して［OK］をクリックします。

モデルを分割する

既存のモデルを二つに分割できます。異なるユーザーがモデルの異なる部分について同時に作業する場合に有用です。分割されたモデルは、異なる編集セクションで使用することも、異なるプロジェクトで使用することもできます。

既存のモデルを分割するには、新しいモデルの最上位要素となるオブジェクトを右クリックし、コンテキスト・メニューから［Split UML Model...］コマンドを実行します。

モデルを分割すると以下のように動作します。

1. 元のモデルのデフォルト・プロジェクトがロードされます。
2. 新しいモデルが作成され、プロジェクトに挿入されます。新しいモデルのデータ同期設定は元のモデルと同じです。
3. そのオブジェクトとその子を現す UML 要素は新しいファイルに移動します。
4. プロジェクトとモデルが保存されます。

両モデルがともにロードされるときに、モデル階層に手は加えられません。つまり、新しいモデルに分割された UML 要素も、構成要素としては依然として元の要素に所有されています。

新しいモデルは元のモデルを収納していない別のプロジェクトで使用できます。この場合、モデルは最上位に表示されます。

注記

元のモデルが複数のプロジェクトから参照されていた場合、新しいモデルはこれらの他のプロジェクトに自動的には追加されません。新しいモデルをこれらのプロジェクトに追加するには、［Add Model to Project］コマンドを使用します。「複数のプロジェクトに属するモデルを使う」を参照してください。

モデルを削除する

UML モデルと関連する属性をフォーマル • モジュールから削除するには、以下の手順を実行します。

- モデルを収納しているフォーマル • モジュールを排他編集モードで開きます。
- ［Tau］メニューから［Disable Tau］コマンドを実行します。

このコマンドを使用すると、モジュールに収納されているすべての UML モデルがその関連する属性やビューとともに削除されます。**UML 要素を現しているオブジェクトは削除されません。**

注記

モデルを削除しても、プロジェクトのモデルへの参照が自動的に削除されるわけではありません。プロジェクトからのモデルの削除は手動で行う必要があります。

モジュール・レベルとオブジェクト・レベル

DOORS 内の情報は二つの異なるレベルに格納できます。「モジュール」と「オブジェクト」です。モデルはどちらのレベルにも格納できます。この仕組みによって、モデルを柔軟に構成でき、任意のフォーマル・モジュール構造に応じることができます。

1 つのフォーマル・モジュールは複数のモデルを収納できます。モジュール・レベルのモデルと各オブジェクト・レベルのモデルです。編集可能セクションとオブジェクト・レベルに収納されたモデルとの間に緊密な関係はありません。モデルは、モジュールにセクションがない場合でも、オブジェクト・レベルで格納できます。

モデルの構造は、フォーマル・モジュールとユーザーによるフォーマル・モジュールの使い方に合せて設定される必要があります。一般的には、以下のシナリオがあてはまります。

- **モジュールを排他編集モードのみで使用する**

  1 つのモデルをモジュール・レベルで収納するモジュールに、1 つのプロジェクトを作成します。

  モデルは、他のプロジェクトに追加することによって他のモデルからも参照できます。「複数のプロジェクトに属するモデルを使う」を参照してください。

  モデルの一部だけを他のプロジェクトから参照する必要がある場合は、その部分を別のモデルとして分割する方法があります。この方法を使うと、他のプロジェクトにロードされるモデルのサイズは小さくなり、パフォーマンスは向上します。モデルの分割の詳細については、「モデルを分割する」を参照してください。

- **モジュールを編集可能セクションに分けて使用する - 非階層型モデル**

  各編集セクションごとに 1 つのモデルを収納しているモジュールに 1 つのプロジェクトを作成します。モデルは、互いに他の要素を参照、使用し合う一連の関連したものになります。

  共有編集モード下でユーザーがあるセクションで作業していて、プロジェクトをロードするときには、編集可能となるロックされたセクションのモデル以外は、すべてのモデルが読み取り専用モードでロードされます。

  この方法を使うと、すべてのモデルは他のモデルの要素を使用でき、プロジェクトをロードする際の編集モードは DOORS の編集モードを反映したものになります。

- **モジュールを編集可能セクションに分けて使用する - 階層型モデル**

  1 つのモデルをモジュール・レベルで収納するモジュールに、1 つのプロジェクトを作成します。モデルを各セクションで分割するには、「モデルを分割する」を参照してください。結果として、いくつかの場所に格納された 1 つの階層モデルになります。

  共有編集モード下でユーザーがあるセクションで作業していて、プロジェクトをロードするときには、編集可能となるロックされたセクション内のブランチ以外は、モデルが読み取り専用モードでロードされます。

Tau 内のプロジェクトを使う

このセクションでは、DOORS に格納されているプロジェクトの Tau での使用法について説明します。

- Tau でプロジェクトを作成する
- DOORS からプロジェクトをロードする

Tau でのプロジェクトの使用法の詳細については、20 ページの「プロジェクトの操作」を参照してください。

参照

Tau 内のモデルを使う

Tau でプロジェクトを作成する

新規の UML プロジェクトを作成するには、以下の手順を実行します。

- ［ファイル］、［新規］コマンドから［新規］ウィザードを開きます。［プロジェクト］タブがアクティブになっていることを確認してください。
- プロジェクト・タイプを選択します。
- プロジェクトを DOORS に格納するには、［保管先］で［DOORS］ラジオボタンをチェックします。
- プロジェクト・モジュールを指定します。以下の 2 つの方法があります。

  新しいモジュールを作成する

  ［名前］エディットボックスにモジュールの名前を入力します。［...］ボタンでモジュールを作成する場所を選びます。プロジェクトかフォルダを選択して、［OK］をクリックします。選択した要素へのパスが［場所］エディットボックスに挿入されます。パスは手動でも入力できます。

  既存のモジュールを使用する

  ［...］ボタンで UML プロジェクトを格納する DOORS モジュールを選びます。モジュールを選択して、［OK］をクリックします。選択したモジュールにしたがって［名前］エディットボックスと［場所］エディットボックスの内容が更新されます。値は手動でも入力できます。
- ［OK］をクリックします。
- ［モデルを作成する］チェックボックスを有効にして、プロジェクトにモデルを作成します。

  プロジェクトの場所と同様にモデルの場所を指定します。必要なデータ同期設定を設定します。
- ［完了］をクリックします。

DOORS からプロジェクトをロードする

DOORS に格納されたプロジェクトをロードするには、以下の手順を実行します。

- ［ファイル］メニューから［DOORS プロジェクトを開く ...］コマンドを実行します。

  このコマンドは、DOORS がまだ起動していない場合は DOORS を起動し、ログインを要求してきます。
- プロジェクトを収納しているモジュールを検索して、［OK］をクリックします。

プロジェクトとすべての参照されているモデルがロードされます。プロジェクトの任意のモデルが DOORS と同期していて、［オープン / 保存時の更新 / コミット］設定が有効になっている場合は、モデルは自動的に DOORS から更新されます。

Tau 内のモデルを使う

このセクションでは、DOORS に格納されているモデルの Tau での使用法について説明します。

- モデルを DOORS に保存する
- 読み取り専用のモデルを編集可能にする

モデルを DOORS に保存する

既存のモデルを DOORS に保存するには、以下の手順を実行します。

- ［モデルビュー］で必要なモデル要素を右クリックして、［DOORS に保存 ...］を選択します。
- モデルを格納したいフォーマル・モジュールを選択して、［OK］をクリックします。

このコマンドを使用して、モデル全体を保存することも、モデルの一部を保存することもできます。モデルは DOORS に保存されるだけであり、同期は取られません。同期を取るには、［DOORS と同期］コマンドを使用します。「UML モデルを同期用に設定する」を参照してください。

注記

この操作では現在モデルをモジュール・レベルに格納することのみをサポートしています。

読み取り専用のモデルを編集可能にする

読み取り専用モデルを編集可能にするには、［モデルビュー］またはダイアグラムのコンテキスト・メニューから［編集可能にする］コマンドを実行します。

このコマンドは、DOORS 内の編集モードを変更してモデルを編集、保存できるようにします。モデルがモジュール・レベルで格納されている場合は、編集モードは排他編集モードに設定されます。モデルがオブジェクト・レベルで格納されていて編集可能セクションがある場合は、共有編集モードが使用されます。

51

DOORS 内の UML 要素

UML モデルの内容を DOORS フォーマルモジュール内にオブジェクトおよび画像として表示させることができます。この機能は、モデルとフォーマルモジュールの「同期」操作を通して実現します。DOORS で作成したモデルは、デフォルトで常に Tau の内容と同期しています。

UML 要素をもつモジュールを、「代理モジュール」と呼びます。また、フォーマルモジュールと同期するモデルは、「エクスポートされたモデル」と呼びます。

以下のトピックを説明します。

- DOORS で UML 要素を使う
- DOORS で同期を取る
- Tau で同期を取る
- データ同期設定
- Analyst View
- 属性
- リンク
- フィルタ

参照

1517 ページの「DOORS へのモデルの格納」
1520 ページの「格納場所と同期」

DOORS で UML 要素を使う

このセクションでは、DOORS で UML 要素を使用する方法と DOORS 属性のモデルへのプロパゲート方法を説明します。

以下のセクションに、UML 情報を DOORS 内でどのように表示するかについての情報があります。

- DOORS における UML 要素の表現
- DOORS 属性の UML 表現

以下のセクションで UML 要素の取り扱い方法を説明します。

- UML 要素を作成する
- UML 要素を削除する
- UML 要素を移動する
- UML 要素の名前を変更する
- UML 要素を Tau で編集する
- ユースケースを作成する

以下のセクションでは、DOORS 属性の値をモデルにプロパゲートし、ダイアグラムで表示させる方法を説明します。

- ダイアグラムでオブジェクト・テキストを表示する
- ダイアグラムで属性を表示する

参照

1523 ページの「DOORS 内のモデルを使う」
1521 ページの「DOORS 内のプロジェクトを使う」
1537 ページの「DOORS で同期を取る」

DOORS における UML 要素の表現

1 つの UML 要素は、1 つのオブジェクトとして表現されます。UML 特有の情報を格納するために、いくつかの属性を使用します。

- UML Kind
- UML Name
- UML Location

ダイアグラムについては、付加的なオブジェクトが作成されて、ダイアグラムの画像情報を保持します（[ダイアグラム画像の表示] 設定が有効の場合）。画像ファイルは Windows メタファイル形式（.wmf ファイル）で保存されます。

DOORS 属性の UML 表現

すべてのテキストベースの属性の値は、モデルにプロパゲートして、ダイアグラムで表示できます。ただし、DXL レイアウト属性はモデルにプロパゲートできません。

オブジェクト テキスト

［オブジェクト テキスト］属性の値は、デフォルトで、同期中に DOORS/Tau 間でプロパゲートされます。この値はモデル内の専用のコメントとして［Object Text］という見出しを付けて格納されます。

コメントの表示、非表示、削除方法については、「ダイアグラムでオブジェクト・テキストを表示する」を参照してください。

オブジェクトテキストの同期は、UML Comment Symbol 属性が制御します。この属性値のオンオフは任意の時点で行えます。

他の属性

任意のテキストベースの属性の値は、同期中に DOORS/Tau 間でプロパゲートできます。「ダイアグラムで属性を表示する」を参照してください。

属性値はモデル内の専用のコメントとして［Attributes］という見出しを付けて格納されます。コメントの本体には属性の名前と値が以下の形式で記述されます。

属性名：属性値

空の値をもつ属性は含まれません。

UML 要素を作成する

フォーマルモジュールに新しい UML 要素を作成するには、以下の手順を実行します。

［Analyst View］を使用する

- 通常のオブジェクトを作成して適当な名前を付けます。
- ［Analysis Type］カラムで要素タイプを選択します。
  一部の要素タイプは他の要素タイプに含ませることができません。不適切な要素タイプを選択すると、エラー・メッセージが表示されます。

任意のビューを使用する

- オブジェクト（オブジェクトがまだない場合は、空白の領域）を右クリックして、コンテキスト・メニューの［Insert UML］メニューを強調表示します。
- 要素タイプと場所を選択して、［OK］をクリックします。S
- 要素に名前を付けます。

モデルは、次回 Tau で同期を取った時に更新されます。DOORS から明示的に同期を取るには［Commit to Tau］コマンドを使用します。

UML 要素を削除する

UML 要素を削除するには、その要素を現しているオブジェクトを削除します。モデルは、次回 Tau で同期を取った時に更新されます。明示的に同期を取るには［Commit to Tau］コマンドを使用します。

UML 要素を移動する

UML 要素を移動するには、DOORS でオブジェクトを移動します。

重要！

オブジェクトは同じモジュール内でのみ移動できます。また、もし編集可能なセクションを使用している場合は、そのセクション内でのみ移動できます。オブジェクトを他のモジュールやセクションに移動すると、同期は失敗し、オブジェクトが削除される可能性があります。

Tau にコミットしていない要素を移動する際は、要素を階層中の正しい位置に移動する必要があります。要素を正しくない位置に移動すると、Analyst View 内のアイコンの上に小さい赤色の感嘆符（!）が表示され、この要素は同期されません。

UML 要素の名前を変更する

モデル要素の名前はオブジェクトのオブジェクト・ヘディングと同じです。モデル内で名前を変更するには、オブジェクト・ヘディングを変更したい値に変えます。モデルは、次回 Tau で同期を取った時に更新されます。明示的に同期を取るには［Commit to Tau］コマンドを使用します。

UML 要素を Tau で編集する

UML 要素を Tau で編集するには、以下の手順を実行します。

- オブジェクトを右クリックして［Open in Tau］を選択します。

ダイアグラムを Tau で編集するには、以下の手順を実行します。

- ダイアグラム画像をダブルクリックします。

この 2 つのコマンドは、要素を含んだモデルをそのモデルのデフォルト・プロジェクトからロードします。詳細はデフォルト・プロジェクト を参照してください。

ユースケースを作成する

このコマンドは、モジュール内にユースケースを作成して元の要件とそのユースケースのリンクを張ることで、要件を詳細化するためのものです。「UML 要素を作成する 」も参照してください。

ユースケースを作成するには、以下の手順を実行します。

- 要件に該当するオブジェクトを選択します。
- オブジェクトを右クリックして、［Create UML Use Case］コマンドを選択します。

ウィザードが起動され、以下の手順にしたがってユースケースの詳細を指定できます。

- **UML Model** は、ユースケースが作成されるモデルを指定します。
- **Context** は、ユースケースが作成される UML 要素を指定します。モデル内のすべての適切な要素が表示されます。
- **Details** は、ユースケースについての情報を指定します。

  ユースケース名はエディット・ボックスに名前を入力するか、［Base for text:］ドロップダウン・リスト内の属性を選択して［Set base］ボタンをクリックして、指定します。後者の操作で名前は、選択した属性の要件値に変更されます。

  詳細な情報として［Descriptions］と［Constraints］も入力できます。

すべての情報を指定して［Finish］ボタンをクリックすると、指定したモデルにユースケースが作成され、ユースケースから要件へのリンクが張られ、最後にモデルが Tau にコミットされます。選択したワークスペース内と代理モジュール内に作成されます。

ダイアグラムでオブジェクト・テキストを表示する

あるオブジェクトのオブジェクト・テキスト属性の値は、デフォルトで同期中にモデルに格納されます。この値は Tau で表示、編集できます。

ダイアグラムでオブジェクト・テキストを表示するには :

- Tau でモジュールを開きます。
- ダイアグラムを開き、必要なシンボルを選択します。
- シンボルを右クリックして、［表示 / 非表示］、［コメントを表示］を選択します。

コメント・シンボルを削除するには :

シンボルを選択して［削除］をクリックします。コメントはダイアグラム上からは削除されますが、モデルからは削除されません。上で述べた手順を使って再表示できます。

Tau でオブジェクト・テキストを編集するには :

- コメント・シンボルにオブジェクト・テキストが表示されていることを確認します。
- テキストを編集します。見出しの［Object Text］は変更しないでください。

Tau でオブジェクト・テキストをオブジェクトに追加するには :

- オブジェクトを現しているシンボルがあることを確認します。
- コメント・シンボルを作成して、他のシンボルに付加します。
- コメント・シンボルの最初の行に "Object Text" と入力します。
- 第二行以降にオブジェクト・テキストにしたいテキストを入力します。

同期中に、テキストは対応する DOORS オブジェクト内のオブジェクト・テキストとして挿入されます。

参照

1536 ページの「ダイアグラムで属性を表示する」

ダイアグラムで属性を表示する

DOORS 属性の値はモデルと同期させてダイアグラム上に表示できます。

オブジェクトの属性値を表示するには、以下の手順を実行します。

- オブジェクトを選択します。
- Tau メニューから［Select Attributes to Show in Tau］コマンドを実行します。
- 表示させたい属性を選択します。
- オブジェクトを右クリックして、［Open in Tau］を選択します。
- オブジェクトを現しているシンボルを選択します。
- シンボルを右クリックして、［表示 / 非表示］、［コメントを表示］を選択します。
- DOORS ツールバーで［コミット］ボタンをクリックして、ダイアグラムでの変更をフォーマル・モジュールにコミットします。

コメント・シンボルが作成されて、シンボルに付加されます。コメントには "Attributes" という見出しが付き、属性名と属性値が以下の形式で表示されます。

属性名：属性値 `attribute name : attribute value`

空の値をもつ属性は含まれません。

コメント・シンボルを削除するには、シンボルを選択して［削除］をクリックします。コメントはダイアグラム上からは削除されますが、モデルからは削除されません。上で述べた手順を使って再表示できます。

参照

1535 ページの「ダイアグラムでオブジェクト・テキストを表示する」

DOORS で同期を取る

このセクションでは、UML 要素を保持するフォーマル・モジュールとモデルの同期方法について説明します。

- 変更をモデルにコミットする
- モデルの変更でモジュールを更新する

変更をモデルにコミットする

フォーマル・モジュールでの変更をモデルにコミットするには、以下の手順を実行します。

モジュール・レベルの場合

- Tau メニューから［Commit to Tau］コマンドを実行します。

オブジェクト・レベルの場合

- モデルを含んでいるオブジェクトを選択します。
- Tau メニューから［Commit to Tau］コマンドを実行します。

重要！

モデルが Tau にロードされていて Tau で変更が加えられていた場合、その変更は、DOORS での変更をコミットしたときに失われます。モデルに変更を加える場合、一時点では 1 つのツールからのみ行うようにしてください。

注記

［オープン / 保存時の更新 / コミット］設定が有効になっている場合、フォーマル・モジュールでの変更は Tau でモデルを開いたときに自動的にコミットされます。

参照

1527 ページの「モジュール・レベルとオブジェクト・レベル」

モデルの変更でモジュールを更新する

モデルの変更でフォーマル・モジュールを更新するには、以下の手順を実行します。

モジュール・レベルの場合

- Tau メニューから［Update from Tau］コマンドを実行します。

オブジェクト・レベルの場合

- モデルを含んでいるオブジェクトを選択します。
- Tau メニューから［Update from Tau］コマンドを実行します。

フォーマル・モジュールは、［データ同期設定］の設定にしたがってモデルの変更を反映して更新されます。

重要！

UML 要素を現しているオブジェクトがフォーマル・モジュール内で変更された場合、その変更は Tau から更新を行うと失われます。モデルに変更を加える場合、一時点では 1 つのツールからのみ行うようにしてください。

注記

［オープン / 保存時の更新 / コミット］設定が有効になっている場合、Tau 内のモデルでの変更は DOORS でモジュールを保存したときに自動的にコミットされます。

参照

1527 ページの「モジュール・レベルとオブジェクト・レベル」

Tau で同期を取る

このセクションでは、Tau でのモデルの同期について説明します。

DOORS とのモデルの同期の起動方法に関する情報は、「UML モデルを同期用に設定する」を参照してください。

Tau/DOORS 間での変更のプロパゲート方法については、以下の項目を参照してください。

- モデルの変更を DOORS にコミットする
- DOORS で行った変更でモデルを更新する

以下のセクションでは、モデル同期のカスタマイズ方法について説明します。

- モデルの同期を無効にする
- 1 つの要素の同期を無効にする
- 1 つの要素の同期を有効にする
- DOORS でダイアグラムの画像を表示 / 非表示にする
- 同期設定の変更

UML モデルを同期用に設定する

UML モデルまたはその一部をフォーマル・モジュールと同期するには、以下の手順を実行します。

1. ［モデルビュー］で要素を選択します。
2. 要素を右クリックして、［DOORS］サブメニューから［同期 ... ］を選択します。
3. ［DOORS と同期する］チェックボックスをチェックします。
4. ［...］ボタンをクリックして同期先の場所を選択します。
5. 必要な データ同期設定を設定します。
6. ［OK］をクリックします。

選択された要素が、指定の同期設定を使用してフォーマル・モジュールと同期します。その結果、同期された要素が選択された要素を最上位にした階層を構成します。同期で使用したメタモデルがどの要素を同期するのかを制御します。

同期した要素には［モデルビュー］で小さい DOORS アイコンが表示され、同期が取られたことを示します。

注記

同期された要素の直接または間接の子要素のように、同期階層にすでに属している要素は同期できません。

モデルの変更を DOORS にコミットする

モデルの変更を DOORS にコミットするには、以下の手順を実行します。

- ［モデルビュー］で、同期する最上位の要素を右クリックします。
- DOORS サブメニューから［コミット］コマンドを実行します。

モデルの変更は、現在の データ同期設定にしたがって DOORS にコミットされます。

重要！

モデル要素が Tau にロードされた後で DOORS で変更された場合、その変更は上書きされます。

注記

［オープン / 保存時の更新 / コミット］設定が有効になっている場合は、モデルは保存時に自動的にコミットされます。

DOORS で行った変更でモデルを更新する

DOORS で行った変更でモデルを更新するには、以下の手順を実行します。

- ［モデルビュー］で、同期する最上位の要素を右クリックします。
- DOORS サブメニューから［更新］コマンドを実行します。

DOORS で行った変更でモデルが更新されます。

重要！

モデル要素が DOORS からロードされた後で Tau で変更された場合、その変更は上書きされます。

注記

［オープン / 保存時の更新 / コミット］設定が有効になっている場合は、モデルはロード時に自動的にコミットされます。

モデルの同期を無効にする

同期されているモデル（またはその一部）の同期を取らない（以前のリリースではエクスポートの取り消しと呼ばれていました）ようにするには、以下の手順を実行します。

1. ［モデルビュー］で要素を選択します。
2. 要素を右クリックして、［DOORS］サブメニューから［同期 ... ］を選択します。
3. ［DOOR と同期する］チェックボックスのチェックをはずします。
4. ［OK］をクリックします。

モデルとフォーマル • モジュールは切り離され、データの同期は無効になります。このコマンドは、UML データをフォーマル • モジュールから削除しません。

モデル内の 1 つの要素の同期を無効にする方法については、「1 つの要素の同期を無効にする」を参照してください。

1 つの要素の同期を無効にする

1 つの要素（その子要素を含む）の同期を無効にするには、以下の手順を実行します。

1. ［モデルビュー］で要素を選択します。
2. 要素を右クリックして、［DOORS］サブメニューから［同期を無効にする］を選択します。

その要素とそのすべての子要素はモデルの同期処理からはずされます。要素がすでに DOORS にある場合、削除されます。この変更を適用するには、モデルを DOORS にコミットする必要があります。

1 つの要素の同期を有効にする

1 つの要素の同期を有効にするには、以下の手順を実行します。

1. ［モデルビュー］で要素を選択します。
2. 要素を右クリックして、［DOORS］サブメニューから［同期を有効にする］を選択します。

その要素は次回 DOORS にコミットする際に同期処理に含まれます。

このコマンドは明示的に同期からはずされた要素のために用意されています。「1 つの要素の同期を無効にする」を参照してください。

DOORS でダイアグラムの画像を表示 / 非表示にする

フォーマル • モジュールでダイアグラムの画像を表示 / 非表示にするには、以下の手順を実行します。

- ［モデルビュー］でダイアグラムを右クリックするか、ダイアグラム・エディタのキャンバスで右クリックします。
- ［ダイアグラム画像の表示］エントリの値を切り替えます。

値を有効にすると、次回の同期処理でダイアグラム画像がフォーマル • モジュールで表示されます。値を無効にすると、すでに表示されているダイアグラム画像がフォーマル • モジュールから取り除かれます。

このコマンドは個々のダイアグラムの同期を制御するために使用されます。モデルのデフォルトの設定を変更するには、［ダイアグラム画像の表示］同期設定を使用します。

同期設定の変更

同期されているモデルの同期設定を変更するには、以下の手順を実行します。

1. ［モデルビュー］で、同期する最上位の要素を右クリックします。
2. 要素を右クリックして、［DOORS］サブメニューから［同期 ... ］を選択します。
3. 必要なデータ同期設定を設定します。

4. ［OK］をクリックします。

データ同期設定

Tau と DOORS の間でのデータ同期を制御する方法は、同期設定によっていくつか方法があります。

同期設定は、DOORS に新規モデルを格納する際に［新規］ウィザードで設定できます。また、エクスポートされている要素向けに［モデルビュー］のコンテキスト・メニューから［同期 ...］コマンドを使って変更できます。

場所

モデルの同期の対象である DOORS モジュールまたはオブジェクト。すでに同期されているモデルについての場所の変更は、同期の無効化と別の場所への同期の有効化と同等です。

メタモデル

同期に使用されるメタモデル。同期はメタモデルベースの処理であり、選択したメタモデルで可視の要素のみについて同期が取られます。同期には、ユーザー定義のものを含め、任意のメタモデルを使用できます。メタモデルの詳細については、321 ページの「メタモデル」を参照してください。

データ同期の方向

ツール間のどちらの方向へのデータの同期を許可するかを制御するコントロール。データ同期の方向は、オブジェクトとリンクのそれぞれについて個別に制御できます。

以下の値を使用できます。

- **None**

  ツール間でモデル・データの同期は取らない。
- **Commit**

  Tau から DOORS に向けてモデル変更を行う。DOORS での変更は動作中に失われる。
- **Update**

  DOORS から Tau に向けてモデル変更を行う。Tau でのモデルの変更は動作中に失われる。
- **Update and Commit**

  モデル変更を両方向に行う。ただし、一時点では一方向のみ。同期先のツールでの変更は動作中に失われる。

ダイアグラム画像の表示

このオプションを有効にすると、コミット操作中にフォーマル • モジュールにダイアグラムの画像が作成され、挿入されます。

最上位要素の表示

このオプションを有効にすると、エクスポートされた最上位の UML 要素が DOORS 内に別オブジェクトとして表示されます。

オープン / 保存時の更新 / コミット

このオプションを有効にすると、モデルのオープン時と保存時にデータの同期が自動的に取られます。モデルがロードされる際は DOORS で行われた変更で更新され、保存の際には任意の変更が DOORS にコミットされます。

Analyst View

［Analyst View］は DOORS での UML 要素を使った作業向けに設計されたビューです。モデルが DOORS に作成されると、このビューは自動的に作成されます。

標準のオブジェクト・ヘディングカラムの他に二つのカラムが定義されています。

- Analysis Type
- UML Element Icon

Analysis Type

The **Analysis Type** カラムには UML Kind 属性の値が格納され、UML 要素のタイプを設定、変更するために使用します。詳細は「UML 要素を作成する」を参照してください。

UML Element Icon

この名前のないカラムは UML 要素を現すアイコンを表示します。アイコンは UMLKind 属性の値に従って変更されます。

このカラムは、要素が無効な位置に移動したことを示すためにも使用されます。同期されていないオブジェクトが無効な場所に移動されると、小さい赤色の感嘆符が UML アイコンの上に表示されます。このマークは、オブジェクトがコンテキストを外れており同期できないことを現しています。

属性

UML 関連の情報をオブジェクトに格納するために、数多くの DOORS 属性が使用されます。最も重要な属性は、以下の属性です。

- UML Kind
- UML Name
- UML Location
- UML Comment Symbol

ここにあげたものに加えて、インテグレーションが使用する属性でユーザーが修正してはいけないものが数多くあります。ここにあげていない属性の手動での変更は、情報が失われるおそれがあり、サポートもされていません。

UML Kind

UML Kind 属性には要素のタイプが格納されます。可能なタイプは同期で使用するメタモデルによって決定されます。

一部の要素タイプ、たとえばダイアグラム内のシンボルなどについては、値 "Other" が使用されます。

UML Kind 属性の値は、モデルを Tau と同期した後は変更できません。

UML Name

要素の UML 名。この属性は情報提供の目的のみで表示され、変更できません。

UML Location

要素の所有者の名前。この属性は情報提供の目的のみで表示され、変更できません。

UML Comment Symbol

この属性はオブジェクトのオブジェクト・テキストとモデル内のコメントの同期を取るかどうかを制御するために使用します。

値が "`True`" （デフォルト）の場合は、オブジェクト・テキストはツールの間でプロパゲートされます。値が "`False`" （デフォルト）の場合は、プロパゲートされません。値はいつでも変更できます。

注記

オブジェクト・テキストを現しているコメントを Tau で削除した場合、DOORS 内のオブジェクト・テキストは削除されません。代わりに、ツールはこの属性を "`False`" に設定してプロパゲートが起こらないようにします。

リンク

このセクションでは DOORS と Tau のリンクの使い方を説明します。

- DOORS 内でリンクを使う
- Tau 内でリンクを使う
- リンク・モジュールとリンクセット

DOORS 内でリンクを使う

UML モデルを収納しているモジュールについても、DOORS における標準的なリンクの操作を行うことができます。インテグレーションは、標準的な操作にいくつかの有用なコマンドを追加します。

Tau へのドラッグアンドドロップ

DOORS 要素から UML 要素へのリンクを作成するには、要件を DOORS からドラッグして、Tau の［モデルビュー］の要素上にドロップします。

DOORS からのドロップは、ターゲットである要素が DOORS 表現を持っている場合に動作します。つまり、その要素が DOORS からインポートされている、または DOORS と同期が取られている、などの場合です。

注記

リンクが自動作成されるのは DOORS 側だけです。Tau でこのリンクを表示するには、モデルを DOORS から更新する必要があります。

Link Requirement to Selected Item in Tau

現在 Tau で選択している UML 要素から、現在 DOORS で選択しているオブジェクトへのリンクを作成します。

Open Linked UML Element in Tau

ある UML 要素に結び付けられた要件からモデルを Tau で開くのに便利な方法です。

そのオブジェクトに対して複数の UML 要素からの内向きリンクがある場合は、ダイアログが表示されて特定の相手を選択できます。

Tau 内でリンクを使う

「リンクの管理」で説明しているとおり、Tau における標準的なリンクの操作を行うことができます。さらに、Tau から DOORS へのドラッグアンドドロップでリンクを作成することもできます。

リンクは、モデルの同期を取るときに DOORS と同期されます「Tau で同期を取る」と「リンク・モジュールとリンクセット」を参照してください。

DOORS へのドラッグアンドドロップ

UML 要素から DOORS の要件へのリンクを作成するには、UML 要素を［モデルビュー］から DOORS にドラッグして、要件オブジェクト上にドロップします。この結果、UML モデルでは `<<trace>>` 依存が作成されます。

この操作はすべての要素に対して実行できます。要素が DOORS 内に表示されていなくても可能です。

注記

リンクが作成されるのは Tau 側だけです。要素が DOORS で表示されている場合は、リンクは、モデルが Tau からコミットされないと DOORS には作成されません。

リンク・モジュールとリンクセット

Tau でリンクが作成され、その後 DOORS にコミットされると、リンクは、DOORS モジュールに指定されたデフォルトのリンク・モジュールとリンクセットに作成されます。

したがって、使用したいリンク・モジュールとリンクセットを DOORS で指定することで、リンク作成に対する完全な制御ができるようになります。

デフォルトのリンク・モジュールやリンクセットが定義されていない場合は、新しいリンク・モジュールやリンクセットが自動的に作成されます。

フィルタ

要求と UML 要素の間のトレーサビリティを管理するために、数多くのフィルタが提供されています。

フィルタを適用するには、以下の作業を実行します。

- フォーマル • モジュールを開きます。
- ［Tau］、［Filter］メニューから必要なコマンドを実行します。

Identify Design Elements Not Justified by Requirements

このフィルタは UML 要素を収納しているモジュールで利用でき、要求と結び付けられていないすべての UML 要素を検出します。

Identify Design Elements by UML Kind

このフィルタは UML 要素を収納しているモジュールで利用でき、特定の種別の要素向けのフィルタを提供します。

このフィルタを実行すると、UML 種別のリストが現れ、特定の種別が選択できます。そして、モジュールにフィルタがかかり、その種別の要素のみが表示されるようになります。

Identify Requirements Not Addressed by Design Elements

このフィルタは、どの UML モデル要素とも結びついていない要件をすべて検出するために使用します。

このフィルタをかけると選択したオブジェクトの前世代も表示されます。これは、見出しの階層を維持するためと各要件にコンテキストを提供するためです。

52

トレーサビリティの管理

このセクションでは、UML モデルと要件の間のトレーサビリティを確立し、維持するために、DOORS と Tau でどのような作業を行うかについて説明します。

以下の 3 つの主要なワークフローがあります。

- Tau におけるトレーサビリティ
- DOORS におけるトレーサビリティ
- Tau と DOORS におけるトレーサビリティ

最後のワークフローは、最も高度な内容ですが、最も一般的なものでもあります。

Tau におけるトレーサビリティ

このワークフローの主な対象は、指定された要件について作業する必要はあるが、必ずしもその要件を DOORS で見る必要のないアナリスト、アーキテクト、設計者、開発者です。これらの作業者が DOORS データベースへの通常のアクセス権すらもっていないケースも考えられます。

このタイプの作業者にとって重要なことは、モデルと要件の間のトレーサビリティを確保でき、維持管理できることです。このトレーサビリティ情報は DOORS に対してプロパゲートされません。

このワークフローを確立するには、以下の手順が必要です。

- 「要求のインポート」で説明しているように要件を Tau にインポートします。

インポートが行われると、依存リンクがモデル要素とインポートされた要件の間に作成されてトレーサビリティを確立します。

要求レポート とダイアグラム・ジェネレータ（「ダイアグラムの生成」参照）を使用してトレーサビリティ分析を行います。グラフィカルなトレーサビリティ・ダイアグラムの作成には［Generate dependency view ... ］ダイアグラム・ジェネレータが有用です。

DOORS で要件が変更された場合は、「DOORS から Tau を更新」で説明している方法で Tau 側でも更新できます。

Tau で要件を変更する必要がある場合、または Tau で要件の間にリンクを張る必要がある場合は、その作業を行って、「Tau から DOORS への変更のコミット」で説明している方法で DOORS にコミットします。

以下の項目は、このワークフローではサポートされません。

- 既成の高度なトレーサビリティ分析

  Tau はグラフィカルなトレーサビリティ分析を行うには有用ですが、その一方で、DOORS のような高度なトレーサビリティ機能は備えていません。
- モデル要素間のリンクの DOORS 表記

  このリンクは DOORS 内での表記をもたないので、DOORS の標準的なリンクとしてインポートする方法がありません（DOORS で外部リンクを使って要件からモデル要素へのリンクを作成する方法もありますが、インテグレーション機能はこういったリンクを作成しません）。
- UML モデルの DOORS 表記

DOORS におけるトレーサビリティ

このワークフローの主な対象は、指定された DOORS 要件について作業する必要があり、その要件が Tau の UML モデルに正しく反映されていることを検証する必要のある要件エンジニア、アナリスト、アーキテクトです。これらの作業者は、必ずしもモデルの作業を Tau で行う必要はありません。Tau モデルへの通常のアクセス権すらもっていないケースも考えられます。

このタイプの作業者にとって重要なことは、すべての要件が UML モデルに正しく反映されていることを DOORS リンクとトレーサビリティ分析で検証できることです。

このワークフローを確立するには、以下の手順が必要です。

- 「UML モデルを同期用に設定する」で説明している方法で Tau モデルと DOORS を同期します。

同期処理が行われると、DOORS リンクが要件と代理モデル要素の間に作成されて、トレーサビリティを確立します。インポートされたモデル要素間のリンクは DOORS で表示されることに注意してください。

トレーサビリティ分析には、DOORS のトレーサビリティ・ツールを使用できます。

Tau でモデルが変更された場合は、「モデルの変更を DOORS にコミットする .」で説明しているとおり［DOORS にコミット］コマンドで DOORS にコミットされます。

以下の項目はこのワークフローではサポートされません。

- UML ダイアグラムを使った要件のグラフィカルな可視化とトレーサビリティ分析。

  要件とリンクを Tau で表示できないため、ダイアグラム・ジェネレータは使用できません。
- DOORS での UML モデルに対する変更

  UML モデルは DOORS では変更できません。この作業には Tau と DOORS の連携が必要です。

Tau と DOORS におけるトレーサビリティ

このワークフローは、前のセクション、「Tau におけるトレーサビリティ」および「DOORS におけるトレーサビリティ」で説明した 2 つのワークフローの組み合わせです。このワークフローの主な対象は、要件と UML モデルの両方について作業する必要があり、Tau と DOORS の両ツールを定常的に使用するアーキテクトと設計者です。

このタイプのユーザーは Tau と DOORS の両方でトレーサビリティ分析を実行する必要があり、両ツールの方向に情報を同期する必要があります。

このワークフローを確立するには、以下の手順が必要です。

- 「要求のインポート」で説明している方法で、要件を Tau にインポートします。
- 「UML モデルを同期用に設定する」で説明している方法で、Tau モデルを DOORS と同期します。

これらが行われると、変更が Tau または DOORS のいずれで行われてももう一方のツールに同期できるようになります。

注記

同期前の変更は一方のツールでのみ行ってください。両方のツールから変更を行うと、同期中に一方が削除されます。

Tau での作業

依存リンクがモデル要素とインポートされた DOORS 要件の間で作成されてトレーサビリティが確保されます。

要求レポート とダイアグラム・ジェネレータ（「ダイアグラムの生成」参照）を使用してトレーサビリティ分析を行います。グラフィカルなトレーサビリティ・ダイアグラムの作成には［Generate dependency view ... ］ダイアグラム・ジェネレータが有用です。

DOORS で要件が変更された場合は、「DOORS から Tau を更新」で説明している方法で Tau 側でも更新できます。

Tau で要件を変更する必要がある場合、または Tau で要件の間にリンクを張る必要がある場合は、その作業を行って、「Tau から DOORS への変更のコミット」で説明している方法で DOORS にコミットします。

DOORS での作業

同期処理が行われると、DOORS リンクが要件と代理モデル要素の間に作成されて、トレーサビリティを確立します。インポートされたモデル要素間のリンクは DOORS で表示されることに注意してください。

トレーサビリティ分析には、DOORS のトレーサビリティ・ツールを使用できます。

Tau でモデルが変更された場合は、「モデルの変更を DOORS にコミットする .」で説明しているとおり［DOORS にコミット］コマンドで DOORS にコミットされます。

Tau と **DOORS** の間の整合性の維持

シンプルな手順を使って、要件とモデルを最新に維持し両者の整合性を確保できます。

重要！

意図しない結果になることがあるため、同期処理をせずに同じ情報を両方のツールで変更しないでください。

たとえば、DOORS であるリンクが削除されたが Tau では削除されていない場合、Tau のモデルが DOOR に対してコミットされると DOORS 側には再び挿入されることになります。

DOORS で行われた変更を Tau と同期するには、以下の手順を行います。

- ［モデルビュー］タブで、すべてのインポートされたフォーマル・モジュールと同期させる UML 要素について［DOORS から更新］を実行します。

Tau で行われた変更を DOORS と同期するには、以下の手順を行います。

- ［モデルビュー］タブで、すべてのインポートされたフォーマル・モジュールと同期させる UML 要素について［DOORS にコミット］を実行します。

System Architect を使用した UML モデリング

本章では、Tau と System Architect を使用した UML モデリングの方法について説明します。

49

Tau と System Architect の協調

この章では Tau と System Architect の両方を使って UML モデリングを行う方法を説明します。

System Architect は、UML2 など数多くの異なる表記法をサポートするモデリングツールです。System Architect の特徴のうち最も重要なのは、ツールがリポジトリベースであることです。リポジトリは「エンサイクロペディア」と呼ばれ、サーバー上に格納して、ブラウザベースのインターフェイスや標準の System Architect クライアントアプリケーションからネットワーク経由でアクセスできます。

System Architect で構築した UML 2 モデルは Tau で構築したモデルと完全互換です。これは、System Architect で UML 2 モデルを作成してサーバー上のエンサイクロペディアに保存し、その後 Tau でそのモデルを開いて Tau から追加修正を行ってエンサイクロペディアに保存できること意味します。

Tau で作成したモデルをローカルのファイルシステムに保存して、そのモデルデータを System Architect にエンサイクロペディアに移動して System Architect で開くことができます。また、System Architect で作成したモデルをローカルのファイルシステムにコピーすることも可能です。

この作業形態を実現する Tau の重要な機能は、System Architect エンサイクロペディアと Tau のプロジェクトを関連付ける機能です。エンサイクロペディアがプロジェクトと関連付けられると、エンサイクロペディア内のすべての UML2 要素は、あたかも Tau で作成した要素のように Tau プロジェクトの一部になります。つまり次のような作業の流れになります。エンサイクロペディアと関連付けられた Tau プロジェクトが Tau にロードされると、System Architect が自動的に起動して、関連付けられたエンサイクロペディアが開きます。System Architect がすでに実行中の場合は、その実行中の System Architect に、対応するエンサイクロペディアがロードされて開きます。

エンサイクロペディアを UML モデルと関連付ける

System Architect インテグレーションは Tau のアドインとして実装されています。したがって、インテグレーション機能を実行するにはまずアドインを有効にする必要があります。アドインを有効にするには以下の手順にしたがいます。

1. ［ツール］>［カスタマイズ］を選択します。
2. ［アドイン］タブを選択します。
3. 表示されているアドインの中から［*System Architect Integration*］をチェックします。
4. ダイアログを閉じます。

1 つのエンサイクロペディアは、［SA エンサイクロペディアと関連付け］コマンドを使用して 1 つの UML プロジェクトと関連付けられます。このコマンドは Tau の［モデルビュー］内の UML モデルのルートを表しているすべてのノード上で使うことができます。［Standard View］を使っている場合（通常の場合）は、このノードは［Model］です。

このコマンドの実行時に System Architect が手動で起動されていて、すでにエンサイクロペディアがロードされている場合は、このエンサイクロペディアが Tau のモデルにロードされます。

System Architect が起動していない場合は、このコマンド実行時に起動されます。ユーザーは手動で作業対象のエンサイクロペディアをロードする必要があります。

System Architect が起動しているがエンサイクロペディアが開いていない場合、エラーメッセージが表示されて、エンサイクロペディアを開くようにユーザーに要求します。

1 つのエンサイクロペディアは 1 つの Tau プロジェクトとだけ関連付けることができます。あるエンサイクロペディアがすでにあるプロジェクトと関連付けられている場合は、ダイアログのメニューでの選択肢に入りません。

エンサイクロペディアに格納された UML 要素の取り扱いは、ローカルのファイルに保存された要素とはさまざまな点で異なります。最も大きな違いは、エンサイクロペディアに格納された要素はプロジェクトを開いただけではロードされず、必要に応じてロードされることです。詳細については 1563 ページの「要素のインクリメンタルロード」を参照してください。

もう 1 つの違いは、複数のユーザーが同時に同じ要素をアクセスすることを防ぐ特別の機構が用意されていることです。この機能については、1564 ページの「UML 要素を作成、編集、保存する」の説明を参照してください。

エンサイクロペディアとの関連付けの解除方法は、1562 ページの「エンサイクロペディアとの関連付けを解除する」で説明しています。

新しい UML プロジェクトを作成したときに、そのプロジェクトを自動的に System Architect エンサイクロペディアに関連付けることができます。System Architect ストレージと 新しいウィザードを参照してください。

エンサイクロペディアとの関連付けを解除する

［SA エンサイクロペディアから切り離し］コマンドを使うと、エンサイクロペディアとの関連付けを解除できます。このコマンドは Tau の［モデルビュー］内の UML モデルのルートを表しているノード上で使うことができます。［Standard View］を使っている場合（通常の場合）は、このノードは［Model］です。

このコマンドを選択すると、関連付けは解除され、エンサイクロペディアに格納されていたすべてのモデル要素はモデルから削除されます。

ただし、モデル要素はエンサイクロペディアからは削除されません。したがってエンサイクロペディアとの関連付けを再び行うと、これらの要素は再表示されます。

要素のインクリメンタルロード

エンサイクロペディアを Tau プロジェクトと関連付けた場合も、すでに関連付けされている Tau プロジェクトを開く場合も、エンサイクロペディアに格納されたモデル要素は、そのタイミングではロードされません。これはパフォーマンスの観点からの仕様です。

要素がロードされるのは、以下の 3 つの状況においてのみです :

- ［モデルビュー］でその要素を含んでいる要素のノードが展開されたとき
- エディタでダイアグラムを開いたとき
- ［すべての子要素をロード］コマンドを実行したとき

［モデルビュー］で展開した場合は、展開されたノードの直下の子ノードに対応する要素のみがロードされます。

エディタでダイアグラムを開いた場合は、ダイアグラム上のシンボルとして表現されている要素がすべてロードされます。

［すべての子要素をロード］コマンドは、System Architect エンサイクロペディアに格納されている要素を表現している、［モデルビュー］内のノード上で使うことができます。このコマンドを実行した結果、すべての子要素、つまり、選択した要素内に直接含まれる要素や間接的に含まれる中間要素がロードされます。

いったん要素がロードされると、［モデルビュー］ノードが折りたたまれたり、エディタを閉じてもロードされたままになります。

このインクリメンタルなロード方式を採用しているため、通常はすべての要素はツールにロードされていません。つまり、モデル要素の一部がロードされていないことに起因して、チェッカーやレポート生成ツール、コード生成ツールなどの「意味」ツールがエラーメッセージを出力することがあります。したがって、これらの意味ツールを使用する場合は、［すべての子要素をロード］コマンドを使用してモデル要素の全体をロードしておくことを推奨します。

UML 要素を作成、編集、保存する

System Architect エンサイクロペディアは複数ユーザーから同時にアクセスできるので、モデルの修正に際してエンサイクロペディアの整合性を維持するために、要素のロックに基づく機構が提供されています。

Tau クライアントアプリケーションでの UML モデルの修正に基づいてエンサイクロペディアを更新する際に適用される方針は以下のとおりです：

- Tau で作成した要素は、すべて即時にエンサイクロペディアにも作成される。
- Tau のグラフィカルエディタ、プロパティページ、モデルビューを使用したモデル要素の修正は、すべて即時にエンサイクロペディアにプロパゲート（伝播）される。
- Tau を使って他の方法（たとえば自動スクリプトなど）で修正したすべての要素は、Tau でモデルを保存したときに、エンサイクロペディアでも修正される。ただし、他のユーザーによって要素がロックされている場合は、修正は失敗する。
- Tau で削除されたすべての要素は、Tau でモデルを保存したときに、エンサイクロペディアから削除される。
- ダイアグラムは Tau で開かれるとロックされ、閉じるとロック解除されます。ダイアグラム内のシンボルのグラフィックプロパティの変更は、参照されるモデル要素をロックせず変更もしません。
- ダイアグラム内のテキスト編集時にテキストモードに入ったとき、そのテキストは、まずエンサイクロペディアの情報で更新され、その後参照されるモデル要素はロックされます。
- ダイアグラム内のテキスト編集時にテキストモードから出る（終了する）とき、エンサイクロペディアはその変更で更新され、参照されるモデル要素のロックが解除されます。

上の方針と他のユーザーがエンサイクロペディアをアクセスし得るという観点から、以下のような状況とその回避方法が導かれます：

- 他のユーザーがロックしている要素が修正したモデルの一部である場合にはその部分を保管できない。これを回避する方法の 1 つは、保存を後で再試行することです。
- エンサイクロペディア内の要素で、Tau にロードされているものに対する変更は、即座には Tau に伝播されない。この結果、他のユーザーが行った変更を Tau から上書きする可能性があります。

System Architect ストレージと 新しいウィザード

新規 UML プロジェクトを作成するためのウィザードに、便利なオプション［SA エンサイクロペディアと関連付ける］があります。このオプションが選択されていると、現在開いている System Architect エンサイクロペディアが、新しく作成したプロジェクトに関連付けられ、エンサイクロペディアの要素が Tau にロードされます。System Architect インテグレーションアドインも自動的に有効になります。

System Architect が起動していない場合は、ここで起動されて、適切なエンサイクロペディアを選ぶことができます。

ルート要素に対するエンサイクロペディアストレージの指定

［SA エンサイクロペディアに保存］コマンドを使用して、モデルのルート要素を、ローカルファイルではなく、エンサイクロペディアに格納できます。

注記

エンサイクロペディアに格納するように指定できるのは、モデルのルート要素のみです。

注記

［新しいファイルで保存］コマンドなどを使って、エンサイクロペディアの要素をローカルのファイルシステムに移動することはできません。要素をエンサイクロペディアからローカルファイルに移すには、［コピー / 貼り付け］を使ってください。

SystemArchitect から Tau への情報の移動

System Architect から Tau に向けて情報を移動する必要が生じることがあります。たとえば、System Architect エンサイクロペディアにある分析モデルを Tau に移して実装のための基盤とするようなケースなどです。

こういった作業フローを実現するには、Tau のコピー / 貼り付け機能を使用するのが便利です。

1. System Architect に格納されていて、エンサイクロペディアから移動したい要素を選択します。
2. ［すべての子要素をロード］コマンドを使って、すべての要素が Tau で使用できることを確認します。
3. 要素をコピーします。
4. 要素を移動したい場所に貼り付けます。

System Architect の分析モデルを Tau で実装することを計画している場合、実装モデルから分析モデルへのトレーサビリティを維持することが有用です。これを行うには、Tau で用意されている自動トレーサビリティ作成機能が有効です。以下の手順で行います。

1. System Architect に格納されていて、実装の基礎としたい要素を選択します。
2. ［すべての子要素をロード］コマンドを使って、すべての要素が Tau で使用できることを確認します。
3. マウスの右ボタンを使って要素を元の場所からコピー先の場所へドラッグします。
4. ポップアップメニューから［トレーサビリティを含むコピー］を選択します。

この操作の結果、対応する元の要素への <<trace>> 依存を含むすべての定義をもつコピーが作成されます。

トレービリティを含むコピーの項も参照してください。

既知の制約事項

名前なしの要素

Tau の要素には名前がないものがあります。System Architect エンサイクロペディアでは、名前のない要素は許されません。名前のない要素をサポートするため、System Architect エンサイクロペディアでは、こういった要素に「メタクラス名と番号」に基づいた特殊な名前をつけて扱います。名前のない要素がエンサイクロペディアに格納されると、System Architect エクスプローラやプロパティページでは、要素のメタクラスに基づいた名前を持つかのように表示されます。

アンドゥ / リドゥ

System Architect エンサイクロペディアに格納された要素については、アンドゥ（動作の取り消し）やリドゥ（動作の再実行）は機能しません。

エンサイクロペディアのモデルルート要素での制約

エンサイクロペディアではパッケージのみがルート要素（モデル階層中の最上位要素）としてサポートされます。他のルート要素を作成しても、エンサイクロペディアに関連付けられたプロジェクトを Tau で開いたときにはロードされません。ただし、要素は System Architect では表示されます。したがって、Tau でも表示したい場合はこの要素を System Architect 上でパッケージに移動してください。

プロファイルとモデルライブラリ

System Architect で使用できるプロファイルとモデルライブラリは、定義済みパッケージと TtdPredefinedStereotypes です。つまり、あるモデルを System Architect で開いた場合、これら以外のプロファイルで定義されているステレオタイプは使用できないということです。こういったプロファイルに定義されているステレオタイプをモデルで使用すると、System Architect でこのモデルを開いた場合、ダイアグラムにはこのステレオタイプインスタンスが表示されません。

ダイアグラムを **Tau** と **System Architect** の両方からアクセスする

1 つのダイアグラムを Tau と System Architect の両方から同時に開くことができます。注意が必要なのは、一人のユーザーが同一のマシンでダイアグラムを 2 回開くと、ダイアグラムがロックされず、アクセスできることです。つまり、両方のツールのエディタから同時に更新がかけられることになります。ダイアグラムの保存時に両方からの更新のマージは行われません。したがって、後から保存したデータでダイアグラムは上書きされます。

一時点では 1 つのエディタからのみダイアグラムを更新すること、および頻繁に更新することを推奨します。

UML と Web サービス

本章では、Web サービス記述言語 (WSDL) を使用して UML 上で Web サービスをモデリングする方法と、Web サービス記述ファイルのインポートとエクスポートをする方法を説明してます。

この機能でサポートするのは、WSDL バージョン 1.1 です。

モデルベリファイヤを使用した Web サービスのシミュレーションもできます。Web サービス シミュレーションの項を参照してください。

50

Web サービスサポート

本章では、Tau における Web サービスのモデリング方法を説明します。説明には、UML における Web サービスのモデリング方法と、その UML モデルから WSDL を生成する方法が含まれます。既存の WSDL ドキュメントを UML モデルにインポートすることも可能です。

UML での Web サービスモデリング

Web サービスを二段階の抽象化レベルで UML モデルで表現できます :

1. **WSDL centric representation -** このモデルでは、WSDL 構築子はステレオタイプ化された UML エンティティで表現されます。WSDL 構築子を表現するのに必要なすべてのステレオタイプを含む WSDL プロファイル が提供されます。
2. **UML centric representation -** このモデルでは、WSDL コードジェネレータによって WSDL 構築子に翻訳される標準的な UML 構築子が使用されます。WSDL 固有のステレオタイプは、対応する UML 構築子のない WSDL 構築子用にのみ必要です。

どちらの手法を使うかは、モデリングの目的によって決まります。UML 用語よりも WSDL 用語に慣れた Web サービスモデラーには、WSDL centric representation が最も適切です。一方、Web サービスをモデルに指定するだけではなくモデルからのコード生成（Java、C# など）で実装する場合には UML centric representation が適しているでしょう。UML centric representation は、通常は、WSDL centric representation と比べると表現がコンパクトになります。

注記

モデルからの WSDL ドキュメントの生成には、これらの 2 つの手法を使用できますが、現在の Tau は、WSDL ファイルのモデルへのインポートには WSDL centric representation のみをサポートします。Web サービスシミュレーションを使用して既存の Web サービスを UML centric representation で表現することはできます。詳細は、Web サービス シミュレーション を参照してください。

WSDL プロジェクトの作成

モデルでの表現を、WSDL または UML centric representation のどちらにするかにかかわらず、WSDL プロジェクトを作成するところから作業は始まります :

1. ［ファイル］>［新規］を選択します。
2. ［UML（WSDL モデリング用)］で新しいプロジェクトを作成します。WSDL シミュレーションのため既存の WSDL ドキュメントをインポートする場合は、［Web サービスシミュレーションをサポートする］がチェックされていることを確認してください。Web サービスのシミュレーションについては、Web サービス シミュレーション を参照してください。

WSDL サポートを既存のプロジェクトに対して有効化することもできます :

1. ［ツール］メニューから［カスタマイズ］ダイアログを選択します。
2. ［アドイン］タブをクリックして、［WSDL］アドインをチェックします。WSDL とともに XSD を使用する場合は、［XSDFramework］もチェックします。
3. ［閉じる］をクリックします。

WSDL アドイン

アドインの主要機能は以下のとおりです :

- UML モデルに WSDL 情報を追記するための WSDL プロファイル
- WSDL ビュー（WSDL セントリックモデルビュー）
- UML モデルから WSDL ドキュメントを生成する WSDL ジェネレータ。コードジェネレータの入力となる UML モデルは、WSDL centric representation でも UML centric representation でもかまいません。この 2 つの表記それぞれに合わせたジェネレータがあります。
- Web サービスの UML モデルを生成できる WSDL インポータ。インポートされた結果は、Web サービスを WSDL centric representation で表現したモデルになります。

注記

WSDL アドインをアクティブにすると、WSDL プロファイルとビューがロードされます。また、XSD プロファイルとビューもロードされます。これは、WSDL 定義に埋め込まれた XML スキーマをサポートするためです。

WSDL ビュー

WSDL ビューは、モデルの WSDL 中心の外観を提供します。このビューでは、WSDL 要素のみが表示され、WSDL 要素の作成のみが可能です。このビューでは、コンテキストメニューの［新規モデル要素］コマンドを使用して、適切な WSDL 構造が作成できるようになっています。

WSDL ビューは、Tau で Web サービスの WSDL centric representation を使って作業するために用意されています。

WSDL ビューを有効にするには :

1. ［表示］メニューで、［モデルビューの再構成 ...］を選択します。
2. ［WSDL View］を選択して、［OK］をクリックします。

［Standard View］と［WSDL View］の切り替えはいつでも行えます。

注記

［WSDL ビュー］ではすべての要素が表示されるわけではありません。すべての要素を表示するには［Standard View］に切り替えてください。

参照

6 ページの「モデル ビュー」
2158 ページの「デフォルトのモデル ビュー」

WSDL プロファイル

WSDL プロファイルは、WSDL で定義された Web サービスのモデリングという点で UML を拡張したものです。プロファイルには、WSDL 固有のステレオタイプがあり、主に WSDL 中心の Web サービスモデリングで使用します。

WSDL プロファイルパッケージは、'wsdl' と呼ばれ、モデルの［Libraries］セクションにあります。

WSDL プロファイルの詳細については、WSDL プロファイルの内容を参照してください。

WSDL の生成

Tau Web サービスモデルは、WSDL ジェネレータを使って WSDL ファイルに格納できるように変換されます。WSDL 生成がどのように行われるかについては、モデルが WSDL centric representation を使っているか、UML centric representation を使っているかに依存します。

注記

モデルが複数の Web サービスを含んでいる場合、一部のサービスを WSDL centric representation で表現して、他のサービスを UML で表現できます。ただし、1 つのサービスについて 2 つの表記を混在することは推奨されません。

WSDL Centric モデルからの WSDL 生成

WSDL centric なモデルから WSDL を生成するには以下の手順にしたがいます :

1. In the context menu of a WSDL package, select **Generate WSDL...**
2. Select a folder where to place the generated WSDL file. Click **Save**.

Step 2 is only necessary the first time you generate WSDL for the package. The path to the generated WSDL file is stored in a WSDL file artifact that will be generated next to the WSDL package. Use the Properties Editor if you want to change this path later. You can also delete the WSDL file artifact and invoke the **Generate WSDL...** command again in order to generate the WSDL file to another location.

The rules for transating a WSDL centric model to WSDL files are the reverse of the translation rules described in WSDL/XSD インポータリファレンス . Note that UML entities must be annotated by the stereotypes described in these translation rules in order to obtain the expected WSDL.

UML Centric モデルからの WSDL 生成

UML centric なモデルから WSDL を生成するには以下の手順にしたがいます :

1. In the Model View select an interface or a package that contains interfaces.
2. In the context menu select **WSDL Generator->New Artifact**. This will create a new WSDL build artifact manifesting the selected element.
3. In the context menu of the WSDL build artifact select **Build (WSDL Generator)->Generate**.

Step 1 and 2 is only necessary the first time you generate WSDL for the package or interface in order to obtain a WSDL build artifact. Regeneration of the WSDL file is made by performing step 3 again.

The rules for translating a UML centric model to WSDL files are described inWSDL コードジェネレータリファレンス

WSDL のインポート

The WSDL/XSD Importer is used for importing WSDL definitions (and/or XSD) into a Tau model. After the import the web services described by imported WSDL files may be further developed in Tau, and then generated to WSDL files again (see WSDL Centric モデルからの WSDL 生成).

The initial import is done using the WSDL/XSD インポートウィザード and re-import can later be perfomed by using the context menus on WSDL and XSD file artifacts.

WSDL/XSD インポートウィザード

WSDL 定義または XSD スキーマを Tau にインポートするには :

1. パッケージまたは「モデル」のノードを選択します。
2. ［ファイル］>［インポート］を選択して、［インポート ウィザード］を起動します。
3. ［**Import WSDL/XSD**］を選択して、［OK］をクリックします。

WSDL/XSD インポートウィザードの第一ステップ

インポート処理の第一ステップは、インポートする XML ファイルの種別を指定することと、ダイアグラム生成を行うかどうかを指定することです。

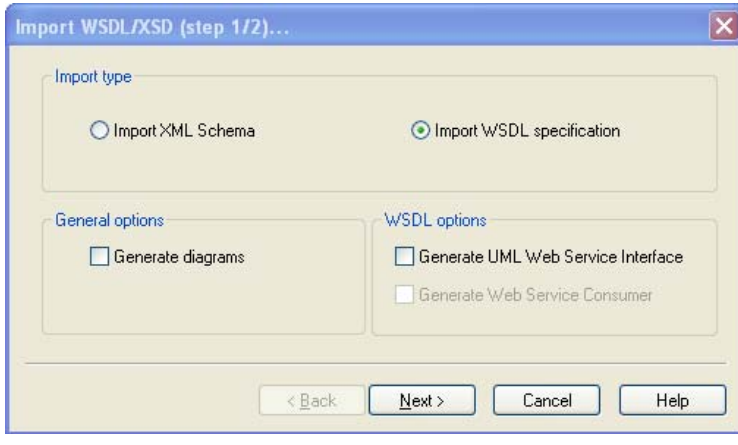


図 269: WSDL/XSD インポートウィザードの最初のステップ

XSD ファイルをインポートする場合は、[**XML スキーマのインポート**] を選択し、WSDL ファイルをインポートする場合は、[**WSDL 仕様のインポート**] を選択します。

インポートした要素をダイアグラム上で表示するには、[**ダイアグラムを生成**] オプションを選択します。

If you decide to import WSDL you can also set some importer options for generating additional information into the model. These options are used when simulating web services; see UML からの Web サービスの呼び出し for more information about these options.

第一ステップを完了するには、[**次へ**] を選択します。

WSDL/XSD インポートウィザードの第二ステップ

The purpose of the second step is to determine the files to be imported. The set of files is listed in the the dialog and can be changed by means of the buttons **Add Files...**, **Remove** and **Clear**.

- Pressing the **Add Files…** button invokes the standard dialog for opening files. Multiple selection is allowed. The files selected in the dialog are added to the file list.
- Pressing the **Remove** button removes the files currently selected from the list.
- Pressing the **Clear** button removes all entries from the list.

It is also possible to import from a URL rather than a file. To do this press the **Add URL** button and enter the URL where to get the WSDL/XSD file from.

注記

If you access the web through a proxy server you may need to set appropriate Proxy settingsto be able to import from an URL

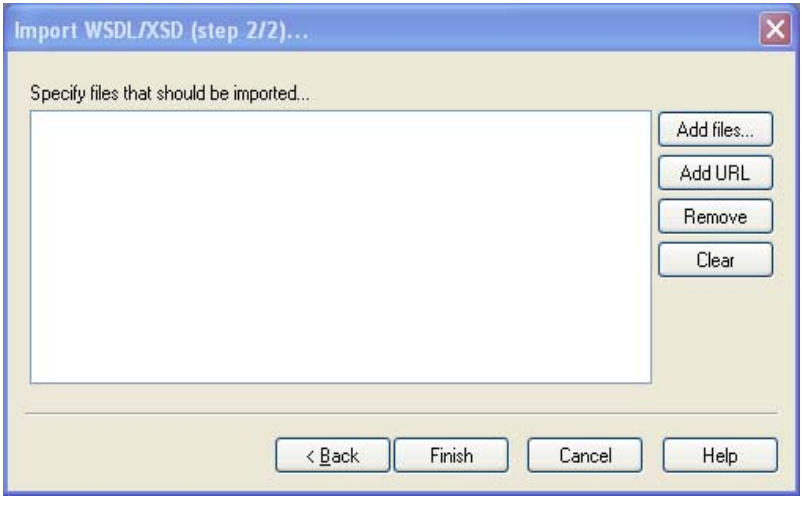


図 270: The second step of the WSDL/XSD Import Wizard

Pressing **Finish** will import the selected files and/or URLs.

インポートの結果

If the **Model** node in the Model View was selected before the import, the importer will create one u2 file per imported WSDL/XSD file, add it to the project and store all imported definitions from that WSDL/XSD file into that u2 file. If a package was selected before the import, the imported definitions will be created inside that selected package and stored in the same u2 file as the package.

For each file that was added in the import wizard, there will be one file artifact generated. For each WSDL file there will be an artifact with the stereotype <<WSDL file>> and for each XSD file there will be an artifact with the stereotype <<XSD file>>. These stereotypes have a tagged value set to point to the corresponding file in the file system.

For each file there will also be one package generated. It contains the imported WSDL or XSD definitions. Each artifact has a <<manifest>> dependency to one package.

再インポート

Once import of WSDL or XSD has taken place it is possible to re-import the content of the files. This is done by right clicking a WSDL or XSD artifact and select **Import WSDL...** or **Import XSD...** respectively. This command will remove all imported elements and redo the import.

注記

Re-importing WSDL or XSD will remove all the content in the imported package, so diagrams or other UML entities that has been added after the first import will be deleted.

51

WSDL コードジェネレータリファレンス

この章は、注釈なし UML モデル（UML セントリック Web サービス表記）を WSDL に翻訳する WSDL ジェネレータのリファレンスガイドです。UML 構築子から WSDL 構築子への変換について説明します。

概要

The WSDL generator implements a mapping of UML constructs to WSDL constructs. The translation rules have been designed to use standard UML constructs, without stereotype annotations, to an as large extent as possible. Only when a certain WSDL construct has no corresponding UML construct it is necessary to apply stereotypes to the UML elements.

WSDL ビルドアーティファクト

The WSDL generator is integrated with the Application Builder. To define which parts of the model to translate to WSDL, and to specify 翻訳オプション , a WSDL build artifact is used.

The WSDL build artifact is recognized by its applied `<<WSDL Generator>>` stereotype. See UML Centric モデルからの WSDL 生成 to learn how a WSDL build artifact can be created in the model.

WSDL is generated by performing the **Generate** command on the build artifact. Which WSDL files that are then generated depends on the kind of element that is manifested by the build artifact:

1. **An interface.** The WSDL file artifact that manifests the interface will be generated.
2. **A package.** All WSDL file artifacts manifesting interfaces contained in the package will be generated.

デフォルトのモデル - ファイルマッピング

If no WSDL files are found when performing the **Generate** command on the build artifact, the WSDL generator will create WSDL file artifacts according to a default model-to-file mapping. For each manifested interface (direct manifestation, or indirect by the manifestation of a containing package) a WSDL file artifact will be created manifesting the interface. The name of the file artifact is the same as the name of the interface.

The WSDL generator also supplies an explicit command **Generate File Mapping** which can be used to generate file artifacts according to the default model-to-file mapping rules, without actually generating the WSDL files. This can be useful if additional information needs to be added to the file artifacts prior to WSDL generation, for example `xsd:import` or `xsd:include` dependencies (see 依存).

文書構造

The following chapters describe the subset of UML that can be translated to WSDL by the WSDL generator. UML constructs not mentioned here are not supported, and will be ignored during translation.

For each supported UML construct a translation rule is given. If there are exceptions to the rule, these are also mentioned.

For most translation rules examples are given using textual UML and WSDL syntax.

注記

The purpose of each example is only to illustrate the translation rule at hand, not to give a precise description of how the generated WSDL will look like. Also note that examples for brevity reasons typically are fragmental. Omitted sections of UML or WSDL are marked with triple dots (...).

インターフェイス

A UML interface is translated to a `<service>` element and a `<portType>` element within the WSDL file.

The name of the WSDL service and the port type is the name of the UML interface. The same name is also used for the containing `<definitions>` element.

例 **587: Translation of interfaces**

UML

```
interface MyWebService {}
```

WSDL

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions name="MyWebService" ...>
  <wsdl:portType name="MyWebService">
  </wsdl:portType>
  <wsdl:service name="MyWebService">
  </wsdl:service>
</wsdl:definitions>
```

コメント

A UML comment attached to an element that has a specified mapping to WSDL is translated to a `<documentation>` tag on the resulting WSDL element.

If the UML element has multiple representations in the WSDL file (as is the case for an インターフェイス for example) the `<documentation>` tag is only placed on one of them (for an interface, it is placed on the `<service>` element).

例 **588: Translation of comments**

UML

```
interface MyWebService comment "My first webservice" {}
```

WSDL

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions name="MyWebService" ...>
  <wsdl:portType name="MyWebService">
  </wsdl:portType>
  <wsdl:service name="MyWebService">
```

```
    <wsdl:documentation>
My first webservice
    </wsdl:documentation>
  </wsdl:service>
</wsdl:definitions>
```

依存

An <<xsd::import>> or <<xsd::include>> dependency from a WSDL file artifact to another file artifact or package is translated to an <import> specification in the WSDL file.

If a 'namespace' tagged value is specified in the <<xsd::import>> stereotype instance it is mapped to a corresponding 'namespace' value of the <import> element.

This use of dependencies provides a means to include other files (typically WSDL or XSD files) into the generated WSDL file.

例 **589: Translation of dependencies from WSDL file artifacts**

UML

```
<<wsdlFile(.path = "MyInterface.wsdl".)>> artifact
'MyInterface.wsdl'
<<xsd::import(.namespace = "ns".)>> dependency to Artifact2
{}

<<xsdFile(.path = "C:\\x.xsd".)>> artifact Artifact2 {}
```

WSDL

```
<wsdl:types>
  <xsd:schema ...>
    <xsd:import namespace="ns" location="C:\x.xsd"/>
  </xsd:schema>
</wsdl:types>
```

Note that since the imported file is an XSD file the generated <xsd:import> will be placed in the <xsd:schema> tag.

操作

A UML operation contained in an interface is translated to a WSDL <operation> within the <portType> element of the WSDL document.

The name of the WSDL operation is the name of the UML operation.

See 例 590 on page 1585 for an example.

パラメータ

Each UML operation is also mapped to one or two WSDL <message> elements in the WSDL document.

The details of this mapping depend on the kinds of parameters the operation has.

In case the UML operation has no parameters that can carry data back to the caller (that is no 'return', 'in/out' or 'out' parameters) there will just be one WSDL message generated. Otherwise there will be two WSDL messages generated. The name of the first WSDL message is the name of the UML operation with the suffix “Request” appended. The name of the second WSDL message is the name of the UML operation with the suffix “Response” appended.

In each message a WSDL <part> declaration is made for each parameter that carries data in the direction represented by the message. That is, in the first message (“Request”) there will be one WSDL part for each 'in' and 'in/out' parameter of the UML operation. In the second message (“Response”) there will be one WSDL part for each 'in/out', 'out' or 'return' parameter of the UML operation.

The name of each WSDL part is the name of the corresponding UML parameter. If the UML parameter has no name, the WSDL part gets the name “par_<index>”, where <index> is an index to make the part name unique. For an unnamed 'return' parameter the name “result” is used instead.

If an auto generated part name (“result” or “par_<index>”) conflicts with the name of another part in the message, the index is incremented by one until a unique name is obtained (“result_<index>” or “par_<index+1>”).

Each WSDL part also gets a type (or element) reference which is the translation of the type of the UML parameter. See 型 for more information.

例 **590: Translation of operations with and without parameters**

UML

```
interface MyWebService {
  void Do();
  void DoParam( Boolean);
  Charstring GetOne();
  Integer GetTwo(out Charstring p1);
  Integer GetInOut(inout Charstring result);
}
```

WSDL

```
...
<wsdl:message name="DoRequest"/>
<wsdl:message name="DoParamRequest">
  <wsdl:part name="par_0" type="xsd:boolean"/>
</wsdl:message>
<wsdl:message name="GetOneRequest"/>
<wsdl:message name="GetOneResponse">
  <wsdl:part name="par_0" type="xsd:string"/>
</wsdl:message>
<wsdl:message name="GetTwoRequest"/>
<wsdl:message name="GetTwoResponse">
  <wsdl:part name="p1" type="xsd:string"/>
```

```
  <wsdl:part name="par_0" type="xsd:integer"/>
</wsdl:message>
<wsdl:message name="GetInOutRequest">
  <wsdl:part name="result" type="xsd:string"/>
</wsdl:message>
<wsdl:message name="GetInOutResponse">
  <wsdl:part name="result" type="xsd:string"/>
  <wsdl:part name="par_0" type="xsd:integer"/>
</wsdl:message>
<wsdl:portType name="MyWebService">
  <wsdl:operation name="Do">
    <wsdl:input message="tns:DoRequest"/>
  </wsdl:operation>
  <wsdl:operation name="DoParam">
    <wsdl:input message="tns:DoParamRequest"/>
  </wsdl:operation>
  <wsdl:operation name="GetOne">
    <wsdl:input message="tns:GetOneRequest"/>
    <wsdl:output message="tns:GetOneResponse"/>
  </wsdl:operation>
  <wsdl:operation name="GetTwo">
    <wsdl:input message="tns:GetTwoRequest"/>
    <wsdl:output message="tns:GetTwoResponse"/>
  </wsdl:operation>
  <wsdl:operation name="GetInOut">
    <wsdl:input message="tns:GetInOutRequest"/>
    <wsdl:output message="tns:GetInOutResponse"/>
  </wsdl:operation>
</wsdl:portType>
...
```

オーバーロードされた操作

To be able to uniquely identify an operation from a set of overloaded operations (which all have the same name) the `<input>` and `<output>` elements of such a WSDL operation will contain a 'name' attribute. The value of this attribute is composed by the name of the operation followed by an index.

WSDL messages generated for overloaded operations also get the same index as suffix to make the message names unique.

例 **591: Translation of overloaded operations**

UML

```
interface MyWebService {
  void foo(Integer);
  void foo(Boolean);
}
```

WSDL

```
<wsdl:message name="foo_1Request">
  <wsdl:part name="par_1" type="xsd:integer"/>
</wsdl:message>
<wsdl:message name="foo_2Request">
```

```
  <wsdl:part name="par_1" type="xsd:boolean"/>
</wsdl:message>
<wsdl:portType name="MyWebService">
  <wsdl:operation name="foo">
    <wsdl:input name="foo_1" message="tns:foo_1Request"/>
  </wsdl:operation>
  <wsdl:operation name="foo">
    <wsdl:input name="foo_2" message="tns:foo_2Request"/>
  </wsdl:operation>
</wsdl:portType>
```

シグナル

A UML signal contained in an interface is translated to a WSDL <operation> within the <portType> element of the WSDL document.

The name of the WSDL operation is the name of the UML signal.

When the signal only contains 'in' parameters (which is the normal case) it is thus an alternative to using an operation for modeling a one-way WSDL operation. However, it can also be used to model a WSDL request-response operation if it is given 'out' or 'inout' parameters. A signal may not have a return parameter.

The translation rules for signal parameters are the same as for operation parameters (see パラメータ).

例 **592: Translation of signals**

UML

```
interface MyWebService {
  signal Call(Charstring, Boolean);
}
```

WSDL

```
<wsdl:message name="CallRequest">
  <wsdl:part name="par_1" type="xsd:string"/>
  <wsdl:part name="par_2" type="xsd:boolean"/>
</wsdl:message>
<wsdl:portType name="MyWebService">
  <wsdl:operation name="Call">
    <wsdl:input message="tns:CallRequest"/>
  </wsdl:operation>
</wsdl:portType>
```

属性

A UML attribute contained in an interface is translated to a WSDL message with one part corresponding to the attribute.

The name of both the message and the part is the name of the attribute. The WSDL part gets a type (or element) reference which is the translation of the type of the UML parameter. See 型 for more information.

例 **593: Translation of attributes**

UML

```
interface MyWebService {
  Integer sessionId;
  UserData nameAndPassword;
}
```

WSDL

```
<wsdl:message name="sessionId">
  <wsdl:part name="sessionId" type="xsd:integer"/>
</wsdl:message>
<wsdl:message name="nameAndPassword">
  <wsdl:part name="nameAndPassword" element="tns:UserData"/>
</wsdl:message>
<wsdl:portType name="MyWebService"/>
```

The WSDL messages generated for interface attributes can for example be referenced in a SOAP binding using the `<soap:header>` tag (see soapHeader::part). SOAP headers are typically used for transmitting additional data with a SOAP request than what is carried by the parameters of the called operation. They are often used for data that is common to many of the operations in a web service, such as user authentication information or session token data. In UML you can set-up the value of such data by assigning values to the interface attributes. These values will then be transmitted whenever an operation is called (provided the appropriate SOAP binding has been defined of course).

例外

A UML exception specification for an interface operation is translated to a `<fault>` element for the corresponding WSDL operation.

A WSDL message is also generated which contains one single part, named “data”, whose type is the WSDL translation of the exception type (see 型).

The name of the WSDL fault is “exception_<index>”, where <index> is an index to make the name unique within the operation. The name of the message is “<opName>Exception_<index>”, where <opName> is the name of the operation.

例 **594: Translation of exception specifications for operations**

UML

```
interface MyWebService {
  void DoSomething() throw Integer,   // Predefined type
                           ErrorInfo; // User-defined type
}
```

WSDL

```
<wsdl:message name="DoSomethingRequest"/>
<wsdl:message name="DoSomethingException_0">
  <wsdl:part name="data" type="xsd:integer"/>
</wsdl:message>
<wsdl:message name="DoSomethingException_1">
  <wsdl:part name="data" element="tns:UserData"/>
</wsdl:message>
<wsdl:portType name="MyWebService">
  <wsdl:operation name="DoSomething">
    <wsdl:input message="tns:DoSomethingRequest"/>
    <wsdl:fault name="exception_0"
message="tns:DoSomethingException_0"/>
    <wsdl:fault name="exception_1"
message="tns:DoSomethingException_1"/>
  </wsdl:operation>
</wsdl:portType>
```

型

UML types used as the type of interface operation parameters, interface attributes and exception types, are translated to XSD types included in the WSDL document in the `<types>` section. Predefined UML types are directly mapped to built-in XSD types according to the table in XS Profile Contents.

Note that tags representing typed elements in WSDL use an attribute called “type” if the type is an XSD simpleType or complexType, while the attribute is called “element” if the type is an XSD element.

If the WSDL file artifact has an `<<xsd::import>>` dependency to an XSD file artifact or package the WSDL generator assumes that the imported XSD file contains all type definitions used in the WSDL document. It will then not generate XSD types in the `<types>` section automatically. However, if no XSD file is manually imported, a `<types>` section will be generated. It will contain all types that are used within the WSDL document (types of interface attributes, operation parameters, operation exception types etc.). It will also contain all types used by such types, recursively, until a closed type system is obtained.

See also the translator option XSD ファイルの生成 which makes it possible to tell the WSDL code generator to generate all XSD types in a separate file and import it.

バインディング アーティファクト

A UML artifact stereotyped by a binding stereotype and with a dependency to an interface manifested by a WSDL file artifact, is translated to a <binding> element in the WSDL document. The dependency is translated to a <port> element in the WSDL service.

The dependency should be stereotyped by an address stereotype specifying the location of the WSDL port described by the dependency.

The name of the WSDL binding is the name of the artifact. The name of the WSDL port is the name of the artifact with the suffix “Port” appended. The port refers to the binding which in turn refers to the portType.

注記

The only binding currently supported is the SOAP binding. See 非 SOAP バインディング for more information about this limitation.

デフォルト バインディング

The WSDL generator supports the generation of a default binding (a SOAP binding) when generating WSDL for an interface for which no binding artifact has been specified. The generated binding artifact uses default values for all properties, except the <soap:adress> ‘location’ which must be specified explicitly. To specify the location of a web service use the Property Editor on the <<soapAddress>> dependency.

例 **595: Generation of a default SOAP binding for a web service**

UML

```
interface MyWebService {}

<<soapBinding>> artifact MyWebServiceSOAPBinding
<<soapAddress>> dependency to MyWebService {}
```

WSDL

```
<wsdl:portType name="MyWebService">
</wsdl:portType>
<wsdl:binding name="MyWebServiceSOAPBinding"
type="tns:MyWebService">
  <soap:binding style="document"
transport="http://schemas.xmlsoap.org/soap/http"/>
</wsdl:binding>
<wsdl:service name="MyWebService">
  <wsdl:port name="MyWebServiceSOAPBindingPort"
binding="tns:MyWebServiceSOAPBinding">
    <soap:address/>
  </wsdl:port>
</wsdl:service>
```

To change some of the binding properties from their default values use the Property Editor on the <<soapBinding>> artifact.

The available properties of the `<<soapBinding>>` and `<<soapAddress>>` stereotypes are described in SOAP バインディング プロパティ .

共通バインディング

It is possible to use a common binding for multiple web services. This is specified by letting the supplier of the dependency from the binding artifact be a package instead of an interface. All interfaces contained in that package will use the same binding.

.

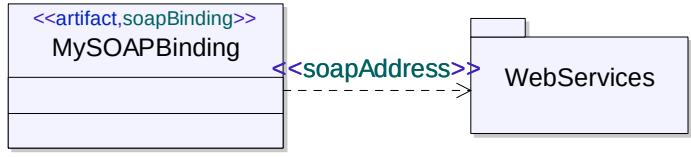


図 271: Specifying a common binding for multiple web services

名前変数

It is possible to use a `${Name}` variable in the 'location' tagged value of the `<<soapAddress>>`. This variable expands to the name of the web service interface. For example, `http://www.ibm.com/${Name}?WSDL`

Although use of this variable is mostly useful when specifying a common binding for multiple web services, it can also be used on per-interface bindings. Then the location URI does not need to be updated if the interface is renamed.

SOAP バインディング プロパティ

The following properties can be set to define a SOAP binding. They are defined as attribute of the `<<soapBinding>>` stereotype, and on classes used as the type of such attributes.

soapBinding::style

This property specifies the default style for each operation of the binding. Eligible values for the property is `document` and `rpc`. The default value is `document`. The style for a particular operation can be overridden by setting the value of the soapOperation::style property.

soapBinding::transport

This property specifies the transport which the SOAP binding corresponds to. The default value specifies the HTTP binding of the SOAP specification (`http://schemas.xmlsoap.org/soap/http`).

soapBinding::action

This property specifies the default SOAP action for each operation of the binding. A common pattern is that this URI consists of a common part followed by the name of the operation. The WSDL generator therefore supports use of the variable `{$Name}` which expands to the name of the operation.

The action for a particular operation can be specified by setting the value of the soapOperation::action property.

soapBinding::operationBinding

This property allows you to customize the binding for a particular operation. The value of the property is a list of `soapOperation` instances, each of which specifies an operation in the interface of the binding, and properties for the binding of that operation.

soapOperation::operation

This property specifies one of the operations in the interface of the binding. The other properties in `soapOperation` specify binding properties for that operation.

soapOperation::style

This property specifies the style for the binding of a particular operation. Eligible values are `document` and `rpc`. If the value is unspecified it defaults to the value of soapBinding::style. If that value also is unspecified, the default style is `document`.

soapOperation::action

This property specifies the SOAP action for a particular operation. Its value should be a valid URI.

soapOperation::input

This property specifies binding information related to the <input> element of the SOAP binding for a particular operation. Its value is an `soapInput` instance.

soapOperation::output

This property specifies binding information related to the <output> element of the SOAP binding for a particular operation. Its value is an `soapOutput` instance.

soapOperation::fault

This property specifies binding information related to the <fault> element of the SOAP binding for a particular operation. Its value is a `soapFault` instance.

soapInput::body

This property specifies how the message parts appear inside the SOAP body element of a SOAP operation input. Its value is a `soapBody` instance.

soapInput::header

This property allows the specification of data that is to be transmitted inside the header element of the SOAP envelope of a SOAP operation input. Its value is a `soapHeader` instance.

soapOutput::body

This property specifies how the message parts appear inside the SOAP body element of a SOAP operation output. Its value is a `soapBody` instance.

soapOutput::header

This property allows the specification of data that is to be transmitted inside the header element of the SOAP envelope of a SOAP operation output. Its value is a `soapHeader` instance.

soapBody::parts

The value of this property is a list of names referring to parts within a WSDL input or output message. It defines which parts that appear within the SOAP body portion of the message. The default value for this property is an empty list, which means that all parts defined by the message will be included.

soapBody::use

This property specifies whether the parts of an input or output message are encoded or not when transmitted. Eligible values for this property are `encoded` (parts are encoded) or `literal` (parts are not encoded). It is mandatory to specify a value for this property - if it is omitted it defaults to `literal`.

soapBody::encodingStyle

The value of this property is used if soapBody::use specifies that message parts are encoded. It specifies which rules that are used for the encoding. Its value is a list of URIs.

soapBody::namespace

The value of this property is used if soapBody::use specifies that message parts are encoded. It should be a valid URI and will be passed as input to the specified encoding.

soapHeader::isFault

This property specifies whether the containing `soapHeader` instance shall be translated to a `<soap:header>` or `<soap:headerfault>` element within the generated binding. The default value for this property is false.

Note that `<soap:headerfault>` elements are nested within the preceding `<soap:header>` element. If a `soapHeader` instance with `isFault` specified to true has no preceding `soapHeader` instance with `isFault` specified to false in the list, it will be ignored by the WSDL generator.

soapHeader::part

The value for this property should be the name of an attribute of the binding interface, or the name of a parameter for the operation for which the SOAP header is specified. Its value is mapped according to the following rules:

- If an interface attribute is specified, the `<soap:header>` (or `<soap:headerfeault>`) element gets its ‘message’ attribute set to the name of the WSDL message that corresponds to the attribute. The ‘part’ attribute is set to the name of the (one and only) WSDL part of that message.
- If an operation parameter is specified, the `<soap:header>` (or `<soap:headerfeault>`) element gets its ‘message’ field set to the name of the WSDL message that contains the part that corresponds to the parameter. The ‘part’ attribute is set to the name of that part.

soapHeader::use

The value of this property is used in the same way as the value of soapBody::use, but for the SOAP header.

soapHeader::encodingStyle

The value of this property is used in the same way as the value of soapBody::encodingStyle, but for the SOAP header.

soapHeader::namespace

The value of this property is used in the same way as the value of soapBody::namespace, but for the SOAP header.

soapFault::name

This property specifies a WSDL fault for an operation. See 例外 for more information about the naming of WSDL faults corresponding to exceptions in UML. If no value is specified for this property it defaults to “exception_0”, which is the name of the WSDL fault generated for the first exception specified for the UML operation.

soapFault::use

The value of this property is used in the same way as the value of soapBody::use, but for the SOAP fault.

soapFault::encodingStyle

The value of this property is used in the same way as the value of soapBody::encodingStyle, but for the SOAP fault.

soapFault::namespace

The value of this property is used in the same way as the value of soapBody::namespace, but for the SOAP fault.

制約事項

WSDL documents generated by the WSDL generator complies with the WSDL 1.1 standard. Deviations and limitations from the standard are listed below.

転送プリミティブ

Only the one-way and request-response transmission primitives are supported. The remaining two (solicit-response and notification) are not supported. They are not frequently used in many web service specifications, and the WSDL standard does not specify a binding for these transmission primitives.

非 **SOAP** バインディング

Only the SOAP binding is supported. Other bindings from the WSDL standard, such as HTTP POST/GET and MIME bindings, are not supported.

翻訳オプション

Translation options are represented by means of tagged values on the WSDL build artifact. They can be set using the Properties Editor.

ターゲット名前空間

This is a string option specifying which target namespace to use for the generated WSDL document. The value of this option should be a URN, and it will be placed in the <definitions> element of the generated WSDL document.

If this option is unspecified the target namespace will default to the name of the build artifact.

パラメータ順の生成

This is a boolean option. If enabled the “parameterOrder” attribute will be generated on WSDL operations. The value of this attribute is a space separated list of parameters of the operation, in the order as they are defined in UML. The ‘return’ parameter is ignored.

例 **596: Parameter order generation**

UML

```
interface MyWebService {
  Boolean Op(inout Real a, Integer b, out Charstring c);
}
```

WSDL

```
<wsdl:message name="OpRequest">
  <wsdl:part name="a" type="xsd:double"/>
  <wsdl:part name="b" type="xsd:integer"/>
</wsdl:message>
<wsdl:message name="OpResponse">
  <wsdl:part name="a" type="xsd:double"/>
  <wsdl:part name="c" type="xsd:string"/>
  <wsdl:part name="result" type="xsd:boolean"/>
</wsdl:message>
<wsdl:portType name="MyWebService">
  <wsdl:operation name="Op" parameterOrder="a b c">
    <wsdl:input message="tns:Op_0Request"/>
    <wsdl:output message="tns:Op_0Response"/>
  </wsdl:operation>
</wsdl:portType>
```

XSD ファイルの生成

This is a boolean option which controls how the WSDL generator should handle types used in the WSDL definition, for the case when the WSDL file artifact does not have an `<<xsd::import>>` dependency to an XSD file artifact or package. If such a dependency exists the WSDL generator assumes the user has manually specified the XSD types in the imported file, and the value of this option is then ignored (see 依存).

Without <<xsd::import>> dependencies present the WSDL code generator will translate all types that are used within the WSDL document (types of interface attributes, operation parameters, operation exception types etc.). This includes indirectly used types. If this option is false the result of this translation is included in the WSDL document in the <types> section. However, if the option is true the WSDL generator will generate the types into an XSD file placed next to the generated WSDL file. An <import> element is then added to import the generated XSD file into the WSDL document.

52

WSDL/XSD インポータリファレンス

この章は、WSDL ファイルを注釈つきの UML モデル（WSDL centric web service representation）に翻訳する WSDL インポータのリファレンスガイドです。XSD ファイルの UML への翻訳についても説明します。

WSDL から UML へのマッピング規則

To understand this document fully the following information sources are needed:

- The XSD and XS Profiles
- The WSDL Profile
- The SOAP Profile
- The SOAPENC Profile
- The XSD to UML Mapping Rules
- The XML namespace mapping rules

WSDL プロファイルの内容

The WSDL profile contains a set of stereotypes used to annotate a UML model with WSDL information. When importing a WSDL file, the resulting UML model is always explicitly annotated.

The stereotypes of the WSDL profile are listed in the table below.

Stereotype	Extends	Description
`<<documentation>>`	`Comment`	Represents `wsdl:documentation`
`<<message>>`	`Class`	Represents `wsdl:message`
`<<part>>`	`Attribute`	Represents `wsdl:part`
`<<portType>>`	`Interface`	Represents `wsdl:portType`
`<<input>>`	`Parameter`	Represents `wsdl:input`
`<<output>>`	`Parameter`	Represents `wsdl:output`
`<<fault>>`	`Parameter`	Represents `wsdl:fault`
`<<binding>>`	`Interface`	Represents `wsdl:binding`
`<<service>>`	`Class`	Represents `wsdl:service`
`<<wsdlPackage>>`	`Package`	Represents `wsdl:definitions`
`<<wsdlImport>>`	`Dependency`	Represents `wsdl:import`
`<<extensibilityElement>>`	`Entity`	A base stereotype for all WSDL extensibiliy elements. Any stereotypes specifying extensions, for example SOAP encoding, should be derived from this stereotype.

Stereotype	Extends	Description
<<wsdlDiagram>>	ClassDiagram	
<<elementReference>>	Dependency	see mapping of wsdl:part for details
<<parameterOrder>>	Operation	see mapping of wsdl:operation for details

マッピング規則

This chapter contains the actual mapping rules. It is organized by the different WSDL elements.

概要

WSDL	UML
<definition>	Package with stereotype <<wsdlPackage>>
<import>	Dependency with stereotype <<wsdlImport>>
<documentation>	Comment with stereotype <<documentation>>
<type>	Package with stereotype <<schema>>.
<message>	Class with stereotype <<message>>
<part>	Attribute with stereotype <<part>>
<portType>	Interface with stereotype <<portType>>
<operation>	Operation with stereotype <<operation>>
<input>	Parameter with stereotype <<input>>
<output>	Parameter with stereotype <<output>>
<fault>	Parameter with stereotype <<fault>>
<binding>	Interface with stereotype <<binding>>
<service>	Class with stereotype <<service>>
<port>	Port with stereotype <<port>>

Each WSDL element may have XML namespace declarations. These declaration are mapped to the stereotype <<xmlNamespace>> according to rules described in the chapter [XML Namepsace Mapping Rules].

拡張要素

WSDL elements can have extensibility elements representing a specific technology (e.g. SOAP, HTTP, etc). WSDL Importer keeps these elements either in specific stereotypes (for SOAP) or in the special stereotype `<<extensibilityElement>>`.

The stereotype instance representing extensibility element is owned by UML element which represents WSDL element owning this extensibility element.

Possible locations of the extensibility elements are defined in the WSDL specification[http://www.w3.org/TR/wsdl#A3].

定義

```
<wsdl:definitions name="nmtoken"?
targetNamespace="uri"?>
```

This is a root level of WSDL specification. It is mapped to a Package with stereotype `<<wsdlPackage>>`.

The attribute `name` is mapped to the `Name` of the Package. If the attribute `name` is omitted, then imported Package will have the same name as value of the `targetNamespace` attribute.

The attribute `targetNamespace` is mapped to the `targetNamespace` tagged value of the stereotype `<<wsdlPackage>>.`

例 **597** ———————————————————————————————————

WSDL:

```
<wsdl:definitions
targetNamespace="http://interpressfact.net/webservices/"
>
```

UML:

```
<<wsdlPackage(. targetNamespace =
http://interpressfact.net/webservices/ .)>>
package 'http://interpressfact.net/webservices/'
{
...
}
```

———————————————————————————————————————

インポート

```
<import namespace="uri" location="uri"/>
```

The `import` element is mapped to a Dependency with stereotype `<<wsdlImport>>`.

The attribute `namespace` is mapped to the `namespace` tagged value of the `<<wsdlImport>>` stereotype.

The attribute `location` is mapped to the `location` tagged value of the `<<wsdlImport>>` stereotype.

WSDL Importer attempts to automatically resolve location of the referenced document and import it. If this operation is successfull, the `Supplier` of the dependency will be set to the package that corresponds to the imported document. If WSDL Importer cannot automatically resolve location of the referenced document, the the `Supplier` of the dependency will be empty.

例 **598**

WSDL:

UML:

documentation

```
<wsdl:documentation .... /> ?
```

Documentation is mapped to a Comment with `<<documentation>>` stereotype owned by a UML representation of the enclosing element.

例 **599**

WSDL:

```
<documentation>This is a comment</documentation>
```

UML:

```
<<wsdl::documenation>> comment "This is a comment"
```

types

An XSD schema defined in a WSDL file is mapped to a schema package withing WSDL package. See the XSD to UML Mapping chapter for details.

message

```
<wsdl:message name="nmtoken">
```

The `message` element is mapped to a Class with stereotype `<<message>>`.

The attribute `name` is mapped to the `Name` of the Class.

例 **600**

WSDL:

```
<wsdl:message name="getJokeSoapIn">
...
</wsdl:message>
```

UML:

```
<<message>> class getJokeSoapIn {
...
```

```
}
```

part

```
<part name="nmtoken" element="qname"? type="qname"?/>
```

The part `element` is mapped to an Attribute with stereotype `<<part>>`.

The attribute `name` is mapped to the `Name` of the Attribute.

The attribute `type` is mapped to the `Type` of the Attribute.

The attribute `element` is mapped to a Dependency with stereotype `<<elementReference>>` owned by the Attribute. The `Supplier` of the Dependency will be set to UML representation of the XSD `<element>`.

例 **601**

WSDL:

```
<wsdl:part name="City" type="s:string"/>
<wsdl:part name="parameters" element="tns:getJoke"/>
```

UML:

```
<<'part'>> xs::string City;
<<'part'>> '' parameters <<elementReference>> dependency to
'http://interpressfact.net/webservices/'::getJoke;
```

portType

```
<wsdl:portType name="nmtoken">
```

The `portType` element is mapped to an Interface with stereotype `<<portType>>`.

The `Name` of the Interface is set to `<name>PortType` where the `<name>` is a value of the attribute `name`. The original name is kept in the stereotype `<<originalName>>`.

例 **602**

WSDL:

```
<wsdl:portType name="getJokeSoap">
....
</wsdl:portType>
```

UML:

```
<<portType, originalName(.name = "getJokeSoap".)>> interface
getJokeSoapPortType {
...
}
```

operation

```
<wsdl:operation name="nmtoken" parameterOrder="nmtokens"?>
```

The `operation` element is mapped to an Operation.

The attribute `name` is mapped to the `Name` of the Operation.

The attribute `parameterOrder` is mapped to the stereotype `<<parameterOrder>>`. The value of this attribute is mapped to the value of the attribute `order` of the stereotype `<<parameterOrder>>`.

例 **603**

WSDL:

```
<wsdl:operation name="getJoke">
...
</wsdl:operation>
```

UML:

```
void getJoke(...) {
...
}
```

input

```
<wsdl:input name="nmtoken"? message="qname">
```

The `input` element is mapped to a Parameter with the stereotype `<<input>>`.

The `direction` of the Parameter is set to `in`.

The attribute `name` is mapped to the `Name` of the Parameter.

If the attribute `name` was omitted, the Name of the Parameter is set to `'<wsdl:input>'` and the stereotype `<<originalName>>` with empty name is applied to the Parameter.

The attribute `message` is mapped to the `Type` of the Parameter.

例 **604**

WSDL:

```
<wsdl:input message="getJokeSoapIn"/>
```

UML:

```
void getJoke(
<<'input', originalName(.name = "".)>> in getJokeSoapIn
'<wsdl:input>'
....
)
```

output

```
<wsdl:output name="nmtoken"? message="qname">
```

The `output` element is mapped to a Parameter with the stereotype `<<output>>`.

The `direction` of the Parameter is set to `out`.

The attribute `name` is mapped to the `Name` of the Parameter.

If the attribute `name` was omitted, the `Name` of the Parameter is set to `'<wsdl:output>'` and the stereotype `<<originalName>>` with empty name is applied to the Parameter.

The attribute `message` is mapped to the `Type` of the Parameter.

例 **605**

WSDL:

```
<wsdl:output message="getJokeSoapOut"/>
```

UML:

```
void getJoke(
<<'output', originalName(.name = "".)>> outgetJokeSoapOut
'<wsdl:output>'
....
)
```

fault

```
<wsdl:fault name="nmtoken" message="qname">
```

The `output` element is mapped to a Parameter with the stereotype `<<output>>`.

The `direction` of the Parameter is set to `out`.

The attribute `name` is mapped to the `Name` of the Parameter.

The attribute `message` is mapped to the `Type` of the Parameter.

例 **606**

WSDL:

```
<wsdl:fault name="getJokeFault" message="getJokeSoapFault"/>
```

UML:

```
void getJoke(
<<'fault', originalName(.name = "".)>> out getJokeSoapFault
getJokeFault
....
)
```

binding

```
<wsdl:binding name="nmtoken" type="qname"> *
    <-- extensibility element (1) --> *
    <wsdl:operation name="nmtoken"> *
       <-- extensibility element (2) --> *
       <wsdl:input name="nmtoken"? > ?
           <-- extensibility element (3) -->
       </wsdl:input>
       <wsdl:output name="nmtoken"? > ?
           <-- extensibility element (4) --> *
       </wsdl:output>
       <wsdl:fault name="nmtoken"> *
           <-- extensibility element (5) --> *
       </wsdl:fault>
    </wsdl:operation>
</wsdl:binding>
```

The `binding` element is mapped to an Interface with the stereotype `<<binding>>`.

The `Name` of the Interface is set to `<binding name>Binding` where the `<binding name>` is a value of the attribute `name`. The original name of the binding is saved in the `<<originalName>>` stereotype.

The attriibute `type` is mapped to a `Generalization`. The `Parent` of the `Generalization` is set to the `Interface` which corresponds to UML representation of the `type`.

The `operation` element in the binding is mapped to an `Operation`. The attribute `name` is mapped to the `Name` of the `Operation`.

The `input` element in the `operation` element is mapped to a `Parameter` with the same properties as corresponding parameter from the `portType` operation.

The `output` element in the `operation` element is mapped to a `Parameter` with the same properties as corresponding parameter from the `portType` operation.

The `fault` element in the `operation` element is mapped to a `Parameter` with the same properties as corresponding parameter from the `portType` operation.

例 **607**

WSDL:

```
<wsdl:binding name="getJokeSoap" type="tns:getJokeSoap">
...
</wsdl:binding>
```

UML:

```
<<binding, originalName(.name = "getJokeSoap".)>> interface
getJokeSoapBinding : getJokeSoapPortType{
....
}
```

service

```
<wsdl:service name="nmtoken">
```

The `service` element is mapped to a `Class` with the stereotype `<<service>>`.

The attribute `name` is mapped to the `Name` of the `Class`.

例 **608**

WSDL:

```
<wsdl:service name="getJoke">
...
</wsdl:service>
```

UML:

```
<<service>> class getJoke {
...
}
```

port

```
<wsdl:port name="nmtoken" binding="qname">
```

The `port` element is mapped to a `Port`.

The attribute `name` is mapped to the `Name` of the `Port`.

The attribute `binding` is mapped to the `Realized` attribute of the `Port`.

例 **609**

WSDL:

```
<wsdl:port name="getJokeSoap" binding="getJokeSoap">
...
</wsdl:port>
```

UML:

```
port getJokeSoap in with getJokeSoap;
```

SOAP 1.1 Mapping Rules

WSDL includes a binding for SOAP 1.1 as part of the specification. This chapter describes mapping rules for SOAP 1.1 extensibility elements.

SOAP Profile Overview

The SOAP profile contains a set of types and stereotypes used to represent SOAP 1.1 elements when importing WSDL document.

Stereotypes of the SOAP profile:

Stereotype	Extends	Description
<<binding>>	Interface	Represents soap:binding
<<operation>>	Operation	Represents soap:operation
<<body>>	Parameter	Represents soap:body
<<header>>	Parameter	Represents soap:header
<<fault>>	Parameter	Represents soap:fault
<<address>>	Port	Represents soap:address

Types of the SOAP profile:

Type	Description
enum UseKind	Specify encoding rules for parts, headers, etc
enum StyleKind	Specify operation style

soap:address

```
<wsdl:port>
<soap:address location="uri"/>
</wsdl:port>
```

The soap:address element extends the wsdl:port. It is mapped to the stereotype <<soap::address>>.

The attributes of the soap:address are mapped to the attributes of the stereotype according to the table:

address attribute	stereotype attribute
location	location : xs::anyURI

soap:binding

```
<wsdl:binding>
<soap:binding style="rpc|document" transport="uri">
...
</wsdl:binding>
```

The `soap:binding` element extends the `wsdl:binding`. It is mapped to the stereotype `<<soap::binding>>`.

The attributes of the `soap:binding` are mapped to the attributes of the stereotype according to the table:

binding attribute	stereotype attribute
`style`	`style : StyleKind`
`transport`	`transport : xs::anyURI`

soap:operation

```
<wsdl:binding>
    <wsdl:operation>
       <soap:operation soapAction="uri"?
style="rpc|document"?>?
    </wsdl:operation>
</wsdl:binding>
```

The `soap:operation` element extends the `wsdl:operation` element. It is mapped to the stereotype `<<soap::operation>>`.

The attributes of the `soap:operation` are mapped to the attributes of the stereotype according to the table:

operation attribute	stereotype attribute
`soapAction`	`soapAction : xs::anyUri [0..1]`
`style`	`style : StyleKind [0..1]`

soap:body

```
<wsdl:binding>
        <wsdl:operation>
           <wsdl:input>
               <soap:body parts="nmtokens"?
use="literal|encoded"?
                          encodingStyle="uri-list"?
```

```
namespace="uri"?>
           </wsdl:input>
           <wsdl:output>
               <soap:body parts="nmtokens"?
use="literal|encoded"?
                          encodingStyle="uri-list"?
namespace="uri"?>
           </wsdl:output>
        </wsdl:operation>
</wsdl:binding>
```

The `soap:body` element extends the `wsdl:input` or `wsdl:output` elements. It is mapped to the stereotype `<<soap::body>>`.

The attributes of the `soap:body` are mapped to the attributes of the stereotype according to the table:

body attribute	**stereotype attribute**
parts	parts : xs::NMTOKEN [*]
use	use : UseKind
encodingStyle	encodingStyle : xs::anyUri[*]
namespace	namespace : xs::anyUri [0..1]

soap:fault

```
<wsdl:binding>
        <wsdl:operation>
          <wsdl:fault>*
            <soap:fault name="nmtoken" use="literal|encoded"
                                encodingStyle="uri-list"?
namespace="uri"?>
            </wsdl:fault>
        </wsdl:operation>
</wsdl:binding>
```

The `soap:fault` element extends the `wsdl:fault` element. It is mapped to the stereotype `<<soap::fault>>`.

The attributes of the `soap:fault` are mapped to the attributes of the stereotype according to the table:

fault attribute	**stereotype attribute**
name	name : xs::NMTOKEN
use	use : UseKind
encodingStyle	encodingStyle : xs::anyUri[*]
namespace	namespace : xs::anyUri [0..1]

soap:header

```
<wsdl:binding>
        <wsdl:operation>
           <wsdl:input>
             <soap:header message="qname" part="nmtoken"
use="literal|encoded"
                          encodingStyle="uri-list"?
namespace="uri"?>*
               <soap:headerfault message="qname"
part="nmtoken" use="literal|encoded"
                                 encodingStyle="uri-list"?
namespace="uri"?/>*
             </soap:header>
           </wsdl:input>
           <wsdl:output>
               <soap:header message="qname" part="nmtoken"
use="literal|encoded"
                            encodingStyle="uri-list"?
namespace="uri"?>*
                 <soap:headerfault message="qname"
part="nmtoken" use="literal|encoded"
                                   encodingStyle="uri-list"?
namespace="uri"?/>*
               </soap:header>
           </wsdl:output>
</wsdl:operation>
```

The `soap:header` element extends `wsdl:input` or `wsdl:output` elements. It is mapped to the stereotype `<<soap::header>>`.

The attributes of the `soap:header` are mapped to the attributes of the stereotype according to the table:

header attribute	stereotype attribute
message	message : xs::QName
part	part : xs:NMTOKENS
use	use : UseKind
encodingStyle	encodingStyle : xs::anyURI [*]
namespace	namespace : xs::anyURI [0..1]

The `soap:headerfault` element is mapped to the instance of the class `soap::headerfault`. This instance is set as value of the `faults` attribute of the stereotype `<<soap::header>>`. The attributes of the soap:headerfault are mapped to the attributes of the class soap::headerfault according to the table:

headerfault attribute	**class attribute**
namespace	namespace : xs::anyURI [0..1]
use	use : UseKind
part	part : xs::NMTOKEN
encodingStyle	encodingStyle : xs:anyURI [*]
name	name : xs:QName

XSD to UML Mapping Rules

XSD Profile Contents

The XSD profile contains a set of stereotypes and helper types used to annotate a UML model with XML Schema information. When importing a XSD file, the resulting UML model shall always be explicitly annotated.

For each XML Schema entity there is the stereotype in the XSD Profile with the same name.

Stereotype	Description
<<any>>	Represents xsd:any
<<key>>	Represents xsd:key
<<unique>>	Represents xsd:unique
<<selector>>	Represents xsd:selector
<<keyref>>	Represents xsd:keyref
<<field>>	Represents xsd:field
<<notation>>	Represents xsd:notation
<<appInfo>>	Represents xsd:appInfo
<<documentation>>	Represents xsd:documentation
<<simpleType>>	Represents xsd:simpleType
<<restriction>>	Represents xsd:restriction
<<list>>	Represents xsd:list
<<union>>	Represents xsd:union
<<include>>	Represents xsd:include
<<import>>	Represents xsd:import
<<simpleContent>>	Represents xsd:simpleContent
<<complexContent>>	Represents xsd:complexContent
<<all>>	Represents xsd:all

Stereotype	Description
<<choice>>	Represents xsd:choice
<<sequence>>	Represents xsd:sequence
<<extension>>	Represents xsd:extension

XS Profile Contents

The XS profile contains a set of types representing XML Schema Datatypes.

XSD type	UML type
anyType	datatype anyType
anySimpleType	datatype anySimpleType : anyType
duration	datatype duration : anySimpleType implicit conversions and assignment operators: • from the Predefined::Charstring
dataTime	datatype dateTime : anySimpleType implicit conversions and assignment operators: • from the Predefined::Charstring
time	datatype time : anySimpleType implicit conversions and assignment operators: • from the Predefined::Charstring
date	datatype date : anySimpleType implicit conversions and assignment operators: • from the Predefined::Charstring
gYearMonth	datatype gYearMonth : anySimpleType implicit conversions and assignment operators: • from the Predefined::Charstring
gYear	datatype gYear : anySimpleType implicit conversions and assignment operators: • from the Predefined::Charstring
gMonthDay	datatype gMonthDay : anySimpleType implicit conversions and assignment operators: • from the Predefined::Charstring
gDay	datatype gDay : anySimpleType implicit conversions and assignment operators: • from the Predefined::Charstring

XSD type	UML type
gMonth	datatype gMonth : anySimpleType implicit conversions and assignment operators: • from the Predefined::Charstring
boolean	datatype boolean : anySimpleType implicit conversions and assignment operators: • from the Predefined::Boolean
base64Binary	datatype base64Binary : anySimpleType implicit conversions and assignment operators: • from the Predefined::Integer
hexBinary	datatype hexBinary : anySimpleType implicit conversions and assignment operators: • from the Predefined::Integer • from the Predefined::Charstring
float	datatype float : anySimpleType implicit conversions and assignment operators: • from the Predefined::Real
double	datatype double : anySimpleType implicit conversions and assignment operators: • from the Predefined::Real
anyURI	datatype anyURI : anySimpleType implicit conversions and assignment operators: • from the Predefined::Charstring
QName	datatype QName : anySimpleType implicit conversions and assignment operators: • from the Predefined::Charstring
NOTATION	datatype NOTATION : anySimpleType implicit conversions and assignment operators: • from the Predefined::Charstring
string	datatype string : anySimpleType implicit conversions and assignment operators: • from the Predefined::Charstring
normalizedString	datatype normalizedString: string implicit conversions and assignment operators: • from the Predefined::Charstring
decimal	datatype decimal : anySimpleType implicit conversions and assignment operators: • from the Predefined::Real

XSD type	UML type
integer	datatype integer : decimal implicit conversions and assignment operators: • from the Predefined::Integer
token	datatype token : normalizedString implicit conversions and assignment operators: • from the Predefined::Charstring
nonPositiveInteger	datatype nonPositiveInteger : integer implicit conversions and assignment operators: • from the Predefined::Integer
long	datatype long : integer implicit conversions and assignment operators: • from the Predefined::Integer
nonNegativeInteger	datatype nonNegativeInteger : integer implicit conversions and assignment operators: • from the Predefined::Integer
language	datatype language : token implicit conversions and assignment operators: • from the Predefined::Charstring
Name	datatype Name : token implicit conversions and assignment operators: • from the Predefined::Charstring
NMTOKEN	datatype NMTOKEN : token implicit conversions and assignment operators: • from the Predefined::Charstring
negativeInteger	datatype negativeInteger : nonPositiveInteger implicit conversions and assignment operators: • from the Predefined::Integer
int	datatype int : long implicit conversions and assignment operators: • from the Predefined::Integer
unsignedLong	datatype unsignedLong : nonNegativeInteger implicit conversions and assignment operators: • from the Predefined::Integer

XSD type	UML type
positiveInteger	datatype positiveInteger : nonNegativeInteger implicit conversions and assignment operators: • from the Predefined::Integer
NCName	datatype NCName : Name implicit conversions and assignment operators: • from the Predefined::Charstring
NMTOKENS	datatype NMTOKENS
short	datatype short : int implicit conversions and assignment operators: • from the Predefined::Integer
unsignedInt	datatype unsignedInt : unsignedLong implicit conversions and assignment operators: • from the Predefined::Integer
ID	datatype ID : NCName implicit conversions and assignment operators: • from the Predefined::Charstring
IDREF	datatype IDREF : NCName implicit conversions and assignment operators: • from the Predefined::Charstring
ENTITY	datatype ENTITY : NCName implicit conversions and assignment operators: • from the Predefined::Charstring
byte	datatype byte : short implicit conversions and assignment operators: • from the Predefined::Integer
unsignedShort	datatype unsignedShort : unsignedInt implicit conversions and assignment operators: • from the Predefined::Integer
IDREFS	datatype IDREFS
ENTITIES	datatype ENTITIES
unsignedByte	datatype unsignedByte : unsignedShort implicit conversions and assignment operators: • from the Predefined::Integer

SOAPENC Profile Overview

The SOAPENC profile contains a set of types representing SOAP 1.1-specific encoding information. Most of these types extends types from the XS profile.

マッピング規則

概要

XSD	UML
<xsd:attribute>	attribute with the stereotype <<attirbute>>
<xsd:attributeGroup>	attribute or class with the stereotype <<attirbuteGroup>>
<xsd:complexType>	class with the stereotype <<complexType>>
<xsd:complexContent>	class with the stereotype <<complexContent>>
<xsd:documentation>	comment with the stereotype <<documentation>>
<xsd:appinfo>	comment with the stereotype <<appinfo>>
<xsd:element>	attribute with the stereotype <<element>>
<xsd:schema>	package with the stereotype <<schema>>
<xsd:simpleContent>	class with the stereotype <<simpleContent>>
<xsd:sequence>	class with the stereotype <<sequence>>
<xsd:key>	informal contstraint with the stereotype <<key>>
<xsd:keyref>	informal constraint with the stereotype <<keyref>>
<xsd:unique>	informal constraint with the stereotype <<unique>>
<xsd:selector>	informal constraint with the stereotype <<selector>>
<xsd:field>	informal constraint with the stereotype <<field>>
<xsd:group>	attribute or class with the stereotype <<group>>
<xsd:all>	class with the stereotype <<all>>
<xsd:any>	attribute with the stereotype <<any>>

XSD	UML
<xsd:anyAttribute>	attribute with the stereotype <<anyAttribute>>
<xsd:choice>	class with the stereotype <<choice>>
<xsd:import>	dependency with the stereotype <<import>>
<xsd:include>	dependency with the stereotype <<include>>
<xsd:redefine>	package with the stereotype <<redefine>>
<xsd:simpleType>	data type with the stereotype <<simpleType>>
<xsd:list>	data type with the stereotype <<list>>
<xsd:union>	data type with the stereotype <<union>>
<xsd:annotation>	stereotype <<annotation>> or artifact with the stereotype <<annotation>>
<xsd:extension>	generalization with the stereotype <<extension>>
<xsd:restriction>	generalization with the stereotype <<restriction>>
<xsd:notation>	artifact with the stereotype <notation>

Importing non-schema elements

When importing XSD document all non-schema elements are ignored.

Importing non-schema attributes

A non-scheme attribute on a XSD entity is mapped to the stereotype <<xmlAttribute>> from the TTDXmlFramework profile.

The name of the attribute is mapped to the attribute name of the stereotype <<xmlAttribute>>.

The value of the attribute is mapped to the attribute value of the stereotype <<xmlAttribute>>.

ID attribute

The id attribute on XSD entities is mapped to the stereotype <<xmlAttribute>>.

attribute declaration

```
<xsd:attribute
  default = string
```

```
    fixed = string
    form = (qualified | unqualified)
    id = ID
    name = NCName
    ref = QName
    type = QName
    use = (optional | prohibited | required) : optional
    {any attributes with non-schema namespace . . .}>
    Content: (annotation?, simpleType?)
  </xsd:attribute>
```

The xsd:attribute element is mapped to an Attribute with the stereotype <<attribute>>.

The name of the element xsd:attribute is mapped to the Name of the Attribute.

The type of the element xsd:attribute is mapped to the Type of the Attribute.

The ref of the element xsd:attribute is mapped to a Dependency with the stereotype <<ref>>. The value of the ref is mapped to the Supplier of the Dependency.

The Type of the Attribute is set to Predefined::ExternalObject if the ref is present, but the type is not specified.

The other attributes of the xsd:attribute are mapped to the attributes of the stereotype <<attribute>> according to the table:

XSD	UML
default	default : Charstring [0..1]
form	form : FormKind [0..1]
use	use : UseKind [0..1]
fixed	fixed : Charstring [0..1]

If xsd:attribute contained a xsd:simpleType, the UML element corresponded to the xsd:simpleType is inserted in the InlineType of the Attribute.

element declaration

```
  <xsd:element
    abstract = boolean : false
    block = (#all | List of (extension | restriction |
substitution))
    default = string
    final = (#all | List of (extension | restriction))
    fixed = string
    form = (qualified | unqualified)
    id = ID
    maxOccurs = (nonNegativeInteger | unbounded)  : 1
```

```
    minOccurs = nonNegativeInteger : 1
    name = NCName
    nillable = boolean : false
    ref = QName
    substitutionGroup = QName
    type = QName
    {any attributes with non-schema namespace . . .}>
    Content: (annotation?, ((simpleType | complexType)?,
  (unique | key | keyref)*))
  </xsd:element>
```

The `xsd:element` element is mapped to an `Attribute` with the stereotype `<<element>>`.

The `name` of the element `xsd:element` is mapped to the `Name` of the `Attribute`.

The `type` of the element `xsd:element` is mapped to the `Type` of the `Attribute`.

The `ref` attribute of the element `xsd:element` is mapped to a `Dependency` with the stereotype `<<ref>>`. The value of the `ref` is mapped to the `Supplier` of the `Dependency`. If the `type` attribute is not present, the `Type` of the `Attribute` is set to `Predefined::ExternalObject`.

The `abstract` attribute of the element `xsd:element` is mapped to the `Abstract` property of the `Attribute`.

The other attributes of the element `xsd:element` are mapped to the attributes of the stereotype `<<element>>` according to the table:

XSD	UML
block	block : BlockKind [*]
default	default : xs::string [0..1]
final	final : FinalKind [*]
fixed	fixed : xs:string [0..1]
form	form : FormKind [0..1]
maxOccurs	maxOccurs : MaxOccurence
minOccurs	minOccurs : xs:nonNegativeInteger [0..1]
substitutionGroup	substitutionGroup : xs::QName [0..1]
nillable	nillable : xs::boolean [0..1]

If the `Content` of the `xsd:element` is a `simpleType` or a `complexType`, the corresponding UML element is inserted in the `InlineType` of the `Attribute`.

If the `Content` of the `xsd:element` is a `unique, key or keyref`, the corresponding UML element is inserted in the `Constraints` of the `Attribute`.

complex type

```
<xsd:complexType
  abstract = boolean : false
  block = (#all | List of (extension | restriction))
  final = (#all | List of (extension | restriction))
  id = ID
  mixed = boolean : false
  name = NCName
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?, (simpleContent | complexContent |
((group | all | choice | sequence)?, ((attribute |
attributeGroup)*, anyAttribute?))))
</xsd:complexType>
```

The `xsd:complexType` element is mapped to a `Class` with the stereotype `<<complexType>>`.

The attribute `name` is mapped to the `Name` of the `Class`.

The other attributes of the element `xsd:complexType` are mapped to the attributes of the stereotype `<<complexType>>` according to the table:

XSD	UML
`block`	`block : BlockKind [0..1]`
`final`	`final : FinalKind [0..1]`
`mixed`	`mixed : xs::boolean`

The UML representation of the `Content` of the `xsd:complexType` is inserted in the `OwnedMember` of the `Class`.

simple content

```
<xsd:simpleContent
  id = ID
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?, (restriction | extension))
</xsd:simpleContent>
```

The `xsd:simpleContent` element is mapped to a `Class` with the stereotype `<<simpleConent>>`.

The `Name` of the `Class` is set to `"<simpleContent.`

simple content : restriction

```
<xsd:restriction
  base = QName
  id = ID
```

```
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?, (simpleType?, (minExclusive |
minInclusive | maxExclusive | maxInclusive | totalDigits |
fractionDigits | length | minLength | maxLength | enumeration
| whiteSpace | pattern)*)?, ((attribute | attributeGroup)*,
anyAttribute?))
</xsd:restriction>
```

The `xsd:restriction` element is mapped to a `Generalization` with the stereotype `<<restriction>>`.

The attribute `base` is mapped to the `Parent` of the `Generalization`.

If the `Content` of the `xsd:restriction` is a `simpleType, attribute, attributeGroup or anyAttribute`, the corresponding UML element is inserted in the `OwnedMember` of the UML element representing parent element of the `xsd:restriction`. In other case the `Content` of the `xsd:restriction` is mapped to the attributes of the stereotype `<<restriction>>` according to the table:

Content	Stereotype attribute
minExclusive	minExclusive : anySimpleTypeRestriction
minInclusive	minInclusive : anySimpleTypeRestriction
maxInclusive	maxInclusive : anySimpleTypeRestriction
maxExclusive	maxExclusive : anySimpleTypeRestriction
totalDigits	totalDigits : positiveIntegerRestriction
fractionDigits	fractionDigits : nonNegativeIntegerRestriction
length	length : nonNegativeIntegerRestriction
minLength	minLength : nonNegativeIntegerRestriction
maxLength	maxLength : nonNegativeIntegerRestriction
enumeration	enumeration : Charstring [0..*]
whiteSpace	whiteSpace : whiteSpaceRestriction
pattern	pattern : stringRestriction

simple content : extension

```
<xsd:extension
  base = QName
  id = ID
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?, ((attribute | attributeGroup)*,
```

```
anyAttribute?))
</xsd:extension>
```

The `xsd:extension` element is mapped to a `Generalization` with the stereotype `<<extension>>`.

The attribute `base` is mapped to the `Parent` of the `Generalization`.

The UML element representing `Content` of the `xsd:extension` is inserted in the `OwnedMember` of the UML element representing parent element of the `xsd:extension`.

simple content : attrbute group

```
<xsd:attributeGroup
  id = ID
  ref = QName
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?)
</xsd:attributeGroup>
```

The `xsd:attributeGroup` element is mapped to an `Attribute` with the stereotype `<<attributeGroup>>`.

The `Name` of the `Attribute` is set to `"<attributeGroup>"`.

The value of the `ref` attribute is set as `Type` of the `Attribute`. A `Dependency` with the stereotype `<<ref>>` is created in the Attribute. The `Supplier` of the `Dependency` is set to the value of the `ref` attribute.

simple content: anyAttribute

```
<xsd:anyAttribute
  id = ID
  namespace = ((##any | ##other) | List of (anyURI |
(##targetNamespace | ##local)) )  : ##any
  processContents = (lax | skip | strict) : strict
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?)
</xsd:anyAttribute>
```

The `xsd:anyAttribute` element is mapped to an `Attribute` with the stereotype `<<anyAttribute>>`.

The `Name` of the `Atttribute` is set to `"<anyAttribute>"`.

The attributes of the `xsd:anyAttribute` are mapped to attributes of the stereotype `<<anyAttribute>>` according to the table:

XSD	UML
namespace	namespace : namespaceSpecification
processContents	processContents : ProcessKind [0..1]

complex content

```
<xsd:complexContent
  id = ID
  mixed = boolean
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?, (restriction | extension))
</xsd:complexContent>
```

The `xsd:complexContent` is mapped to a `Class` with the stereotype `<<complexContent>>`.

The `Name` of the `Class` is set to `"<complexContent>"`.

The attributes of the `xsd:complexContent` are mapped to the attributes of the stereotype `<<complexContent>>` according to the table:

XSD	UML
`mixed`	`mixed : xs::boolean [0..1]`

complex content:restriction

```
<xsd:restriction
  base = QName
  id = ID
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?, (group | all | choice | sequence)?,
((attribute | attributeGroup)*, anyAttribute?))
</xsd:restriction>
```

The `xsd:restriction` element is mapped to a `Generalization` with the stereotype `<<restriction>>`.

The `base` attribute is mapped to the `Parent` of the `Generalization`.

UML representation of the `Content` of the `xsd:restriction` is inserted in the `OwnedMember` of UML representation of the parent element.

complex content: extension

```
<xsd:extension
  base = QName
  id = ID
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?, ((group | all | choice | sequence)?,
((attribute | attributeGroup)*, anyAttribute?)))
</xsd:extension>
```

The `xsd:extension` element is mapped to a `Generalization` with the stereotype `<<extension>>`.

The `base` attribute is mapped to the `Parent` of the `Generalization`.

UML representation of the Content of the xsd:extension is inserted in the OwnedMember of UML representation of the parent element.

attribute group definition

```
<xsd:attributeGroup
  id = ID
  name = NCName
  ref = QName
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?, ((attribute | attributeGroup)*,
anyAttribute?))
</xsd:attributeGroup>
```

If the xsd:attributeGroup element has the ref attribute, it is mapped to an Attribute with the stereotype <<attributeGroup>>. The attribute name is mapped to the Name of the Attribute. The value of the attribute ref is mapped to the Type of the Attribute. A Dependency with the stereotype <<ref>> is insertd in the Attribute. The Supplier of the Dependency is set to the value of the ref attribute.

If the xsd::attributeGroup element does not have the ref attribute, it is mapped to a Class with the stereotype <<attributeGroup>>. The attribute name is mapped to the Name of the Class. UML representation of the Content of the xsd:attributeGroup is inserted in the OwnedMember of UML representation of the parent element.

model group definition

```
<xsd:group
  id = ID
  maxOccurs = (nonNegativeInteger | unbounded)  : 1
  minOccurs = nonNegativeInteger : 1
  name = NCName
  ref = QName
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?, (all | choice | sequence)?)
</xsd:group>
```

If the xsd:group element has the ref attribute, it is mapped to an Attribute with the stereotype <<group>>. The attribute name is mapped to the Name of the Attribute. The value of the attribute ref is mapped to the Type of the Attribute. A Dependency with the stereotype <<ref>> is insertd in the Attribute. The Supplier of the Dependency is set to the value of the ref attribute.

If the xsd::group element does not have the ref attribute, it is mapped to a Class with the stereotype <<group>>. The attribute name is mapped to the Name of the Class. UML representation of the Content of the xsd:group is inserted in the OwnedMember of UML representation of the parent element.

Other attributes of the xsd:group element are mapped to the attributes of the stereotype <<group>> according to the table:

XSD	UML
maxOccurs	maxOccurs : MaxOccurence
minOccurs	minOccurs : xs::nonNegativeInteger [0..1]

model group schema component: all

```
<xsd:all
  id = ID
  maxOccurs = 1 : 1
  minOccurs = (0 | 1) : 1
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?, element*)
</xsd:all>
```

The `xsd:all` element is mapped to a `Class` with the stereotype `<<all>>`.

The `Name` of the `Class` is set to `"<all>"`.

The UML representation of the `Content` of the `xsd:all` is inserted in the `OwnedMember` of the `Class`.

The attributes of the `xsd:all` are mapped to the attributes of the stereotype `<<all>>` according to the table:

XSD	UML
maxOccurs	maxOccurs : MaxOccurence
minOccurs	minOccurs : xs::nonNegativeInteger [0..1]

model group schema component: choice

```
<xsd:choice
  id = ID
  maxOccurs = (nonNegativeInteger | unbounded)  : 1
  minOccurs = nonNegativeInteger : 1
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?, (element | group | choice | sequence
| any)*)
</xsd:choice>
```

The `xsd:choice` element is mapped to a `Class` with the stereotype `<<choice>>`.

The `Name` of the `Class` is set to `"<choice>"`.

The UML representation of the `Content` of the `xsd:choice` is inserted in the `OwnedMember` of the `Class`.

The attributes of the `xsd:all` are mapped to the attributes of the stereotype `<<choice>>` according to the table:

XSD	UML
maxOccurs	maxOccurs : MaxOccurence
minOccurs	minOccurs : xs::nonNegativeInteger [0..1]

model group schema component: sequence

```
<xsd:sequence
  id = ID
  maxOccurs = (nonNegativeInteger | unbounded)  : 1
  minOccurs = nonNegativeInteger : 1
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?, (element | group | choice | sequence
| any)*)
</xsd:sequence>
```

The `xsd:sequence` element is mapped to a `Class` with the stereotype `<<sequence>>`.

The `Name` of the `Class` is set to `"<sequence>"`.

The UML representation of the `Content` of the `xsd:sequence` is inserted in the `OwnedMember` of the `Class`.

The attributes of the `xsd:sequence` are mapped to the attributes of the stereotype `<<sequence>>` according to the table:

XSD	UML
maxOccurs	maxOccurs : MaxOccurence
minOccurs	minOccurs : xs::nonNegativeInteger [0..1]

wildcard schema component : any

```
<xsd:any
  id = ID
  maxOccurs = (nonNegativeInteger | unbounded)  : 1
  minOccurs = nonNegativeInteger : 1
  namespace = ((##any | ##other) | List of (anyURI |
(##targetNamespace | ##local)) )  : ##any
  processContents = (lax | skip | strict) : strict
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?)
</xsd:any>
```

The `xsd:any` element is mapped to an `Attribute` with the stereotype `<<any>>`.

The Name of the Attribute is set to "<any>".

The attributes of the xsd:any are mapped to the attributes of the stereotype <<any>> according to the table:

XSD	UML
maxOccurs	maxOccurs : MaxOccurence
minOccurs	minOccurs : xs::nonNegativeInteger [0..1]
namespace	namespace : NamespaceSpecification [0..1]
processContents	processContents : ProcessKind [0..1]

identity-constraint definition schema component:unique

```
<xsd:unique
  id = ID
  name = NCName
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?, (selector, field+))
</xsd:unique>
```

The xsd:unique element is mapped to an InformalConstraint with the stereotype <<unique>>.

The attributes of the xsd:unique element are mapped to the attributes of the stereotype <<unique>> according to the table:

XSD	UML
name	name : xs::NCName

identity-constraint definition schema component:key

```
<xsd:key
  id = ID
  name = NCName
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?, (selector, field+))
</xsd:key>
```

The xsd:key element is mapped to an InformalConstraint with the stereotype <<key>>.

The attributes of the xsd:key element are mapped to the attributes of the stereotype <<key>> according to the table:

XSD	UML
name	name : xs::NCName

identity-constraint definition schema component:keyref

```
<keyref
  id = ID
  name = NCName
  refer = QName
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?, (selector, field+))
</keyref>
```

The `xsd:keyref` element is mapped to an `InformalConstraint` with the stereotype `<<keyref>>`.

The attributes of the `xsd:keyref` element are mapped to the attributes of the stereotype `<<keyref>>` according to the table:

XSD	UML
name	name : xs::NCName
refer	refer : xs::QName

identity-constraint definition schema component:selector

```
<xsd:selector
  id = ID
  xpath = a subset of XPath expression
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?)
</xsd:selector>
```

The `xsd:selector` element is mapped to an `InformalConstraint` with the stereotype `<<selector>>`.

The attributes of the `xsd:selector` element are mapped to the attributes of the stereotype `<<selector>>` according to the table:

XSD	UML
xpath	name : xs::string

identity-constraint definition schema component:field

```
<xsd:field
  id = ID
```

```
    xpath = a subset of XPath expression
    {any attributes with non-schema namespace . . .}>
    Content: (annotation?)
</xsd:field>
```

The `xsd:field` element is mapped to an `InformalConstraint` with the stereotype `<<field>>`.

The attributes of the `xsd:field` element are mapped to the attributes of the stereotype `<<field>>` according to the table:

XSD	UML
xpath	name : xs::string

notation declaration

```
<xsd:notation
  id = ID
  name = NCName
  public = token
  system = anyURI
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?)
</xsd:notation>
```

The `xsd:notation` element is mapped to an `Artifact` with the stereotype `<<notation>>`.

The `name` of the `xsd:notation` is mapped to the `Name` of the `Artifact`.

Other attributes of the `xsd:notation` are mapped to the attributes of the stereotype `<<notation>>` according to the table:

XSD	UML
public	public : xs::token [0..1]
system	system : xs::anyURI [0..1]

annotation

```
<xsd:annotation
  id = ID
  {any attributes with non-schema namespace . . .}>
  Content: (appinfo | documentation)*
</xsd:annotation>
```

If the `xsd:annotation` is defined in the `xsd:schema` or in the `xsd:redefine`, it is mapped to an Artifact with the stereotype `<<annotation>>`.

In other cases, the `xsd:annotation` is mapped to the stereotype `<<annotation>>` which is applied to the UML element representing parent element of the `xsd:annotation`

annotation : appinfo

```
<xsd:appinfo
  source = anyURI
  {any attributes with non-schema namespace . . .}>
  Content: ({any})*
</xsd:appinfo>
```

The `xsd:appinfo` element is mapped to a `Comment` with the stereotype `<<appinfo>>`.

The attributes of the `xsd:appinfo` are mapped to the attributes of the stereotype `<<appinfo>>` according to the table:

XSD	UML
`source`	`source : xs::anyURI [0..1]`

The `Content` of the `xsd:appinfo` is mapped to the `Text` of the `Comment`.

annotation : documentation

```
<xsd:documentation
  source = anyURI
  xml:lang = language
  {any attributes with non-schema namespace . . .}>
  Content: ({any})*
</xsd:documentation>
```

The `xsd:documentation` element is mapped to a `Comment` with the stereotype `<<documentation>>`.

The attributes of the `xsd:documentation` are mapped to the attributes of the stereotype `<<documentation>>` according to the table:

XSD	UML
`source`	`source : xs::anyURI [0..1]`
`xml:lang`	`'xml:lang' : xs::language [0..1]`

The `Content` of the `xsd:documentation` is mapped to the `Text` of the `Comment`.

simple type definition

```
<xsd:simpleType
  final = (#all | List of (list | union | restriction))
  id = ID
  name = NCName
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?, (restriction | list | union))
</xsd:simpleType>
```

The xsd:simpleType element is mapped to a DataType with the stereotype <<simpleType>>.

The name of the xsd:simpleType is mapped to the Name of the DataType.

The attributes of the xsd:simpleType are mapped to the attributes of the stereotype <<simpleType>> according to the table:

XSD	UML
final	final : FinalKind

simple type : restriction

```
<xsd:restriction
  base = QName
  id = ID
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?, (simpleType?, (minExclusive |
minInclusive | maxExclusive | maxInclusive | totalDigits |
fractionDigits | length | minLength | maxLength | enumeration
| whiteSpace | pattern)*))
</xsd:restriction>
```

The xsd:restriction element is mapped to a Generalization with the stereotype <<restriction>>.

The attribute base is mapped to the Parent of the Generalization.

If the Content of the xsd:restriction is a simpleType, the corresponding UML element is inserted in the OwnedMember of the UML element representing parent element of the xsd:restriction. In other case the Content of the xsd:restriction is mapped to the attributes of the stereotype <<restriction>> according to the table:

Content	Stereotype attribute
minExclusive	minExclusive : anySimpleTypeRestriction
minInclusive	minInclusive : anySimpleTypeRestriction
maxInclusive	maxInclusive : anySimpleTypeRestriction
maxExclusive	maxExclusive : anySimpleTypeRestriction
totalDigits	totalDigits : positiveIntegerRestriction
fractionDigits	fractionDigits : nonNegativeIntegerRestriction
length	length : nonNegativeIntegerRestriction

Content	Stereotype attribute
minLength	minLength : nonNegativeIntegerRestriction
maxLength	maxLength : nonNegativeIntegerRestriction
enumeration	enumeration : Charstring [0..*]
whiteSpace	whiteSpace : whiteSpaceRestriction
pattern	pattern : stringRestriction

simple type : list

```
<xsd:list
  id = ID
  itemType = QName
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?, simpleType?)
</xsd:list>
```

The xsd:list element is mapped to a DataType with the stereotype <<list>>.

The Name of the DataType is set to "<list>".

The attributes of the xsd:list are mapped to the attributes of the stereotype <<list>> according to the table:

XSD	UML
itemType	itemType : xs::QName

simple type : union

```
<xsd:union
  id = ID
  memberTypes = List of QName
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?, simpleType*)
</xsd:union>
```

The xsd:union is mapped to a DataType with the stereotype <<union>>.

The Name of the DataType is set to "<union>".

The attributes of the xsd:union are mapped to the attributes of the stereotype <<union>> according to the table:

XSD	UML
memberTypes	memberTypes : xs::QName [*]

schema

```
<xsd:schema
  attributeFormDefault = (qualified | unqualified) :
unqualified
  blockDefault = (#all | List of (extension | restriction |
substitution))  : ''
  elementFormDefault = (qualified | unqualified) :
unqualified
  finalDefault = (#all | List of (extension | restriction |
list | union))  : ''
  id = ID
  targetNamespace = anyURI
  version = token
  xml:lang = language
  {any attributes with non-schema namespace . . .}>
  Content: ((include | import | redefine | annotation)*,
(((simpleType | complexType | group | attributeGroup) |
element | attribute | notation), annotation*)*)
</xsd:schema>
```

The `xsd:schema` element is mapped to a `Package` with the stereotype `<<schema>>`.

The `targetNamespace` attribute is mapped to the `Name` of the `Package`. If the `targetNamespace` is omitted, the `Name` of the `Package` is set to `"<schema>"`.

The other attributes of the `xsd:schema` element are mapped to the attributes of the stereotype `<<schema>>` according to the table:

XSD	UML
attributeFormDefault	attributeFormDefault : FormKind
blockDefault	blockDefault : BlockKind [0..1]
elementFormDefault	elementFormDefault : FormKind
finalDefault	finalDefault : FinalKind [0..1]
version	version : xs::token [0..1]
xml:lang	'xml:lang' : xsd::language [0..1]

include element

```
<xsd:include
  id = ID
```

```
    schemaLocation = anyURI
    {any attributes with non-schema namespace . . .}>
    Content: (annotation?)
  </xsd:include>
```

The `xsd:include` element is mapped to a `Dependency` with the stereotype `<<include>>`.

The attributes of the `xsd:include` are mapped to the attributes of the stereotype `<<include>>` according to the table:

XSD	UML
schemaLocation	schemaLocation : xs::anyUri

redefine element

```
  <xsd:redefine
    id = ID
    schemaLocation = anyURI
    {any attributes with non-schema namespace . . .}>
    Content: (annotation | (simpleType | complexType | group |
  attributeGroup))*
  </xsd:redefine>
```

The `xsd:redefine` element is mapped to a `Package` with the stereotype `<<redefine>>`.

The `Name` of the `Package` is set to `"<redefine>"`.

The attributes of the `xsd:redefine` are mapped to the attributes of the stereotype `<<redefine>>` according to the table:

XSD	UML
schemaLocation	schemaLocation : xs::anyUri

import element

```
  <xsd:import
    id = ID
    namespace = anyURI
    schemaLocation = anyURI
    {any attributes with non-schema namespace . . .}>
    Content: (annotation?)
  </xsd:import>
```

The `xsd:import` element is mapped to a `Dependency` with the stereotype `<<import>>`.

The attributes of the `xsd:import` are mapped to the attributes of the stereotype `<<import>>` according to the table:

XSD	UML
schemaLocation	schemaLocation : xs::anyUri [0..1]
namespace	namespace : xs::anyUri [0..1]

後処理

In order to reduce the number of levels in the imported UML model, the XSD Importer performs postprocessing when import of the XSD document is completed. During the postprocessing the XSD Importer merges some elements in the imported UML model.

The following merge rules are applied recursively (until merge is not possible) to the built model:

- A `Class` with the stereotype `<<complexType>>` is merged with a nested `Class` with the stereotype `<<comlexContent>>`.
- A `Class` with the stereotype `<<complexType>>` is merged with a nested `Class` with the stereotype `<<simpleContent>>`.
- A `Class` with the stereotype `<<complexType>>` is merged with a nested `Class` with the stereotype `<<all>>`.
- A `Class` with the stereotype `<<complexType>>` is merged with a nested `Class` with the stereotype `<<sequence>>`.
- A `Class` with the stereotype `<<complexType>>` is merged with a nested `Class` with the stereotype `<<choice>>`.

The merge between two elements `E1` and `E2` (`E1` owns `E2`) is possible if and only if:

- Both elements `E1` and `E2` do not have the `<<xmlNamespace>>` stereotype.
- Both elements `E1` and `E2` do not have the `<<xmlAttribute>>` stereotype.
- If element `E1` has any of the stereotypes `<<sequence>>,<<all>> or <<choice>>` and element `E2` has any of these stereotype then merge is not possible.
- Both elements `E1` and `E2` do not have comments.

XML 名前空間マッピング規則

Elements and attributes in a XML document may be placed in a XML namespace using the mechanisms described in the related specification [http://www.w3.org/TR/REC-xml-names/].

The Tau XML Framework supports XML namespace declarations by means of the special stereotype `<<xmlNamespace>>` which is defined in the profile `TTDXmlFramework`.

The stereotype `<<xmlNamespace>>` has the following attribute:

```
declarations : xmlNamespaceDecl[*]
```

The type `xmlNamespaceDecl` is defined as follows:

```
class xmlNamespaceDecl
{
    public Charstring name;
    public Charstring [0..1] prefix;
}
```

The instance of `xmlNamespaceDecl` with empty defines default namespace.

The XML namespace declarations on a XML element are mapped to one instance of the stereotype `<<xmlNamespace>>` according to the following rule:

XML:

```
<element
xmlns:prefix1="ns1"
xmlns:prefix2="ns2"
...
xmlns="default ns"/>
```

UML:

```
<<<xmlNamespace (. declarations =
{
xmlNamespaceDecl (. name = "ns1", prefix = "prefix1" .),
xmlNamespaceDecl (. name = "ns2", prefix = "prefix2" .),
...
xmlNamespaceDecl (. name = "ns1" .)
}
.)>>
```

XML UML スキーマモデリング

本章では UML を使って XML スキーマをモデリングする方法とスキーマ (XSD) ファイルのインポートとエクスポートの方法を説明します。

53

XML スキーマモデリング

本章では、UML での XML スキーマのモデリング方法を説明します。

はじめに

XML スキーマのモデリングを開始するには :

- Tau を起動して、新しい ［UML(XSD モデリング用)］プロジェクトを作成します。
- XML スキーマ図 (XML Sche) を作成し、モデリングを開始します。

スキーマモデルから XSD ファイルを生成するには :

- スキーマを右クリックして［XSD の生成］を選択します。
- フォルダを選択して、ファイル名を入力し、［保存］をクリックします。

参照

XMLFramework アドイン

XSD ビュー

XSD プロファイル

XSD ファイルのインポート

XSD ファイルの生成

XMLFramework アドイン

XMLFramework アドインは Tau における XML スキーマモデリングをサポートします。

アドインの主要機能は以下のとおりです :

- UML モデルに XSD 情報を追記するための XSD プロファイル
- XSD ビュー （XSD セントリックモデルビュー）

アドインをアクティブにするには、XMLFramework アドインの活動化 . を参照してください。

XMLFramework アドインの活動化

XMLFramework アドインをアクティブにするには :

1. [ツール] メニューから、[カスタマイズ] を選択します。
2. ［アドイン］タブをクリックして、`XMLFramework` アドインをチェックします。
3. ［OK］をクリックします。

XSD ビュー

XSD ビューはモデルの XSD セントリックなビューを提供します。このビューでは、XSD 要素のみが表示され、作成できます。

モデルビューのコンテキストメニューで［新規］コマンドを使用すると、適切な XSD 構造が表示されます。

XSD ビューをアクティブにするには :

1. ［表示］メニューで［モデルビューの再構成］を選択します。
2. ダイアログで［XSD View］を選択して［OK］をクリックします。

Standard View と XSD ビューはいつでも切り替えが可能です。

注記

XSD ビューでは一部の UML 要素が表示されません。すべての要素を表示するには Standard View に切り替えてください。

参照

第 1 章「Tau 4.3 の紹介」の 6 ページ、「モデル ビュー」

第 72 章「ダイアログ ヘルプ」の 2158 ページ、「デフォルトのモデル ビュー」

XSD プロファイル

XSD プロファイルは、XML スキーマのモデリングのために UML を拡張します。

プロファイルの xs パッケージは XML スキーマ仕様からのすべてのプリミティブ型を含んでいます。

XSD ファイルのインポート

XSD ファイルのインポートエクスポートについては、WSDL/XSD インポータリファレンス を参照してください。

XSD ファイルの生成

モデルからの XSD ファイルの生成方法は WSDL/XSD インポータリファレンス を参照してください。

注記

XSD ファイルは任意の UML パッケージから生成できます。コンテンツが XSD プロファイルからの XSD 情報で注釈されていない場合は、デフォルトのマッピングが使用されます。

UML モデルの探索

本章では UML エクスプローラの使用法を説明します。

54

Tau エクスプローラ

この章はエクスプローラのユーザー インターフェイス、および探索で使用される用語のリファレンス ガイドです。

注記

Tau エクスプローラは Windows でのみ使用可能なツールです。Windows Vista を使用している場合、エクスプローラ GUI との通信がポートを使用できるようにするため、実行ユーザーには管理者権限が必要です。

アプリケーションの探索

基本的な原理と用語

Tau エクスプローラは、*状態空間探索*という技術に基づいています。状態空間探索とは、分散システムを自動分析するための技術です。UML の状態空間探索ツールは、UML システムが到達可能な状態空間を自動生成する、という考え方に基づいています。

動作ツリー

Tau エクスプローラは「動作ツリー」という構造、つまり到達可能性グラフで実行されます。動作ツリーは、UML システムの動作（振る舞い）を表す樹状構造です。

ツリーのノードは UML の「システム状態」を表します。システム状態は次の項目によって定義されます。

- アクティブであるアクティブ クラス インスタンス
- そのアクティブ クラスの変数値
- アクティブ クラス インスタンスの UML コントロール フロー状態
- 任意の操作（ローカル変数などを使用）
- システムのキュー（待ち行列）に入っているシグナル（パラメータ付き）
- アクティブ タイマー
- その他

ツリー内のノード間のエッジは、UML システムをあるシステム状態から別のシステム状態に移行させる、アトミック UML イベントを表します。このことから、エッジは「動作ツリー遷移」とも呼ばれます。エクスプローラの構成方法によりますが、エッジはタスク、入力、出力などの各種 UML 文、または完全な UML 遷移でもかまいません。

したがって動作ツリーのサイズと構造はさまざまであり、エクスプローラ オプションの数によって決まります。エクスプローラのオプションによって、UML 遷移用に生成するシステム状態の数や、動作ツリー内の状態遷移の可能数が変わります。

状態空間探索

動作ツリーによって表されるシステム状態は、すべてシステムの「状態空間」といいます。動作ツリー内を移動することで、UML システムの振る舞いが探索でき、到達したシステム状態を調べることができます。これを「状態空間の探索」といいます。状態空間の探索は、手動または自動で実行できます。

注記

動作ツリー内のノードの「子」は、状態空間探索が実際にそのノードに到達するまで生成されません。つまり、ツリーは、エクスプローラの起動時に生成される静的構造ではありません。

状態空間探索中に到達した各システム状態について、UML システム内のエラーや問題点を検出するため、各種の「ルール」を調べます。ルールに違反している場合は、「レポート」が出力されます。レポートおよびレポートが生成された箇所のシステム状態を調べることによって、エラーの原因を判別できます。

状態と経路

システムの元の開始状態は「システム開始状態」といいます。システム開始状態とは、静的アクティブ クラス インスタンスが作成されたが最初の開始遷移が実行されなかった場合の、システム状態です。

「現在の状態」は、現在調査中のシステム状態です。現在の状態は、動作ツリー内を手動で移動した場合、またはレポートが生成されたシステム状態に移動した場合に、変更されます。最初は、システム開始状態に設定されています。

動作ツリーの「現在のルート」は任意のシステム状態をとることができます。エクスプローラのコマンドや機能は、現在のルートを操作の開始点として使用します。最初は、動作ツリーのシステム開始状態に設定されています。これは、「オリジナル ルート」とも呼ばれます。再定義を行った場合は、オリジナル ルートを再設定しない限り、動作ツリー内の現在のルートより上の状態には到達できなくなります。

動作ツリー内の 2 つの状態間の「経路」は、一連の整数によって示されます。各整数は経路内の 2 つの状態間の移行にどのような遷移が使用されたかを示しています。「現在の経路」は、動作ツリー内を手動で移動する場合、またはレポートが生成されたシステム状態へ移動する場合に、設定される経路です。設定された現在の経路は、現在のルートと現在の状態の間の経路となります。現在の経路は、エクスプローラが現在の経路の一部ではない状態に移動した場合、たとえば現在の経路以外のシステム状態に手動で移動した場合に、変更されます。ただし、現在の経路に沿って上下に移動した場合には変わりません。

状態空間自動探索の実行

ここでは、状態空間の自動探索とその結果を調べる方法について説明します。状態空間自動探索を使用する適用分野については、「UML システムの探索」で詳しく説明します。

エクスプローラでは、3 種類の状態空間自動探索が使用できます。それぞれ、異なるアルゴリズムとして実装されます。

- ビット探索：大規模な UML システムに適した効率的なアルゴリズム
- ランダム探索：非常に大規模な UML システムに使用できる単純なアルゴリズム
- 全探索：小規模な UML システムのみに適したアルゴリズム

各アルゴリズムの特徴の詳細については、それぞれのオプション ビューに関する説明、ビット探索ビュー、ランダム探索ビュー、および全探索ビューを参照してください。3 つのアルゴリズムに共通する特徴は以下のとおりです。

- 現在のシステム状態から開始する。したがって、探索を開始するために適切な開始状態へ移動する必要が生じる場合があります。

- 開始状態から特定の深さまでの状態空間を探索する。これによって、状態空間を無限に探索することを避けます。

状態空間探索の実行方法と結果は、状態空間がどのように構成されているかによって大きく変わります。詳細については、状態空間ビューを参照してください。

探索中にチェックされるルール

状態空間探索中、UML システム内のエラーや問題点を検出するため、いくつかのルールがチェックされます。ルールに従っている場合は、ユーザーに対してレポートが出力されます。

ルールは、通常 UML システムの予期せぬ振る舞いを原因とする、設計エラーの検出のために使用されます。エラーは、システムの別の場所において同時に発生するイベントによって発生することがよくあります。たとえば、タイマーが切れると同時にシステムの外部環境からシグナルを受信した場合などです。いわゆるシグナル競合は、多くの場合、エラー状況を引き起こす原因となります。

定義済みルールのほか、システムのその他のプロパティをチェックするためのルールを独自に定義できます。詳細については、Define-Rule を参照してください。

探索統計の解釈

出力される統計情報は、探索アルゴリズムの種類によって多少異なります。その中で注目すべき重要な統計情報は、以下のとおりです。

- `No of reports: x`
  検出されたエラー状況の数。
- `Truncated paths: x`
  探索の最大の深さに到達する回数。実行経路が打ち切られ、探索は別の状態において継続されます。この値が「0」より大きい場合、状態空間には探索されなかった箇所があります。ただし無限の状態空間を持つ UML システムでは、これは正常な結果です。
- `Collision risk: x`
  ビット探索の場合は、生成されたシステムの状態を表すハッシュ テーブルにおける衝突のリスク（パーセント）。ビット探索ビューを参照してください。この値は**非常に**小さい値（0-1%）でなければなりません。この値がそれより大きい場合は、ハッシュ テーブルのサイズを大きくする必要があります。衝突が起きると、実行経路が誤って打ち切られてしまうことがあります。
- `Current depth: x`
  探索が完了したまたは停止された時点での探索の深さ。この値が「-1」の場合、探索は完了しています。深さが「0」より大きい場合、探索は停止されています。「0」より大きい場合は、その深さから探索を継続できます。
- `Symbol coverage: x`
  探索中に到達した、システム内の UML シンボルの割合です。この値が 100 未満の場合、システムには探索されなかった箇所があります。

出力された統計情報に応じた対応については、「UML システムの探索」を参照してください。

エクスプローラの生成と起動

ここでは、モデル エクスプローラ アプリケーションのビルドと実行の方法について説明します。

重要！

実行形式のモデル エクスプローラ アプリケーションを生成するには、C/C++ コンパイラがインストールされている必要があります。また、使用するコンパイラは経路内に置く必要があります。

ビルド アーティファクトの作成

モデル バーリデータを有効にするには、[ツール] -> [カスタマイズ] -> [アドイン] を選択して、[ModelValidator] チェックボックスがオンになっていることを確認します。

モデル エクスプローラ アプリケーションをビルドするには、ビルド アーティファクト（REF MISSING）を作成する必要があります。[モデル ビュー] を右クリックしてコンテキスト メニューから [モデル エクスプローラ] -> [新しいアーティファクト] を選択してください。新しいビルド アーティファクトが作成されます。作成されたビルド アーティファクトを右クリックして [ビルド ルートの選択] を選択し、このビルドのビルド ルートとするモデル要素を設定します。

または、モデル ビュー内でビルド ルートを右クリックして、[モデル エクスプローラ] -> [新しいアーティファクト] を選択しても同じです。

エクスプローラ アプリケーションのビルドと起動

ビルド アーティファクトを右クリックして [ビルド (Model Validator)] を選択すると、次のオプションを持つサブメニューが表示されます。[チェック]、[生成]、[メイク]、[ビルド]、[起動]、[クリーン]。ビルドするためには、[ビルド] コマンドを選択してください。

ビルド アーティファクトのカスタマイズ

Model Verifier ビルド アーティファクトには、次のプロパティを設定できます。

- Target Directory：モデル ベリファイヤによって生成されたファイルの格納場所を指定します。
- Error Limit：ビルドに関連するエラーがこの上限を超えると、ビルドが中止されます。
- Target Kind：生成されたファイルにどのコンパイラを使用するかを指定します。
- Make-Template File：ユーザーが作成したテンプレート ファイルをインクルードするように make ツールに指示します。
- Supress C Level warnings：C コンパイラおよびリンカーからの警告を出さないようにします。
- Generate reference package：このプロパティを設定すると、生成されたファイルが「Result of C generation」という名前のパッケージ内のモデルに追加されます。

- TCP/IP Port：Tau がエクスプローラ アプリケーションとの通信に使用するポートを指定します。
- Launch Console：このプロパティを設定すると、エクスプローラ UI から送信されたコマンドおよびエクスプローラ アプリケーションからの応答を反映した、読み取り専用のコンソール ウィンドウが開きます。
- Additional Preprocessor Defines：追加のコンパイル スイッチを設定できます。-DUSERDEF1 のようなコンパイラ コマンド ラインの構文で指定します。

注記

ビルド アーティファクトからプロパティを取り除くには、コンテキスト メニューから［値の削除］を選択します。ほとんどの場合、プロパティは空白でも有効です。プロパティが設定されていないと、入力フィールドは黄色で示されます。

エクスプローラのユーザー インターフェイス

エクスプローラ ユーザー インターフェイス（エクスプローラ UI）は、Validator ビルド アーティファクトの「起動」コマンドを実行するか、Tau の［エクスプローラ］メニューから［モデルエクスプローラの表示］を選択すると表示されます。

注記

ファイル システムからはエクスプローラ UI を開くことはできません。Tau からのみ開くことができます。

エクスプローラの状態

エクスプローラ UI を起動すると、自動的にエクスプローラ実行形式ファイルの状態を検出し、その状態を常に追跡します。エクスプローラ UI の状態は、以下の 4 種類の状態のいずれかです。

- エクスプローラは実行されていません：エクスプローラ UI は無効であり、グレー表示になっています。
- エクスプローラは起動中です：この状態は Tau アイコンの点滅によって示されます。
- エクスプローラはコマンド待機中です：エクスプローラ UI は有効です。この状態は、標準の Tau アイコンで示されます。
- エクスプローラはコマンド処理中です：この状態は Tau アイコンの回転によって示され、エクスプローラ UI は無効です。この状態ではエクスプローラ コマンド プロンプトからのコマンド送信、または［中断］オプションと［終了］オプションの選択のみが可能です。

エクスプローラ ビュー

エクスプローラ UI は、一連のビューに基づいています。これらのビューは、メニューからアクセスできます。

- エクスプローラ ビュー
- ナビゲータ ビュー
- レポート ビュー

- テスト値ビュー
- 一般オプション ビュー
- ビット探索ビュー
- レポート ビュー
- 状態空間ビュー
- ランダム探索ビュー
- 全探索ビュー
- 樹状探索ビュー

エクスプローラ コマンド プロンプト

どのビューからも、エクスプローラ コマンド プロンプトを使用してエクスプローラにコマンドを送信できます。コマンド プロンプトはメニューの下にあります。コマンドを入力し、Enter キーを押して（または［コマンド送信］ボタンをクリックして) エクスプローラにコマンドを送信します。コマンドの実行結果は通常、ログ ウィンドウで確認できます。

ログ ウィンドウ

エクスプローラ実行形式ファイルから送受信されるすべての情報は、Tau の出力タブ、［Model Validator］に表示されます。

エクスプローラ ビュー

このビューには、4 種類の探索を選択できるサブメニューがあります。

- ビット探索：コマンド Bit-State-Exploration を参照してください。
- ランダム探索：コマンド Random-Walk を参照してください。
- 全探索：コマンド Exhaustive-Exploration を参照してください。
- 樹状探索：コマンド Tree-Search を参照してください。

いずれかのオプションを選択すると、対応するエクスプローラ コマンドが Tau 経由で非同期にエクスプローラ実行形式ファイルに送信されます。

ナビゲータ ビュー

エクスプローラには、UML システムの動作ツリー内を対話形式で探索する機能が用意されています。この機能は、状態空間内の手動 *ナビゲーション* とも呼ばれます。

ナビゲータは、以下の 3 種類の状況で使用できます。

1. エクスプローラなどの状態空間探索ツールの操作を学習する場合など、ナビゲータは UML システムの動作ツリーを対話形式で調べるための便利なツールとなります。
2. 状態空間自動探索を使用する場合は、UML システムのシステム開始状態以外の開始点から探索を開始する必要が生じることがあります。このような場合、ナビゲータを使用すれば、自動探索を開始できる適切なシステム状態へ移動できます。
3. 自動探索中に生成されたレポートを調べる場合は、ナビゲータを使用すれば、レポートが生成された状況への経路において代替可能な動作をチェックできます。

動作ツリー内で上に移動

動作ツリー内で 1 つ上のレベルに移動するには、前の状態を記述しているリンクを選択します。このリンクは［上へ］見出しのすぐ下にあります。エクスプローラ実行形式ファイルが動作ツリー内の最初のノードにある場合、使用可能なリンクはなく、［ノードなし］と表示されます。この場合、エクスプローラ UI はエクスプローラ UI に「Up 1」コマンドを送信します。このコマンドを送信した後で、ナビゲータ ビューはいくつかのコマンドを送信して、新しい上の状態が何になるか、次に使用可能な状態は何かを調べます。

動作ツリー内で下に移動

動作ツリー内で下のレベルに移動するには、［下へ］見出しの下にあるリンクの 1 つを選択します。現在の状態の次に何も状態がない場合、［下へ］リストは空白になります。［下へ］からいずれかのリンクを選択すると、エクスプローラ UI はコマンド「Next x」を送信します。ここで x は、選択されたリンクに対応する使用可能な［下へ］のリスト内の添字です。このコマンドを送信した後で、ナビゲータ ビューはいくつかのコマンドを送信して、新しい上の状態が何になるか、次に使用可能な状態は何かを調べます。

モデルで検索

［モデルで検索］オプションを選択すると、エクスプローラ UI は［エクスプローラシステム状態］タブ内に要素のリストを表示します。各要素は、現在の状態空間の一部です。このリスト内の要素をクリックすると、それに対応する要素が検索されます。通常はダイアグラムが開き、指定された要素が選択されます。

コマンド プロンプトを使用して状態空間内を移動することももちろん可能です。このために使用できるコマンドがいくつかあります。コマンドのアルファベット順リストを参照してください。

シーケンス図トレース

［シーケンス図トレース］を選択すると、Tau に新しいタブが開き、状態空間の現在の位置に導いたイベントを示すシーケンス図が表示されます。このタブは、レポートが生成されたイベントを理解するために、非常に役立ちます。シーケンス図は、Generate-SQD-Trace コマンドで開くこともできます。

注記

コマンド プロンプトを使用してコマンドを送信しても、エクスプローラ UI の表示は更新されません。したがって、ナビゲータ ビューを表示しているときにテキスト ナビゲーション コマンドを使用すると、ナビゲータ ビューの表示が最新ではなくなります。これを回避するため、テキスト コマンドを実行した後で、ナビゲータ ビューで遷移を行う前にエクスプローラ UI の表示を更新することを強く推奨します。最も簡単に表示を更新するには、メニューにある［Navigate］オプションをクリックします。この方法はどのビューにも当てはまりますが（エクスプローラ実行形式ファイルにコマンドを送信したときは、どのビューでも表示が更新されません）、ナビゲータ ビューで最も顕著です。

レポート ビュー

探索の実行後、レポート ビューに UML モデル内で見つかった問題のリストが表示されます。いずれかのレポートをクリックすると、エクスプローラ実行形式ファイルはレポートされた問題が発生した状態に移動し、エクスプローラ UI はナビゲータ ビューを表示します。この手法で、特定の問題のデバッグを行うことができます。エクスプローラ実行形式ファイルがレポートを何も検出しなかった場合、このビューは空白です。各レポートには［トレースの表示］オプションがあります。このオプションを選択すると、Tau に新しいタブが開き、レポートが生成されたイベントを示すシーケンス図が表示されます。

テスト値ビュー

テスト値ビューには定義されたテスト値が表示され、テスト値の削除や新しいテスト値の追加を行うことができます。テスト値には以下の 3 種類があります。

- シグナル
- 値
- パラメータ

テスト値ビューが選択されると、エクスプローラ UI は一連のコマンドを送信して、定義されているテスト値を調べます。これは、テスト値を追加または削除した後にも行われます。定義されたテスト値は、次に「シグナル定義」、「テスト値」、および「パラメータ テスト値」の各リストに表示されます。

シグナル テスト値の追加

シグナルにパラメータが何もない場合は、［シグナル追加］セクションにシグナル名を入力してシグナルを追加します。シグナルにパラメータがある場合は、セパレータとして空白を使って、各パラメータに値を入力する必要があります。その例を以下に示します。

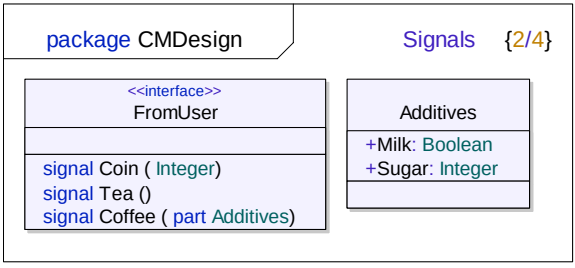
```
Coffee true 1
```

値テスト値の追加

値を追加するには、［値追加］セクションの最初のテキスト フィールドに値を追加し、2 番目のフィールドにその型を指定します。

パラメータ テスト値の追加

特定のシグナルのパラメータ テスト値を追加するには、3 番目のテキスト フィールドにシグナル名、2 番目のテキスト フィールドにパラメータ番号、最初のテキスト フィールドにパラメータの値を入力します。その例を以下に示します。



値：

```
true 2
```

パラメータ：

```
1
```

シグナル：

```
Coffee
```

テスト値の削除

テスト値を削除するには、定義されたテスト値の隣にある［削除］リンクを選択します。

特定の型のすべてのテスト値を削除するには、その型の見出しの隣にある［すべてを削除］リンクを選択します。

一般オプション ビュー

［オプション］メニューを選択すると、このビューが表示されます。このビューが、すべてのオプション ビューの最初のビューです。一般オプション ビューは、［一般］サブメニュー オプションを選択したときにも表示されます。

一般オプション ビューからは、以下のオプションを実行できます。

- 詳細設定
- オプションをデフォルトに設定
- すべてのオプションをリセット
- すべてのオプションを表示

詳細設定

［詳細設定］は、エクスプローラ実行形式ファイルに以下のコマンドを送信します。

```
Define-Scheduling All
```

```
Define-Priorities 1 1 1 1 1
Define-Max-Input-Port-Length 2
Define-Report-Log MaxQueuelength Off
```

これらのオプションを設定する理由は、以下のとおりです。

- ここではシグナル競合を探索しており、シグナル競合状態 の特性が内部イベントの順序に依存しているため、スケジューリングを［すべて］に設定する必要がある。
- すべてのタイプのイベントに対して優先度を「1」に設定する必要がある。
- 状態空間のサイズを減らすため、最大キュー長を非常に小さい数に設定する必要がある。なぜなら、外部環境がシステムにいつでもシグナルを送信できる場合、外部環境からシグナルを受信できるキューが急速に大きくなるからです。
- これらのオプションによって最大キュー長レポートが多数生成されるため、このレポートのレポート グはオフに設定しておく必要があります。なお、このレポートのレポート アクションは［枝刈り］（デフォルト）に設定してください。

オプションをデフォルトに設定

コマンド Default-Options を参照してください。

このコマンドを送信する前に、エクスプローラ UI は確認ダイアログを表示します。確認ダイアログで、コマンドを中止できます。

すべてのオプションをリセット

コマンド Reset を参照してください。

このコマンドを送信する前に、エクスプローラ UI は確認ダイアログを表示します。確認ダイアログで、コマンドを中止できます。

すべてのオプションを表示

コマンド Show-Options を参照してください。

ビット探索ビュー

ビット探索は、大規模な UML システムに適した効率的な状態空間自動探索アルゴリズムです。状態空間で深さを優先した探索を行い、探索中に横断した状態をビット配列を使用して保存します。

探索中に新しいシステム状態が生成されると、システム状態から 2 つのハッシュ値を算出します。以下のように、ビット配列がチェックされます。

- ハッシュ値が示す位置の両方がすでに設定されている場合は、以前にその状態が探索されたと考えられます。状態空間の特定経路の探索が枝刈りされると、直前のシステム状態に戻り、他の場所で探索が継続されます。
- 両方の位置が設定されていない場合は、以前に探索されたことがない新しい状態です。両方の位置がビット配列に設定され、後続の状態で探索が継続されます。

ビット探索ビューからは、以下のオプションを設定できます。

- 反復ステップ
- 探索の深さ

- ハッシュ サイズ

反復ステップ

デフォルト値は「0」、つまり、機能は無効です。

コマンド Define-Bit-State-Iteration-Step も参照してください。

探索の深さ

探索の深さは、状態空間内でエクスプローラが特定の実行経路を探索する最大の深さです。この深さに到達すると、探索は打ち切られ、直前のシステム状態に戻ります。

デフォルト値は「100」です。

コマンド Define-Bit-State-Depth を参照してください。

ハッシュ サイズ

ハッシュ テーブルとして使用されるビット配列のサイズは、ビット探索の振る舞いを定義する重要な要因です。なぜなら、新しい状態のハッシュ値を以前のハッシュ値と比較して新しい状態をチェックするときに、衝突というリスクがあるからです。ハッシュ テーブルが大きいほど、衝突のリスクは小さくなります。

デフォルト値は「1,000,000」（バイト）です。

コマンド Define-Bit-State-Hash-Table-Size も参照してください。

レポート ビュー

レポート タイプごとに、レポート検出時にどのような動作を実行するか、またレポートをユーザーに送るべきかを定義できます。

レポート アクション

レポート アクションは、状態空間探索中にレポート状況が発生したときにどのようなアクションをとるべきかを決定します。以下の 3 とおりの可能性があります。

- 継続：レポート状況が発生しなかったかのように探索を継続します。
- 枝刈り：探索は枝刈りされ、アルゴリズムに応じて適切な動作がとられます。たとえばビット探索を行っている場合、探索の最大の深さに到達して探索が打ち切られたときと同じように、探索は 1 つ前の状態に戻り次の別の遷移で継続されます。
- 中止：探索は中止され、コマンド プロンプトが表示されます。

デフォルト値は、どのレポート タイプの場合も［枝刈り］です。

コマンド Define-Report-Continue、Define-Report-Prune、Define-Report-Abort も参照してください。

注記

レポートのタイプによっては、たとえばデッドロックの場合、［継続］は選択できません。

レポート ログ

レポート ログ設定は、生成されたレポートのリストにレポートを記録するかどうかを決定します。特定レポート タイプに対してレポート ログをオフに設定すると、そのタイプのレポートはレポート リストに表示されません。ただし、レポートが記録されない場合でもレポート アクションは実行されます。

デフォルト値はどのレポート タイプの場合もオンです。

コマンド Define-Report-Log も参照してください。

状態空間ビュー

UML システム用に生成できる状態空間の構造とサイズは、状態空間オプションを使用してさまざまな方法で変更できます。デフォルト値は、できるだけ多くのアプリケーションでエクスプローラがすぐに有効になるように、状態空間をできるだけ小さくする設定になっています。この設定では、エクスプローラが行う探索は、その可能性と比べてかなり小規模なものとなります。このような探索では、到達不可能な部分でのみ発生するエラー状況は、見逃されることになります。

状態空間ビューからは、以下のオプションを設定できます。

- シンボル実行時間
- 最大状態数
- タイマー追加
- 最大インスタンス数
- 遷移の長さ
- 入力ポートの長さ
- イベント優先度
- スケジューリング
- 遷移

シンボル実行時間

UML システムの分析で一般的に行われる簡素化は、シンボルの実行（アクションまたは出力など）に要する時間をゼロと見なすことです。この時間はもちろん実際のシステムではゼロではありませんが、多くの場合、システム内のタイマー継続時間に比較すると非常にわずかな時間であり、システムの分析時には無視できます。

アクティブ クラスが持続時間 5 のタイマーを設定し、長時間を要する長いループなどを実行した後、持続時間 1 のタイマーを設定する例を考えてみましょう。シンボル実行時間がゼロと見なされると、2 番目のタイマーが必ず先に切れます。ゼロではないと考えた場合は、2 つのタイマーのいずれか一方が先に切れます。

エクスプローラでは、このオプションを使用して、UML シンボルの実行時間を［ゼロ］とするか、［未定義］とするかを選択できます。

このオプションのデフォルト値は「ゼロ」です。

コマンド Define-Symbol-Time も参照してください。

最大状態数

エクスプローラが状態空間を探索するとき、システム状態の保存に内部バッファが使用されます。このバッファのサイズにより、エクスプローラが扱えるシステム状態の最大数が決まります。

デフォルト値は「100000」（バイト）です。

コマンド Define-Max-State-Size も参照してください。

タイマー追加

エクスプローラで実行できる 1 つのテストは、非進捗ループ（つまり状態空間内の進捗していないループ）の探索です。このテストの目的は、UML システムが内部通信中で、外部オブザーバに対しては無効に見える状況を探索することです。

このオプションは、非進捗ループのチェック時にタイマーのタイムアップを進捗と見なすかどうかを定義します。

デフォルト値はオンです。

コマンド Define-Timer-Progress も参照してください。

最大インスタンス数

状態空間内の作成動作の無限連鎖を避けるため、エクスプローラはすべての型についてアクティブ クラス インスタンスの最大数を使用します。状態空間探索中にこの最大数を超えると、「インスタンス作成」レポートが生成されます。

デフォルト値は「100」です。

コマンド Define-Max-Instance も参照してください。

遷移の長さ

状態空間の遷移内の無限ループを検出できるようにするため、1 つの遷移内で実行可能な UML シンボルの最大数を定義します。状態空間探索中にこの最大数を超えると、「MaxTransLen」レポートが生成されます。

デフォルト値は「1000」です。

コマンド Define-Max-Transition-Length も参照してください。

入力ポートの長さ

エクスプローラでは入力ポート キューの長さは無限ではありません。キューが無限に大きくなる場合は、実際問題として設計エラーの可能性があるからです。状態空間探索中にキューの長さが定義された最大長を超えると、「MaxQueuelength」レポートが生成されます。

デフォルト値は「3」です。

コマンド Define-Max-Input-Port-Length も参照してください。

イベント優先度

動作ツリーに表示されているイベントは、以下の 5 つのクラスに分類できます。

- 内部イベント：システム内のアクティブ クラスに固有のイベント（タスク、分岐、入力、出力など）。

- 外部環境からの入力 : 外部環境からのシグナルの受信。このシグナルはアクティブ クラス インスタンスの入力ポートまたはコネクタ キューに入る。
- タイムアウト イベント :UML タイマーのタイムアウト。タイマー シグナルはアクティブ クラス インスタンスの入力ポートに入る。
- コネクタ出力 : シグナルはコネクタ キューから削除され、別のコネクタ キューまたはアクティブ クラス インスタンスの入力ポートに入る。
- 自発遷移 : 入力なしに発生したアクティブ クラス内の遷移。

これらのイベント クラスのそれぞれに対して、優先度 1、2、3、4、または 5 を割り当てます。これらの優先度は、状態空間探索中に各システム状態から生成すべき遷移を決定するために使用されます。優先度 1 のイベントが最初に考慮されます。現在の状態で優先度 1 のイベントが何も可能ではない場合に限り、優先度 2 のイベントが考慮されます。現在の状態で優先度 1 または 2 のイベントが何も可能ではない場合に限り、優先度 3 のイベントが考慮されます。

注記

なお、[シンボル実行時間] オプションの設定も、各システム状態で実行可能なイベントに影響することに注意してください。シンボル実行時間を参照してください。

イベント クラスに優先度を割り当てる、最も一般的な方法は、次の 2 つの方法です。

- すべてのイベント クラスに優先度 1 を割り当てる。
- 内部イベントとチャンネル出力に優先度 1 を割り当て、外部、タイムアウト、および自発遷移イベントに優先度 2（デフォルト）を割り当てる。

最初の選択肢は、異なるイベントに対してタイム スケール（時間の尺度）を何も想定できない状況に適しています。2 番目の選択肢は、外部環境の実行速度とタイムアウト期間に比べて、内部遅延が非常に短い状況に適しています。

自発遷移優先度のデフォルトは「2」です。

コネクタ出力優先度のデフォルトは「1」です。

タイムアウト イベント優先度のデフォルトは「2」です。

ENV 入力優先度のデフォルトは「2」です。

内部イベント優先度のデフォルトは「1」です。

コマンド Define-Priorities も参照してください。

スケジューリング

スケジューリング アルゴリズムは、システム状態のどのクラス インスタンスの実行が許可されるかを定義します。以下の 2 つの選択肢があります。

- レディ キュー内のすべてのアクティブ クラス インスタンスの実行を許可する（コマンドの値 [すべて]）。
- レディ キュー内の最初のアクティブ クラス インスタンスのみ実行を許可する（コマンドの値 [最初]）。

デフォルト値は [最初] です。

コマンド Define-Scheduling も参照してください。

遷移

状態空間探索の動作ツリー遷移のタイプとして、以下の 2 つの選択肢があります。

完全な UML クラス グラフ遷移と等しい（コマンドの［UML］）。

UML クラス グラフ遷移の一部（コマンドの値［シンボル順］）。

UML クラス グラフ遷移と等しい場合は、遷移が開始されると、別の事象の発生を許可する前に遷移が完了します。このことは、動作ツリー内のすべてのシステム状態にあるすべてのアクティブ クラスが、常に UML クラス グラフ状態にあることを意味します。

UML プロセス グラフ遷移の一部にすぎない場合は、動作ツリー内の遷移は、アクティブ クラス インスタンスに固有な一連のローカル イベントであり、その後に非ローカル イベントが続くと考えられます。ローカル イベントには、タスクや分岐などがあります。非ローカル イベントには、他のクラス インスタンスへの／からのシグナルの作成や入力 / 出力などがあります。

デフォルト値は［UML］です。

コマンド Define-Transition も参照してください。

ランダム探索ビュー

ランダム探索は、非常に大規模な UML システムに適した、状態空間自動探索アルゴリズムです。ランダム探索は、ランダムに実行する遷移を選択することにより、状態空間で深さを優先した探索を実行します。

このような「ランダム探索」中に探索が最大の深さに到達すると、探索は元の状態から再開され、新しいランダム探索が実行されます。ただし、探索済み経路の再探索を回避する仕組みはありません。つまり、システム状態が何度も探索される可能性があります。

ランダム探索ビューからは、以下のオプションを設定できます。

- 探索の深さ
- 繰り返し数

探索の深さ

探索の深さは、探索が枝刈りされるまでの遷移の実行回数を決定します。この回数の遷移が実行されると、探索は枝刈りされ、また最初から再開されます。

デフォルト値は「100」です。

コマンド Define-Random-Walk-Depth も参照してください。

繰り返し数

開始状態からこの回数だけランダム探索を繰り返すと、探索が終了します。

デフォルト値は「100」です。

コマンド Define-Random-Walk-Repetitions も参照してください。

全探索ビュー

全探索は、正確性の要求が非常に高い小規模な UML システムを対象とした、状態空間自動探索アルゴリズムです。

このアルゴリズムは、ビット探索と同じように、状態空間内で深さを優先した探索を行いますが、衝突のリスクを伴いません。なぜなら、横断済みのシステム状態がすべてメイン メモリに記憶され、探索中に新たに生成されたシステム状態が探索済みであるかどうかを常に判断できるからです。

このアルゴリズムの欠点は、横断済み状態をすべて記憶しておくために大量のメインメモリを必要とすることです。このため、このアルゴリズムを適用できる UML システムの複雑性が制限されます。

探索の深さ

探索の深さは、状態空間内でエクスプローラが特定の実行経路を探索する最大の深さです。この深さに到達すると、探索は打ち切られ、直前のシステム状態に戻ります。

デフォルト値は「100」です。

コマンド Define-Exhaustive-Depth も参照してください。

樹状探索ビュー

樹状探索は、すべての組み合わせ可能なアクションが実行される、状態空間自動探索アルゴリズムです。

探索の深さ

探索の深さは、状態空間内でエクスプローラが特定の実行経路を探索する最大の深さです。この深さに到達すると、探索は打ち切られ、直前のシステム状態に戻ります。

デフォルト値は「100」です。

コマンド Define-Tree-Search-Depth も参照してください。

モデル探索のガイドライン

UML システムの探索

ここでは、エクスプローラの自動状態空間探索機能を使用して、UML システムの矛盾と設計エラーを探す方法について説明します。探索の主な目的は、アプリケーションの信頼性、および予期しない状況に対する反応をテストすることです。基本的に、探索とは以下のような質問に答えることです。

- 設計者が想定した順序で、ユーザーがボタンをクリックしなかった場合はどうなるか。
- アプリケーションをサポートする、オペレーティング システムのスケジューリング アルゴリズムが変更されたらどうなるか。
- タイマーが切れると同時に、外部環境がシステムに入力を送るとどうなるか。

および、設計者が想定し得なかったその他の質問。

ここでの説明は、中程度の大きさと複雑性を持つ UML システムを想定しています。大規模な UML システムの 探索については、大規模なシステムの探索を参照してください。

デフォルトの探索の使用

エクスプローラを使用して、新規 UML システムのエラーを初めて探すときには、デフォルトのオプションを使用してビット探索を行うことを推奨します。

エクスプローラで開いたシステムの 探索を行うには、以下の手順に従います。

1. 開いたエクスプローラのコマンドをすでに実行している場合、エクスプローラをリセットします。Reset コマンドを入力するか、エクスプローラ UI の右上にある［リセット］ボタンをクリックします。
2. また、デフォルトの状態空間および探索オプションを使用していることも確認します。Default-Options コマンドを入力するか、オプション ビューで［オプションをデフォルトに設定］オプションをクリックします。
3. エクスプローラ ビューで［ビット探索］をクリックして、ビット探索を開始します。探索を少なくとも 5 ～ 10 分間続けます。
4. 探索が完了すると、モデル エクスプローラの出力ウィンドウに統計情報が出力されます。シンボル カバレッジの値に注目してください。
5. レポート ビューに移動して、レポートが作成されているか確認します。レポートをクリックすると、動作ツリー内のそのノードに移動します。任意の内観コマンド（List-Active-Class、List-Input-Port、Examine-Variable など）を使用して、特定の状態の詳細情報を得たり、トレースおよびビュー機能を使用してレポートを調べたりできます。まず、［トレースの表示］オプションを選択して、レポートが作成されたイベントを示すシーケンス図を開くことを推奨します。この操作は、Generate-SQD-Trace コマンドを使用して行うこともできます。

6. システムのエラーを発見したら、その場で修正することもできます。その場合、修正したシステムの新しいエクスプローラを生成して、上記で説明した探索の手順に戻ります。または、探索を完了したとみなすかどうか確認します（下記を参照）。

探索完了の判断

すべてのレポートをチェックし、検出したエラーを修正した後で、次の質問が生じます。システムの 探索はいつ完了するか。この質問に答えるには、以下の項目を調べます。

- 自動探索後の統計に、どのようなシンボル カバレッジが報告されたか。
- 探索は自動的に終了したか、ユーザーが停止したか。

ここでは以下の可能性があります。

1. シンボル カバレッジは 100%、探索は自動的に終了した。

すべてのシンボルが実行され、可能なアクションのほとんどの組み合わせでテストされました。この場合、探索を続ける意味はなく、完了したとみなすことができます。

ただし、探索が中断されたり、ハッシュ テーブル内で衝突が起きた可能性があり、状態空間探索の構成を変えることで組み合わせが増えることもあるので、アクションのすべての組み合わせがテストされたわけではありません。必要なら、エクスプローラ オプションを変更して、新しい探索を開始できます。

2. シンボル カバレッジは 100%、ただし探索は手動で停止された。

この場合は、自動的に終了するまで、探索を続けることを推奨します。まだ実行されていない可能なアクションの組み合わせが残っているため、さらにレポートが生成される可能性があります。

3. シンボル カバレッジが 100% 未満だった。

システム内に、探索中に一度も到達していない部分がある。この場合は、探索が自動的に終了したとしても、探索が完了したとみなすことはできません。次にその理由と低いシンボル カバレッジの解決方法を説明します。

シンボル カバレッジが低い場合の対応

探索後のシンボル カバレッジが 100% であれば、システムのすべての部分が一度は実行されています。シンボル カバレッジが 100% 未満の場合は、状態空間の一部に到達していない可能性があります。以下のような理由が考えられます。

- すべてのシンボルに到達する前に、探索が手動で停止された。

この場合は、自動的に終了するまで探索を続けます。

- テスト値が不適切だった。

テスト値は、外部環境からの可能なシグナル類を定義するために使用します。自動的に生成されたテスト値は、すべての UML システムに適合するわけではありません。たとえば、このテスト値によって、分岐文の 1 つのブランチが一度も実行されない可能

性があります。この問題を解決するには、適切なシグナル パラメータに対応するテスト値を再定義します。テスト値の詳細については、外部環境からのシグナルの定義を参照してください。

- レポート後に、探索が枝刈りされた。

通常、エクスプローラは、レポートが見つかると直ちに特定パスの探索の「枝刈り」を行います。つまり、当該状態以下の探索は継続されません。ユーザーがレポートを調べ、レポートに対する処理を何も行わないと判断したとしても、エクスプローラは次にレポートを見つけるとまた探索を枝刈りします。この問題を解決するには、この種のレポートについて、オプション ビューのレポート サブビューで、レポート アクションを［枝刈り］から［継続］に変更します。

- システムの一部が、実質的に到達不可能。

UML システムの一部がまったく到達不可能な場合、システムの設計エラーが考えられます。

- タイマーのタイムアップに問題がある。

エクスプローラは、デフォルトで、状態空間のサイズを減らすように構成されています。タイマーが切れる前に、内部アクション（たとえば、タスク、分岐、内部入出力）を実行しようとします。どのタイマーが切れるよりも前に、必ずシステムが実行することを想定しています（タイマーは、外部環境からの入力を待ってタイムアップする可能性もあります）。

- 探索の深さが足りない。

デフォルトの探索の深さは 100 です。たとえば、初期化フェーズが非常に長い場合など、システムによってはこの深さでは不十分なことがあります。単にオプション ビューでその探索方法の探索の深さを増やすだけで、この問題を解決できる場合もあります。

- 探索空間が大きすぎる。

複雑性が高い UML システムは、非常に大きな状態空間を持っている場合が多く、一度ですべてを探索することは不可能です。この特徴として、シンボル カバレッジが低く、経路の中断、探索の手動停止や高い（10% より高い）衝突リスクなどが生じることがあります。このようなケースについては、大規模なシステムの探索で説明しています。

詳細探索の使用

状態空間探索のデフォルトのオプション、特に状態空間の構造を定義するオプションは、システムの最初の探索で良い結果を出せるように最適化されています。これらは、まず UML システムの内部の矛盾をテストして、カバレッジを上げることを目的としています。デフォルト オプションは、かなり「良好」な外部環境、つまり、システムで内部的に何も起こり得ないときにのみシグナルを送るような環境を想定しています。

これは高いカバレッジを保ちながら、状態空間のサイズを減らす効果があります。欠点は、ある種のエラー状況がまったく検知されないことです。デフォルト オプションではまったく検知されないエラーには、外部環境から送られたシグナルによるシグナ

ル競合や、同時に発生するタイマーのタイムアップなどがあります。たとえば、2 つの接続要求が同時に異なるアクセス ポイントに送られた場合に、通信プロトコルがシステム状態の矛盾を引き起こすケースがあります。

この種のエラーを検出するには、オプションを変更して、UML システムの探索をやり直す必要があります。適切なオプション設定を「詳細オプション」といいます。オプションにこれらの値を使用すると、通常、UML システムの状態空間は非常に大きくなります。したがって、大規模なシステムの探索で説明しているテクニックを駆使しても、状態空間の完全なカバレッジを得ることはできません。良い結果を得るためには、状態空間の探索時に、ランダム探索アルゴリズムを使うのがもっとも良い方法です。詳細については、「ランダム探索の使用」を参照してください。

詳細オプションを設定するには、オプション ビューの［詳細設定］ボタンをクリックします。

大規模なシステムの探索

ここでは、大規模な UML システムの設計および探索に便利な、各種のテクニックを説明します。大規模なシステムとは、一度の自動状態空間探索では、完全に探索できない大きさの状態空間を持つシステムのことです。ここで説明するテクニックは、実用的で、状態空間全体を探索しなくてもエラーを見つけられることを目的としています。

以下のテクニックを説明します。

- 分解探索
- 効率的なビット探索
- 状態空間サイズの削減
- *ランダム探索の使用*
- インクリメンタル 探索

分解探索

分解探索を使用する目的は、1 つの大きな探索の代わりにいくつかの小さな探索を使用することです。多くの場合、UML システムの振る舞いは、いくつかの「フェーズ」または「機能」に分けることができます。これらの各フェーズまたは機能を、別々に探索することが分解探索の目的です。この方法の利点は、すべてのフェーズの組み合わせを探索するより、異なるフェーズを個別に探索する方がはるかに簡単なことです。欠点は、異なるフェーズまたは機能間の相互作用が原因となるエラーが見つからないことです。しかし、大規模な UML システムでは、最低でも完全なシンボル カバレッジを得るためには、これが唯一の方法の場合があります。

必要な部分探索の種類と数を求めるプロセスは、部分探索の対象となる可能性のある機能やフェーズを特定するための、反復プロセスと計画案件の組み合わせとなります。インクリメンタルな設計プロセスを使用する場合、異なる反復作業を使用して、部分探索の選択の指針とすることができます。インクリメンタル 探索と比較してください。

以下に、必要な部分探索を見つけるために使用する、一般的な方法を示します。

1. システム開始状態から探索を開始します。
2. すべてのレポートをチェックして、システムのエラーを修正します。新しいエクスプローラを生成して、再び探索を行います。
3. レポート結果をすべて修正したら、シンボル カバレッジをチェックします。カバレッジが 100% であれば、探索は完了です。それ以外の場合は次のステップに進みます。
4. 適切なシステム状態に移動して、その状態から新しい探索を開始します。
5. シンボル カバレッジが 100% になるまで、この手順を繰り返します。

この方法には、次の 2 つの問題があります。

- 各部分探索をどこから開始するか。
- 各部分探索をどのように制限するか。

部分探索をどこから開始するか

新しい探索をどこから開始するか確認するには、当然システムに依存し、UML システムの知識を必要とします。通常は、シーケンス図で適切なレポートを調べ、開始点を探す手助けとするとよいでしょう。Generate-SQD-Trace コマンドまたはレポート ビューの［トレースの表示］オプションを使用します。システム状態を選択したら、次はエクスプローラでどのようにその状態に到達するかです。これにはいくつかの方法があります。

- ナビゲータ ビューを使用する

  ナビゲータ ビューのボタンを使用して、対象状態にナビゲートします。さらに、Command-Log-On コマンドを使用して、ナビゲーションをファイルに保存できます。Command-Log-Off でログを閉じます。保存したコマンドは、Include-File コマンドを使用してロードできます。
- Path コマンドを使用する

  対象状態で、Print-Path コマンドを使用して、ルートから現在の状態へのパスを出力します。後で、先のコマンドで出力したパスに従って、Goto-Path コマンドを使用できます。
- MSC ファイルを使用する

  対象状態で、MSC-log-file < ファイル名 > コマンドを使用して、現在の状態へのトレースを保存します。前の状態に戻るには、Navigate-To-MSC < ファイル名 > コマンドを使用して、指定した状態に戻ります。
- ユーザー定義ルールを使用する

  対象状態の状態や他の条件を記述するユーザー定義ルールを作成します。「ユーザー定義ルールの管理」および「ユーザー定義ルール」を参照してください。探索を使用して、この状態のレポートを出力します。上記の、Path コマンドまたは MSC コマンドを使用できます。

注記

モデルを変更した場合は、Path コマンドおよび MSC-trace ファイルが無効になり、やり直さなければならないことがあります。

部分探索をどのように制限するか

次の問題は、各部分探索を状態空間の意図した部分に限定することです。探索の範囲の制御に使用できる要因がいくつかあります。

- 探索の深さ
- 外部環境からのシグナル
- ユーザー定義ルール

探索の深さは、最も使い易い制限要因です。探索の深さを、たとえば 10 または 20 に減らすことにより、探索の規模は、デフォルトの深さの 100 と比較して大幅に小さくなります。

外部環境から送られるシグナルのリストを変えることにより、システムのどの部分を探索するかを制御できます。たとえば、接続指向プロトコル仕様のデータ転送フェーズをテストしたい場合は、以下の方法が適しています。

- 接続を確立するシステム状態に移動する。
- 外部環境からのシグナルを、データ転送に関連するシグナルに限定するように定義して、探索を開始する。外部環境から送ることができるシグナルの定義、およびシグナルをリストから削除する方法については、「外部環境からのシグナルの定義」を参照してください。

ユーザー定義ルールによって、探索の範囲を制限することもできます。探索を枝刈りする位置のシステム状態と一致するルールを定義して、ユーザー定義ルールのレポート アクションが探索の枝刈りであることを確認します。たとえば、以下のルールは、

```
state(initiator:1)=idle
```

イニシエータ プロセスが 「idle」状態になるたびに、探索を枝刈りします。ユーザー定義ルールについては、「ユーザー定義ルールの管理」および「ユーザー定義ルール」を参照してください。

効率的なビット探索

ビット探索は、ハッシュ値に基づくアルゴリズムを使用して、探索した状態空間を保管します。残念ながら、システム状態からハッシュ値を算出する操作は負荷が高いため、ビット探索の実行時間の多くがハッシュ値の計算に費やされます。通常、ハッシュ アルゴリズムの実行時間は、各システム状態の大きさに比例します。ハッシュ アルゴリズムが使用する最大と最小のシステム状態サイズは、各ビット探索後に出力される統計情報に含まれます。探索が遅い場合は、これを確認する必要があります。Bit-State-Exploration を参照してください。

システム状態が大きい（10,000 バイトより大きい）場合、エクスプローラのビット探索はかなり遅くなります。この場合は、ハッシュ値の算出時に、エクスプローラが使用する状態のサイズを減らして、パフォーマンスを最適化することを推奨します。これは、ハッシュ値の算出時に、エクスプローラにいくつかの変数をスキップするように指定することで可能です。エクスプローラには、これを目的とした Define-Variable-Mode コマンドがあります。たとえば、以下のコマンドでは、

```
define-variable-mode monitor subscrTab skip
```

エクスプローラは、モニタ クラスのインスタンスで `monitor` のすべての `subscrTab` 変数をスキップします。

この機能が役立つ典型的な例としては、システムが大きな配列（または他の大きなデータ構造）を含んでおり、これがシステムの起動時に初期化され、初期化後には、探索される状態空間の一部として一定であることがわかっているケースがあります。エクスプローラでこれをうまく利用する方法は、以下のとおりです。

1. 配列が初期化されるシステム状態に移動します。「部分探索をどこから開始するか」を参照してください。
2. ルートを、現在の状態に再定義します。Define-Root を参照してください。
3. テーブル変数のモードを「Skip」に変えます。
4. ビット探索を開始します。

この方法を使用することで、エクスプローラのパフォーマンスをかなり上げることができます。

また、システムの動的な動作に影響がないことがわかっている変数がシステム内にある場合にも、変数モードを「Skip」に変更できます。

状態空間サイズの削減

UML システムに関する知識とアサーションを使用して、探索が必要な状態空間を減らす方法はいくつかあります。通常、これは、UML システムの状態空間が、一部の詳細（探索の目的にあまり役立たない）を除き、同等な各種「下位状態空間」を含むという事実に基づいています。以下に、そのような詳細の例を示します。

- ローカル変数の値
- プロセス タイプのインスタンスの数
- 大きなデータ構造のサイズ
- 動的動作に影響しない変数

ローカル変数の値

以下に、ローカル変数の値が状態空間のサイズに影響する例を示します。アクティブ クラスが、外部環境から特定のシグナルが送られる回数を数える整数値を含み、外部環境から要求されたときにこの整数値を返すケースがあるとします。このローカル変数に可能なすべての値に対して、UML システムの動作を調べることは、あまり意味がありません。代わりに、妥当な数の値を選択し、この選択に基づいて状態空間探索を誘導するべきです。

ユーザー定義ルール（「ユーザー定義ルールの管理」を参照）は、変数値を制限することで、状態空間のサイズを減らす効果的な方法を提供します。上記の例では、最初の 3 回まで変数を増やして、動作を調べるというのが妥当な制限です。このルールは以下のようになります。

```
proc:1->var<4;
```

このルールを定義し、ユーザー定義ルールに違反したときのレポート アクションを枝刈り（デフォルト）に設定すると、状態空間内の必要な部分だけが探索されます。

アクティブ クラス インスタンスの数

もう 1 つの問題は、各クラス タイプに使われるクラス インスタンスの数です。この数が多く、しかもすべてが同じこと（たとえば、接続指向プロトコルの各接続のモデル化）を行っている場合、すべてのインスタンスの組み合わせを探索することはあまり意味がありません。代わりに、探索で許可するインスタンスの数を制限する方が効果的です。これは、Define-Max-Instance コマンドを使用して行うことができます。また、ユーザー定義ルールを使用しても、同じ結果を得ることができます。

大きなデータ構造のサイズ

エクスプローラのパフォーマンスが低下する 3 つ目の要因は、UML システムで大きなデータ構造（たとえば、配列）を使用するケースです。大きなデータ構造は、状態空間探索に 2 つの悪影響を及ぼします。

- データ構造のサイズが大きくなるにつれて、到達可能な状態空間の大きさが大幅に大きくなります。
- ビット探索アルゴリズムの効率は、システム状態のサイズが大きいほど低くなります。基本的に、新しいシステム状態を算出する時間は、システム状態の大きさに比例します。

この場合、探索を行う前に、UML システム内の大きなデータ構造のサイズをできるだけ小さくすることが効果的です。もう 1 つの方法として、「効率的なビット探索」で説明しているように、ハッシュ値を算出する変数をスキップする方法もあります。

動的動作に影響しない変数

多くの場合、UML システムには、システムの動的動作に影響しない変数がいくつかあります。基本的に、分岐でたどる経路、または出力シグナルの受信側を計算するときに使用する式に（直接または間接的に）影響しないすべての変数は、システムの動的動作に影響しません。

これらの変数は、状態空間探索を行う際に無視してもかまいません。このためには、Define-Variable-Mode コマンドを使用して、エクスプローラにこれらの変数をスキップするように指示します。これにより、エクスプローラが探索すべき状態空間の大きさが大幅に減り、エクスプローラのパフォーマンスが大きく向上します。

この観点では、Sender/Parent/Offspring などの暗黙的な変数も、変数とみなされます。特に、Sender はシグナルを受信するたびに値が変化するので、使用していなければスキップする価値があります。

たとえば、クラス「p」で Sender を使用していない場合、以下のコマンドによって、エクスプローラは 2 つのシステム状態を比較するときに、Sender 暗黙変数を無視します。

```
define-variable-mode p Sender skip
```

ランダム探索の使用

大規模な UML システムの探索を行う際に、ここで説明するより精緻なテクニックを使用できない場合もあります。また、探索に使用できる時間とリソースが、あまりにも限られている場合があります。非常に大規模な UML システムの探索を行う手法として、ビット探索アルゴリズムの代わりにランダム探索を使用する方法があります。

その理由は、ランダム探索アルゴリズムは、ランダムに選んだ状態空間の部分探索を行うことができるからです。さらに、探索のシンボル カバレッジは、どれだけ長く探索を継続できるかによってのみ決まります。このアルゴリズムの欠点は、状態空間の大部分をカバーするような長時間の探索を行った場合に、探索済みの経路を再探索するのを防ぐ仕組みがないことです。

インクリメンタル 探索

大きな UML 仕様と設計を開発する一般的な方法として、インクリメンタルな開発手法があります。まず、基礎的な機能を実装し、次に各種の機能を徐々に追加してゆきます。このような開発プロセスでは、システムの探索を計画するには、各追加部分で行う状態空間探索を定義するとよいでしょう。

まず、異なる開始状態、および場合によっては異なるテスト値で、いくつかの状態空間探索を実行します。これらの探索を行うことで、基本機能を示す UML システムのプロセス グラフ カバレッジが得られます。

各追加開発に対して、UML システムの新しい機能をカバーする追加探索を行います。

必要なプロセス グラフ カバレッジを得るために、実行すべき各種探索を自動的に実行できるコマンド スクリプトを定義するのも効果的です。この方法により、システムに追加された各種新機能に対して、すべての探索を簡単に実行できます。

外部環境からのシグナルの定義

状態空間探索テクニックに共通する問題は、すべて分析対象の UML システムの外部環境に関連しています。たとえば、状態空間探索時に、外部環境から整数パラメータを持つシグナルを受け取ることがあるケースを考えます。整数値は無数にあるので、現在のシステム状態のサクセサも、パラメータ値が 0、1 など、無数にあります。

これは当然、状態空間探索を行うときに容認できるケースではありません。Tau エクスプローラでは、これを避けるため以下の 3 つの方法をとることができます。

1. UML を使用するシステムの外部環境を指定することで、閉じたシステムを作る。これで問題は解決しますが、外部環境の UML モデルを作らなければならないという、新しい問題を引き起こします。
2. 外部環境からシステムへ送ることができるシグナルを指定する。これは、この問題を避ける簡単な方法です。外部環境が送ることができるシグナルをパラメータと共に列挙することにより、状態空間の各システム状態で有限の分岐が保証されます。

2 つ目の方法が最も一般的です。エクスプローラのテスト値機能は、外部環境からのシグナルを定義しやすいように設計されています。

テスト値

エクスプローラの開始時に、状態空間探索時に外部環境から利用可能なシグナル リストとして使われる、シグナルのリストが自動的に算出されます。このシグナル リストは、テスト値の概念に基づいて作られます。テスト値はデータ型およびシグナル パラメータ用に定義できます。シグナル リストの作成時に、エクスプローラは、パラメー

タ（またはパラメータ データ型）にテスト値が定義されている外部環境から送られる各シグナルを調べます。その後、パラメータのテスト値の各組み合わせに対して、1 つのシグナル インスタンスを作成します。

状態空間探索時に、エクスプローラが外部環境からの入力が可能な状態にあるとき、テスト値で定義されたシグナルのリストを調べます。

以下に、シンプルなデータ型のデフォルトのテスト値を示します。

データ型	デフォルト テスト値
`Integer`	-55, 0, 55
`Boolean`	true, false
`Real`	-55, 0, 55
`Natural`	0, 55
`Character`	‘a’
`Charstring`	“test”
`Duration`	0
`Time`	0
`PId`	Environment PId
`Bit`	0, 1
`Octet`	00, FF
`Bit_string`	’01’B
`Octet_string`	’00FF’H

上記以外のデータ型については、以下のようにテスト値が決定されます。

- 列挙型：型内のすべての値
- 定義済みデータ型の下位範囲：範囲内のすべての値
- 構造：個別フィールドのテスト値のすべての組み合わせ
- 配列：コンポーネント タイプのテスト値すべての組み合わせ
- Ref 型：NULL + Ref が指すもののテスト値へのポインタ
- Own 型：NULL
- ORef 型：NULL

テスト値の制限とオプション

算出されるテスト値には、2 つの制限があります。

- 特定のデータ型またはシグナル パラメータのテスト値の数が最大数を超えると、ランダムに選択したテスト値が生成される。

- 特定のシグナル タイプのシグナル インスタンスの数が最大数を超えると、そのシグナル タイプにはランダムに選択したシグナル インスタンスが生成される。

上記制限に関連するオプションを設定するには、以下の 2 つのコマンドを使用できます。

- 任意のデータ型またはシグナル パラメータのテスト値の最大数を定義するには、Define-Max-Test-Values コマンドに続けてテスト値の数を入力します。デフォルト値は「10」です。
- 任意のシグナル タイプのシグナル インスタンスの最大数を定義するには、Define-Max-Signal-Definitions コマンドに続けてシグナル インスタンスの数を入力します。デフォルト値は「10」です。

> 注記
>
> これらのオプションは状態空間に影響します。「状態空間サイズの削減」を参照してください。

テスト値の定義とリスト

デフォルトのテスト値は、多くのアプリケーションに便利なように定義されていますが、修正が必要な場合があります。必要以上に多くのテスト値がある場合に、状態空間探索のパフォーマンスを上げるために、テスト値の一部を削除できます。また、自動テスト値生成機能で、使用されているデータ型の一部を処理できないため、手動でテスト値を定義しなければならないケースもあります。

したがって、テスト値の変更は、エクスプローラの動作を微調整する場合、または、外部環境からのシグナルがユーザー定義パラメータまたは特殊なデータ型のパラメータを持つ場合にのみ必要です。

> 注記
>
> テスト値の変更は状態空間に影響します。「状態空間サイズの削減」を参照してください。

テスト値は、データ型、個別シグナル パラメータ、シグナル インスタンスの 3 つの「レベル」で、定義および削除できます。テスト値を定義または削除すると、環境からのシグナルのリストが再生成されます。データ型と個別シグナル パラメータ、またはシグナル インスタンスのテスト値を定義することを推奨します。この 2 つの手法は、組み合わせないでください。

テスト値に関するモニタ コマンド類は、エクスプローラ UI のテスト値ビューにあります。

データ型のテスト値

以下のコマンドは、データ型（ソート）のテスト値に作用します。

- ソートの新しいテスト値を定義するには、Define-Test-Value コマンドを使用するか、エクスプローラ UI から［値追加］を選択します。
- すべてのソートに定義されている新しいテスト値を表示するには、List-Test-Values コマンドを入力するか、テスト値ビューの［テスト値］セクションを表示します。

- ソートのすべてのテスト値を削除するには、Clear-Test-Values コマンドを入力するか、［すべてを削除］ボタンをクリックします。パラメータとしてはソート、またはすべてのソートを意味する「-」を指定します。

シグナル パラメータのテスト値

以下のコマンドは、シグナルへの個別パラメータのテスト値に作用します。

- シグナル パラメータの新しいテスト値を定義するには、Define-Parameter-Test-Value コマンドを入力するか、［パラメータ追加］フォームを使用します。パラメータは、シグナル、シグナル パラメータの順序数、および値です。その例を以下に示します。

  ```
  Define-Parameter-Test-Value Score 1 -5
  ```

- すべてのシグナル パラメータに定義されている新しいテスト値を表示するには、List-Parameter-Test-Values コマンドを入力するか、テスト値ビューの［シグナル定義］セクションを表示します。
- シグナル パラメータのすべてのテスト値を削除するには、Clear-Parameter-Test-Values コマンドを入力するか、［すべてを削除］ボタンをクリックします。パラメータとしては、シグナルとシグナル パラメータの順序数を指定します。パラメータ数に「-」を使用して**すべての**シグナル パラメータを表すか、シグナルに「-」のみを使用して**すべての**シグナル パラメータの**すべての**シグナルを表すことができます。

シグナル インスタンスのテスト値

以下のコマンドは、特定のシグナル インスタンスのテスト値に作用します。

- シグナル インスタンスに一連の新しいテスト値を定義するには、Define-Signal コマンドを入力するか、［シグナル追加］ボタンをクリックします。パラメータは、シグナルとパラメータのオプション値です。複数の Define-Signal コマンドを使用して、シグナル タイプが同じで値が異なる複数のシグナル インスタンスを定義できます。その例を以下に示します。

  ```
  Define-Signal Test 10 'hello' true
  Define-Signal Test -5 'bye'
  ```

  注記

  このコマンドを使用して定義したシグナルは、ソートまたはシグナル パラメータのテスト値を定義した場合など、シグナル リストが再生成されたときに削除されます。

- 現在定義されているすべてのシグナル インスタンスを表示するには、List-Signal-Definitions コマンドを入力するか、テスト値ビューの［パラメータ テスト値］セクションを表示します。
- シグナル タイプのすべてのテスト値を削除するには、Clear-Signal-Definitions コマンドを入力するか、［すべてを削除］ボタンをクリックします。パラメータとしては、シグナルまたはすべてのシグナルを意味する「-」を指定します。

テスト値の保存

現在の一連のテスト値は、ファイルに保存して、後でファイルを読み込んで再現できます。ファイルには、保存した一連のテスト値を再現して、他のテスト値を破棄するモニタ コマンドが含まれます。

テスト値を保存するには、Save-Test-Values コマンドに続けてファイル名を入力します。保存したテスト値を読み込むには、Include-File コマンドに続けてファイル名を入力します。

外部 C コードによるシステムの探索

Tau では、UML システムと共に外部 C コードを使用できますが、エクスプローラでも同じように使用できます。多くの場合、外部 C コードを使用するシステムで、直接エクスプローラを使用できます。しかし、状態空間探索の特殊要件のため、外部 C コードには一部制限があり、エクスプローラと適切に機能するために外部コードの部分的な修正が必要となることがあります。

状態空間探索を可能にするには、エクスプローラが C コードで直接実装されているすべてのデータ構造を含む、システム状態の完全なコピーを作成できる必要があります。エクスプローラは、システム状態の各コピーを個別に変更できる必要があります。これは、次のことを意味します。

- C コードで定義されている変数はエクスプローラでは扱えない。
- C の共用体は、ポインタ、ポインタにより実装されるデータ型（UML 型の String や Bag に類似）、UML Pid を含むことができない。

C コードに変数がある場合、エクスプローラはそれを検出できない。エクスプローラが正しく機能しているように見えますが、C コードで定義されている変数は、エクスプローラがシステム状態をコピーするときにコピーされません。1 つのシステム状態で実施されるアクションによって変数の値が変更される場合、この値はエクスプローラが現在扱うすべてのシステム状態の値を変えます。たとえば、エクスプローラが自動探索時にバックトラックして、特定のシステム状態の可能なサクセサをさらにテストする場合に、C で定義した変数の値が、以前にそのシステム状態になったときの値と異なり、状態空間探索が正しく行われない可能性があります。

システム状態のコピーを可能にするには、エクスプローラは、たとえば、ポインタ ベースのデータ構造を正確にコピーするために、システム内のすべてのデータ領域のソートに関する正確な情報が必要です。1 つの結果として、エクスプローラは C の共用体のソートをサポートできません。これは、共用体がポインタ ベース ソートを含む可能性がある場合、エクスプローラは結合の現在のソートを認識できないので、共用体をポインタとみなすべきか判断できないためです。UML Pid もエクスプローラで特殊に扱われるため、C の共用体の一部になることはできません。

外部 C コードで動的メモリ割り当てを使用する場合は、エクスプローラが正しく機能するには、いくつかの特別な追加手順が必要です。これは、エクスプローラが各システム状態の一部として、動的に割り当てたすべてのデータ領域のリストを維持するために必要です。外部 C 関数がメモリを割り当てる場合、エクスプローラは割り当てられたデータ領域について通知される必要があります。C 関数がメモリを解放するときも同様です。これは、C コードから次の 2 つの関数を呼び出すことで実現できます。

```
extern void UserMalloc (void *data);
extern void UserFree (void *data);
```

UserMalloc はデータ領域を割り当てたときに呼び出し、UserFree はデータ領域を解放する直前に呼び出す必要があります。どちらの関数も、データ領域へのポインタをパラメータとして持ちます。

UserMalloc の目的は、エクスプローラが管理する動的に割り当てたデータ領域のリストに、新しい要素を挿入することです。エクスプローラには、割り当てたデータの種類や、その大きさを知らせる必要はありません。これは、エクスプローラが、データ領域を指し示す UML エンティティ（たとえば、変数）を見つけ、このエンティティが与えるソートと大きさが正しいと仮定することで、自動的に行われます。データ領域を指し示す UML エンティティが見つからない場合は、エラーとみなされ、エクスプローラ レポートが生成されます。

UserFree 関数の目的は、データ領域が解放され、動的に割り当てたデータ領域のリストからそれを削除する必要があることを、エクスプローラに通知することです。

特殊な C マクロ XVALIDATOR_LIB があります。これは外部 C ファイルでコードがエクスプローラ カーネルと共にコンパイルされたか確認するために使用できます。したがって、次の例のように、このマクロを使用して C コードがエクスプローラと共にコンパイルされた場合にのみ、UserMalloc/UserFree の呼び出しを入れることができます。

```
...
v = malloc( 10 );
#ifdef XVALIDATOR_LIB
UserMalloc( (void *)v );
#endif
```

ユーザー定義ルールを使用する

エクスプローラでは、状態空間探索時に遭遇したシステム状態のプロパティを調べるために使用する、ユーザー定義ルールを定義できます。ユーザー定義ルールに合致するシステム状態が見つかると、レポートが生成されます。一度に 1 つのユーザー定義ルールのみ定義できます。

さまざまな使用例

ユーザー定義ルールが有用であるケースが 3 つあります。

- UML システムのプロパティを確認する場合。

ユーザー定義ルールは、システム状態のプロパティを記述します。したがって、自動状態空間探索を使用することにより、指定したプロパティを満たすシステム状態の存在を確認できます。状態空間が小さくて完全に探索できる場合は、指定したプロパティのシステム状態が状態空間に含まれないことも確認できます。

- 特定のシステム状態を探索する場合。

ユーザー定義ルールを使用すれば、エクスプローラ モニタのナビゲーション コマンドを使用せずに、状態空間の特定のシステム状態に移動できます。ルールによって目的の状態を記述して、自動状態空間探索を使用することにより、ルールに合致するレポートに直接移動できます。この場合、ユーザー定義ルール レポートのレポート アクションは、「中止」に設定する必要があります。

- 探索する状態空間を減らす場合。

多くの UML システムでは、状態空間は非常に大きく、無限になることさえあります。その結果、効果的な状態空間探索が困難になります。しかし、多くの場合、状態空間は、何らかの理由で探索する意味のない、大きな下位空間を含んでいます。たとえば、ある特定の変数の値を除いて、状態空間の他の部分と同じかもしれません。そのような場合、ユーザー定義ルールを使用して、探索する必要のないシステム状態を指定して、探索を制限できます。そのような状態に遭遇すると、探索は中断されて別のノードで続行されます。

ルールの例

以下に、システム プロパティを調べるルールの例を示します。

```
exists P:Proc | P->var=12;
```

これは、変数「var」の値が「12」の「Proc」型のアクティブ クラスがある、すべてのシステム状態で true となります。

以下に、システム状態を探索するルールの簡単な例を示します。

```
state(initiator:1)=disconnected;
```

これはアクティブ クラス インスタンス「initiator:1」の状態が、「disconnected」であるすべてのシステム状態で true となります。

以下に、このようなルールの複雑な例を示します。

```
state(Game:1)=Winning and sitype(signal(Game:1))=Probe
```

これは、アクティブ クラス インスタンス「Game:1」の状態が「Winning」と等しく、同じアクティブ クラス インスタンスが消費するシグナルのタイプが「Probe」であるすべてのシステム状態で true となります。

以下に、状態空間を減らすルールの例を示します。

```
(Game:1->Count > 2) or (Game:1->Count < -2)
```

これは、プロセス インスタンス「Game:1」の変数「Count」の絶対値が 2 より大きい、すべてのシステム状態で true となります。

ユーザー定義ルールの機能と構文の詳細な説明は、「ユーザー定義ルール」を参照してください。

ユーザー定義ルールの管理

ユーザー定義ルールを定義するには、Define-Rule コマンドに続けてルールの定義を入力します。

ユーザー定義ルールを削除するには、Clear-Rule コマンドを入力します。

現在のユーザー定義ルールを出力するには、Print-Rule コマンドを入力します。

現在のシステム状態でユーザー定義ルールが満たされているか評価するには、Evaluate-Rule コマンドを入力します。

アサーションの使用

エクスプローラ ライブラリによって、ユーザーは独自の実行時エラーまたはアサーションを定義できます。アサーションは、特定の変数が予想範囲内にあるかの検査など、実行時に行うテストです。アサーションは、C 関数 `xAssertError` を呼び出す式を導入して記述します。以下に例を示します。

例 **610:** アサーションの使用

```
[[if (#(I) < #(K))
    xAssertError("I is less than K");
]]
```

Tau エクスプローラでは、状態空間探索時にアサーションがチェックされます。遷移の実行時に、`xAssertError` が呼び出されるたびにレポートが生成されます。ユーザー定義ルールを使用する代わりに、この方法でアサーションを定義する利点は、エクスプローラがユーザー定義ルールよりはるかに効率的にインラインのアサーションを計算することです。

以下のプロトタイプを持つ `xAssertError` 関数は、

```
extern void xAssertError ( char *Descr )
```

アサーションを記述する文字列をパラメータとして、通常の実行時エラーと似た SDL 実行時エラーを出力します。この関数は、コンパイル スイッチ XASSERT が定義されている場合にのみ使用できます。標準のライブラリについては、これはアプリケーション ライブラリを除くすべてのライブラリに当てはまります。

モデル エクスプローラ リファレンス

ここでは、エクスプローラで使用可能なコマンドをアルファベット順に一覧表示します。エクスプローラへの入力はすべて、大文字／小文字の区別はありません。

コマンドのアルファベット順リスト

?（対話形式のコンテキスト依存ヘルプ）

パラメータ：
（なし）

「?」（疑問符）コマンドに対して、エクスプローラは現在の位置で許可されるすべての値を一覧表示します。値の列記が不適切な場合は型名を表示します。一覧表示の後、入力を継続できます。

?（コマンドの実行）

パラメータ：

```
<Command name>
```

パラメータとして与えられたエクスプローラ コマンドを実行します。これはエクスプローラ UI 用の便利な機能です。パラメータとして「?」を入力すると、使用可能なコマンド名が一覧表示されます。

Assign-Value

パラメータ：

```
[ ‘(’ <PId value> ‘)’ ]
<Variable name> <Optional component selection>
<New value>
```

クラス インスタンス内の特定の変数、または現在の範囲によって与えられた操作に、新しい値を割り当てます。

コマンドを指定すると、動作ツリーのルートが現在のシステム状態に設定されます。

コマンド Examine-Variable と同様に、割り当て対象の値の前にクラスの属性名または有効な配列インデックスを付加することで、構造化変数（クラス、文字列、配列）内のコンポーネントを扱うことが可能です。インデックス値とクラス属性名のリストを入力することで、入れ子になったクラスや配列を扱うことができます。

かっこ内にインスタンス識別子を指定すると、範囲は一時的にこのインスタンスに変更されます。

Bit-State-Exploration

パラメータ：
（なし）

ビット状態空間アルゴリズムを使用して、現在のシステム状態から状態空間自動探索を開始します。探索が開始されると、以下の情報が出力されます。

- 探索の深さ : 探索の最大の深さ。これはコマンド Define-Bit-State-Depth で設定できます。
- ハッシュ テーブル サイズ : ビット探索中に使用されるハッシュ テーブルのサイズ。これはコマンド Define-Bit-State-Hash-Table-Size で設定できます。

探索は、定義された深さまで完全に状態空間を探索するか、［中断］コマンドが送信されるまで継続されます。その後、システムは探索が開始される前の状態に戻ります。

ビット探索を開始したが現在停止している場合、このコマンドを実行すると、停止した位置から探索を継続するか、最初から再開するかを確認されます。

20,000 回の遷移が実行されるごとに、ステータス メッセージが出力されます。

探索が終了または停止すると、デフォルトでレポート ビューが表示されます。以下の統計情報も出力されます。

- No of reports：List-Reports コマンドによってリストされる、報告された状況の数。
- Generated states：生成されたシステム状態の総数。
- Truncated paths：探索が最大の深さに到達し、現在の実行経路が打ち切られた回数。
- Unique system states：動作ツリーで重複していない、生成されたシステム状態の数。
- Size of hash table：ハッシュ テーブルのサイズ（ビット数とバイト数）。
- No of bits set in hash table：生成された状態空間を示すために実際に使用されるビット数。
- Collision risk：2 つの異なるシステム状態に関して、ハッシュ テーブル内で発生する衝突のリスク（パーセント）。これによって、新たに生成された状態内で実行パスが誤って打ち切られてしまうことがあります。
- Max depth：探索中に到達した動作ツリー内の最大レベル数。
- Current depth：探索が停止された時点で到達していた動作ツリーのレベル。「-1」の場合、探索は完了しています。つまり、指定された深さまでの完全な動作ツリーが探索されています。この値が「>0」より大きい場合、コマンドを再度実行して、このレベルから探索を継続できます。
- Min state size: ハッシュ値の算出時にシステム状態の保存に使用される最小バイト数。
- Max state size：ハッシュ値の算出時にシステム状態の保存に使用される最大バイト数。
- Symbol coverage：プロセス グラフ内で少なくとも一度実行されたシンボルの割合（パーセント）。

Bottom

パラメータ：

（なし）

現在のパスの最後まで、動作ツリーを下に移動します。

Cd

パラメータ：

```
<Directory>
```

現在の作業ディレクトリを指定ディレクトリに変更します。

Connector-Disable

パラメータ：

```
<connector> | ‘-’
```

指定コネクタを使用するすべてのシグナルに定義された、すべてのテスト値を無効にします。ë-í は「すべてのコネクタ」を意味します。使用を再開するには、Connector-Enable を使用します。併せて Signal-Disable も参照してください。シグナルのテスト値は、シグナルを転送するコネクタとシグナルの両方が有効な場合にのみ使用されます。デフォルトでは、すべてのシグナルとコネクタは有効です。

Connector-Enable

パラメータ：

```
<connector> |‘-’
```

指定コネクタを使用するすべてのシグナルに定義された、すべてのテスト値を有効にします。‘-’ は「すべてのコネクタ」を意味します。併せて Signal-Enable も参照してください。シグナルのテスト値は、シグナルを転送するコネクタとシグナルの両方が有効な場合にのみ使用されます。デフォルトでは、すべてのシグナルとコネクタは有効です。

Clear-Coverage-Table

パラメータ：

（なし）

テスト カバレッジ情報をクリアします。

Clear-Parameter-Test-Values

パラメータ：

```
( <Signal> |‘-’ ) ( <Parameter number> | ‘-’ )
( <Value> |‘-’ )
```

コマンドのパラメータとして与えられたシグナル パラメータの、値パラメータによって記述されているテスト値をクリアします。指定された値パラメータが ‘-’ の場合、コマンドのパラメータとして与えられたシグナル パラメータの、すべてのテスト値がクリアされます。パラメータ番号の代わりに ‘-’ が指定されている場合は、そのシグナルのすべての パラメータのテスト値がクリアされます。シグナル名の代わりに ‘-’ が指定されている場合は、すべてのシグナルのすべての パラメータのすべてのテスト値がクリアされます。

状態空間探索中に、外部環境から送信可能な一連のシグナルを再生成します。

Clear-Reports

パラメータ：
（なし）

最新の状態空間探索から現在のレポートを削除します。

Clear-Rule

パラメータ：
（なし）

現在定義されているルールを削除します。

Clear-Signal-Definitions

パラメータ：
```
<Signal> | ‘-’
```

パラメータによって指定されたシグナルの、現在定義されているテスト値をすべてクリアします。‘-’ が指定されている場合は、すべてのシグナルのテスト値がクリアされます。

注記

このコマンドによってクリアされたシグナルは、ソートのテスト値を定義するコマンドまたはパラメータが使用された場合に再生成されることがあります。

Clear-Test-Values

パラメータ：
```
( <Sort> | ‘-’ ) ( <Value> | ‘-’ )
```

パラメータによって指定されたソートのすべてのテスト値をクリアします。指定されたソート パラメータが ‘-’ の場合は、すべてのソートのすべてのテスト値がクリアされます。値パラメータが ‘-’ の場合は、指定されたソートのすべてのテスト値がクリアされます。

状態空間探索中に、外部環境から送信可能な一連のシグナルを再生成します。

Command-Log-Off

パラメータ：
（なし）

コマンド ログ機能をオフにします。詳細については、コマンド Command-Log-On を参照してください。

Command-Log-On

パラメータ：
```
<Optional file name>
```

エクスプローラで指定されたすべてのコマンドのログ記録を有効にします。初めてこのコマンドを入力するときは、ログ ファイルのファイル名をパラメータとして指定する必要があります。その後は、ファイル名が指定されていない Command-Log-On コマ

ンドによって直前のログ ファイルに情報が追加されます。また Command-Log-On コマンドでファイル名が指定されている場合は、古いログ ファイルが閉じられて、指定された名前の新しいファイルの使用が開始されます。

最初は、コマンド ログ機能はオフになっています。明示的にオフにするには、コマンド Command-Log-Off を使用します。

生成されたログ ファイルは、コマンド Include-File で直接ファイルとして使用できます。ただし、コマンド エラーによって実行されなかったコマンドも含め、セッションで指定されたコマンドのみが対象になります。最後の Command-Log-Off コマンドもログ ファイルの一部になります。

Continue-Until-Branch

パラメータ：

```
<Optional node number>
```

最初に、現在のシステム状態の子である、指定された番号を持つシステム状態ノードに移動します。これは、Next コマンドと同じです。次に、子が 1 つである限り、つまりブランチが見つかる限り下に移動します。

Continue-Up-Until-Branch

パラメータ：

（なし）

子が 1 つである限り、つまりブランチが見つかる限り、動作ツリーを上に移動します。

Default-Options

パラメータ：

（なし）

エクスプローラ内のすべてのオプションをデフォルト値にリセットし、すべてのレポートをクリアします。また現在の状態を現在のルートに設定します。コマンド Reset と比較してください。

Define-Bit-State-Depth

パラメータ：

```
<Depth>
```

パラメータはビット探索の最大の深さ、つまり動作ツリー内で到達されるレベルの数です。探索中にこの深さに到達すると現在の経路の探索が打ち切られ、動作ツリーの別の枝で探索が継続されます。デフォルト値は「100」です。

Define-Bit-State-Hash-Table-Size

パラメータ：

```
<Size in bytes>
```

ビット探索中に生成された状態空間を表すためのハッシュ テーブルのサイズを設定します。デフォルト値は「1,000,000」バイトです。

Define-Bit-State-Iteration-Step

パラメータ：

```
<Step>
```

Bit-State-Exploration アルゴリズムは、探索ごとに深さを増して自動的に探索を繰り返す機能を備えています。反復は、探索の深さが Define-Bit-State-Depth コマンドによって定義されている最大の深さよりも大きくなるまで、または打ち切ることなく 1 つの探索が終了するまで継続されます。

このコマンドはその機能を有効にし、反復ごとに深さをどれだけ深くするかを定義します。<Step> を「0」に設定すると、反復探索は無効になります。「0」以外の場合、<Step> に探索ごとの反復の最大の深さを定義します。

デフォルト値は「0」、反復機能は無効です。

Define-Connector-Queue

パラメータ：

```
<connector name> ( “On” | “Off” )
```

指定されたコネクタのキューを追加または削除します。コネクタにキューを追加すると、コネクタに送信されたシグナルは、コネクタに対応付けられているキューに入れられます。デフォルトでは、コネクタにキューは対応付けられていません。

Define-Exhaustive-Depth

パラメータ：

```
<Depth>
```

パラメータによって、全探索を実行するときの探索の深さを定義します。デフォルト値は「100」です。

Define-Integer-Output-Mode

パラメータ：

```
“dec” | “hex” | “oct”
```

整数値を 10 進、16 進、8 進のどの形式で出力するかを定義します。16 進形式では出力の先頭に「0x」が付き、8 進形式では出力の先頭に「0」（ゼロ）が付きます。

入力では、形式を 16 進または 8 進に設定すると、以下のように文字列によって基数が決定されます。オプションの先行符号の後の先行ゼロは、8 進変換を示します。また、先行する「0x」は 16 進変換を示します。それ以外の場合は、10 進変換が使用されます。

デフォルトは「dec」です。入力変換は実行されません。

Define-Max-Input-Port-Length

パラメータ：

```
<Number>
```

入力ポート キューの最大長を定義します。状態空間探索中にこの長さを超えると、レポートが生成されます（最大キュー長越えを参照してください）。デフォルト値は「3」です。

Define-Max-Instance

パラメータ：

```
<Number>
```

特定プロセス タイプに許可される最大インスタンス数を定義します。状態空間探索中にこの数を超えると、レポートが生成されます。デフォルト値は「100」です。

Define-Max-Signal-Definitions

パラメータ：

```
<No of signals>
```

特定シグナルについて、外部環境からシグナルのリストに追加されるシグナルの最大数を定義します。デフォルト値は「10」です。

Define-Max-State-Size

パラメータ：

```
<Size in bytes>
```

システム状態のハッシュ テーブルを算出するときに、各システム状態の保存に使用される内部配列のサイズを指定します。デフォルト値は「100,000」バイトです。

Define-Max-Test-Values

パラメータ：

```
<No of text values>
```

特定のデータ型またはシグナル パラメータ用に生成される、テスト値の最大数を定義します。デフォルト値は「10」です。

Define-Max-Transition-Length

パラメータ：

```
<Number>
```

動作ツリーの遷移中に実行が許可される UML シンボルの最大数を定義します。状態空間探索中にこの数を超えると、レポートが生成されます。デフォルト値は「1,000」です。

Define-Parameter-Test-Value

パラメータ：

```
<Signal> <Parameter number> <Value>
```

コマンド パラメータによって記述されているパラメータ テスト値を、現在のテスト値設定に追加します。外部環境から送信可能なシグナルのリストは、新しいテスト値設定に基づいて再生成されます。

Define-Priorities

パラメータ：

```
<Internal events> <Input from ENV> <Timeout events>
<Connector output> <Spontaneous transitions>
```

各種イベント クラスに優先度を定義します。個々に、優先度 1、2、3、4、または 5 を設定できます。各クラスおよび優先度の詳細については、状態空間ビューを参照してください。

デフォルトの優先度は以下のとおりです。

- 内部イベント : 1
- ENV 入力 : 2
- タイムアウト イベント : 2
- コネクタ出力 : 1
- 自発遷移 : 2

なお、エクスプローラ内で処理されるイベントの優先度は、コマンド Define-Symbol-Time の影響を受けることに注意してください。

Define-Random-Walk-Depth

パラメータ :

```
<Depth>
```

パラメータによって、ランダム探索を実行するときの探索の深さを定義します。デフォルト値は「100」です。

Define-Random-Walk-Repetitions

パラメータ :

```
<No of repetitions>
```

パラメータによって、ランダム探索を実行するときに開始状態から実行される探索の回数を定義します。デフォルト値は「100」です。

Define-Report-Abort

パラメータ :

```
<Report type> | ‘-’
```

指定されたタイプのレポートが生成されたときに状態空間探索が停止されるように定義します。

使用可能なレポート タイプは、コマンド Show-Options で一覧表示できます。動作ツリー内を指定されたレベル数まで上に移動します。Up 1 が指定されると、現在のシステム状態の親状態に移動します。パラメータが大きすぎる場合、Up は動作ツリーのルート（開始状態）で停止します。にも説明があります。パラメータに ‘-’ が指定された場合は、同じ方法ですべてのレポート タイプが定義されます。

Define-Report-Continue

パラメータ :

```
<Report type> | ‘-’
```

指定されたタイプのレポートが生成されたときにも、状態空間探索が継続されるように定義します。このように定義しておくと、動作ツリーの探索は生成されるレポートの影響を受けません。

使用可能なレポート タイプは、コマンド Show-Options で一覧表示できます。動作ツリー内を指定されたレベル数まで上に移動します。Up 1 が指定されると、現在のシステム状態の親状態に移動します。パラメータが大きすぎる場合、Up は動作ツリーのルート（開始状態）で停止します。にも説明があります。パラメータに ë-í が指定された場合は、同じ方法ですべてのレポート タイプが定義されます。

Define-Report-Log

パラメータ：

```
<Report type> ( “Off” | “One” | “All” )
```

状態空間探索中に遭遇した指定タイプのレポートについて、検出レポートのリストに保存する数を定義します。すべてのレポート タイプのデフォルト値は “One” です。

レポート タイプに “Off” が指定された場合、このタイプのレポートは保存されません。これはたとえば、List-Reports コマンドでも一覧表示されないことを意味します。ただし、コマンド Define-Report-Abort、Define-Report-Continue、および Define-Report-Prune で指定されているように、レポートは生成され、適切なアクションがとられます。

“One” を指定すると、報告された状況の 1 つのオカレンスだけが保存されます。

“All” を指定すると、異なる実行経路を持つ、報告された状況のすべてのオカレンスがリストに保存されます。

使用可能なレポート タイプは、コマンド Show-Options で一覧表示できます。パラメータに ‘-’ が指定された場合は、同じ方法ですべてのレポート タイプが定義されます。

Define-Report-Prune

パラメータ：

```
<Report type> | ‘-’
```

指定されたタイプのレポートが生成されたときに、状態空間探索が継続されないように定義します。したがって、動作ツリーのその状態より下にある部分は探索されません。その代わり、その状態の子または親で探索が継続されます。これはその状態における動作ツリーの「枝刈り」と呼ばれます。枝刈りは、すべてのレポート タイプのデフォルト動作です。

使用可能なレポート タイプは、コマンド Show-Options で一覧表示できます。パラメータに ‘-’ が指定された場合は、同じ方法ですべてのレポート タイプが定義されます。

Define-Root

パラメータ：

```
“Original” | “Current”
```

動作ツリーのルートとして、UML システムの現在のシステム状態または元の開始状態のいずれかを定義します。

注記

ルートを再定義すると、すべての経路（MSC トレースやレポート経路など）は、元の開始状態ではなく新しいルートから開始されます。

Define-Rule

パラメータ：

```
<User-defined rule>
```

状態空間探索中にチェックする新しいルールを定義します。

Define-Scheduling

パラメータ：

```
“All” | “First”
```

各状態において、レディ キュー内のどのアクティブ クラス インスタンスの実行を許可するかを定義します。パラメータによって、以下のようにスケジューリングを定義します。

- `All`
  各状態において、レディ キュー内のすべてのアクティブ クラス インスタンスの実行を許可します。
- `First`
  各状態において、レディ キュー内の最初のアクティブ クラス インスタンスのみの実行を許可します。

デフォルトは “First” です。

Define-Signal

パラメータ：

```
<Signal> <Optional parameter values>
```

外部環境から UML システムに送信するシグナルを定義します。シグナルは名前のほか、オプションとしてそのパラメータ値によっても定義できます。異なるパラメータ値を持つ同じシグナルを定義する場合は、複数の Define-Signal コマンドを使用できます。

注記

このコマンドによって定義したシグナルは、シグナルが再生成されたとき、つまりソートのテスト値を定義しているコマンド、またはシグナル パラメータが使用されたときに消滅します。

Define-Spontaneous-Transition-Progress

パラメータ：

```
“On” | “Off”
```

非進捗ループをチェックの実行時に、自発遷移（入力なし）を進捗と見なすかどうかを定義します。デフォルトでは自発遷移を進捗と考えます、“On”。

Define-Symbol-Time

パラメータ：

```
“Zero” | “Undefined”
```

UML アクティブ クラス内の、入力、タスク、分岐などのある 1 つのシンボルを実行するために要する時間を定義します。“Zero”（ゼロ）に設定すると、アクティブ クラス インスタンスによって実行されるすべてのアクションが、システム内で使用されるタイマー値に比べて非常に高速であると想定されます。シンボル実行時間を "undefined"（未定義）に設定されていると、アクティブ クラスがシンボルの実行に要する時間については何も想定されません。アクティブ クラスが持続時間 5 のタイマーを設定し、長時間を要する長いループなどを実行した後、持続時間 1 のタイマーを設定する例を考えてみましょう。シンボル実行時間をゼロに設定していると、2 番目のタイマーが必ず先に切れます。シンボル タイマーを未定義に設定している場合は、どちらか一方のタイマーが先に切れます。

なお、シンボル実行時間をゼロに設定していると、コマンド Define-Priorities によって内部イベントとタイマー イベントに同じ優先度が設定されていても、内部アクションが可能な場合はどのタイマーもタイムアップしないことに注意してください。

シンボル実行時間オプションのデフォルト値は “Zero” です。

Define-Test-Value

パラメータ：

```
<Sort> <Value>
```

パラメータによって記述されているテスト値を、現在のテスト値設定に追加します。外部環境から送信可能なシグナルのリストは、新しいテスト値設定に基づいて再生成されます。

注記

シグナル セットを再生成すると、Define-Signal コマンドを使用して手動で定義したシグナルはすべて失われます。

Define-Timer-Progress

パラメータ：

```
“On” | “Off”
```

非進捗ループのチェック時に、タイマーのタイムアップを進捗と見すかどうかを定義します。デフォルトではタイマーのタイムアップを進捗と考えます、“On”。

Define-Transition

パラメータ：

```
“UML” | “Symbol-Sequence”
```

動作ツリー内の遷移のセマンティクスと長さを定義します。パラメータによって、以下のように遷移を定義します。

- `UML`
  完全な UML 状態機械 グラフ遷移に対応する動作ツリー遷移。
- `Symbol-Sequence`
  他のアクティブ クラス インスタンスとやりとりせずに実行できる UML シンボルの最長シーケンスに対応する動作ツリー遷移。

デフォルトは “UML” です。

Define-Tree-Search-Depth

パラメータ：

```
<Depth>
```

樹状探索の最大の深さを定義します。

デフォルト値は「100」です。

Define-Variable-Mode

パラメータ：

```
<Active class> [ <Variable name> | “Parent” | “Offspring” |
“Sender” ]
[ “Compare” | “Skip” ]
```

状態空間探索中に 2 つのシステム状態を比較するときに、指定変数がどのように扱われるかを定義します。変数用の値が “Compare” の場合、2 つのシステム状態を比較するときにこの変数が考慮されます。値が “Skip” の場合、この変数は考慮されません。つまり 2 つのシステム状態間の違いが “Skip” モード内の変数値の違いだけであれば、システム状態は等しいと考えられます。

変数の “Skip” モードの目的は、状態空間探索を最適化することです。このコマンドを使用できるのは、以下の 2 つの状況です。

- 探索中一定であるとわかっているすべての変数については、“Skip” を宣言できます。
- システムの動的な振る舞いに影響しない、つまり “output to” 分岐や式を経由する経路に影響しないすべての変数については、“Skip” できます。

“Skip” モードのコンスタント変数の利点は、ハッシュ値を算出するときにエクスプローラがこれらの変数を無視することです。配列などの大きいデータ構造体の場合、これによってエクスプローラのパフォーマンスを大幅に向上させることができます。

Detailed-Exa-Var

パラメータ：

（なし）

このコマンドを指定すると、デフォルト値を持つコンポーネントが入ったデータ構造体を出力するときに、これらの値が明示的に出力されます。

Down

パラメータ：

```
<Number of levels>
```

現在の経路の一部である現在のシステム状態の子を選択しながら、動作ツリー内を指定されたレベル数まで下に移動します。パラメータが大きすぎると、Down は現在の経路の最後で停止します。

Evaluate-Rule

パラメータ：

（なし）

現在のシステム状態について、現在定義されているルールを評価します。このコマンドは、ルールが満たされているかどうかを出力します。

Examine-connector-Signal

パラメータ：

```
<connector name> <Entry number>
```

指定されたコネクタのキュー内のエントリ番号と等しい位置にある、シグナル インスタンスのパラメータを出力します。エントリ番号は、コマンド List-Connector-Queue を使用したときにシグナル インスタンスに対応付けられた番号です。

Examine-PId

パラメータ：

（なし）

現在の範囲によって与えられるアクティブ クラス インスタンスについての情報を出力します（範囲の説明については Set-Scope コマンドを参照してください）。この情報には、Parent、Offspring、Sender の現在の値、およびアクティブ クラス インスタンスが実行した現在アクティブなすべての操作呼び出しのリストが含まれます。このリストは最新の操作呼び出しから始まり、アクティブ クラス インスタンス自体で終わります。

Examine-Signal-Instance

パラメータ：

```
<Entry number>
```

現在の範囲によって与えられるアクティブ クラス インスタンスの入力ポート内の指定された位置にある、シグナル インスタンスのパラメータを出力します（範囲の説明については Set-Scope コマンドを参照してください）。エントリ番号は、コマンド List-Input-Port を使用したときにシグナル インスタンスに対応付けられた番号です。

Examine-Timer-Instance

パラメータ：

```
<Entry number>
```

指定されたタイマー インスタンスのパラメータを出力します。エントリ番号は、コマンド List-Timer を使用したときにタイマーに対応付けられた番号です。

Examine-Variable

パラメータ：

```
[ ‘(’ <PId value> ‘)’ ]
<Optional variable name>
<Optional component selection>
```

現在の範囲にある指定された変数または仮パラメータの値を出力します（範囲の説明については Set-Scope コマンドを参照してください）。変数名は略記できます。変数名を指定しない場合、現在の範囲によって与えられるアクティブ クラス インスタンスの

すべての変数値、および仮パラメータ値が出力されます。Sender、Offspring、Parent も、この方法で調べることができます。ただし、これらの名前は略記できません。また、すべての変数のリストにも含まれません。

注記

変数をエクスポートしている場合は、現在の値とエクスポート先の値の両方が出力されます。

追加パラメータとしてクラスの属性名または有効な配列インデックス値を付加することにより、クラスの属性の値のみ、または文字列変数や配列変数のコンポーネントを調べることができます。リスト コンポーネント選択パラメータを指定すれば、クラスや配列内のクラスと配列を任意の深さまで扱うことができます。単に空白を使用するだけでなく、'!' や "( )" を持つ構文を使用して、名前とインデックス値を区切ることができます。

配列名の後に擢 romIndex : ToIndexî を指定することで、配列の範囲を出力することも可能です。なお、FromIndex が名前 (列挙リテラル) の場合は、':' の前に空白が必要です。範囲指定の後で、コンポーネントは選択できません。

変数内で使用可能なコンポーネントを確認するには、変数名に空白と '?' を付加して入力する必要があります。コンポーネントまたは型名のリストが出力されます。その後、入力を継続できます。コンポーネント名の後ろに再度 '?' を付加して、使用可能なサブコンポーネントを一覧表示できます。

かっこで囲んで PId を指定すると、範囲は一時的にこのアクティブ クラス インスタンスに変更されます。

Exhaustive-Exploration

パラメータ：

（なし）

生成された状態空間全体がメインメモリに保存されている、現在のシステム状態から状態空間の自動探索を開始します。これは小さい状態空間を持つ UML システムにのみ推奨されます。

定義された深さまで完全な状態空間が探索されるまで、またはコマンド プロンプトから［ `<Return>` ］が押されるか、*Break* コマンドが送信されるまで、探索は継続されます。その後、システムは最初のシステム状態に戻ります。探索の最大の深さは、コマンド Define-Exhaustive-Depth で設定できます。

全探索を開始したが現在停止している場合、このコマンドを実行すると、停止した位置から探索を継続するか、最初から再開するかを確認されます。

50,000 の状態が生成されるごとに、ステータス メッセージが出力されます。探索が終了または停止すると、ビット探索の場合と同じ情報が出力されます。

Exit

パラメータ：

（なし）

実行中のエクスプローラを終了します。コマンドを略記すると、エクスプローラに確認されます。エクスプローラ のいずれかのオプションを変更している場合は、変更したオプションを保存すべきかどうかエクスプローラに確認されます。保存する場合、変更されたオプションは、エクスプローラ実行形式ファイルが開始されたディレクトリ内のファイル `.valinit`（UNIX の場合）、またはファイル `valinit.com`（Windows の場合）に保存されます。このファイルは、次回同じディレクトリからエクスプローラを開始したときに、自動的に読み込まれます。これによって、前に保存したオプションが復元されます。

これは Quit と同じコマンドです。

Generate-SQD-Trace

状態空間のルートから現在の位置に導いたイベントを示すシーケンス図を表示します。

Goto-Path

パラメータ：

```
<Path>
```

経路によって指定されたシステム状態に移動します。経路の詳細については、コマンド Print-Path を参照してください。

Goto-Report

パラメータ：

```
<Report number>
```

対応する番号を持つレポートが検出された動作ツリー内の状態に移動します。レポート状況が出力される前に実行された最後の動作ツリー遷移が出力されます。これには、シミュレーション中のフルトレースの場合と同じ情報が伴います。レポート番号は、コマンド List-Reports を使用したときにレポートに対応付けられた番号です。レポートが 1 つしかない場合、レポート番号はオプションです。

Help

パラメータ：

```
<Optional command name>
```

パラメータなしで Help コマンドを実行すると、使用可能なコマンドが一覧表示されます。パラメータとしてコマンド名を指定すると、そのコマンドの説明が表示されます。

Include-File

パラメータ：

```
<File name>
```

テキスト ファイルに保存されている一連のエクスプローラ コマンドの実行を可能にします。Include-File 機能は、たとえば初期化シーケンスや完全なテスト ケースをインクルードするのに便利です。Include-File は、コマンドのインクルード シーケンス内で使用できます。インクルードファイルは最大 5 つまでのネストレベルが可能です。

List-Connector-Queue

パラメータ：

```
<connector name>
```

指定されたコネクタ キュー内の、すべてのシグナル インスタンスのリストを出力します。各シグナル インスタンスについて、エントリ番号、シグナル タイプ、および送信プロセス インスタンスが与えられます。エントリ番号は、コマンド Examine-connector-Signal 内で使用できます。

List-Input-Port

パラメータ：

```
[ ‘(’ <PId value> ‘)’ ]
```

現在の範囲によって与えられるアクティブ クラス インスタンスの入力ポート内のすべてのシグナル インスタンスのリストを出力します（範囲の説明については Set-Scope コマンドを参照してください）。各シグナル インスタンスについて、エントリ番号、シグナル タイプ、および送信アクティブ クラス インスタンスが与えられます。エントリ番号の前の ‘*’ は、対応するシグナル インスタンスがアクティブ クラス インスタンスによって実行される次の遷移で消費されるシグナル インスタンスであることを示しています。エントリ番号は、コマンド Examine-Signal-Instance 内で使用できます。

かっこで囲んで PId を指定すると、代わりにこのアクティブ クラス インスタンスについての情報が出力されます。

List-Next

パラメータ：

（なし）

現在のシステム状態の次に可能な動作ツリー遷移のリストを出力します。

注記

可能な遷移数は、選択した状態空間オプションによって決まります。

List-Parameter-Test-Values

パラメータ：

（なし）

シグナル パラメータ用に現在定義されているテスト値をすべて一覧表示します。

List-Active-Class

パラメータ：

```
<Optional active class name>
```

指定されたアクティブ クラス名を持つすべてのアクティブ クラス インスタンスのリストを出力します。アクティブ クラス名を指定しない場合は、システム内のアクティブ クラス インスタンスがすべてリストされます。リストには List-Ready-Queue コマンドで説明したものと同じ詳細情報が含まれます。

List-Ready-Queue

パラメータ：
（なし）

レディ キュー内のアクティブ クラス インスタンスのリストを出力します。

List-Reports

パラメータ：
（なし）

状態空間探索中に報告されたすべての状況を出力します。レポートごとに、動作ツリー内で最初に発生した深さとともに、状況を記述したエラーまたは警告メッセージが出力されます。報告された状況ごとに、動作ツリーのルートに最も近い経路を持つ 1 つのオカレンスのみが出力されます。出力されたレポート番号は、コマンド Goto-Report 内で使用できます。

List-Signal-Definitions

パラメータ：
（なし）

現在定義されているすべてのシグナルのリストを出力します。

List-Test-Values

パラメータ：
（なし）

現在定義されているすべてのテスト値を一覧表示します。

List-Timer

パラメータ：
（なし）

現在アクティブなすべてのタイマーのリストを出力します。タイマーごとに、対応するプロセス インスタンスと、対応付けられている時間が与えられます。エントリ番号もこのリストの一部であり、コマンド Examine-Timer-Instance 内で使用できます。

Load-Signal-Definitions

パラメータ：

```
<File name>
```

Define-Signal コマンドを持つコマンド ファイルを読み込み、シグナルを定義します。このコマンドは、下位互換性を保つためのものです。

Log-Off

パラメータ：
（なし）

コマンド Log-On で説明している対話ログ機能をオフにします。

Log-On

パラメータ：

```
<Optional file name>
```

オプションでファイル名をパラメータとして取り、画面に表示されているユーザーとエクスプローラ間のすべての対話のログ記録を有効にします。初めてこのコマンドを入力するときは、ログ ファイルのファイル名をパラメータとして指定する必要があります。その後は、ファイル名が指定されていない Log-On コマンドによって直前のログ ファイルに情報が追加されます。また Log-On コマンドにファイル名を指定しなかった場合は、古いログ ファイルが閉じられて、指定された名前の新しいファイルの使用が開始されます。

最初は対話ログ機能はオフになっています。明示的にオフにするには、コマンド Log-Off を使用します。

Merge-Report-File

パラメータ：

```
<File name>
```

既存のレポート ファイルを開き、ファイル内のレポートを現在のレポートに追加します。

New-Report-File

パラメータ：

```
<File name>
```

レポート保管用の新しいレポートファイルを作成します。現在のレポートは削除されます。

Next

パラメータ：

```
<Transition number>
```

動作ツリー内の次のレベルにあるシステム状態、つまり現在のシステム状態の子に移動します。パラメータは、List-Next コマンドによって指定された遷移番号です。

Open-Report-File

パラメータ：

```
<File name>
```

既存のレポート ファイルを開き、ファイル内のレポートを読み込みます。現在のレポートは削除されます。

Print-Evaluated-Rule

パラメータ：

（なし）

現在定義されているルールを、最後のルール評価から得られた値とともに出力します。出力される情報は、いわゆる解析ツリーの形をとります。この情報を正しく解釈するには、このような構造体についての知識が必要です。

Print-File

パラメータ：

```
<File name>
```

指定されたテキスト ファイルの内容を画面に表示します。

Print-Path

パラメータ：

（なし）

現在のシステム状態への経路を出力します。経路は、現在のシステム状態に到達方法を記述した整数列で、最後は 0 です。最初の数はルートから選択する遷移、2 番目の数はこの状態から選択する遷移、••• といったように、各遷移を示しています。経路によって指定された状態に移動するには、コマンド Goto-Path を使用します。

Print-Report-File-Name

パラメータ：

（なし）

現在のレポート ファイルの名前を出力します。

Print-Rule

パラメータ：

（なし）

現在定義されているルールを出力します。

Print-Trace

パラメータ：

```
<Number of levels>
```

現在のシステム状態に至るまでのテキスト トレースを出力します。各動作ツリー遷移について、シミュレーション中のフル トレースの場合と同じ情報が出力されます。パラメータによって、トレースをどのレベルから開始するかが決まります。たとえば Print-Trace 10 は最後の 10 の遷移を出力します。

Quit

パラメータ：

（なし）

実行中のエクスプローラを終了します。コマンドを略記すると、エクスプローラに確認されます。エクスプローラ のいずれかのオプションを変更している場合は、変更したオプションを保存すべきかどうかエクスプローラに確認されます。保存する場合、変更されたオプションは、エクスプローラ実行形式ファイルが開始されたディレクトリ内のファイル `.valinit`（UNIX の場合）、またはファイル `valinit.com`

（Windows の場合）に保存されます。このファイルは、次回同じディレクトリからエクスプローラを開始したときに、自動的に読み込まれます。これによって、前に保存したオプションが復元されます。

これは Exit と同じコマンドです。

Random-Down

パラメータ：

```
<Number of levels>
```

現在のシステム状態の子を選択しながら、動作ツリー内を指定されたレベル数まで下に移動します。

Random-Walk

パラメータ：

（なし）

現在のシステム状態から状態空間の自動探索を実行します。ランダム探索は、あるシステム状態から複数の遷移が可能な場合はそのうちの 1 つがランダムに選択される、という考え方に基づいています。探索が開始されると、以下の情報が出力されます。

- 探索の深さ : 探索の最大の深さ。これはコマンド Define-Random-Walk-Depth で設定できます。
- 繰り返し数 : 開始状態から実行するランダム探索の数。これはコマンド Define-Random-Walk-Repetitions で設定できます。

探索が終了するまで、またはコマンド プロンプトから［ `<Return>` ］が押されるか、*Break* コマンドが送信されるまで、探索は継続されます。その後、システムは探索が開始される前の状態に戻ります。

探索が終了または停止すると、統計情報が出力されます。説明についてはコマンド Bit-State-Exploration を参照してください。

Reset

パラメータ：

（なし）

エクスプローラの状態を初期状態にリセットします。このコマンドはエクスプローラ内のすべてのオプションとテスト値を初期値にリセットし、すべてのレポートとユーザー定義ルールをクリアします。現在の状態と現在のルートも、元の開始状態に設定します。

このコマンドは `.valinit`（UNIX の場合）または `valinit.com`（Windows の場合）ファイルを読み込みます（コマンド Exit を参照してください）。エクスプローラを終了して再開することと同じです。コマンド Default-Options と比較してください。

Save-As-Report-File

パラメータ：

```
<File name>
```

現在のレポートを、指定されたファイルに保存します。現在のレポート ファイルの名前が、新しいファイルに設定されます。

Save-Coverage-Table

パラメータ：

```
<File name>
```

テスト カバレッジ情報を、指定されたファイルに保存します。テスト カバレッジ テーブルは 2 つの部分、すなわちプロファイリング情報セクションとカバレッジ テーブル詳細セクションで構成されます。これはシミュレータによって生成されるファイルと同じタイプです。

Save-Options

パラメータ：

```
<File name>
```

パラメータで指定された名前で、エクスプローラ コマンド ファイルを作成します。ファイルには、エクスプローラのオプションを定義したコマンドが含まれています。このファイルが読み込まれると（コマンド Include-File を使用）、オプションは保存された値に復元されます。

Save-State-Space

パラメータ：

```
<File name>
```

生成された状態空間を表すラベル付遷移システム（LTS）をファイルに保存します。

注記

状態空間をファイルに保存する前に、Exhaustive-Exploration コマンドを実行しておく必要があります。

Save-Test-Values

パラメータ：

```
<File name>
```

エクスプローラ コマンドを含むコマンド ファイルを生成します。Include-File コマンドで読み込まれると、現在のテスト値定義が再作成されます。

Scope

パラメータ：

（なし）

現在の範囲を出力します。範囲の説明については、コマンド Set-Scope を参照してください。

Scope-Down

パラメータ：

```
<Optional service name>
```

操作呼び出しスタック内で、範囲を 1 ステップ下に移動します。コマンド Stack、Set-Scope、Scope-Up も参照してください。

Scope-Up

パラメータ：
（なし）

操作呼び出しスタック内で、範囲を 1 ステップ上に移動します。コマンド Set-Scope、Stack、Scope-Down も参照してください。

Set-Application-All

パラメータ：
（なし）

エクスプローラの状態空間オプションを、UML コード ジェネレータによって生成されるアプリケーションのセマンティクスに従って状態空間探索を実行するように設定します。タイムアウト値や外部環境のパフォーマンスと比較した UML システムのパフォーマンスについては、何も想定されません。

このコマンドは、以下のコマンドを実行することで、探索モード ファクタを設定します。

```
Define-Transition UML
Define-Scheduling First
Define-Priorities 1 1 1 1 1
```

Set-Application-Internal

パラメータ：
（なし）

エクスプローラの状態空間オプションを、C コード ジェネレータによって生成されるアプリケーションのセマンティクスに従って状態空間探索を実行するように設定します。外部環境の応答時間やタイムアウト値と比して、UML システムが内部動作を実行するのに要する時間は非常に短いと想定されます。

このコマンドは、以下のコマンドを実行することで、探索モード ファクタを設定します。

```
Define-Transition UML
Define-Scheduling First
Define-Priorities 1 2 2 1 2
```

Set-Scope

パラメータ：

```
<PId value> <Optional service name>
```

スタックの一番下の操作呼び出しにおいて、指定されたアクティブ クラスに範囲を設定します。範囲はアクティブ クラス インスタンスへの参照であり、おそらくこのプロセスから呼び出された操作インスタンスへの参照です。範囲は、アクティブ クラス イ

ンスタンスのローカル プロパティを調べるため、ほかにもさまざまなコマンドに使用されます。範囲は、現在のシステム状態に至るまでの遷移で実行されたプロセスに、自動的に設定されます。

コマンド Scope、Stack、Scope-Down、Scope-Up も参照してください。

Set-Specification-All

パラメータ：
（なし）

エクスプローラの状態空間オプションを、タイムアウト値や環境のパフォーマンスと比較した UML システムのパフォーマンスについて何も想定されないように設定します。

このコマンドは、以下のコマンドを実行することで、探索モード ファクタを設定します。

```
Define-Transition Symbol-Sequence
Define-Scheduling All
Define-Priorities 1 1 1 1 1
```

Set-Specification-Internal

パラメータ：
（なし）

エクスプローラの状態空間オプションを、外部環境の応答時間やタイムアウト値に比べて UML システムが内部動作を実行するのに要する時間は非常に短いと想定されるように設定します。

このコマンドは、以下のコマンドを実行することで、探索モード ファクタを設定します。

```
Define-Transition Symbol-Sequence
Define-Scheduling All
Define-Priorities 1 2 2 1 2
```

Show-Mode

パラメータ：
（なし）

現在の実行モードの要約や、Tau エクスプローラの現在の状態についてのその他の情報を表示します。

Show-Options

パラメータ：
（なし）

レポート タイプごとに定義されているレポート アクションを含め、エクスプローラに定義されているすべてのオプションの値を出力します。

Show-Versions

パラメータ：
（なし）

C コード コンパイラのバージョンと、現在実行中のプログラムを生成したランタイム カーネルを表示します。

Signal-Disable

パラメータ：
`<Signal> | '-'`

指定されたシグナルに定義されたすべてのテスト値を無効にします。'-' は「すべてのシグナル」を意味します。使用を再開するには、Signal-Enable を使用します。併せて Connector-Disable も参照してください。シグナルのテスト値は、シグナルを転送するコネクタとシグナルの両方が有効な場合にのみ使用されます。デフォルトでは、すべてのシグナルとコネクタは有効です。

Signal-Enable

パラメータ：
`<Signal> | '-'`

指定されたシグナルに定義されたすべてのテスト値を有効にします。'-' は「すべてのシグナル」を意味します。併せて Connector-Enable も参照してください。シグナルのテスト値は、シグナルを転送するコネクタとシグナルの両方が有効な場合にのみ使用されます。デフォルトでは、すべてのシグナルとコネクタは有効です。

Signal-Reset

パラメータ：
`<Signal> | '-'`

指定されたシグナル用の既存のすべてのテスト値を削除し、代わりにデータ型とシグナル パラメータ用の現在のテスト値の設定を使用して、テスト値のデフォルト設定を定義します。'-' は「すべてのシグナル」を意味します。

Stack

パラメータ：
（なし）

範囲によって定義されている PId の操作呼び出しスタックを出力します。スタック内の各エントリについて、インスタンスの型（操作／プロセス）、インスタンス名、および現在の状態が出力されます。コマンド Set-Scope、Scope-Down、Scope-Up も参照してください。

Top

パラメータ：
（なし）

現在の経路の開始状態（ルート システム状態）まで、動作ツリーを上に移動します。

Tree-Search

パラメータ：
（なし）

現在のシステム状態から状態空間の樹状探索を実行します。樹状探索は、すべての組み合わせ可能なアクションが実行される探索です。探索されるツリー（樹）は、ナビゲータ機能 (または Next / List-Next コマンドを使用した手動探索）を使用して手動で調査できるツリーとまったく同じです。

樹状探索の深さには限界があり、Define-Tree-Search-Depth コマンドによって定義されます。デフォルトの深さは「100」です。

このコマンドは、*Break* コマンドを送信することによって中止できます。

Tree-Walk

パラメータ：

```
<Timeout> <Coverage>
```

現在のシステム状態から状態空間の自動探索を実行します。Random-Walk とは対照的に、Tree-Walk は、到達可能性グラフ内の各種状態から開始して、深さを増しながら樹状探索のシーケンスを実行する決定論アルゴリズムに基づいています。樹状ウォークは、深さを優先した探索と横型探索を組み合わせた手法です。到達可能性グラフの深いレベルに位置する状態を探索すると同時に、特定の状態への短い経路を検出できます。樹状ウォークは、シンボル カバレッジ発見的方法によって導かれます。このため、特に自動テスト ケース生成に適しています。

時間が <Timeout> （分で指定）を超えるか、目標とするカバレッジ（パーセントで指定）に到達すると、計算は停止します。または、*Break* コマンドを送信することで、探索は任意の時点で停止できます。

樹状ウォークは “TreeWalk” レポートを作成します。

Up

パラメータ：

```
<Number of levels>
```

動作ツリー内を指定されたレベル数まで上に移動します。Up 1 が指定されると、現在のシステム状態の親状態に移動します。パラメータが大きすぎる場合、Up は動作ツリーのルート（開始状態）で停止します。

ユーザー定義ルール

ユーザー定義ルールは、状態空間探索時に遭遇したシステム状態のプロパティを調べるために使用します。ユーザー定義ルールが true となるシステム状態が見つかった場合、List-Reports コマンドの実行時に他のレポートと共に表示されます。探索時に複数のユーザー定義ルール レポートを生成できます。各ルールに対して可能な各値の代入について、1 つのレポートが出力されます。値の代入は、Print-Evaluated-Rule コマンドで出力される値です。

ルールは、基本的に、ある 1 つのシステム状態のプロパティを記述する述語の定義を可能にします。ルールは、以下で説明する述語とそれに続くセミコロン（;）で構成されます。ルールでは、すべての識別子と予約語は、それが一意であれば短縮できます。

注記

1 度に使用できるルールは 1 つだけです。複数のルールが必要な場合、以下に示すブール演算子を使用して、ルールを 1 つのルールに作り直します。

述語

述語には、以下の種類があります。

- アクティブ クラス インスタンスの限定子および入力ポートのシグナル。
- 「and」、「not」、「or」などのブール演算子述語。
- 「=」や「>」などの関係演算子述語。

述語は、かっこでまとめることができます。

限定子

以下に示す限定子は、インスタンスまたはシグナルのアクティブ クラスを示すルール変数の定義に使用します。ルール変数は、述語のアクティブ クラスまたはシグナル関数で使用できます。

```
exists <RULE VARIABLE> [: <ACTIVE CLASS TYPE>]
[ | <PREDICATE>]
```

この述語は、指定した述語が true となるアクティブ クラス インスタンス（指定タイプの）が存在するときに、true となります。アクティブ クラス タイプと述語は、両方とも省略できます。アクティブ クラス タイプを省略すると、すべてのアクティブ クラス インスタンスがチェックされます。述語を省略すると、それは true とみなされます。

```
all <RULE VARIABLE> [ : <ACTIVE CLASS TYPE>]
[ | <PREDICATE>]
```

この述語は、指定した述語が true となるすべてのアクティブ クラス インスタンス（指定タイプ）で、true となります。アクティブ クラス タイプと述語は、両方とも省略できます。アクティブ クラス タイプを省略すると、すべてのアクティブ クラス インスタンスがチェックされます。述語を省略すると、それは true とみなされます。

```
siexists <RULE VARIABLE> [ : <SIGNAL TYPE>]
[ - <ACTIVE CLASS INSTANCE>] [ | <PREDICATE>]
```

この述語は、指定した述語が true となる指定したアクティブ クラスの入力ポートにシグナル（指定タイプ）が存在するときに、true となります。シグナル タイプを指定しないと、すべてのシグナルが考慮されます。アクティブ クラス インスタンスがまったく指定されていない場合、すべてのアクティブ クラス インスタンスの入力ポートが考慮されます。述語を指定しないと、それは true とみなされます。指定アクティブ クラスは、以前 exists または all 述語で定義されたルール変数、またはアクティブ クラス インスタンス識別子（<ACTIVE CLASS TYPE>:<INSTANCE NO>）です。

```
siall <RULE VARIABLE> [ : <SIGNAL TYPE>]
[ - <ACTIVE CLASS INSTANCE>] [ | <PREDICATE>]
```

この述語は、指定した述語が true となる指定したアクティブ クラスの入力ポートのすべてのシグナル（指定タイプ）に対して、true となります。シグナル タイプを指定しないと、すべてのシグナルが考慮されます。アクティブ クラスがまったく指定されていない場合、すべてのアクティブ クラス インスタンスの入力ポートが考慮されます。述語を指定しないと、それは true とみなされます。指定アクティブ クラスは、以前 exists または all 述語で定義されたルール変数、またはアクティブ クラス インスタンス識別子（<ACTIVE CLASS TYPE>:<INSTANCE NO>）です。

ブール演算子述語

以下のブール演算子が含まれます（従来の解釈による）。

```
not <PREDICATE>
<PREDICATE> and <PREDICATE>
<PREDICATE> or <PREDICATE>
```

演算子は優先度順に表示されますが、かっこを使用して優先度を変更できます。

関係演算子述語

以下の関係演算子述語があります。

```
<EXPRESSION> = <EXPRESSION>
<EXPRESSION> != <EXPRESSION>
<EXPRESSION> < <EXPRESSION>
<EXPRESSION> > <EXPRESSION>
<EXPRESSION> <= <EXPRESSION>
<EXPRESSION> >= <EXPRESSION>
```

これらの述語の解釈は従来どおりです。演算子は、それが定義されているデータ型に対してのみ適用できます。

式

関係演算子述語に使用できる式の分類は、以下のとおりです。

- アクティブ クラス関数：アクティブ クラス インスタンスから値を抽出します。
- シグナル関数：シグナルから値を抽出します。
- グローバル関数：システム状態のグローバルな側面を検査します。
- UML リテラル：従来の UML 定数値。

アクティブ クラス関数

ほとんどのアクティブ クラス関数は、クラス インスタンスをパラメータとして持ちます。このクラス インスタンスは、以前 exists または all 述語で定義されたルール変数、クラス インスタンス識別子（<ACTIVE CLASS TYPE>:<INSTANCE NO>）、またはクラス インスタンスを返す関数（たとえば、sender または from）です。

```
state( <ACTIVE CLASS INSTANCE> )
```

クラス インスタンスの現在の状態を返します。

```
type( <ACTIVE CLASS INSTANCE> )
```

クラス インスタンスのタイプを返します。

```
iplen( <ACTIVE CLASS INSTANCE> )
```

クラス インスタンスの入力ポート キューの長さを返します。

```
sender( <ACTIVE CLASS INSTANCE> )
```

アクティブ クラス インスタンスの命令演算子 sender（クラス インスタンス）の値を返します。

```
parent( <ACTIVE CLASS INSTANCE> )
```

アクティブ クラス インスタンスの命令演算子 parent （クラス インスタンス）の値を返す。

```
offspring( <ACTIVE CLASS INSTANCE> )
```

アクティブ クラス インスタンスの命令演算子 offspring（クラス インスタンス）の値を返します。

```
self( <ACTIVE CLASS INSTANCE> )
```

アクティブ クラス インスタンスの命令演算子 self （クラス インスタンス）の値を返します。

```
signal( <ACTIVE CLASS INSTANCE> )
```

アクティブ クラス インスタンスが UML 状態にあるときに消費されるシグナルを返します。それ以外の場合は、アクティブ クラス インスタンスが遷移の途中であれば、最後の入力文で消費されたシグナルを返します。

```
<ACTIVE CLASS INSTANCE> -> <VARIABLE NAME>
```

指定した変数の値を返します。<ACTIVE CLASS INSTANCE> が以前定義したルール変数である場合は、ルール変数を定義した exists または all 述語も、アクティブ クラス タイプ仕様を含む必要があります。

```
<RULE VARIABLE>
```

値 <RULE VARIABLE> のアクティブ クラス インスタンス値を返します。これは、exists または all 述語でアクティブ クラス インスタンスに結び付けられたルール変数です。

シグナル関数

ほとんどのシグナル関数は、パラメータとしてシグナルを必要とします。このシグナルは、siexists または siall 述語で以前定義されたルール変数か、シグナルを返す関数（例：signal）です。

```
sitype( <SIGNAL> )
```

シグナルのタイプを返します。

```
to( <SIGNAL> )
```

シグナルの受信者のアクティブ クラス インスタンス値を返します。

```
from( <SIGNAL> )
```

シグナルの送信者のアクティブ クラス インスタンス値を与えます。

```
<RULE VARIABLE> -> <PARAMETER NUMBER>
```

指定したシグナル パラメータの値を返します。ルール変数を定義する siexists または siall 述語は、シグナル タイプ仕様も含む必要があります。

```
<RULE VARIABLE>
```

シグナル値 <RULE VARIABLE> を返します。これは、siexists または siall 述語でシグナルにバインドされているルール変数です。

グローバル関数

```
maxlen( )
```

システム内の最長の入力ポート キューの長さを与えます。

```
instno( [<PROCESS TYPE>] )
```

タイプ <PROCESS TYPE> のインスタンスの数を返します。<PROCESS TYPE> を省略すると、アクティブ クラス インスタンスの合計が返されます。

```
depth( )
```

動作ツリー／状態空間内の、現在のシステム状態の深さを与えます。

UML リテラル

```
<STATE ID>
```

UML 状態の名前。

```
<ACTIVE CLASS TYPE>
```

アクティブ クラス タイプの名前。

```
<ACTIVE CLASS INSTANCE>
```

<ACTIVE CLASS TYPE>:<INSTANCE NO> 形式のアクティブ クラス インスタンス識別子（例：Initiator:1）。

```
<SIGNAL TYPE>
```

シグナル タイプ名。

```
null
```

UML ヌル アクティブ クラス インスタンス値

```
env
```

環境のアクティブ クラス インスタンスの値、UML システムの環境から送られたすべてのシグナルの送信者を返します。

```
<INTEGER LITERAL>
true
false
<REAL LITERAL>
<CHARACTER LITERAL>
<CHARSTRING LITERAL>
```

Tau のカスタマイズ

「Tau のカスタマイズ」セクションの各章では、Tau に新機能を開発して追加する方法について説明しています。

55

Tau のカスタマイズ

この章では、以下のようなカスタマイズ方法について説明します。

- ユーザー インターフェイスの拡張
- コマンド ラインからの Tau の起動
- UML モデルの情報内容の拡張
- プログラム可能なインターフェイスを使用して UML モデルの内容にアクセスし、変更する
- Tcl アドインの作成と使用
- セマンティック チェッカーの拡張
- ファイル／フォルダ インポータに統合されたカスタム拡張モジュールの定義

また、さまざまなツール設定を制御するステレオタイプと属性の参照方法についても説明します。

概要

Tau には、標準機能を拡張するいろいろな方法があります。ユーザー インターフェイスの拡張といった単純なものから、UML モデルに保存された情報や表示方法の全面的な変更まで、多様なカスタマイズを行うことができます。この章では、これらのカスタマイズの概要について説明します。

TCL API

スクリプティング ソリューションを選択する場合、スクリプト言語としては Tcl を推奨します。Tcl API を使用するプログラムまたはスクリプトを追加して、ツールの振る舞いをカスタマイズできます。

COM API

高級言語を使用する場合、パフォーマンスを重視する複雑なアプリケーションに適した COM API が用意されています。

C++ API

高級言語を使用する場合、パフォーマンスを重視する複雑なアプリケーションに適した C++ API が用意されています。

エージェント

エージェントを使用すると再利用可能なカスタマイズ用の機能を実装できます。エージェントは上記の API を使用して実装できます。エージェントを使用すると、UML モデルで頻繁に使われる振る舞いを定義し、実装 API にかかわらない決まった方法でその振る舞いを起動できるようになります。エージェントは異なる API で実装された機能の橋渡しとしての役割も果たしています。

エージェントは、クエリ、ダイアグラムジェネレータ、プロパティエディタのカスタマイズ、コードジェネレータのカスタマイズなど多くの機能のカスタマイズに活用できます。

オブジェクト モデル

プログラムを使用して UML モデルの内容にアクセスしたり、修正を行う場合は、以下を区別して考えることが重要です。

- オブジェクト モデル
- UML メタモデル
- UML プロファイル

オブジェクト モデルは、UML ツール セットで作成され、UML ファイル（.u2 ファイル）に保存される情報です。オブジェクト モデルは、約 200 のクラスと多数の属性、関連で構成されます。Tcl スクリプトを使用するか、あるいは COM API または C++

API をベースとしたプログラムを使用する場合は、オブジェクト モデルの詳細を理解している必要があります。これらのアプリケーションが、直接オブジェクト モデル クラスとその機能に作用するからです。

オブジェクト モデルの詳細については、Tau カスタマ サポートにお問い合わせください。

UML メタモデル

メタモデルは、<<metamodel>> によってステレオタイプ化された UML パッケージです。メタモデルは、UML モデルに格納されている情報に対する特定のビューを提供します。実際には、どのメタモデルが使用されるかによって、[モデル ビュー]、プロパティ ページ、およびモデル ナビゲータで表示される内容が決定します。あるアプリケーション ドメインに特化した詳細なカスタマイズを行う場合は、新しいメタモデルを使ってビューを新規に作成すると効率的です。

UML プロファイル

プロファイルは、<<profile>> によってステレオタイプ化された UML パッケージです。プロファイルの機構によって以下のような操作が簡単に行えます。

- さまざまな UML モデル要素に情報を追加する。
- アプリケーション モデリングに使用する新しい概念を導入する。
- グラフィカル エディタで使用できる新しいシンボルを導入する。

プロファイルを設計する場合、必ず特定のメタモデルをベースにする必要があります。通常は、ツールをインストールすると提供される **TTDMetamodel** を利用するのが便利です。この場合は、UML モデルに情報を追加する際に新しいメタモデルを作成する必要はありません。

オブジェクト モデル、メタモデル、プロファイルの関係を 1729 ページの図 272 に示します。

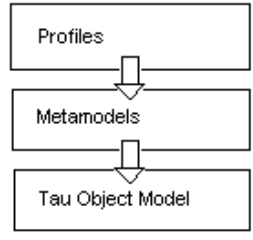


図 272

プロファイルを追加すると、UML モデルに格納されている情報を拡張できます。プロファイルはメタモデルをベースに定義します。メタモデルは基盤となる Tau のリポジトリのビューを提供し、あらかじめ組み込まれたオブジェクト モデルをベースに定義されます。

アドイン

設計をカスタマイズする際に考慮すべき重要な考え方がもう 1 つあります。「アドイン」のコンセプトです。アドインは、プロファイル、メタモデル、Tcl スクリプト、COM ベースのアプリケーション、カスタマイズに含めるその他のファイルなどを包含するコンテナとして機能します。したがって、Tau に組み込まれた機能を拡張する場合には、アドインがポイントとなります。本セクションでは、アドインに持たせることができるさまざまな項目について重点的に説明します。

IBM RationalEnterprise Architect for DoDAF は、テンプレートとヘルプ情報を備えたアドインであり、Tau とは別に配布されます。

SysML は Tau とともに配布されるアドインです。

ステレオタイプと属性

Tau が提供する UML パッケージでは、さまざまなプロファイルが利用できます。こらのパッケージは、［Library］フォルダ下のモデルで見つけることができます。

このプロファイルにあるステレオタイプを適用し、ステレオタイプの属性に値を指定することで、さまざまなツール機能を制御する設定を変更できます。ステレオタイプの属性は、コード生成の設定、モデルからファイルへのマッピング スキーマ、およびコンピュータ、OS、make、コンパイルなどで使用される規則として何を採用するかを制御します。

コマンド ラインからの起動

引数として **Tcl** スクリプトを使用した **Tau** の起動

Tau は、Tcl スクリプトを引数として指定してコマンド ラインから起動できます。さらに、パラメータをスクリプトに渡すことができます。

例 **611:** コマンド ラインからの **Tau** の起動 ——————————————————————

Windows

```
VCS.EXE -Script u2merge.tcl forceVersion1
reviewDifferencesAlways suppressSetupNever false
C:/work/version1/project.ttp C:/work/version1/project.ttp
C:/work/version2/project.ttp [C:/work/ancestor/project.ttp
[C:/work/version2ancestor/project.ttp]]
```

UNIX

```
tau -script u2merge.tcl forceVersion1 reviewDifferencesAlways
suppressSetupNever false /home/user/version1/project.ttp
```

```
/home/user/version1/project.ttp
/home/user/version2/project.ttp
[/home/user/ancestor/project.ttp
[/home/user/version2ancestor/project.ttp]]
```

Tau が u2merge.tcl スクリプトとともに起動されます。スクリプト名の後のパラメータがスクリプトに渡されます。

Tcl でのパラメータの解析方法などについては、以下の例を参照してください。

```
$TAU_INSTALLATION/etc/u2merge.tcl.
```

注記

–Script オプションは、コマンド ラインで最後のオプションとして指定する必要があります。また、*.ttw または *.ttp ファイルのロードなどの機能と組み合わせることはできません。スクリプト自体で、Tcl API を使用してロードを行わなければなりません。

ワークスペース（***.ttw**）またはプロジェクト（***.ttp**）を指定した **Tau** の起動

Tau は、ワークスペースまたはプロジェクトを指定して起動できます。

例 **612:** プロジェクトを指定した **Tau** の起動

Windows

```
VCS.EXE C:¥MyProject¥Version1¥MyProject.ttp
```

UNIX

```
tau /home/me/MyProject/Version1/MyProject.ttp
```

スプラッシュ画面の抑止

Windows で Tau のスプラッシュ画面を抑止するには、以下のように指定します。

```
VCS.EXE -SuppressSplashScreen
```

UNIX で Tau のスプラッシュ画面を抑止するには、以下のように指定します。

```
tau -SuppressSplashScreen
```

注記

ここで説明したコマンド ライン パラメータでは、大文字と小文字を区別しません。

アドイン

アドインの適用領域

アドインは、独自開発の機能をパッケージ化する際に使用できる基本的な機構を提供します。Tau にはあらかじめ多くのアドインが組み込まれており、コード生成などの機能を実現するために利用されています。

アドインの起動

アドインは、[ツール] メニューから [カスタマイズ] を選択して [カスタマイズ] ダイアログを開き、[アドイン] タブから起動します。

組み込みアドインは、インストール ディレクトリの addins ディレクトリにあります。通常は以下の場所です。

```
C:¥Program Files¥IBM¥Rational¥TAU¥4.3¥¥addins¥
```

アドインごとにサブディレクトリがあり、以下のファイル、ディレクトリがあります。

- <addin name>.mod ファイル
- アドインの一部である実行形式ファイルまたは共有ライブラリを含む「bin」サブディレクトリ
- Tcl スクリプト ファイルを含む「Script」サブディレクトリ
- その他のファイル（通常はプロファイル定義を持つ UML モデル）を含む「Etc」サブディレクトリ

<addin name>.mod ファイルは、アドインのプロパティを定義し、そのアドインに含まれる他のファイルをリストします。

ユーザー アドインはユーザーごとに決めたディレクトリに配置されます。

```
C:¥Documents and Settings¥<USER>¥Application
Data¥IBM¥Rational¥Tau 2¥addins¥
```

ユーザー アドインのディレクトリ構造は、組み込みアドインと同じです。

新規のアドインは、このユーザー アドイン ディレクトリに配置されます。Tau を使用して新規アドイン モジュールの作成を行う場合、そのディレクトリがなければ、新たに作成されます。

ユーザー定義アドインは、環境変数 TAU_USER_ADDINS_DIR で定義した場所に格納できます。この変数を設定すると、新しいアドインは以下の場所に格納されます :

```
%TAU_USER_ADDINS_DIR%¥addins
```

グループで共有したアドインを以下の場所に格納できます。

グループ向けのアドインはデフォルトで以下のディレクトリにあります :

```
C:¥Documents and Settings¥<USER>¥Application
Data¥IBM¥Rational¥Shared¥TeamAddins
```

グループ用のアドインの格納場所は、TAU_TEAM_ADDINS_DIR 環境変数で上書きできます。

```
%TAU_TEAM_ADDINS_DIR%
```

部門や会社全体で共有したい場合は、以下のディレクトリに格納されます:

```
C:¥Documents and Settings¥<USER>¥Application
Data¥IBM¥Rational¥Shared¥CompanyAddins
```

この場所も、環境変数 TAU_COMPANY_ADDINS_DIR で上書きできます。

```
%TAU_COMPANY_ADDINS_DIR%
```

定義済み の URN マッピング、u2useraddins、u2teamaddins、u2companyaddins があります。 これらをアドインディレクトリに指定して環境変数の代わりに使用できます。これらの URN マッピングは上書きできません。

アドインの内容と構造

以下の例でアドインの内容を説明します。独自のアドインを作成するには、必要なファイルを一揃い作成し、アドイン ディレクトリに保存します。

新規アドイン モジュールの作成

Tau を使用してアドインの設定をするには、［ツール］メニューから［カスタマイズ］を選択し、［カスタマイズ］ダイアログで［アドイン］タブを選択します。［作成］をクリックします。［新しいアドイン モジュールの作成］ダイアログが表示されます。それぞれのタブに値を入力すると、アドインとして「.mod」ファイルが作成されます。

例 **613: MyAddin** アドインのモジュール ファイル

MyAddin アドインの「.mod」ファイルの例を以下に示します。

```
[MyAddin]
"scope"="PROJECT"
"version"="1.0"
"longname"="My Addin"
"description"="Gives an example of a .mod file."
"product"="u2"
[MyAddin/Bin]
"listBin"=""
[MyAddin/Script]
"listScript"="MyAddinScript.tcl"
[MyAddin/Etc]
"listEtc"="MyAddinProfile.u2"
```

- 1733 ページの例 613 の文字列「MyAddin」は［新しいアドイン モジュールの作成］ダイアログの［識別子］フィールドに表示されます。この文字列が［アドイン］タブに表示され、起動中の場合はチェックマークが付いています。

- "scope" プロパティは、アドインをロードするタイミングを定義します。
  - 値が "PROJECT" の場合、アドインはプロジェクトのロード時のみロードされます。
  - 値が "GENERAL" の場合、プロジェクトのない状態で Tau を起動したときもロードされます。
- "version"、"longname"、"description" プロパティ値は、アドインに関するユーザー情報を提供するためにさまざまなダイアログで使用されます。"longname" プロパティの値は［新しいアドイン モジュールの作成］ダイアログの［名前］フィールドに表示されます。"description" プロパティのテキスト値は、アドインの起動 / 停止を指定するダイアログに表示されます。
- "product" プロパティには、アドインをどのツールで使用するかを指定します。UML プロジェクト用のアドインでは、互換性の理由から "u2" を指定します。"scope" プロパティを "GENERAL" にした場合、"product" プロパティには "" を指定します。［新しいアドイン モジュールの作成］ダイアログを使って値を変更する場合は、［Tau］を選択すると "product" プロパティが "u2" に設定されます。
- "listBin" プロパティの値は、通常カスタマイズには不要なので、ブランクにします。"listBin" プロパティは、［バイナリ］タブにリストされるファイル名に対応しています。
- "listScript" プロパティの値は、［スクリプト］タブにリストされるファイル名に対応しています。アドインを起動すると、このリストに表示された Tcl スクリプトが実行されます。自動で起動させたい Tcl スクリプトはすべてここに入力します。
- "listEtc" プロパティは、［その他のファイル］タブにリストされるファイル名に対応しています。"listEtc" は、プロジェクトのロード時にロードされる *.u2 ファイルの名前がために使用されます。
- 「.mod」ファイルを適切なサブディレクトリに配置すると、［カスタマイズ］ダイアログ内の［アドイン］タブに、指定したアドインが表示されます。

注記

テキスト エディタで「.mod」ファイルを変更する場合、複数のファイル名を挿入するときには、区切り文字をセミコロンにします。

アドイン **Tcl** スクリプト実行

Tau に特定の機能を追加する目的で作成するアドインは、Tau から起動できる Tcl スクリプトとして実装する必要があります。すべての機能を Tcl API を使用した Tcl スクリプトで実装することもできますが、機能の本体を COM API や C++ API を使用して実装し、Tau からの起動の部分のみを Tcl コマンド u2::InvokeAgent を使用した Tcl スクリプトで実装するという方法もあります。

Tau でアドインを無効にすると、そのタイミングで Tcl プロシージャ BeforeUnload が実行されます。このプロシージャは、ライブラリやプロファイルのアンロードなど、作業のクリーンアップを行う目的で使用できます。

ユーザー インターフェイスのカスタマイズ

概要

Tau のユーザー インターフェイスを機能拡張する簡単な方法として、Tcl スクリプトのアドインの利用があります。

アドインの作成

1732 ページの「アドイン」で説明した “MyAddin” を機能拡張する例を下に示します。このためにメニュー項目［MyCommand］を持つ［MyMenu］メニューを追加する単純なスクリプトを用意します。［MyCommand］は起動されるとメッセージ ボックスを表示します。

まず、アドインの `script` サブディレクトリに `MyAddinScript.tcl` という名前で Tcl ファイルを作成します。次に、以下のテキストをコピーして `MyAddinScript.tcl` に貼り付けます。

```
std::Output "MyAddin loading ... "

package require commands
package require dialogs

proc OnMyCommand {cmd} {
  std::MessageDialog -name "My message dialog" ¥
  -message "This is my message!"
}

proc OnTrue {e} {
  return 1
}

proc Init {} {
  std::AddCommand -variable cmdMyCommand ¥
  -name "My command" -statusmessage ¥
  "My command status msg" -tooltip ¥
  "My tooltip" -imagefile "" ¥
  -OnActivateCommand OnMyCommand ¥
  -OnEnableCommand OnTrue
  std::AddMenu -variable MyMenu ¥
  -commands { cmdMyCommand } ¥
  -path { &MyMenu } -position {after &Tools}
}

Init

std::Output "Done.¥n";
```

アドインのロード

［ツール］->［カスタマイズ］コマンドで開いた［カスタマイズ］ダイアログでアドインを起動すると、新しいメニューが表示されます。このメニューで［My Command］を選択すると、指定したテキストを持つメッセージ ボックスがポップアップ表示されます。

注記

アドインには PROJECT スコープが設定されているので、アドインを起動する前に必ず UML プロジェクトをロードしてください。

参照

第 59 章「Tcl API」の 1928 ページ、「ユーザー インターフェイス アドイン固有コマンド」

第 59 章「Tcl API」の 1906 ページ、「汎用コマンド」

プロファイル

プロファイルの適用領域

プロファイルは UML 言語の拡張に使用します。具体的には以下のことを実行できます。

- 既存の UML 概念に情報を追加する。
- Tau に組み込まれた UML 概念に加えて、新しい概念を導入する。
- Tau に組み込まれた UML ダイアグラムのシンボルに加えて、新しい図形シンボルを導入する。
- Tau のカスタマイズされた動作をエージェントとして実装する

プロファイルは、UML で <<profile>> ステレオタイプ パッケージとして定義されます。したがって、プロファイルを作成するには、通常のパッケージを作成し、パッケージに定義済みの <<profile>> ステレオタイプを適用します。<<profile>> パッケージ内では、ステレオタイプによって新しい概念が定義され、ステレオタイプの属性によって UML モデルに格納される追加情報が定義されます。

注記

プロファイルにあるすべての最上位の定義は、修飾なしで、ユーザー モデル内のすべての場所で表示されます。これは、プロファイルが読み込まれるときに、最上位の <<access>> 依存関係が自動的に追加されるからです。

プロファイルの生成

UML モデルの各クラスに以下の 2 つの情報を追加するプロファイルを作成する方法を以下に示します。

- クラスの作者（文字列）
- クラスの作成日（文字列）

このプロファイルを定義するには、以下のことを行うだけですみます。

1. <<profile>> パッケージを作成します。パッケージを別ファイルに入れます。この作業は厳密には必要ありませんが、プロファイルの適用を簡単にします。
2. <<profile>> パッケージにクラス図を追加します。
3. ダイアグラム内に新しいステレオタイプを作成します。ここでは、「ClassInfo」という名前にします。これには、ツールバーから［ステレオタイプシンボル］を選択して、ダイアグラム内をクリックします。
4. ［モデル ビュー］で次の手順を行います。［ライブラリ］フォルダの下にある **TTDMetamodel** パッケージを開きます。TTDMetamodel 内の「Class」というクラスを見つけ、クラス図内の上記で作成した「ClassInfo」の隣にドラッグします。「Class」が <<metaclass>> としてステレオタイプ化されたことを確認してください。詳細は後で説明します。

5. 「ClassInfo」に「Author」と「CreatedDate」という 2 つの属性を「Charstring」型として定義します。
6. 「ClassInfo」ステレオタイプから「Class」クラスへの拡張ラインを作成します。
7. 拡張ラインに関連付けられたテキスト フィールドに「1」と入力します。
8. これで終了です。結果は、1738 ページの図 273 のようになります。

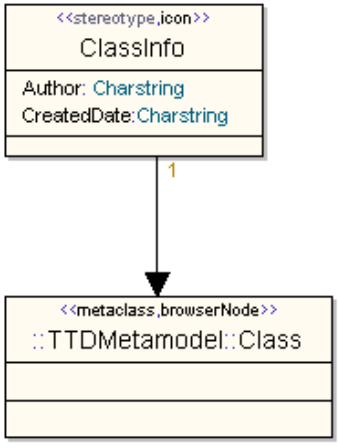


図 273

これで、プロファイルが作成できました。このプロファイルを適用すると、モデル内のすべてのクラスに作者と作成日の情報が追加されます。ユーザーの視点から見ると、自分のクラスのプロパティの［フィルタ］から［Class Info］を選択して表示されるページに、2 つのエントリとして作者と作成日が表示されます。

プロファイルのテスト

プロファイルをテストするには、ダイアグラム内で他のクラスを作成し、そのクラスの［プロパティ］ペインを表示します。作成したクラスの［プロパティ］ペインに作者と作成日が表示されるはずです。

プロファイルの使用

プロファイル設計の次のステップは、作成したプロファイルを他のユーザーにも公開することです。これには、アドイン機能を使うことを推奨します。以下の 3 つの手順を行います。

- アドインの `Etc` サブディレクトリにプロファイルを追加します。
- アドインの一部である Tcl スクリプトの 1 つに、`LoadLibrary` コマンドを追加します。Tcl スクリプトは、`"listScript"` プロパティで指定する必要があります。

- アドインの「.mod」ファイルで、プロファイルを含むファイルを指定します。厳密にはこの手順は必要ありませんが、ツールの今後のバージョンとの互換性のため、実行することを推奨します。

MyAddin/Etc ディレクトリの MyAddinProfile.u2 というファイルにプロファイルが保存されていると仮定した場合、アドインの Tcl スクリプトに以下のコードを追加してプロファイルをロードできます。

```
set ProfilePath [std::GetUserAddinsDirectory]/MyAddin/Etc/MyAddinProfile.u2
std::Output "Loading MyAddinProfile.¥n"
u2::LoadLibrary $ProfilePath
```

「.mod」ファイルの listEtc フィールドを使用して、ロードする「.u2」ファイルを指定できます。

「.mod」ファイルに「urn」を関連付けることもできます。

```
[MyAddin/Etc]
"listEtc"="urn:u2useraddins:MyAddin/Etc/MyAddinProfile.u2"
```

注記

1 つのプロファイルが複数の .u2 ファイルに分かれている場合、そのうちの 1 つがプロファイルの最上位になっているようにしてください。また、.mod ファイルにその階層順でファイルを記述してください。

プロファイルの詳細

アドインが有効になると、［モデル ビュー］の［ライブラリ］フォルダに他のプロファイルとともに表示されます。

ここまで、プロファイルを使用して既存の UML クラスに情報を追加する単純な例について説明しました。さらに、ステレオタイプを分類手段として使用して、一般的な UML 概念のセマンティックをきめ細かく定義することもできます。たとえば、UML に新しい概念「Documentation artifact」を導入し、アーティファクトを 1 つのドキュメント アーティファクトとマーキングできるようにする例を考えてください。このためには、すでに説明した方法でプロファイルを定義し、ステレオタイプを「Documentation」という名前にして、前の例の「Class」の代わりに TTDMetamodel の「Artifact」メタクラスから拡張します。さらに、拡張ラインに「1」ではなく「0..1」と入力します。結果は、1740 ページの図 274 のようになります。

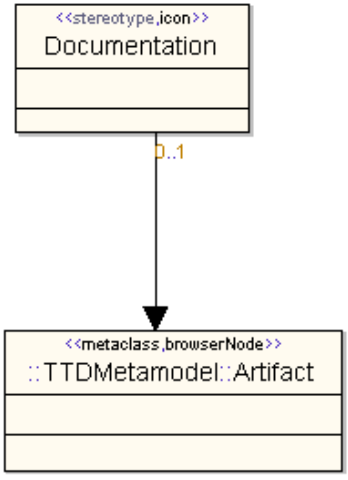


図 274:

また、<<Document>> アーティファクトを示す専用のアイコンを定義してダイアグラムで使用することもできます。これには、作成した <<Document>> ステレオタイプに、定義済みステレオタイプ **TTDStereotypeDetails::icon** を適用する必要があります。適用がすむと、<<Document>> ステレオタイプのプロパティのフィルタ［Icon］のページに、<<Document>> アイコン用のビットマップが保存されているファイルへのパスを定義できます。［Icon File］プロパティで指定するパス名は絶対パスですが、インストール ディレクトリを参照する環境変数 `$INSTALLPATH` も使用できます。

この手順を終了したら、<<Document>> によってステレオタイプ化されたアーティファクト シンボルのショートカット メニューの［アイコン モード］を有効にして、通常のアーティファクト シンボルの代わりに指定したビットマップを表示できます。

上級ユーザー向けとして、［プロパティ］ペインでのステレオタイプ属性の表示方法を、値の表示と編集に使用される編集コントロールの定義によってカスタマイズすることも可能です。詳細については、Tau に組み込まれたプロファイル、TTDExtensionManagement プロファイルに定義されています。このプロファイルは、<<profile>> パッケージ内のステレオタイプ定義に適用できる、数多くのステレオタイプを定義しています。TTDExtensionManagement プロファイルを調べると、さらに何ができるかについての詳細な知識が得られます。このファイルは Tau をインストールすれば提供されます。任意の UML モデルを開けば、［ライブラリ］フォルダで見つけることができます。

参照

プロパティ エディタのカスタマイズ

メタモデル

モデル アクセス

モデル アクセスの適用領域

本章でこれまでに説明したアドインは、メニューおよびダイアログを追加するか、ステレオタイプおよびグラフィック アイコンの導入により、ツールの表示を変更するに過ぎませんでした。真の新機能を追加したわけではありません。現実に有用な機能には、読み取り専用アクセスまたはモデルを変更できる読み取り／書き込み可能なアクセス権限で、現在ロードされているモデルにアクセスできることが必要です。Tau はモデリング ツールです。ツール環境の最終的な目的は、さまざまなタイプの UML モデルを作成することです。

モデル アクセス機能の追加

Tau には、ロードされたモデルにアクセスできる機能を追加する API を提供します。つまり、Tcl スクリプティングやコンパイルされたコードからの C++ API または COM API 呼び出しです。どの方法を使うかは、開発に要する労力と要求される実行パフォーマンスとの得失評価によって決定します。スクリプティングは、プロトタイプ作業や高度な演算処理が不要な、対話形式で実行される小規模な拡張の作成時などに有効です。単純な報告書ジェネレータや小さな変換スクリプトなどがその例です。より高度なアプリケーションには、COM API や C++ API を使ってコーディングされたコンパイル済み ツールのほうが適しています。特にパフォーマンスの点ではこの方法が有利です。UML モデルに基づくプログラミング言語でアプリケーションを作成するコード ジェネレータがその例です。

TCL API の使用

Tcl API の使用法を理解するため、MyAddin 内のスクリプトをさらに拡張します。目的は、［MyAddin］メニューのコマンドを変更して、クラスの作者属性の使用方法についてのメトリクスを計算することです。具体的には、ロードされたモジュール内のクラス数を数えて、そのうちのいくつの作者属性に値が定義されているかを調べます。

実装は以下の Tcl スクリプトになります。

```
set NoOfClasses 0
set NoOfAuthorClasses 0

proc CheckClass {e} {
  global NoOfClasses
  global NoOfAuthorClasses

  if { [u2::IsKindOf $e Class] } {
    set NoOfClasses [expr $NoOfClasses + 1]
    set Author [u2::GetTaggedValue $e "ClassInfo (. Author .)"]
    if {$Author != 0} {
      set NoOfAuthorClasses [expr $NoOfAuthorClasses + 1]
    }
  }
}
proc OnMyCommand {cmd} {
   global $std::activeproject
   global NoOfClasses
```

```
    global NoOfAuthorClasses
    set ActiveModel [std::GetModels -kind U2 -project
$std::activeproject]
    u2::MetaVisit $ActiveModel CheckClass
    set str "The model contains $NoOfClasses classes, of which
$NoOfAuthorClasses have an author."
    std::MessageDialog -name "Author report" -message $str
  }
```

1735 ページの「ユーザー インターフェイスのカスタマイズ」で作成した `MyAddinScript.tcl` の `OnMyCommand` プロシージャを上記のコードに変更した場合、［My Menu］メニューの［My command］を選択するとメトリクス ダイアログがポップアップ表示されます。

スクリプトで使用される主な Tcl コマンドは、モデル間を移動する `u2::MetaVisit` コマンドと、見つけたクラスの作者を抽出する `u2::GetTaggedValue` コマンドです。

セマンティック チェックの追加

セマンティック チェックの適用領域

コード ジェネレータなどのアドインを定義するときに、非常に役立つ拡張機能があります。つまり、［ビルド］メニューまたはツールバーから［ビルド］または［チェック］を選択した際に、Tau に組み込まれたセマンティック チェックとともに実行される、独自のセマンティック チェックを定義できます。

この機能が最も適しているのは、プロファイルで指定した情報が正しく使用されているかを確認するケースです。

例 **614:** 作者属性のチェック

作者属性が現行モデル内のすべてのクラスに与えられていることを確認する方法を以下に示します。これには、前のセクションで導入した `MyAddinScript.tcl` スクリプトに、以下の Tcl コードを追加します。

```
proc SemCheckClass {e} {
  set NoOfClasses [expr $NoOfClasses + 1]
  set Author [u2::GetTaggedValue $e  "ClassInfo (. Author
.)"]
  if {$Author != 0} {
    u2::SemMessage werror "No Author" $e
  }
}

u2::::CreateSemGroup "/" "MyChecks"

u2::CreateSemRule "/MyChecks" "CheckAuthor" Class 30
"SemCheckClass"
```

このスクリプトは、新しいセマンティック チェックのグループ「`MyChecks`」を追加し、さらにこのグループに規則「`CheckAuthor`」を追加します。実際のセマンティック チェックは、モデル内のすべてのクラスから自動的に呼び出されるプロシージャ「`SemCheckClass`」によって実行されます。

注記

このアドインを無効化する際に、有効化したときに追加したセマンティックルールを削除してください。これを行わないと、セマンティックチェッカはすでに利用できなくなっている Tcl スクリプトを呼び出そうとします。この動作はツールのクラッシュを引き起こす可能性があります。下例のように、BeforeUnload プロシージャを使って追加したルールを必ず削除してください。

```
proc BeforeUnload {} {
  u2::DeleteSemEntity "/" "MyChecks"
}
```

セマンティック チェッカ イベントをトリガするエージェントを使用した新しいセマンティックチェッカを作成することもできます。この方法を使うと、C++ API や COM API を使用したチェッカを作成できます。

参照

第 59 章「Tcl API」の 1946 ページ、「セマンティック チェッカ コマンド」

コード ジェネレータの追加

概要

コード ジェネレータは以下のようなコンポーネントで構成されます。

- ビルド ステレオタイプを定義するプロファイル パッケージ
- 実際のコードジェネレータ。通常は実行形式ファイルまたはエージェントの形式を取ります。
- 追加のランタイム ライブラリ

ほとんどの場合、これらはすべて 1 つのアドインにまとめられます。

ビルド ステレオタイプ

このセクションでは、2 つのビルド操作を定義する仮想 C# コード ジェネレータの例を示します。

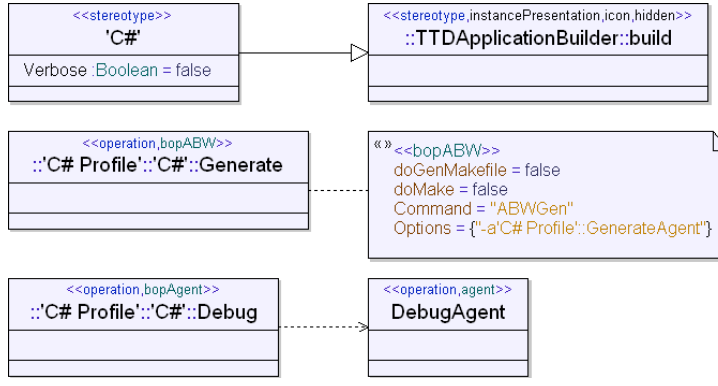


図 275:

ビルド ステレオタイプ <<C#>> は、定義済みの <<build>> ステレオタイプを継承し、「Verbose」属性を定義します（有効化されるとコードジェネレータが追加メッセージを出力します）。また、<<C#>> は、2 つのビルド操作を定義します。つまり、モデルから C# コードを生成する［Generate］と外部デバッガに接続する［Debug］です。したがって、ユーザーが <<C#>> でステレオタイプ化されたビルドアーティファクトを右クリックすると、2 つの選択肢［Generate］と［Debug］が表示されます。

操作「'C#'::Generate」は、<<bopABW>> ステレオタイプによってステレオタイプ化されます。これは、この操作が選択されたときには、コマンド ライン引数「-a'C#Profile'::GenerateAgent」を使用して、外部実行形式ファイル（ABWGen.exe) が起動されることを指定しています。

操作「'C#'::Debug」は、<<bopAgent>> ステレオタイプによってステレオタイプ化され、エージェント「DebugAgent」に依存します。これは、この操作が選択されると、Tau エージェントが起動されることを指定しています。

ABWGen

実行形式ファイル ABWGen は外部プロセスでエージェントを実行するために使用されます。これは、<<bopABW>> ステレオタイプを使用して定義されたビルド操作と組み合わせて使用します。

注記

ABWGen は外部プロセスで実行するため、起動されるエージェントは非対話形式のエージェントです。したがって、TCL で実装されるエージェントは、ABWGen では使用できません。

構文

```
ABWGen [options] [input-file]
```

`input-file`
input-file が付加されると、stdin の代わりにビルド パラメータの読み出しに使用されます。この引数は、主にデバッグで使用します。

`options`

`-v`
冗長モードを有効にします。追加メッセージが表示されます。

`-p<profile-file-name>`
ライブラリとしてロードする u2 ファイルの名前。これにはエージェントが含まれます。

`-E<event-guid>`
発行するイベント。

`-e<event-name>`
発行するイベント。イベントは完全修飾名を使用して指定します。

`-A<agent-guid>`
呼び出すエージェント。

`-a<agent-name>`
完全修飾エージェント名。

注記

'-v' オプション以外のオプションは複数回指定可能です。

イベントまたはエージェントを指定しない場合、GUID 付きのデフォルト イベント名「@TTDAB@Generate」が使用されます。

実行

プログラムはエージェントとイベントを以下の順序で実行します。

1. エージェントパラメータのフォーマット：

   ```
   error list : CeErrorList
   selection  : Cu2Element[*]
   ```
   エージェント コンテキストは常にビルド アーティファクトです。

2. 「-e」および「-E」オプションを使用して与えられたイベントごとに、コマンドラインでの指定順に、「Before processing」が呼び出されます。
3. 「-a」および「-A」オプションを使用して一覧表示された各エージェントが、コマンド ラインでの指定順で呼び出されます。
4. 「-e」および「-E」オプションを使用して与えられたイベントごとに、コマンドラインでの指定とは逆順に「After processing」が呼び出されます。

参照

ツール イベントによってトリガされるエージェント起動

インポータの追加

インポータとは、Tau の外部の情報源に基づいて UML モデル（またはその一部）を作成するためのユーティリティです。インポータの使用目的例は以下のとおりです。

- UML モデルとダイアグラムを使用した情報を、可視化、分析するため。
- 外部のソフトウェアインターフェイスを Tau からアクセスできるようにインポートするため。
- 他のツール、言語で開発したソースコードまたはモデルを Tau にマイグレートするため。

インポータを実行するための Tau GUI は、［インポートウィザード］と呼ばれます。複数のダイアログから構成され、どのインポータを実行するかを選択できます。選択したインポータでは、それぞれさまざまなオプションが指定でき、最終的にそのインポータが実行されます。

インポートウィザードを開くには以下の手順を行います :

1. エンティティを［モデルビュー］で選択します。一部のインポータでは、選択したエンティティを、新しいエンティティを作成する際の基準となる場所とみなします。また、別のインポータでは、新しいエンティティを作成する際に、常にモデル内の決まった場所に作成します。
2. ［ファイル］メニューから［インポート ...］を選択します。

Tau は、出荷時にいくつかの組込み済みインポータを提供します。このインポータはさまざまな UML モデルのインポートをサポートします。有効にしているアドインの種類によって、有効になるインポータの種類は異なります。

新しいインポータの作成

ユーザーは独自のインポータを定義できます。このインポータをインポートウィザードに統合することもできます。これには、`ImportWizard` インターフェイスを実装するクラスを定義します。

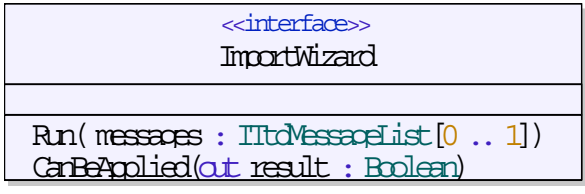


図 276 ImportWizard インターフェイス

このインターフェイスは、`TTDImporters` プロファイルに定義されています。このプロファイルには、このインターフェイスを実装していて、かつ組み込み済みのインポータに対応したクラスもあります。

新しいインポータを作成する手順は以下のとおりです:

1. `ImportWizard` インターフェイスを実装するクラスを作成します。通常はそのクラスは別のパッケージに配置し、別のファイルに保存します。こうすると、アドインを使用してライブラリとしてロードできます。詳細は、プロファイルの生成を参照してください。
2. クラスに内容を説明する名前を付けます。この名前がインポートウィザードに表示されます。
3. クラスに <<icon>> ステレオタイプを適用して、インポータにアイコンを割り当てることもできます。アイコンはインポートウィザードのインポータのリストに表示されます。
4. クラスに実行する内容のコメントを記述します。このコメントテキストは、インポートウィザードで選択した時に表示されます。
5. クラスに操作 `Run` と `CanBeApplied` をエージェントとして定義して、`ImportWizard` インターフェイスを実装します。これらの操作の詳細は、Run と CanBeApplied を参照してください。

CanBeApplied

このエージェントのモデルコンテキストは、インポータのコンテキストです。つまり、インポートウィザード開始時の選択されたエンティティです。エンティティが得委託されていない場合は、コンテキストはモデルノードの最上位です。

エージェントは、'result' out パラメータに対してブール値を割り当てる必要があります。この値が、インポートウィザードに対してインポータを実行できるかどうかを伝えます。実行できる場合は、'true' を、そうでない場合は 'false' を設定する必要があります。

CanBeApplied によってインポータを使用できるかどうかについての条件判断を動的に行うことができます。たとえば、インポータが特定のコンテキスト要素の選択を必要とする場合や、特定のライブラリの存在を必要とする場合などに対応できます。

Run

このエージェントのモデルコンテキストは、インポータのコンテキストです。つまり、インポートウィザード開始時の選択されたエンティティです。エンティティが得委託されていない場合は、コンテキストはモデルノードの最上位です。

このエージェントは、インポータのユーザーインターフェイスを実装する責任があります。インポートウィザードでインポータが選択され、[OK] ボタンが押されたときに呼び出されます。エージェントは Tau 内でダイアグラムなどの GUI をオープンするか、ユーザーインターフェイスを提供する外部プログラムを起動します。

ユーザーが GUI を使って作業している最中にエージェントが何らかのメッセージを出力する必要がある場合は、'message' パラメータを使用できます。ただし、通常は、GUI 内で直接メッセージを出力すべきです。

Import

一部のインポータクラスは、`Import` エージェントを定義しています。このエージェントが実際のインポートの操作を行います。このエージェントへのパラメータは、インポータオプションに対応しています。インポータごとにインポートオプションは異なるため、`Import` は `ImportWizard` インターフェイスの一部ではないことに注意してください。

インポータクラスに `Import` エージェントを定義することにした場合、Run エージェントの実装は、ユーザーインターフェイスからのインポートを実行する際にそのエージェントを使用します。

インポート操作を、Run 操作の内部ではなく、別個のエージェントとして外部に見せる 1 つの利点は、インポータを API（InvokeAgentAPI メソッド）を使ってプログラムからアクセスできることです。

1 つの例

カスタムインポータ作成の例として、Tcl API を使用してシンプルなディレクトリインポータを作成しましょう。このインポータは Tau に組み込まれたファイル / フォルダインポータの簡易バージョンです。

まず、`ImportWizard` インターフェイスを実装するクラスを追加し、名前とコメントを記述します。操作 `Run`、`CanBeApplied`、`Import` をクラスに定義し、各操作のコンテキストメニューから、［ユーティリティ］>［Turn into Tcl agent］コマンドで TCL エージェントに変換します。

結果は以下のようになります。

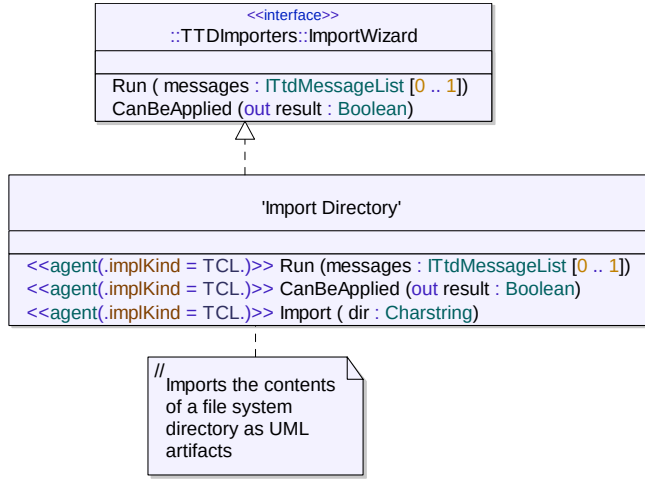


図 277 カスタムインポータの定義

`CanBeApplied` エージェントを実装して行きます。このシンプルなインポータは、常に利用可能ですので、'result' out パラメータには単純に true を設定します。

```
[[
proc CanBeApplied { triggeredBy timing context server
agentParameters } {
  upvar 1 $agentParameters ap
  lset ap 0 1
}
]]
```

'result' は第一番目であり、かつ唯一のエージェントパラメータ（添え字は '0'）であること、Tcl では true は '1' で現されることなどに注意してください。

これで、インポートウィザードを開いて、利用可能なインポータとして新しく作成したインポータが表示されるかどうかを確認する用意が整いました。

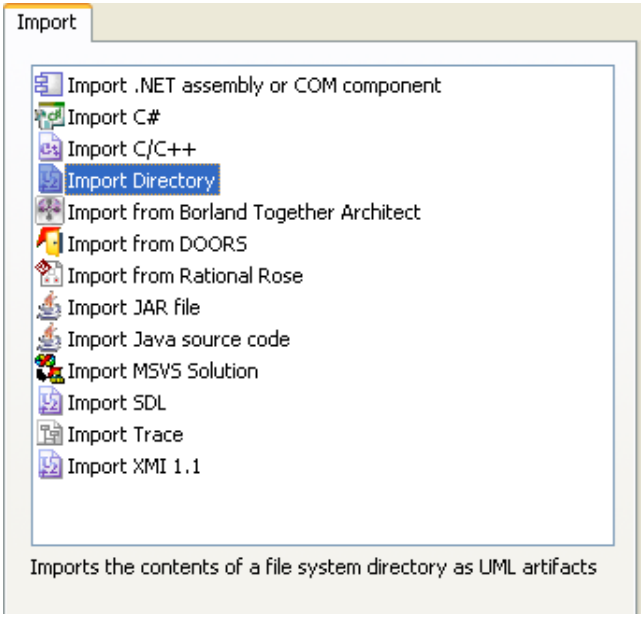


図 278 カスタムインポータを含むインポートウィザード

図のとおり、作成したインポータには、デフォルトのアイコンが割り当てられています。アイコンは <<icon>> ステレオタイプを適用して変更可能です。

この時点でこのインポータを実行すると、何も起こりません。まだ Run エージェントを実装していないからです。それでは、これから Run エージェントを実装しましょう。

```
[[
proc Run { triggeredBy timing context server agentParameters
} {
  upvar 1 $agentParameters ap
  set dir [std::DirectoryDialog]
  set model [u2::GetModel $context]
  set importAgent [u2::FindByName $model "'Import
Directory'::Import"]
  set p [list $dir]
  u2::InvokeAgent $model $importAgent $model p
}
]]
```

インポータの GUI を実装する Run エージェントが、std::DirectoryDialog を使用した標準のダイアログをオープンすることが分かります。このダイアログで、ファイルシステム内のディレクトリを選択できます。その後、選択したディレクトリパスをエージェントのパラメータとして `Import` エージェントを起動します。`'Import Directory'` クラスは <<profile>> パッケージに置かれており、u2::FindByName に修

飾子付きのパッケージ名を渡さなくてもアクセス可能であることに注意してください。プロファイル パッケージ、およびそれに含まれる定義のアクセス性に対する影響の詳細については、プロファイルの適用領域の注記を参照してください。

`Import` エージェントを見つける別の方法として、u2::FindByGuid を使うこともできます。

後は、`Import` エージェントの実装コードを記述するだけです。

```
[[
proc Import { triggeredBy timing context server
agentParameters } {
  upvar 1 $agentParameters ap
  set dir [lindex $ap 0]
  set model [u2::GetModel $context]
  set pkg [u2::Create $model "Package"]
  u2::SetValue $pkg "Name" $dir
  set files [glob -directory $dir "*.*"]
  foreach f $files {
    set a [u2::Create $pkg "Artifact"]
    regsub -all {\\} $f {/} path
    u2::SetValue $a "Name" $path
    set ie [u2::Parse $model "file (. path = \"$path\".)" -
parseAs Expression]
    u2::SetEntity $a "StereotypeInstance" $ie
  }
}
]]
```

コード例では、`Import` エージェントは最上位レベルのパッケージを作成して、その名前を選択したディレクトリの名前にしています。その後、ファイルシステム中のディレクトリの内容を Tcl `glob` コマンドを使用して読み取ります。ファイルごとにファイルアーティファクトが作成されます。u2::Parse が、パスのディレクトリ表記で使用しているバックスラッシュを有効な U2 構文のテキストとして受け取れるように、置き換える必要があることに注意してください。

これで、インポータのテストをできる段階になりました。インポートウィザードから 'Import Directory' インポータを選択して、ファイルシステムを参照できることを確認してください。インポータ実行の結果、モデルは図のようになるはずです。

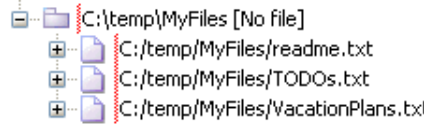


図 279 3 つのテキストファイルをもつディレクトリをインポートした結果

ファイルアーティファクトをダブルクリックすると、該当ファイルを Tau のテキストエディタで開くことができます。

XML ベースのインポータ

よく使われるインポータの 1 つとして、XML ファイルを入力として読み取るインポータがあります。このようなインポータを簡単に作成するために、Tau は XML を解析して UML 表記として取り込む XML フレームワークを提供しています。詳細は、XML 文書のインポート を参照してください。

ダイアグラムを生成するインポータ

インポートした要素を UML ダイアグラムで表示する機能を持つインポータを作成するには、ダイアグラムジェネレータをインポータから起動します。ダイアグラムジェネレータの詳細は、ダイアグラムの生成を参照してください。

TTDDiagramAgents ライブラリには数多くのダイアグラムジェネレータが提供されています。インポータ中からこれらのジェネレータを使用できます。詳細は、ダイアグラムジェネレータのプログラムからの起動 を参照してください。

ダイアグラムジェネレータの追加

カスタムダイアグラムを生成するために、独自のダイアグラムジェネレータを作成できます。ダイアグラムジェネレータは、ダイアグラム作成に使用する Tau API エージェント を使用して実装します。

新しいダイアグラムジェネレータを作成するには、以下の手順を実行します。

1. ダイアグラムを生成したい要素種別を選択します。選択した種別の要素は、コンテキストエンティティとしてダイアグラムジェネレータエージェントに渡されます。
2. ［ダイアグラムの生成］>［新規ダイアグラムジェネレータの作成］を選択します。
3. ダイアログで、新しいダイアグラムジェネレータの名前と説明、格納先となるモデル内での場所を記述します。すべてのダイアグラムジェネレータを、別個の u2 ファイルに格納されるプロファイルパッケージのような共通の場所に置くと便利です。こうすると、作成したダイアグラムジェネレータを複数のプロジェクトで使うことができます。
4. ダイアグラムジェネレータエージェントを作成するのに Tcl API を使用したい場合は、［Tcl スタブ実装を生成する］っチェックボックスをチェックします。
5. ［OK］を押して、ダイアグラムジェネレータを生成します。

ダイアグラムジェネレータは、モデル内で <<diagramGenerator>> ステレオタイプを適用された通常のエージェントとして表現されます。

モデルにダイアグラムジェネレータを作成すると、選択した種別のエンティティに適用可能なダイアグラムジェネレータのリストに、遅滞なく表示されます。作成したダイアグラムジェネレータを他の種別にも適用したい場合は、新しいダイアグラムジェネレータ用に追加の <<agent command>> 依存を作成する必要があります。プロパティエディタで <<agent command>> ステレオタイプインスタンスを編集して、標準ダイアグラムジェネレータパラメータ にデフォルト値を与えることもできます。<<agent command>> ステレオタイプの詳細については、エージェント コマンドを参照してください。

標準ダイアグラムジェネレータパラメータ

カスタムダイアグラムジェネレータパラメータの他に、通常のエージェントパラメータを使用して、ダイアグラムジェネレータフレームワークが標準的なパラメータを渡すように定義できます。このパラメータはダイアグラムジェネレータエージェントで正しく取り扱われる必要があります。ダイアグラムジェネレータが起動時に受け取るパラメータは以下のとおりです。

1) inout diagram : ITtdEntity

2) inout synthesizedEntities : ITtdEntity[*]

3...N) < カスタムダイアグラムジェネレータ固有パラメータ >

‘diagram’ パラメータが NULL の場合、ダイアグラムジェネレータエージェントがダイアグラムを作成する責任を負います。ダイアグラムは任意の場所に置けますが、通常はコンテキスト要素の下に置きます。‘diagram’ パラメータは inout パラメータであり、エージェントが、フレームワークへ制御を戻す前に新規作成ダイアグラムを参照するように設定する必要があります。

‘diagram’ パラメータが NULL でない場合、パラメータは生成済みのダイアグラムを参照します。ダイアグラムジェネレータエージェントがそのダイアグラムの再生成の責任を負います。ダイアグラムはフレームワークによってまず空にされます。このロジックを省略できることで、ダイアグラムジェネレータエージェントの実装は簡単になります。

‘synthesizedEntities’ パラメータは、ダイアグラムとともにダイアグラムジェネレータによって作成されるエンティティのリストです。これによって、ダイアグラムジェネレータは、起動時にはモデルに明示的には存在しない情報を可視化できます。たとえば、ダイアグラムジェネレータはモデルエンティティ間のある関係を算出し、モデル内の依存として表現します。この依存を ‘synthesizedEntities’ リストに追加すると、フレームワークはこのような synthesizedEntities を追跡できます。以前生成したダイアグラムを再生成するためにダイアグラムジェネレータエージェントを起動すると、‘synthesizedEntities’ にそのダイアグラムを生成したときに生成したエンティティが含まれています。したがって、エージェントは新しい情報を算出してダイアグラムを生成する前に、そのエンティティを削除できます。

追加エンティティを合成（synthesize）するダイアグラムジェネレータは、<<diagramGenerator>> ステレオタイプ内のブール値属性 ‘synthesizesAdditionalEntities’ を true に設定する必要があります。

ダイアグラムジェネレータエージェントの実装

ダイアグラムジェネレータエージェントは、ダイアグラムの作成とダイアグラム内へのエンティティの配置のために、Tau API で使用できる任意の機構を使うことができます。ただし、多くのダイアグラムジェネレータに共通する作業があること、およびそういった作業のうち実装がやや困難なものがあることなどを考慮して、Tau は、ユーティリティエージェントによる専用ライブラリを提供してきます。このライブラリは、ダイアグラムジェネレータエージェントの実装で使用できます。

ライブラリの名前は **TTDDiagramAgents** です。このライブラリには、以下の機能を持つユーティリティエージェントが含まれています。

- クラス図にシンボルとラインを追加する
- シーケンス図にシンボルとラインを追加する
- シンボルのデフォルトサイズを設定する
- ラインの経路を自動的に調節して見やすくする
- ダイアグラム内のシンボルレイアウトを異なるアルゴリズムにしたがって変更する

これらのエージェントについての詳細は、TTDDiagramAgents ライブラリ内のダイアグラムにある説明を参照してください。

典型的な実装手順

一般的なダイアグラムジェネレータエージェント実装の処理手順は以下のとおりです

1. モデルに適当な種別のダイアグラムを作成する。このステップは、既存ダイアグラムの再生成の場合はスキップします。
2. ダイアグラム内の情報を集めて可視化します。たとえば、クエリを使用してこれを行います。可視化すべき情報がモデル内に明示的には表現されていない場合は、エージェントは情報を算出し、たとえば依存のようなモデルエンティティによって表現します。
3. TTDDiagramAgents ライブラリのユーティリティエージェントを使って、ダイアグラムにシンボルとラインを配置し、シンボルの適切なサイズを算出し、ラインの経路を調節し、シンボルのレイアウトを調節します。

例

Tau は、Tcl で実装されたカスタムダイアグラムジェネレータの例を提供しています。このサンプルを開くには、［ファイル］>［新規］>［サンプル］から umlAgentCallGraph を選択してください。

ダイアグラムジェネレータのプログラムからの起動

ダイアグラムジェネレータはプログラムから起動できます。ただし、InvokeAgent API メソッドを使った直接的な呼び出しはできません。その代わりに、DiagramGeneratorFramework エージェントを使用します。このエージェントは、ダイアグラムジェネレータのフレームワーク機能を使用するための入り口点となります。

例 **615:** ダイアグラムジェネレータのプログラムからの起動

以下の Tcl スクリプトは InheritanceView ダイアグラムジェネレータを起動して、選択した要素の Inheritance 図を生成します。

```
set curProject [std::GetActiveProject]
set model [std::GetModels -kind U2 -project $curProject]
set a [u2::FindByGuid $model
"@TTDDiagramAgents@DiagramGeneratorFramework"]
set dg [u2::FindByGuid $model
"@TTDDiagramAgents@InheritanceView"]
set p [list $dg "false"]
u2::InvokeAgent $model $a [std::GetSelection] p
```

DiagramGeneratorFramework エージェントへの第一パラメータは、起動するダイアグラムジェネレータです。第二パラメータは、ダイアグラムジェネレータに渡すパラメータです。

DiagramGeneratorFramework エージェントへの第二パラメータは、式のカンマ区切りリストの形でエンコードされたダイアグラム ジェネレータ パラメータを指定します。式は、下表に示す対応する仮ダイアグラム パラメータと一致する必要があります。

式	パラメータ タイプ
真偽の識別子	Boolean
文字列リテラル	Charstring
整数リテラル	Integer
GUID 式、guid(“<guid>“)	ITtdEntity (特定の GUID をもつモデル エンティティ)

ダイアグラムの再生成用のエージェント、DiagramRegenerationFramework があります。このエージェントは、再生成するダイアグラムをモデルコンテキストとしてとらえ、パラメータはないものと考えます。

例 **616:** ダイアグラムのプログラムからの再生成

1757 ページの例 615 で生成したダイアグラムの再生成（生成したダイアグラムを選択したと仮定）

```
set a [u2::FindByGuid $model
"@TTDDiagramAgents@DiagramRegenerationFramework"]
u2::InvokeAgent $model $a [std::GetSelection]
```

ファイル／フォルダ インポータの拡張モジュールの追加

ファイル / フォルダ インポータは、Tau でファイルおよび／またはフォルダのモデル表現を得るために、それらをインポートするツールです。インポートする際、インポートしたファイル／フォルダをさらに処理してモデルに情報を追加できる、拡張モジュールを指定できます。たとえば、拡張モジュールはインポートしたファイルを解析し、ファイル間の論理的な依存関係を示すために、対応するファイル アーティファクト間に依存関係を作成できます。

ファイル／フォルダ インポータと統合された、独自のカスタム拡張モデルを定義できます。このためには、`Import Files/Folders` インポータ クラスにある `ExtensionModule` クラスを継承するクラスを定義します。このクラスは、`TTDImporters` プロファイルにあります。

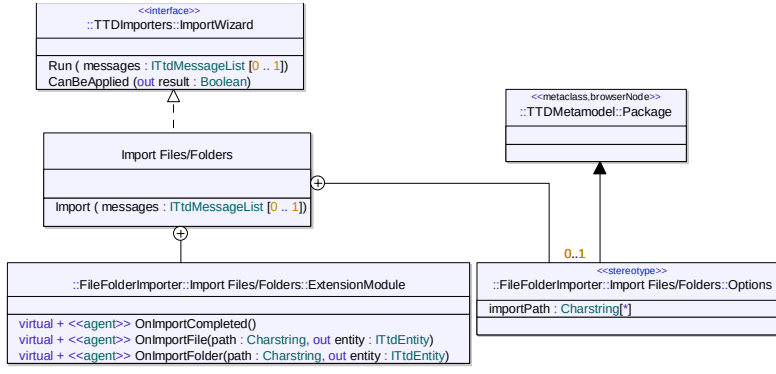


図 280 ファイル／フォルダ インポータに関連するクラスとステレオタイプ

図に示すように、`EtensionModule` クラスはいくつかの仮想エージェント操作を定義します。これらのエージェントは、ファイル／フォルダ インポータのデフォルトの振る舞いを実装します。新しいクラスでエージェントを定義して、これら操作の一部またはすべてをオーバーライドすることにより、この振る舞いをカスタマイズできます。詳細については、再定義可能エージェントを参照してください。

拡張モジュールのオプション

カスタム拡張モジュールがオプションを必要とする場合、`Options` クラスを継承するステレオタイプを定義することで、オプションを定義します。特定の ExtensionModule サブクラスにどの Option サブステレオタイプが属するかを指定するためには、依存関係を使用します。依存関係は、ExtensionModule サブクラスから Option サブステレオタイプに向かいます。

1 つのオプションがすべての拡張モジュールで共通となります。インポートするファイル／フォルダへのパスのリストです。これは、ファイル / フォルダ インポートウィザードの最初のステップで入力したパスと正確に対応します。

再定義可能エージェント

以下のエージェント操作は、拡張モジュールがファイル／フォルダ インポータの振る舞いをカスタマイズするために実装できます。

OnImportFile

ファイル／フォルダ インポータがインポートするファイルを見つけたときに、呼び出されます。

モデル コンテキスト：インポータがファイルの表現を入れるエンティティです。

パラメータ 'path' (in)：ファイルへのパス。

パラメータ 'entity' (out)：可能な場合に、モデル コンテキストに挿入されるファイルの表現。このパラメータは、この特定のファイルのインポートをスキップするために、ヌルに設定できます。

デフォルトの実装では、ファイルのファイル アーティファクトを作成して、それをコンテキストに挿入します。このコンテキストは、デフォルトではパッケージです（インポータが作成する最上位のパッケージまたはフォルダを表すパッケージ）。

拡張モジュールは、モデル内でファイルのカスタム表現を使用するため、このエージェントを実装できます。また、デフォルト ファイル表現の一部を再利用したい場合に、継承した `OnImportFile` エージェント操作を呼び出すこともできます。

OnImportFolder

ファイル／フォルダ インポータがインポートするフォルダを見つけたときに、呼び出されます。

モデル コンテキスト：インポータがフォルダの表現を入れるエンティティです。

パラメータ 'path' (in)：フォルダへのパス。

パラメータ 'entity' (out)：可能な場合に、モデル コンテキストに挿入されるフォルダの表現。このパラメータは、この特定のフォルダのインポートをスキップするために、ヌルに設定できます。

デフォルトの実装では、フォルダのパッケージを作成して、それをコンテキストに挿入します。このコンテキストは、デフォルトではパッケージです（インポータが作成する最上位のパッケージまたはフォルダを表すパッケージ）。

拡張モジュールは、モデル内でフォルダのカスタム表現を使用ため、このエージェントを実装できます。また、デフォルト フォルダ表現の一部を再利用したい場合に、継承した `OnImportFolder` エージェント操作を呼び出すこともできます。

OnImportCompleted

ファイル／フォルダ インポータが指定されたすべてのファイルおよびフォルダのインポートを完了したときに、呼び出されます。

モデル コンテキスト：インポータが作成した最上位のパッケージです。

デフォルトの実装では何も行いません。

拡張モジュールは、インポートしたファイルまたはフォルダの解析を行うために、このエージェントを実装できます。たとえば、対応するファイル間の何らかの関係を表すために、ファイル アーティファクト間に依存関係を追加することがあります。

1 つの例

ファイル／フォルダ インポータのカスタム拡張モジュール追加の例として、ファイル アーティファクトの代わりにクラスを使用するように、テキスト ファイルのデフォルト表現を変更してみましょう。テキスト ファイルの内容は、クラスのコメントとして添付します。この拡張モジュールには、`OnImportFile` エージェントの再定義した振る舞いを実装するために、Tcl API を使用します。

コードを記述する前に、まず、拡張モジュール クラスを定義します。この例では、ファイルのテキストをクラスのコメントとして添付すべきかを制御する、1 つのブール オプション `addComments` を持つ Options ステレオタイプも追加します。`Commentizer` クラスから `CommentizerOptions` ステレオタイプへの依存関係に注意してください。この依存関係は、拡張モジュールをそのオプションと関連付けます。

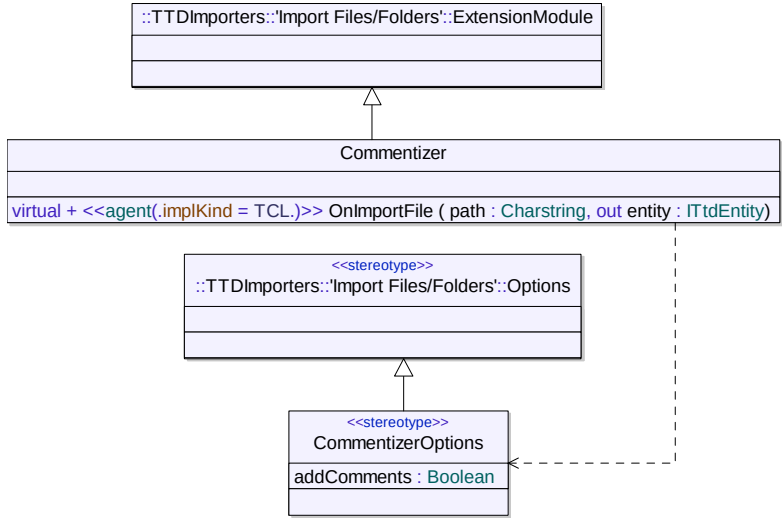


図 281 拡張モジュール クラス例と Options ステレオタイプ

次にファイル／フォルダ インポート ウィザードを開くと、ウィザードの最後のページに Commentizer 拡張モジュールが表示されます。［オプション］ボタンをクリックして `addComments` オプションを設定することもできます。

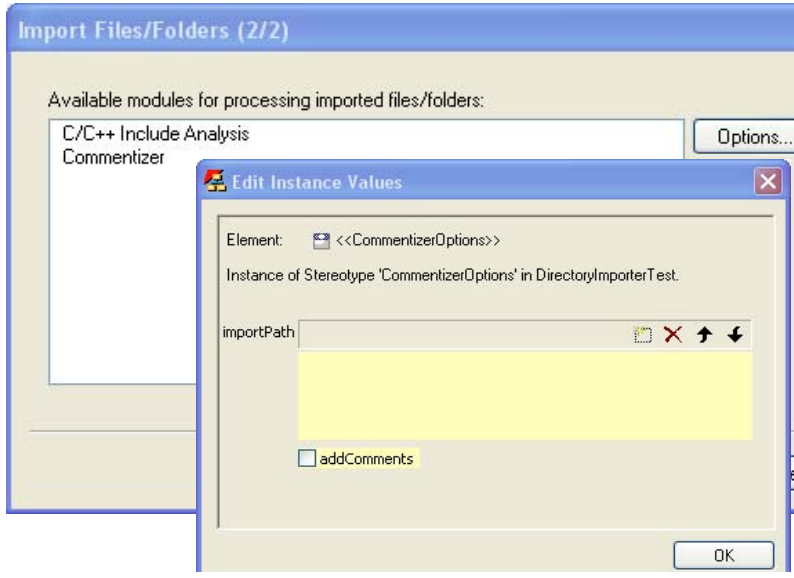


図 282 カスタマイズされたインポート ウィザード の 2 ページ目

これで、`OnImportFile` エージェントを実装する準備が整いました。以下の Tcl スクリプトを使用します。

```
proc OnImportFile { triggeredBy timing context server
agentParameters } {
  upvar 1 $agentParameters ap

  set path [lindex $ap 0]
  set entity [lindex $ap 1]

  if {[file extension $path] != ".txt"} {
    ## Only import text files
    return
  }

  set cls [u2::Create $context "Class"]
  u2::SetValue $cls "Name" $path

  lset ap 1 $cls

  ## Find Commentizer options
  set model [u2::GetModel $context]
  set e $context
  while {[u2::GetOwner $e] != $model} {
    set e [u2::GetOwner $e]
  }
  set ac [u2::GetTaggedValue $e
```

```
"CommentizerOptions(.addComments.)"]
  if {$ac != 0} {
    if {[u2::Unparse $ac] == "true"} {
      ## Read file contents into a comment
      set fp [open $path r]
      set contents [read $fp]
      close $fp
      set comment [u2::Create $cls "Comment"]
      u2::SetValue $comment "Text" $contents
    }
  }
}
```

このスクリプトは、まずエージェント パラメータを抽出してから、'path' がテキスト ファイルを指定しているかを確認します。指定していなければ、このファイルにはこれ以上何も行いません。

テキストファイルを指定している場合は、コンテキスト内にそれを表すクラスを作成します。また、作成したクラスに 'entity' out パラメータを設定します。

残りのスクリプトは、インポート パッケージから 'addComments' タグ値を読み取ります。値が 'true' であれば、ファイルの内容をクラスに添付したコメントに読み込みます。

56

定義済みのステレオタイプと属性

このセクションは、使用可能なステレオタイプと属性のリファレンスです。アルファベット順に説明します。

定義済みのプロファイル ライブラリのモデル情報があります。

完全なリストはオンライン ヘルプ ファイルでのみ参照できます。

57

エージェント

この章ではエージェントの概念と、エージェントを使用して IBM Rational Tau をカスタマイズする方法について説明します。

エージェントは、Tau に接続して、新機能の追加やツールの既存の振る舞いをカスタマイズできる実行形式モジュール（1 つのコード）です。エージェントは UML では操作として定義されますが、現時点では、その実装は、Tau の公開 API（COM、C++、または Tcl）などの実装言語で提供する必要があります。

エージェントの UML 定義では、エージェントを起動するタイミングをエージェントの操作から**ツール イベント**への依存関係を使用して指定できます。1 つのツール イベントは、Tau アプリケーション内で発生する 1 つのイベントを表現します。あるイベントが Tau 内で発生し、そのイベントには対応するツール イベントが定義されている場合には、「ツール イベントが**トリガ**された」と言います。「ツール イベントがトリガ」されると、このツールイベントに依存関係を持つエージェントがすべて起動します。

Tau API を使用して、エージェントを手動で起動することもできます。この方法はアドインからエージェントを起動する場合に便利です。実際に、この API は Tau をカスタマイズするための 2 つの主要技術、アドインとエージェントをつなぐブリッジになっています。アドインは全体を Tcl スクリプトとして実装する必要はなく、エージェントの部分については、別の言語で実装できます。このようなアプローチには、パフォーマンスが向上する、アドインのデバッグにより優れた手段を使用できる、別の言語で記述されたソフトウェアを再利用できる、などの利点があります。

エージェントは、起動されると、呼び出し元からの入力パラメータとして**モデル コンテキスト**を取得します。通常、このモデル コンテキストは、エージェントによる処理の対象であるモデル エンティティです。この場合の「処理」の意味は非常に大まかです。たとえば、カスタマイズされたセマンティック チェックの実行、外部ユーザーに対するレポートの作成、エンティティの修正など、さまざまな処理を意味する可能性

があります。ただし、どのような場合でも、エージェントは与えられたモデル エンティティのコンテキストから処理を実行します。これが、モデル コンテキストと呼ばれる所以です。

モデル コンテキストの他に、エージェントは、呼び出し元から他にいくつかの実パラメータを取得します。これは、ツール イベントのトリガによってエージェントが起動した場合にも、API から明示的に起動された場合にも、当てはまります。取得するパラメータは IN/OUT パラメータとして使用可能なので、エージェントは情報を呼び出し元に渡すことができます。

注記

エージェントは、常に Tau API のクライアントです。一般に API クライアントに適用されるものはすべて、エージェントにも適用されます。たとえば、対話型 エージェントとは、対話型 API クライアントのことです。つまり、IBM Rational Tau IDE と同じメモリ領域で実行されるクライアントです。非対話型 エージェントとは、バッチ コード ジェネレータの実行形式ファイルなど、他の Tau アプリケーションのメモリ領域で実行されるクライアントのことです。

エージェントの定義

エージェント作成の第一歩は、UML でエージェントを定義することです。つまり、モデル内に操作を定義し、適切な名前を付けて、`TTDAgent` プロファイル（すべての UML モデルで使用できる内蔵ライブラリ）である <<agent>> ステレオタイプを適用します。エージェントの操作はどこに作成してもかまいませんが、通常はプロファイル パッケージなどの別パッケージに配置します。

1768 ページの図 283 にエージェント「CheckClassName」の定義を示します。このエージェントは、クラス名の有効性を確認するセマンティック チェック機能を追加するためのものです。

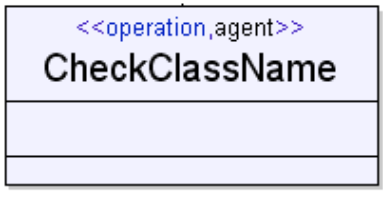


図 283: エージェントの定義

ツール イベントによってトリガされるエージェント起動

エージェント作成の次のステップは、エージェントの起動方法と起動タイミングの設定です。ツール イベントがトリガされた時点でエージェントを起動させるのが、最も一般的です。

エージェントと同様、ツール イベントも UML の操作として表示されます。ツールイベントには、ステレオタイプ <<'tool event'>> が適用されます。TTDAgent プロファイルには、使用可能なすべてのツール イベントが含まれます。特定のツール イベントがトリガされると必ずエージェントが起動するよう指定するには、エージェントからツール イベントへの依存関係を作成する必要があります。依存関係は、TTDAgent プロファイルの以下のいずれかのステレオタイプによってステレオタイプ化されている必要があります。

- <<'before processing'>>
  ツール イベントのデフォルト処理が始まる前にエージェントが起動するように指定します。
- <<'after processing'>>
  ツール イベントのデフォルト処理の完了後にエージェントが起動するように指定します。

例 **617:** カスタム セマンティック チェック エージェント

すべてのクラスに適用されるカスタム セマンティック チェックの 2 つの追加機能について説明します。最初のチェックは、クラスの標準チェックが実行される前に実行され、2 つ目のチェックは、すべての標準チェックが完了した後で実行されます。2 つのエージェントを 1769 ページの図 284 のように定義します。

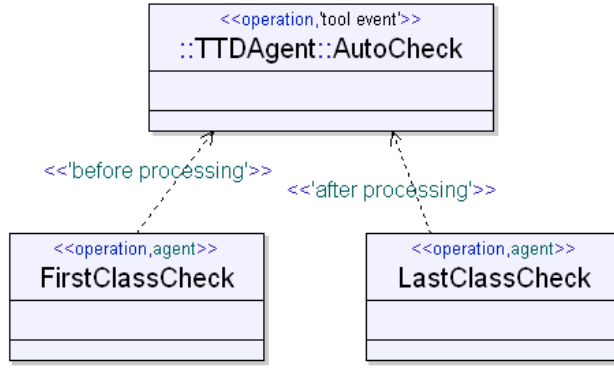


図 284: エージェントの定義

あるエージェントが起動されたときに、別のエージェントが起動するように指定することもできます。これは、依存関係の供給元がツール イベントではなく他のエージェントであるという点を除いて、1769 ページの例 617 と同じ方法になります。この方法で、単一のツール イベントのトリガによって、複数のエージェントを起動させる「起

動ツリー」を作成できます。起動の順序は、常に、<<'before processing'>> ステレオタイプと <<'after processing'>> ステレオタイプによってステレオタイプ化された依存関係を用いて指定します。依存関係については以下について注意が必要です。

1. 2 つのエージェント A と B が、1 つのツール イベント T に対して同様の順序の依存関係を持つ場合、T がトリガされた時点で A と B のどちらが先に起動するかは、定義できません。A と B 間に順序の依存関係がある場合も同様です。
2. 循環する依存関係を指定しないでください。起動がループに陥ります。エージェント A の起動時にループが検知された場合、エージェント フレームワークは A の起動を拒否して、代わりに、次の起動順序にあるエージェントを起動します。

上記の点について、1770 ページの図 285 に図解します。

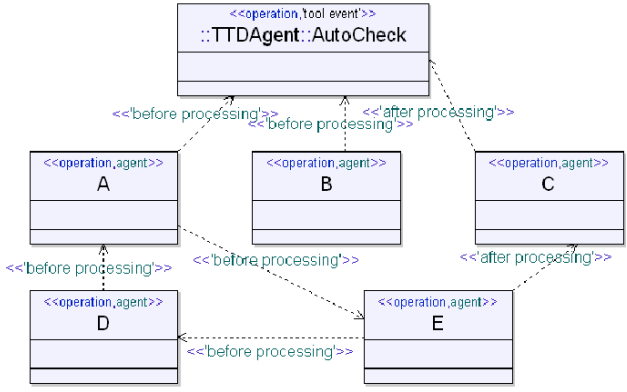


図 285: 順序の依存関係

AutoCheck ツール イベントがトリガされると、エージェントは E D A B C D A E の順序で起動します。あるいは、A と B 間の起動順序が定義されていないため、B E D A C D A E の順序になります。A と E 間の依存関係は、起動順序のループを誘引するため無視されます。

API から エージェントをプログラムとして起動

順序の依存関係を使ってエージェントの起動を静的に指定する代わりに、Tau API（COM、C++、Tcl）のいずれかからエージェントをプログラムとして起動できます。これには、API メソッド InvokeAgent を使用します。任意の数の実 パラメータとともに、指定したモデル コンテキスト上の指定したエージェントを起動できます。

注記

API から明示的にエージェントを起動する場合は、順序の依存関係は意味を持ちません。InvokeAgent 起動によって起動されるエージェントは、1 つのみだからです。

エージェントの実装

ここでは、起動されたエージェントが実行する内容を定義する方法を説明します。エージェントは、現時点で C++（C++ API を使用）、COM/.NET 対応言語（COM API を使用）、Tcl（Tcl API を使用）、または、クエリ式として実装できます。エージェントの実装方法と実装場所は、<<agent>> ステレオタイプ インスタンスのタグ付き値で指定します。

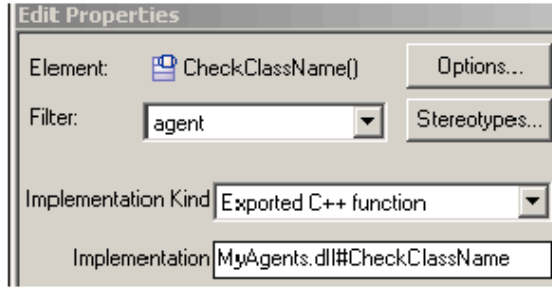


図 286: <<agent>> ステレオタイプ内のタグ付き値

［Implementation Kind］は以下のとおりです。

- COM object または .NET assembly
  COM API を使用してエージェントを実装する場合に選択します。
- Exported C++ function
  C++ API を使用してエージェントを実装する場合に選択します。
- TCL
  Tcl API を使用してエージェントを実装する場合に選択します。
- Query expression
  エージェントをクエリ式として実装する場合に選択します。この指定は、実装するエージェントがクエリである場合にのみ可能です（クエリの詳細についてはクエリを参照してください）。
- Internal
  内部使用向けに予約。

［Implementation］は、エージェントの実装場所を指定する文字列です。このフィールドは、上で選択した実装の種類に従って指定します。

- COM/.NET の場合、起動する COM/.NET オブジェクトのプログラム ID を指定します。

- C++ の場合、以下の形式で指定します。
  `<path>#<function>`
  `<path>` は、実装が格納されている動的リンク ライブラリまたは共有オブジェクトへのパスです。パスは、絶対パスでも相対パスでもかまいません。相対パス指定の場合、パスは PATH（Windows 上）環境変数または LD_LIBRARY_PATH（Unix 上）環境変数に基づいて変換されます。このパスには URN を含むこともできます。ライブラリがロードできない場合、相対パスを使っていると、もう一度ロードを試みます。これには、相対パスを、エージェントが定義されている U2 ファイルに対する場所として解釈して行います。パス名がファイル拡張子を含まない場合は、デフォルトの拡張子が使用されます（Windows 環境では、.dll、Unix 環境では、.so）。Unix については、デフォルトのファイル接頭辞 ”lib” も連結されます。
  `<function>` は呼び出す C++ 関数の名前です。
- Tcl については、実装文字列は次の形式でなければなりません。
  `<path>#<procedure>`
  `<path>` は、実装を格納する Tcl スクリプト ファイルへのパスです。パスは、絶対パスでも相対パスでもかまいません。相対パス指定の場合、エージェントが定義されている U2 ファイルの場所に基づいて変換されます。このパスには URN を含むこともできます。
  `<procedure>` は、指定したファイルに含まれる、呼び出される対象の Tcl プロシージャの名前です。

  また、Implementation の値を未指定のままにしておくこともできます。この場合、Tcl スクリプトは、エージェントのある操作本体内に非形式な実装として格納します。この方法を使うと、エージェントの実装を UML モデル内に格納でき、外部ファイルを参照するより使い勝手がよくなり、効率も若干よくなります。呼び出す Tcl プロシージャはエージェントと同じ名前を持っている必要があります。
- クエリ式の場合、実装文字列はクエリ式を格納するファイルを指定しなければなりません。ファイルは絶対パスまたは相対パスのどちらでも指定できます。相対パス指定の場合、エージェントが定義されている U2 ファイルの場所に基づいて変換されます。このパスには URN を含むこともできます。

  また、実装文字列を未指定のままにしておくこともできます。この場合、クエリ式は、エージェントのある操作本体内に非形式な実装として格納します。この方法を使うと、エージェントの実装を UML モデル内に格納でき、外部ファイルを参照するより使い勝手がよくなり、効率も若干よくなります。

エージェント実装時の一般的なガイドライン

エージェントの実装を記述する際、実装言語（汎用のプログラミング言語でないクエリ式の使用時を除く）に共通するガイドラインがあります。

1. 実装は、エージェントの実行に不要な条件がないかどうかをチェックするテストとともに行います。不要な条件があれば、エージェントは直ちにリターンするようにします。これは、「低レベル」のツール イベントのトリガによって起動する対話型 エージェントには、特に重要です。なぜならば、この種のエージェントは頻繁に起動し、Tau のパフォーマンスを低下させる可能性があるからです。これを避けるためには、エージェントに不要な作業を実行させないことが肝心です。
2. エージェントが対話型でない場合は、ポップアップ ダイアログ ボックスによるエラー レポートは避けるべきです。エージェントで発生したエラー レポートには、エージェントの実行環境に適した方法でエラーメッセージを出力する `ITtdModelAccess::WriteMessage` メソッドを推奨します。
3. エージェントは、非同期タスクの実行のためにスレッドを生成することがありますが、これには注意が必要です。複数スレッドからモデルへの同時アクセスは、安全ではありません。

参照

COM API を使用した実装

C++ API を使用した実装

Tcl API を使用した実装

クエリ式を使用した実装

COM API を使用した実装

COM API には、インターフェイス「`ITtdAgent`」が含まれます。これはすべてのエージェントが実装する必要のあるコールバック インターフェイスです。このインターフェイスには、`1` つのメソッド `Execute` が含まれ、エージェントの起動時に Tau に呼び出されます。

COM オブジェクトのビルドと配置を行い、適切なプログラム ID を付与します。この ID はエージェントの定義に用いる「エージェント」ステレオタイプの「実装」タグ値として使用されます。Tau が COM エージェントを起動しようとすると、特定の ID を持つ COM オブジェクトのインスタンスが作成されます。インスタンスが作成されると、次に、オブジェクトの `ITtdAgent` インターフェイスに `QueryInterface` が作成されます。インターフェイスが作成されると `Execute` メソッドが呼び出されます。

例 **618COM** エージェント

ITtdAgent インターフェイスを実装する COM オブジェクト向けに、Visual Studio .NET ATL COM によって生成されたスケルトン C++ コード は、以下のようになります。

```
class ATL_NO_VTABLE CMyCOMAgent :
  public CComObjectRootEx<CComSingleThreadModel>,
  public CComCoClass<CMyCOMAgent, &CLSID_MyCOMAgent>,
```

```
    public ITtdAgent {
public:
DECLARE_REGISTRY_RESOURCEID(IDR_MYCOMAGENT)
DECLARE_PROTECT_FINAL_CONSTRUCT()
BEGIN_COM_MAP(CMyCOMAgent)
    COM_INTERFACE_ENTRY(ITtdAgent)
END_COM_MAP()
virtual HRESULT __stdcall raw_Execute (
  ITtdEntity* pTriggedBy,
  VARIANT bBeforeProcessing,
  ITtdEntity* pModel,
  IUnknown* pServer,
  SAFEARRAY ** eventProperties ) {
// Implementation...
}
};
```

COM オブジェクトでエージェントを実装する代わりに、COM の相互運用性に対応するように構成されている .NET アセンブリを使用することもできます。

例 **619: C#** エージェント

以下に、接頭辞 "Class_" をそのモデル コンテキスト（クラス）の名前に付加するエージェントの C# 実装の例を示します。

```
using System;
using U2ModelAccessTypeLib;

namespace CSharpAgent
{
  public class C : ITtdAgent
  {
    public void Execute(ITtdEntity triggeredBy,
                        object beforeProcessing,
                        ITtdEntity entity,
                        object server,
                        System.Array eventProperties)
    {
      if (entity.IsKindOf("Class"))
      {
         string s = "Class_";
         entity.SetValue("Name", s + entity.GetValue("Name",
0), 0);
      }
    }
  }
}
```

"Implementation" タグ付き値として使用するプログラマティック ID は、この場合には、CSharpAgent.C（エージェントを実装する C# クラスの修飾名）となります。

C++ API を使用した実装

C++ API にはヘッダー ファイル「U2Agent.h.」が含まれます。このヘッダー ファイルには以下のマクロ定義が含まれます。

```
#define AGENT_PARAMETERS const u2::ITtdEntity* pTriggeredBy,
u2::EventTiming timing, u2::ITtdEntity* pContext,
u2::IUnknown* pServer, u2::AgentParameters& agentParameters,
u2dll::Cu2Changer& changer
#define AGENT_SIGNATURE(name) void name(AGENT_PARAMETERS)

#ifdef _MSC_VER
#define DLLEXPORT __declspec(dllexport)
#else
#define DLLEXPORT
#endif

// Use this macro to define the implementation of an agent as
an exported function.
#define AGENT_IMPL(name) extern "C" DLLEXPORT
AGENT_SIGNATURE(name)
```

AGENT_IMPL マクロは、C++ でのエージェント実装を簡潔に定義するために使用されます。上記に示すよう、C リンクでエクスポートされた関数に拡張されます。

例 **620**

新たに作成された定義の名前にデフォルトの接頭辞を追加する単純なエージェントの C++ 実装は、以下のようになります。

```
AGENT_IMPL(SetNameForMemberAttribute) {
  if (!pContext->IsKindOf(_T("Definition")))
    return; // Not a definition - return immediately!
  tstring strName;
  pContext->GetValue(_T("Name"), strName);
  tstring strNewName(_T("MYPREFIX_"));
  strNewName += strName;
  pContext->SetValue(_T("Name"), strNewName, 0, changer);
}
```

‘Create Entity’ ツール イベントのトリガ時にエージェントが起動する（<<‘after processing’>>）ことを前提としています。

注記

Windows プラットフォームでエージェントを実装する場合、Visual Studio .NET 2008 SP1 のランタイム ライブラリとリンクさせる必要があります。それ以外のバージョンの Visual Studio .NET で提供されるランタイム ライブラリは、Tau で使用されるランタイム ライブラリと互換性がありません。2 つのランタイム ライブラリを混用すると、メモリ処理にエラーが発生し、不安定になります。同様の理由で、Debug 版ではなく、Release 版のランタイム ライブラリを使用することが重要です。詳細については、第 60 章「C++ API」の 1984 ページ、「Visual Studio .NET での C++ エージェントのデバッグ」を参照してください。

Tcl API を使用した実装

Tcl を使用してエージェントを実装する前に、Tau の Tcl インタープリタは IDE プロセス（vcs.exe）のみで利用できることに注意してください。つまり、Tcl API を使用した場合、実装できるのは対話型のエージェントのみです。

Tcl エージェント実装する一番便利な方法は以下の手順です。

- エージェントを定義する UML 操作を作成します。それに名前を付け、パラメータがあればパラメータを定義します。
- ［モデル ビュー］でその操作を選択し、右クリックして表示されたショートカットメニューから ［ユーティリティ］>［Turn into Tcl agent］を選択します。このコマンドは、UML の操作を、スケルトン Tcl スクリプトによって実装されるエージェントに変換します。

エージェントの実装となる Tcl プロシージャには、以下のシグニチャが必要です。

```
proc <name> { triggeredBy timing context server agentParameters }
```

Tcl スクリプトがエージェントのインラインの非形式的な実装として UML モデルに格納される場合、`<name>` はエージェントの名前と同じでなければなりません。

このプロシージャへのパラメータは、ITtdAgent インターフェイスのパラメータと直接一致します。ただし、これらのパラメータの Tcl 表現については以下の点に注意してください

- `server` パラメータは適用されません。これは常に NULL（0）となります。これは Tcl エージェントが対話型 エージェントであるからです。Tau IDE とのやりとりに `server` パラメータを使用する代わりに、Tcl API 全体を直接使用できます。
- `agentParameters` パラメータは、エージェント パラメータを表す文字列の Tcl リストです。これらの文字列は、Tcl コマンド u2::InvokeAgent（モデル コマンド参照）のドキュメントに説明されているルールに従ってエンコードされます。また、これは in/out パラメータであることに注意してください。すなわち、プロシージャ内でこのパラメータにアクセスするには upvar コマンドを使用しなければなりません。

例 **621:** 単純な **Tcl** エージェント

以下の例は、クラスのアクティブ、非アクティブを切り替える単純なエージェントの Tcl 実装です。このエージェントは、クラスがパッシブとなった場合には false（0）のエージェント パラメータを返し、クラスがアクティブとなった場合には true（1）のエージェント パラメータを返します。

```
proc MyAgent { triggeredBy timing context server
agentParameters } {
upvar 1 $agentParameters ap
  if {[u2::IsKindOf $context "Class"]} {
    set isActive [u2::GetValue $context "isActive"]
    if {$isActive == false} {
      u2::SetValue $context "isActive" true
      set ap [list 1]
    } else {
      u2::SetValue $context "isActive" false
      set ap [list 0]
```

```
      }
    }

  }
```

注記

指定した Tcl プロシージャの外に "global" コードが Tcl スクリプトに存在する場合、そのコードは Tcl プロシージャが呼ばれる前に実行されます。

例 **622: Tcl** を使用したクエリ エージェントの実装

この例は、クエリ結果エンティティを結果リストに加える方法を示しています。このエンティティは常に最初のエージェント パラメータとなります。このエージェントはそのモデル コンテキストのモデル（Session）を追加し、その後に、モデル コンテキスト自身を追加します。

```
proc MyQueryAgent { triggeredBy timing context server
agentParameters } {
  upvar 1 $agentParameters ap
  set model [u2::GetEntity $context Session]
  set ap [lreplace $ap 0 0 [list $model $context] ]
}
```

クエリ式を使用した実装

クエリ式は汎用のプログラミング言語ではありません。したがって、クエリ式は、クエリであるエージェントの実装のみに使用します。

クエリ式を使ってエージェントを実装する一番便利な方法は、［クエリ］ダイアログで［保存］ボタンを使用することです。詳細についてはクエリ式を新規クエリとして保存を参照してください。しかし、クエリ式エージェントを手動で作成し、そのクエリ式を外部ファイルに入力するか、またはエージェントの非形式的な実装として入力することももちろん可能です。クエリ式の構文は、［クエリ］ダイアログで使用される構文と同じです。

例 **623:** クエリ式を使用したクエリ エージェントの実装

述語エージェント `MyPredicateAgent`（モデル コンテキストのある状態に基づいて true または false を返すエージェント）が存在すると仮定した場合、次のクエリ式は、述語を満たしているモデル コンテキストに起因するモデル内の全エンティティを検索するクエリ エージェントの実装として使用できます。

```
GetAllEntities().select(MyPredicateAgent())
```

注記

エージェントがクエリ式を使って実装された場合、そのエージェントは仮パラメータを持つことはできません。

エージェント パラメータ

エージェントは UML では操作として定義されます。したがって、他の操作と同じように、エージェントには仮パラメータを持たせることができます。ただし、そういったエージェント操作の呼び出しは、遅延呼び出し（つまり、実行時までパラメータ解決がなされない）となります。したがって、通常の操作のパラメータを使用するよりもエージェント パラメータを使用する方が柔軟になります。柔軟性の代償は、予想されるエージェント パラメータの数と型のチェックを呼び出されたエージェントが行わなければならないことです。

通常、エージェントは任意数の実パラメータを持つことができます。各パラメータは、以下のいずれかの型を持つ必要があります。

データ型	**C++ データ型**	**COM データ型**
テキスト文字列	`tstring`	`BSTR`
オブジェクトの一般的な（未知の）インターフェイス	`u2::IUnknown`	`IUnknown`
未知のインターフェイスのリスト	`std::list<u2::ITtdEntity*>`	`ITtdEntities`
整数	`long`	`long`
ブール値	`bool`	`VARIANT_BOOL`
型付けられていないポインタ	`void*`	`N/A`

また、この表には、C++ API と COM API で特定のエージェント パラメータを示すために使用するデータ型も示しています。これらのデータ型の Tcl マッピングについては、第 59 章「Tcl API」の 1933 ページ、「モデル コマンド」を参照してください。

UML のエージェントの宣言には、そのエージェントが使用するパラメータとパラメータの型の仕様が含まれます。現時点では、エージェントの呼び出し時にエージェント フレームワークはこの情報を使用しません。しかし、エージェントを文書化し、そのエージェントをクライアントから利用しやすくするには、この情報は役に立ちます。

実パラメータを予測するエージェントの実装は、エラー ケースが処理される方法で記述する必要があります。エージェントには特定数の実パラメータを想定したり、パラメータ型の仮定すべきではありません。以下の例に、1 つ目のブール型と 2 つ目の文字列型の 2 つの実パラメータを想定する、典型的な C++ エージェントの実装を示します。

例 **624**

```
AGENT_IMPL(MyAgent) {
  if (agentParameters.size() != 2) {
    // Error: Too few or too many actual parameters!
    return;
  }

  bool par1;
```

```
  tstring par2;
  try {
    par1 = agentParameters.front()->GetBoolean();
    par2 = agentParameters.back()->GetString();
  }
  catch (u2::AgentParameter::ETypeMismatch) {
    // Error: Wrong parameter type!
    return;
  }
}
```

ツール イベント

このセクションでは、トリガ時にユーザー定義エージェントを起動できるツール イベントをリストします。これらのツール イベントの UML 定義は、特に述べない限りは、`TTDAgent` プロファイルにあります。

ツール イベントごとに、以下について説明します。

- Tau でのトリガのタイミング
- ツール イベントに before processing と after processing の両方の順序依存関係を使用する意味があるかどうか（ツール イベントによっては、いずれか一方しか意味がない場合があります）
- ツール イベントとともに渡される実パラメータとその使用方法

ツール イベントは、トリガとなった Tau の機能に従い分類されます。

注記

一部のアドインが定義しているツールイベントは、ここでは説明されていません。そういったツールイベントはアドイン実装のために内部的に使用されています。アドイン機能のカスタマイズのためにそのツールイベントを使用する場合は、それらが定義されているプロファイルに記述されているドキュメントを参照してください。

セマンティック チェッカ イベント

AutoCheck

このイベントは、セマンティック チェッカがエンティティをオートチェックすると、Tau IDE によって送信されます。このイベントの目的は、オートチェック時に、カスタム セマンティック チェックの実行を可能にすることです。

注記

Tau が休止状態にあるとき、最近修正されたエンティティで AutoCheck イベントが送信されます。このため、このイベントのトリガによって起動するエージェントは、自身をモデルからの読み取り専用に制限します。モデルを変更した場合、元に戻すことはできません。

'before processing' でトリガされるエージェントは、エンティティのチェックが実行される前に起動します。'after processing' でトリガされるエージェントは、エンティティのチェックがすべて実行された後で起動します。

タイミング

1. ‘before processing’
2. セマンティック チェッカがエンティティのすべての標準的なチェックを実行した時。
3. ‘after processing’

モデル コンテキスト

チェックするモデル エンティティ

パラメータ

```
[in] messageList: ITtdMessageList
```

チェック メッセージが報告されるメッセージ リスト

Check

このイベントは、セマンティック チェッカがエンティティの正確性をチェックするときに送信されます。これは、Tau IDE で［選択部分をチェック］または［すべてをチェック］を選択した場合、または Tau のコード ジェネレータでコード生成前に発生します。このイベントの目的は、カスタム セマンティック チェックの実行を可能にすることです。

'before processing' でトリガされるエージェントは、エンティティのチェックが実行される前に起動します。'after processing' でトリガされるエージェントは、エンティティのチェックがすべて実行された後で起動します。

タイミング

1. ‘before processing’
2. セマンティック チェッカがエンティティのすべての標準的なチェックを実行した時。
3. ‘after processing’

モデル コンテキスト

チェックするモデル エンティティ

パラメータ

```
[in] messageList: ITtdMessageList
```

チェック メッセージが報告されるメッセージ リスト

Application Builder イベント

AB AutoSave

このイベントは、Application Builder がビルド開始前にモデルを自動保存すると、TauIDE によって送信されます。

'before processing' でトリガされるエージェントは、モデルが保存される前に起動します。したがって、このエージェントが行ったすべての変更は、ビルドを実行するコード ジェネレータによってロードされたモデル内に存在します。これは、モデルに対して、コード生成に影響を与える変更を行った場合、IDE にロードされたモデル内で変更を維持するために使用できます。

'after processing' でトリガされるエージェントは、モデルが保存された後に起動します。したがって、このエージェントが行った変更は、ビルドを実行するコード ジェネレータからは見えません。モデルの変更を行わないタスクを実行する場合、このイベントの 'after processing' が最も便利です。

タイミング

1. ‘before processing’
2. Application Builder がビルド直前のモデルを保存した時。
3. ‘after processing’

モデル コンテキスト

ビルドするビルド アーティファクト

パラメータ

```
[in] entity1: ITtdEntity
[in] entity2: ITtdEntity
...
[in] entityN: ITtdEntity
```

Application Builder がビルドする直前のエンティティごとに入力パラメータが 1 つあります。ビルド アーティファクトをビルドする場合は、ビルド アーティファクトでマニフェストされるエンティティです。他のエンティティを選択してビルドが開始された場合、これらのエンティティが渡されます。

Insert cross reference file

このイベントは、コード ジェネレータ（より一般的には Application Builder クライアント）が「相互参照ファイル」を生成したことを検出すると、Tau IDE から送信されます。このファイルには、コードジェネレータで作成された全ファイルの UML 表現が含まれます。また、もとの UML モデルのエンティティから生成済みファイルの位置へのマッピングも含まれます。C および C++ コードジェネレータは、どちらも相互参照ファイルを共通フォーマットで生成します。

相互参照ファイルの UML 表現は、生成済みファイルを示すファイル アーティファクトのリストを含むパッケージです。各ファイル アーティファクトには、変換後の UML エンティティに対応する生成済みファイル内の位置を示す、<<filePosition>> ステレオタイプ インスタンスのリストがあります。

他の機能と同じように、モデルから生成済みコードへ、またはその逆へ移動を可能にするため、Application Builder は生成済み相互参照ファイルを UML モデルに挿入します。

タイミング

1. Application Builder が、生成された「相互参照」ファイルを検知した時。
2. ‘before processing’
3. Application Builder が、生成された「相互参照」ファイルをモデルにロードした時。
4. ‘after processing’

モデル コンテキスト

ビルドされたビルド アーティファクトビルド アーティファクト。

パラメータ

```
[in] fileName : Charstring
```

相互参照ファイルの名前です。

```
[in] messageList : ITtdMessageList
```

Application Builder のメッセージ リストです。このリストに追加されたメッセージは、［ビルド］タブに出力されます。

```
[in] resultPackage : ITtdEntity
```

（‘after processing’ の場合のみ）
相互参照ファイルをロードした結果のパッケージです。

AB Client File Response

このイベントは、「Insert cross reference file」イベントのより一般的なタイプです。コード ジェネレータ（より一般的には Application Builder クライアント）が一般的な「応答ファイル」（Application Builder がデフォルト アクションを実行しないファイル）を生成したことを検出すると、このイベントが Tau IDE から送信されます。このイベントの目的は、Application Builder クライアントがコード ジェネレータから Tau IDE に情報を伝えることを可能にすることです。

このイベントのトリガ時に実行されるデフォルト タスクがないため、‘before processing’ と ‘after processing’ の相違はありません。

タイミング

1. Application Builder が生成された「応答ファイル」を検出した時。
2. ‘before processing’
3. ‘after processing’

モデル コンテキスト

ビルドされたビルド アーティファクト

パラメータ

```
[in] fileName : Charstring
```

応答ファイルの名前です。

```
[in] messageList : ITtdMessageList
```

Application Builder のメッセージ リストです。このリストに追加されたメッセージは、［ビルド］タブに出力されます。

Process BuildArtifact

このイベントは、ビルド アーティファクトの処理時に Application Builder のコード ジェネレータによって送信されます。したがって、コード ジェネレータが起動するたびに 1 回だけこのイベントが送信されます。‘before processing’ の場合、このイベントの目的は、生成後のモデルの事前変換を可能にすることです。‘after processing’ の場合、このイベントは標準のコード ジェネレータで生成されないコードの生成をトリガするために使用できます。

タイミング

1. ‘before processing’
2. コード ジェネレータがビルド アーティファクトのコードを生成する時。
3. ‘after processing’

モデル コンテキスト

コードを生成するビルド アーティファクト

パラメータ

```
[in] messageList : ITtdMessageList
```

コード ジェネレータが変換メッセージを報告するために使用するメッセージ リストです。コード生成が Tau IDE から開始された場合、このリストに追加されたメッセージは［ビルド］タブに出力されます。コード生成が Tau バッチ 実行形式ファイルから開始された場合、このメッセージは `stdout` に出力されます。

Editor イベント

OpenDiagram

このイベントは、任意の種別のダイアグラムが開いた、または有効になった時に、Tau IDE から送信されます。このイベントの目的は、ダイアグラムが開いているときに起動する必要のある他ツールとのインテグレーションをサポートすることです。

タイミング

1. ‘before processing’
2. ダイアグラムがエディタで可視状態になった時。
3. ‘after processing’

モデル コンテキスト

開くダイアグラム

パラメータ

パラメータなし

InsertDiagramElement

このイベントは、ダイアグラム要素（シンボルまたはライン）がダイアグラムに挿入された時に、Tau IDE から送信されます。ダイアグラム要素の挿入はさまざまな形で起こります。たとえば、ツールバーから新しいシンボルを配置、クリップボードからのシンボルやラインの貼り付け、などです。このイベントの目的は、新規追加したダイアグラム要素をカスタマイズできるようにすることです。たとえば、シンボルのサイズや位置の調節や、追加の要素の自動作成などです。

注記

このイベントは、<<after processing>> 上でのみトリガされます。つまり、ダイアグラム要素がダイアグラムに追加された後のみです。

タイミング

1. ダイアグラム要素がダイアグラムに挿入された時。
2. ‘after processing’

モデル コンテキスト

挿入されたダイアグラム要素

パラメータ

パラメータなし

モデルの相互作用イベント

Change Entity Property

このイベントは、モデル エンティティのプロパティ（メタ特性値）が変更されると、Tau IDE によって送信されます。所有者と合成リンク以外のすべてのメタ特性を変更された場合、このイベントが送信されます。このイベントの目的は、特に、エンティティのプロパティを変更した後に、変換の実行をサポートすることです。

注記

元に戻す操作またはやり直し操作の結果としてプロパティが変更された場合、このイベントは送信されません。このため、このイベントによりトリガされるエージェントの実装が大幅に簡素化されます。このエージェントによって実行された変更はすべて、自動的に、同じ元に戻す／やり直し操作の一部となります。［元に戻す］または［やり直し］による変更時にトリガするには、Entity Modified イベントを参照してください。

タイミング

1. ‘before processing’
2. エンティティの値が変更された時。
3. ‘after processing’

モデル コンテキスト

修正されるエンティティ

パラメータ

```
[in] metaFeatureName : Charstring
```

修正されるメタ特性の名前

```
[in] value : Charstring
```

文字列としてエンコードされた値。‘before processing’ の場合、これから設定される新しい値です。‘after processing’ の場合、変更前の古い値です。

値を文字列としてエンコードする方法について第 58 章「COM API」の 1820 ページ、「GetValue」の表に詳細が掲載されています。

メタ特性がメタクラス型の場合、新しい値はエンティティへのポインタとなります。この場合、値パラメータには、ITtdEntity 型が含まれます。

Entity Modified

このイベントは、［元に戻す］または［やり直し］操作による変更も含め、モデル内のエンティティが変更されたときに Tau IDE によって送出されます。このイベントの目的は、一般的なモデル変更の検出の手段を提供することと、エージェントがモデルのカスタム ビューを変更できるようにすることです。

注記

このイベントは、［元に戻す］または［やり直し］操作によってエンティティが変更されたときにも送出されるので、このイベントでトリガしてモデルを変更するエージェントには推奨されません。トリガされるエージェントは、モデルのビューの更新だけに自身を限定する必要があります。

タイミング

1. エンティティの値が変更された時。
2. ‘before processing’
3. エンティティがファイル（リソース）に保存され、そのファイルに変更されたというマークが付けられた時。

4. エンティティに対してセマンティック チェック（AutoCheck）が予定されている時。
5. ‘after processing’

モデル コンテキスト

修正されるエンティティ

パラメータ

パラメータなし

Create Entity

このイベントは、新しいエンティティがモデル内に作成されると、TauIDE によって送信されます。特に、このイベントの目的は、特に、エンティティの作成時に、追加のエンティティの自動作成をサポートすることです。

タイミング

1. ‘before processing’
2. エンティティが作成され、モデルに挿入された時。
3. ‘after processing’

モデル コンテキスト

‘before processing’ の場合、モデル コンテキストは、新しいエンティティの所有者となるエンティティです。‘after processing’ の場合、モデル コンテキストは新たに作成されたエンティティです。

パラメータ

```
[in] metaClassName : Charstring
```

（‘before processing’ の場合のみ）

これから作成されるエンティティのメタクラスの名前

Move Entity

このイベントは、モデル内でエンティティが移動されると、TauIDE によって送信されます。このイベントの目的は、特に、デフォルトの 'move' セマンティックを再定義することです。

注記

移動されたエンティティに他のエンティティが含まれる場合、これらも一緒に移動されます。ただし、‘Move Entity’ イベントはトップ エンティティにのみ送信されます。

タイミング

1. ‘before processing’
2. `defaultProcessing` フラグが「true」の場合、エンティティは移動されます。それ以外では何も起こりません。

3. ‘after processing’

モデル コンテキスト

移動されるエンティティ

パラメータ

```
[in] target : ITtdEntity
```

エンティティの移動先エンティティ（エンティティ移動後の新しい所有者など）

```
[in/out] defaultProcessing : Boolean
```

(‘before processing’ の場合のみ)
このフラグは、エージェントの呼び出し時に Tau によって「true」に設定されます。エージェントが変更されない場合（呼び出し後も「true」のままの場合)、通常どおり、移動が実行されます。エージェントが移動操作の意味を再定義する場合、戻る前に、このフラグを「true」に設定する必要があります。

Editor イベント

AutoLayout

このイベントは、[自動レイアウト] コマンドがエディタで選択されると、TauIDE によって送信されます。このイベントの目的は、デフォルトの自動レイアウトの振る舞いをカスタマイズするか完全に上書きすることです。

タイミング

1. ‘before processing’
2. `defaultProcessing` フラグが「true」の場合、デフォルトのアルゴリズムに従って自動レイアウトが実行されます。それ以外では何も起こりません。
3. ‘after processing’

モデル コンテキスト

これから処理されるダイアグラム要素を含むダイアグラム

パラメータ

```
[in/out] defaultProcessing : Boolean
```

(‘before processing’ の場合のみ)

このフラグは、エージェントの呼び出し時に Tau によって「true」に設定されます。エージェントが変更されない場合（呼び出し後も「true」のままの場合)、通常どおり、自動レイアウトが実行されます。エージェントが移動操作の意味を再定義する場合、戻る前に、このフラグを「false」に設定する必要があります。

```
[in] entity1 : ITtdEntity
[in] entity2 : ITtdEntity
...
[in] entityN : ITtdEntity
```

これから処理されるダイアグラム要素です。

保存イベント

LoadModel

このイベントは、新しいモデルがロード（プロジェクト ファイルによるロード）されると、Tau IDE によって送信されます。このイベントの目的は、新しいモデルが Tau にロードされる直前あるいは直後に、何らかのタスクの実行を可能にすることです。

タイミング

1. 空のモデルが作成された時。
1. ‘before processing’
2. プロジェクト ファイルがロードされ、その結果がモデルに挿入された時。モデルもバインドされます。
3. ‘after processing’

モデル コンテキスト

モデル（ITtdModel）

パラメータ

```
[in] messageList : ITtdMessageList
```

モデルのロード時に、Tau により使用されるメッセージ リストです。このリストに追加されたメッセージは、［メッセージ］タブに出力されます。

LoadResource

このイベントは、リソース（通常、`.u2` ファイル）がモデルにロードされると、TauIDE によって送信されます。このイベントの目的は、リソースが Tau にロードされる直前あるいは直後に、何らかのタスクの実行を可能にすることです。

タイミング

1. ‘before processing’
2. リソースがロードされ、その結果がモデルに挿入された時。
3. ‘after processing’

モデル コンテキスト

‘before processing’ の場合、モデル コンテキストは、ロードの結果が挿入されるモデル（ITtdModel）です。‘after processing’ の場合、モデル コンテキストは、ロードされたリソース（ITtdResource）です。

パラメータ

```
[in] path : Charstring
```

ロードされるリソースのパス（より一般的には URI）

```
[in] messageList : ITtdMessageList
```

モデルのロード時に、Tau によって使用されるメッセージ リストです。このリストに追加されたメッセージは、［メッセージ］タブに出力されます。

SaveResource

このイベントは、リソース（通常、.u2 ファイル）が保存されると、TauIDE によって送信されます。このイベントの目的は、リソースが保存される直前あるいは直後に、何らかのタスクの実行を可能にすることです。

タイミング

1. ‘before processing’
2. リソースが保存された時。
3. ‘after processing’

モデル コンテキスト

保存されるリソース（ITtdResource）

パラメータ

パラメータなし

C++ アプリケーション ジェネレータイベント

Print C++ Source File

このイベントは、生成後の C++ ソース ファイルの出力時に、C++ アプリケーション ジェネレータによって送信されます。このイベントの目的は、生成後のファイルにカスタム ヘッダーまたはフッターを出力できるようにすることです。

タイミング

1. ‘before processing’
2. ソース ファイルのデフォルトの内容が出力される時。
3. ‘after processing’

モデル コンテキスト

モデルに生成されたソース ファイルを表現するファイル アーティファクト。これはモデルからファイルへのマッピングを行うファイル アーティファクトです（コード ジェネレータで自動作成されています）。

パラメータ

```
[in] sourceBuffer : ITtdSourceBuffer
```

生成ファイルを表します。生成ファイルにテキストを追加するために、ソース バッファ オブジェクトが使用されます。

Print C++ Definition

このイベントは、C++ アプリケーション ジェネレータが C++ 定義を生成後の C++ ソース ファイルに出力するとき、送信されます。このイベントの目的は、定義の直前あるいは直後の出力を可能にすることです。

タイミング

1. ‘before processing’
2. C++ 定義が生成後のファイルに出力される時。
3. ‘after processing’

モデル コンテキスト

出力された C++ 定義。C++ 定義はメタクラス定義の ITtdEntity によって表現されます。

パラメータ

```
[in] sourceBuffer : ITtdSourceBuffer
```

生成ファイルを表します。生成ファイルにテキストを追加するために、ソース バッファ オブジェクトが使用されます。

Entity File Position

このイベントは、C++ アプリケーション ジェネレータが生成後のファイルを出力するときに、送信されます。このイベントの目的は、生成後のエンティティを出力するファイルの場所をエージェントに知らせることです。ファイルの場所は、生成後のファイルの名前（フル パス）、ファイル内の行と列で構成されます。このイベントは以下のエンティティの出力時に送信されます。

- 定義
  イベントは定義の名前の出力時に送信されます。
- 操作本体
  イベントは開き中かっこ（{）の出力時に送信されます。
- 文（アクション）
  イベントは文を終端するセミコロン（;）の出力時に送信されます。

このイベントは［ソースに移動］メニュー項目の実装として、C++ アプリケーション ジェネレータの内部で使用されます。また、ある種のデータベース、生成後のファイルのインデックス、レポートまたは内容を作成しするエージェントにも関連します。

タイミング

1. ‘before processing’
2. ‘after processing’
3. 生成後のファイルを書き込む時。

モデル コンテキスト

出力されるエンティティ

パラメータ

```
[in] path: Charstring
```

モデル コンテキスト エンティティの出力先となる、生成後のファイルのフル パス

```
[in] line : Natural
```

エンティティの出力先となる行番号

```
[in] column : Natural
```

エンティティの出力先となる列番号

Java コードジェネレータイベント

Java コードジェネレータのツールイベントは、`TTDJavaModelCodeSync` ライブラリの `Java tool events` パッケージにあります。

JavaPrintFile

このイベントは、Java コードジェネレータが生成したソースファイルの出力時に、Java コードジェネレータから送信されます。このイベントの目的は、生成後のファイルにカスタム ヘッダーまたはフッターを出力できるようにすることです。

タイミング

1. ‘before processing’
2. ソースファイルのデフォルトの内容が出力される時。
3. ‘after processing’

モデルコンテキスト

モデルに生成されたソース ファイルを表現するファイル アーティファクト。これはモデルからファイルへのマッピングを行うファイル アーティファクトです（コード ジェネレータで自動作成されています）。

パラメータ

```
[in] sourceBuffer : ITtdSourceBuffer
```

生成ファイルを表します。生成ファイルにテキストを追加するために、ソース バッファ オブジェクトが使用されます。

このイベントでトリガされるエージェントの例については、1952 ページの例 711 を参照してください。

JavaPrintDefinition

このイベントは、Java コードジェネレータが Java 定義を生成後の Java ソース ファイルに出力するとき、送信されます。このイベントの目的は、定義の直前あるいは直後の出力を可能にすることです。

タイミング

1. ‘before processing’
2. Java 定義が生成後のファイルに出力される時。
3. ‘after processing’

モデルコンテキスト

出力される Java 定義。Java 定義はメタクラス定義の 1 つの ITtdEntity として表現されます。

パラメータ

```
[in] sourceBuffer : ITtdSourceBuffer
```

生成ファイルを表します。生成ファイルにテキストを追加するために、ソース バッファ オブジェクトが使用されます。

Transformation

このイベントは、UML 構築子を Java 構築子に変換する、Java コード生成内でのステップを表現します。

このイベントの <<before processing>> でトリガされるエージェントは、Java 構築子への標準の変換が用意されていない UML 構築子のための、カスタム変換を実装できます。たとえば、コード生成時に、ドメイン固有の意味を持つ特殊なステレオタイプを適用された 1 つの定義があるとします。エージェントはそのステレオタイプを適用された定義を探して、Java 構築子への標準の変換が用意されている他のモデル構築子で置き換えることができます。

このイベントの <<after processing>> でトリガされるエージェントは、標準の変換の結果を検査でき、その結果を修正できます。つまり、標準の UML から Java への変換規則をカスタマイズできます。

Transformation ツールイベントは生成 Java ソースファイルごとに 1 回トリガされます。

タイミング

1. ‘before processing’
2. Java ファイルアーティファクトにマニフェストされた UML 定義が変換された時。
3. ‘after processing’

モデルコンテキスト

生成された Java ソースファイルを表現している Java ファイルアーティファクトのコピー。

パラメータ

[in] messages : ITtdMessageList

Java コードジェネレータが使用するメッセージのリスト。 AddMessage を使用してこのリストに警告やエラーのメッセージを追加報告できます。

```
[in] buildartifact : ITtdEntity
```

使用する Java ビルドアーティファクト。

```
[in] roots : ITtdEntities
```

Java ファイルアーティファクトにマニフェストされ、Java に変換されるエンティティのコピー。マニフェストされたエンティティの「コピー」なので、エージェントは自由に変更可能です。ただし、エンティティの GUID は元のエンティティと同一です。これは、'originalModel' パラメータと FindByGuid を使って元のエンティティを検索できるようにするためです。通常は、エージェントは ' 元エンティティ ' から情報を読み取り、' コピーエンティティ ' に情報を書き出します。

```
[in] originalModel : ITtdModel
```

元のモデルを表現しています。変更対象である元のエンティティを見つけるための k ンテキストとして使用します。エージェントはこのモデル内を変更してはいけません。変更すると、その変更が元のモデルに表示されます。

Model Verifier イベント

ModelVerifierTextTrace

このイベントは、モデル ベリファイヤ（Model Verifier）がコンソール ウィンドウでテキスト メッセージのトレースを開始するときに、Tau IDE から送信されます。これは、生成済み C アプリケーションからモデル ベリファイヤ（Model Verifier）を介して Tau IDE に情報を伝えるために使用できます。

タイミング

1. 'before processing'
2. `shouldPrint` フラグを true に設定すると、コンソール ウィンドウでテキストがトレースされます。それ以外では何も起こりません。
3. 'after processing'

モデル コンテキスト

モデル ベリファイヤ（Model Verifier）でベリファイするモデルの最上位エンティティ。

パラメータ

```
[in/out] text : Charstring
```

コンソール ウィンドウに表示するテキスト メッセージ。表示するテキストを変更したい場合は（'before processing' の場合のみ可能）、このパラメータを変更する必要があります。

```
[in/out] shouldPrint : Boolean
```

このフラグは、エージェントの呼び出しで、Tau により「true」に設定されます。エージェントが変更しない場合（つまり、呼出し後も「true」のままである場合）、テキスト パラメータの文字列はコンソール ウィンドウに表示されます。エージェントがこのフラグと false に設定すると、テキストは表示されません

エージェント コマンド

エージェント コマンドによって Tau ユーザー インターフェイスから Tau エージェントを呼び出せます。この仕組みによって、使用頻度の高い一般的なエージェントの呼び出しが簡単になります。独自のエージェント コマンドを定義して、Tau ユーザー インターフェイスからユーザー独自のエージェントを呼び出すこともできます。

エージェント コマンドの定義

エージェント コマンドを定義するには、以下の手順を行います。

- エージェントとメタクラスの依存関係を、エージェントが期待するモデル コンテキストの種類を表す TTDMetaModel 内に作成します。たとえば、クラスをモデル コンテキストとして期待するエージェントは、`TTDMetaModel::Class` メタクラスとの依存関係が存在しなければなりません。エージェントが複数の種類のモデル コンテキストを取り扱うことが可能な場合、複数の依存関係を作成できます。
- `<<agent command>>` ステレオタイプを依存関係に適用します。このステレオタイプのタグ付き値を使用することで、エージェント コマンドを Tau ユーザーインターフェイスに表示する方法を制御できます。
  - **Target**：ユーザー インターフェイスのどこにエージェント コマンドを表示するかを指定します。現時点では、ショートカット メニューにエージェント コマンドをバインドすることだけが可能です。
  - **Parameter**：パラメータが存在する場合、パラメータはエージェント パラメータの実際の値を指定します。この値は UML 式のテキスト構文を使用して指定されます。たとえば、文字列とブール値パラメータを期待するエージェントは次の実際の値を持つことができます。`"hello", true`
  - **Name**：ユーザーインターフェイス内のコマンドの名前を指定します。名前が未指定の場合、エージェントの名前がコマンドの名前として使用されます。
- 依存関係に関するコメントを作成できます。このコメントはエージェント コマンドを説明し、ユーザーインターフェイスのステータス バーまたはツールチップのテキストとして表示されます。
- `<<icon>>` ステレオタイプを使用すれば、エージェント コマンドの 16x16 アイコンを指定できます。このアイコンはコンテキスト メニューに表示されます。

エージェント コマンドの使用

エージェント コマンドの定義を終了したら、[モデル ビュー] で正しい種類の要素を右クリックして、そのエージェント コマンドをすぐに使用してみることができます。エージェント コマンドは、以下のルールに規則に従ってショートカット メニューに表示されます。

- エージェントがクエリの場合、エージェント コマンドは [クエリ] サブメニューに表示されます。呼び出されると、クエリの結果は [検索結果] タブに表示され、ナビゲーションはこのタブから実行できます。
- 他の種類のエージェントについては、エージェント コマンドは [ユーティリティ] サブメニューに表示されます。

選択したエンティティに対して少なくとも 1 つのエージェント コマンドを使用できる場合、これらのサブメニューはショートカット メニューのみに表示されます。

ユーティリティ エージェント

Tau には、重要なツール機能をクライアントに公開する、さまざまな種類のユーティリティ エージェントがあります。これらのエージェントはライブラリで定義され、常時使用可能なものと、アドインの起動時にロードされるものがあります。エージェントのドキュメントもモデル内のライブラリに格納されています。

これらのユーティリティ エージェントを使用するには（たとえば、別のエージェントを実装するときなど）、InvokeAgent API メソッドを使用する必要があります。ユーティリティ エージェントの一部はエージェント コマンドとしても定義されます。これにより、Tau ユーザー インターフェイスから直接エージェントを使用できるようになります。

58

COM API

この章は、Tau COM API のリファレンス ガイドです。使用できる COM オブジェクトと COM インターフェイス、およびインターフェイスに定義されたメソッドとプロパティについて説明します。

対象読者は、COM API を使用して UML モデルや Tau IDE の機能にアクセスするクライアント アプリケーションの開発者です。クライアント アプリケーションには、小規模の対話的な処理用のアドインから本格的なコード ジェネレータ、さらにはインポート アプリケーションに至るまでのあらゆるアプリケーションが含まれます。この章全体を通して、COM と C++ の基本知識が前提になります。

COM API は、Windows プラットフォームでのみ使用できます。

概要

COM API は、UML モデルへの完全なアクセス機能を提供します。この API は、多数のメソッドをもったインターフェイスのグループから構成されています。このメソッドには、モデルから情報を抽出するだけのもの（読み取り API）と、モデルの変更が可能なもの（書き込み API）があります。このメソッドを使って、新しいモデルを最初からビルドして、1 つまたは複数のファイルに保存することもできます。

COM API を使って、Tau IDE の特定の機能にアクセスすることもできます。たとえば、COM をサポートする環境から Tau を自動操作する、といった用途で使用できます。

COM API は、独自のメモリ領域で実行されるクライアントで使用できます。この種のクライアントは、「非対話型クライアント」と呼ばれます。このクライアントでは、自分専用のモデルのコピーを使用し、Tau との情報交換は、通常はファイルを通して行われます。一方、COM API は、Tau がロードしたモデルに直接アクセスするような「対話型 クライアント」でも使用できます。たとえば、モデルに対話的にアクセスするアドイン モジュールの開発に使用できます。

この章では、すべての COM インターフェイスとそのメソッドについて説明します。全般にわたって、メソッドごとに 1 つ以上のクライアント使用例を紹介します。使用例では、C++ 言語を使用しており、大半で ATL（Active Template Library）を使用しています。

注記

例は、C# や Visual Basic などの他の COM 有効言語に変換して考えることができます。

これらの例の主目的は、特定の API メソッドの使用法を示すことです。したがって、エラー処理はほぼ省略されています（処理用の例外ハンドラが存在していると仮定しているといってもいいでしょう）。

注記

記述を簡潔にするため、インターフェイスやスマート ポインタなどのすべての参照で、タイプ ライブラリ名前空間は省略されています。修飾子なしでその名前空間の定義へのアクセスを可能にする「using」句がある、と仮定してください。

インターフェイスの概要

COM API には複数のインターフェイスがあり、それぞれが一連の関連性ある機能を提供しています。インターフェイスは、対話型 クライアントと非対話型 クライアントの両方で使用されます。

すべてのインターフェイスは、アーリーバインディングをサポートするクライアント（C/C++、Java など）と、レイト バインディングのみをサポートするクライアント（スクリプト クライアント）の両方から使用できます。

主として以下の 2 種類のインターフェイスが使用されます。

- モデル インターフェイス
- ユーティリティ インターフェイス

モデル インターフェイスは、UML メタモデルの実装内のメタクラスを表すクラスによって実装されます。したがって、このインターフェイスのメソッドを活用するには、メタモデルの知識が必要になります。

ユーティリティ インターフェイスは、その他のすべてのインターフェイスです。このインターフェイスは、Tau のさまざまな機能のうち COM API に公開されているものをを表します。

API のアクセス

COM API にアクセスする方法は、クライアントが対話型か非対話型かによって異なります。

非対話型 クライアントから API へのアクセス

非対話型 クライアントでは、COM API にアクセスするために、API が提供する COM クラスの 1 つのインスタンスを作成します。以下の表に示す 2 つの異なる COM クラスを利用できます。

COM クラス	Prog ID	バイナリの場所	説明
TTD_ModelAccess	Telelogic.TauDeveloperModelAPI	U2EXTU.DLL	この COM クラスをインスタンス化して、UML モデルにアクセスします。この COM クラスは ITtdModelAccess インターフェイスを実装します。
TTD_StudioAccess	Telelogic.TauDeveloperStudioAPI	VCS.EXE	この COM クラスをインスタンス化して、Tau IDE にアクセスします。この COM クラスは ITtdStudioAccess インターフェイスを実装します。

上表の COM クラスの programmatic ID は、バージョン非依存です。複数バージョンの Tau がインストールされている場合に使用する、バージョン依存の ID もあります。

注記

複数バージョンの Tau をインストールした環境でバージョン非依存 ID を使用した場合は、Tau の最新のインストールのものが使用されます。

COM クラスを含んでいるこのバイナリは、COM API タイプライブラリも含んでいます。

非対話型 API クライアントのもう 1 つのカテゴリは、Tau IDE とは別の Tau 実行形式ファイル（VCS.EXE）で実行されるエージェントです。通常、エージェントでは、COM API にアクセスするために、ある COM クラスのインスタンスを作成する必要はありません。その代わりにエージェントは、動作対象となる、ロード済みのモデル内のエンティティのポインタを受け取ります。またエージェントは、エージェント用にサービスを提供するサーバ インターフェイスも取得します。たとえば、C++ アプリ

ケーション ジェネレータで実行されるエージェントでは、一部のコード生成に関連するサービスをエージェントに提供する ITtdCppAppGenServer インターフェイスを取得します。

対話型 クライアントから **API** へのアクセス

対話型 COM API クライアントでは、ITtdAgent インターフェイス（自分がエージェントになる）または ITtdInteractiveClient を実装する必要があります。前者のインターフェイスを使用すると、クライアントを他のすべての API（C++、COM および TCL）から起動できます。また、前者を使用すると、実引数をクライアントに渡せるという利点もあります。エージェントについての詳細は、エージェントの章を参照してください。

ITtdInteractiveClient インターフェイスは、ITtdAgent インターフェイス導入以前に開発されたクライアントをサポートするための後方互換性用として提供されます。ITtdInteractiveClient インターフェイスには、対話型 クライアントの実行時に Tau アプリケーションによって呼び出されるメソッド `OnExecute` が含まれています。`OnExecute` メソッドの最初の引数は、ITtdInteractiveServer インターフェイスへのポインタです。このインターフェイスは、対話型 クライアントのサーバーとして動作する Tau アプリケーションを表します。`OnExecute` メソッドの 2 番目の引数は、ITtdEntities エンティティのコレクションです。そのコレクション内のエンティティ上でクライアントは、COM API を使用できます。

対話型 COM クライアントを使用すると、Tcl スクリプトを使用する場合よりも効率良くアドイン モジュールを実装できます。ただし、現在の仕様では、COM クライアントのみによるアドイン開発はサポートされていません。したがって、最低限の Tcl スクリプトを作成する必要があります。この Tcl スクリプトでは、COM クライアントに実行を移すために、Tcl API コマンド `u2::InvokeAgent`（または、より古く汎用性の少ない std::ExecuteCOMClient）を使用します。

アドインを開発する際に、どの部分を Tcl で開発し、どの部分を COM クライアントで処理するかを決めるのは困難な仕事です。この決定を行う際に、以下のガイドラインが役立ちます。

- たとえば、新しいメニューを追加する場合、Tcl を使用して Tau のユーザー インターフェイスにフックします。現在は、この用途で使用できる COM API はありません。
- アドインで、単純なダイアログではなく高度なユーザー インターフェイスが必要な場合は、COM クライアントでそのインターフェイスを実装します。
- COM クライアントをコンパイル言語（たとえば、C++）で開発して、Tau と同じプロセスで実行すると、対応する Tcl の場合よりもパフォーマンスが大きく向上します。
- COM クライアントでは、実行時に Tau への逆方向の通信が必要になる場合があります。例として、出力タブでの書き込みやダイアグラムの表示などがあります。これを行うには、ITtdInteractiveServerInterpretTclScript メソッドを使用します。

対話型 クライアントにすべての API（Tcl API 以外のものも含めて）からアクセスできるようにするには、エージェントとして実装することを考慮する必要があります。エージェントには、実際の引数を受け入れることができるという利点もあります。COM エージェントによって、ITtdAgent インターフェイスが実装されます。

クライアントの制限事項

COM API のクライアントについて、以下の注意事項があります。

ベアのみ

COM サーバーは、複数のスレッドからの API への同時アクセスをサポートしていません。対話型 クライアントは、Tau アプリケーションのメイン スレッドで実行されます。

Unicode 文字列のみ

COM API のメソッドに渡される文字列、およびメソッドの結果として返される文字列の型は、BSTR 型です。BSTR 文字列は、32 ビット Windows プラットフォーム上のワイド（ダブルバイト）Unicode 文字列です。COM クライアントで Unicode を使用していない場合は、API に渡す前に文字列を Unicode に変換する必要があります。同様に、クライアントでは、API から渡された文字列を、Unicode からクライアントが使用するエンコーディングに変換する必要があります。

ITtdModelAccess

ITtdModelAccess インターフェイスは、TTD_ModelAccess COM クラスのデフォルトのインターフェイスであり、このクラスのインスタンスを作成するときに取得されます。API の他の多くのインターフェイスと異なり、ITtdModelAccess インターフェイスは直接的にはメタモデルのメタクラスに対応せず、すべての非対話型 クライアントの API 全体のエントリ ポイントとして使用されます。対話型 クライアントでは、通常このインターフェイスを使用する必要はありません。

ITtdModelAccess には、以下のメソッドがあります。

LoadProject	プロジェクト（.ttp）をクライアント アプリケーションのメモリにロードします。新しいモデルを最初から作成するときに使用できます。
LoadFile	モデル ファイル（.u2）をクライアント アプリケーションのメモリにロードします。新しいモデルを最初から作成するときに使用できます。
WriteMessage	COM クライアントの環境にメッセージを書き込みます。

LoadProject

Tau プロジェクト（拡張子 ttp）を、クライアント アプリケーションのメモリにロードするか、または新しいモデルを作成します。

```
HRESULT LoadProject(
          [in] BSTR strProjectPath,
          [out, retval] ITtdModel** ppModel);
```

パラメータ

`[in]  strProjectPath`

ロードするプロジェクト ファイルを指定する文字列。該当するファイルが存在しない場合は、新しいモデルが作成されます。strProjectPath が相対パス指定の場合、そのパスはクライアント アプリケーションの現在の作業ディレクトリを基準に解釈されます。

`[out, retval]  ppModel`

ロードされたモデルまたは作成されたモデルのポインタ。

戻り値

返された HRESULT から取得される戻り値は、以下のいずれかです。

戻り値	意味
S_OK	指定されたプロジェクト ファイルのモデルは、正しくロードされました、または作成されました。
E_FAIL	指定されたプロジェクト ファイルのモデルは、ロードまたは作成できませんでした。一般に、このエラーが発生するのは、必要な IBM Rational Tau ライセンスが見つからない場合、または、操作実行のためのメモリが不足している場合です。詳細については、COM エラー オブジェクトが提供する情報を確認してください。

コメント

通常、LoadProject メソッドは、非対話型クライアントによって呼び出される最初のメソッドです。LoadProject を複数回呼び出して、2 つ以上のプロジェクト ファイルのモデルをロードしたり、作成したりできます。ただし、同じプロジェクトを 2 回以上ロードする場合は、十分に注意してください。いずれかのモデルで変更を行うと、他のモデル内の保存していない変更が上書きされる場合があります。

LoadProject メソッドは、モデルがロードまたは作成されると、ITtdEntity::Bind を暗黙的に呼び出して、そのメソッドがリターンするときには、モデルのすべてのリンクと参照を直接ナビゲートできることを保証します。

例 625

以下の例は、プロジェクト ファイル MyModel.ttp に格納されているプロジェクトをロードする方法と、存在しないプロジェクト ファイル NewModel.ttp の新しいモデルを作成する方法を示しています。

```
ITtdModelPtr pITtdModel, pITtdNewModel;
pITtdModel = pITtdModelAccess->LoadProject(_T("MyModel.ttp"));
pITtdNewModel = pITtdModelAccess->LoadProject(_T("NewModel.ttp"));
```

参照

LoadFile

LoadFile

UML モデル データ ファイル（拡張子 u2）を、クライアント アプリケーションのメモリにロードします。ファイルの内容を反映した新しいモデルが作成されます。指定されたファイルが存在しない場合は、新しいモデルが作成されます。

```
HRESULT LoadFile(
          [in] BSTR strFileName,
          [in, optional] profile
          [out, retval] ITtdModel** ppModel);
```

パラメータ

`[in]  strFileName`

ロードするファイルを指定する文字列。このファイルをロードできない場合は（たとえば、存在しない、など）、新しいモデルが作成されます。`strFilePath` が相対パス指定の場合、そのパスはクライアント アプリケーションの現在の作業ディレクトリを基準にして解釈されます。

`[in, optional]  profile`

ファイルに含まれているモデル フラグメントをプロファイルライブラリとしてロードするかどうかを示します。このパラメータはオプションです。省略されている場合は、デフォルトで false になります。

`[out, retval]  ppModel`

ロードされたモデルまたは作成されたモデルのポインタ。

戻り値

返された HRESULT から取得される戻り値は、以下のいずれかです。

戻り値	意味
`S_OK`	ファイルは正しくロードされています。
`E_OUTOFMEMORY`	ロードされたファイルの内容を反映したモデルを作成するにはメモリが不足しています。
`E_FAIL`	何らかの理由によりファイルのロードに失敗しました。詳細については、COM エラー オブジェクトが提供する情報を確認してください。

コメント

`LoadFile` メソッドは、単一のファイルに格納されているモデルをロードするときに使用できる、`LoadProject` の代替メソッドです。`LoadProject` と同じように、新しいモデルの作成にも使用できます。`LoadFile` を呼び出すと、返された ITtdModel インターフェイス ポインタを使ってアクセスできる、新しいモデル表現が作成されます。

`LoadFile` から返されたモデルには、ロードされるファイルを表す Resource（リソース）が含まれます。この Resource（リソース）は、モデル内で行った変更をファイルに保存するために使用できます。

`LoadFile` メソッドは、モデルがロードまたは作成されると、ITtdEntity::Bind を暗黙的に呼び出して、そのメソッドがリターンするときには、モデルのすべてのリンクと参照を直接ナビゲートできることを保証します。

例 **626**

以下の例は、ファイル MyModel.u2 をロードする方法を示しています。

```
ITtdModelPtr pITtdModel;
pITtdModel = pITtdModelAccess->LoadFile("MyModel.u2");
```

参照

LoadProject

Bind

ITtdResource

WriteMessage

COM クライアントの環境で表示されるメッセージを書き込みます。

```
HRESULT WriteMessage(
          [in] BSTR strMessage);
```

パラメータ

`[in]  strMessage`

メッセージ文字列。¥n など、従来のフォーマット用エスケープ シーケンスを挿入できます。

戻り値

返された HRESULT から取得される戻り値は、以下のいずれかです。

戻り値	意味
S_OK	メッセージは正しく環境に対して書き込まれました。
E_FAIL	メッセージを書き込めませんでした。詳細については、COM エラー オブジェクトが提供する情報を確認してください。

コメント

WriteMessage を COM クライアントで使用して、クライアントの実行環境にメッセージを書き込むことができます。対話型 COM クライアントの場合は、メッセージが［メッセージ］タブに出力されますが、非対話型 COM クライアントの場合は、メッセージが stdout 上に出力されます。

例 **627**

```
pITtdModelAccess->WriteMessage(_T("COM client running!¥n"));
```

GetLicense

ランタイムライセンスの取得を要求します。この機能はライセンスが必要な API クライアントによる使用を意図しています。

```
HRESULT GetLicense(
         [in] BSTR strFeature);
```

パラメータ

`[in] strFeature`

取得する必要のあるライセンスフィーチャの名称。

戻り値

HRESULT から取得した戻り値は、以下のいずれかです：

戻り値	意味
S_OK	ライセンスが正しく取得できました。
E_FAIL	ライセンスを取得できませんでした。詳細については COM エラーオブジェクトが提供する情報を参照してください。

コメント

GetLicense は、ライセンスが必要な機能を使う COM クライアントが使用します。たとえば、Telelogic Tau 用の COM API を使用する商用アドインを開発する際などに使用します。

ITtdModel

ITtdModel インターフェイスは、UML モデルの最上位レベル エンティティを表すメタモデルの Session クラスによって実装されます。

ITtdModel インターフェイスには、実行に際して特定のモデル エンティティ コンテキストを必要としないメソッドがあります。こういったメソッドは、通常は、特定のエンティティではなくモデル全体に作用します。

注記

Session（セッション）は、Entity（エンティティ）でもあるので、ITtdModel から ITtdEntity への QueryInterface は常に成功します。

ITtdModel には、以下のメソッドがあります。

FindByGuid	指定された GUID を持つエンティティを探します。
New	新しいモデル エンティティを作成します。作成されたエンティティ上の ITtdEntity インターフェイス ポインタが返されます。
Parse	UML テキスト構文（U2P）を解析して、解析成功時に結果のエンティティ（ポインタ）を返します。
XMLDecode	XML としてエンコードされているモデル フラグメントをデコードして、デコーディングに成功した結果のエンティティ（ポインタ）を返します。
Save	モデルに含まれているすべてのリソースを保存して、モデルを保存します。
CreateResource	モデル内の新しいリソースを作成します。これは、新しいファイルにモデルの一部を保存するための最初の手順です。
LoadFile	UML データ ファイル（.u2 ファイル）を、モデルにロードします。
InvokeAgent	指定されたモデル コンテキスト上で、エージェントを起動します。

FindByGuid

指定された GUID を持つエンティティを探します。

```
HRESULT FindByGuid(
          [in] BSTR strGuid,
          [out, retval] ITtdEntity** ppEntity);
```

パラメータ

`[in]  strGuid`

GUID が含まれている文字列。

`[out, retval]  ppEntity`

指定された GUID を持つエンティティ、そのようなエンティティがモデルに存在しない場合は NULL。

戻り値

返された HRESULT から取得される戻り値は、以下のいずれかです。

戻り値	意味
S_OK	成功。
E_FAIL	内部エラーによる失敗。

コメント

FindByGuid メソッドを使用すると、既知の GUID を持つエンティティ上の ITtdEntity ポインタを取得できます。エンティティの GUID は、エンティティのライフタイム（生存期間）中は常に同じなので、エンティティをモデル内の現在の位置に関係なく探す場合に適しているメソッドです。

モデルに、指定された GUID を持つエンティティが含まれていない場合は、呼び出し後に ppEntity は NULL をポイントします。

例 **628**

以下の行は、定義済みの Boolean データ型にアクセスするための FindByGuid メソッドの使用法を示しています。

```
ITtdModelPtr pITtdModel;
pITtdModel = pITtdModelAccess->LoadProject("MyModel.ttp");
ITtdEntityPtr pEntity;
pEntity = pITtdModel->FindByGuid("@Predefined@Boolean");
```

参照

GUID

New

新しいモデル エンティティを作成します。作成されたエンティティ上の ITtdEntity インターフェイス ポインタが返されます。

```
HRESULT New(
          [in] BSTR strMetaClass,
          [out, retval] ITtdEntity** ppEntity);
```

パラメータ

```
[in]   strMetaClass
```

メタモデル内のメタクラスの名前。文字列には、大文字と小文字を区別して既存のメタクラスを指定する必要があります。メタクラスを正しく指定しないと、メソッドは失敗します。

`[out, retval]  ppEntity`

作成されたエンティティ。

戻り値

返された HRESULT から取得される戻り値は、以下のいずれかです。

戻り値	意味
S_OK	成功。
E_OUTOFMEMORY	新しいエンティティを割り当てるために使用できるメモリが不足しています。
E_FAIL	作成に失敗しました。詳細については、COM エラー オブジェクトが提供する情報を確認してください。

コメント

New メソッドでは、指定されたメタクラスのインスタンスを作成します。このメタクラスが存在し、非抽象クラスである必要があります。

注記

返されたエンティティを処理する役割はクライアントが担います。エンティティはモデルに挿入するか、削除して、メモリ リークを避ける必要があります。

例 **629**

以下の例は、New を使用して Package（パッケージ）を最上位レベルのモデル エンティティとして作成し、その後に ITtdEntity::SetEntity を使用して、作成したエンティティを挿入する方法を示しています。

```
ITtdModelPtr pITtdModel;
ITtdEntityPtr pSession = pITtdModel;
pITtdModel = pITtdModelAccess->LoadProject("MyModel.ttp");
ITtdEntityPtr pPackage;
pPackage = pITtdModel->New("Package");
pPackage->SetEntity("Namespace", pSession);
```

参照

Create

Parse

UML テキスト構文（U2P）を解析して、解析成功時に結果のエンティティ（ポインタ）を返します。

```
HRESULT Parse(
          [in] BSTR strConcreteSyntax,
          [in, optional] VARIANT parseAs,
          [out, retval] ITtdEntities** ppEntities);
```

パラメータ

`[in]  strConcreteSyntax`

U2P 構文が含まれている文字列。

`[in, optional]  parseAs`

指定された文字列の解析をどのように試行するかをパーサに指示します。Definition、Expression、Action のいずれかのメタクラスを指定する必要があります。デフォルトでは、Definition として解析が試行されます。

`[out, retval]  ppEntities`

成功した解析の結果として得られるエンティティ。

戻り値

返された HRESULT から取得される戻り値は、以下のいずれかです。

戻り値	意味
S_OK	成功。
E_FAIL	解析に失敗しました。詳細については、COM エラー オブジェクトが提供する情報を確認してください。

コメント

`Parse` メソッドは、テキスト構文の記述に基づいてモデル フラグメントを作成する場合に便利です。多くの場合は、エンティティを 1 つずつ作成して（`ITtdModel::New` または `ITtdEntity::Create` を使用）、それらを接続する（`ITtdEntity::SetEntity` を使用）よりも、こちらの方法のほうが便利です。パーサがモデルの作成を処理するので、メタモデルの知識もさほど要求されません。

注記

返されたエンティティを処理する役割はクライアントが担います。エンティティはモデルに挿入するか、削除して、メモリ リークを避ける必要があります。

例 **630**

以下の例は、Parse メソッドを使用して、1809 ページの例 629 で作成した Package（パッケージ）の Attribute（属性）で Class（クラス）を作成する方法を示しています。パーサでは、デフォルトの振る舞いである Definition として指定された文字列を解析します。

```
ITtdEntitiesPtr pParseResult;
pParseResult = pITtdModel->Parse("class C { public Integer a; };");
ITtdEntityPtr pClass;
pClass = pParseResult->GetItem(1);
pPackage->SetEntity("OwnedMember", pClass);
```

参照

Unparse

XMLDecode

XML としてエンコードされているモデル フラグメントをデコードして、デコーディングに成功した結果のエンティティ（ポインタ）を返します。

```
HRESULT XMLDecode(
          [in] BSTR strXMLEncoding,
          [out, retval] ITtdEntities** ppEntities);
```

パラメータ

`[in]  strXMLEncoding`

XML でエンコードされたモデル フラグメントが含まれている文字列。

`[out, retval]  ppEntities`

成功した XML デコーディングの結果として得られるエンティティ。

戻り値

返された HRESULT から取得される戻り値は、以下のいずれかです。

戻り値	意味
S_OK	成功。
E_FAIL	デコーディングに失敗しました。詳細については、COM エラー オブジェクトが提供する情報を確認してください。

コメント

XMLDecode メソッドを使用して、XML エンコーディングからモデル フラグメントを作成できます。通常、XML エンコーディングは ITtdEntity::XMLEncode を呼び出した結果として取得されますが、.u2 ファイルの内容の読み取りなど、他の手段でも取得できます。

注記

返されたエンティティを処理する役割はクライアントが担います。エンティティはモデルに挿入するか、削除して、メモリ リークを避ける必要があります。

例 **631**

以下の例は、ITtdEntity::XMLEncode メソッドを使用して、1809 ページの例 629 で作成した Package（パッケージ）を XML としてエンコードし、その後に XMLDecode を使用して、その XML を Package（パッケージ）にデコードする方法を示しています。

```
CComBSTR bstrXMLEncoding((TCHAR*) pPackage->XMLEncode());
ITtdEntitiesPtr pRoots;
pRoots = pITtdModel->XMLDecode((BSTR) bstrXMLEncoding);
```

Save

モデルに含まれているすべてのリソースを保存して、モデルを保存します。

```
HRESULT Save();
```

戻り値

返された HRESULT から取得される戻り値は、以下のいずれかです。

戻り値	意味
S_OK	モデルは正しく保存されました。
E_FAIL	モデル内の少なくとも 1 つのリソースを保存できませんでした。詳細については、COM エラー オブジェクトが提供する情報を確認してください。

コメント

Save メソッドを使用して、モデル内で行ったすべての変更を保存できます。このメソッドを使用すると、モデル内のすべてのリソースを繰り返し使用してそれぞれに対して逐一 ITtdResource::Save を呼び出す手間が省けます。

Save メソッドの使用法の例は、1813 ページの例 632 にあります。

CreateResource

モデル内の新しいリソースを作成します。これは、新しいファイルにモデルの一部を保存するための最初の手順です。

```
HRESULT CreateResource(
          [in] BSTR strFileName,
          [out, retval] ITtdResource** ppResource);
```

パラメータ

`[in]  strFileName`

リソースのファイルの名前。

`[out, retval]  ppResource`

作成されたリソース。

戻り値

返された HRESULT から取得される戻り値は、以下のいずれかです。

戻り値	意味
S_OK	リソースは正しく作成されました。
E_OUTOFMEMORY	新しいリソースを割り当てるために使用できるメモリが不足しています。
E_FAIL	内部エラーによる失敗。

コメント

`CreateResource` メソッドは、モデル内で新しいリソースを作成する方法として推奨されています。もう 1 つの方法は、より一般的な `ITtdEntity::Create` で、Resource メタクラスを引数として使用します。後者のアプローチの短所は、リソースのファイル名の入力を要求するダイアログが表示されることです。これは、必ずしも望ましい振る舞いとはいえません。なぜなら、クライアントがすでに他の方法によってファイル名を取得している場合があるからです。

例 **632**

以下の例では、`CreateResource` を使用して、1809 ページの例 629 で作成した Package（パッケージ）をその専用ファイルに保存します。

```
ITtdEntityPtr pResource;
pResource = pITtdModel->CreateResource("D:¥temp¥COMtest.u2");

// Insert the created package pPackage as a root of pResource
pResource->SetEntity("Root", pPackage);
```

```
pITtdModel->Save(); // Saves all resources in the model
```

LoadFile

UML データ ファイル（.u2 ファイル）を、モデルにロードします。

```
HRESULT LoadFile(
          [in] BSTR strFileName,
          [in, optional] VARIANT profile,
          [out, retval] ITtdResource** ppResource);
```

パラメータ

`[in]  strFileName`

ロードするファイルを指定する文字列。このファイルをロードできない場合は（存在しない、など）、エラー コード E_FAIL が返されます。`strFilePath` が相対パス指定の場合、そのパスはクライアント アプリケーションの現在の作業ディレクトリを基準にして解釈されます。

`[in, optional]  profile`

ファイルに含まれているモデル フラグメントをプロファイルライブラリとして扱うかどうかを示します。このパラメータはオプションです。省略されている場合は、デフォルトで false になります。

`[out, retval]  ppResource`

ロードされたファイルのリソースのポインタ。上記の `profile` が true の場合は、このポインタが NULL に設定されます。

戻り値

返された HRESULT から取得される戻り値は、以下のいずれかです。

戻り値	意味
S_OK	ファイルは正しくロードされています。
E_OUTOFMEMORY	ロードされたファイルの内容を表すリソースを作成するにはメモリが不足しています。
E_FAIL	何らかの理由によりファイルのロードに失敗しました。詳細については、COM エラー オブジェクトが提供する情報を確認してください。

コメント

`LoadFile` メソッドを使用して、ファイルに格納されているモデル フラグメントを、モデルに追加でロードできます。通常このメソッドが使用されるのは、別々のファイルに格納されたプロファイルを、このモデルで使用するためにロードする必要がある場合です。ただし、通常のモデル セグメントでもロードは可能です。ロードするモデル フラグメントをプロファイルライブラリとして扱うかどうかは、引数 `profile` を使用して制御します。

`LoadFile` は、ロードされたファイルを表す Resource（リソース）を返します。この Resource（リソース）は、モデル内で行った変更をファイルに保存するために使用できます。

`LoadFile` メソッドは、ファイルがロードされると、`ITtdEntity::Bind` を暗黙的に呼び出して、そのメソッドがリターンするときには、モデルのすべてのリンクと参照を直接ナビゲートできることを保証します。

例 **633**

以下の例は、それぞれプロジェクト ファイル `MyProj.ttp` とファイル `MyProfile.u2` に格納されているプロジェクトとプロファイルの定義をロードする方法を示しています。

この場合のプロファイルは通常の u2 ファイルとしてロードされることに注意してください。つまり、モデルの Library セクションには読み込まれません。

```
ITtdModelPtr pITtdModel;
ITtdResourcePtr pITtdResource;
pITtdModel = pITtdModelAccess->LoadProject(_T("MyProj.ttp"));
pITtdResource = pITtdModel->LoadFile(_T("MyProfile.u2"));
```

参照

LoadProject

ITtdResource

InvokeAgent

指定されたモデル コンテキスト上で、エージェントを起動します。

```
HRESULT InvokeAgent(
          [in] ITtdEntity* agent,
          [in] ITtdEntity* modelContext,
          [in, out, optional] VARIANT* agentParameters);
```

パラメータ

`[in]  agent`

起動するエージェントの定義。

`[in] modelContext`

エージェント起動のモデル コンテキストとなるエンティティ。エージェントは、このエンティティのコンテキストで動作します。

`[in, out, optional] agentParameters`

オプションの実エージェント パラメータ リスト（安全な配列）。

戻り値

返された HRESULT から取得される戻り値は、以下のいずれかです。

戻り値	意味
S_OK	エージェントは正しく起動されています。
E_FAIL	何らかの理由によりエージェントを起動できませんでした。詳細については、COM エラー オブジェクトが提供する情報を確認してください。

コメント

`InvokeAgent` メソッドを使うと、エージェントをプログラムから起動できます。このメソッドは、`ExecuteCOMClient` Tcl コマンドとの多くの共通点がありますが、起動されたエージェントを COM で実装する必要がないため、より一般的です。

起動されたエージェントが実引数が期待している場合は、これらを、変数の安全な配列として指定する必要があります。変数のデータ型は以下のとおりです。

- COM インターフェイス ポインタ（IUnknown）。ITtdEntity または ITtdEntities を、渡されたインターフェイスから取得できる必要があります。
- 文字列（BSTR）
- 整数（long）
- ブール値（VARIANT_BOOL）

エージェント パラメータ リストは、in/out パラメータです。したがって、エージェントでは情報を呼び出し側に戻す目的で、エージェント パラメータを変更できます。したがって、呼び出し側ではエージェント起動後の実パラメータの値に依存するコードを作成してはいけません。エージェントは、実引数の数とデータ型の両方を変更している可能性があります。

例 **634**

以下の例は、既知の GUID を定義として保持しているエージェントの実行方法を示しています。モデル コンテキストは、モデルの任意のエンティティです。この場合実引数はエージェントに渡されません。

```
ITtdEntityPtr pAgent;
pAgent = pITtdModel->FindByGuid(_T("MyAgentGuid"));
pITtdModel->InvokeAgent(pAgent, pModelContext);
```

参照

第 57 章「エージェント」

ITtdAgent::Execute

ITtdEntity

ITtdEntity インターフェイスは、UML モデルの一般エンティティを表すメタモデルの Entity クラスによって実装されます。

ITtdEntity インターフェイスには、実行に際して特定のモデル エンティティ コンテキストを必要とするメソッドがあります。こういったメソッドは、通常は、呼び出し元のエンティティに作用します。

メソッド	説明
ApplyStereotype	与えられたステレオタイプをインスタンス化して、エンティティに適用します。
Bind	モデルフラグメント中のすべての参照、または 1 つのエンティティの単一参照をバインドします。
Clone	エンティティのクローンを作成します。
Create	エンティティのコンテキストで新しいエンティティを作成します。つまり、新しい直接的または間接的な子をエンティティに追加します。
CreateInstance	シグニチャのインスタンスを作成します。
Delete	エンティティをモデルから削除します。
GetEntities	エンティティのメタ特性の値をエンティティのコレクションとして返します。
GetEntity	エンティティのメタ特性の値をエンティティとして返します。
GetMetaClassName	エンティティのメタクラスの名前を返します。
GetModel	エンティティの属するモデルを返します。
GetOwner	エンティティの所有者を返します。
GetReference	参照を表すメタ特性の識別子を返します。
GetReferringEntities	特定のメタ特性を通じてエンティティを参照するエンティティのコレクションを返します。
GetTaggedValue	要素の指定されたプロパティ（タグ付き値）を返します。インスタンス式から任意の値を取得するときにも使用できます。

ITtdEntitiy メソッド

メソッド	説明
GetValue	エンティティのメタ特性の値を文字列として返します。
HasAppliedStereotype	要素にステレオタイプが適用されているかどうかを判定します。
IsKindOf	エンティティが特定のメタクラスの種類かどうかを判定します。
MetaVisit	モデル フラグメントをトラバースして、コールバック インターフェイスで、含まれているエンティティごとにメソッドを呼び出します。
MetaVisitEx	トラバースで参照を含むことができる MetaVisit の拡張バージョン。
Move	エンティティをモデル内の現在の位置から別のオーナーに移動します。
SetEntity	エンティティのメタ特性の値をエンティティとして設定します。
SetTaggedValue	要素上のプロパティ（タグ付き値）を設定します。インスタンス式の任意の値を設定するときにも使用できます。
SetValue	エンティティのメタ特性の値を文字列として設定します。
Unparse	エンティティの具体的な構文表現への逆構文解析。
XMLEncode	エンティティを XML 表現にエンコードします。

ITtdEntitiy メソッド

ApplyStereotype

与えられたステレオタイプをインスタンス化してエンティティに適用します。ステレオタイプへの参照で使用される修飾子は、以下の例のように `referencekind` に基づいて算出されます。`pInsertElement` が与えられている場合は、ステレオタイプはホストエンティティ上でインスタンス化されることになりますが、物理的には `InsertElement` 上でインスタンス化されます。この場合、ステレオタイプインスタンスはホストエンティティを指します。このように、ステレオタイプインスタンスはあるエンティティに適用されますが、エンティティそのものの変更はしません。

```
HRESULT ApplyStereotype(
          [in] ITtdEntity* stereotype,
          [in] TtdReferenceKind referenceKind,
          [in, optional] VARIANT insertElement,
          [out, retval] ITtdEntity** result);
```

パラメータ

`[in]` `stereotype`

エンティティをインスタンス化するステレオタイプ。

`[in]` `referenceKind`

ステレオタイプへの参照の修飾子は `referenceKind` に基づき算出されます。

TtdReferenceKind	説明
`TTD_RK_GUID`	参照は GUID 参照のみを含みます。
`TTD_RK_NO_QUALIFIER`	参照は限定（修飾）されません。つまり、ステレオタイプ名のみを含みます。
`TTD_RK_FULL_QUALIFIER`	参照は完全修飾子を含みます。
`TTD_RK_MINIMAL_QUALIFIER`	参照はステレオタイプを参照するのに必要な最低限の修飾子を含みます。このオプションは最大でも `TTD_RK_RELATIVE_QUALIFIER` と同じ修飾子を返します。注記：`<<access>>/<<import>>` 依存がある場合は、修飾子が短くなる可能性があります。
`TTD_RK_RELATIVE_QUALIFIER`	参照はステレオタイプに対する相対的な修飾子になります。ステレオタイプとホストエンティティに共通の上方スコープがない場合は、完全修飾子が算出されます。そうでない場合は、直近の共通スコープから始まる、短い修飾子が算出されます。

Tcl ユーザー向けに、短縮版の `referenceKind` 値が用意されています。短縮値は Tcl API においてのみ使用できます。以下の表に値をリストします。

COM	Tcl
`TTD_RK_GUID`	`GUID`
`TTD_RK_NO_QUALIFIER`	`noQualifier`
`TTD_RK_FULL_QUALIFIER`	`fullQualifier`
`TTD_RK_MINIMUM_QUALIFIER`	`minimumQualifier`
`TTD_RK_RELATIVE_QUALIFIER`	`relativeQualifier`

注記

通常は `minimalQualifiers` を使用することを推奨します。

`[in, optional]  insertElement`

`insertElement` が与えられた場合、ステレオタイプは論理的にはホストエンティティ上でインスタンス化され、物理的には `insertElement` 上でインスタンス化されます。この場合、ステレオタイプインスタンスはホストエンティティを指します。このようにステレオタイプインスタンスはエンティティに適用されますが、エンティティそのものは変更しません。このテクニックは「Stereotype Injection」として知られています。

`[out, retval]  result`

インスタンス化されたステレオタイプ。

戻り値

返された HRESULT から取得される戻り値は、以下のいずれかです。

戻り値	意味
S_OK	成功。
E_FAIL	内部エラーによる失敗。

GetValue

メソッドの呼び出し元であるエンティティのメタ特性の値を返します。値は文字列として表されます。

```
HRESULT GetValue(
          [in] BSTR strMetaFeature,
          [in, optional] VARIANT nPos,
          [out, retval] BSTR* strValue);
```

パラメータ

`[in]  strMetaFeature`

値を抽出したいメタ特性を指定する文字列。文字列には、メソッドの呼び出し元のエンティティの既存のメタ特性を、大文字と小文字を区別して指定する必要があります。メタ特性を正しく指定しないと、メソッドは失敗します。

`[in, optional]  nPos`

値を抽出したいメタ特性のインデックス。インデックスは 1 から開始されます。インデックス 0 を使用して、メタ特性の最後のインデックスを指定できます。このパラメータを省略すると、デフォルトで 0 になります。インデックスを指定する場合は、有効な範囲内にする必要があります。範囲を超えていると、メソッドは失敗します。

`[out, retval]  strValue`

文字列で表現される、取得された値。この文字列のフォーマットの詳細については、下の「コメント」セクションの表を参照してください。

戻り値

返された HRESULT から取得される戻り値は、以下のいずれかです。

戻り値	意味
`S_OK`	成功。
`E_FAIL`	何らかの理由により、指定された値を取得できませんでした。詳細については、COM エラー オブジェクトが提供する情報を確認してください。

コメント

`GetValue` メソッドは、文字列値を保持できるエンティティのすべてのメタ特性上で使用できます。ただし、メタ特性のうち、メタクラス型から派生した機能、所有者リンク、合成リンクは除きます。

`GetValue` によって返される文字列のフォーマットは、指定されたメタ特性のデータ型によって異なります。以下の表は、メタ特性と返される文字列のフォーマットを示しています。

メタ特性のデータ型	文字列	例
列挙（リテラルを持つデータ型）	リテラルの名前	`"true"` `"VkVirtual"`
Charstring または識別子（`CeIdent`）	文字列または識別子	`"MyClass"`
数値型（整数、実数、自然数など）	文字列としての数値	`"14"` `"217.5"`
GUID（`CeGuid`）	GUID	`"63Lkm3hRE7K*uRE"`
位置またはサイズ型（`SPoint` または `SSize`）	スペースで区切られた X 座標と Y 座標	`"1240 1380"`
位置のリスト（`PointVector`）	スペースで区切られたそれぞれの位置のエンコーディング	`"1940 1020 1940 1120"`

メタ特性のデータ型	文字列	例
メタクラス	割り当てられた値がメタ特性にない場合： 空の文字列	""
	メタ特性が GUID で設定されているか、ポインタで直接設定されている場合： 文字列 "uid:" とその後に続くターゲット エンティティの GUID	"uid:63Lkm3hRE7K*uRE"
	メタ特性が名前によって設定されている場合： 文字列 "ref:" とその後に続くターゲット エンティティの名前（修飾可能）	"ref:MyClass::MyAttr"

例 635

以下の例では、定義済みのブール データ型の名前にアクセスするために GetValue メソッドが使用されています。

```
ITtdModelPtr pITtdModel;
pITtdModel = pITtdModelAccess->LoadProject("MyModel.ttp");
ITtdEntityPtr pEntity;
pEntity = pITtdModel->FindByGuid("@Predefined@Boolean");
CComBSTR bstrName((TCHAR*) pEntity->GetValue("Name"));
```

参照

GetEntity

GetTaggedValue

メタモデル クラス

GetEntity

メソッドの呼び出し元であるエンティティのメタ特性の値を返します。値は ITtdEntity ポインタとして表されます。

```
HRESULT GetEntity(
          [in] BSTR strMetaFeature,
          [in, optional] VARIANT nPos,
          [out, retval] ITtdEntity** ppEntity);
```

パラメータ

```
[in]   strMetaFeature
```

値を抽出したいメタ特性を指定する文字列。文字列には、メソッドの呼び出し元のエンティティの既存のメタ特性を、大文字と小文字を区別して指定する必要があります。メタ特性を正しく指定しないと、メソッドは失敗します。

`[in, optional]  nPos`

値を抽出したいメタ特性のインデックス。インデックスは 1 から開始されます。インデックス 0 を使用して、メタ特性の最後のインデックスを指定できます。このパラメータを省略すると、デフォルトで 0 になります。インデックスを指定する場合は、有効な範囲内にする必要があります。範囲を超えていると、メソッドは失敗します。

`[out, retval]  ppEntity`

ITtdEntity ポインタとして表される、取得された値。値が存在しない場合は、このポインタが NULL になります。

戻り値

返された HRESULT から取得される戻り値は、以下のいずれかです。

戻り値	意味
`S_OK`	成功。
`E_FAIL`	何らかの理由により、指定された値を取得できませんでした。詳細については、COM エラー オブジェクトが提供する情報を確認してください。

コメント

`GetEntity` メソッドは、メタクラス型のエンティティのすべてのメタ特性上で使用できます。これらのメタ特性では、多くの場合、`GetValue` よりこのメソッドのほうが便利です。これは、取得された ITtdEntity ポインタを、後続のメソッド呼び出しに直接使用できるためです。ただし、`GetEntity` から取得されたエンティティ ポインタが NULL の場合は、対応する `GetValue` 呼び出しを行って、これがその後の時点でバインドできることを表しているか、または値がメタ特性に指定されていないかどうかを効果的に調べることができます。後者の場合、`GetValue` は空の文字列を返しますが、前者の場合は、メタ特性が現在バインドされていなくても、返される文字列はエンコードされたメタ特性値になります。

注記

メタ特性がメタモデル内の参照を表す場合は、`GetValue` より `GetReference` のほうが効果的な代替メソッドになります。

例 **636**

以下の例では、`pAttribute` によってポイントされた属性の型にアクセスするために、`GetEntity` メソッドを使用しています。その type メタ特性が Definition（定義）にバインドされている場合は、その Definition（定義）の名前が抽出されます。

```
ITtdEntityPtr pType;
pType = pAttribute->GetEntity("Type");
if (pType != 0 && pType->IsKindOf("Class"))
  CComBSTR bstrName((TCHAR*) pType->GetValue("Name"));
```

参照

GetValue

GetEntities

GetEntities

メソッドの呼び出し元のエンティティのメタ特性の値を返します。戻り値はエンティティのコレクション、つまり TtdEntities ポインタです。

```
HRESULT GetEntities(
          [in] BSTR strMetaFeature,
          [out, retval] ITtdEntities** ppEntities);
```

パラメータ

`[in]  strMetaFeature`

値を抽出したいメタ特性を指定する文字列。文字列には、メソッドの呼び出し元のエンティティの既存のメタ特性を、大文字と小文字を区別して指定する必要があります。メタ特性を正しく指定しないと、メソッドは失敗します。

`[out, retval]  ppEntities`

メタ特性の値になっているエンティティが含まれた ITtdEntities コレクションのポインタ。

戻り値

返された HRESULT から取得される戻り値は、以下のいずれかです。

戻り値	意味
S_OK	成功。
E_FAIL	何らかの理由により、指定された値を取得できませんでした。詳細については、COM エラー オブジェクトが提供する情報を確認してください。

コメント

GetEntities メソッドは、メタクラス型のエンティティのすべてのメタ特性上で使用できます。1 つではない多重度を持つメタ特性のすべてのエンティティを取得する場合、異なるインデックスを使用して GetEntity の複数回の連続呼び出しを行うよりも、このメソッドを使う方が便利です。ただし、単一多重度を持つメタ特性にもこのメソッドは使用できます。その場合、取得されたコレクションに 1 つのエンティティが含まれるか、またはエンティティが含まれません。

メタ特性が対応するインデックスでバインドされていない場合は、返されるコレクションに NULL ポインタが含まれることがあります。そのようなバインドされていないメタ特性の詳細情報を検索するには、GetValue または GetReference を使用します。

例 **637**

以下の例では、pPackage によってポイントされたパッケージ内のクラス（および他のシグニチャ）にアクセスするために、GetEntities メソッドを使用しています。

```
ITtdEntitiesPtr pSignatures;
pSignatures = pPackage->GetEntities("Signature");
long lCount = pSignatures->Count;
ITtdEntityPtr pSignature;
// N.B. Index are 1-based
for (int i = 1; i <= lCount; i++){
  pSignature = pSignatures->GetItem(i);
  // Do something with pSignature...
}
```

参照

GetEntity

GetReference

メソッドの呼び出し元のエンティティのメタ特性の参照を返します。メタ特性は、メタモデル内の（非派生型）参照を表している必要があります。

```
HRESULT GetReference(
          [in] BSTR strMetaFeature,
          [in, optional] VARIANT nPos,
          [out, retval] ITtdEntity** ppEntity);
```

パラメータ

[in] strMetaFeature

参照を抽出する必要があるメタ特性を指定する文字列。文字列には、メソッドの呼び出し元のエンティティの既存のメタ特性を、大文字と小文字を区別して指定する必要があります。また、メタ特性は、メタモデル内の（非派生型）参照を表している必要があります。メタ特性を正しく指定しないと、メソッドは失敗します。

`[in, optional]  nPos`

参照を抽出する必要があるメタ特性のインデックス。インデックスは 1 から開始されます。インデックス 0 を使用して、メタ特性の最後のインデックスを指定できます。このパラメータを省略すると、デフォルトで 0 になります。インデックスを指定する場合は、有効な範囲内にする必要があります。範囲を超えていると、メソッドは失敗します。

`[out, retval]  ppEntity`

取得された参照の ITtdEntity ポインタ。参照は、識別子（`Ident` メタクラスのインスタンス）によって表されます。参照が存在しない場合は、このポインタが NULL になります。このようになるのは、参照が識別子ではなく GUID またはポインタによって設定されている場合、または参照がまったく設定されていない場合です。

戻り値

返された HRESULT から取得される戻り値は、以下のいずれかです。

戻り値	意味
`S_OK`	成功。
`E_FAIL`	何らかの理由により参照を取得できませんでした。詳細については、COM エラー オブジェクトが提供する情報を確認してください。

コメント

`GetReference` メソッドは、メタモデル内の非派生型参照を表すすべてのメタ特性上で使用できます。これは、所有者リンクと複合 リンクを除いたメタクラス 型のすべてのメタ特性に当てはまります。

返される参照は、モデル内の識別子として表されます。この識別子には名前があります。また、そのバインドに必要な追加情報（たとえば、範囲修飾子）が含まれることもあります。

一般に、参照はかなり複雑になることがあるため、参照を表すメタ特性では、多くの場合 `GetReference` のほうが `GetValue` よりも効果的な代替メソッドになります。その理由は、結果が、文字列表現ではなく識別子のモデル表現となるからです。

例 **638**

以下の例では、`pAttribute` によってポイントされた属性の type メタ特性の参照にアクセスするために、`GetReference` メソッドを使用しています。返された参照は、その後に `Unparse` メソッドを使用して、逆構文解析されます。

```
ITtdEntityPtr pRef;
pRef = pAttribute->GetReference("Type");
if (pRef != NULL){
    CComBSTR bstrText((TCHAR*) pRef->Unparse());
```

```
}
```

参照

GetValue

GetOwner

メソッドの呼び出し元のエンティティの所有者を返します。

```
HRESULT GetOwner(
          [out, retval] ITtdEntity** ppEntity);
```

パラメータ

`[out, retval]` `ppEntity`

メソッドの呼び出し元エンティティの直接の所有者であるエンティティのポインタ。エンティティに所有者が存在しない場合は、NULL。

戻り値

返された HRESULT から取得される戻り値は、以下のいずれかです。

戻り値	意味
S_OK	成功。

コメント

`GetOwner` は、エンティティの所有者を探すためのメソッドとして推奨されています。所有者を探す代替テクニックとして、所有者メタ特性用の `GetEntity` 呼び出しがあります。ただし、エンティティに 2 つ以上の所有者メタ特性がある場合は、所有者メタ特性ごとに `GetEntity` 呼び出しを 1 回行うよりも、`GetOwner` を呼び出すほうが簡単です。

例 **639**

以下の例は、`GetOwner` を使用して、`pEntity` によってポイントされたエンティティの Session（セッション）を探す方法を示しています。Session（セッション）は、モデルの最も外側のエンティティであることに注意してください。

```
ITtdEntityPtr pOwner = pEntity;
ITtdEntityPtr pSession;
while (pOwner != 0){
    pSession = pOwner;
    pOwner = pOwner->GetOwner();
}
// pSession now points at the Session of pEntity (provided pEntity
```

```
// is part of the model of course)
```

なお、Session（セッション）を探すためのより簡単な方法として、すべてのエンティティ上で使用可能な導出メタ特性「Session」で `GetEntity` を使用できます。GetModel も使用できます。

参照

GetEntity

GetMetaClassName

メソッドの呼び出し元のエンティティのメタクラスの名前を返します。

```
HRESULT GetMetaClassName(
          [out, retval] BSTR* strName);
```

パラメータ

`[out, retval]  strName`

エンティティのメタクラスの名前。

戻り値

返された HRESULT から取得される戻り値は、以下のいずれかです。

戻り値	意味
S_OK	成功。

コメント

`GetMetaClassName` は、ITtdEntity ポインタが参照しているエンティティの種類を探すときに便利なメソッドです。多くの場合は、`IsKindOf` を使用して、エンティティが特定のメタクラスの種類かどうかを十分探すことができますが、エンティティの正確なメタクラスを探すことが必要になる場合があります。

例 **640**

以下の例は、`GetMetaClassName` と `IsKindOf` を使用して、ITtdEntity ポインタが参照しているエンティティの種類を区別する場合の違いを示しています。

```
CComBSTR bstrMetaClass((TCHAR*) pEntity->GetMetaClassName());
if (bstrMetaClass == "Operation"){
  // Will only get here if pEntity's metaclass is Operation,
  // that is  not if pEntity is a StateMachine
}
if (pEntity->IsKindOf("Operation")){
```

```
// Will get here if pEntity is an Operation (including
// StateMachine since the StateMachine metaclass inherits
// the Operation metaclass)
```

参照

IsKindOf

GetReferringEntities

GetReferingEntities と同じ関数です。GetReferingEntities という関数は削除されています。現在は API で下位互換を確保するために使用されます。

特定のメタ特性を通じて、メソッドの呼び出し元のエンティティを参照しているエンティティのコレクションを返します。

```
HRESULT GetReferringEntities(
          [in] BSTR strMetaFeature,
          [out, retval] ITtdEntities** ppEntities);
```

パラメータ

[in] strMetaFeature

エンティティを参照するためのメタ特性を指定する文字列。これは、メソッドの呼び出し元のエンティティ（参照先エンティティ）ではなく、参照エンティティ上のメタ特性です。文字列には、大文字と小文字を区別して既存のメタ特性を指定する必要があります。メタ特性を正しく指定しないと、結果として空のコレクションが返されます。

[out, retval] ppEntities

指定されたメタ特性を通じてエンティティを参照するエンティティが含まれている ITtdEntities コレクションのポインタ。

戻り値

返された HRESULT から取得される戻り値は、以下のいずれかです。

戻り値	意味
S_OK	成功。

コメント

GetReferringEntities メソッドの目的は、逆参照と呼ばれるものを探すことです。モデル内のある特定のメタ特性に対応する参照がバインドされている場合は、参照先エンティティから参照エンティティへの逆参照も確立されます。GetReferringEntities は、これらの逆参照のターゲット エンティティ、つまり参照エンティティへのアクセス機能を提供します。

1830 ページの図 287 は、逆参照の概念を示しています。Class（クラス）によって型指定された Attribute（属性）を示します。Attribute（属性）からそのデータ型にナビゲートするには、「type」メタ特性の GetEntity メソッドを使用します。ただし、type から Attribute（属性）にナビゲートするには、代わりに GetReferringEntities を使用する必要があります。

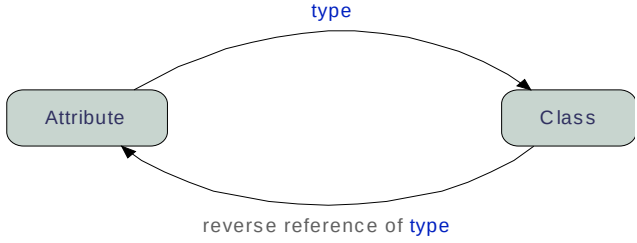


図 287: 逆参照

例 641 ——————————————————————————————————

以下の例では、定義済みのブール型によって型指定されたすべてのエンティティを探すために、GetReferringEntities メソッドが使用されています。

```
ITtdModelPtr pITtdModel;
pITtdModel = pITtdModelAccess->LoadProject("MyModel.ttp");
ITtdEntityPtr pEntity;
pEntity = pITtdModel->FindByGuid("@Predefined@Boolean");

ITtdEntitiesPtr pReferringEntities;
pReferringEntities = pEntity->GetReferringEntities("Type");
// pReferringEntities now contains all entities typed by Boolean
```

——

GetTaggedValue

要素のプロパティ（タグ付き値）を返すか、または一般にインスタンス式からの値を返します。

```
HRESULT GetTaggedValue(
          [in] BSTR strSelector,
          [in, optional] VARIANT interpretIdentsAsGuids,
          [out, retval] ITtdEntity** ppValue);
```

パラメータ

`[in]  strSelector`

取り込む値（タグ付き）を指定するセレクタ パターンが含まれている文字列。このパターンのフォーマットについて以下で説明します。

`[in, optional]  interpretIdentsAsGuids`

セレクタ パターン内の識別子を通常の名前ではなく GUID として解釈するかどうかを示します。このパラメータはオプションです。省略されている場合は、デフォルトで false になります。

`[out, retval]  ppValue`

セレクタ パターンによって選択された値（タグ付き）。または、（指定された値が存在しないか、そのセレクタ パターンが十分に構成されていないという理由により）そのような値が存在しない場合は、NULL。値は、Expression（式）です。

戻り値

返された HRESULT から取得される戻り値は、以下のいずれかです。

戻り値	意味
`S_OK`	成功。
`E_FAIL`	内部エラーによる失敗。

コメント

`GetTaggedValue` は、要素からプロパティ（タグ付き値）を抽出するために便利なメソッドです。プロパティはモデル内で通常の値として表されるため、このメソッドは、InstanceExpr メタクラスによってモデル内で表されている任意のインスタンス上で使用することもできます。

メソッドの呼び出し元のエンティティが `InstanceExpr` でもステレオタイプ適用可能な要素でもない場合は、NULL が返されます。

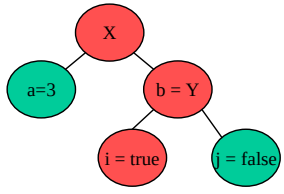
もちろん、単に `GetEntity` メソッドを使用して（および値がモデル内でどのように表されるかを正確に把握して）、プロパティを抽出することもできます。ただし、`GetTaggedValue` を使用すると、以下のようにアプローチが効率的になります。

- パターンでは、`InstanceExpr` によって表される「値のツリー」で任意の深度の値を選択できます。
- `GetTaggedValue` を 1 回呼び出すと、`GetEntity` の複数の呼び出しを行う必要がなくなります。したがって、クライアント コードはよりコンパクトで読みやすくなります。

- パターンは、シグニチャ継承をサポートします。つまり、共通の親シグニチャを指定するパターンを使用して、継承された複数のシグニチャのインスタンスを一致させることができます。
- 選択された値が `InstanceExpr` 内に存在せず、対応する Attribute（属性）にデフォルト値がある場合は、その値が返されます。
- このメソッドは、対応するメタクラスの必須機能拡張がある場合、自動的に適用されるステレオタイプの値も処理します。
- 値のモデル表現での一部の特殊ケースは、`GetTaggedValue` の実装によって処理され、クライアントではこれらを考慮する必要がなくなります。

パターンによって、間違ったステレオタイプ インスタンス内のプロパティ（タグ付き値）が選択されるリスクを防止するには、パラメータ `bInterpretIdentsAsGuids` を使用する必要があります。これは、通常の場合、処理対象のプロファイル パッケージ内のステレオタイプのプロパティを抽出するクライアントに役立ちます。クライアントで、プロパティのアクセス元にするステレオタイプの GUID が既知となっている場合は、プレーンな名前ではなく GUID を持つセレクタ パターンを使用できます。1833 ページの例 642 は、その例を示しています。

セレクタ パターンの構文は、インスタンス式のテキスト UML 構文（U2P）ですが、識別子、インスタンス式、割り当てのみを使用できるように制限されます。また、インスタンス式ごとの割り当ては 1 つだけにする必要があります。結果は、「値のツリー」をたどるパスを指定するパターンになります。パスの終わりの値が選択されます。



X(. b = Y(. i .) .)

図 288: セレクタ パターンの例

1832 ページの図 288 は、あるステレオタイプ X のインスタンス内の Y::i のプロパティ（タグ付き値）がパターンによってどのように選択されるかを示しています。

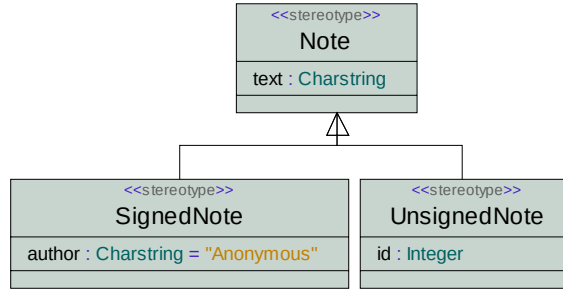


図 289: ノートを記述するステレオタイプ

例 642: GetTaggedValue の使用法

1833 ページの図 289 に示すモデルが使用されています。

`pElement` によってポイントされた要素があることを想定します。以下の行では、その要素の SignedNote:: 作者プロパティを抽出します。

```
ITtdEntityPtr pTaggedValue;
pTaggedValue = pElement->GetTaggedValue("SignedNote (. author .)");
```

SignedNote インスタンス内で作者属性の明示的な値が指定されていませんが、SignedNote:: 作者にはデフォルトの値（Anonymous）があるため、上記の `pTaggedValue` は、CharstringValue をポイントします。

以下のコードでは、パターン `“Note (. text .)”` を使用して、SignedNotes と UnsignedNotes 上でタグ付き値 `“text”` にします。

```
ITtdEntityPtr pTaggedValue;
pTaggedValue = pElement->GetTaggedValue("Note (. text .)");
```

以下の行には、要素上に UnsignedNote ステレオタイプが適用されているかどうかを判定するための単純なパターンが含まれています。

```
ITtdEntityPtr pTaggedValue;
pTaggedValue = pElement->GetTaggedValue("UnsignedNote (. .)");
```

UnsignedNote が要素上に適用されている場合は、`pTaggedValue` によって、ステレオタイプ インスタンスを表す InstanceExpr がポイントされます。適用されていない場合は、NULL になります。ただし、HasAppliedStereotype メソッドに関する以下の注記をお読みください。特定のステレオタイプを適用している場合、このメソッドが最適です。

UnsignedNote ステレオタイプの GUID が `@UnsignedNote` であるとわかっている場合は、`bInterpretIdentsAsGuid` パラメータを使用して、同じ名前の別のステレオタイプのインスタンスを取得することを回避できます。GUID は、アポストロフィで囲んで、構文的に正しいパターンにする必要があります。UnsignedNote が要素上に適用されている場合は、返されたステレオタイプ インスタンス上で `GetTaggedValue` の

別の呼び出しを行うことができます。その呼び出しでは、パターンに GUID が含まれている必要はありません。これは、適切な InstanceExpr を操作しているとわかっているためです。

```
ITtdEntityPtr pTagValue;
pTagValue = pElement->GetTaggedValue("'@UnsignedNote' (. .)",
      true /*idents as GUIDs*/);
if (pTaggedValue != NULL){
  ITtdEntityPtr pIdValue;
  pIdValue = pTagValue->GetTaggedValue("UnsignedNote (. id .)");
```

注記

エンティティにステレオタイプ X が適用されていルカをテストするために、“X (. .)”形式のセレクタ パターンを使用することもできますが、テストには HasAppliedStereotype を使用したほうがより効果的です。これは、自動的に適用されたステレオタイプ（一致するメタクラスの必須拡張により）が、少なくとも属性の 1 つがのデフォルトと異なるタグ付き値を取得するまで、インスタンス化されないためです。このため、GetTaggedValue には、このような特定のケースで返されるステレオタイプ インスタンスはありません。ただし、GetTaggedValue が、タグ付き値を選択するセレクタ パターンと同時に使用されると、自動的に適用されるステレオタイプについても考慮されるようになります。

参照

GetEntity

SetTaggedValue

HasAppliedStereotype

HasAppliedStereotype

要素にステレオタイプが適用されているかどうかを判定します。

```
HRESULT HasAppliedStereotype(
         [in] BSTR strStereotype,
         [in, optional] VARIANT guid,
         [out, retval] VARIANT_BOOL* result);
```

パラメータ

`[in]  strStereotype`

エンティティに適用されたステレオタイプの中から検索対象のステレオタイプを特定する文字列。guid が false の場合、この文字列はステレオタイプの名前です。true の場合、この文字列はステレオタイプの GUID です。

`[in, optional]  guid`

strStereotype を検索対象のステレオタイプ名としてではなく、GUID として解釈すべきかどうかを示します。このパラメータはオプションです。省略するとデフォルトで false になります。

```
[out, retval]   result
```

指定したステレオタイプが要素に適用されている場合は True 、それ以外の場合は False です。

戻り値

返された HRESULT から取得される戻り値は、以下のいずれかです。

戻り値	意味
S_OK	成功。
E_FAIL	内部エラーによる失敗。

コメント

HasStereotypeApplied は、エンティティに適用されたステレオタイプのチェック用に推奨される関数です。この関数は、明示的に適用されたステレオタイプと、エンティティのメタクラスと一致するメタクラスの必須拡張により自動的に適用されたステレオタイプの両方を考慮します。

参照

1830 ページの「GetTaggedValue」

IsKindOf

メソッドの呼び出し元のエンティティが特定のメタクラスの種類かどうかを判定します。

```
HRESULT IsKindOf(
          [in] BSTR strMetaClass,
          [out, retval] VARIANT_BOOL* isKindOf);
```

パラメータ

```
[in]   strMetaClass
```

メタモデル内のメタクラスの名前。

```
[out, retval]   isKindOf
```

メソッドの呼び出し元のエンティティが指定されたメタクラスの種類の場合は true、それ以外の場合は false。

戻り値

返された HRESULT から取得される戻り値は、以下のいずれかです。

戻り値	意味
S_OK	成功。
E_FAIL	内部エラーによる失敗。

コメント

`IsKindOf` メソッドは、以下のいずれかの条件が満たされた場合に、true を返します。

1. エンティティのメタクラスが、strMetaClass パラメータで指定されたメタクラスになっている。
2. エンティティのメタクラスが、strMetaClass パラメータで指定されたメタクラスを直接または間接に継承する。

この最初の条件のみが当てはまるかどうかチェックするために使用できる `GetMetaClassName` メソッドとの違いに注意してください。1828 ページの例 640 は、`IsKindOf` の使用例を示しています。

参照

GetMetaClassName

Unparse

メソッドの呼び出し元のエンティティの具体的な構文表現への逆構文解析。

```
HRESULT Unparse(
          [out, retval] BSTR* strConcreteSyntax);
```

パラメータ

`[out, retval]` `strConcreteSyntax`

メソッドの呼び出し元のエンティティの具体的な構文表現。

戻り値

返された HRESULT から取得される戻り値は、以下のいずれかです。

戻り値	意味
S_OK	成功。
E_FAIL	逆構文解析に失敗しました。詳細については、COM エラー オブジェクトが提供する情報を確認してください。

コメント

Unparse メソッドでは、ほとんどのモデルとモデル フラグメントを逆構文解析できます。特に、Parse の呼び出しによって取得されたすべてのモデル エンティティを逆構文解析できます。ただし、明確に逆構文解析するにはモデル コンテキストが必要なため、それだけでは逆構文解析できないエンティティの種類もあります。そのようなエンティティ上で Unparse を呼び出すと、失敗します。

例 643

以下の例は、1811 ページの例 630 の Parse を使用して作成されたクラスを逆構文解析する方法を示しています。

```
CComBSTR bstrText((TCHAR*) pClass->Unparse());
// bstrText will become "class C { public Integer a; };" (perhaps
// formatted slightly differently)
```

参照

Parse

SetValue

メソッドの呼び出し元のエンティティのメタ特性の値を設定します。値は文字列として表されます。

```
HRESULT SetValue(
          [in] BSTR strMetaFeature,
          [in] BSTR strValue,
          [in, optional] VARIANT nPos);
```

パラメータ

`[in]  strMetaFeature`

値を設定する必要があるメタ特性を指定する文字列。文字列には、メソッドの呼び出し元のエンティティの既存のメタ特性を、大文字と小文字を区別して指定する必要があります。メタ特性を正しく指定しないと、メソッドは失敗します。

`[in]  strValue`

設定する値（文字列として表される）。文字列のフォーマットは、GetValue メソッドで説明しています。

`[in, optional]  nPos`

値を設定する必要があるメタ特性のインデックス。値はこのインデックスの現在値の前に挿入されます。インデックスは 1 から開始されます。インデックス 0 を使用して、メタ特性の最後のインデックスを指定できます。このパラメータを省略すると、デフォルトで 0 になります。インデックスが有効な範囲を超えていると、値はメタ特性内の最後の位置に挿入されます。

戻り値

返された HRESULT から取得される戻り値は、以下のいずれかです。

戻り値	意味
S_OK	成功。
E_FAIL	何らかの理由により値を設定できませんでした。詳細については、COM エラー オブジェクトが提供する情報を確認してください。

コメント

`SetValue` メソッドは、文字列としてエンコードされた値を持つことができるエンティティのすべての書き込み可能メタ特性で使用できます。これは、派生した機能（読み取り専用）、所有者リンク、合成リンクを除いたすべてのメタ特性に当てはまります。

文字列のフォーマットは、GetValue メソッドのドキュメントで説明しています。

例 644

以下の例は、`pClass` のコンテキストで Attribute（属性）を作成し、`SetValue` メソッドで名前 ”CreatedAttribute” を指定する方法を示しています。

```
ITtdEntityPtr pCreatedEntity;
pCreatedEntity = pClass->Create(_T("Attribute"));
if (pCreatedEntity){
    pCreatedEntity->SetValue("Name", "CreatedAttribute");
}
```

やや高度な `SetValue` の使用法として、識別子によってモデル内の参照を設定する方法があります。以下の例では、上で作成した属性のデータ型に対してこれを実行します。参照をバインドされた状態にして、`GetEntity` で、作成された属性のデータ型にナビゲートできるようにするには、属性上で `Bind` メソッドを呼び出します。

```
pCreatedEntity->SetValue("Type", "ref:Integer");
```

参照

SetEntity

GetValue

GetEntity

SetEntity

メソッドの呼び出し元のエンティティのメタ特性の値を設定します。値はエンティティとして表されます。

```
HRESULT SetEntity(
          [in] BSTR strMetaFeature,
          [in] ITtdEntity* pEntity,
          [in, optional] VARIANT nPos);
```

パラメータ

`[in]  strMetaFeature`

値を設定する必要があるメタ特性を指定する文字列。文字列には、メソッドの呼び出し元のエンティティの既存のメタ特性を、大文字と小文字を区別して指定する必要があります。メタ特性を正しく指定しないと、メソッドは失敗します。

`[in]  pEntity`

設定する値（ITtdEntity ポインタとして表される）。pEntity が NULL の場合は、指定されたメタ特性の値がリセットされます。

`[in, optional]  nPos`

値を設定する必要があるメタ特性のインデックス。エンティティはこのインデックスの現在値の前に挿入されます。インデックスは 1 から開始されます。インデックス 0 を使用して、メタ特性の最後のインデックスを指定できます。このパラメータを省略すると、デフォルトで 0 になります。インデックスが有効な範囲を超えていると、エンティティはメタ特性内の最後の位置に挿入されます。

戻り値

返された HRESULT から取得される戻り値は、以下のいずれかです。

戻り値	意味
S_OK	成功。
E_FAIL	何らかの理由により値を設定できませんでした。詳細については、COM エラー オブジェクトが提供する情報を確認してください。

コメント

SetEntity メソッドは、メタクラス型のエンティティのすべての書き込み可能（非派生型）メタ特性上で使用できます。指定されたメタ特性が参照の場合、SetEntity は、特定のエンティティを参照するようにポインタによって参照を設定します。識別子または GUID によって参照を設定するには、代わりに SetValue を使用します。

例 645

以下の例は、SetEntity メソッドを使用し、ポインタによって参照を設定する方法を示しています。以下のように、同じ参照が代わりに識別子によって設定される 1838 ページの例 644 と比較してください。

```
// First find the Integer datatype using FindByGuid
ITtdEntityPtr pType;
pType = pITtdModel->FindByGuid("@Predefined@Integer");
pCreatedEntity->SetEntity("Type", pType);
```

参照

SetValue

GetValue

GetEntity

SetTaggedValue

要素のプロパティ（タグ付き値）を設定するか、または一般にインスタンス式内の値を設定します。

```
HRESULT SetTaggedValue(
          [in] BSTR strSelector,
          [in] BSTR strValue,
          [in, optional] VARIANT overwrite /* = true */);
```

パラメータ

[in] strSelector

設定する値（タグ付き）を指定するセレクタ パターンが含まれている文字列。このパターンのフォーマットは、GetTaggedValue メソッドのドキュメントで説明しています。パターンが正しく構成されていないか、一致しないと、メソッドは失敗します。

[in] strValue

設定する値（文字列としてエンコードされる）。この文字列は、有効な式にする必要があります。文字列は式パーサによって解析され、結果として得られる式が「値のツリー」内のセレクタ パターンによって決定される位置に挿入されます。この値の文字列を式として解析できない場合、メソッドは失敗します。

`[in, optional]  overwrite`

このパラメータが ture の場合は、セレクタ パターンによって選択された既存の値が、新しい値によって上書きされます。このパラメータは、通常の場合、特定の属性のプロパティ（タグ付き値）が 2 つ以上になることは好ましくないため、デフォルトで true になります。

戻り値

返された HRESULT から取得される戻り値は、以下のいずれかです。

戻り値	意味
`S_OK`	成功。
`E_FAIL`	何らかの理由によりプロパティ（タグ付き値）を設定できませんでした。詳細については、COM エラー オブジェクトが提供する情報を確認してください。

コメント

`SetTaggedValue` は、ステレオタイプ適用可能な要素またはインスタンス式（メタクラス InstanceExpr）上で呼び出すことができます。要素上で呼び出すと、セレクタ パターンの一致対象になる要素上で最初に適用された使用可能ステレオタイプ インスタンスでプロパティ（タグ付き値）が設定されます。インスタンス式上で呼び出すと、プロパティ（タグ付き値）が、そのインスタンス式で設定されます（ただし、セレクタ パターンと一致することが前提になります）。

overwrite パラメータを使用して 1 つまたは複数の既存の値を上書きできるようにするには、セレクタ パターンの一致対象であるインスタンス式をインスタンスの元になるシグニチャにバインドする必要があります。`Bind` メソッドを使用して、インスタンス式が正しくバインドされるようにします。

注記

`SetTaggedValue` によって既存の値が上書きされると（overwrite パラメータを使用して）、前の値が削除されます。削除されたこれらの値上のポインタはその後無効になるので、使用できません。クライアント コード内の削除されたエンティティの処理方法については、`Delete` メソッドと比較してください。

例 646

以下の例では、`SetTaggedValue` を使用し、`pPackage` によってポイントされた Package（パッケージ）のコンテキストで作成された Class（クラス）上でプロパティ（タグ付き値）を設定します。使用されるステレオタイプは、1833 ページの例 642 に基づく SignedNote ステレオタイプです。プロパティ（タグ付き値）を設定するには、事前にステレオタイプを適用する必要があります（ここでは、`CreateInstance` メソッドを使用して、その後に `SetEntity` 呼び出しを行う）。

```
ITtdEntityPtr pClass;
pClass = pPackage->Create("Class");

ITtdEntityPtr pStereotypeInstance;
```

```
// Assuming that pStereotype points to the SignedNote stereotype...
pStereotypeInstance->CreateInstance(pStereotype);
pClass->SetEntity("StereotypeInstance", pStereotypeInstance);

pClass->SetTaggedValue("SignedNote (. author .)", "¥"Elvis¥"");
```

設定するプロパティ（タグ付き値）は引用符で囲んで、正当な Charstring 式にする必要があります。引用符を省略すると、プロパティは識別子として解釈されます。

参照

GetTaggedValue

Create

メソッドの呼び出し元のエンティティのコンテキストで、新しいエンティティを作成します。つまり、新しい（直接または間接的な）子がエンティティに追加されます。

```
HRESULT Create(
          [in] BSTR strMetaClass,
          [in, optional, defaultvalue(true)] VARIANT
buildModelForPresentations,
          ß[in, optional] VARIANT strMetaFeature,
          [out, retval] ITtdEntity** ppCreatedEntity);
```

パラメータ

`[in]  strMetaClass`

メタモデル内のメタクラスの名前。文字列には、大文字と小文字を区別して既存のメタクラスを指定する必要があります。メタクラスを正しく指定しないと、メソッドは失敗します。

`[in, optional]  buildModelForPresentations`

作成されたプレゼンテーション要素で、モデル表現を自動的にビルドするかどうかを指定します。このオプションのパラメータは、デフォルトで true になります。新しい要素ではなく既存のモデル要素を参照するように作成されたプレゼンテーション要素を設定する必要がある場合は、このパラメータを false に設定すると効果的です。

`[in, optional]  strMetaFeature`

新しいエンティティを作成する必要があるメタ特性を指定する文字列。このパラメータはオプションです。作成されたエンティティを挿入する場所が明確でない場合にのみ指定します。ほとんどの場合、指定されたメタクラスのエンティティはある特定のメタ特性にのみ適合するので、このパラメータは無視されます。パ

ラメータを指定する場合は、メソッドの呼び出し元のエンティティの既存のメタ特性を、大文字と小文字を区別して指定する必要があります。メタ特性を正しく指定しないと、メソッドは失敗します。

`[out, retval]  ppCreatedEntity`

作成されたエンティティ。

戻り値

返された HRESULT から取得される戻り値は、以下のいずれかです。

戻り値	意味
S_OK	成功。
E_FAIL	作成に失敗しました。詳細については、COM エラー オブジェクトが提供する情報を確認してください。

コメント

`Create` は、エンティティを、メソッドの呼び出し元のエンティティの直接的または間接的な子として作成するための推奨メソッドです。`ITtdModel::New` に似ていますが、以下のような利点があるため、通常はこのメソッドが推奨されます。

- 新しいエンティティを作成し、1 つのアトミック操作としてそのエンティティをモデルにリンクするので、孤立エンティティのリスクが軽減されます。
- 複数の作成「ショートカット」を提供します。つまり、多くの場合は、メソッドの呼び出し元としてエンティティの直接的な子として挿入できない場合でも、エンティティを作成できます。`Create` では、その後に新しいエンティティを挿入するために必要な中間エンティティが自動的に作成されます。ツールの［モデル ビュー］の［新規］メニューの動作と比較してください。
- 作成されたプレゼンテーション要素のモデル表現は、自動的にビルドされます（`buildModelForPresentations` パラメータが false に設定されていない限り）。

例 647

以下の例は、最上位レベル モデル エンティティ（Session）内のパッケージを作成するための `Create` の使用法を示しています。洗練度の低い `ITtdModel::New` メソッドを使用して、同じ動作を実行する 1809 ページの例 629 と比較してください。

```
ITtdModelPtr pITtdModel;
pITtdModel = pITtdModelAccess->LoadProject("MyModel.ttp");
ITtdEntityPtr pSession = pITtdModel;
ITtdEntityPtr pPackage;
pPackage = pSession->Create("Package");
```

ここで、パッケージ内のシーケンス図の作成を続行できます。これは、モデルでシーケンス図をパッケージの直接的な子として挿入できない場合でも可能です。`Create` によって、Package（パッケージ）内にユース ケースが自動的に作成され、ユース

ケース内に Interaction（相互作用）が作成されて、Interaction 内に Interaction Implementation が作成され、最後にその Interaction（相互作用）にシーケンス図が作成されます。

```
ITtdEntityPtr pSQDiagram;
pSQDiagram = pPackage->Create("SequenceDiagram");
```

上記の `Create` の呼び出しでは、エンティティを作成する場所が明確なため、メタ特性を指定する必要はありません。ただし、`pBinaryExpr` によってポイントされたバイナリ式で左オペランドと右オペランドを作成する必要がある以下の例では、メタ特性を指定する必要があります。これは、Expression（式）が適合できる BinaryExpr の 2 つの可能なメタ特性（つまり、LeftOperand と RightOperand）が存在するためです。

```
// Create an assignment expression: myVar = 4
ITtdEntityPtr pIdentifier;
ITtdEntityPtr pValue;
pIdentifier = pBinaryExpr->Create("Ident", "LeftOperand");
pValue = pBinaryExpr->Create("IntegerValue", "RightOperand");
pIdentifier->SetValue("Name", "myVar");
pValue->SetValue("Value", "4");
```

参照

New

CreateInstance

メソッドの呼び出し元のエンティティのインスタンスを作成します。したがって、そのエンティティはインスタンス化できるシグニチャになっている必要があります。

```
HRESULT CreateInstance(
          [out, retval] ITtdEntity** ppInstance);
```

パラメータ

`[out, retval]  ppInstance`

作成されたインスタンス。

戻り値

返された HRESULT から取得される戻り値は、以下のいずれかです。

戻り値	意味
`S_OK`	成功。
`E_FAIL`	作成に失敗しました。詳細については、COM エラー オブジェクトが提供する情報を確認してください。

コメント

CreateInstance は、シグニチャのインスタンスを作成する UML セマンティックを実装します。インスタンスは、モデル内で InstanceExpr メタクラスによって表されます。

注記

返されたインスタンスを処理する役割はクライアントが担います。インスタンスはモデルに挿入するか、削除して、メモリ リークを避ける必要があります。

例 648

CreateInstance は、任意のシグニチャ上で呼び出せますが、要素に適用するためにステレオタイプ上で呼び出されることもあります。以下の例は、定義済みのステレオタイプ「usecase」を、pOperation によってポイントされる操作上に適用する方法を示しています。

```
ITtdEntityPtr pStereotype;
pStereotype = pITtdModel->FindByGuid("@Predefined@usecase");
ITtdEntityPtr pInstance;
pInstance = pStereotype->CreateInstance();
pOperation->SetEntity("StereotypeInstance", pInstance);
```

Delete

メソッドの呼び出し元のエンティティを削除します。

```
HRESULT Delete();
```

戻り値

返された HRESULT から取得される戻り値は、以下のいずれかです。

戻り値	意味
S_OK	成功。
E_FAIL	内部エラーによる失敗。

コメント

Delete を使用して、エンティティをモデルから削除し、さらにエンティティによって占有されているメモリを削除します。エンティティによって他のエンティティが所有されている場合は（直接的または間接的に）、これらのエンティティも、複合関係のセマンティックに従って削除されます。

重要！

削除されたエンティティ上のポインタは削除後に無効になるので、使用しないように注意してください。

例 649

以下の例では、`Delete` を使用して、1843 ページの例 647 で作成した Package（パッケージ）を削除します。`Detach` は、削除後にスマート ポインタ上で呼び出します。削除後に無効になるそのインターフェイス ポインタから分離するために使用します。その後に、`pPackage` を新しいエンティティに割り当てることができます。

```
pPackage->Delete();
pPackage->Detach(); // Important since the pointer is invalid!
```

`Detach()` は、C++ スマート ポインタ クラスを処理するための Microsoft API の一部です（`_com_ptr_t::Detach`）。

XMLEncode

メソッドの呼び出し元としてエンティティを、XML 表現にエンコードします。

```
HRESULT XMLEncode(
         [out, retval] BSTR* strXMLEncoding);
```

パラメータ

`[out, retval]` `strXMLEncoding`

メソッドの呼び出し元のエンティティの XML エンコーディングが含まれている文字列。

戻り値

返された HRESULT から取得される戻り値は、以下のいずれかです。

戻り値	意味
`S_OK`	成功。
`E_FAIL`	XML エンコーディングに失敗しました。詳細については、COM エラー オブジェクトが提供する情報を確認してください。

コメント

通常は、XMLEncode と ITtdModel::XMLDecode を組み合わせて使用します。これらのメソッドの組み合わせは、文字列表現との間でモデル フラグメントをシリアル化する手段になります。1812 ページの例 631 は、XMLEncode の使用法の例を示しています。

参照

ITtdModel::XMLDecode

MetaVisit

このメソッドを呼び出したエンティティをルートとする、モデル フラグメントをトラバースします。トラバース中に検出されたエンティティごとに、コールバック インターフェイスのメソッドが呼び出されます。呼び出し元は、このコールバックを使ってトラバースしたエンティティに対して作用できます。

```
HRESULT MetaVisit(
          [in] ITtdMetaVisitCallback* pMetaVisitCallback,
          [in, optional] VARIANT visitAll /* false */);
```

パラメータ

`[in]  pMetaVisitCallback`

モデルのトラバース時にコールバックに使用されるインターフェイスのポインタ。このパラメータが NULL の場合は、メソッドは失敗します。

`[in, optional]  visitAll`

ライブラリ パッケージと定義済みパッケージをトラバースするかどうかを制御するフラグ。このパラメータを省略すると、デフォルトで false になります。つまり、ライブラリ パッケージと定義済みパッケージをトラバースしません。これらのパッケージがデフォルトではトラバースされないのは、パフォーマンス上の理由（特に、メソッドがスクリプト クライアントから呼び出される場合を考慮）からです。

戻り値

返された HRESULT から取得される戻り値は、以下のいずれかです。

戻り値	意味
S_OK	成功。
E_FAIL	内部エラーによる失敗。
E_POINTER	無効なポインタ引数。pMetaVisitCallback ポインタは、NULL にはできません。

コメント

MetaVisit は、モデルまたはモデルの一部をトラバースするのに便利なメソッドです。UML メタモデルの知識は必要ありません。どのメタ特性かを問わず、あるエンティティの直接、間接の合成子がトラバースの対象になります。

コールバック インターフェイス ITtdMetaVisitCallback を実装することで、クライアントは、エンティティをアクセスした時にそのエンティティに作用させたい機能を実行できます。MetaVisit メソッドは、さまざまに応用できます。

例 650

以下の例は、ロードされたモデル内のすべての定義を検索するための MetaVisit の使用法を示しています。

```
ITtdModelPtr pITtdModel;
pITtdModel = pITtdModelAccess->LoadProject("MyModel.ttp");
ITtdEntityPtr pSession = pITtdModel;
pSession->MetaVisit(pCallbackHandler);
```

ここでは、pCallbackHandler で ITtdMetaVisitCallback インターフェイスを実装するあるクラスをポイントしている仮定しています。たとえば、モデル内のすべての定義の名前を出力することがトラバースの目的だとします。その場合、pCallbackHandler は、以下のクラスのインスタンスをポイントすると想定できます。

```
class CDefinitionPrinter : public ITtdMetaVisitCallback {
public:

  virtual HRESULT __stdcall raw_OnVisitedEntity (ITtdEntity*
pVisitedEntity){
    if (pVisitedEntity->IsKindOf(_T("Definition"))){
      CComBSTR bstrValue((TCHAR*) pVisitedEntity-
>GetValue("Name"));
    }
    return S_OK;
  }

  virtual HRESULT __stdcall raw_OnAfterVisitedEntity (ITtdEntity*
pVisitedEntity){
    return S_OK;
}

  virtual HRESULT STDMETHODCALLTYPE QueryInterface(REFIID iid, void
** ppvObject){
    *ppvObject = NULL;
    if (iid == __uuidof(ITtdMetaVisitCallback)) {
      *ppvObject = static_cast<ITtdMetaVisitCallback*>(this);
      AddRef();
      return S_OK;
    }
  return E_FAIL;
    }

  virtual ULONG STDMETHODCALLTYPE AddRef(){return 0;}; // No ref
counting

  virtual ULONG STDMETHODCALLTYPE Release(){return 0;}; // No ref
counting
};
```

ITtdMetaVisitCallback インターフェイスには、アクセスされたエンティティごとに呼び出される `OnVisitedEntity` という メソッドがあります。アクセスされたエンティティは、パラメータとしてこのメソッドに渡されます。上の例のクラスでは、`IUnknown` インターフェイスのメソッドも実装する必要があります。これは、ITtdMetaVisitCallback によって `IUnknown` が継承されるためです。これは他のすべての COM インターフェイスと同様です。

第 2 のコールバックメソッド OnAfterVisitedEntity もあります。このメソッドもアクセスされたエンティティごとに呼び出されますが、OnVisitedEntity とは異なり、あるエンティティについて、そのエンティティの合成子がすでにアクセスされている場合のみ呼び出されます。したがって、呼び出し元はモデルトラバースにおける「後方再帰（back recursion）」時に何らかの動作をできるようになります。

参照

New

OnVisitedEntity

MetaVisitEx

MetaVisit メソッドの拡張バージョンであり、参照もモデル トラバースの対象にします。

```
HRESULT MetaVisitEx(
          [in] ITtdMetaVisitCallback* pMetaVisitCallback,
          [in, optional] VARIANT visitAll /* false */);
          [in, optional] VARIANT visitRefs /* false */);
```

パラメータ

MetaVisitEx では、以下のようにオプションのパラメータが MetaVisit シグニチャに追加されます。

`[in, optional]  visitRefs`

参照をトラバースするかどうかを制御するフラグ。このパラメータを省略すると、デフォルトで false になります。つまり、参照のモデルはトラバースされません。

戻り値

MetaVisit を参照してください。

Bind

このメソッドを呼び出したエンティティをルートとするモデル フラグメント内のすべての参照について、バインドを試みます。このメソッドは、エンティティ上の特定の参照のみをバインドする場合にも使用できます。

```
HRESULT Bind(
         [in, optional] VARIANT strMetaFeature);
```

パラメータ

`[in, optional]  strMetaFeature`

このパラメータには、バインドする必要があるメタ特性を指定する文字列を指定します。文字列には、メソッドの呼び出し元のエンティティの既存のメタ特性を、大文字と小文字を区別して指定する必要があります。メタ特性を正しく指定しないと、メソッドは失敗します。

戻り値

返された HRESULT から取得される戻り値は、以下のいずれかです。

戻り値	意味
S_OK	成功。
E_FAIL	エラーが発生しました。詳細については、COM エラー オブジェクトが提供する情報を確認してください。

コメント

`Bind` はモデル内のすべての参照のバインドを試行します。`LoadProject` と `LoadFile` では暗黙的に `Bind` が呼び出されます。これは、新規にロードされたモデルのすべてのリンクと参照をナビゲートできるようにするためです。

通常、`Bind` を呼び出す必要があるのは、`SetValue` メソッドを使用して、参照が名前または GUID によって設定されている場合です。その場合、メタ特性をパラメータとして `Bind` メソッドに渡して、その特定のメタ特性のみをバインドできるようにします。

例 651

以下の例は、`LoadFile` メソッドを使用して、ファイル `MyModel.u2` をロードする方法を示しています。

```
ITtdModelPtr pITtdModel;
pITtdModel = pITtdModelAccess->LoadFile("MyModel.u2");
```

`LoadFile` では、モデルがロードされたときに `Bind` を暗黙的に呼び出すので、バインド可能なすべてのリンクと参照がバインドされて、`GetEntity` メソッドまたは `GetEntities` メソッドでナビゲートできるようになります。

pITtdSession 内にパッケージを作成して、そのパッケージ内に属性を作成します。属性のデータ型が名前によって定義済みのデータ型 Integer に設定され、`GetEntity` メソッドを使用してデータ型にナビゲートできるように、`Bind` が、属性の Type メタ特性上で呼び出されます。

```
ITtdEntityPtr pPackage, pAttribute;
pPackage = pITtdSession->Create(_T("Package"));
pAttribute = pPackage->Create(_T("Attribute"));
pAttribute->SetValue("Type", "ref:Integer");
pAttribute->Bind("Type");

ITtdEntityPtr pIntegerDatatype;
pIntegerDatatype = pAttribute->GetEntity("Type");
```

Clone

エンティティのクローンを作成します。デフォルトでは、クローンはバインドされないで、新しい一意の GUID をとります（エンティティ自体のコピーとすべての内包されるエンティティのコピーが新しい一意の GUID を取得します）。

```
HRESULT Clone(
          [in, optional] VARIANT preserveBindings,
          [in, optional] VARIANT preserveGuids,
          [out, retval] ITtdEntity** result);
```

パラメータ

`[in, optional]  preserveBindings`

このオプションのパラメータを指定する場合は、元のエンティティのバインドをクローンで保持するかどうかを指定するブール値を使用します。デフォルトでは、バインドは保持されません。

`[in, optional]  preserveGuids`

このオプションのパラメータを指定する場合は、元のエンティティ（およびその子）の GUID をクローンで保持するかどうかを指定するブール値を使用します。デフォルトでは、GUID は保持されません。

`[out, retval]  result`

作成されたクローン。

戻り値

返された HRESULT から取得される戻り値は、以下のいずれかです。

戻り値	意味
S_OK	成功。
E_FAIL	エラーが発生しました。詳細については、COM エラー オブジェクトが提供する情報を確認してください。

コメント

Clone を使用して、エンティティと同一のコピーを作成します。「ディープ」クローニングが行われ、クローンには元のエンティティのすべての子が再帰的にコピーされます。

重要！

GUID を変更せずにエンティティのクローンを作成するときは注意してください。元のエンティティと同じモデルにこのようなクローンを挿入してはなりません。さもなければ、GUID の衝突が発生します。GUID が衝突しているモデルを保存した場合、再ロードができないことがあります。

クローンのバインディングを保持するには、元のエンティティはモデルに所属していなければなりません（すなわち、元のエンティティ上で呼び出された GetModel を返してはなりません）。

注記

戻されたエンティティの処理はクライアントの責任になります。戻されたエンティティは、メモリリークを防ぐためにモデルに挿入するか、削除します。

Move

エンティティをモデル内の現在の位置から新しい所有者ののコンテキストに移動します。

```
HRESULT Move(
          [in] ITtdEntity* newOwner,
          [in, optional] VARIANT metafeature,
          [in, optional] VARIANT index);
```

パラメータ

`[in]` `newOwner`

エンティティの移動先となる新しい所有者。

`[in, optional]` `metafeature`

エンティティの移動先となる新しい所有者のメタ特性。このパラメータはオプションです。エンティティを移動する場所が明確でない場合にのみ指定します。ほとんどの場合、移動したエンティティは 1 つの特定のメタ特性のみに適合できるので、このパラメータは無視されます。パラメータを指定する場合は、メソッ

ドの呼び出し元の newOwner エンティティの既存のメタ特性を、大文字と小文字を区別して指定する必要があります。メタ特性を正しく指定しないと、メソッドは失敗します。

```
[in, optional]   index
```

ターゲットのメタ特性が複数の多重度を持つ場合、このオプション パラメータを使用して、エンティティの移動先となる位置を指定できます。

戻り値

返された HRESULT から取得される戻り値は、以下のいずれかです。

戻り値	意味
S_OK	成功。
E_FAIL	エラーが発生しました。詳細については、COM エラー オブジェクトが提供する情報を確認してください。

コメント

Move を使用して、エンティティをある所有者から別の所有者に移動します。エンティティの ID（GUID）は失われません。エンティティが所有者を持たない場合、モデルに挿入するには SetEntity を使用したほうがよいでしょう。

参照

SetEntity

GetModel

エンティティが所属するモデルを返します。

```
HRESULT GetModel(
          [out, retval] ITtdModel** result);
```

パラメータ

```
[out, retval]   result
```

エンティティが所属するモデル、エンティティがモデルに所属しない場合は NULL。

戻り値

メソッドは必ず成功します（S_OK を返します）。

コメント

`GetModel` を使用して、エンティティが所属するモデルの最上位レベル エンティティの `ITtdModel` インターフェイスを取得します。このメソッドは、ITtdEntity インタフェースのコンテキストから ITtdModel インタフェースを取得するもっとも便利な方法としてよく使用されます。

UnlinkFromOwner

現在のオーナーからモデル内のエンティティへのリンクを解除します。

```
HRESULT UnlinkFromOwner();
```

戻り値

メソッドは常に成功します。(つまり S_OK が戻ります)

コメント

`UnlinkFromOwner` を使用して、エンティティをモデル内の現在のオーナーからリンク解除します。呼び出しの後はエンティティは誰からもポイントされなくなります。つまり、GetOwner または GetModel 呼び出しは NULL を戻します。

注記

エンティティは削除されません。したがって、呼び出しの後にどう処理するかはクライアントの責任です。メモリリークを防ぐためにモデルに再び挿入するか、削除します。

参照

Delete

Move

Replace

エンティティを他のエンティティで置き換えます。元のエンティティは削除されません

```
HRESULT Replace(
         [in] ITtdEntity* replacementEntity);
```

パラメータ

```
[in]             replacementEntity
```

メソッドが呼ばれたエンティティを、このエンティティが置き換えます。置き換えるエンティティが元のエンティティのある場所に挿入できない場合は、メソッドは失敗します。

戻り値

HRESULT から取得される戻り値は以下のとおりです :

戻り値	意味
S_OK	エンティティは正しく置き換わりました。
E_FAIL	エラーが発生しました。詳細は COM エラーオブジェクトが提供する情報を参照してください。

コメント

Replace を使用してエンティティを他のエンティティで置き換えます。置き換えエンティティは、モデル内のもとのエンティティがあった場所に挿入されます。

注記

元のエンティティは削除されません。したがって、呼び出しの後にどう処理するかはクライアントの責任です。メモリリークを防ぐためにモデルに再び挿入するか、削除します。

注記

メソッドが呼ばれたエンティティが参照を表す識別子の場合は、replacementEntity のクローンで置き換わります。replacementEntity 自身ではありません。この場合、呼出し後に置き換えエンティティをどうするかはクライアントの責任になります。

GetContainerMetaFeature

エンティティが包含されるメタ特性の名称を取得します。

```
HRESULT GetContainerMetaFeature(
          [out, retval] BSTR* result);
```

パラメータ

```
[out, retval]    result
```

コンテナメタ特性の名称。この文字列は、エンティティが他から参照されていない場合は空になります。

戻り値

メソッドは常に成功します。(つまり S_OK が戻ります)

コメント

GetContainerMetaFeature を使用して、エンティティが含まれているメタ特性の名前を検索します。このメタ特性名は、たとえば SetEntity、GetEntity 、GetEntities などの呼び出しで使用されます。

FindByName

メソッドが呼び出されたエンティティのコンテキストから名前検索を実行します。その目的は、特定の名前 (修飾名) のエンティティを検索するためです。

```
HRESULT FindByName(
         [in] BSTR strName,
         [out, retval] ITtdEntity* result);
```

パラメータ

`[in]` `strName`

検索対象のエンティティの名前。名前は有効な URL 識別子として修飾されている場合もありえます。

`[out, retval]` `result`

検索されたエンティティ、または、合致するエンティティがなかった場合は、NULL。

戻り値

HRESULT から取得される戻り値は以下のとおりです :

戻り値	意味
S_OK	成功。
E_FAIL	エラーが発生しました。詳細は COM エラーオブジェクトが提供する情報を参照してください。

コメント

FindByName を使用して定義を名前で検索します。この名前を通じて、メソッドが呼ばれたエンティティのコンテキストから参照可能になります。これは FindByGuid を使用して、GUID ではなく名前でモデル内のエンティティを検索する方法の別法です。

éQèý

FindByGuid

GetDescriptiveName

エンティティのテキスト形式の詳細を取得します。

```
HRESULT GetDescriptiveName(
          [out, retval] BSTR* result);
```

パラメータ

`[out, retval]` `result`

エンティティのテキスト説明。

戻り値

メソッドは常に成功します。(つまり S_OK が戻ります)

コメント

GetDescriptiveName を使用して、メソッドが呼ばれたエンティティの文字型の説明を取得します。説明にはエンティティの メタクラス 、その名称 (イベントクラスの完全シグニチャ) およびモデル内での場所が入ります。

ITtdEntities

ITtdEntities インターフェイスは、エンティティのコレクションを表します。クライアントではほとんどの場合、コレクションからの読み取りのみが必要になりますが（つまり、コレクションのエンティティのトラバースとアクセス)、このインターフェイスは、読み取りと書き込みの両方を実行できるコレクションを指定します。ITtdEntities インターフェイスに対応する UML メタモデルには、特定のメタクラスはありません。

ITtdEntities インターフェイスのメソッドは、読み取り／書き込み COM コレクション インターフェイス上で使用できる通常のメソッドです。

`_NewEnum`	コレクションの列挙インターフェイス ポインタを返して、含まれているエンティティを繰り返し使用できるようにします。
`Item`	コレクション内の指定されたインデックスにあるエンティティを返します。
`Count`	コレクション内のエンティティの数を返します。
`Add`	エンティティをコレクションに追加します。
`Remove`	コレクションからエンティティを削除します。

_NewEnum

メソッドの呼び出し元のエンティティ コレクションの列挙インターフェイス ポインタを返します。列挙インターフェイスは、標準の自動化列挙インターフェイスです。コレクション内の一連のエンティティをたどるときに使用できます。

_NewEnum は、読み取り専用プロパティとして定義されます。

```
HRESULT _NewEnum(
          [out, retval] IUnknown** ppUnk);
```

パラメータ

[out, retval]　ppUnk

コレクションの列挙インターフェイスのポインタ。

戻り値

返された HRESULT から取得される戻り値は、以下のいずれかです。

戻り値	意味
S_OK	成功。
E_FAIL	エラーが発生しました。詳細については、COM エラー オブジェクトが提供する情報を確認してください。

コメント

_NewEnum メソッドは、ITtdEntities コレクション内のエンティティを繰り返し使用するための 1 つの方法です。他の方法は、コレクションのインデックスの繰り返しループを行うことと、Item メソッドを呼び出して、特定のインデックスにあるエンティティを取得することです。通常は、_NewEnum のほうが効率の良い代替メソッドになります。

例 652

以下の例は、_NewEnum メソッドを使用して、クラス（pClass によってポイントされた）内の属性を繰り返し使用する方法を示しています。

```
ITtdEntitiesPtr pAttributes = pClass->GetEntities(_T("Attribute"));
// Iterate through the collection using the enumerator...
// Get the VARIANT enumerator from the collection
IEnumVARIANTPtr spEnum = pAttributes->Get_NewEnum();
// nBatchSize is the number of items requested in each call to
IEnumVARIANT::Next.
// The actual number of items returned may not equal nBatchSize.
const ULONG nBatchSize = 5;
// nReturned will store the number of items returned by a call to
// IEnumVARIANT::Next
ULONG nReturned = 0;
```

```
// arrVariant is the array used to hold the returned items
VARIANT arrVariant[nBatchSize] = {0};

HRESULT hr = E_UNEXPECTED;
do {
  hr = spEnum->Next(nBatchSize, &arrVariant[0], &nReturned);
  if (FAILED(hr))
    return hr;

  ITtdEntityPtr pAttribute;
  for (ULONG i = 0; i < nReturned; ++i){
    _variant_t vt(arrVariant[i]);
    pAttribute = vt;
    if (pAttribute){
      // Do something with pAttribute
    }
    ::VariantClear(&arrVariant[i]);
  }
} while (hr != S_FALSE); // S_FALSE indicates end of collection
```

_NewEnum は、ITtdEntities インターフェイスの読み取り専用プロパティとして定義されます。したがって、接頭辞「Get」を使用して呼び出されます。

参照

Item

Item

メソッドの呼び出し元のエンティティ コレクション内の指定されたインデックスにあるエンティティを返します。

Item は、読み取り専用プロパティとして定義されます。

```
HRESULT Item(
          [in] long Index,
          [out, retval] ITtdEntity** ppEntity);
```

パラメータ

[in] Index

エンティティを抽出するコレクションのインデックス。インデックスは、1（コレクション内の最初のエンティティ）と Count（コレクション内のエンティティの数）を呼び出した結果の範囲内にある有効なインデックスになっている必要があります。

[out, retval] ppEntity

コレクション内の指定されたインデックスにあるエンティティ、またはそのようなエンティティが存在しない場合は NULL。

戻り値

返された HRESULT から取得される戻り値は、以下のいずれかです。

戻り値	意味
S_OK	成功。
E_FAIL	エラーが発生しました。詳細については、COM エラー オブジェクトが提供する情報を確認してください。

コメント

Item メソッドを使用して、ITtdEntities コレクション内の任意のエンティティにアクセスします。Item を使用して、コレクション内のエンティティを繰り返し使用することもできますが、通常は、_NewEnum メソッドから取得した列挙インターフェイスを使用する場合よりも効率が低下します。

例 653

以下の例は、Item メソッドを使用して、クラス（pClass によってポイントされた）内の属性を繰り返し使用する方法を示しています。代わりに _NewEnum メソッドを使用する方法は、1858 ページの例 652 で説明しています。

```
ITtdEntitiesPtr pAttributes = pClass->GetEntities(_T("Attribute"));
// Iterate through the collection using the Item method...

long lCount = 0;
lCount = pAttributes->Count;

ITtdEntityPtr pAttribute;
// N.B. Index are 1-based
for (long i = 1; i <= lCount; i++){
  pAttribute = pAttributes->GetItem(i);
  if (pParameter != 0){
    // Do something with pAttribute
  }
}
```

Item は、ITtdEntities インターフェイスの読み取り専用プロパティとして定義されます。したがって、接頭辞「Get」を使用して呼び出されます。

参照

_NewEnum

Count

Count

メソッドの呼び出し元のエンティティ コレクション内のエンティティの数を返します。

Count は、読み取り専用プロパティとして定義されます。

```
HRESULT Count(
          [out, retval] long* pVal);
```

パラメータ

`[out, retval]  pVal`

コレクション内のエンティティの数。

戻り値

返された HRESULT から取得される戻り値は、以下のいずれかです。

戻り値	意味
S_OK	成功。
E_FAIL	エラーが発生しました。詳細については、COM エラー オブジェクトが提供する情報を確認してください。

コメント

`Count` メソッドを使用して、ITtdEntities コレクション内のエンティティの数を判定します。NULL ポインタを含めて、コレクション内のすべてのアイテムがカウントされます。1860 ページの例 653 には、`Count` の使用法の例があります。

Add

メソッドの呼び出し元のエンティティ コレクションにエンティティを追加します。

```
HRESULT Add(
          [in] ITtdEntity* pEntity,
          [in, optional] VARIANT nIndex);
```

パラメータ

`[in]  pEntity`

コレクションに追加するエンティティ。

`[in, optional]  nIndex`

コレクション内のどの場所にエンティティを挿入するかを指定するインデックス。インデックスは 1 から開始されます。インデックス 0 を使用して、コレクションの終わりにエンティティを追加するように指定できます。このパラメータ

を省略すると、デフォルトで 0 になります。インデックスを指定する場合は、有効な範囲内にする必要があります。範囲を超えていると、メソッドは失敗します。

戻り値

返された HRESULT から取得される戻り値は、以下のいずれかです。

戻り値	意味
S_OK	成功。
E_FAIL	エラーが発生しました。詳細については、COM エラー オブジェクトが提供する情報を確認してください。

コメント

Add メソッドを使用して、ITtdEntities コレクションにエンティティを追加します。ITtdEntities コレクションには NULL ポインタを含めることができます。また、同じエンティティを 2 回以上含めることもできます。

例 654

以下の例では、GetEntities を使用して、エンティティのコレクションを取得します。コレクションの最初のエンティティは、その後にもう一度コレクションの終わりに追加されます。最後に Remove メソッドを使用して、コレクション内のエンティティのすべてのオカレンスを削除します。

```
ITtdEntitiesPtr pAttributes = pClass->GetEntities(_T("Attribute"));
ITtdEntityPtr pEntity = pAttributes->GetItem(1);
pAttributes->Add(pEntity); // Add once more last in collection
pAttributes->Remove(pEntity); // Removes all occurrences of pEntity
```

Remove

メソッドの呼び出し元のエンティティ コレクションからエンティティのすべてのオカレンスを削除します。

```
HRESULT Remove(
         [in] ITtdEntity* pEntity);
```

パラメータ

[in] pEntity

コレクションから削除するエンティティ。このパラメータを NULL にすることはできません。

戻り値

返された HRESULT から取得される戻り値は、以下のいずれかです。

戻り値	意味
S_OK	成功。
E_POINTER	NULL ポインタは、pEntity パラメータとして指定されます。

コメント

Remove メソッドを使用して、エンティティのすべてのオカレンスを ITtdEntities コレクションから削除します。1862 ページの例 654 には、Remove メソッドの使用法の例があります。

ITtdResource

ITtdResource インターフェイスは、UML モデルを永続的に格納できるリソースを表すメタモデルの Resource クラスによって実装されます。通常、Resource は .u2 ファイルに相当します。

注記

Resource（リソース）は Entity（エンティティ）でもあるので、ITtdResource から ITtdEntity への QueryInterface は常に成功します。

ITtdResource には、以下のメソッドがあります。

Save	リソースに関連付けられたモデル エンティティを保存します（通常は、対応する .u2 ファイルを保存します）。

Save

メソッドの呼び出し元のリソースに関連付けられているモデル エンティティを保存します。リソースが .u2 ファイルを表している一般的ケースでは、そのファイルが保存されることになります。

```
HRESULT Save();
```

戻り値

返された HRESULT から取得される戻り値は、以下のいずれかです。

戻り値	意味
S_OK	リソースに関連付けられたエンティティは、正しくリソースに保存されました。
E_FAIL	保存中にエラーが発生しました。このエラーが発生するのは、たとえば、リソースのファイルが読み取り専用の場合、またはディスク上のスペースが不足している場合です。詳細については、COM エラー オブジェクトが提供する情報を確認してください。

コメント

Save メソッドを使用して、リソースに関連付けられたエンティティ（つまりそのルート）をリソースに保存します。モデル全体を保存する必要がある場合は、ITtdModel::Save を呼び出したほうが便利です。

例 655

以下の例では、Save を使用して、1809 ページの例 629 で作成した Package（パッケージ）をその専用ファイルに保存します。

```
ITtdEntityPtr pResource;
pResource = pITtdModel->CreateResource("D:¥temp¥COMtest.u2");

// Insert the created package pPackage as a root of pResource
pResource->SetEntity("Root", pPackage);

pITtdResource->Save(); // Saves all resources in the model
```

参照

Save

ITtdPresentationElement

ITtdPresentationElement インターフェイスは、メタモデルの PresentationElement クラスによって実装されます。プレゼンテーション要素は、たとえば、ダイアグラムシンボルやラインなどの図形的な表示が含まれた要素です。

注記

PresentationElement は Entity（エンティティ）でもあるので、ITtdPresentationElement から ITtdEntity への QueryInterface は常に成功します。

ITtdPresentationElement には、以下のメソッドがあります。

GenerateEMF	プレゼンテーション要素の EMF ファイル（Enhanced Meta File）を生成します。(非推奨)
GenerateEMFEx	イメージのスケーリングをサポートするプレゼンテーション要素の EMF ファイル（Enhanced Meta File）を生成します。
	プレゼンテーション要素の画像ファイルを生成します。

GenerateEMF

プレゼンテーション要素のグラフィック表示のための EMF ファイル（Enhanced Meta File）を生成します。この関数は削除されました。代わりに GenerateEMFEx を使用してください。この EMF ファイルでは、プレゼンテーション要素がツールのエディタに表示されるときと同じ外観になります。

```
HRESULT GenerateEMF(
          [in] BSTR strFileName,
          [in, optional] VARIANT maxWidth,
          [in, optional] VARIANT maxHeight,
          [in, optional] VARIANT optimizeForVectorGraphics,
          [in, optional] VARIANT includeFrame);
```

パラメータ

`[in]  strFileName`

生成する EMF ファイルのファイル名。strFileName が相対パス指定の場合、そのパスはクライアント アプリケーションの現在の作業ディレクトリを基準にして解釈されます。

`[in, optional]  maxWidth`

`[in, optional]  maxHeight`

これらのオプションのパラメータを使用して、生成されたイメージの最大サイズを指定できます。イメージは、指定されたサイズに合わせて拡大・縮小されます。パラメータを省略すると、生成されたイメージはツールのエディタに表示されるサイズと同じになります。高さと幅の数値の単位は、1/10 mm です。現在、これらのパラメータが考慮されるのは、メソッドの呼び出し元のプレゼンテーション要素がダイアグラムの場合のみです。

`[in, optional]  optimizeForVectorGraphics`

このオプションのパラメータが true に設定されていると、EMF 生成がベクトルグラフィックス用に最適化されます。デフォルトの振る舞いでは、この最適化は行われません。現在、このパラメータが考慮されるのは、メソッドの呼び出し元のプレゼンテーション要素がダイアグラムの場合のみです。このパラメータは、このパラメータは削除が考慮されており、下位互換を保つために使用されます。

`[in, optional]  includeFrame`

このオプションのパラメータは、ダイアログのフレーム シンボルを EMF 生成に含めるかどうかを指定します。デフォルトでは、フレーム シンボルが含まれます。現在、このパラメータが考慮されるのは、メソッドの呼び出し元のプレゼンテーション要素がダイアグラムの場合のみです。

戻り値

返された HRESULT から取得される戻り値は、以下のいずれかです。

戻り値	意味
S_OK	EMF ファイルは正しく生成されました
E_FAIL	エラーが発生しました。このエラーが発生するのは、たとえば、EMF ファイルを指定された場所で保存できない場合、または必須の Tau ライセンスを取得できなかった場合です。詳細については、COM エラー オブジェクトが提供する情報を確認してください。

コメント

GenerateEMF メソッドは、たとえば、UML モデルからレポートを生成するクライアントで、プレゼンテーション要素を可視化するための手段として使用できます。このメソッドは、すべての種類のプレゼンテーション要素に作用します。また、プレゼンテーション要素に他のプレゼンテーション要素が含まれている場合は（たとえば、ダイアグラムの場合と同じように）、含まれているこれらのプレゼンテーション要素も EMF ファイルにインクルードされます。

GenerateEMFEx

プレゼンテーション要素の図形的表示のための EMF ファイル（Enhanced Meta File）を生成します。この EMF ファイルでは、プレゼンテーション要素がツールのエディタに表示されるときと同じ外観になります。プレゼンテーション要素の EMF ファイル作成のための推奨メソッドです。

```
HRESULT GenerateEMFEx(
         [in] BSTR strFileName,
         [in, optional] VARIANT maxWidth,
         [in, optional] VARIANT maxHeight,
         [in, optional] VARIANT optimizeForVectorGraphics,
         [in, optional] VARIANT includeFrame);
         [in, optional] VARIANT scaleFactor);
```

パラメータ

`[in]  strFileName`

生成する EMF ファイルのファイル名。`strFileName` が相対パス指定の場合、そのパスはクライアント アプリケーションの現在の作業ディレクトリを基準にして解釈されます。

`[in, optional]  maxWidth`

`[in, optional]  maxHeight`

これらのオプションのパラメータを使用して、生成されたイメージの最大サイズを指定できます。`scaleFactor` が指定されていない場合、イメージは、指定されたサイズに合わせて拡大・縮小されます。パラメータを省略すると、生成されたイメージはエディタに表示されるサイズと同じになります。高さと幅の数値の単位は、1/10 mm です。現在、これらのパラメータが考慮されるのは、メソッドの呼び出し元のプレゼンテーション要素がダイアグラムの場合のみです。

`[in, optional]  optimizeForVectorGraphics`

このオプションのパラメータが true に設定されていると、EMF 生成がベクトル グラフィックス用に最適化されます。デフォルトの振る舞いでは、この最適化は行われません。現在、このパラメータが考慮されるのは、メソッドの呼び出し元のプレゼンテーション要素がダイアグラムの場合のみです。このパラメータは、このパラメータは削除が考慮されており、下位互換を保つために使用されます。

`[in, optional]  includeFrame`

このオプションのパラメータは、ダイアログのフレーム シンボルを EMF 生成に含めるかどうかを指定します。デフォルトでは、フレーム シンボルが含まれます。現在、このパラメータが考慮されるのは、メソッドの呼び出し元のプレゼンテーション要素がダイアグラムの場合のみです。

`[in, optional]  scaleFactor`

オプションのパラメータを指定すると、イメージの生成前に、元のダイアグラムが拡大・縮小されます。拡大・縮小率は、整数で指定します。この値は元のダイアグラム サイズに対するパーセントに変換されます。`scaleFactor` と `maxWidth`/`maxHeight` の両方を引数として指定すると、操作の結果として複数のイメージが生成されます。`scaleFactor` パラメータを指定した場合、生成後のファイル名は `strFileName` パラメータを指定した場合と同じになりますが、ファイル拡張子の前に数字が振られます。

戻り値

返された HRESULT から取得される戻り値は、以下のいずれかです。

戻り値	意味
S_OK	EMF ファイルは正しく生成されました
E_FAIL	エラーが発生しました。このエラーが発生するのは、たとえば、EMF ファイルを指定された場所で保存できない場合、または必須の Tau ライセンスを取得できなかった場合です。詳細については、COM エラー オブジェクトが提供する情報を確認してください。

コメント

GenerateEMFEx メソッドは、たとえば、UML モデルからレポートを生成するクライアントで、プレゼンテーション要素を可視化するための手段として使用できます。このメソッドは、すべての種類のプレゼンテーション要素に作用します。また、プレゼンテーション要素に他のプレゼンテーション要素が含まれている場合は（たとえば、ダイアグラムの場合と同じように）、含まれているこれらのプレゼンテーション要素も EMF ファイルにインクルードされます。

例 656

以下の例は、GenerateEMFEx メソッドを使用して、pPackage によってポイントされたパッケージ内のダイアグラムごとに EMF ファイルを作成できる例を示しています。

```
ITtdEntitiesPtr pDiagrams;
pDiagrams = pPackage->GetEntities(_T("Diagram"));

long lCount = pDiagrams->Count;

ITtdPresentationElementPtr pDiagram;
// N.B. Index is 1-based
for (long i = 1; i <= lCount; i++){
  pDiagram = pDiagrams->GetItem(i);
  if (pDiagram != 0){
    // A diagram is also an entity...
    ITtdEntityPtr pE = pDiagram; // ... so this is OK!
    pDiagram->GenerateEMFEx(pE->GetValue(_T("Name")));
  }
}
```

GenerateImage

プレゼンテーション要素のグラフィカルな外観に対して特定の種別の画像ファイルを生成します。ツールのエディタで表示した場合、プレゼンテーション要素は画像ファイル内と同じ外観を持ちます。

```
HRESULT GenerateImage(
          [in] ImageKind imgKind,
          [in] BSTR strFileName);
```

パラメータ

`[in] imgKind`

生成する画像ファイルの種別。このパラメータの値として有効なのは以下の表のとおりです:

イメージ種別	説明
IK_JPEG	JPEG 画像ファイルを生成します。
IK_BMP	BMP 画像ファイルを生成します
IK_GIF	GIF 画像ファイルを生成します
IK_TIFF	TIFF 画像ファイルを生成します
IK_TARGA	TGA (Targa) 画像ファイルを生成します
IK_DIB	装置非依存のビットマップファイルを生成します
IK_PCX	PCX 画像ファイルを生成します

`[in] strFileName`

生成する画像ファイルの名前。strFileName が相対パス指定の場合は、クライアントアプリケーションの現在の作業ディレクトリからの相対値になります。

戻り値

返された HRESULT から取得される戻り値は、以下のいずれかです:

戻り値	意味
S_OK	画像ファイルが正しく生成されました。
E_FAIL	エラーが発生しました。たとえば、画像ファイルが指定した場所に保存できなかった、または必要な Tau のライセンスが取得できなかったなどの場合が考えられます。詳細については、COM エラー オブジェクトが提供する情報を確認してください。

コメント

`GenerateImage` メソッドは UML モデルから、プレゼンテーション要素の可視化のためにレポートを生成するようなクライアントで使用されます。このメソッドは、すべてのプレゼンテーション要素上で動作します。またプレゼンテーション要素が他のプレゼンテーション要素を含んでいる場合 (ダイアグラムなどの場合) は、画像ファイルにもこの包含関係が反映されます。

ITtdSymbol

ITtdSymbol インターフェイスは メタモデルのシンボルクラスによって実装されます。あるシンボルは、ダイアグラム内のグラフィカルな外観が二次元の領域を占める表現要素と定義されます。したがって、シンボルにはサイズと位置があります。

注記

シンボルはある Entity の PresentationElement でもあるので、ITtdSymbol から ITtdPresentationElement または ITtdEntity への QueryInterface は常に成功します。

ITtdSymbol は、以下のメソッドを含みます :

`SetSize`	シンボルのサイズを設定します。
`SetPosition`	シンボルも位置を設定します。

SetSize

シンボルのサイズ設定します。シンボルのサイズを変更する際の推奨メソッドです。

```
HRESULT SetSize(
          [in] long width,
          [in] long height);
```

パラメータ

`[in]  width`

シンボルの幅です。単位は、1/10 ミリメートルです。正の値であることが必要です。

`[in]  height`

シンボルの高さです。単位は、1/10 ミリメートルです。正の値であることが必要です。

戻り値

返された HRESULT から取得される戻り値は、以下のいずれかです :

戻り値	意味
S_OK	シンボルのサイズが変更されました。
E_FAIL	エラーが発生しました。詳細については、COM エラー オブジェクトが提供する情報を確認してください。

コメント

大半のシンボルについては、SetSize メソッドは、size メタ特性上の ITtdEntity::SetValue を使用して新しいサイズを設定するために機能します。しかし、いくつかのシンボルについては、サイズが意味的な重要性をもっています。このようなシンボルのサイズ変更は、オブジェクトモデルの変更につながる場合があります。また、1 つのシンボルのサイズ変更が他のシンボルのサイズ変更を引き起こす可能性もあります。このような「シンボルのサイズ変更に付随する効果」を発生し得うるのが、SetSize メソッドです。シンボルのサイズ変更の際にこのメソッドを使用することを推奨するのはこの理由からです。

シンボルのサイズを読み取るには、size メタ特性上の ITtdEntity::GetValue を使用してください。

SetPosition

シンボルの位置を設定します。シンボルの位置を変更する際の推奨メソッドです。

```
HRESULT SetPosition(
          [in] long x,
          [in] long y);
```

パラメータ

`[in]  x`

シンボルの水平方向の位置。単位は、1/10 ミリメートルです。正の値であることが必要です。

`[in]  y`

シンボルの垂直方向の位置。単位は、1/10 ミリメートルです。正の値であることが必要です。

戻り値

返された HRESULT から取得される戻り値は、以下のいずれかです :

戻り値	意味
S_OK	シンボルは位置変更されました。
E_FAIL	エラーが発生しました。詳細については、COM エラー オブジェクトが提供する情報を確認してください。

コメント

大半のシンボルについては、SetSize メソッドは、position メタ特性上の ITtdEntity::SetValue を使用して新しいサイズを設定するために機能します。しかし、いくつかのシンボルについては、位置が意味的な重要性をもっています。このようなシンボルの位置変更は、オブジェクトモデルの変更につながる場合があります。また、1 つのシンボルの移動が他のシンボルの移動を引き起こす可能性もあります。このような「シンボルの移動に付随する効果」を発生し得うるのが、SetSize メソッドです。シンボルの位置変更の際にこのメソッドを使用することを推奨するのはこの理由からです。

シンボルの位置は、左上の隅を原点とする座標で表現されます。

シンボルの位置を読み取るには、position メタ特性上の ITtdEntity::GetValue を使用してください。

ITtdExpression

ITtdExpression インターフェイスはメタモデル の Expression クラスの実装されています。Expression はモデルのさまざまな場所に現れる可能性があります。

注記

Expression は Entity の 1 つなので、ITtdExpression から ITtdEntity への QueryInterface は常に成功します。

ITtdExpression は以下のメソッドを含みます :

GetType	式のタイプを算出します。
EvaluateConstantIntegralExpression	定数式の整数値を評価します。
GetInstanceChildExpression	あるインスタンスの子式を検索します。

GetType

式の型を算出して戻します。型が算出できない場合は (たとえば、式がバインドされない参照を含む場合など) 、NULL が戻されます。

```
HRESULT GetType(
          [out, retval] ITtdEntity** result);
```

パラメータ

`[out, retval]  result`

式のタイプ。

戻り値

返された HRESULT から取得される戻り値は、以下のいずれかです :

戻り値	意味
S_OK	成功。
E_FAIL	内部エラーで失敗しました。

コメント

式の型を検索するには、GetType を使用します。すべての式は型を保持しますが、そのタイプがモデル内で見つかるかどうかは保証されません (たとえば、戻されたエンティティ上で GetModel を実行すると NULL が戻ります)。 これは型が暗示的に定義された場合に起こります。 ポインタを保存していないこと、一時オブジェクトを保存していないことを確認してください。

EvaluateConstantIntegralExpression

式の値を評価します。通常は値は整定数です。

```
HRESULT EvaluateConstantIntegralExpression(
          [out, retval] long* value);
```

パラメータ

`[out, retval]  value`

整数値としての式の評価。

戻り値

返された HRESULT から取得される戻り値は、以下のいずれかです :

戻り値	意味
S_OK	式は評価されました。
E_FAIL	エラーが発生しました。バインドされていない参照があるため式が評価できない状況などが考えられます。詳細については、COM エラー オブジェクトが提供する情報を確認してください。

コメント

式が評価される結果の整定数を検索するには EvaluateConstantIntegralExpression を使用します。このメソッドは、式を使用するが、値のみに興味があり、式がどのように構築されているかには興味がないクライアントには有用です。

GetInstanceChildExpression

あるインスタンスに含まれる割り当て式の右側値を取得します。割り当ての左側値は、特定の識別子です。

```
HRESULT GetInstanceChildExpression(
          [in] BSTR strName,
          [out, retval] ITtdExpression** result);
```

パラメータ

`[in]  strName`

割り当て式の左側値である識別子の名前です。この名前は、通常は修飾子を含む、有効な識別子である必要があります。

`[out, retval]  result`

一致する子式が戻されます。

戻り値

返された HRESULT から取得される戻り値は、以下のいずれかです：

戻り値	意味
S_OK	成功。
E_FAIL	エラーが発生しました。詳細については、COM エラー オブジェクトが提供する情報を確認してください。

コメント

インスタンス式 (たとえばステレオタイプインスタンス) 上で `GetInstanceChildExpression` を使用してそのインスタンスに含まれる割り当て式の右側値を取得します。左側値は、`strName` での指定値に一致する有効な識別子です。

一致する子式が見つからない場合は、NULL が戻されます。

`GetInstanceChildExpression` を使用してタグ付き値をステレオタイプインスタンスから抽出できますが、GetTaggedValue でサポートされるすべての場合をカバーするわけではありません。

ITtdMetaVisitCallback

ITtdMetaVisitCallback インターフェイスは、ITtdEntity::MetaVisit メソッドを使用するクライアントで実装する必要があるコールバック インターフェイスです。

ITtdMetaVisitCallback には、以下のメソッドがあります。

`OnVisitedEntity`	モデルのトラバース時にアクセスされるエンティティごとに呼び出されます。
`OnAfterVisitedEntity`	OnVisitedEntity と同様です。ただし、呼び出しは、エンティティのすべての合成子がアクセスされた後に行われます。

OnVisitedEntity

モデルのトラバース時にエンティティが検出される（アクセスされる）ときに、ITtdEntity::MetaVisit 実装によって呼び出されます。

```
HRESULT OnVisitedEntity(
          [in] ITtdEntity* pVisitedEntity);
```

パラメータ

`[in]  pVisitedEntity`

モデルのトラバース中に現在アクセスされているエンティティ。

戻り値

`OnVisitedEntity` によって返された値は、トラバースの続行方法を判定するために、`MetaVisit` 実装によって使用されます。返された HRESULT は、以下のいずれかになっている必要があります。

戻り値	意味
S_OK	エンティティは正しくアクセスされています。その合成子のトラバースを続行します。
S_FALSE	エンティティは正しくアクセスされていますが、その合成子をトラバースしません。この戻り値を使用して、クライアントにとって意味がないモデルのパートのトラバースをスキップできます。
E_FAIL	モデルのトラバースを中止します。この戻り値は、たとえば、特定のエンティティを探すために MetaVisit が使用されるときに使用できます。pVisitedEntity が結果的に探しているエンティティになる場合は、効果的にトラバースを中止するために、OnVisitedEntity で E_FAIL を返すことができます。
他のいずれかの戻り値	E_FAIL として扱われます。MetaVisit 呼び出しも失敗します（つまり、E_FAIL を返す）。これは、呼び出し側のクライアント コードに反映させる必要がある pVisitedEntity のアクセス中にエラーが発生したときに使用できます。

コメント

OnVisitedEntity コールバック メソッドは、MetaVisit メソッドの呼び出し元のエンティティを含めて、アクセスされたすべてのエンティティ用に呼び出されます。クライアントでは、アクセスされているエンティティに対して必要な操作を行い、次に上記の表に従って戻り値を返して、モデルのトラバースの続行方法を指定する必要があります。

1848 ページの例 650 には、モデルのメタモデル駆動型トラバースを実装するための OnVisitedEntity の使用法を示す例が含まれています。

OnAfterVisitedEntity

Called by the モデルのトラバース中にあるエンティティに出会った場合に ITtdEntity::MetaVisit の実装に呼ばれます。エンティティのすべての合成子がアクセスされてから呼び出しが行われます。

```
HRESULT OnAfterVisitedEntity(
         [in] ITtdEntity* pVisitedEntity);
```

パラメータ

[in] pVisitedEntity

モデルのトラバース中に現在アクセスされているエンティティ。

戻り値

OnAfterVisitedEntity によって返された値は、トラバースの続行方法を判定するために、MetaVisit 実装によって使用されます。返された HRESULT は、以下のいずれかになっている必要があります。

戻り値	意味
S_OK	エンティティは正しくアクセスされています。トラバースは続行できます。
E_FAIL	モデルのトラバースを中止します。この戻り値は、たとえば、特定のエンティティを探すために MetaVisit が使用されるときに使用できます。pVisitedEntity が結果的に探しているエンティティになる場合は、効果的にトラバースを中止するために、OnAfterVisitedEntity で E_FAIL を返すことができます。
any other return value	E_FAIL として扱われます。MetaVisit 呼び出しも失敗します（つまり、E_FAIL を返す）。これは、呼び出し側のクライアント コードに反映させる必要がある pVisitedEntity のアクセス中にエラーが発生したときに使用できます。

コメント

OnAfterVisitedEntity コールバック メソッドは、MetaVisit メソッドの呼び出し元のエンティティを含めて、アクセスされたすべてのエンティティ用に呼び出されます。この呼び出しは、エンティティのすべての直接的または間接的合成子をすでにアクセスしたときに発生し、モデル トラバースの後方向再帰の手段となります。クライアントでは、アクセスされているエンティティに対して必要な操作を行い、次に上記の表に従って戻り値を返して、モデル トラバースの続行方法を指定する必要があります。1848 ページの例 650 に、メタモデルドリブンのモデル トラバースを実装するための OnAfterVisitedEntity の使用方法の例があります。

ITtdInteractiveClient

ITtdInteractiveClient インターフェイスは、COM API のすべての対話型 クライアントで実装する必要があるインターフェイスです。対話型 クライアントは、Tau アプリケーションによって起動され、そのアプリケーションにロードされたモデルにアクセスできるクライアントです。ITtdInteractiveClient は、Tau でクライアントと通信するためのインターフェイスを定義します。

ITtdInteractiveClient には、以下のメソッドがあります。

OnExecute	クライアントで実行を開始する必要があるときに Tau によって呼び出されます。

OnExecute

このメソッドは、Tau での実行準備が整ったときにクライアント上で呼び出されます。

```
HRESULT OnExecute(
         [in] ITtdInteractiveServer* pServer,
         [in] ITtdEntities* pEntities);
```

パラメータ

`[in]  pServer`

ITtdInteractiveServer インターフェイスを実装するサーバ オブジェクトのポインタ。クライアントでは、そのインターフェイスを通じて、サーバ、Tau アプリケーションと通信できます。

`[in]  pEntities`

サーバによってクライアントに提供されるエンティティのコレクション。これらのエンティティは、クライアントで作業を開始するためのコンテキストを構成します。

戻り値

`OnExecute` によって返される HRESULT 値は、以下のいずれかになっている必要があります。

戻り値	意味
`S_OK`	クライアントは正しく実行完了しました。
`E_FAIL`	実行中にクライアントでエラーが発生しました。

コメント

`OnExecute` メソッドは、対話型 クライアントが実行できるように Tau によって同期的に呼び出されます。クライアントでは、このメソッドの実装でそのすべてのアクションを実行する必要があります。

Tau を通じて、クライアント上でメソッドを呼び出すには、`ExecuteCOMClient` という Tcl コマンドを使用する必要があります。この Tcl コマンドは、たとえば、Tau アドイン モジュール用に指定される Tcl スクリプトの一部にできます。`OnExecute` メソッドが失敗した場合は、Tcl コマンドも失敗します。

Tau がクライアント上のメソッドを呼び出すためには、Tcl コマンド std::ExecuteCOMClient が使用されます：

この Tcl コマンドは、たとえば、Tau アドインモジュールで指定した Tcl スクリプトの一部などです。`OnExecute` メソッドが失敗した場合、Tcl コマンドも失敗します。

注記

ITtdInteractiveClient インターフェイスは、ITtdAgent インターフェイスの導入以前に開発されたクライアントとの後方互換性のためのみ提供されます。新たに開発するクライアントでは、すべての API（C++、COM、Tcl）の起動の可能性および実パラメータを渡すことができるという利点を考慮して、ITtdAgent インターフェイスを使用することをお勧めします。詳細は、エージェント を参照してください。

ITtdInteractiveServer

ITtdInteractiveServer インターフェイスは、対話型 クライアントを起動する Tau モジュールによって実装されます。クライアントがその実行中に Tau と通信するためのインターフェイスを定義します。

注記

ITtdModelAccess インターフェイスを ITtdInteractiveServer インターフェイスから取得できます。これは、そのインターフェイスによって提供されたメソッドを利用する必要がある対話型 COM クライアントにとって便利な場合があります。

ITtdInteractiveServer には、以下のメソッドがあります。

`:CreateEntityCollection`	エンティティの空のコレクションを作成します。
`InterpretTclScript`	サーバ上の Tcl スクリプトを解釈します。

:CreateEntityCollection

エンティティの空のコレクションを作成して、呼び出し側に戻します。

```
HRESULT CreateEntityCollection(
          [out, retval] ITtdEntities** pEntityCollection);
```

パラメータ

`[out, retval]  pEntityCollection`

作成されたエンティティの空のコレクション。

戻り値

返された HRESULT から取得される戻り値は、以下のいずれかです。

戻り値	意味
`S_OK`	エンティティのコレクションは正しく作成されています。
`E_OUTOFMEMORY`	メモリ不足により新しいコレクションを作成できませんでした。

コメント

`CreateEntityCollection` は、エンティティのコンテナを取得するためにクライアントで使用できるヘルパー メソッドです。クライアントではこのコンテナを内部的に使用できます。または、引数として ITtdEntities コレクションが必要な ITtdInteractiveServer インターフェイスの他のメソッドを呼び出すために使用できます。

作成されたコレクションは、クライアントで削除することはできません。コレクションはその参照の所有者が存在しなくなると、それ自体を削除します。

1881 ページの例 657 には、`CreateEntityCollection` の使用法の例が含まれています。

InterpretTclScript

サーバ上の Tcl スクリプトを解釈して、結果を呼び出し側に返します。このメソッドは、対話型 クライアントがサーバと通信するための手段となります。

```
HRESULT InterpretTclScript(
          [in] BSTR strScript,
          [in, optional] VARIANT pEntities,
          [out, retval] BSTR* strResult);
```

パラメータ

`[in]  strScript`

解釈する Tcl スクリプトが含まれている文字列。文字列には、`pEntities` コレクション内の対応するエンティティの Tcl id によって置き換えられる " 置換マーカー " が含まれている場合があります。

`[in, optional]  pEntities`

`strScript` 内の Tcl スクリプトの引数であるエンティティの ITtdEntities コレクション。これはオプションのパラメータです。指定する必要があるのは、Tcl スクリプトに湯 u 換マーカー狽 ™ 含まれる場合のみです。

`[out, retval]  strResult`

Tcl スクリプトの解釈の結果。

戻り値

返された HRESULT から取得される戻り値は、以下のいずれかです。

戻り値	意味
S_OK	スクリプトは正しく解釈されています。
E_FAIL	引数の置換またはスクリプトの解釈中にエラーが発生しました。詳細については、COM エラー オブジェクトが提供する情報を確認してください。

コメント

現在、InterpretTclScript は、対話型 クライアントがその実行中に Tau と通信するための唯一の手段となっています。COM メソッドの呼び出しではなく Tcl スクリプトを使用するのは多少不便に感じられるかもしれませんが、この方法では、Tau によってサポートされている Tcl API 全体へのクライアント アクセスを行うことができます。

Tcl スクリプトには、#n の形式で " 置換マーカー " を含めることができます。ここでの n は、エンティティ コレクション内のエンティティのインデックスです。サーバによってスクリプトが事前処理され、指定されたエンティティ コレクション内の対応するエンティティの Tcl id ですべてのマーカーが置き換えられます。通常、インデックスは 1 から開始されます。

例 657

以下の例は、InterpretTclScript メソッドを使用して、どのように対話型 クライアントからサーバに通信できるかを示しています。以下のコードは、クライアントの OnExecute メソッドの一部です。また、pServer は、クライアントがそのメソッドで引数として取得する ITtdInteractiveServer インターフェイス ポインタです。

```
// First create an empty collection of entities that are arguments
// to the Tcl script.
ITtdEntitiesPtr pEntities = pServer->CreateEntityCollection();

// Add an entity to the collection
pEntities->Add(pEntity);

// Specify the Tcl script using a substitution marker for the first
// entity of the collection
CComBSTR strScript(_T("output #1"));

// Interpret the Tcl script and save the result in strResult
CComBSTR strResult((BSTR) pServer->InterpretTclScript((BSTR)
strScript, pEntities));
```

ITtdSourceBuffer

ITtdSourceBuffer インターフェイスは、主にコード生成時に、生成されたファイルの表現として使用されるテキスト（通常はソース コード）のバッファを表します。たとえば、生成されたファイルに追加テキストを加える C++ アプリケーション ジェネレータのエージェントが使用できます。

ITtdSourceBuffer には、以下のメソッドがあります。

AddText	テキストをソース バッファに追加します。

AddText

テキストをソース バッファに追加します。

```
HRESULT AddText(
        [in] BSTR text);
```

パラメータ

`[in]  text`

ソース バッファに追加するテキスト文字列。

戻り値

返された HRESULT から取得される戻り値は、以下のいずれかです。

戻り値	意味
S_OK	テキストは正しくソース バッファに追加されています。
E_FAIL	エラーが発生しました。詳細については、COM エラー オブジェクトが提供する情報を確認してください。

コメント

AddText メソッドでは、テキストをソース バッファに書き込みます。ソース バッファは順次に書き込まれます。また、テキストはソース バッファ内の現在の位置に追加されます。

通常、このメソッドは、コード ジェネレータ エージェントで追加テキストを生成済みファイルに書き込む必要があるときに使用されます。ファイルは、コード ジェネレータによって入力されるソース バッファとして表されます。エージェントでは、特定のエンティティがバッファに出力されるときにコード ジェネレータによって送信される特定のツール イベントをリスンできます。したがって、追加テキストは、そのようなエンティティの出力の直前または直後にソース バッファに追加できます。

ITtdMessageList

ITtdMessageList インターフェイスは、一連のメッセージを表します。そのようなリストは、たとえば、セマンティック分析またはコード生成からのエラーを報告するときに使用されます。インターフェイスは、エージェントが、たとえば、カスタム セマンティック チェックに基づくカスタム メッセージを追加するときに使用できます。

ITtdMessageList には、以下のメソッドがあります。

AddMessage	新しいメッセージをメッセージ リストに追加します。

AddMessage

新しいメッセージをメッセージ リストの終わりに追加します。

```
HRESULT AddMessage(
          [in] BSTR text,
          [in] MessageSeverity severity,
          [in, optional] VARIANT subject);
```

パラメータ

`[in]  text`

追加するメッセージのテキスト。

`[in]  severity`

メッセージの重要度。以下のリテラルを使用できます。

- MS_INFORMATION
  情報メッセージにこのリテラルを使用します。
- MS_WARNING
  警告メッセージにこのリテラルを使用します。
- MS_ERROR
  エラー メッセージにこのリテラルを使用します。
- MS_FATAL
  重大なエラー メッセージ（回復不能なエラー）にこのリテラルを使用します。

`[in, optional]  subject`

このパラメータは、メッセージが付加されるオプションのサブジェクト エンティティです。サブジェクト エンティティは、Tau IDE でメッセージが特定（ダブルクリック）されるときに特定されます。

戻り値

返された HRESULT から取得される戻り値は、以下のいずれかです。

戻り値	意味
S_OK	メッセージは正しくメッセージ リストに追加されています。
E_FAIL	エラーが発生しました。詳細については、COM エラー オブジェクトが提供する情報を確認してください。

コメント

AddMessage メソッドでは、新しいメッセージをメッセージ リストの終わりに追加します。

通常、このメソッドが使用されるのは、コード ジェネレータまたはセマンティック チェッカ エージェントで［チェック］または［ビルド］タブにメッセージを報告する必要があるときです。エージェントはその後に、コード ジェネレータまたはセマンティック チェッカ（エージェント登録が行われたツール イベントをトリガする）から ITtdMessageList インターフェイスを受け取ります。

ITtdAgent

ITtdAgent インターフェイスは、COM API でエージェントを実装するときに使用されます。エージェントの実装である COM オブジェクトでは、ITtdAgent インターフェイスを実装する必要があります。

ITtdAgent には、以下のメソッドがあります。

Execute	エージェントの起動時に Tau によって呼び出されます。

Execute

このメソッドは、Tau での実行準備が整ったときにエージェント上で呼び出されます。

```
HRESULT Execute(
          [in] ITtdEntity* triggeredBy,
          [in] VARIANT beforeProcessing,
          [in] ITtdEntity* modelContext,
          [in] IUnknown* server,
          [in, out] SAFEARRAY(VARIANT)* agentParameters);
```

パラメータ

```
[in]   triggeredBy
```

エージェントをトリガした操作に関する UML 定義のポインタ。操作はツール イベントまたはエージェント定義です。エージェントでは、2 つ以上のツール イベントまたはエージェントによる起動が可能な場合に、この情報が必要になることがあります。エージェントがプログラムで呼び出される場合（InvokeAgent を使用して）、このパラメータは NULL になります。

`[in] beforeProcessing`

このパラメータが true になるのは、<<before processing>> ステレオタイプによってステレオタイプ化された依存関係の設定を通じてエージェントが起動される場合です。それ以外の場合は、false になります。エージェントでは、2 つ以上の依存関係の設定を通じてトリガが可能な場合に、この情報が必要になることがあります。エージェントがプログラムで起動される場合は（InvokeAgent を使用して）、このパラメータを適用できません。

`[in] modelContext`

エージェント起動のモデル コンテキストに相当するエンティティ。

`[in] server`

エージェントが、起動した Tau アプリケーションと通信するためのサーバ オブジェクトのポインタ。対話型 エージェントは、ITtdInteractiveServer インターフェイスを実装するサーバ オブジェクトを受け取ります。非対話型 エージェントの場合は、サーバ オブジェクトで、その特定のアプリケーションに関連したサービスを提供する別のインターフェイスを実現できます。C++ アプリケーション ジェネレータ実行形式ファイルで実行されるエージェント用のサービスを提供する ITtdCppAppGenServer などを参照してください。

`[in, out] agentParameters`

このパラメータは、エージェントの起動プロセスをトリガした（直接的または間接的）ツール イベント によって実行されるエージェント パラメータのリスト（安全な配列）です。渡されるパラメータはツール イベントによって異なります。起動に必要な依存関係の設定によって異なる場合もあります。

エージェントがプログラムで起動される場合（InvokeAgent を使用して）、このリストには呼び出し側からエージェントに渡された実際の引数が含まれます。

エージェントでは、情報を呼び出し側に戻すために、パラメータ リストを変更できます。

戻り値

`Execute` によって返される HRESULT 値は、以下のいずれかになっている必要があります。

戻り値	意味
`S_OK`	エージェントは正しく実行完了しました。
`E_FAIL`	実行中にエージェントでエラーが発生しました。対話型 エージェントの場合は、エラー メッセージが［メッセージ］タブに出力されます。通常、非対話型 エージェントの場合は、エラー メッセージが `stderr` 上に出力されます。

コメント

Execute メソッドは、エージェントが実行できるように Tau によって同期的に呼び出されます。エージェントでは、このメソッドの実装でそのすべてのアクションを実行する必要があります。

ITtdCppAppGenServer

ITtdCppAppGen インターフェイスは、C++ アプリケーション ジェネレータによって実装されます。COM オブジェクトとして実装されたエージェントを使用して、C++ コード生成をカスタマイズするときに使用されます。エージェントでは、その Execute メソッドの呼び出しで、server パラメータからこのインターフェイスを取得できます。

ITtdCppAppGenServer には、以下のメソッドがあります。

ScheduleForDeletion	このメソッドを使用して、モデル内のエンティティを削除します。

ScheduleForDeletion

エージェントでエンティティを削除する場合は、このメソッドを使用します。C++ アプリケーション ジェネレータによって、モデルから削除するエンティティのリンクがすぐに解除されるため、その後はコード ジェネレータから見えない状態になります。ただし、エンティティはコード ジェネレータで妥当と判断されるまでは実際に削除されません。

```
HRESULT ScheduleForDeletion(
          [in] ITtdEntity* entity);
```

パラメータ

[in] entity

削除するエンティティ。

戻り値

ScheduleForDeletion によって返される HRESULT 値は、以下のいずれかです。

戻り値	意味
S_OK	エンティティは削除のスケジュールが正しく設定されました。
E_FAIL	エラーが発生しました。詳細については、COM エラー オブジェクトが提供する情報を確認してください。

コメント

エージェントでは、Delete メソッドではなく、ScheduleForDeletion を使用することが重要です。後者のメソッドでは、エンティティがすぐに削除されるため、C++ アプリケーション ジェネレータや他のアクティブなカスタマイズ エージェントで問題が発生する可能性があります。たとえば、エージェントなどでは、削除されたエンティティのポインタを格納していて無効になる場合があります。

ITtdStudioAccess

ITtdStudioAccess インターフェイスは、TTD_StudioAccess COM クラスのデフォルトインターフェイスであり、インスタンス作成時に取得されます。UML モデリングだけではない Tau IDE の機能にアクセスするためのメインのエントリポイントです。

ITtdStudioAccess は以下のメソッドを含んでいます

OpenWorkspace	ワークスペース (.ttw ファイル) を開きます
NewWorkspace	新しいワークスペースを作成します。
OpenProject	プロジェクト (.ttp ファイル) を開きます。
GetWorkspace	現在ロードされているワークスペースへの参照を取得します。
InterpretTclScript	Tcl スクリプトを解釈します。
GetApplicationName	実行中の Tau の完全名を取得します。
GetApplicationPID	実行中の Tau の PID (プロセス ID) を取得します。
GetApplicationVersion	実行中の Tau のバージョンを取得します。
GetApplicationUserName	実行中の Tau のユーザー名を取得します。

OpenWorkspace

Tau ワークスペースファイル (拡張子 `ttw`) を開きます。ワークスペースに含まれるすべてのプロジェクトがロードされます。

```
HRESULT OpenWorkspace(
          [in] BSTR strPath,
          [out, retval] ITtdWorkspace** workspace);
```

パラメータ

`[in]  strPath`

ロードするワークスペースを指定する文字列。指定された文字列が相対パス指定の場合は、現在の作業ディレクトリに変換されます (通常は Tau の bin ディレクトリ)。URN 参照を含む場合もあります。

`[out, retval]  workspace`

ロードされたワークスペースへのポインタ。

戻り値

返された HRESULT から取得される戻り値は、以下のいずれかです :

戻り値	意味
S_OK	ワークスペースが開かれました。
E_FAIL	ワークスペースを開けませんでした。詳細については、COM エラー オブジェクトが提供する情報を確認してください。

コメント

`OpenWorkspace` メソッドは、ワークスペースファイルを読んで、ワークスペースを開きます。このメソッドの機能は、メニューからの［ファイル］>［ワークスペースを開く］と同等です。

NewWorkspace

新規の Tau ワークスペースを作成し、ワークスペースファイル (拡張子 `ttw`) と関連付けます。

```
HRESULT NewWorkspace(
          [in] BSTR strPath,
          [out, retval] ITtdWorkspace** workspace);
```

パラメータ

`[in]  strPath`

ワークスペースのワークスペースファイルをどこに保存するかを指定する文字列。指定された文字列が相対パス指定の場合は、現在の作業ディレクトリに変換されます (通常は Tau の bin ディレクトリ)。

`[out, retval]  workspace`

新しいワークスペースへのポインタ。

戻り値

返された HRESULT から取得される戻り値は、以下のいずれかです :

戻り値	意味
S_OK	ワークスペースが作成されました。
E_FAIL	ワークスペースが作成できませんでした。詳細については、COM エラー オブジェクトが提供する情報を確認してください。

コメント

NewWorkspace メソッドはワークスペースを新規に作成し、ワークスペースファイル (.ttw) に保存します。すでに開いているワークスペースがある場合は、いったん閉じます。このメソッドの機能は、メニューからの［ファイル］>［新規］と同等です。

OpenProject

Tau プロジェクトファイル (拡張子 ttp) を開きます。プロジェクトに関連付けられたモデルがロードされます。

```
HRESULT OpenProject(
          [in] BSTR strPath,
          [out, retval] ITtdProject** project);
```

パラメータ

`[in]  strPath`

ロードするプロジェクトファイルを指定する文字列。指定された文字列が相対パス指定の場合は、現在の作業ディレクトリに変換されます (通常は Tau の bin ディレクトリ)。URN 参照を含む場合もあります。

`[out, retval]  project`

ロードされたプロジェクトへのポインタ。

戻り値

返された HRESULT から取得される戻り値は、以下のいずれかです :

戻り値	意味
S_OK	プロジェクトがロードされました。
E_FAIL	プロジェクトがロードできませんでした。詳細については、COM エラー オブジェクトが提供する情報を確認してください。

コメント

`OpenProject` メソッドは Tau プロジェクトファイルを読み込んで既存のプロジェクトを開きます。このメソッドの機能は、メニューからの［ファイル］>［オープン］と同等です。

開かれたプロジェクトは現在のワークスペースに挿入されます。ワークスペースがない場合は、メソッドは失敗します。ワークスペースが使用可能かどうかは GetWorkspace メソッドで確認してください。

注記

本メソッドで開かれたプロジェクトは、同時にアクティブにもなります。

GetWorkspace

現在のワークスペースを戻します。

```
HRESULT GetWorkspace(
          [out, retval] ITtdWorkspace** workspace);
```

パラメータ

`[out, retval]  workspace`

現在のワークスペースへのポインタ、または 、ワークスペースがない場合は NULL。

戻り値

返された HRESULT から取得される戻り値は、以下のいずれかです :

戻り値	意味
`S_OK`	ワークスペースが取得できました。
`E_FAIL`	ワークスペースが取得できませんでした。詳細については、COM エラー オブジェクトが提供する情報を確認してください。

コメント

`GetWorkspace` メソッドは、現在開いているワークスペースを戻します。また、現在利用可能なワークスペースがあるかどうかの判定にも使用できます。

InterpretTclScript

TCL スクリプトを解釈します。

```
HRESULT OpenWorkspace(
         [in] BSTR strScript,
         [out, retval] BSTR* result);
```

パラメータ

`[in]  strScript`

解釈される TCL スクリプトを含む文字列。

`[out, retval]  result`

TCL スクリプト解釈の結果。

戻り値

返された HRESULT から取得される戻り値は、以下のいずれかです :

戻り値	意味
S_OK	スクリプトは解釈されました。
E_FAIL	スクリプトの解釈中にエラーが発生しました。詳細については、COM エラー オブジェクトが提供する情報を確認してください。

コメント

`InterpretTclScript` メソッドによって、Tau は任意の TCL スクリプトを解釈可能になります。スクリプト内のどの TCL コマンドが利用可能になるかは、Tau に何がロードされているかに依存します。一般原則としては、`std` 接頭辞の TCL スクリプトは常に利用可能ですが、`u2` 接頭辞のものは UML モデルがロードされている場合のみ利用可能になります。

GetApplicationName

Tau の製品名を戻します。

```
HRESULT GetApplicationName(
         [out, retval] BSTR* name);
```

パラメータ

`[out, retval]  name`

アプリケーション製品名。

戻り値

メソッドは常に成功します。(つまり S_OK が戻ります)

コメント

`GetApplicationName` メソッドは実行中の Tau の製品名を文字列として戻します。Tau の機能は製品ごとに異なるので、このメソッドはクライアントがどの Tau 機能が使えるかを判断する際に有用です。

GetApplicationPID

Tau の PID(プロセス id) が戻ります。

```
HRESULT GetApplicationPID(
          [out, retval] BSTR* pid);
```

パラメータ

`[out, retval]  pid`

Tau の PID(プロセス id)。

戻り値

メソッドは常に成功します。(つまり S_OK が戻ります)

コメント

`GetApplicationPID` メソッドは実行中の Tau の PID (プロセス id) を戻します。このメソッドは、1 つのマシン上に複数の Tau が実行されていて特定の Tau と通信したい場合などに有用です。PID は異なる Tau インスタンスを見分ける手段となります。

GetApplicationVersion

Tau のバージョンを戻します。

```
HRESULT GetApplicationVersion(
          [out, retval] BSTR* version);
```

パラメータ

`[out, retval]  version`

Tau のバージョン。

戻り値

メソッドは常に成功します。(つまり S_OK が戻ります)

コメント

`GetApplicationVersion` メソッドは Tau のバージョン番号を文字列で戻します。Tau の機能はバージョンごとに異なるので、このメソッドはクライアントがどの Tau 機能が使えるかを判断する際に有用です。

GetApplicationUserName

実行中の Tau のユーザー名を戻します。

```
HRESULT GetApplicationUserName(
          [out, retval] BSTR* name);
```

パラメータ

`[out, retval]` `name`

Tau を実行しているユーザーのユーザー名。

戻り値

メソッドは常に成功します。(つまり S_OK が戻ります

コメント

`GetApplicationUserName` メソッドは Tau を実行しているユーザーのユーザー名を戻します。このメソッドは 1 つのマシンで複数の Tau インスタンスが実行中で、ユーザーも複数いる場合にユーザーを見分けるのに有用です。このメソッドを使用するとある Tau が自分と同じユーザーで稼動しているか別のユーザーで稼動しているかを判別できます。

ITtdWorkspace

ITtdWorkspace インターフェイスは Tau ワークスペースを表現し、ワークスペースを操作するメソッドが含まれます。

ITtdWorkspace には以下のメソッドが含まれます :

GetPath	ワークスペースのパスを取得します。
GetProject	ワークスペースに含まれるプロジェクトを取得します。
GetActiveProject	ワークスペースに含まれるアクティブプロジェクトを取得します。
SetActiveProject	ワークスペース内のプロジェクトをアクティブに設定します。

GetPath

ワークスペースが格納されているワークスペースファイルのパスを戻します。

```
HRESULT GetPath(
          [out, retval] BSTR* path);
```

パラメータ

`[out, retval]  path`

ワークスペースパス。

戻り値

メソッドは常に成功します。(つまり S_OK が戻ります)

コメント

`GetPath` メソッドはワークスペースファイルのフルパスを戻します。

GetProject

このワークスペースに含まれるプロジェクトを指定したパスで検索します。

```
HRESULT GetProject(
          [in] BSTR strPath,
          [out, retval] ITtdProject** project);
```

パラメータ

`[in]  strPath`

検索するプロジェクトファイルパス。

`[out, retval]  project`

プロジェクト。

戻り値

返された HRESULT から取得される戻り値は、以下のいずれかです：

戻り値	意味
S_OK	検索条件に一致するプロジェクトが見つかりました。
E_FAIL	検索条件に一致するプロジェクトが見つかりませんでした。詳細については、COM エラー オブジェクトが提供する情報を確認してください。

コメント

GetProject は、指定したパスにあるプロジェクトファイルに含まれるプロジェクトをワークスペースから検索します。プロジェクトファイルパスの比較は、strPath に対して行われることに注意してください。パスの正規化や URN 拡張は行われません。

GetActiveProject

ワークスペースで現在アクティブになっているプロジェクトを戻します。

```
HRESULT GetActiveProject(
          [out, retval] ITtdProject** project);
```

パラメータ

```
[out, retval]   project
```

アクティブプロジェクト。

戻り値

返された HRESULT から取得される戻り値は、以下のいずれかです：

戻り値	意味
S_OK	成功しました。
E_FAIL	アクティブなプロジェクトが見つかりません。詳細については、COM エラー オブジェクトが提供する情報を確認してください。

コメント

1 つの Tau ワークスペースには複数のプロジェクトがあり得ますが、アクティブなのはそのうちの 1 つだけです。`GetActiveProject` はアクティブなプロジェクトへの参照を戻します。

SetActiveProject

プロジェクトをワークスペースでアクティブに設定します。

```
HRESULT SetActiveProject(
          [in] ITtdProject* project);
```

パラメータ

`[in]  project`

ワークスペースでアクティブに設定されるプロジェクト。

戻り値

返された HRESULT から取得される戻り値は、以下のいずれかです :

戻り値	意味
`S_OK`	プロジェクトはアクティブになりました。
`E_FAIL`	プロジェクトをアクティブにできませんでした。詳細については、COM エラー オブジェクトが提供する情報を確認してください。

コメント

`SetActiveProject` は、プロジェクトをワークスペースでアクティブに設定します。ワークスペースでアクティブにできるのは 1 つのプロジェクトだけです。したがってこのメソッド呼び出しの後には、以前アクティブだったプロジェクトはアクティブでなくなります。

ITtdProject

ITtdProject インターフェイスは Tau プロジェクトを表現します。そのプロジェクト上で動作可能なメソッドを含みます。

ITtdProject は以下のメソッドを含んでいます :

GetPath	プロジェクトのパス。
GetName	プロジェクトの名前。
GetModel	プロジェクトの UML モデル。

GetPath

プロジェクトが格納されているプロジェクトファイルのフルパス名を戻します。

```
HRESULT GetPath(
          [out, retval] BSTR* path);
```

パラメータ

```
[out, retval]    path
```

プロジェクトパス。

戻り値

メソッドは常に成功します。(つまり S_OK が戻ります

コメント

GetPath メソッドはプロジェクトのフルパス名を戻します。通常は、プロジェクトファイルのパス名です。

GetName

プロジェクト名を戻します。

```
HRESULT GetName(
          [out, retval] BSTR* name);
```

パラメータ

```
[out, retval]    name
```

プロジェクト名。

戻り値

メソッドは常に成功します。(つまり S_OK が戻ります)

コメント

GetName メソッドはプロジェクト名を戻します。これは、通常は、パスのないプロジェクトファイル名と一致します。

GetModel

このプロジェクトの UML モデルを戻します。

```
HRESULT GetModel(
          [out, retval] IDispatch** model);
```

パラメータ

[out, retval]　model

プロジェクトのモデルです。

戻り値

返された HRESULT から取得される戻り値は、以下のいずれかです:

戻り値	意味
S_OK	プロジェクトのモデルが取得できました。
E_FAIL	プロジェクトのモデルが取得できませんでした。詳細については、COM エラー オブジェクトが提供する情報を確認してください。

コメント

GetModel はプロジェクトのモデルへの参照を戻します。このメソッドは TTD_StudioAccess COM クラスと TTD_ModelAccess COM クラスの間のブリッジとして動作します。これらの COM クラスのタイプライブラリは分かれており、model パラメータは IDispatch で型付けされます。しかし、戻ったポインタ上で QueryInterface を使用すると、ITtdModel ポインタを取得できます。

59

Tcl API

この章は、Tau Tcl API のリファレンス ガイドです。使用可能な Tcl コマンドと、その使用法について説明します。

対象読者は、Tcl API を使用して UML モデルにアクセスしたり、Tau にダイアログとメニューを追加するクライアント アプリケーションの開発者です。クライアント アプリケーションには、小規模の対話処理用のアドインから本格的なコード ジェネレータ、さらにはインポート アプリケーションに至るまでのあらゆるアプリケーションが含まれます。この章全体を通して、Tcl の基本知識が前提になります。

概要

Tcl API は、以下のグループのコマンドに分類されます。

- 汎用コマンドには、現在の選択へのアクセス、ドキュメントのロード、レポートペインへの出力の生成などの機能があります。
- ユーザー インターフェイス アドイン固有コマンドでは、メニュー、ダイアログ、ツールバーを追加したり、起動時に実行されるアクションを定義したりできます。ユーザー インターフェイス コマンドは、アドインを介して実行される Tcl スクリプトのみから使用できます。
- モデル コマンドでは、現在ロードされている UML モデルに適用可能なさまざまな操作を実行できます。
- エンティティ コマンドでは、現在ロードされている UML モデルの詳細への読み込み／書き込みアクセスを実行できます。したがって、これらのコマンドでは、対話型でモデルを変更するスクリプトを作成できます。
- リソース コマンドには、モデルを物理的な格納ユニット（最も一般的にはファイル システム内のファイル）に保存するために必要なコマンドが含まれています。
- プレゼンテーション要素コマンドには、画像形式でダイアグラムなどのプレゼンテーション要素表現を作成できる機能があります。
- ライブラリ ハンドリング コマンドには、UML ライブラリのロード機能とアンロード機能があります。
- セマンティック チェッカ コマンドは、セマンティック チェッカに対してユーザー定義セマンティック チェックを作成したり追加したりするための手段を提供します。
- ユーティリティインターフェイスコマンド は、公開 Tau API のインターフェイスに応じて、種々の機能を提供します。他のコマンドグループではカバーされない機能も含まれています。

Tcl コマンドへの COM のマッピング

COM API と Tcl API は、以下のセクションのように多くの共通点があります。

- モデル コマンド
- エンティティ コマンド
- リソース コマンド
- プレゼンテーション要素コマンド

一般に、Tcl コマンドの名前は、COM メソッドと同じです。ただし、‘u2’ Tcl 名前空間の一部であることを示す接頭辞が付きます。必須の追加引数とオプションの引数の処理の違いを除いて、コマンド引数も非常に類似しています。追加引数は、モデル コマンドを呼び出すときに常に最初に指定する必要があります。この引数は、COM メソッドの動作対象になるオブジェクトに対応します。つまり、本セクションのすべてのコマンドにおいて、最初の引数はモデルへの参照になります。オプション COM パラメータについては、以下の構文に従ったオプションとして指定されます。

-parameter *value*

ここでの *parameter* は、COM パラメータの名前を表します。*value* は、パラメータ値です。文字列、整数、ブール値といった値の種類は、パラメータ名の最初の文字によって示される場合もあります。値の名称そのものから明らかな場合もあります。

ブール値に ”true” を 設定する場合には、値を設定せずにオプションを指定するだけで十分です。以下の 2 つの例は、同一と見なされます。

```
-bOverwrite true
-bOverwrite
```

引数とオプションの詳細については、それぞれの COM コマンド ドキュメントと比較参照してください。

COM と比較した場合、エラー処理が Tcl では異なります。COM の場合は、すべてのメソッドによってエラー値が返されますが、Tcl コマンドは、エラーが検出されると例外を投げます。したがって、COM API ドキュメントで説明している戻り値は、対応する Tcl コマンドに当てはまらないことになります。代わりに、Tcl コマンドの戻り値は、COM API の ‘retval’ のような種類のパラメータとして示されます。

例 **658:** ———————————————————————————————

通常、Tcl コマンドへの COM メソッドのマッピングは単純明快です。たとえば、COM API ドキュメントで以下のように定義されている FindByGuid コマンドがあります。

```
HRESULT FindByGuid(
         [in] BSTR strGuid,
         [out, retval] ITtdEntity** ppEntity);
```

対応する Tcl コマンドは、以下のとおりです。

u2::FindByGuid *modelRef guid*

このコマンドはモデルの参照を返し、2 つのパラメータ、1 つのモデル参照、*modelRef*、GUID が含まれている文字列、*guid* を取ります。以下のように呼び出せます。

```
set myObj [u2::FindByGuid $model "@Predefined@integer"]
```

実行時のエラーが正しく処理されるように、以下のように呼び出しを catch 句に入れることもできます。

```
if [catch {
 set myObj [u2::FindByGuid $model "@Predefined@integer"]
} ret] {
 std::Output "Error during execution of Tcl script¥n$ret¥n"
}
```

———————————————————————————————————————

汎用コマンド

汎用パートには、以下のコマンドが含まれます。

コマンド	説明
std::BrowserReport	<childobject> という新しいオブジェクトをツリーに追加します。
std::BrowserReportInit	このコマンドは、Browser ビュー内のツリー タブを表示します。必要に応じてタブを作成します。
std::Button	
std::ComboBox	
std::Dialog	ダイアログボックスを作成し、直ちに表示します。
std::DirectoryDialog	［directory selection］ダイアログを表示します。
std::ExecuteCOMClient	対話型 COM クライアントを起動します。
std::FileOpenDialog	［file selection］ダイアログを表示します。
std::FileSaveDialog	［file save］ダイアログを表示します。
std::Frame show-window	Tau フレームウィンドウの表示／非表示を切り替えます。
std::GetActiveProject	現在のワークスペース内のアクティブなプロジェクトを取得します。
std::GetInstallationDirectory	Tau インストール ディレクトリのパスを取得します。
std::GetKind	Get
std::GetModels	ロードされたモデルのリストを取得します。
std::GetProject	プロジェクトのリストを取得します。
std::GetProjectPath	このコマンドは、プロジェクトの絶対パスを返します。
std::GetSelection	現在選択されているエンティティのリストを取得します。
std::GetUserAddinsDirectory	ユーザーアドインディレクトリのパスを取得します。
std::GetUserDirectory	ユーザー情報ディレクトリのパスを取得します。
std::HtmlReport	`html` ファイルを開きます。
std::IsModified	プロジェクトが変更されているかどうかをチェックします。

std::Label	
std::Locate	このコマンドは、<locatestring> によって記述されたオブジェクトを検索します。この文字列は、DataServer によって理解されなければなりません。
std::MessageDialog	［message］ダイアログを表示します。
std::OpenDocument	ドキュメントを開きます。
std::Output	メッセージを［Tcl output］タブに出力します。
std::OutputTab	このコマンドは、出力タブの内容を解除するか、出力タブをアクティブにします。
std::Report	このコマンドは、レポート タブに復帰改行を追加するために使用します。
std::Quit	［close window］ボタンと同じ働きをします。
std::ReportInit	このコマンドは、出力ウィンドウに新しいレポート タブを作成するために使用します。
std::SaveAll	ワークスペースまたはプロジェクトを保存します。
std::TextReport	ファイルを開き、カーソルを移動します。
std::View	std::View コマンドにより、ウィンドウのキャンバスを制御できます。

std::BrowserReport

表記

```
std::BrowserReport [-expanded] [-userdata <userdata>] <childobject>
[<parentobject>]
```

説明

<childobject> という新しいオブジェクトをツリーに追加します。

-expanded を指定し、オブジェクトに子を指定した場合は、ノードが展開されます。それ以外の場合は、ノードは折りたたんだ状態になります。

<parentobject> を指定した場合、ノードは <parentobject> に対応するノードの子として作成されます。

-userdata を指定した場合、<userdata> がツリーのこのノードに関連付けられます。

例 **659:**

```
std::BrowserReportInit MyBrowser
std::BrowserReport -expanded [std::GetActiveProject]
```

```
[std::GetSelection]
```

std::BrowserReportInit

表記

```
std::BrowserReportInit <tabname> [<iconfile>] [-keep] [-cb
<filename>]
```

説明

このコマンドは、Browser ビュー内のツリー タブを表示します。必要に応じてタブを作成します。

<iconfile> を指定した場合、タブ イメージとしてアイコンが使用されます。

-keep を指定した場合、タブの内容（ある場合）が保持されますが、それ以外の場合はリセットされます。

-cb <filename> を指定した場合、ツリー オブジェクトをダブルクリックすると関連付けられている Tcl スクリプト ファイルが評価されます。そして、ツリーオブジェクトをダブルクリックすると、この Tcl スクリプトの OnDoubleClick proc が、ダブルクリックしたオブジェクトをパラメータとして呼び出されます。

例 **660:**

```
std::BrowserReportInit MyBrowser
std::BrowserReport -expanded [std::GetActiveProject]
[std::GetSelection]
```

std::Button

表記

```
std::Button [-name <buttoncaption>] [-command <commandproc>]
```

例 **661:**

```
std::Button -variable B1 -name "Cancel" -parent $parentWnd -x 20
-y 90 -w 60 -h 25 -command OnCancel
$B1 -name "New Name" -x 40 -y 90 -w 100 -h 25
```

std::ComboBox

表記

```
std::ComboBox [-stringlist <list>][-edit <editstate>][-sort
<sortstate>][-selected <item>][-selchangecmd <selchgproc>][-
editchangecmd <editchgproc>]
```

説明

- `-stringlist <list>`：コンボボックス コントロールの項目を一覧表示します。
- `-edit <editstate>`：コンボ コントロールの編集機能を設定します。パラメータ editstate はブール値です。値を「true」に設定すると、コンボのテキストを編集できます。「false」に設定した場合はコンボのテキストを編集できません。デフォルト値は「false」です。
- `-sort <sortstate>`：パラメータ sortstate はブール値です。値を「true」に設定すると、コンボに表示されるリストがソーティングされます。
- `-selected <item>`：選択項目を設定します。
- `-selchangecmd <selchgproc>`：コンボ コントロールの選択を変更したときに呼び出されるコマンドの名前を設定します。コマンド selchgproc にはパラメータが 1 つあります。これは新しい選択文字列です。
- `-editchangecmd <editchgproc>`：コンボ コントロールの編集ボックス内のテキストをユーザが変更したときに呼び出されるコマンドの名前を設定します。コマンド editchgproc にはパラメータが 1 つあります。これは編集ボックスのテキストです。

例 **662:**

```
std ::ComboBox -variable C1 -name "" -parent $parentWnd -x 130 -
y 55 -w 70 -h 150
$C1 -stringlist {one two three}
```

std::Dialog

表記

```
std::Dialog -variable <dialogvar> -name <dialogname> -w
<dialogwidth> -h <dialogheight> -onbuilddialog <dialogbuildproc> -
oninitdialog <dialoginitproc> -onclosedialog <dialogcloseproc> -
closecmddialog <dialogclosecmd>
```

説明

ダイアログボックスを作成し、直ちに表示します。

ダイアログボックスの説明は、dialogbuildproc コマンドで行います。ダイアログボックスはモードによって異なります。

例 **663:**

```
std::Dialog -variable "MyDialogBox" -name "Projects" -w 450 -
h 300 -onbuilddialog OnBuildDialog -oninitdialog OnInitDialog
-onclosedialog OnCloseDialog -closecmddialog CloseDialog
```

- -variable <dialogvar> : ダイアログボックスに含む変数を設定します。
- -name <dialogname> : ダイアログボックスのタイトルを設定します。
- -w <dialogidth> : ダイアログボックスの幅を設定します。ダイアログボックスは画面の中央に作成されます。
- -h <dialogheight> : ダイアログボックスの高さを設定します。
- -onbuilddialog <dialogbuildproc> : ダイアログボックスの作成時に呼び出される Tcl コマンドを指定します。ここで、ウィンドウのコントロールを作成できます。ダイアログボックスは表示のたびに作成する必要があります。

例 **664:**

```
proc OnBuildDialog {adrDialog} {
  std::Label -name "Quality Model:" -parent $parentWnd -x 10 -y
25 -w 70 -h 25
}
```

- -oninitdialog <dialoginitproc> : ダイアログボックスを表示するときに呼び出される Tcl コマンドを指定します。ここで、ダイアログボックスに表示されるデータを初期化できます。

例 **665:**

```
proc OnInitDialog {adrDialog} {
        std ::Output  "Dialog initialization¥n"
}
```

- -onclosedialog <dialogcloseproc> : ダイアログボックスを閉じるときに呼び出される Tcl コマンドを指定します。

例 **666:**

```
proc OnCloseDialog {parentWnd} {
    return 1
}
```

- -closecmddialog <dialogclosecmd> : ダイアログボックスをプログラムで閉じるために必要な Tcl 環境の新しいコマンドを定義します。

std::DirectoryDialog

［directory selection］ダイアログを表示します。

表記

```
package require dialogs
std::DirectoryDialog ?-fromdir directory?
```

説明

このコマンドは、ユーザーがディレクトリを参照して選択できる［directory selection］ダイアログを表示します。オプションで、*directory* を使用して、ダイアログで選択される初期ディレクトリを設定できます。デフォルト値は、現在のディレクトリです。

戻り値は選択されたディレクトリのパスです。ユーザーが操作をキャンセルする場合は、0 になります。

例 667:

以下の例では、最初に C:¥Temp に設定されている［directory selection］ダイアログを表示して、実行された選択を変数に格納します。

```
package require dialogs
set dirPath [std::DirectoryDialog -fromdir "C:¥Temp"]
```

std::ExecuteCOMClient

対話型 COM クライアントを起動します。

表記

```
std::ExecuteCOMClient COMAddInProgId entityRef ?entityRef ...?
```

説明

このコマンドは、インターフェイス ITtdInteractiveClient を実装する対話型 COM クライアントを呼び出すために使用されます。このコマンドは、2 つの引数、COM クライアントのプログラム ID を表す *COMAddinProgId*、COM クライアントに渡されるモデル エンティティを表す 1 つ以上の *entityRefs* を取ります。

例 668

現在のワークスペース内のアクティブなプロジェクトに対して動作する Tau.CodeGen という ID の COM クライアントは、以下のように呼び出されます。

```
set project [std::GetActiveProject]
set session [std::GetModels -kind U2 -project $project]
std::ExecuteCOMClient "Tau.CodeGen"  $session
```

参照

COM API

std::FileOpenDialog

［file selection］ダイアログを表示します。

表記

```
package require dialogs
std::FileOpenDialog ?-filter visibility?
```

説明

このコマンドは、ユーザーがファイルを参照して選択できる［file selection］ダイアログを表示します。オプションの *visibility* 引数では、以下のように文字列を使用して、どのようなファイルをダイアログで表示可能にするかを制御します。

- `text|pattern|`
  text は、表示するテキストです。*pattern* は、実際のフィルタリングを制御します。

戻り値は選択されたファイルのパスです。ユーザーが操作をキャンセルする場合は、0 になります。

例 669:

［file selection］ダイアログ

```
package require dialogs

set res [std::FileOpenDialog -filter "Text Files
(*.txt)|*.txt|"]
```

std::FileSaveDialog

［file save］ダイアログを表示します。

表記

```
package require dialogs
std::FileSaveDialog ?-filename filename? ?-fileext extension?
```

説明

このコマンドは、ユーザーがファイルを参照して指定できる［file save］ダイアログを開きます。オプションの引数は 2 つ受け入れられます。*filename* は、ファイルのデフォルト名を設定します。*extension* は、ファイルのデフォルト拡張子を設定します。つまり、ユーザーによって特に指定されない場合に使用する拡張子です。

戻り値は指定されたファイルのパスです。ユーザーが操作をキャンセルする場合は、0 になります。

指定されたファイルがすでに存在する場合、ユーザーは置換を確定するように要求されます。

例 670:

以下の例は、ファイル名とファイル拡張子の初期設定で［file save］ダイアログを表示する方法と、ユーザーの選択を変数に格納する方法を示しています。

```
package require dialogs

set res [std::FileSaveDialog -filename "myTextfile" -fileext
"txt"]
```

std::Frame show-window

Tau フレームウィンドウの表示／非表示を切り替えます。

表記

```
std::Frame show-window show-window-value
```

説明

`std::Frame show-window` には、以下のいずれかの値を指定できます。

- hide：このウィンドウを非表示にし、他のウィンドウをアクティブにします。
- `minimize`：ウィンドウを最小化し、システムのリストの最上位のウィンドウをアクティブにします。
- `restore`：ウィンドウをアクティブにして表示します。ウィンドウが最小化または最大化されている場合、Windows によって元のサイズと位置に戻されます。
- `show`：ウィンドウをアクティブにし、現在のサイズと位置で表示します。
- `showmaximized`：ウィンドウをアクティブにし、最大化して表示します。
- `showmaximized`：ウィンドウをアクティブにし、アイコンとして表示します。

- showminnoactive：ウィンドウをアイコンとして表示します。現在アクティブなウィンドウがアクティブになります。
- showna：ウィンドウを現在の状態で表示します。現在アクティブなウィンドウがアクティブになります。
- shownoactivate：ウィンドウを最後に表示されたサイズと位置で表示します。現在アクティブなウィンドウがアクティブになります。
- shownormal：ウィンドウをアクティブにして表示します。ウィンドウが最小化または最大化されている場合、Windows によって元のサイズと位置に戻されます。

例 **671:**

```
std::Frame show-window minimize
```

std::GetActiveProject

現在のワークスペース内のアクティブなプロジェクトを取得します。

表記

std::GetActiveProject

説明

Tau ワークスペースには、そのうちの 1 つが常にアクティブになる 1 つ以上のプロジェクトが含まれます。このコマンドはアクティブなプロジェクトの参照を返します。

例 672

以下の例は、現在アクティブになっているプロジェクトの参照を変数に格納する方法を示しています。

```
set curProj [std::GetActiveProject]
```

std::GetInstallationDirectory

Tau インストール ディレクトリのパスを取得します。

表記

std::GetInstallationDirectory

説明

このコマンドは、Tau がインストールされているディレクトリのパスが含まれた文字列を返します。

例 673:

以下のコードは、Tcl 内蔵コマンド 'file' を使用して、Tau インストール ディレクトリと、addins サブディレクトリのパスを結合する方法を示しています。

```
set installdir [std::GetInstallationDirectory]
set addinsDir [file join $installDir addins]
```

std::GetKind

Tcl オブジェクトの一般的な種別を識別する文字列を取得します。

表記

```
std::GetKind object
```

説明

このコマンドは Tcl オブジェクト全体にわたる分類を識別するための文字列を返します。主として、ある Tcl コマンドに入力として渡されるオブジェクトの種別が、そのコマンドによってサポートされているかどうかを確認するために使用します。現在、戻り値として 2 つの異なる種類の識別子あります。

- **U2**
  オブジェクトは UML モデルのエンティティです。モデルビューで表示されるオブジェクトの大半はこの種類です。
- **project**
  オブジェクトは “project model” エンティティです。ファイルビューで表示されるオブジェクトの大半はこの種類です。

例 674:

以下のコードは、ワークスオブジェクトペースウィンドウ（モデルビューまたはファイルビュー）で選択されているノードの種別を出力します。

```
output \n[std::GetKind [std::GetSelection] ]
```

std::GetLocaleDirectory

表記

```
std::GetLocaleDirectory
```

説明

`std::GetLocaleDirectory` コマンドは、インストールディレクトリへの絶対パスを返します。このディレクトリにはロケール固有の DLL とファイルが格納されています。

例 675:

現在のロケールが日本語（jp）の場合、std::GetLocaleDirectory は Tau インストールディレクトリへの絶対パスの連結と、「/locale/jp」を返します。

std::GetModels

ロードされたモデルのリストを取得します。

表記

`std::GetModels` ?**`-kind`** *`modelKind`*? ?**`-project`** *`projRef`*?

説明

このコマンドは、ロードされたモデルのリストを返します。引数を指定しない場合は、それぞれの種類に関係なく、現在のワークスペース内のすべてのモデルが返されます。

オプションの引数 *modelkind* を使用して、出力に特定の種類のモデルのみが含まれるように制限できます。現在は以下の値がサポートされています。

- **U2**
  UML モデルのみが返されます。

オプションのパラメータ *projRef* を使用して、出力に当該のプロジェクトに基づくモデルのみが含まれるように制限できます。

例 676:

以下のように、ワークスペース内の全プロジェクトからすべてのモデルを取得します。

```
set allModels [std::GetModels]
```

例 677:

以下のように、ワークスペース内の全プロジェクトからすべての UML モデルを取得します。

```
set umlModels [std::GetModels -kind U2]
```

例 678:

以下のように、ワークスペース内のアクティブ プロジェクトから UML モデルを取得します。

```
set curProject [std::GetActiveProject]
set model [std::GetModels -kind U2 -project $curProject]
```

std::GetProject

プロジェクトのリストを取得します。

表記

std::GetProject ?*entityRef*?

説明

このコマンドは、プロジェクトのリストを返します。引数を指定しない場合は、現在のワークスペース内のすべてのプロジェクトの参照が含まれているリストが返されます。

特定のエンティティが含まれているプロジェクトの参照が必要な場合は、オプションの引数 *entityRef* を使用します。

例 679:

以下の例は、現在のワークスペース内のすべてのプロジェクト上で、プロシージャ “ProcessIt” を呼び出す方法を示しています。

```
set allProjs [std::GetProject]

foreach p $allProjs {
  ProcessIt $p
}
```

例 680:

以下の例は、エンティティが含まれているプロジェクトを取得する方法を示しています。

```
set thisProj [std::GetProject $anEntity]
```

std::GetProjectPath

表記

```
std::GetProjectPath <projectRef>
```

説明

このコマンドは、プロジェクトの絶対パスを返します。

例 **681:**

```
# first select a project node in the FileView
set selection [std::GetSelection]
set pathname [std::GetProjectPath $selection]

if { $selection != "" } {
std::Output "This is the selected project's path: $pathname¥n";
} else {
        std::Output "A non-project node is selected.¥n";
}
```

std::GetSelection

現在選択されているエンティティのリストを取得します。

説明

このコマンドは、現在選択されているすべてのエンティティの参照が含まれているリストを返します。

例 682:

以下の例は、選択されているすべての要素上で、プロシージャ “ProcessIt” を呼び出す方法を示しています。

```
set sel [std::GetSelection]

foreach s $sel {
  ProcessIt $s
}
```

std::GetUserAddinsDirectory

ユーザーアドインディレクトリのパスを取得します。

表記

```
std::GetUserAddinsDirectory
```

説明

このコマンドは、ユーザー アドインの格納用ディレクトリのパスが含まれている文字列を返します。

Windows のデフォルト パス：

```
c:¥Documents and Settings¥<username>¥Application
Data¥IBM¥Rational¥<Tau version>¥addins
```

UNIX のデフォルト パスでは、$HOME 変数を使用し、次に「IBM/Rational」を追加します。そこからは、パスは同じになります。

環境変数 `TAU_USER_ADDINS_DIR` を設定して、デフォルト パスを変更できます。

std::GetTeamAddinsDirectory

チームアドイン ディレクトリのパスを取得します。

表記

```
std::GetTeamAddinsDirectory
```

説明

このコマンドは、チームで使用するユーザー アドインの格納用ディレクトリのパスが含まれている文字列を返します。

Windows のデフォルト パス：

```
c:¥Documents and Settings¥<username>¥Application Data¥IBM¥
Rational¥Shared¥TeamAddins
```

UNIX のデフォルト パスでは、$HOME 変数を使用し、次に「IBM/Rational」を追加します。そこからは、パスは同じになります。

環境変数 `TAU_TEAM_ADDINS_DIR` を設定して、デフォルト パスを変更できます。

std::GetCompanyAddinsDirectory

会社規模で使用するアドイン ディレクトリのパスを取得します。

表記

```
std::GetCompanyAddinsDirectory
```

説明

このコマンドは、会社規模で使用するユーザー アドインの格納用ディレクトリのパスが含まれている文字列を返します。

Windows のデフォルト パス :

```
c:¥Documents and Settings¥<username>¥Application Data¥IBM¥
Rational¥Shared¥CompanyAddins
```

UNIX のデフォルト パスでは、$HOME 変数を使用し、次に「IBM/Rational」を追加します。そこからは、パスは同じになります。

環境変数 TAU_COMPANY_ADDINS_DIR を設定して、デフォルト パスを変更できます。

std::GetUserDirectory

ユーザー情報ディレクトリのパスを取得します。

表記

```
std::GetUserDirectory
```

説明

このコマンドは、ユーザー情報の格納用ディレクトリのパスが含まれている文字列を返します。例 :

```
c:¥Documents and Settings¥<username>¥Application
Data¥IBM¥Rational¥<Tau version>
```

std::GetWebServerPort

Tau Web サーバーが現在使用している TCP/IP ポート番号を取得します。.

表記

```
std::GetWebServerPort
```

説明

このコマンドは、Tau Web サーバーが現在使用している TCP/IP ポート番号を返します。通常は、std::HtmlReport コマンドに渡される URL を組み立てるために、使用されます。以下の例をご覧ください。

例 683:

```
set url
"http://localhost:[std::GetWebServerPort]/file/index.html"
```

```
   std::HtmlReport $url
```

std::HtmlReport

`html` ファイルを開きます。

表記

`std::HtmlReport` *`url`*

説明

このコマンドは、Tau に `html` ファイルをロードして表示します。*url* 引数は、当該の URL が含まれている文字列です。

例 684:

Tau デスクトップ上の IBM Rational ホーム ページの表示は、以下のように行います。

```
   std::HtmlReport "www.ibm.com/rational"
```

例 685:

Tau デスクトップ上の `html` ファイルの表示は、以下のように実行されます。

```
   std::HtmlReport "C:¥test.htm"
```

std::IsModified

プロジェクトが変更されているかどうかをチェックします。

表記

`std::IsModified` *`projRef`*

説明

このコマンドは、プロジェクト参照 *projRef* で指定されたプロジェクトが変更されているかどうかをチェックします。変更されていない場合は 0、それ以外の場合は 1 を返します。

例 686: ――

以下の例では、アクティブ プロジェクトが変更されている場合、ワークスペースを保存します。

```
if { [std::IsModified $std::activeproject] } {
  std::SaveAll
}
```

――

std::Label

表記

```
std::Label [-name <staticname>]
```

説明

`-name <staticname>` : 静的コントロールのキャプションを設定します。

例 **687:** ――

```
std::Label -name "Quality Model:" -parent $parentWnd -x 10 -y 25
-w 70 -h 25
```

――

std::Locate

表記

```
std::Locate <locatestring>
```

説明

このコマンドは、<`locatestring`> によって記述されたオブジェクトを検索します。この文字列は、`DataServer` によって理解されなければなりません。

std::MessageDialog

［message］ダイアログを表示します。

表記

```
package require dialogs

std::MessageDialog ?-name caption? ?-message message? ?-style style?
?-icon icon?
```

説明

このコマンドは、*caption* に設定されたタイトルと、*message* と同じメッセージ テキストが含まれる［message］ダイアログを表示します。2 つの引数 *style*、*icon* は、ボタンの構成と、ダイアログに表示するアイコンを制御します。引数はすべてオプションです。

戻り値は、ダイアログを終了するときにユーザーがクリックしたボタンを反映します。可能な戻り値として、Ok を表す 1、キャンセルを表す 2、再試行を表す 4、Yes を表す 6、No を表す 7 があります。

style は、以下のいずれかの値を取ることができます。

- **ok**
  ［OK］ボタンが提供されます。これはデフォルトの振る舞いです。
- **okcancel**
  ［OK］ボタンと［cancel］ボタンが提供されます。
- **retrycancel**
  ［retry］ボタンと［cancel］ボタンが提供されます。
- **yesno**
  ［yes］ボタンと［no］ボタンが提供されます。
- **yesnocancel**
  ［yes］ボタン、［no］ボタン、［cancel］ボタンが提供されます。

icon は、以下のいずれかの値を取ることができます。

- **stop**
  ［stop］アイコンが提供されます。
- **question**
  ［question］アイコンが提供されます。
- **warning**
  ［warning］アイコンが提供されます。
- **information**
  ［information］アイコンが提供されます。これはデフォルトの振る舞いです。

例 688:

以下の例では、［question］ダイアログをポップアップ表示して、ユーザーの答えである yes (6)、no (7)、または cancel (2) を変数に格納します。

```
package require dialogs

set res [std::MessageDialog -name "IBM Rational Tau" -message
"Save changes?" -style yesnocancel -icon question]
```

std::OpenDocument

ドキュメントを開きます。

表記

std::OpenDocument *filename*

説明

このコマンドは、引数 *filename* によって指定されているドキュメントを Tau で開きます。通常、このコマンドは、Tau ワークスペース（`.ttw` ファイル）または Tau プロジェクト（`.ttp` ファイル）をロードするために使用されますが、たとえば、プレーン テキスト ドキュメントをロードして表示する場合にも使用できます。ファイルの拡張子によって、ロードの結果が制御されます。

指定されたファイルを開けない場合は、Tcl エラーが生成されます。

例 689:

Tau ワークスペースは以下のように開かれます。

```
std::OpenDocument "C:¥Work¥MyProjects.ttw"
```

std::Output

メッセージを［Tcl output］タブに出力します。

表記

std::Output <text>

説明

Output コマンドは、*message（メッセージ）*を、出力ウィンドウの［Tcl output］タブに出力します。

例 690:

以下の例では、［Tcl output］タブに、文字列「Hello World」、続けて復帰改行が出力されます。

```
std::Output "Hello World¥n"
```

std::OutputTab

表記

```
std::OutputTab clear [clear [-ident <tabident> | <tabname>] |
activate [-ident <tabident>] | <tabname>]
```

説明

このコマンドは、出力タブの内容を解除するか、出力タブをアクティブにします。

例 **691:**

```
set selection [std::GetSelection]
std::ReportInit MyTab -ident mt MyFirst 100 0 MySecond 200 0
std::Report $selection -ident mt "This is the second column."
std::OutputTab clear -ident mt
```

std::Report

表記

std::Report <sourceobject> [-ident <tabident>] [<string>]*

説明

このコマンドは、レポート タブに復帰改行を追加するために使用します。

<sourceobject> によって、最初の列を自動的に埋めます（アイコン + テキスト）。

<string> の値は、残りの列を埋めるために使用されるラベルです。

例 **692:**

```
set selection [std::GetSelection]
std::ReportInit MyTab -ident mt MyFirst 100 0 MySecond 200 0
std::Report $selection -ident mt "This is the second column."
```

std::Quit

［close window］ボタンと同じ働きをします。

表記

std::Quit

説明

std::Quit コマンドは、「終了」メッセージを Tau のメイン ウィンドウに掲示します。これは、メイン ウィンドウの右上の［close window］ボタンをクリックするのと同じです。

例 **693:**

```
std::Quit
```

注記

これで完全に終了できるとは限りません。保存が必要なファイルが開いたままになっている場合があります。この場合、Tau は「Do you want to save?」というメッセージボックスを表示します。

std::ReportInit

表記

```
std::ReportInit <tabname> [-ident <tabident>] [-check] [-cb
<filename>] [<columnlabel> <columnwidth> <columnalignment>]+
```

説明

このコマンドは、出力ウィンドウに新しいレポート タブを作成するために使用します。

`-check` を使用すると、各結果行の前にチェックボックスが表示されます。これらのチェックボックスを使用して、より効率的なナビゲーションを行うことができます。F4 キーでナビゲートする場合は、チェックを付けた行のみ選択され、他は無視されます。

`-cb <filename>` を指定した場合、ライン オブジェクトをダブルクリックすると関連付けられている Tcl スクリプトファイルが評価され、ダブルクリックしたオブジェクトとともにこの Tcl スクリプトの `OnDoubleClick proc` がパラメータとして呼び出されます。

`<columnlabel> <columnwidth> <columnalignment>` の各項目については、対応するラベル、幅、行揃えで新しい列が作成されます。行揃えの「0」は左揃え、「1」は右揃えを意味します。ソーティングは、右揃えの列は数字、左揃えの列はテキストとして処理されます。

例 **694:**

```
set selection [std::GetSelection]
std::ReportInit MyTab -ident mt MyFirst 100 0 MySecond 200 0
std::Report $selection -ident mt "This is the second column."
```

std::SaveAll

ワークスペースまたはプロジェクトを保存します。

表記

```
std::SaveAll ?filePath?
```

説明

このコマンドは、ワークスペースの内容全体を保存します。オプションで、ロードされたプロジェクトの *filePath* を指定した場合は、プロジェクトの内容全体が保存されます。

例 695:

ワークスペース全体は以下のように保存されます。

```
std::SaveAll
```

std::TextReport

ファイルを開き、カーソルを移動します。

表記

```
std::TextReport <filename> [<line>] [<column>]
```

説明

このコマンドは、ファイル <filename> を開き、指定された場合は <line> および <column> にカーソルを移動します。列を指定せずに行のみ指定できます。

例 **696:**

```
std::TextReport [std::GetLocaleDirectory]/java.ini 5 0
```

std::View

デスクトップを制御します。

表記

```
std::View [maximize | restore | minimize | close | coordinates]
[<view>]
activate [<view>|next]
getactive
count
```

説明

std::View コマンドにより、ウィンドウのキャンバスを制御できます。

- maximize : 指定したビューを最大化します。
- restore : 最小化したビューを元のサイズに戻します。

- `minimize` : 指定したビューを最小化します。
- `close` : 指定したビューを閉じます。
- `coordinates` : 指定したビューの座標を返します。
- `activate` : 指定したビューまたは「次の」ビューをアクティブにします。
- `getactive` : 上記オプションで使用するアクティブ ビューのハンドルを返します。
- `count` : 開かれているビューの数を返します。

例 **697:**

```
set activeview [std::View getactive]
std::View minimize $activeview
```

ユーザー インターフェイス アドイン固有コマンド

以下のコマンドは、アドインから実行される Tcl スクリプトによって使用されます。その目的は、Tau のユーザーインターフェイスをカスタマイズであり、通常はそのアドイン用のユーザーインターフェイスを作成することです。

コマンド	説明
std::AddCommand	新しいコマンドを追加します。
std::AddContextMenu	ショートカット メニューを追加します。
std::AddMenu	メニューを追加します。
std::AddToolbar	ツールバーを追加します。
std::Declare	定義と使用を分離します。

このセクションのコマンドを使用する場合は、以下のように `commands` パッケージを Tcl インタープリタにロードする必要があります。

```
package require commands
```

ユーザーインターフェイスアドイン固有のカスタマイズコマンドは、アドイン 経由で実行される Tcl スクリプトからのみ使用できます。スタンドアローン Tcl スクリプトで実行されるエージェントのようなアドイン外部の Tcl スクリプトでは利用できません。

std::AddCommand

新しいコマンドを追加します。

表記

```
package require commands

std::AddCommand -variable var -name name -statusmessage message
-tooltip tooltip -accelerator accelerator -imagefile imagefile
-onactivatecommand activateproc ?-onenablecommand enableproc?
?-oncheckcommand checkproc?
```

説明

このコマンドは、*var* によって識別された新しいコマンドを定義して、メニューかツールバーまたはその両方から呼び出せるようにします。コマンドは、たとえば、std::AddMenu や std::AddToolbar を呼び出す前に定義する必要があります。

Tau ステータス バーとツール チップに表示するメニュー コマンドのテキストは、それぞれ引数 *name*、*message*、*tooltip* によって指定します。コマンドのアクセラレータまたはショートカット キーは、「Ctrl+」、「Shift+」または「Alt+」（またはその両方）と、その後に続く文字（Shift がアクセラレータ コマンドの一部かどうかに関係なく大文字）の組み合わせが含まれている文字列を使用して、引数 *accelerator* で定義します。また、ツールバーに表示するイメージは、引数 *imagefile* を使用して、Tau bin ディレクトリを基準にしたビットマップ ファイルのパスで指定できます。

activateproc は、コマンドの起動時に呼び出すプロシージャの名前です。オプションとして、コマンド メニュー項目またはツールバー ボタンを表示するかどうかを制御するプロシージャの名前を、*enableproc* 引数で指定できます。同じように、*checkproc* は、メニュー項目またはツールバー ボタンのクリック（ON/OFF）を制御します。これら 3 つのプロシージャでは、実行時にコマンドの名前を割り当てられる 1 つのプロシージャを必ず受け入れる必要があります。また、*enableproc* と *checkproc* は、いずれもブール値（0 は false、その他は true を表す数値形式、または true を表す true か yes、false を表す false か no などの文字列値）を返す必要があります。

例 698:

以下の例は、可能なすべての引数を使用するコマンド定義を示しています。

```
package require commands

proc OnTest { cmd } {
  std::Output "OnTest was called¥n"
}

proc OnEnableTest { cmd } {
  return 1
}

proc OnCheckTest { cmd } {
  return 0
}

std::AddCommand -variable cmdTst -name "Test" -statusmessage
"To test a command" -tooltip "Test command" -accelerator
"Ctrl+Shift+Z" -imagefile "../addins/Tst/etc/Tst.bmp" -
onactivatecommand OnTest -onenablecommand OnEnableTest -
oncheckcommand OnCheckTest
```

参照

std::Declare

std::AddContextMenu

コマンド構文の管理については、第 72 章「ダイアログ ヘルプ」の 2152 ページ、「その他の作業」

std::AddContextMenu

ショートカット メニューを追加します。

表記

```
package require commands

std::AddContextMenu -variable menuvar -commands commands
-onenablemenu enableproc
```

説明

このコマンドは、*menuvar* 引数によって識別されたショートカット メニューを追加します。*commands* 引数は、ショートカット メニューに表示するコマンドをその表示順で表す一連のコマンド識別子です。2 つのメニュー項目間にセパレータを挿入するには、コマンド名「separator」を使用します。ショートカット メニューを表示するかどうかを制御するプロシージャは、*enableproc* 引数で指定します。このプロシージャには 1 つの引数があります。ビュー、［output］タブ、または［browser］タブなどのショートカット メニューの呼び出し元になるウィンドウの ID です。このプロシージャは、ブール値を返します。

例 699:

以下の例は、［ファイル ビュー］から起動した場合に表示されるショートカット メニューの追加方法を示しています。

```
package require commands

proc OnMenuEnable { ident } {
  if { $ident == "FILEVIEW" } {
    return 1
  } else {
    return 0
  }
}

std::AddContextMenu -variable cmTest -commands { cmdTst } -
onenablemenu OnMenuEnable
```

参照

std::AddCommand

std::Declare

コマンド構文の管理については、第 72 章「ダイアログ ヘルプ」の 2152 ページ、「その他の作業」

std::AddMenu

メニューを追加します。

表記

```
package require commands

std::AddMenu -variable menuvar -commands commands -path path
?-position pos?
```

説明

このコマンドは、*menuvar* 引数によって識別されたメニューを追加します。*commands* 引数は、メニューに表示するコマンドをその表示順で表す一連のコマンド識別子です。2 つのメニュー項目間にセパレータを挿入するには、コマンド名「separator」を使用します。メニューの位置は、先頭にメニュー名がある一連の文字列形式の引数 *path* によって制御されます。パス内の存在しない要素は、自動的に作成されます。

オプションの引数 *pos* を使用して、メニューの位置付けを制御できます。*pos* は、以下のいずれかの形式を取ることができます。

- **after** `menu`
  新しいメニュー項目は、兄弟メニュー項目識別子である *menu* の後に挿入されます。
- **first**
  新しいメニュー項目は、その親の最初の子として挿入されます。
- **last**
  新しいメニュー項目は、その親の最後の子として挿入されます。

例 700:

以下の例は、`cmdTest` という追加コマンド用のセパレータとメニュー項目を、［ツール］メニューの最後に追加する方法を示しています。

```
package require commands

std::AddMenu -variable mTest1 -commands { separator cmdTst }
-path { &Tools } -position last
```

例 701:

もう一つの例は、［ツール］メニューのサブメニュー項目からコマンドにアクセスできるようにする方法を示しています。

```
std::AddMenu -variable mTest2 -commands { cmdTst } -path {
&Tools Test }
```

参照

std::AddCommand

std::Declare

std::AddToolbar

ツールバーを追加します。

表記

```
package require commands
std::AddToolbar -variable tbvar -commands commands
```

説明

このコマンドは、*tbvar* 引数によって識別されたツールバーを追加します。*commands* 引数は、ツールバーに表示するコマンドをその表示順で表す一連のコマンド識別子です。2 つのツールバー ボタン間にセパレータを挿入するには、コマンド名「separator」を使用します。

例 702:

コマンド `cmdTest` に関連付けられたボタン付きのツールバーは、以下のように追加されます。

```
package require commands
std::AddToolbar -variable tbTest -commands { cmdTst }
```

参照

std::AddCommand

std::Declare

std::Declare

定義と使用を分離します。

表記

```
package require commands
std::Declare option
```

説明

このコマンドでは、たとえば、コマンドの定義と使用を相互に分離できます。したがって、定義をファイルの先頭、または別のファイル内で行う一方で他の場所で使用できます。

option 引数は、以下のいずれかの値を取ることができます。

- **addcommand** `args`
  コマンドは、std::AddCommand に使用される引数を表す *args* で宣言されます。
- **addcontextmenu** `args`
  コマンドは、std::AddContextMenu に使用される引数を表す *args* で宣言されます。
- **addmenu** `args`
  コマンドは、std::AddMenu に使用される引数を表す *args* で宣言されます。
- **addtoolbar** `args`
  コマンドは、std::AddToolbar に使用される引数を表す *args* で宣言されます。

例 703:

コマンドの定義と使用を分離する方法の例は、以下のとおりです。

```
package require commands

std::Declare addcommand -variable cmdTst -name "Test" -
statusmessage "For testing" -tooltip "Test command" -
accelerator "CTRL+SHIFT+Z" -imagefile "" -onactivatecommand
OnTest

proc OnTest { cmd } {
std::Output "OnTest was called¥n"
}

proc Init{} {
  std::AddCommand cmdTst

  std::AddMenu -variable mTest -commands { separator cmdTst }
-path { &Tools } -position last
}
```

参照

std::AddCommand

std::AddContextMenu

std::AddMenu

std::AddToolbar

モデル コマンド

モデル アクセスに使用できるほとんどの Tcl コマンドは、ITtdModel セクション内の、対応する COM API メソッドと同じです。

u2::FindByGuid	指定された GUID を持つエンティティを探します。
u2::New	新しいエンティティを作成します。
u2::Parse	U2P 構文がある文字列を解析して、結果のエンティティを返します。
u2::XMLDecode	XML としてエンコードされている 1 つのモデルをデコードして、結果のエンティティを返します。
u2::Save	モデルを保存します。
u2::CreateResource	モデルの新しいリソース（ファイル）を作成します。
u2::LoadFile	UML モデル ファイル（.u2）をモデルにロードします。
u2::InvokeAgent	指定されたモデル コンテキスト上で、エージェントをプログラムによって起動します。

例 **704:**

オプションの COM パラメータの Tcl オプションへのマッピングは、以下の COM モデル アクセス コマンドと合わせて説明しています。

```
HRESULT Parse(
         [in] BSTR strConcreteSyntax,
         [in, optional] VARIANT parseAs,
         [out, retval] ITtdEntities** ppEntities);
```

対応する Tcl コマンドは、以下のようになります。

u2::Parse *modelRef strConcreteSyntax* ?-**parseAs** *value*?

このコマンドはモデル フラグメントの参照を返します。2 つのパラメータ、モデル参照、*modelRef*、テキスト記述が含まれている文字列、*strConcreteSyntax* を指定する必要があります。オプションで、*value* を、指定された文字列の解析の試行方法に関するパーサのヒントとして使用できます。以下のように呼び出すことができます。

```
set expr [u2::Parse $model "x = 10" -parseAs "Expression"]
```

Tcl コマンド `InvokeAgent` は、やや特殊です。対応する COM メソッドの場合は、in/out パラメータを使用して、起動されたエージェントとの間でエージェント パラメータのやりとりをします。

u2::InvokeAgent *modelRef agentRef modelContextRef* ?agentParameters in/out <list>?

つまり、このパラメータに対応する Tcl 文字列を渡すのではなく、エージェント パラメータのリストを保持している Tcl 変数を渡す必要があります。このリストの文字列は、以下のルールに従って（優先順）、呼び出されたエージェントで特定のタイプの値として解釈されます。

1. ITtdEntity インターフェイスを実現するオブジェクトを表す Tcl id 文字列は、エンティティとして解釈されます。
2. 文字列「0」と「1」は、それぞれブール値の `false`、`true` として解釈されます。
3. どの数値文字列も整数値として解釈されます。
   整数の `0`、`1` と、ブール値の `false`、`true` を明確に区別するには、代わりに文字列「00」と「01」を使用できます。
4. すべての要素が ITtdEntity インターフェイスを実現するオブジェクトである整形式の Tcl リストである文字列は、エンティティのリストとして解釈されます。
   ITtdEntity オブジェクトと、ITtdEntity オブジェクトのリスト（これらの Tcl 表現は同一）を明確に区別するには、リストに「0」を追加できます。追加されるこれらの「0」は、タイプ ITtdEntity の NULL ポインタに変換されませんが、1 つのオブジェクトのみが含まれているリストを指定するために使用できます。
5. 上記のいずれのルールも適用されない場合、文字列はプレーン文字列として解釈されます。

例 **705:** ———————————————————————————————

以下の例は、`$modelContext` によって参照されているモデル コンテキスト上で、`$agent` によって参照されているエージェントを呼び出す方法を示しています。エージェントは、以下のタイプの 5 つの実引数を受け取ります。

1. ブール値の `true`
2. 整数値の `2`
3. `$entity` によって参照されるエンティティ
4. `$entity` によって参照されるエンティティが含まれているエンティティのリスト
5. The string ì`hello`î.

エージェントが起動されると、結果のパラメータ リストが出力されます。

```
set p [lappend p 1 2 $entity [list $entity 0] "hello"]
u2::InvokeAgent $model $agent $modelContext p
output "$p¥n"
```

———————————————————————————————————————

参照

Tcl コマンドへの COM のマッピング

u2::SelectMetaModel

どちらのメタモデルを有効にするかを選択します。

表記

u2::SelectMetaModel *metaModelPackage*

説明

このコマンドは、モデル内でどちらのメタモデルをアクティブにするかの制御を可能にします。［表示］>［モデルビューの再構成］メニューコマンドに対応しています。詳細については、モデル ビューの再構成を参照してください。

このコマンドの引数は、<<metamodel>> ステレオタイプが適用されたパッケージへの参照である必要があります。

例 706:

現在のワークスペースで見つかった最初のモデルについて、TTDDiagramView をアクティブなメタモデルとして設定します。

```
set model [lindex [std::GetModels -kind U2] 0]
set mm [u2::FindByName $model "TTDDiagramView"]
u2::SelectMetaModel $mm
```

エンティティ コマンド

エンティティ アクセス用の Tcl コマンドは、ITtdEntity セクション内の COM API メソッドで有効な当該メソッドと同じです。すべてのエンティティ アクセス コマンドの最初の Tcl 引数は、モデル エンティティの参照です。

u2::ApplyStereotype	与えられたステレオタイプをインスタンス化して、エンティティに適用します。
u2::Bind	モデル フラグメント内のすべての参照、またはエンティティの 1 つの参照をバインドします。
u2::Clone	エンティティのクローンを作成します。
u2::Create	エンティティのコンテキストで新しいエンティティを作成します。つまり、新しい直接的または間接的な子をエンティティに追加します。
u2::CreateInstance	シグニチャのインスタンスを作成します。
u2::Delete	モデルからエンティティを削除します。

u2::FindByName	あるエンティティのコンテキストから、別のエンティティを修飾名で検索するために、名前ルックアップを実行します。
u2::GetContainerMetaFeature	エンティティが含まれているメタ特性の名前を返します。
u2::GetDescriptiveName	エンティティの説明を返します。
u2::GetEntities	エンティティのメタ特性の値をエンティティのコレクションとして返します。
u2::GetEntity	エンティティのメタ特性の値をエンティティとして返します。
u2::GetMetaClassName	エンティティのメタクラスの名前を返します。
u2::GetModel	エンティティの属するモデルを返します。
u2::GetOwner	エンティティの所有者を返します。
u2::GetReference	参照を表すメタ特性の識別子を返します。
u2::GetReferringEntities	特定のメタ特性を通じてエンティティを参照するエンティティのコレクションを返します。
u2::GetTaggedValue	要素の指定されたプロパティ（タグ付き値）を返します。インスタンス式から任意の値を取得するときにも使用できます。
u2::GetValue	エンティティのメタ特性の値を文字列として返します。
u2::HasAppliedStereotype	エンティティに特定のステレオタイプが適用されているか判断します。
u2::IsKindOf	エンティティが特定のメタクラスの種類かどうかを判定します。
u2::MetaVisit このコマンドは COM メソッドの MetaVisitEx に対応します。	モデル フラグメントをトラバースして、コールバック インターフェイスで、含まれているエンティティごとにメソッドを呼び出します。このコマンドの使用例については、1938 ページの例 707 を参照してください。
u2::Move	エンティティをモデル内の現在位置から別の位置（所有者）に移動します。
u2::Replace	エンティティを別のエンティティで置き換えます。
u2::SetEntity	エンティティのメタ特性の値をエンティティとして設定します。
u2::SetTaggedValue	要素上のプロパティ（タグ付き値）を設定します。インスタンス式の任意の値を設定するときにも使用できます。
u2::SetValue	エンティティのメタ特性の値を文字列として設定します。

u2::UnlinkFromOwner	エンティティをモデル内の現在の所有者から切り離します。
u2::Unparse	エンティティの具体的な構文表現への逆構文解析。
u2::XMLEncode	エンティティを XML 表現にエンコードします。

Tcl は、エンティティ アクセス コマンドの 1 つである MetaVisit の場合、使い方が COM とやや異なります。インターフェイスのポインタを使用してコールバック関数を指定するのではなく、Tcl コマンドではプロシージャを受け入れます。以下にその実例を示します。

例 **707:** モデル エンティティのトラバース

以下の例は、現在ロードされている全モデル内のすべてのモデル エンティティがどのようにトラバースされるかを示してます。アクセスされたモデル エンティティごとにチェックが実行され、当該の要素が Definition（定義）かどうかが確認されます。そうなっている場合は、出力メッセージが生成されます。

```
proc PrintDef { def } {
  if { [u2::IsKindOf $def "Definition"] } {
    set name [u2::GetValue $def "Name"]
    std::Output "Found def: $name¥n"
  }
}

u2::MetaVisit [std::GetModels -kind U2] PrintDef
```

参照

Tcl コマンドへの COM のマッピング

リソース コマンド

リソースは、モデルが格納される物理格納ユニット（通常は、拡張子 ë.u2 窒 ™ 付くテキスト ファイル）を表します。リソース処理用の Tcl コマンドは、ITtdResource セクション内の COM API メソッドで有効な当該メソッドと同じです。したがって、すべてのリソース処理コマンドで、最初の Tcl 引数はリソースの参照になります。

注記

リソースがエンティティである場合もあります。したがって、すべてのエンティティ コマンドはリソースに対して使用できます。

コマンド	説明
u2::SaveResource このコマンドは COM メソッドの Save に対応します。	リソースと関連付けられたモデルエンティティを保存します。（通常は対応する .u2 ファイルを保存します）

参照

Tcl コマンドへの COM のマッピング

プレゼンテーション要素コマンド

プレゼンテーション要素は、たとえば、ダイアグラム、シンボル、ラインなどの図形表示が含まれた要素です。Tcl プレゼンテーション要素コマンドは、ITtdPresentationElement セクション内の COM API メソッドで有効な当該メソッドと同じです。したがって、すべてのプレゼンテーション要素コマンドで、最初の Tcl 引数はプレゼンテーション要素の参照になります。

注記

プレゼンテーション要素がエンティティである場合もあります。したがって、すべてのエンティティ コマンドはプレゼンテーション要素に対して使用できます。

u2::GenerateEMF	プレゼンテーション要素の EMF ファイル（Enhanced Meta File）を生成します。（非推奨）
u2::GenerateEMFEx	プレゼンテーション要素の EMF ファイル（Enhanced Meta File）をスケーリングのサポート付きで生成します。
	Generates an image file for a presentation element.

u2::GenerateEMF

プレゼンテーション要素のグラフィック表示に対応する EMF ファイル（Enhanced Meta File）を生成します。この関数は削除されています。代わりに、`u2::GenerateEMFEx` を使用してください。

表記

```
u2::GenerateEMF ITtdPresentationElement strFileName ?maxWidth
<Integer>? ?maxHeight <Integer>? ?optimizeForVectorGraphics
<Boolean>? ?includeFrame <Boolean>?
```

説明

この EMF ファイルでは、プレゼンテーション要素がツールのエディタに表示されるときと同じ外観になります。

パラメータ

`ITtdPresentationElement` の引数は生成する EMF ファイルのファイル名です。`strFileName` が相対パス指定の場合、そのパスはクライアント アプリケーションの現在の作業ディレクトリを基準にして解釈されます。

他の 4 つのオプションの引数は受け入れられます。

maxWidth および maxHeight パラメータを使用して、生成されたイメージの最大サイズを指定できます。イメージは、指定されたサイズに合わせて拡大・縮小されます。パラメータを省略すると、生成されたイメージはツールのエディタに表示されるサイズと同じになります。高さと幅の数値の単位は、1/10 mm です。現在、これらのパラメータが考慮されるのは、メソッドの呼び出し元のプレゼンテーション要素がダイアグラムの場合のみです。

optimizeForVectorGraphics が true に設定されていると、EMF 生成がベクトルグラフィックス用に最適化されます。デフォルトの振る舞いでは、この最適化は行われません。このパラメータは削除が考慮されており、下位互換を保つために使用されます。

includeFrame パラメータは、ダイアログのフレーム シンボルを EMF 生成に含めるかどうかを指定します。デフォルトでは、フレーム シンボルが含まれます。現在、このパラメータが考慮されるのは、メソッドの呼び出し元のプレゼンテーション要素がダイアグラムの場合のみです。

戻り値

このコマンドは値を返しません。

u2::GenerateEMFEx

プレゼンテーション要素のグラフィック表示に対応する EMF ファイル（Enhanced Meta File）を生成します。

表記

```
u2::GenerateEMFEx ITtdPresentationElement strFileName ?maxWidth
<Integer>? ?maxHeight <Integer>? ?optimizeForVectorGraphics
<Boolean>? ?includeFrame <Boolean>? ?scaleFactor <Integer>?
```

説明

この EMF ファイルでは、プレゼンテーション要素がツールのエディタに表示されるときと同じ外観になります。

パラメータ

ITtdPresentationElement の引数は生成する EMF ファイルのファイル名です。strFileName が相対パス指定の場合、そのパスはクライアント アプリケーションの現在の作業ディレクトリを基準にして解釈されます。

他の 4 つのオプションの引数は受け入れられます。

maxWidth および maxHeight パラメータを使用して、生成されたイメージの最大サイズを指定できます。イメージは、指定されたサイズに合わせて拡大・縮小されます。scaleFactor が指定されていない場合、イメージは、指定されたサイズに合わせて拡大・縮小されます。パラメータを省略すると、生成されたイメージはツールの

エディタに表示されるサイズと同じになります。高さと幅の数値の単位は、1/10 mm です。現在、これらのパラメータが考慮されるのは、メソッドの呼び出し元のプレゼンテーション要素がダイアグラムの場合のみです。

`optimizeForVectorGraphics` が true に設定されていると、EMF 生成がベクトルグラフィックス用に最適化されます。デフォルトの振る舞いでは、この最適化は行われません。このパラメータは削除が考慮されており、下位互換を保つために使用されます。

`includeFrame` パラメータは、ダイアログのフレーム シンボルを EMF 生成に含めるかどうかを指定します。デフォルトでは、フレーム シンボルが含まれます。現在、このパラメータが考慮されるのは、メソッドの呼び出し元のプレゼンテーション要素がダイアグラムの場合のみです。

オプションのパラメータ `scaleFactor` を指定すると、イメージの生成前に、元のダイアグラムが拡大・縮小されます。拡大・縮小率は、整数で指定します。この値は元のダイアグラム サイズに対するパーセントに変換されます。`scaleFactor` と `maxWidth`/`maxHeight` の両方を引数として指定すると、操作の結果として複数のイメージが生成されます。`scaleFactor` パラメータを指定した場合、生成後のファイル名は `strFileName` パラメータを指定した場合と同じになりますが、ファイル拡張子の前に数字が振られます。

戻り値

このコマンドは値を返しません。

参照

Tcl コマンドへの COM のマッピング

u2::GenerateImage

このコマンドは、プレゼンテーション要素のグラフィック表示に対応する指定された種類の画像ファイルを生成します。ツールのエディタに表示されるプレゼンテーション要素の外観はこの画像ファイルと同じになります。

表記

```
u2::GenerateImage ITtdPresentationElement imgKind strFileName
```

説明

ツールのエディタに表示されるプレゼンテーション要素の外観はこの画像ファイルと同じになります。

パラメータ

`imgKind` パラメータは、どの種類の画像ファイルを生成するかを指定します。パラメータの有効な値を下表に示します。

imgKind 値	**説明**
`JPEG`	`JPEG` 画像ファイルを生成します。
`BMP`	`BMP` 画像ファイルを生成します。
`GIF`	`GIF` 画像ファイルを生成します。
`TIFF`	`TIFF` 画像ファイルを生成します。
`TARGA`	`TGA (Targa)` 画像ファイルを生成します。
`DIB`	装置非依存ビットマップファイルを生成します。
`PCX`	`PCX` 画像ファイルを生成します。

`strFileName` 引数は生成する画像ファイルの名前です。`strFileName` が相対パスで表されている場合は、クライアントアプリケーションの現在の作業ディレクトリからの相対パスとして解釈されます。

戻り値

このコマンドには戻り値はありません。

参照

Tcl コマンドへの COM のマッピング

シンボルコマンド

シンボルは、要素を二次元グラフィック表示したものです。シンボルを操作する Tcl コマンドと同じ COM API メソッドが ITtdSymbol セクションに用意されています。すべてのシンボル操作 Tcl コマンドの第一引数は、操作するシンボルへの参照です。

注記

シンボルは PresentationElement であり Entity でもあるので、すべての プレゼンテーション要素コマンド とすべての エンティティ コマンド はシンボルに対して使用できます。

コマンド	説明
u2::SetSize	シンボルのサイズを設定します。
u2::SetPosition	シンボルの位置を設定します。

参照

Tcl コマンドへの COM のマッピング

式コマンド

式はモデルのさまざまな場所で使用されます。式を操作する Tcl コマンドと同じ COM API メソッドが ITtdExpression セクションに用意されています。すべてのシンボル操作 Tcl コマンドの第一引数は、操作するシンボルへの式です。

注記

式は Entity でもあるので、すべてのエンティティ コマンド は式に対して使用できます。

コマンド	説明
u2::GetType	式の型を算出します。
u2::EvaluateConstantIntegralExpression	定数式の値を評価します。
u2::GetInstanceChildExpression	あるインスタンスの割り当て式の右側値を取得します。

参照

Tcl コマンドへの COM のマッピング

ライブラリ ハンドリング コマンド

ライブラリ ハンドリング パートには、以下のコマンドが含まれます。

コマンド	説明
u2::LoadLibrary	UML ライブラリをロードします。
u2::UnloadLibrary	UML ライブラリをアンロードします。
u2::LoadProfile	UML プロファイルをロードします。
u2::UnloadProfile	UML プロファイルをアンロードします。

参照

LoadFile

u2::LoadLibrary

UML ライブラリをロードします。

表記

u2::LoadLibrary *filename*

説明

このコマンドは、UML ライブラリを Tau にロードします。ライブラリは、引数 *filename* で指定します。UML ライブラリが含まれている .u2 ファイルのパスを指定する文字列を使用します。

参照

u2::UnloadLibrary

u2::UnloadLibrary

UML ライブラリをアンロードします。

表記

u2::UnloadLibrary *packageRef*

説明

このコマンドは、ロードされた UML ライブラリをアンロードします。ライブラリは、UML ライブラリ パッケージの参照が含まれている必要がある引数 *packageRef* で指定します。

-deleteDependencies オプションがあると、ライブラリパッケージの最上位の依存も削除されます。この動作は、<<access>> 依存が自動的に追加されるため、アンロードされるライブラリがプロファイルの場合には有用です。ただし、パッケージを参照するすべての最上位の依存は、ユーザーが手動で追加した場合でも、削除されることに注意が必要です。

例 708:

ロードされたライブラリは、アドインの非活動化時に呼ばれる特別のプロシージャ BeforeUnload でアンロードできます。以下のように、ライブラリはロードされているモデルごとに 1 回づつアンロードする必要があります。

```
proc BeforeUnload { } {
  set models [std::GetModels -kind U2]
```

```
    foreach model $models {
      set profile [u2::FindByGuid $model "@MyProfile"]
      u2::UnloadLibrary $profile
    }
  }
```

---

u2::LoadProfile

UML プロファイルをロードします。

表記

`u2::LoadProfile` *`filename`*

説明

代わりに `LoadLibrary` を使用します。`LoadProfile` は廃止が決定された関数なので、将来のバージョンで削除されます。

このコマンドは、UML プロファイルを Tau にロードします。プロファイルは、引数 *filename* で指定します。UML プロファイルが含まれている `.u2` ファイルのパスを指定する文字列を使用します。

参照

u2::UnloadProfile

u2::UnloadProfile

UML プロファイルをアンロードします。

表記

`u2::UnloadProfile` *`packageRef`*

説明

代わりに `UnLoadLibrary` を使用します。`UnLoadProfile` は廃止が決定された関数なので、将来のバージョンで削除されます。

このコマンドは、ロードされた UML プロファイルをアンロードします。プロファイルは、UML プロファイル パッケージの参照が含まれている必要がある引数 *packageRef* で指定します。

例 709:

ロードされたプロファイルは、アドインの非活動化時に呼び出される特殊なプロシージャ `BeforeUnload` でアンロードできます。プロファイルは、以下のようにロードされているモデルごとに 1 回アンロードする必要があります。

```
proc BeforeUnload { } {
  set models [std::GetModels -kind U2]

  foreach model $models {
    set profile [u2::FindByGuid $model "@MyProfile"]
    u2::UnloadProfile $profile
  }
}
```

セマンティック チェッカ コマンド

セマンティック チェッカ コマンドを使用して、ユーザー定義 UML セマンティック チェックを内蔵セマンティック チェック セットに追加できます。代表的なシナリオでは、UML ライブラリで定義されている制限をテストするセマンティック チェックを追加します。

以下のセマンティック チェッカ コマンドが用意されています。

コマンド	説明
u2::Check	モデルをセマンティックにチェックします。
u2::CreateSemGroup	新しいセマンティック チェッカ グループを作成します。
u2::CreateSemRule	新しいセマンティック チェッカ ルールを作成します。
u2::DeleteSemEntity	セマンティック チェッカ グループまたはルールを削除します。
u2::EnableSemEntity	セマンティック エンティティを有効または無効にします。
u2::GetSemEntities	グループ内のエンティティの名前を返します。
u2::IsSemEntityEnabled	セマンティック エンティティのステータスを返します。
u2::IsSemGroup	エンティティがセマンティック グループかどうかをチェックします。
u2::QuickCheck	モデルをセマンティックにチェックします。
u2::SemMessage	セマンティック チェッカ エラーを報告します。

u2::Check

モデルをセマンティックにチェックします。

表記

u2::Check *entityRef*

説明

このコマンドは、モデルまたはエンティティの参照になっている必要がある *entityRef* 引数で指定されたモデル要素の階層に対して完全なセマンティック チェックを実行します。

例 710:

以下の例は、現在ロードされているすべてのモデル上でチェックを実行する方法を示しています。

```
set models [std::GetModels]

foreach m $models {
  u2::Check $m
}
```

u2::CreateSemGroup

新しいセマンティック チェッカ グループを作成します。

表記

u2::CreateSemGroup *path name*

説明

このコマンドは、*name* 引数によって指定された名前で新しいセマンティック チェッカ グループを作成します。

path 引数は、セマンティック チェッカ グループ階層内のグループの場所を定義する文字列です。文字列の先頭にルート グループを示す「/」を付けて、その後にすべて「/」で区切られた任意の数の親グループ候補を指定する必要があります。*path* 内の親グループが存在しない場合は、Tcl エラーが生成されます。グループがすでに存在する場合、エラーは報告されません。

参照

u2::CreateSemRule

u2::CreateSemRule

新しいセマンティック チェッカ ルールを作成します。

表記

```
u2::CreateSemRule path name metaclass priority script
```

説明

このコマンドは新しいセマンティック チェッカ ルールを定義します。引数 *path* を使用して、セマンティック チェック グループ階層内の新しいルールの場所を指定し、*name* を使用して、ルールの名前を設定します。

metaclass 引数は、新しいルールをそのメタクラスのエンティティ上のみで呼び出すことを決定します。

priority は、他のルールに対する当該ルールの優先順位を設定します。

script 引数は、ルールの実行時に呼び出す Tcl スクリプトの名前です。モデル エンティティの Tcl id は、スクリプトの終わりに追加されます。したがって、スクリプトに戻りパラメータを使用することを推奨します。*script* は、変換プロセスで起動元になるモデル エンティティを削除する場合、値「1」または「true」を返す必要があります。これにより、セマンティック分析ではこのエンティティ上で他のルールを呼び出さないようにできます。モデル エンティティの削除後に「1」または「true」を返せないと、プログラムは正しく動作しなくなる可能性があります。

u2::DeleteSemEntity

セマンティック チェッカ グループまたはルールを削除します。

表記

```
u2::DeleteSemEntity path name
```

説明

このコマンドは、*path* によって指定されたとおりのセマンティック チェッカ グループ階層内の *name* というセマンティック チェッカ グループを削除します。**定義済みの**ルールをこのコマンドで削除することはできません。

u2::EnableSemEntity

セマンティック エンティティを有効または無効にします。

表記

```
u2::EnableSemEntity path status
```

説明

path 引数は、セマンティック チェッカ グループ階層内のエンティティの場所を定義する文字列です。

status は、以下のようにエンティティの目的のステータスを設定する文字列引数です。

- **有効**
- **無効**

u2::GetSemEntities

グループ内のエンティティの名前を返します。

表記

```
u2::GetSemEntities path
```

説明

コマンド **GetSemEntities** は、特定のグループ内のセマンティック サブエンティティの名前が含まれているリストを返します。

path 引数は、セマンティック グループの UNIX スタイルのパスです。たとえば、ルート グループのパスは「/」、UML グループ（ルート グループ内にある）のパスは「/UML」になります。

この名前付け規則があるため、セマンティック エンティティの名前には、**スラッシュ** (/) を使用できません。これは、スラッシュがパスの区切り文字と見なされるためです。スラッシュ文字に対応するエスケープ文字はありません。セマンティック チェッカでは常にスラッシュをセパレータとして扱います。

u2::IsSemEntityEnabled

セマンティック エンティティの *ステータス* を返します。

表記

```
u2::IsSemEntityEnabled path
```

説明

path によって指定されたセマンティック エンティティの *ステータス* を返します。

path 引数は、セマンティック チェッカ グループ階層内のエンティティの場所を定義する文字列です。

status は、以下のようにエンティティのステータスを示す文字列です。

- **有効**
- **無効**

u2::IsSemGroup

エンティティがセマンティック グループかどうかをチェックします。

表記

```
u2::IsSemGroup path
```

説明

path によって指定されたセマンティック エンティティが、セマンティック グループか（true）、セマンティック グループでないか（false）によって、**true** または **false** を返します。

path 引数は、セマンティック チェッカ グループ階層内のエンティティの場所を定義する文字列です。

u2::QuickCheck

モデルをセマンティックにチェックします。

表記

```
u2::QuickCheck entityRef
```

説明

このコマンドは、モデルまたはエンティティの参照になっている必要がある *entityRef* 引数で指定されたモデル要素の階層に対して完全なセマンティック チェックを実行します。標準のエラー リスト（u2::Check の場合と同じ）が使用されます。このコマンドは、セマンティック チェックのサブセットを実行し、起動元のエンティティに対してのみ動作します（合成子はチェックされません）。

u2::SemMessage

セマンティック チェッカ エラーを報告します。

表記

```
u2::SemMessage severity message entityId
```

説明

このコマンドは、Semantic Analyzer によって使用されるエラー リストにエラー メッセージを入れます。この関数を使用して、チェック時にエラー メッセージ、警告メッセージ、情報メッセージを報告する必要があります。報告しない場合、ユーザー定義チェックによって生成されたエラー後にセマンティック分析は停止しません。

severity は、以下の値を使用できる文字列です。

- **error**
  エラー メッセージが報告されます。
- **fatal**
  重大エラー メッセージが報告されます。
- **information**
- **warning**
  警告メッセージが報告されます。

message は、エラー リストに報告されるメッセージが含まれる文字列です。

entityId は、メッセージが関連するモデル要素のオプションの参照です。

参照

u2::CreateSemRule

ユーティリティインターフェイスコマンド

Tau の C++ および COM API は、Tcl API とは対照的に、複数のインターフェイスに基づいています。これらのインターフェイスの大半は Tau メタモデルのメタクラス（モデルインターフェイス）で実装されています。つまり、このインターフェイスに定義された関数には、上で述べたグループに属する、対応する Tcl コマンドがあります。ただし、どのメタクラス（ユーティリティインターフェイス）によっても実装されていない API インターフェイスもあります。これらは、Tau IDE の他のオブジェクトによって実装されています。Tcl からのこのような機能にアクセスするには、特定のインターフェイスの関数に対応した Tcl コマンドを使用します。

このセクションでは、ユーティリティインターフェイスの関数に対応する Tcl コマンドを説明します。

コマンド	説明
u2::AddSourceBufferText	テキストをソースコードのバッファに追加します。
u2::AddMessage	メッセージリストにメッセージを追加します。

u2::AddSourceBufferText

テキストをソースコードのバッファに追加します。

このコマンドは ITtdSourceBuffer インターフェイスの AddText API 関数に対応します。

表記

```
u2::AddSourceBufferText ITtdSourceBuffer text
```

説明

このコマンドはテキストバッファにテキストを追加します。通常は、このバッファは、コードジェネレータで生成されたソースコードのファイルを表現しています。

ITtdSourceBuffer 参照が、対話的なツールイベントでトリガされたエージェントのパラメータとして取得されます。

例 711: Java ファイルの ' ヘッダ ' 情報を生成する

JavaPrintFile ツールイベントでトリガされる Tcl エージェントの実装サンプルは以下のとおりです。

```
proc AddJavaHeader { triggeredBy timing context server
agentParameters } {
  upvar 1 $agentParameters ap
  u2::AddSourceBufferText [lindex $ap 0] "This is a generated
file! Do not edit!"
}
```

u2::AddMessage

メッセージリストにメッセージを追加します。

このコマンドは ITtdMessageList インターフェイスの API 関数 AddMessage に対応します。

表記

u2::AddMessage *ITtdMessageList text severity ?-subject ITtdEntity?*

説明

このコマンドは、メッセージリストにメッセージを追加します。メッセージリストは、通常はエージェントのパラメータとして取得でき、セマンティックチェック中またはコード生成中に出力されたメッセージなどを表現します。

重大度パラメータは以下のいずれかの値をとります。

- **information**
  メッセージがエラーではない情報の伝達用であることを示します。
- **warning**
  メッセージが警告であることを示します。
- **error**
  メッセージがエラーであることを示します。
- **fatal**
  メッセージが致命的なエラーであることを示します。

-subject スイッチを使用すると、モデルエンティティをメッセージの主としてアタッチできます。サブジェクトエンティティのあるメッセージは、出力タブ上でダブルクリックすると、そのエンティティへナビゲートされます。

例 712: Tcl によるカスタムセマンティックチェックの実装

Check ツールイベントにトリガされる Tcl エージェントのサンプル実装は以下のとおりです。

```
proc MyTclCheck { triggeredBy timing context server
agentParameters } {
  upvar 1 $agentParameters ap
  if {[u2::IsKindOf $context "Class"] } {
    if {[u2::GetEntity $context "Destructor"] == 0 } {
      u2::AddMessage [lindex $ap 0] "A class should have a
destructor." "warning" -subject $context
    }
  }
}
```

60

C++ API

この章は、Tau C++ API のリファレンス ガイドです。機能的には C++API は COM API によく似ています。したがって、説明の多くで COM API の説明を参照します。

対象読者は、C++ API を使用して UML モデルにアクセスするクライアント アプリケーションの開発者です。クライアント アプリケーションには、小規模の対話的な処理用のアドインから本格的なコード ジェネレータ、さらにはインポート アプリケーションに至るまでのあらゆるアプリケーションが含まれます。この章全体を通して、C++ の基本知識が前提になります。

概要

C++ API は、多数のメソッドをもったインターフェイス（抽象 C++ クラス）のグループから構成されています。API のデザインは、COM API とよく似ています。実際、COM API は、C++ API と関連しながら実装されています。

C++ API のクライアントは、さまざまな環境で実行できます。一部の API 関数の振る舞いは、クライアント コードの実行環境によって異なります。たとえば、クライアントが IDE で実行される（対話型 クライアント）場合は、モデルを変更する API 関数を実行する際に、対話型使用（元に戻す／やり直し機能など）に適した実装を呼び出します。一方、バッチアプリケーションで実行されるクライアント（非対話型クライアント）の場合は、IDE との連携が必要になるような一部の API が使用できなくなります。

API のアクセス

C++ API を使用するクライアントでは、ファイル `U2ModelAccess.h` をインクルードする必要があります。このファイルは、インストール ディレクトリの `/include/ToolAPI` にあります。また、クライアントアプリケーションは、Tau インストール ディレクトリ内の対応するライブラリとリンクする必要があります。C++ API を使用するために必要なプラットフォーム固有の手順の詳細については、1981 ページの「C++ API のセットアップ」を参照してください。

クライアントを正しく設定すると、API にアクセスできます。API にアクセスする方法は、クライアントが対話型か非対話型かによって異なります。

非対話型 クライアントから API へのアクセス

非対話型 クライアントでは、`u2::GetModelAccess` 関数を呼び出して C++ API にアクセスします。この関数は、`u2::ITtdModelAccess` インターフェイスを実装した singleton オブジェクトのポインタを返します。このインターフェイスの詳細については、1960 ページの「ITtdModelAccess」を参照してください。

注記

非対話型 クライアントから API を使い始めるには、その前に API を初期化することが重要です。初期化するには、`u2::InitializeModelAccess` を呼び出します。最後の API 呼び出しの後は、`u2::FinalizeModelAccess` を呼び出して、API をファイナライズする必要もあります。初期化を行った場合は、必ずファイナライゼーションを行います。

例 **713**

API の初期化、`u2::ITtdModelAccess` インターフェイス上のポインタの取得、その後の API のファイナライズ

```
u2::InitializeModelAccess();
u2::ITtdModelAccess* pMA = u2::GetModelAccess();
u2::FinalizeModelAccess();
```

対話型 クライアントから **API** へのアクセス

対話型 C++ API のクライアントは、通常、`u2::ITtdEntity` インターフェイス上の実行コンテキストを表すポインタを受け取るエージェントです。このポインタは、そのコンテキストからモデルにアクセスする開始点として使用できます。したがって、対話型 クライアントでは、必ずしも `u2::ITtdModelAccess` インターフェイスを介して API にアクセスする必要はありません。ただし、`u2::GetModelAccess` 関数は対話型 クライアントでも使用できます。この場合、この関数が返すオブジェクトは、対話型実行環境で使用に適したものになっています。

対話型 クライアントの場合、API の初期化とファイナライゼーションは必要ありません。

対話型 C++ クライアントを使用すると、Tcl スクリプトを使用する場合よりも効率良くアドイン モジュールを実装できます。ただし、現在の仕様では、C++ クライアントのみによるアドイン開発はサポートされていません。したがって、最低限の Tcl スクリプトを作成する必要があります。この Tcl スクリプトでは、C++ で実装されているエージェントに実行を移すために、Tcl API コマンド `InvokeAgent` を使用します。

参照

アドイン機能のどの部分を Tcl で開発し、どの部分を C++ で開発するかを決定ためのガイドラインについては、1800 ページの「対話型 クライアントから API へのアクセス」を参照してください。

重要！

対話型 C++ クライアントは Tau と同じメモリー空間で実行されます。したがって、このクライアントがクラッシュすると、Tau もクラッシュします。開発途中の C++ エージェントを実行する場合は、実行前にすべての変更を保存するのを忘れないでください。

チェンジャ オブジェクト

モデルを変更する API 関数は、`Cu2Changer`（`u2dll` 名前空間で定義）クラス型の「changer」引数をとります。このクラスの目的は、対話型環境と非対話型環境の両方で、モデルの変更が正しく機能するようにすることです。（対話型環境では「元に戻す」操作と「やり直し」操作のために、変更を「記録」する必要があります。非対話型環境では、この記録を行わずに変更を行うことができます）`Cu2Changer` の定義は、API で公開されず、前方宣言のみが存在します。公開されない理由は、クライアント コードでこのクラスを明示的に使用することが許されないからです。すべての「changer」引数は、未指定でもかまいません。指定しない場合は、デフォルトで、操作の記録を行わず変更をすぐに実行するチェンジャ オブジェクトになります。エージェントなどの対話型 クライアントは、Tau IDE から起動されると、`Cu2Changer` オブジェクトを受け取ります。モデル変更を「元に戻す」操作と「やり直し」操作ができるようにして行うには、C++ API 関数を呼び出す際に、チェンジャ オブジェクトを明示的に使用する必要があります。

インターフェイス キャスティング

ポインタを 1 つのインターフェイスから別のインターフェイスに動的にキャストするメカニズムは、`QueryInterface` という仮想関数に基づいています。この関数は、`u2::IUnknown` インターフェイスで定義されています。すべてのインターフェイスは `u2::IUnknown` を継承するため、この関数は、すべてのインターフェイスで使用できます。

注記

C++ API は、インターフェイス キャスティングのデザインと用語について、COM のものを流用していました。そのため、名前として QueryInterface と IUnknown が使用されています。COM IUnkown インターフェイスと異なり、u2::IUnknown では参照カウントを使用しません。つまり、インターフェイス ポインタ上で AddRef または Release を呼び出す必要はありません。

C++ API には、便利なユーティリティ テンプレート関数 `u2::cast` が用意されています。このユーティリティを使用すると、使い慣れた C++ キャスト構文とセマンティックを使用できます。

例 **714**

`u2::cast` を使用したインターフェイス間の動的キャスティング

```
u2::ITtdModelAccess* pMA = u2::GetModelAccess();
u2::ITtdModel* pModel = pMA->LoadFile(_T("x.u2"));
u2::ITtdEntity* pEntity = u2::cast<u2::ITtdEntity>(pModel);
if (pEntity) {
// Casting succeeded! you may safely use pEntity here!
}
```

注記

標準の `dynamic_cast` 演算子を使用してインターフェイス キャスティングを行わないでください。Tau オブジェクト モデルには、標準の C++ ランタイム型の情報（RTTI）は含まれていません。

API エラーの処理

C++ API は、`u2::APIError` 例外をスローして、すべてのエラーを報告します。クライアント コードでは、この例外を捉えてエラーに備える必要があります。エラーの説明は、`u2::GetAPIErrorMessage` 関数を使用して取得できます。

例 **715**

C++ API エラーを処理する典型的な catch 句は、以下のとおりです。

```
try {
// access the C++ API
}
catch (u2::APIError e) {
  cout << "Error while using the Tau Developer C++ API:" << endl;
  wcout << u2::GetAPIErrorMessage(e) << endl;
}
```

Unix では、`GetAPIErrorMessage` はシングルバイト文字列を返すので、`wcout` ではなく `cout` を使用します。

クライアント制限

C++ API では、クライアントに課される制限はほとんどありません。ただし、API クライアントを設計するときは、以下の点に注意してください。

ベアのみ

API は、複数のスレッドからのモデルの安全な同時アクセスをサポートしていません。対話型 クライアントは、Tau アプリケーションのメイン スレッドで実行されます。

一貫した文字列のエンコード

C++ API 関数に渡されるか、または C++ API 関数の結果として取得される文字列は、`tstring` 型によって型指定されます。このデータ型は、Windows プラットフォーム上ではワイド（2 バイト）の Unicode 文字列、Unix プラットフォーム上ではナロー（1 バイト）の ANSI 文字列となります。API クライアントは、文字列の入出力時の変換を正しく実行する責任があります。

API のインターフェイスと関数

C++ API のすべてのインターフェイスは、`u2` 名前空間にあります。このセクションでは、すべての API インターフェイスと関数について簡単に説明します。また、代表的な使用例を提供します。ただし、特定の API 関数またはインターフェイスの使用例についてその一部を示しているだけで完全ではないことに注意してください。特に、本来ならば推奨されるエラー処理（1958 ページの「API エラーの処理」を参照）の多くが、例をシンプルで理解しやすくするために省略されています。

C++ インターフェイスとそのメンバー関数の名前は、対応する COM API のインターフェイスとメソッドに一致しています。したがって、C++ API の詳細な解説と例については、COM API の対応する項を参照してください。

多くのインターフェイスは、UML メタモデルの実装である「メタクラス」を表現するクラスによって実装されます。したがって、C++ API の一部のメソッドではメタモデルの知識が必要になります。

注記

この項の記述では、「C++ インターフェイス」は「抽象クラス」を意味します。つまり、そのメンバー関数のすべてが純仮想関数になるクラスを意味します。

ITtdModelAccess

ITtdModelAccess インターフェイスには、モデルに直接作用しない関数があります。このインターフェイスを非対話型 クライアントから使用し、ファイル（プロジェクト ファイルまたは `..u2` ファイル）からロードした UML モデルにアクセスするか、新規のモデルを作成します。また、対話型 クライアントでは、このインターフェイスを使用して Tau IDE の特定部分にアクセスできます。

クライアント アプリケーションから `ITtdModelAccess` インターフェイスを取得する方法については、1956 ページの「API のアクセス」を参照してください。

`ITtdModelAccess` については、COM API の章で詳しく説明しています。

LoadProject

```
virtual u2::ITtdModel*
LoadProject(const tstring& strProject,
  bool bBind = true,
  ITtdMessageList* pMessages = 0)
throw(u2::APIError) = 0;
```

指定されたプロジェクト ファイル（または URI）に格納されているプロジェクトをロードします。プロジェクト ファイルが存在しないか、プロジェクトのロード中にエラーが生じた場合は、APIError 例外が発生します。`strProject` が相対パス指定の場合、クライアント アプリケーションの現在の作業ディレクトリが基準になります。

デフォルトでは、モデルはロード後にバインドされますが、bBind を ”false” に設定することにより、このバインドを止めることができます。

メッセージ リストが関数に渡されると、ロード時とバインド時に作成されるすべてのメッセージがそのリストに加えられます。メッセージ リストは、GetMessageList を呼び出すことによって取得できます。

注記

パラメータ `bBind' および `pMessages' は、対話型 エージェントから使用した場合は効果がありません。

例 **716**

非対話型 クライアントからのプロジェクトのロード

```
u2::ITtdModelAccess* pMA = u2::GetModelAccess();
u2::ITtdModel* pModel = pMA->LoadProject(_T("x.ttp"));
```

ロード メッセージとバインド メッセージの印刷のための、プロジェクトのロードとメッセージ リストの使用 :

```
u2::ITtdMessageList* pMessages = pMA->GetMessageList();
u2::ITtdModel* pModel = pMA->LoadProject(_T("y.ttp"), true /* bBind
*/, pMessages);
tstring strErrors;
pMessages->GetDescription(strErrors, _T("¥n"));
pMA->WriteMessage(strErrors);
delete pMessages;
```

LoadFile

```
virtual u2::ITtdModel*
LoadFile(const tstring& strFile,
  bool bLibrary = false)
throw(u2::APIError) = 0;
```

指定された .u2 ファイル（または URI）をロードします。ファイルが存在しないか、プロジェクトのロード中にエラーが生じた場合は、APIError 例外が発生します。strFile が相対パス指定の場合、クライアント アプリケーションの現在の作業ディレクトリが基準になります。bLibrary が ”true” の場合は、ファイルがライブラリとしてロードされます。

LoadFile の使用法の例については、1958 ページの例 714 を参照してください。

CreateModel

```
virtual u2::ITtdModel*
CreateModel()
throw(u2::APIError) = 0;
```

新しい空のモデルを作成します。新しいモデルを作成できない場合（メモリやライセンス問題などにより）、APIError 例外が発生します。

作成された空のモデルは、新しい UML モデルを作成するために使用できます。以下の例では、1 つのパッケージに 1 つのクラスを含む単純なモデルを作成して、その新しいモデルを保存する方法を示しています。

例 **717**

新規モデルの作成とモデルの .u2 ファイルへの保存：

```
u2::ITtdModel* pModel = pModelAccess->CreateModel();
u2::ITtdEntity* pModelRoot = u2::cast<u2::ITtdEntity>(pModel);
u2::ITtdEntity* pPackage = pModelRoot->Create(_T("Package"), false,
_T("OwnedMember"));
u2::ITtdEntity* pClass = pPackage->Create(_T("Class"));
u2::ITtdResource* pResource;
pResource = pModel->CreateResource(_T("test.u2"));
// Insert the created package pPackage as a root of pResource
u2::cast<u2::ITtdEntity>(pResource)->SetEntity(_T("Root"),
pPackage);
pModel->Save(); // Saves all resources in the model
```

GetActiveProject

```
virtual ITtdModel*
GetActiveProject()
throw(u2::APIError) = 0;
```

開かれているワークスペース内で現在アクティブなプロジェクトのモデルを返します。この関数は、対話型 API クライアントからのみ使用されるものとします。この関数を非対話型 クライアントから使用すると、APIError が発生します。

GetProjectItem

```
virtual ITtdModel*
GetProjectItem(long index)
throw(u2::APIError) = 0;
```

開かれているワークスペース内で指定されたインデックスを持つプロジェクトのモデルを返します。この関数は、対話型 API クライアントからのみ使用されるものとします。この関数を非対話型 クライアントから使用すると、APIError が発生します。

GetProjectCount

```
virtual long
GetProjectCount()
throw(u2::APIError) = 0;
```

開かれているワークスペース内のプロジェクトの数を返します。この関数は、対話型 API クライアントからのみ使用されるものとします。この関数を非対話型 クライアントから使用すると、APIError が発生します。

WriteMessage

```
virtual void
WriteMessage(const tstring& strMessage)
throw(u2::APIError) = 0;
```

デフォルトのメッセージ領域（通常は、バッチ環境内の `stdout`、または IDE 内の［メッセージ］タブ）にメッセージを書き込みます。

`WriteMessage` の使用例については、1960 ページの例 716 を参照してください。

GetMessageList

```
virtual ITtdMessageList*
GetMessageList() = 0;
```

新しいメッセージ リストを作成して返します。このメッセージ リストは、ITtdMessageList へのメッセージのレポートをサポートする関数への入力として使用できます。メッセージ リストを作成できない場合、NULL が返されます。

使用しなくなったメッセージ リストの削除は、呼び出し側の責任で行います。

`GetMessageList` の使用例については、1960 ページの例 716 を参照してください。

GetDefaultMessageList

```
virtual ITtdMessageList*
GetDefaultMessageList() = 0;
```

現在の実行環境に合ったデフォルトのメッセージリストを返します。たとえば、コードジェネレータの実行環境では返されるメッセージリストはコードジェネレータのビルドログにマップされます。

返されるメッセージリストは、Tau の管理下にあるため、呼び出し元による削除は許されません。

GetLicense

```
virtual void
GetLicense(const tstring& strFeature)
throw (u2::APIError) = 0;
```

指定した名前のライセンスフィーチャのためのランタイムライセンスを要求します。成功した場合は、ライセンスが取得されます。不成功の場合は、APIError 例外が発生します。

この機能はライセンスを必要とする API クライアントでの使用を意図したものです。

ITtdModel

ITtdModel インターフェイスは、UML モデルの最上位レベル エンティティを表すメタモデルの Session クラスによって実装されます。

ITtdModel インターフェイスには、実行用に特定のモデル エンティティ コンテキストを必要としないメソッドがあります。通常、これらのメソッドは、特定のエンティティではなく、モデル全体に作用します。

注記

Session は Entity でもあるので、ITtdModel から ITtdEntity へのキャストは常に成功します。

ITtdModel については、COM API の章で詳しく説明しています。

FindByGuid

```
virtual u2::ITtdEntity*
FindByGuid(const tstring& strGuid) const
throw(u2::APIError) = 0;
```

指定された GUID を持つモデル内のエンティティを返します。そのようなエンティティが存在しない場合は、NULL を返します。

FindByGuid については、COM API の章で詳しく説明しています。

New

```
virtual u2::ITtdEntity*
New(const tstring& strMetaClass) const
throw(u2::APIError) = 0;
```

指定されたメタクラスの新しいインスタンスを作成します。指定された名前のメタクラスが存在しない場合、または他の理由で何らかのエラーが起こった場合は、APIError 例外が発生します。

New については、COM API の章で詳しく説明しています。

Parse

```
virtual void
Parse(const tstring& strConcreteSyntax,
  std::list<u2::ITtdEntity*>& listEntities,
  const tstring& strParseAs = _T("Definition")) const
```

```
throw(u2::APIError) = 0;
```

テキスト UML 構文の指定された部分を解析します。オプションの後続パラメータ `strParseAs` によって、解析時に使用する文法が指定されます。デフォルトでは、Definition 文法が使用されます。つまり、テキストでは 1 つまたは複数の定義を指定する必要があります。サポートされている他の文法は、Expression と Action です。存在しない文法が指定された場合、またはテキストに構文エラーが含まれている場合は、APIError 例外が発生します。パーサが作成した結果は、リストに挿入されます。

Parse については、COM API の章で詳しく説明しています。

XMLDecode

```
virtual void
XMLDecode(const tstring& strXMLEncoding,
  std::list<u2::ITtdEntity*>& listEntities) const
throw(u2::APIError) = 0;
```

XML エンコード文字列を、一連のモデル エンティティにデコードします。デコードに失敗すると（たとえば、XML の構文エラーなどの理由で）、APIError 例外が発生します。

XMLDecode については、COM API の章で詳しく説明しています。

Save

```
virtual void
Save() const
throw(u2::APIError) = 0;
```

モデルをそのリソースに保存します。ただし、モデルにリソースが含まれていない場合は、何も実行されません。

モデルが保存できない場合は、APIError 例外が発生します。

Save については、COM API の章で詳しく説明しています。

CreateResource

```
virtual u2::ITtdResource*
CreateResource(const tstring& strFile,
  u2dll::Cu2Changer& changer = u2dll::defaultChanger)
throw(u2::APIError) = 0;
```

モデルの新しいリソースを作成します。リソースは指定されたファイル名のファイルになります。作成に失敗すると、APIError 例外が発生します。モデルの変更は、チェンジャを使用して実行されます。

CreateResource の使用例については、1961 ページの例 717 を参照してください。

CreateResource については、COM API の章で詳しく説明しています。

LoadFile

```
virtual u2::ITtdResource*
LoadFile(const tstring& strFile,
```

```
    bool bLibrary = false)
  throw(u2::APIError) = 0;
```

指定されたファイルをモデルにロードします。ファイルを表すリソースが作成されて返されます。bLibrary が ”true” の場合は、ファイルがライブラリとしてロードされます。エラー（ロード エラーなど）が発生した場合は、APIError 例外が発生します。

LoadFile については、COM API の章で詳しく説明しています。

InvokeAgent

```
  virtual void
  InvokeAgent(u2::ITtdEntity* pAgent,
    u2::ITtdEntity* pModelContext,
    u2::AgentParameters& agentParameters) const
  throw(u2::APIError) = 0;
```

指定されたモデル コンテキスト上で、指定されたエージェントを起動します。

AgentParameters 型は、起動されたエージェントに渡される実際の引数を表す一連のエージェント パラメータのリストです。

InvokeAgent については、COM API の章で詳しく説明しています。

ITtdEntity

ITtdEntity インターフェイスは、UML モデルの一般エンティティを表すメタモデルの Entity クラスによって実装されます。

ITtdEntity インターフェイスには、実行する際に、特定のモデル エンティティ コンテキストを必要とするメソッドがあります。通常、これらのメソッドは、呼び出し元のエンティティに作用します。

ApplyStereotype

```
virtual u2::ITtdEntity*
ApplyStereotype(u2::ITtdEntity* pStereotype,
  u2::TtdReferenceKind referenceKind =
TTD_RK_MINIMAL_QUALIFIER,
  u2::ITtdEntity* pInsertElement = 0,
  u2dll::Cu2Changer& changer = u2dll::defaultChanger)
throw(u2::APIError) =0;
```

与えられたステレオタイプをインスタンス化して、ホストエンティティとして参照されるエンティティに適用します。ステレオタイプへの参照で使用される修飾子は、以下の例のように referencekind に基づいて決定されます。u2::TtdReference に対して可能な値については、以下の表を参照してください。pInsertElement が与えられている場合は、ステレオタイプは論理的にはホストエンティティ上でインスタンス化され、物理的には pInsertElement 上でインスタンス化されます。この場合、ステレオタイプインスタンスはホストエンティティを指します。このように、ステレオタイプインスタンスは、エンティティそのものの変更をせずに、エンティティに適用できます。このテクニックは「Stereotype Injection」として知られています。

ステレオタイプの適用が失敗した場合は、APIError 例外が発生します。

u2::TtdReferenceKind	説明
TTD_RK_GUID	参照は GUID 参照のみを含みます。
TTD_RK_NO_QUALIFIER	参照は修飾されません。つまり、ステレオタイプ名のみを含みます。
TTD_RK_FULL_QUALIFIER	参照は完全修飾子を含みます。
TTD_RK_MINIMAL_QUALIFIER	参照はステレオタイプを参照するのに必要な最低限の修飾子を含みます。このオプションが返す最長の修飾子は、relativeQualifier です。<<access>>/<<import>> 依存がある場合は、修飾子が短くなる可能性があります。
TTD_RK_RELATIVE_QUALIFIER	参照はステレオタイプに対する相対的な修飾子になります。ステレオタイプとホストエンティティに共通の上方スコープがない場合は完全修飾子になります。それ以外の場合は、直近の共通スコープから始まる、より短い修飾子になります。

GetValue

```
virtual void
GetValue(const tstring& strMetaFeature,
  tstring& strValue,
  u2dll::eposition index = E_POSITION_END) const
throw(u2::APIError) = 0;
```

エンティティに指定されたメタ特性の値を取得します。指定された名前のメタ特性がない場合は、APIError 例外が発生します。取得される値は文字列です。このメソッドは、単一多重度または複数多重度のメタ特性に使用できます。複数多重度の場合は、index を使用して、取得する値を指定します。

GetValue 関数は、あるエンティティの、文字列値を保持できるすべてのメタ特性について使用できます。つまり、メタクラス型から派生した特性、所有者リンク、合成リンクを**除いた**、すべてのメタ特性で使用できます。

注記

index は 1 から始まるインデックス値です。E_POSITION_END (==0) は最後のエンティティを示します。

例 **718**

GetValue による定義名の取得

```
if (pEntity->IsKindOf(_T("Definition"))) {
  tstring name;
  pEntity->GetValue(_T("Name"), name);
}
```

GetValue については、COM API の章で詳しく説明しています。

GetEntity

```
virtual u2::ITtdEntity*
GetEntity(const tstring& strMetaFeature,
  u2dll::eposition index = E_POSITION_END) const
throw(u2::APIError) = 0;
```

指定されたメタ特性の値を ITtdEntity ポインタとして取得します。単一多重度または複数多重度のメタクラス型のメタ特性に使用します。複数多重度の場合は、インデックスを使用して、取得するエンティティを指定します。指定された名前のメタ特性がない場合は、APIError 例外が発生します。

注記

メタ特性がバインドされていない場合は、pValue が NULL になります。その後、GetValue 関数を使用して、値を代わりに文字列表現として取得するか、GetReference を使用して、バインドされていない参照のモデル表現を取得できます。

注記

index は 1 から始まるインデックス値です。E_POSITION_END (==0) によって常に最後のエンティティが指定されます。

GetEntity については、COM API の章で詳しく説明しています。

GetEntities

```
virtual void
GetEntities(const tstring& strMetaFeature,
  std::list<u2::ITtdEntity*>& listEntities) const
throw(u2::APIError) = 0;
```

指定されたメタ特性の値を一連のエンティティとして取得します。単一多重度または複数多重度のメタクラス型のメタ特性に使用します。単一多重度の場合は、結果リストに 1 つのエンティティがあるか、またはエンティティはありません。指定された名前のメタ特性がない場合は、APIError 例外が発生します。

GetEntities については、COM API の章で詳しく説明しています。

GetReference

```
virtual u2::ITtdEntity*
GetReference(const tstring& strMetaFeature,
  u2dll::eposition index = E_POSITION_END) const
throw(u2::APIError) = 0;
```

メタ特性参照の識別子表現を返します。複数多重度のメタ特性の場合は、インデックスを使用して、取得する参照を指定します。指定された名前のメタ特性がない場合は、APIError 例外が発生します。

注記

index は 1 から始まるインデックス値です。E_POSITION_END (==0) によって常に最後のエンティティが指定されます。

GetReference については、COM API の章で詳しく説明しています。

GetOwner

```
virtual u2::ITtdEntity*
GetOwner)() const = 0;
```

エンティティの所有者を返します。エンティティの所有者が存在しない場合は、NULL が返されます。

GetOwner については、COM API の章で詳しく説明しています。

GetMetaClassName

```
virtual void
GetMetaClassName(tstring& strMetaClassName) const = 0;
```

エンティティのメタクラスの名前を `strMetaClassName` に格納します。

GetMetaClassName については、COM API の章で詳しく説明しています。

GetReferingEntities

```
virtual void
GetReferingEntities(const tstring& strMetaFeature,
  std::list<u2::ITtdEntity*>& listReferee) const = 0;
```

指定されたメタ特性を通じてこのエンティティを参照しているエンティティを検索します。これらの参照エンティティへのポインタが、結果リストに出力されます。

任意のメタ特性からの、すべての参照エンティティを検索するには、`strMetaFeature` が空の文字列である必要があります。

GetReferringEntities については、COM API の章で詳しく説明しています。

GetTaggedValue

```
virtual u2::ITtdEntity*
GetTaggedValue(const tstring& strSelector,
  bool bIdentsAreGuids = false) const
throw(u2::APIError) = 0;
```

セレクタ パターンによって選択されたタグ付き値を表す式を返します。エンティティは、拡張可能要素（拡張可能要素の適用済みステレオタイプ インスタンスからタグ付き値を検索）またはインスタンス式（この特定インスタンスのみからタグ付き値を検索）である必要があります。

セレクタパターンは、インスタンス式のテキスト UML 構文を使用して、エンティティからタグ付き値までのパスを指定します。関数は、パターンがインスタンス ツリー（ストラクチャによる）と対応するシグニチャ（名前または GUID による）の両方に一致しているかどうかをチェックします。`bIdentsAreGuids` が ”true” の場合は、パターンの識別子が GUID として解釈されます。それ以外の場合は、名前として解釈されます。セレクタ パラメータによってタグ付き値が選択されなかった場合は、NULL が返されます。

セレクタ パターンの一例は以下のとおりです。

```
"T1 (. .)"
```

上のパターンは、適用された T1 ステレオタイプの最初のインスタンスを返します

```
"T1 (. x .)"
```

上のパターンは、T1 ステレオタイプ内の属性 x のタグ付き値を返します

```
"T1 (. a1 = T2 (. a2 .) .)"
```

上のパターンは、値が T1.a1 である T2 インスタンス内の a2 の値を返します

注記

ステレオタイプ X がエンティティに適用されているかをテストするために、"X (. .)" 形式のセレクタ パターンを使用することもできますが、テストには HasAppliedStereotype を使用した方がより効果的です。これは、一致するメタクラスの必須拡張により自動的に適用されたステレオタイプが、少なくとも属性の 1 つがデフォルトと異なるタグ付き値を取得するまで、インスタンス化されないためです。この場合には、GetTaggedValue はステレオタイプ インスタンスを返しません。ただし、GetTaggedValue が、タグ付き値を選択するセレクタ パターンと同時に使用される場合は、自動的に適用されるステレオタイプについても返されるようになります。

GetTaggedValue については、COM API の章で詳しく説明しています。

HasAppliedStereotype

```
virtual bool
HasAppliedStereotype(const tstring& strStereotype,
  bool bGuid = false) const = 0;
```

エンティティに適用された特定のステレオタイプがあるかどうか判断します。ステレオタイプは名前、または GUID で指定できます。GUID で指定する場合、bGuid は ”true” に設定します。エンティティに適用されたステレオタイプのチェック用に推奨される関数です。この関数は、明示的に適用されたステレオタイプと、エンティティのメタクラスと一致するメタクラスの必須拡張により自動的に適用されたステレオタイプの両方を返します。

IsKindOf

```
virtual bool
IsKindOf(const tstring& strMetaClass) const = 0;
```

エンティティが指定されたメタクラスになっている場合は ”true”、それ以外の場合は ”false” を返します。

`IsKindOf` の使用法の例については、1966 ページの例 718 を参照してください。

IsKindOf については、COM API の章で詳しく説明しています。

Unparse

```
virtual void
Unparse(tstring& strText) const
throw(u2::APIError) = 0;
```

テキスト UML 構文を使用して、エンティティを `strText` に逆構文解析します。以下の種類のエンティティを逆構文解析できます。定義、アクション、式。他の種類のエンティティを逆構文解析しようとすると、APIError 例外になります。

Unparse については、COM API の章で詳しく説明しています。

SetValue

```
virtual void
SetValue(const tstring& strMetaFeature,
  const tstring& strValue,
  u2dll::eposition index = E_POSITION_END,
  u2dll::Cu2Changer& changer = u2dll::defaultChanger)
throw(u2::APIError) = 0;
```

メタ特性の値を設定します。値は文字列です。

SetValue 関数は、あるエンティティの、文字列値を保持できるすべての書き込み可能メタ特性に対して使用できます。つまり、派生した特性（読み取り専用）、所有者リンク、合成リンクを**除いた**すべてのメタ特性で使用できます。

多重度が 1 ではないメタ特性の場合は、インデックス引数を使用して、位置を指定できます。

メタ特性が存在しないなどの理由でエラーが発生すると、APIError 例外が発生します。

SetValue については、COM API の章で詳しく説明しています。

SetEntity

```
virtual void
SetEntity(const tstring& strMetaFeature,
  u2::ITtdEntity* pValue,
  u2dll::eposition index = E_POSITION_END,
  u2dll::Cu2Changer& changer = u2dll::defaultChanger)
throw(u2::APIError) = 0;
```

メタ特性の値を設定します。値はエンティティ ポインタです。このメソッドは、メタクラス型のすべての書き込み可能メタ特性に使用できます。多重度が 1 ではないメタ特性の場合は、インデックス引数を使用し、その値の前の位置にエンティティを挿入できます。

注記

pValue が NULL の場合は、メタ特性のバインドが解除されます。ただし、この解除は所有者リンクにはあてはまりません。所有者リンクを解除する、つまりエンティティをコンポジション所有者からリンク解除するには、UnlinkFromOwner を使用します。

メタ特性が存在しないなどの理由でエラーが発生すると、APIError 例外が発生します。

SetEntity については、COM API の章で詳しく説明しています。

SetTaggedValue

```
virtual void
SetTaggedValue(const tstring& strSelector,
  const tstring& strValue,
  bool bOverwrite = true,
  u2dll::Cu2Changer& changer = u2dll::defaultChanger)
throw(u2::APIError) = 0;
```

セレクタ パターンによって選択される属性のタグ付き値を設定します。セレクタパターンの形式については、GetTaggedValue の説明を参照してください。エンティティは、適用済みステレオタイプがある要素、またはインスタンス式です。前者の場合は、最初に一致する適用済みステレオタイプ インスタンスが使用され、後者の場合は、インスタンス式に対してパターンが一致照合されます。

デフォルトでは、選択された属性の既存の値が上書きされますが、`bOverwrite` が ”false” の場合は、上書きされません。

注記

指定された属性の既存の値を上書きできるようにするには、インスタンス式をインスタンスの元になるシグニチャにバインドする必要があります。そうなっていない場合は、APIError 例外が発生します。

新しい値が設定されると、この関数の呼び出し後に、設定された値に GetTaggedValue から直接アクセスできるように、エンティティ内のすべての参照のバインドが試みられます。

モデルの変更は、チェンジャを使用して実行されます。

SetTaggedValue については、COM API の章で詳しく説明しています。

Create

```
virtual u2::ITtdEntity*
Create(const tstring& strMetaClass,
  bool bBuildModelForPresentations = true,
  const tstring& strMetaFeature = _T(""),
  u2dll::Cu2Changer& changer = u2dll::defaultChanger)
throw(u2::APIError) = 0;
```

指定されたメタクラスのエンティティを、このエンティティの子として作成します。IDE 実行環境では、この関数は COM API の Create と同じです。IDE 実行環境でない場合は、この関数は指定されたメタクラスのインスタンスを作成し、そのインスタンスをエンティティの直接の子として、指定されたメタ特性に挿入しようとします。その場合、プレゼンテーション要素のモデル要素は自動的には作成されません。

エンティティが、作成されるエンティティを含むことのできるメタ特性のみを持つ場合は、メタ特性は指定せずに空の文字列にします。

モデルの変更は、チェンジャを使用して実行されます。

Create については、COM API の章で詳しく説明しています。

CreateInstance

```
virtual u2::ITtdEntity*
CreateInstance() const
throw(u2::APIError) = 0;
```

インスタンス化できるシグニチャであるエンティティ上でこの関数を呼び出して、そのシグニチャのインスタンスを作成します。一般にこの関数は、あるステレオタイプのインスタンスを作成して、そのステレオタイプをある要素に適用するときに使用します。作成できないと、APIError 例外が発生します。

CreateInstance については、COM API の章で詳しく説明しています。

Delete

```
virtual void
Delete(u2dll::Cu2Changer& changer = u2dll::defaultChanger) =
0;
```

エンティティを削除します。モデルの変更は、チェンジャを使用して実行されます。

Delete については、COM API の章で詳しく説明しています。

XMLEncode

```
virtual void
XMLEncode(tstring& strXMLEncoding) const
throw(u2::APIError) = 0;
```

エンティティの XML エンコーディングを `strXMLEncoding` に格納します。エラーが起こると、APIError 例外が発生します。

XMLEncode については、COM API の章で詳しく説明しています。

MetaVisit

```
virtual void
MetaVisit(u2::ITtdMetaVisitCallback* pCallback,
  bool bVisitAll = false,
  bool bVisitRefs = false) const
throw(u2::APIError) = 0; throw(u2::APIError) = 0;
```

このエンティティをルートとする、モデルのメタモデル駆動によるトラバースを実行します。`visitAll` が ”false” の場合は、ライブラリと定義済みパッケージがトラバース対象から除外されます。`visitRefs` が ”false” の場合は、参照の識別子表現がトラバース対象から除外されます。

アクセスされたエンティティごとに、`pCallback` オブジェクトの OnVisitedEntity 関数が呼び出されます。ただし、この関数が呼ばれるのは、モデルのトラバースがあるエンティティに到達したが、そのエンティティに含まれるエンティティにはまだ到達していない場合のみです。すべての含まれるエンティティがアクセスされている場合は、`pCallback` オブジェクトの OnAfterVisitedEntity 関数が呼び出されます。したがって、モデルトラバースにおける「後方再帰 (back recursion)」時に何らかのアクションを取れるようになります。

例 **719**

以下の例は、モデル内のすべての定義名を出力する場合の、`MetaVisit` 関数の使用法を示しています。

```
class DefinitionFinder : public u2::ITtdMetaVisitCallback {
public:
  virtual bool OnVisitedEntity(u2::ITtdEntity* pEntity) {
  if (pEntity->IsKindOf(_T("Definition"))) {
    tstring name;
    pEntity->GetValue(_T("Name"), name);
    std::wcout << name << std::endl;
```

```
    }
  return true;
  }

virtual void OnAfterVisitedEntity(u2::ITtdEntity* pEntity) {}
};

void foo() {
  u2::ITtdEntity* pEntity = u2::cast<u2::ITtdEntity>(pModel);
  DefinitionFinder finder;
  pEntity->MetaVisit(&finder, false /* visitAll */, false /*
visitRefs */);
}
```

MetaVisit 関数に対応する COM API 内のメソッドは、MetaVisitEx です。COM メソッド MetaVisit は、MetaVisitEx の機能限定バージョンです。

Bind

```
virtual void
Bind(const tstring& strMetaFeature = _T(""))
throw(u2::APIError) = 0;
```

エンティティ上の指定されたメタ特性のバインドを試みます。`strMetaFeature` が空の場合は、すべてのメタ特性が対象になります。存在しないメタ特性が指定された場合は、PIError 例外が発生します

Bind については、COM API の章で詳しく説明しています。

Locate

```
virtual void
Locate() const
throw(u2::APIError) = 0;
```

エンティティを、［モデルビュー］または図、あるいはその両方でエンティティを検索します。IDE ではエンティティがモデル ビューと図（エンティティのプレゼンテーション表現がダイアグラムに存在する場合のみ）に表示されます。IDE が使用できないときにこの関数を使用すると、APIError 例外が発生します。

Clone

```
virtual u2::ITtdEntity*
Clone(bool bPreserveBindings = false,
  bool bPreserveGuids = false) const
throw(u2::APIError) = 0);
```

エンティティのクローンを作成します。デフォルトでは、クローンはバインドされず、新しい一意の GUID をとります（エンティティのコピーとそのエンティティに含まれるすべてのエンティティのコピーについて、新しい一意の GUID が取得されます）。オプションの引数を使用して、元のエンティティと同じバインディングを持つクローン、または元のエンティティと同じ GUID を持つクローンを作成できます。

重要！

GUID を変更せずにエンティティのクローンを作成するときは注意してください。元のエンティティと同じモデルにこのようなクローンを挿入すると GUID の衝突が発生します。GUID が衝突しているモデルは、保存後に再ロードできない可能性があります。

クローンのバインディングを維持可能にするには、元のエンティティはモデルに属していなければなりません（つまり、元のエンティティ上で呼び出された GetModel は NULL を返してはなりません）。

クローン作成に失敗した場合は、常に APIError 例外が発生します。

Move

```
virtual void
Move(u2::ITtdEntity* pNewOwner,
  const tstring& strMetaFeature = _T(""),
  eposition index = E_POSITION_END,
  u2dll::Cu2Changer& changer = u2dll::defaultChanger)
throw(u2::APIError) = 0;
```

エンティティをモデル内の現在の位置から `pNewOwner` のコンテキストに移動します。エンティティが新しい所有者の複数のメタ特性に適合できる場合、`strMetaFeature` を指定して、どのメタ特性かを区別する必要があります。ターゲットのメタ特性が複数の多重度を持つ場合、`index` 引数を指定して、エンティティの移動先となる位置を制御できます。

モデルの変更は、チェンジャを使用して実行されます。

移動できない場合は、APIError 例外が発生します。

GetModel

```
virtual u2::ITtdModel*
GetModel() const=0;
```

エンティティが属しているモデルを返します。エンティティがモデルに属していない場合は、NULL が返されます。この関数は、ITtdEntity インタフェースのコンテキストから ITtdModel インタフェースを取得するもっとも便利な方法として、よく使用されます。

UnlinkFromOwner

```
virtual void
UnlinkFromOwner(u2dll::Cu2Changer& changer =
u2dll::defaultChanger) const = 0;
```

エンティティを現在の所有者からリンク解除します。エンティティは削除されません。たとえば、モデルの他の場所や他のモデルに挿入される可能性があります。

モデルの変更は、チェンジャを使用して実行されます。

Replace

```
virtual void
Replace(u2::ITtdEntity* pReplacementEntity,
  u2dll::Cu2Changer& changer = u2dll::defaultChanger) const
throw(u2::APIError) = 0;
```

エンティティを他のエンティティで置き換えます。元のエンティティは削除されません。置き換えができない場合は、APIError 例外が発生します。

モデルの変更は、チェンジャを使用して実行されます。

注記

エンティティが、特定の参照を表現する識別子である場合は、エンティティは、`pReplacementEntity` そのものではなく、`pReplacementEntity` のクローンで置き換えられます。

GetContainerMetaFeature

```
virtual void
GetContainerMetaFeature(tstring& strMetaFeature,
  u2dll::eposition& index) const = 0;
```

`strMetaFeature` に、このエンティティを含んでいるメタ特性の名前を設定します。そのエンティティが孤立している場合は、空文字列になります。`index` 値は、メタ特性内のエンティティが存在する場所を指すように設定します。

FindByName

```
virtual u2::ITtdEntity*
FindByName(const tstring& strName) const = 0;
```

エンティティを名前（通常は修飾名）で検索します。`strName` は有効な識別子である必要があります。

エンティティが見つからない場合は、NULL が返ります。

GetDescriptiveName

```
virtual void
GetDescriptiveName(tstring& strDescription) const = 0;
```

`strDescription` をエンティティの説明名に設定します。説明には、エンティティのメタクラス、名前（イベントクラスの場合は完全シグニチャ）、モデル内での位置を含みます。

ITtdResource

ITtdResource インターフェイスは、UML モデルを永続的に格納できるリソースを表すメタモデルの Resource クラスによって実装されます。通常、Resource は `.u2` ファイルです。

注記

Resource は Entity でもあるので、ITtdResource から ITtdEntity へのキャストは常に成功します。

Save

```
virtual void
Save() const
throw(u2::APIError) = 0;
```

メソッドの呼び出し元リソースに関連付けられているモデル エンティティを保存します。リソースが .u2 ファイルの場合は、そのファイルが保存されます。

Save については、COM API の章で詳しく説明しています。

ITtdPresentationElement

ITtdPresentationElement インターフェイスは、メタモデルの PresentationElement クラスによって実装されます。プレゼンテーション要素は、たとえば、シンボル、ライン、またはダイアグラムなどのグラフィックな表示が含まれる要素です。

注記

PresentationElement は Entity でもあるので、ITtdPresentationElement から ITtdEntity へのキャストは常に成功します。

GenerateEMF

```
virtual void
GenerateEMF(const tstring& strFileName,
  long maxWidth = 0,
  long maxHeight = 0,
  bool bOptimizeForVectorGraphics = false,
  bool bIncludeFrame = false) const
throw(u2::APIError) = 0;
```

プレゼンテーション要素のグラフィック表示に対応する EMF ファイル（Enhanced Meta File）を生成します。この関数はすでに非推奨になっています。この関数の代わりに GenerateEMFEx を使用してください。EMF ファイル内のプレゼンテーション要素は、Tau のエディタで表示されるものと同じ外観になります。

GenerateEMF については、COM API の章で詳しく説明しています。

GenerateEMFEx

```
virtual void
GenerateEMFEx(const tstring& strFileName,
  long maxWidth = 0,
  long maxHeight = 0,
  bool includeFrame = false)
  long scaleFactor = 0) const
throw(u2::APIError) = 0;
```

プレゼンテーション要素のグラフィック表示に対応する EMF ファイル（Enhanced Meta File）を生成します。EMF ファイル内のプレゼンテーション要素は、Tau のエディタで表示されるものと同じ外観になります。

注記

現在、GenerateEMF と GenerateEMFEx は、対話型実行環境のみで使用できます。他の実行環境からこの関数を呼び出そうとすると、APIError 例外になります。

GenerateEMFEx については、COM API の章で詳しく説明しています。

GenerateImage

```
virtual void
GenerateImage(ImageKind nKind,
  const tstring& strFileName) const
throw(u2::APIError) = 0;
```

プレゼンテーション要素のグラフィカルな外観から画像ファイルを生成します。プレゼンテーション要素は、画像ファイル内においても、ツールで表示される場合と同一の外観になります。

`ImageKind` 列挙内の有効な値とその意味は、下の表のとおりです:

u2::ImageKind	説明
IK_JPEG	JPEG 画像ファイルを生成します。
IK_BMP	BMP 画像ファイルを生成します。
IK_GIF	GIF 画像ファイルを生成します。
IK_TIFF	TIFF 画像ファイルを生成します。
IK_TARGA	TGA (Targa) 画像ファイルを生成します。
IK_DIB	装置非依存のビットマップファイルを生成します。
IK_PCX	PCX 画像ファイルを生成します。

ITtdSymbol

ITtdSymbol インターフェイスは、メタモデルの Symbol クラスによって実装されます。シンボルは 2 次元のグラフィック表示を持つプレゼンテーション要素です。

注記

Symbol は PresentationElement でも Entity でもあるので、ITtdSymbol から ITtdPresentationElement または ITtdEntity へのキャストは常に成功します。

SetSize

```
virtual void
SetSize(unsigned long width,
  unsigned long height,
  u2dll::Cu2Changer& changer = u2dll::defaultChanger) = 0;
```

シンボルのサイズを設定します。幅と高さの単位は、1/10 mm です。非対話型の実行環境では、SetSize はシンボルの "size" メタ特性の値を更新するだけです。対話型の実行環境では、SetSize がサイズ変更に関連してモデル変更を行うこともできます。たとえば、シンボル サイズにモデルの意味として重要性がある場合などが考えられます。したがって、シンボルのサイズの設定には必ず SetSize を使用することを推奨します。

モデルの変更は、チェンジャを使用して実行されます。

SetPosition

```
virtual void
SetPosition(long x,
  long y,
u2dll::Cu2Changer& changer = u2dll::defaultChanger) = 0;
```

シンボルの位置を設定します。x パラメータと y パラメータの単位は、1/10 mm です。非対話型の実行環境では、SetPosition は、シンボルの "position" メタ特性の値を更新するだけです。対話型の実行環境では、SetPosition は、シンボルの再配置に関連してモデル変更を行うこともできます。たとえば、シンボル位置にモデルの意味として重要性がある場合などが考えられます。したがって、シンボルの位置設定には必ず SetPosition を使用することを推奨します。

モデルの変更は、チェンジャを使用して実行されます。

ITtdExpression

ITtdExpression インターフェイスは、メタモデルの Expression クラスによって実装されます。モデル内の式を表現します。

注記

Expression もまた Entity なので、ITtdExpression から ITtdEntity へのキャストは常に成功します。

GetType

```
virtual u2::ITtdEntity*
GetType() const = 0;
```

式の型を戻します。型が算出できない場合（たとえば、式がバインドされていない参照を含む場合）は、NULL が返されます。

注記

返されたエンティティは、必ずしもモデルの一部ではありません。モデルにおいて明示的に型が定義されていない場合もあり、そのような場合は、型を表現する一時的なエンティティが戻されます。こういった一時エンティティを削除しないよう注意してください。

EvaluateConstantIntegralExpression

```
virtual long
EvaluateConstantIntegralExpression() const
throw (u2::APIError) = 0;
```

式の値を評価します。値が整数定数式でない場合、または何らかの理由で評価が失敗した場合は、APIError 例外が発生します。

GetInstanceChildExpression

```
virtual u2::ITtdExpression*
GetInstanceChildExpression(const tstring& strName) const
throw(u2::APIError) = 0;
```

この関数をインスタンス式（たとえば特定のステレオタイプインスタンスなど）上で使用すると、割り当ての右辺値が取得できます。割り当ての左辺値は strname の値です（したがって strname は有効な識別子である必要があります）。

一致する式が見つからない場合は、NULL が戻されます。

誤って使用すると、APIError 例外が発生します。

ITtdMetaVisitCallback

ITtdMetaVisitCallback インターフェイスは、MetaVisit 関数を使用するクライアントで実装する必要があるコールバック インターフェイスです。

OnVisitedEntity

```
virtual bool
OnVisitedEntity(u2::ITtdEntity* pEntity) = 0;
```

MetaVisit を使用してモデルをトラバースするときに呼び出されます。pEntity は、モデル内で現在アクセスされているエンティティです。この関数は、pEntity の合成子をアクセスする前に呼び出されます。”false” が返された場合、トラバースは pEntity の合成子については続行されません。

OnVisitedEntity については、COM API の章で詳しく説明しています。

OnAfterVisitedEntity

```
virtual void
OnAfterVisitedEntity(u2::ITtdEntity* pEntity) = 0;
```

MetaVisit を使用してモデルをトラバースするときに呼び出されます。pEntity は、モデル内で現在アクセスされているエンティティです。この関数は、pEntity の合成子がアクセスされた後に呼び出されます。したがって、モデル トラバースの「後方再帰 (backward recursion)」において何らかのアクションをする際に使用されます。

ITtdSourceBuffer

ITtdSourceBuffer インターフェイスは、主にコード生成時に、生成されたファイルの表現として使用されるテキスト（通常はソース コード）のバッファを表します。たとえば、生成されたファイルに追加テキストを加える C++ アプリケーション ジェネレータのエージェントが使用できます。

AddText

```
virtual void
AddText(const tstring& strText)
throw(u2::APIError) = 0;
```

指定されたテキストをソース バッファに書き込みます。

AddText については、COM API の章で詳しく説明しています。

ITtdMessageList

ITtdMessageList インターフェイスは、一連のメッセージを表します。このリストは、たとえば、セマンティック分析またはコード生成からのエラーを報告するときに使用されます。このインターフェイスは、エージェントがカスタム セマンティック チェックなどに基づくカスタム メッセージを追加するときに使用できます。

AddMessage

```
virtual void
AddMessage(const tstring& strMessage,
  MessageSeverity severity,
  u2::ITtdEntity* pSubject = 0)
throw(u2::APIError) = 0;
```

指定された重要度とオプションによるサブジェクト エンティティ付きのリストに新しいメッセージを追加します。サブジェクト エンティティはメッセージに関連付けられます。対話型実行環境では、サブジェクト エンティティをエラーから探すことができます。

`MessageSeverity` の列挙は、以下のように定義されます。

```
enum MessageSeverity {
  MS_INFORMATION,
  MS_WARNING,
  MS_ERROR,
  MS_FATAL
};
```

AddMessage については、COM API の章で詳しく説明しています。

GetDescription

```
virtual void
GetDescription(tstring& strErrorMessage,
  const tstring& strSeparator) const = 0;
```

`strErrorMessage` 文字列にメッセージリスト中のすべてのメッセージの説明を書き出します。メッセージは指定された区切り文字で区切られます。

GetCount

```
virtual unsigned int
GetCount(MessageSeverity severity) const = 0;
```

`severity` の値と同一の重大度をもったメッセージの数が戻されます。

ITtdInteractiveServer

ITtdInteractiveServer インターフェイスは、対話型 API クライアントのサーバーとしての Tau IDE を表します。対話型 クライアントが、その実行中に Tau と通信するためのインターフェイスを定義します。

InterpretTclScript

```
virtual void
InterpretTclScript(const tstring& strScript,
  std::list<u2::ITtdEntity*>& entities,
  tstring& strResult)
throw(u2::APIError) = 0;
```

サーバー上の Tcl スクリプトを解釈して、結果を呼び出し側に返します。この関数は、対話型 クライアントが IDE と通信するための手段となります。Tcl API はすべて使用できます。

InterpretTclScript については、COM API の章で詳しく説明しています。

ITtdCppAppGenServer

ITtdCppAppGenServer インターフェイスは、非対話型 API クライアントのサーバーとしての Tau C++ アプリケーション ジェネレータを表します。エージェントが、その実行中にコード ジェネレータと通信するためのインターフェイスを定義します。エージェントは、起動時の「server」引数としてこのインターフェイスを取得します。

ScheduleForDeletion

```
virtual void
ScheduleForDeletion(u2::ITtdEntity* pEntity)
throw(APIError) = 0;
```

エンティティをモデルからリンク解除して、削除をスケジュールします。エンティティは、C++ アプリケーション ジェネレータが妥当と判断したタイミングで削除されます。

ScheduleForDeletion については、COM API の章で詳しく説明しています。

C++ API のセットアップ

この章では、C++ API を使用するためのクライアント アプリケーションのセットアップ プロセスについて、順を追って説明します。このプロセスの大半はプラットフォームに依存するため、Unix プラットフォームと Windows プラットフォームそれぞれについて説明しています。

Windows クライアント

Tau C++ API を使用するために、以下の Visual Studio .NET プロジェクト セットアップ手順を実行します。

1. Tau インストール ディレクトリの `include¥ToolAPI` ディレクトリを、プロジェクトのインクルード パス リストに追加します。この手順を実行しない場合は、API ヘッダのインクルード ディレクティブで適切な相対パスを使用する必要があります。
2. `#include "U2ModelAccess.h"` を、API のアクセス元にする実装ファイルに追加します。
3. Tau インストール ディレクトリの `lib¥win32-vc` にあるライブラリ `U2DLLU.lib` と `SBL10U.lib` にリンクされるように、プロジェクトをセットアップします。
4. “Multi-threaded DLL” ランタイム ライブラリを使用するように、プロジェクトをセットアップします。
5. クライアントがワイド（Unicode）文字列を使用していることを確認します。アプリケーションでシングルバイト文字列を使用する場合は、API がワイド（Unicode）文字列の使用を前提としているため、変換を行う必要があります。Unicode 文字列の使用を指定するには、マクロ `UNICODE` とマクロ `_UNICODE` を設定します。
6. `PATH` 環境変数に、Tau インストール ディレクトリの `bin` ディレクトリが含まれていることを確認します。対話型 API クライアント（エージェントなど）をビルドする場合は、この手順を実行する必要はありません。

参照

Visual Studio .NET での C++ エージェントのデバッグ

Unix クライアント

Tau C++ API を使用する Unix アプリケーション用に、以下の makefile 作成手順を実行します。

1. Tau インストール ディレクトリの `include/ToolAPI` ディレクトリを、プリプロセッサのインクルード パス リストに追加します。この手順を実行しない場合は、API ヘッダのインクルード ディレクティブで適切な相対パスを使用する必要があります。
2. `#include "U2ModelAccess.h"` を、API のアクセス元にする実装ファイルに追加します。
3. Tau インストール ディレクトリの `lib` ディレクトリにあるライブラリ `u2dll.lib` と `sbl10.lib` にリンクします。サポートするコンパイラとプラットフォームごとに構築されたライブラリを持つサブフォルダが存在します。間違いなく正しいフォルダを選択してください。

4. Solaris 上の CC と Linux 上の g++ を使用してコンパイルします。Solaris 上のフラグ `-mt` または Linux 上の
`- D_REENTRANT` を使用して、マルチスレッド ランタイム ライブラリの使用を指定します。
5. クライアントで ASCII 文字列が使用されていることを確認します。API では ASCII 文字列が想定されるため、アプリケーションでマルチ バイトの文字列を使用する場合は、必要な変換を実行する必要があります。
6. `LD_LIBRARY_PATH` 環境変数に、Tau インストール ディレクトリの `bin` ディレクトリが含まれていることを確認します。エージェントなどの対話型 API クライアントをビルドする場合は、この手順を実行する必要はありません。
Linux RedHat 5 については、LD_LIBRARY_PATH が Tau インストールの bin/.rh5 を bin ディレクトリより前に含んでいる必要があります。

Visual Studio .NET での C++ エージェントのデバッグ

この項では、Visual Studio .NET 2008 を使用して、C++ エージェントまたは任意の Tau C++ API クライアントをデバッグする方法について説明します。

適切なデバッグ構成のセットアップ

まず注意することは、Visual Studio .NET を使用してデバッグ構成でビルドされたプログラムでは、デフォルトで、デバッグ バージョンの C ランタイムライブラリが使用される点です。エージェント DLL の場合は、マルチ スレッド DLL バージョンのランタイム ライブラリ、つまり MSVCR8D ライブラリが使用されます。ただし、すべての Tau のバイナリコードは、リリース構成でビルドされて提供されるため、非デバッグ バージョンのライブラリ（MSVCR8）が使用されます。したがって、デバッグ構成でビルドされたエージェントを Tau バイナリ コード（VCS.EXE など）内で実行しようとすると、デバッグ バージョンとリリース バージョンの両方のランタイム ライブラリがメモリに混在することにより、多くの場合、問題が発生します。エージェント DLL に割り当てられたメモリが Tau で割り当てが解除されたり、また逆のことも発生するため、デバッグ アサーションを引き起こして、アプリケーションがクラッシュすることがあります。

この問題を解消するために、エージェント DLL は必ずリリース構成でビルドします。こうすることで、安定してランタイム ライブラリを使用できます。ただし、リリース構成では、デフォルトでデバッグ情報がオフになっているため、このようなエージェントをデバッグするには手動でデバッグ情報をオンにする必要があります。このためには、以下の手順を行います。

- デフォルトのリリース構成のデフォルトを変更するか、デフォルトのリリース構成をベースに新しい構成を作成して、リリース構成から起動します。新しい構成は、Visual Studio .NET で［ビルド］-> ［構成マネージャ］を選択して作成できます。
- プロジェクトのプロパティ ページを開いて、［C/C++］タブ（カテゴリは［全般］）を選択します。［デバッグ情報の形式］を［プログラム データベース］に設定し、最適化（カテゴリは［最適化］）をすべて無効にします。
- ［リンカ］タブ（カテゴリは［デバッグ］）を選択します。［デバッグ情報の生成］オプションを「はい」に設定します。
- 最後に、［デバッグ］ページで、Tau のインストール ディレクトリの適切な Telelogic Tau バイナリ（たとえば、VCS.EXE）をデバッグ セッションの実行形式ファイル（コマンド）として設定します。また、作業ディレクトリとして、Tau インストール ディレクトリを指定します。
- これで、エージェント DLL をビルドして、ビルドの実装にブレークポイントを設定して、デバッグ セッションを開始できます。Tau が起動してエージェントが呼び出されると、ブレークポイントが検出されて、デバッグが実行されます。

デバッグ ユーティリティ

Tau ライブラリには、C++ エージェントや C++ API クライアントのデバッグに役立ついくつかのユーティリティ関数が組み込まれています。これらのユーティリティ関数は、Tau インストールの一部である API ヘッダーファイルでは一切、宣言されていませんが、エージェントの実装ファイルで関数プロトタイプを宣言すると、これらの関数にアクセスできます。これらの関数をデバッガ内から呼び出すこともできます。

すべてのデバッグ ユーティリティ関数は、DLL からエクスポートされた関数として定義されます。Visual Studio .NET デバッガからこれらの関数を呼び出す方法については、以下の U2ViewModel を参照してください。原理は他の関数と同じです。

下表に使用可能なデバッグ ユーティリティ、それが存在する Tau DLL、定義されている名前空間を示します。

関数	名前空間	DLL
U2ViewModel	u2dll	u2dllu.dll
DbgInterpretTclScript	u2ext	u2extu.dll

U2DLL DLL は、対話型、非対話型のどちらのエージェントでも使用できますが、U2EXT DLL は対話型 エージェントでのみ使用可能です。

U2ViewModel

```
namespace u2dll {
  void U2ViewModel(const void* pEntity);
```

この関数は、pEntity をルートとするモデルを XML 形式の一時ファイルにダンプします。この関数は、モデルの内容を検査するのに便利です。Internet Explorer(iexplore.exe) がパスに含まれる場合、IE が起動して XML ファイルが表示されます。それ以外の場合は、生成ファイルは、生成先の一時ファイル用ユーザ ディレクトリから開くことができます。`pEntity` は、`u2::ITtdEntity` インターフェイスを実現するオブジェクトのアドレスです。

デバッガからこの関数を呼び出すには、以下に示すように、[Command] ウィンドウ（「直接」モード）または [Quick Watch] ウィンドウを開いて、`U2ViewModel` を呼び出します。引数はアドレスとして指定する必要があります（[Watch] ウィンドウに表示されているように）。

```
{,,u2dllu} u2dll::U2ViewModel((void*) 0x12ea4d24)
```

エージェント実装からこの関数を呼び出すには、エージェントの実装ファイルの先頭にこの宣言を入れます。

```
namespace u2dll {
    __declspec(dllimport)
    void U2ViewModel(const void* pEntity);
}
```

これで、エージェント実装の ITtdEntity ポインタで U2ViewModel を呼び出すことができます。

ヒント

Visual Studio .NET デバッガは、DLL にある関数のデバッグ シンボルが見つからない場合、呼び出しを拒否することがあります。このような問題が発生した場合は、以下のように解決してみてください。
上記 U2ViewModel に示しているように、呼び出したい関数の宣言をエージェントの実装ファイルに追加します。その後、その関数を呼び出すローカル ラッパー関数を作成します。

```
void U2ViewModelWrapper(const void* pEntity) {
  u2dll::U2ViewModel(pEntity);
}
```

エージェントを再ビルドします。これで、デバッガから U2ViewModelWrapper を直接呼び出すことができます。

```
U2ViewModelWrapper((void*) <address>)
```

DbgInterpretTclScript

```
namespace u2ext {
  char* DbgInterpretTclScript(char* arg);
}
```

この関数は、Tau Tcl API 全体へのアクセスを可能にします。エージェントのデバッグに使用できます。特に、デバッグ対象のアプリケーション内のモデルに移動する場合に便利です。関数の入力引数は Tcl スクリプト、戻り値は Tau Tcl インタープリタによってスクリプトが解釈された結果です。

Tcl スクリプト文字列は、デバッグ対象アプリケーション内のモデル エンティティを参照してもかまいません。そのような参照を形成するには、Visual Studio .NET デバッガに表示されるように、エンティティのアドレスを # で囲みます。

例 **720: Visual Studio .NET** デバッガの **DbgInterpretTclScript** を使用 ——————

Visual Studio .NET の［Command］ウィンドウを開き、「直接」モードに入ります（「immed」コマンドを入力）。

アドレス 0x0f958720 を持つエンティティの所有者を検索すると、

```
{,,u2extu} u2ext::DbgInterpretTclScript("u2::GetOwner
#0x0f958720#")
```

以下の形式で結果が出力されます。

```
0x0d5bcd58 "i.66.f891eb0"
```

結果の最初のアドレスは、文字列の内部アドレスなので関係ありません。引用符に囲まれた文字列が、Tcl スクリプトの実行結果です。この例のように結果が Tcl のオブジェクト ID であった場合、アドレスから接頭辞を取り除くだけで、デバッグ対象のア

プリケーションのアドレスに変換できます。したがって、この例の Tcl オブジェクトはアドレス 0x0f891eb0 に対応しています。このアドレスは、他のデバッグ ユーティリティ関数の呼び出しのための入力に使用できます。

Tcl API は対話型の Tau IDE に実装されているので、この関数は対話型 エージェントからのみアクセスが可能です。

61

Java API

This chapter is the reference documentation of the Tau Java API, which enables access of a Tau model from a Java program.

Intended readers are developers of client applications that use the Java API to access a UML model. A basic knowledge of Java is assumed throughout this chapter.

Introduction

The Java API consists of a set of interfaces, each with a number of methods. The design of the API is very similar to the design of the C++ API. In fact the Java API is implemented in terms of the C++ API using the Java Native Interface (JNI).

A client of the Java API is a Java program which executes inside a Java Virtual Machine (JVM). This means that it is only possible to build non-interactive clients using the Java API. An interactive client, such as an agent, cannot be implemented in Java.

Java version

To use the Java API the client program needs to run inside a JVM for Java 5 or later.

Accessing the API

A client that wants to use the Java API should place the `tauaccess.jar` in its classpath. This JAR file can be found in the Tau installation at `/lib/Java`. The client must also set-up the environment variable `PATH` (`LD_LIBRARY_PATH` on Unix) so that it includes the Tau installation `bin` directory. Note that for Linux RedHat 5 `LD_LIBRARY_PATH` must also contain the Tau installation `bin/.rh5` directory, and this directory must be listed before the `bin` directory.

Execution Environments

Usually a Java API client loads or creates a UML model inside the JVM where it executes. The client is said to execute in a non-interactive environment because the Tau IDE is not involved.

The starting point for accessing the API from a non-interactive client is the method `TauModelAPI.getModelAccess()`. This method returns a reference to a singleton object that implements the ITtdModelAccess interface.

However, by using Tau Access it is also possible to run a Java API client in an interactive execution environment where the implementation of the Java API operates on a running instance of Tau, where the IDE is available.

The starting point for accessing the API from an interactive client is through interfaces provided by the Tau Access API. See Tau Access for more information.

In most cases there is no functional difference of the Java API when used in an interactive vs. non-interactive execution environment. However, the following differences apply:

- API methods that modify the model will be undoable in an interactive execution environment (i.e. Undo/Redo can be used to undo/redo the actions performed by the API methods). In a non-interactive execution environment model modifications are not undoable.
- Some API methods have parameters that are specific for a particular execution environment. The value of such parameters in the other execution environment will be ignored.
- Some API methods can only be used in a particular execution environment - typically because their implementation rely on features of the Tau IDE.

- Some API methods work slightly differently in different execution environments. Typically they may be more complete in their support in an interactive execution environment since features of the Tau IDE then are available.

API Initialization and Finalization

It is important to initialize the API before starting to use it. This is done by calling `TauModelAPI.initializeModelAccess()`.
After the last API call the API should also be finalized, by a call to `TauModelAPI.finalizeModelAccess()`. It is allowed to repeat initialization and finalization as long as the calls are balanced.

例 721 API initialization and finalization

Initializing the Java API, obtaining a reference to the `ITtdModelAccess` interface, and finalizing the API afterwards:

```
TauModelAPI.intializeModelAccess();
ITtdModelAccess ma = TauModelAPI.getModelAccess();
TauModelAPI.finalizeModelAccess();
```

Interface Casting

One interface can be casted to another interface dynamically using the usual Java cast operator. If the cast is successful, meaning that the object which implements the source interface also implements the target interface, the result of the cast is a reference typed by the target interface. However, if the cast fails a `java.lang.ClassCastException` will be thrown. Make sure to catch this exception unless you are certain that the object at hand implements the target interface.

例 722 Interface casting

Dynamic casting between interfaces using the Java cast operator.

```
ITtdModelAccess ma = TauModelAPI.getModelAccess();
ITtdModel model = ma.loadProject("x.u2", true, new MessageList());
ITtdEntity e = (ITtdEntity) model; // A model is an entity
try
{
  ITtdSymbol s = (ITtdSymbol) model.findByGuid("abc123");
}
catch (ClassCastException err)
{
  // Error handling here
}
```

The cast from `ITtdModel` to `ITtdEntity` will always succeed (because the model node is also an entity). Therefore we don't need to catch any exception from that cast. However, the cast from `ITtdEntity` to `ITtdSymbol` will not always succeed (not all entities are symbols). Hence we should be prepared for a `ClassCastException` in this case.

It is also possible to use the Java `instanceof` operator to test whether an object implements a certain interface.

Handling API Errors

The Java API reports all errors by throwing an `APIError` exception. Client code should be prepared for errors by catching this exception. The exception class is defined like this:

```
package com.telelogic.tau;

// Common exception class describing an error occurring while
using the Tau APIs.
public class APIError extends Throwable
{
  public APIError(String msg)
  {
    super(msg);
  }
}
```

All methods inherited from the `Throwable` class can be used on `APIError`. In particular the textual description of the error is obtained by calling `getMessage()`.

例 723 Catching APIError exceptions

A typical catch clause for handling Java API errors could look like this:

```
try {
  // access the Java API
}
catch (APIError err)
{
  String s = err.getMessage();
  System.out.println("Error while using the Tau Java API: " + s);
}
```

Memory Management

Java programmers tend not to think much about memory management due to the presence of garbage collection in Java. However, it's important to understand that the Java garbage collector will only manage Java objects. Objects that are created in the Tau object model are not Java objects, although they implement a Java interface. Therefore such objects should be manually deleted when they are no longer used, to avoid memory leaks. Use `ITtdEntity::`delete to delete an entity from the model.

In practise, however, a UML model that is created or loaded through the Java API often lives throughout the entire lifetime of the Java application, and hence memory management becomes no concern.

Client restrictions

The Java API imposes few restrictions on its clients. However, please keep in mind the following when designing the API client:

Bare only

The API does not support safe simultaneous access of the model from multiple threads.

API Interfaces and Methods

All interfaces of the Java API are placed in the package `com.telelogic.tau`. This section lists all API interfaces and methods and gives a short description of each. Some examples of typical use are provided. Note, however, that the examples only serve as an illustration of how to use a particular API method or interface, and are not always complete. In particular the recommended error handling (see Handling API Errors) is usually omitted from the examples for brevity reasons.

The names of the Java interfaces and their methods are the same as the corresponding interfaces and methods of the COM and C++ APIs. However, note that by convention Java method names start with a lowercase letter.

Many of the interfaces are implemented by classes representing metaclasses in the implementation of the UML メタモデル . Some of the methods in the Java API require knowledge of the metamodel in order to be useful.

ITtdModelAccess

The `ITtdModelAccess` interface contains methods that do not operate directly on the model. Use it to get access to a UML model from a non-interactive API client by loading it from files (project file or `.u2` file), or create a new model from scratch.

See Accessing the API to learn how to obtain the `ITtdModelAccess` interface from the client application.

loadProject

```
ITtdModel loadProject(String strProject, boolean bind,
ITtdMessageList messages) throws APIError;
```

Loads the project stored in the specified project file (or URI). If the project file does not exist, or some other error occurs while loading the project, an `APIError` exception will be thrown. If `strProject` is a relative path, it will be interpreted as relative to the current working directory of the client application.

If the `bind` parameter is set to true, the model will be bound after it has been loaded. Set it to false to suppress this binding.

If a message list is passed to the function all messages that are produced during loading and binding will be added to that list. A message list can be obtained by defining a class which implements the ITtdMessageList interface. In case load messages are not of interest you may pass `null` for this parameter.

例 724

This example loads a model from a Tau project file, without binding the loaded model. All messages are ignored.

```
ITtdModelAccess ma = TauModelAPI.getModelAccess();
ITtdModel model = ma.loadProject("x.ttp", false, null);
```

This example loads a project and uses a message list for printing load and bind messages:

```
public class Test
{
  public static class MessageList implements ITtdMessageList
  {
    public void addMessage(String text, MessageSeverity severity,
ITtdEntity subject)
    {
      System.out.println(text);
      if (subject != null)
        System.out.println(subject.getMetaClassName());
    }
  }

  public static void main(String[] args) throws APIError
  {
    TauModelAPI.intializeModelAccess();
    ITtdModelAccess ma = TauModelAPI.getModelAccess();
    ITtdModel model = ma.loadProject("x.ttp", true, new
MessageList());
    TauModelAPI.finalizeModelAccess();
  }
}
```

loadFile

```
ITtdModel loadFile(String strFile, boolean library) throws
APIError;
```

Loads the specified `.u2` file (or URI). If the file does not exist, or some error occurs while loading it, an `APIError` exception will be thrown. If `strFile` is a relative path, it will be interpreted as relative to the current working directory of the client application. If `library` is true, the file will be loaded as a library. In that case the file should contain a package, which then will be placed in the Libraries section of the model.

例 725

This example loads a model from a Tau `.u2` file as a library. It then prints the name of all library packages in the model. The last package printed will be the package contained in the loaded `.u2` file.

```
ITtdModelAccess ma = TauModelAPI.getModelAccess();
ITtdModel model = ma.loadFile("x.u2", true);
ITtdEntity entityModel = (ITtdEntity) model;
List<ITtdEntity> lst = entityModel.getEntities("Library");
for (ITtdEntity e : lst)
{
  System.out.println(e.getValue("Name", 0));
}
```

Note that `ITtdModelAccess::loadFile()` loads the file into a new model. In order to load a `.u2` file into an existing model use `ITtdModel::loadFile()`.

createModel

```
ITtdModel createModel() throws APIError;
```

Creates a new empty model. If a new model cannot be created (for example due to a memory or license problem), an `APIError` exception will be thrown.

The created empty model can be used as a starting-point for creating a whole new UML model. The example below shows the creation of a simple model containing one package with one class. It also shows how to save the new model afterwards.

例 726

This example creates a new model, containing a package with a class, and saves it to a `.u2` file.

```
ITtdModelAccess ma = TauModelAPI.getModelAccess();
ITtdModel model = ma.createModel();
ITtdEntity modelRoot = (ITtdEntity) model;
ITtdEntity pkg = modelRoot.create("Package", false, "OwnedMember");
ITtdEntity cls = pkg.create("Class", false, "");
ITtdResource resource = model.createResource("C:\\temp\\test.u2");
// Insert the created package pkg as a root of the resource
((ITtdEntity) resource).setEntity("Root", pkg, 0);
model.save(); // Saves all resources in the model
```

writeMessage

```
void writeMessage(String message);
```

Writes a message to the default message area (typically to `stdout` in a non-interactive execution environment, or to the Message tab in an interactive execution environment).

例 727

```
ITtdModelAccess ma = TauModelAPI.getModelAccess();
ma.writeMessage("Hello from Java!");
```

ITtdModel

The `ITtdModel` interface is implemented by the Session class of the メタモデル , which represents the top-level entity of a UML model.

The `ITtdModel` interface contains methods that do not need a specific model entity context for their execution. Typically these methods operate on the model as a whole, rather than on a particular entity.

注記

Since a Session also is an Entity, a cast from `ITtdModel` to `ITtdEntity` will always succeed.

findByGuid

```
ITtdEntity findByGuid(String strGuid);
```

Returns the entity in the model with the specified GUID, or `null` if no such entity exists.

例 728

This example creates a new model, and then locates the predefined Integer datatype by its GUID.

```
ITtdModelAccess ma = TauModelAPI.getModelAccess();
ITtdModel model = ma.createModel();
ITtdEntity integerType = model.findByGuid("@Predefined@Integer");
ma.writeMessage("Integer is a " + integerType.getMetaClassName());
```

New

```
ITtdEntity New(String strMetaClass) throws APIError;
```

Creates a new instance of the specified メタクラス . If a metaclass with the specified name does not exist, or the method fails for some other reason, an `APIError` exception is thrown.

注記

It is the responsibility of the client to take care of the returned entity. It should either be inserted into the model, or deleted, to avoid a memory leak. See Memory Management for more information.

The `New` method is intended to be used for low-level creation of object model entities, for example to create temporary model fragments which are not inserted into a model. Do not use New to build up a complete model - use `ITtdEntity::`create instead.

例 729

This example creates a new temporary package and sets up its name. It then prints its textual (unparsed) representation. Finally the package is deleted.

```
ITtdModelAccess ma = TauModelAPI.getModelAccess();
ITtdModel model = ma.createModel();
ITtdEntity pkg = model.New("Package");
pkg.setValue("Name", "MyPackage", 0);
ma.writeMessage(pkg.unparse());
pkg.delete();
```

parse

```
List<ITtdEntity> parse(String strConcreteSyntax, String
parseAs) throws APIError;
```

Parses the specified piece of concrete textual UML syntax. The parameter `parseAs` specifies the grammar to use when parsing, i.e. which kind of entities that should be defined in the text. Supported grammars are Definition, Expression and Action. In case a non-supported grammar is specified or the text contains syntax errors, an `APIError` exception will be thrown. The result built by the parser will be inserted into the returned list.

注記

It is the responsibility of the client to take care of the returned entities. They should either be inserted into the model, or deleted, to avoid a memory leak. See Memory Management for more information.

例 730

This example creates two classes by parsing their textual definitions. The names of the classes are then printed. Finally the classes are deleted.

```
ITtdModelAccess ma = TauModelAPI.getModelAccess();
ITtdModel model = ma.createModel();
List<ITtdEntity> lst = model.parse("class C1 {} class C2 {}",
"Definition");
for (ITtdEntity e : lst)
{
  System.out.println(e.getValue("Name", 0));
  e.delete();
}
```

XMLDecode

```
List<ITtdEntity> XMLDecode(String strXMLEncoding) throws
APIError;
```

Decodes the U2 XML encoded string into a list of model entities. If the decoding fails (e.g. because of a syntax error in the XML) an `APIError` will be thrown.

注記

It is the responsibility of the client to take care of the returned entities. They should either be inserted into the model, or deleted, to avoid a memory leak. See Memory Management for more information.

例 731

This example creates a model with one class. It then encodes the class as an XML string using `ITtdEntity::`XMLEncode. Then it calls XMLDecode to obtain the class again, and prints its metaclass name. The new class is in effect a clone of the original class. Note that `ITtdEntity::`clone is an easier way to clone a model entity.

```
ITtdModelAccess ma = TauModelAPI.getModelAccess();
ITtdModel model = ma.createModel();
ITtdEntity pkg = ((ITtdEntity) model).create("Class", false, "");
String xml = pkg.XMLEncode();
List<ITtdEntity> lst = model.XMLDecode(xml);
for (ITtdEntity e : lst)
{
  System.out.println(e.getMetaClassName());
  e.delete();
}
```

save

```
void save() throws APIError;
```

Saves the model into its resources. If the model does not contain any resources, nothing will happen.

If the model cannot be saved an `APIError` exception will be thrown.

See 例 726 on page 1995 for an example of using `save` to save a model.

createResource

```
ITtdResource createResource(String strFileName) throws
APIError;
```

Creates a new resource for the model. The resource will be a file with the specified file name. If the creation fails an `APIError` will be thrown.

See 例 726 on page 1995 for an example of using `createResource` to create a new resource in a model.

loadFile

```
ITtdResource loadFile(String strFileName, boolean profile)
throws APIError;
```

Loads the specified file into the model. A resource representing the file will be created and returned. If `profile` is true, the file will be loaded as a library. In cases of error (e.g. load errors) an `APIError` will be thrown.

例 732

This example creates a model and loads an existing file into it. It then obtains the last resource from the model (the one created by `loadFile`), and prints its URI.

```
ITtdModelAccess ma = TauModelAPI.getModelAccess();
ITtdModel model = ma.createModel();
model.loadFile("x.u2", false);
ITtdEntity resource = ((ITtdEntity) model).getEntity("Resource",
0);
String uri = resource.getValue("uri", 0);
System.out.println(uri);
```

invokeAgent

```
void invokeAgent(ITtdEntity agent, ITtdEntity modelContext,
List<Object> agentParameters) throws APIError;
```

Invokes the specified agent on the specified model context.

The `agentParameters` type is a list of エージェント パラメータ , representing actual arguments passed to the invoked agent. An agent parameter is in Java represented as an `Object` instance.

If the agent cannot be invoked, or the invoked agent signals an error, an `APIError` exception will be thrown.

例 733

This example invokes the predefined query agent `GetGeneralizationChildren` in order to find all definitions that inherit from the predefined `Collection` type. The agent takes two parameters; a list of resulting definitions and a boolean specifying whether also indirect generalization children should be returned.

Note that the first parameter is an in/out parameter. See the note below for how to deal with such parameters.

```
ITtdModelAccess ma = TauModelAPI.getModelAccess();
ITtdModel model = ma.createModel();
ITtdEntity collection = model.findByGuid("@Predefined@Collection");
ITtdEntity query =
model.findByGuid("@TTDQuery@GetGeneralizationChildren");

LinkedList<Object> agentParameters = new LinkedList<Object>();
agentParameters.add(new LinkedList<Object>());
agentParameters.add(true);

model.invokeAgent(query, collection, agentParameters);

List<Object> lst = (List<Object>) agentParameters.element();

for (Object obj : lst)
{
  if (obj instanceof IUnknown)
  {
    ITtdEntity entity = (ITtdEntity) obj;
    System.out.println(entity.getValue("Name" ,0) + " is a
Collection!");
  }
}
```

注記

If the invoked agent has `in/out` or `out` parameters, the values for these parameters shall be obtained after the call to `InvokeAgent` by iterating over the agent parameter list. After the agent invocation it is not safe to access a local reference to such a parameter that has been obtained prior to the agent invocation.

The table below shows the mapping of agent parameter types to Java types:

Agent parameter type (UML)	**Java type**
`Charstring`	`String`
`Integer`	`Integer`
`Boolean`	`Boolean`
`Reference to interface (for example u2::ITtdEntity)`	`com.telelogic.tau.IUnknown`
`u2::ITtdEntity [*]`	`List<ITtdEntity>`

ITtdEntity

The `ITtdEntity` interface is implemented by the Entity class of the メタモデル , which represents a general entity of a UML model.

The `ITtdEntity` interface contains methods that need a specific model entity context for their execution. Typically these methods operate on the entity on which they are called.

applyStereotype

```
ITtdEntity applyStereotype(ITtdEntity stereotypeToApply,
TtdReferenceKind referenceKind, ITtdEntity insertElement) throws
APIError;
```

Instantiates the given stereotype and applies it on the entity (referred to as the host entity). The qualifier, if any, in the reference to the stereotype is calculated based on the `referenceKind`, see table below for a description of possible values. If `insertElement` is given the stereotype is logically instantiated on the host entity, but physically placed on the `insertElement`. In that case the stereotype instance will point to the host entity. In this way, stereotype instances may be “applied” to an entity without modifying the entity itself. This technique is known as “stereotype injection”.

If the application of the stereotype fails an `APIError` exception is thrown.

TtdReferenceKind	Description
`TTD_RK_GUID`	The reference will only contain a GUID reference.
`TTD_RK_NO_QUALIFIER`	The reference will not be qualified. That is, it will only contain the name of the stereotype.
`TTD_RK_FULL_QUALIFIER`	The reference will contain a full qualifier.
`TTD_RK_MINIMAL_QUALIFIER`	The reference will contain the minimum qualifier needed to reference the stereotype. This option may at most return the same qualifier as `TTD_RK_RELATIVE_QUALIFIER` would. The presence of <<`access`>> or <<`import`>> dependencies may make it shorter.
`TTD_RK_RELATIVE_QUALIFIER`	The reference will be a relative qualifier to the stereotype. If there are no common upper scopes of the stereotype and the host entity, a full qualifier is calculated, otherwise a shorter qualifier starting from the nearest common scope is calculated.

Usually `TTD_RK_MINIMAL_QUALIFIER` is the recommended value to use, because it gives the same behavior as when the stereotype is applied using the Tau user interface.

例 734

This example creates a model with an actor inside. An actor is represented by an attribute stereotyped by the predefined `<<actor>>` stereotype. To verify that the operation was successful, the unparsed representation of the actor is printed.

```
ITtdModelAccess ma = TauModelAPI.getModelAccess();
ITtdModel model = ma.createModel();
ITtdEntity a = ((ITtdEntity) model).create("Attribute", false, "");
a.setValue("Name", "MyActor", 0);
ITtdEntity actorStereotype = model.findByGuid("@Predefined@actor");
a.applyStereotype(actorStereotype,
TtdReferenceKind.TTD_RK_MINIMAL_QUALIFIER, null);
System.out.println(a.unparse());
```

---

getValue

```
String getValue(String strMetaFeature, int index) throws
APIError;
```

Gets the value of the specified metafeature for the entity. If no metafeature has the specified name, an `APIError` exception will be thrown. The value is encoded as a string. Use it for metafeatures with single or multiple multiplicity. In the case of multiple multiplicity use the `index` to specify which value to get.

The `getValue` method can be used on all metafeatures of an entity that can have their values encoded as a string. This is the case for all metafeatures except derived features of メタクラス type, owner links and composition links. If such a metafeature is specified an `APIError` exception will be thrown.

注記

`index` is an index starting at 1, and 0 specifies the last entity in the metafeature collection.

例 735

Using `getValue` to retrieve some information about the predefined `PLUS_INFINITY` attribute:

```
ITtdModelAccess ma = TauModelAPI.getModelAccess();
ITtdModel model = ma.createModel();
ITtdEntity def = model.findByGuid("@Predefined@PLUS_INFINITY");
String name = def.getValue("Name", 0);
boolean isConst = def.getValue("changeability",
0).equals("CkFrozen");
String type = def.getValue("type", 0);
System.out.println(name + " is a " + (isConst ? "constant" : "non-
constant") + " attribute of type " + type);
```

Running this example will print the output:

```
PLUS_INFINITY is a constant attribute of type ref:Real
```

Note the format of the text encoded value of a metafeature that is a reference (like ‘type’ above). The name of the target definition will be prefixed according to the rules described in 第 58 章「COM API」の 1820 ページ、「GetValue」.

---

getEntity

```
ITtdEntity getEntity(String strMetaFeature, int index) throws
APIError;
```

Gets the value of the specified metafeature as an `ITtdEntity` reference. Use it for metafeatures of メタクラス type that have either single or multiple multiplicity. In the case of multiple multiplicity use the `index` to specify which entity to get. If no metafeature has the specified name, an `APIError` exception will be thrown.

注記

If the metafeature is unbound, the return value will be `null`. You may then want to use the getValue method to get the value as a string representation instead, or getReference to obtain the model representation of the unbound reference.

注記

`index` is an index starting at 1, and 0 specifies the last entity in the metafeature collection.

例 736

Using `getEntity` to retrieve the type of the predefined `PLUS_INFINITY` attribute:

```
ITtdModelAccess ma = TauModelAPI.getModelAccess();
ITtdModel model = ma.createModel();
ITtdEntity def = model.findByGuid("@Predefined@PLUS_INFINITY");
ITtdEntity type = def.getEntity("type", 0);
if (type != null)
  System.out.println("The type of PLUS_INFINITY is " +
type.getValue("Name", 0));
```

Running this example on a bound model will print the output:

```
The type of PLUS_INFINITY is Real
```

---

getEntities

```
List<ITtdEntity> getEntities(String strMetaFeature) throws
APIError;
```

Gets the value of the specified metafeature as a list of entities. Use it for metafeatures of メタクラス type that have either single or multiple multiplicity. In the case of single multiplicity the result list will contain one or zero entities. If no metafeature has the specified name, an `APIError` exception will be thrown.

See 例 725 on page 1994 for an example of using `getEntities` to obtain the list of libraries in a model.

getReference

```
ITtdEntity getReference(String strMetaFeature, int index)
throws APIError;
```

Returns the identifier representation of a metafeature reference. In the case of a metafeature with multiple multiplicity use the index to specify which reference to get. If no metafeature has the specified name, or the metafeature is not a reference, an `APIError` exception will be thrown.

注記

`index` is an index starting at 1, and 0 specifies the last entity in the metafeature collection.

例 737

This example constructs an attribute with a type reference that is not just a simple name. The model representation of the type reference is obtained by calling `getReference`, and its unparsed representation is printed.

```
ITtdModelAccess ma = TauModelAPI.getModelAccess();
ITtdModel model = ma.createModel();
List<ITtdEntity> lst = model.parse("Predefined::String<MyClass>
my_var;", "Definition");
for (ITtdEntity e : lst)
{
  ITtdEntity typeRef = e.getReference("Type", 0);
  System.out.println("The type of 'my_var' is " +
typeRef.unparse());
  e.delete();
}
```

Running this example prints the output:

```
The type of 'my_var' is Predefined::String<MyClass>
```

---

getOwner

```
ITtdEntity getOwner();
```

Returns the composition owner of the entity. If the entity has no owner `null` is returned.

例 738

This example locates the `equal` operation of the predefined `Integer` type. It then prints the composition owners of that operation.

```
ITtdModelAccess ma = TauModelAPI.getModelAccess();
ITtdModel model = ma.createModel();
ITtdEntity e =
model.findByGuid("@Predefined@Integer@equal@Boolean@Integer@Integer
");
do
{
  if (e.isKindOf("Definition"))
    System.out.println(e.getValue("Name", 0));
  else
    System.out.println(e.getMetaClassName());

  e = e.getOwner();
}
while (e != null);
```

Running this example prints the output:

```
equal
Integer
Predefined
Session
```

---

getMetaClassName

```
String getMetaClassName();
```

Returns the name of the entity's メタクラス .

See 例 738 on page 2003 for an example of using `getMetaClassName`.

getReferingEntities

```
List<ITtdEntity> getReferingEntities(String strMetaFeature);
```

Returns the entities that refer to the entity through the specified metafeature. `strMetaFeature` may be an empty string in order to find all referring entities regardless of through which metafeature they refer to the entity.

例 739

This example locates the predefined `Boolean` type, and examines the number of entities that refer to this type, either as their type or in another way.

```
ITtdModelAccess ma = TauModelAPI.getModelAccess();
ITtdModel model = ma.createModel();
ITtdEntity integer = model.findByGuid("@Predefined@Boolean");
List<ITtdEntity> r1 = integer.getReferingEntities("Type");
List<ITtdEntity> r2 = integer.getReferingEntities("");
System.out.println("There are " + r1.size() + " type references to
Boolean.");
System.out.println("There are " + (r2.size() - r1.size()) + " other
references to Boolean.");
```

getTaggedValue

```
ITtdEntity getTaggedValue(String strSelector, boolean
interpretIdentsAsGuids) throws APIError;
```

Returns the expression representing the tagged value selected by the selector pattern. The entity should be either an extendable element (in which case the tagged value is looked for among the applied stereotype instances of the extendable element) or an instance expression (in which case the tagged value is looked for in that particular instance only).

A selector pattern specifies the path from the entity to the tagged value using the textual UML syntax of an instance expression. The function will check that the pattern matches both the instance tree (by structure) and the corresponding signatures (by name or GUID). If `interpretIdentsAsGuids` is true, identifiers of the pattern will be interpreted as GUIDs. Otherwise they will be interpreted as names. If no tagged value is selected by the selector pattern, `null` is returned.

Some selector pattern examples:

```
"T1 (. .)"
```

will return the first instance of the applied T1 stereotype

```
"T1 (. x .)"
```

will return the tagged value of the attribute x in the T1 stereotype

```
"T1 (. a1 = T2 (. a2 .) .)"
```

will return the value of a2 in a T2 instance being the value of T1.a1.

注記

Although a selector pattern on the form "`X (. .)`" can be used to test if a stereotype X is applied on an entity, it is better to use hasAppliedStereotype for this purpose. The reason is that a stereotype that is automatically applied (due to a non-optional extension on a matching metaclass) will not be instantiated until at least one of its attributes gets a tagged value that differs from the default value of the attribute.`getTaggedValue` thus has no stereotype instance to return for that particular case. However, when used with a selector pattern that selects a tagged value, `getTaggedValue` will also consider such automatically applied stereotypes.

例 740

This example locates the `TTDMetamodel` representation of the `Model` metaclass. It then uses `getTaggedValue` to extract the 'base' tagged value of the <<metaclass>> stereotype.

```
ITtdModelAccess ma = TauModelAPI.getModelAccess();
ITtdModel model = ma.createModel();
ITtdEntity e = ((ITtdEntity)
model).findByName("TTDMetamodel::Model");
ITtdEntity taggedValue = e.getTaggedValue("metaclass (. 'base' .)",
false);
System.out.println("Model's base metaclass is " +
taggedValue.unparse());
```

Running this example prints the output:

```
Model's base metaclass is "Session"
```

hasAppliedStereotype

```
boolean hasAppliedStereotype(String strStereotype, boolean
guid);
```

Determines if the entity has a certain stereotype applied. The stereotype can be specified either by name or by GUID. In the latter case `guid` should be set to true.

This is the recommended method for checking for applied stereotypes on an entity. It will consider both explicitly applied stereotypes, and stereotypes that are automatically applied due to non-optional extensions from a metaclass that matches the metaclass of the entity.

例 741

This example prints the names of all libraries in a model, and whether the library is an ordinary library or a profile library. Profile libraries have the `<<profile>>` stereotype applied.

```
ITtdModelAccess ma = TauModelAPI.getModelAccess();
ITtdModel model = ma.createModel();
List<ITtdEntity> lst = ((ITtdEntity) model).getEntities("Library");
for (ITtdEntity e : lst)
{
  System.out.print(e.getValue("Name", 0) + " is ");
  if (e.hasAppliedStereotype("@Predefined@profile", true))
    System.out.println("a profile library");
  else
    System.out.println("an ordinary library");
}
```

isKindOf

```
boolean isKindOf(String strMetaClass);
```

Returns true if the entity is of the specified メタクラス , false otherwise.

`isKindOf` returns true also if the entity's metaclass inherits from the specified metaclass. To perform an exact match use getMetaClassName.

例 742

This example creates a new StateMachine entity. A StateMachine is a special kind of Operation.

```
ITtdModelAccess ma = TauModelAPI.getModelAccess();
ITtdModel model = ma.createModel();
ITtdEntity sm = model.New("StateMachine");
if (sm.isKindOf("StateMachine"))
  System.out.println("sm is a StateMachine");
if (sm.isKindOf("Operation"))
  System.out.println("sm is an Operation");
String metaclass = sm.getMetaClassName();
System.out.println("Exact metaclass is " + metaclass);
```

Running this example prints the output:

```
sm is a StateMachine
sm is an Operation
Exact metaclass is StateMachine
```

---

unparse

```
String unparse() throws APIError;
```

Returns the unparsed representation of the entity using textual UML syntax. The following kinds of entities can be unparsed:

- Definitions
- Actions
- Expressions

An attempt to unparse another kind of entity will yield an `APIError`. The text returned by unparse can later be passed to `ITtdModel::`parse in order to build the corresponding model entities from the syntax again.

See 例 737 on page 2003, 例 740 on page 2005 or 例 743 on page 2007 for an example of using `unparse`.

setValue

```
void setValue(String strMetaFeature, String strValue, int
index) throws APIError;
```

Sets the value of a metafeature. The value is encoded as a string.

The `setValue` method can be used on all writable metafeatures of an entity that can have their values encoded as a string. This is the case for all metafeatures except derived features (which are read-only), owner links and composition links.

If the metafeature has non-single multiplicity, the `index` parameter can be used to insert the value before the value at the specified position.

注記

`index` is an index starting at 1, and 0 specifies the last entity in the metafeature collection.

If an error occurs (e.g. because of a non-existing metafeature) an `APIError` exception is thrown.

例 743

This example creates an attribute in a model, and sets up the name and type of the attribute using `setValue`:

```
ITtdModelAccess ma = TauModelAPI.getModelAccess();
ITtdModel model = ma.createModel();
ITtdEntity attribute = ((ITtdEntity) model).create("Attribute",
false, "");
attribute.setValue("Name", "a", 0);
attribute.setValue("Type", "ref:Integer", 0);
System.out.println(attribute.unparse());
```

Running this example will print the output:

```
Integer a;
```

Note the format of the text encoded value of a metafeature that is a reference (like 'Type' above). The name of the target definition will be prefixed according to the rules described in 第 58 章「COM API」の 1820 ページ、「GetValue」.

重要！

Calling `setValue` on a reference metafeature implicitly deletes the entity that represents the reference (obtained by getReference). See 例 748 on page 2010 for more information.

setEntity

```
void setEntity(String strMetaFeature, ITtdEntity entity, int
index) throws APIError;
```

Sets the value of a metafeature. The value is an entity reference. Use it for all writable metafeatures of メタクラス type. If the metafeature has non-single multiplicity, the index argument can be used to insert the entity before the value at the specified position.

注記

`index` is an index starting at 1, and 0 specifies the last entity in the metafeature collection.

注記

If `entity` is `null`, the metafeature will be made unbound. However this does not apply for owner links. To unset an owner link, i.e. to unlink an entity from its composition owner, use unlinkFromOwner.

If an error occurs (e.g. because of a non-existing metafeature) an `APIError` exception is thrown.

例 744

This example creates a class from its textual syntax, and inserts the class into a model using `setEntity`.

```
ITtdModelAccess ma = TauModelAPI.getModelAccess();
ITtdModel model = ma.createModel();
ITtdEntity cls = model.parse("class C {}", "Definition").get(0);
((ITtdEntity) model).setEntity("OwnedMember", cls, 0);
if (cls.getOwner() == (ITtdEntity) model)
  System.out.println("Class successfully inserted in model!");
```

重要！

Calling `setEntity` *on a reference metafeature implicitly deletes the entity that represents the reference (obtained by getReference). See* *例 748 on page 2010* *for more information.*

setTaggedValue

```
void setTaggedValue(String strSelector, String strValue,
boolean overwrite) throws APIError;
```

Sets the tagged value of the attribute selected by the selector pattern. See getTaggedValue for the format of this pattern. The entity can either be an element with applied stereotypes or any instance expression. In the latter case the pattern is matched against the instance expression, while in the former case the first matching applied stereotype instance will be used.

Usually you want an existing value for the selected attribute to be overwritten, but if `overwrite` is false this will not be the case.

注記

In order to be able to overwrite an existing value for the specified attribute, the instance expression must be bound to the Signature of which it is an instance. If that is not the case, an `APIError` exception will be thrown. Use bind to make sure the instance expression is bound before calling `setTaggedValue`.

After the new value has been set all references in the entity is attempted to be bound, so that the set value can be accessed by getTaggedValue directly after the call to this method.

`strValue` must be a valid UML expression, and the type of this expression should match the type of the attribute for which the tagged value is set. For example, if the attribute is typed by `Charstring`, `strValue` should be a string literal.

例 745

This example creates a package in a new model. It then applies the <<icon>> stereotype on the package. Finally `setTaggedValue` is used to set the 'IconFile' tagged value.

```
ITtdModelAccess ma = TauModelAPI.getModelAccess();
ITtdModel model = ma.createModel();
ITtdEntity iconStereotype = ((ITtdEntity)
model).findByName("TTDStereotypeDetails::icon");
ITtdEntity pkg = ((ITtdEntity) model).create("Package", false,
"OwnedMember");
pkg.applyStereotype(iconStereotype,
TtdReferenceKind.TTD_RK_MINIMAL_QUALIFIER, null);
```

```
pkg.bind("");
pkg.setTaggedValue("icon (. IconFile .)",
"\"C:\\\\pics\\\\x.jpg\"", true);
System.out.println(pkg.unparse());
```

Running this example will print the output:

```
<<icon(.IconFile = "C:\\pics\\x.jpg".)>> package '' {

}
```

Note that since `IconFile` is a `Charstring` attribute, the tagged value must be enclosed in double quotes, and contained backslashes must be escaped.

---

create

```
ITtdEntity create(String strMetaClass, boolean
buildModelForPresentations, String strMetaFeature) throws
APIError;
```

Creates an entity of the specified metaclass as an immediate child of this entity. The created entity will be inserted in the specified metafeature. The metafeature can be left unspecified (an empty string) as long as the entity only contains one metafeature that can contain the created entity.

注記

The parameter `buildModelForPresentations` is only used in an interactive execution environment. It can then be set to true in order to automatically create the semantic entity behind a created presentation element (such as a symbol or a line). For example if creating a ClassSymbol, the corresponding Class would be automatically created.

例 746

This example creates a package with a class diagram in a new model.

```
ITtdModelAccess ma = TauModelAPI.getModelAccess();
ITtdModel model = ma.createModel();
ITtdEntity pkg = ((ITtdEntity) model).create("Package", false,
"OwnedMember");
ITtdEntity diagram = pkg.create("ClassDiagram", false, "");
```

Note that there are two metafeatures at the top model level where packages can be created: “Library” (for library packages) and “OwnedMember” for ordinary packages. Therefore we must specify the name of the metafeature in the first call to `create` above.

---

createInstance

```
ITtdEntity createInstance() throws APIError;
```

Call this method on entities that are signatures that can be instantiated, to create an instance of the signature. One use for this method is to create an instance of a stereotype in order to apply the stereotype on an element (however, applyStereotype is the easiest way to achieve this). Another use is for creating instance models.

In case the creation fails an `APIError` exception will be thrown.

注記

It is the responsibility of the client to take care of the returned entity. It should either be inserted into the model, or deleted, to avoid a memory leak. See Memory Management for more information.

`createInstance` implements the UML semantics of signature instantiation. If the signature contains parts with an initial cardinality these will also be instantiated recursively.

例 747

This example locates the predefined `<<cppHeaderFile>>` stereotype, and creates an instance of it. The unparsed representation of that instance is then printed.

```
ITtdModelAccess ma = TauModelAPI.getModelAccess();
ITtdModel model = ma.createModel();
ITtdEntity stereotype =
model.findByGuid("@TTDFileModel@cppHeaderFile");
ITtdEntity instance = stereotype.createInstance();
System.out.println(instance.unparse());
```

Running this example will print the output:

```
cppHeaderFile (.includeProtSettings = IncludeProtectionSettings
(..).)
```

`<<cppHeaderFile>>` has an attribute 'includeProtSettings' which is a part with multiplicity 1. Hence, when `<<cppHeaderFile>>` is instantiated a contained instance of `IncludeProtectionSettings` will be created too.

---

delete

```
void delete();
```

Deletes the entity. Memory allocated by the entity will be freed.

Note that the Java garbage collector only can manage the memory allocated by the JVM. Model entities must be deleted, when they no longer are needed, by calling `delete` (see Memory Management).

重要！

Do not access a reference to an entity after it has been deleted! Doing so usually leads to an access error and program crash.

In most cases it is not difficult to comply with the above rule, since deleting an entity is an explicit operation. However, there is one situation when use of other API methods will implicitly delete an entity. This happens when setting a value of a reference metafeature using setValue or setEntity as shown in the example below.

例 748

This example creates an attribute typed by `Integer`. The entity representing the type reference is then obtained by calling getReference. Then the attribute is retyped to `Boolean` using setEntity. This call will implicitly delete the `Integer` reference entity. Hence, the program must be careful not to access this entity afterwards.

```
ITtdModelAccess ma = TauModelAPI.getModelAccess();
ITtdModel model = ma.createModel();
ITtdEntity attribute = model.parse("Integer a;",
"Definition").get(0);
ITtdEntity typeRef = attribute.getReference("Type", 0);
attribute.setEntity("Type",
model.findByGuid("@Predefined@Boolean"), 0);
// System.out.println(typeRef.unparse()); <--- ERROR! typeRef is
deleted!
```

For an example of using an explicit call of `delete`, see 例 730 on page 1997.

XMLEncode

```
String XMLEncode() throws APIError;
```

Returns the XML encoding of the entity. If an error occurs, an `APIError` exception will be thrown.

See 例 731 on page 1997 for an example of how to use `XMLEncode`.

metaVisit

```
void metaVisit(ITtdMetaVisitCallback callbackInterface,
boolean visitAll) throws APIError;
```

Performs a メタモデル driven traversal of the model rooted at the entity. If `visitAll` is false libraries and the predefined package will be excluded from the traversal.

For each visited entity, the onVisitedEntity method will be called on the `callbackInterface` object. This method is called for an entity when the model traversal reaches that entity, but before its contained entities have been visited. When all contained entities have been visited, the onAfterVisitedEntity method will be called on the `callbackInterface` object. This allows actions to be performed on the “back recursion” of the traversal.

例 749

This example shows how the `MetaVisit` method can be used in order to print the name of all definitions in a model. First a class implementing the callback interface is defined:

```
public class DefinitionFinder implements ITtdMetaVisitCallback
{
  public boolean onVisitedEntity(ITtdEntity visitedEntity)
  {
    if (visitedEntity.isKindOf("Definition"))
    {
      try
      {
        System.out.println(visitedEntity.getValue("Name", 0));
      }
      catch (APIError e) {}
    }
    return true;
  }

  public void onAfterVisitedEntity(ITtdEntity visitedEntity)
```

```
{
}
```

`MetaVisit` can now be called on the model like this:

```
ITtdModelAccess ma = TauModelAPI.getModelAccess();
ITtdModel model = ma.createModel();
((ITtdEntity) model).metaVisit(new DefinitionFinder(), true);
```

For more information about the callback interface used with `MetaVisit`, see ITtdMetaVisitCallback.

bind

```
void bind(String strMetaFeature) throws APIError;
```

Attempts to bind the specified metafeature on the entity (or all metafeatures if `strMetaFeature` is empty). If a non-existing metafeature is specified, an `APIError` exception will be thrown.

例 750 ———————————————————————————————————————

This example creates an attribute typed by Integer. The type reference of the attribute is accessed before and after calling `bind` on the type reference.

```
ITtdModelAccess ma = TauModelAPI.getModelAccess();
ITtdModel model = ma.createModel();
ITtdEntity attribute = ((ITtdEntity) model).create("Attribute",
false, "");
attribute.setValue("Type", "ref:Integer", 0);
ITtdEntity type = attribute.getEntity("Type", 0);
System.out.println("type is " + ((type == null) ? "null" :
type.getValue("Name", 0)));
attribute.bind("Type");
type = attribute.getEntity("Type", 0);
System.out.println("type is " + ((type == null) ? "null" :
type.getValue("Name", 0)));
```

Running this example will print the output:

```
type is null
type is Integer
```

As can be seen the type reference is not bound to the `Integer` definition until after `bind` has been called.

---

locate

```
void locate() throws APIError;
```

This method is only available in an interactive execution environment.

The method locates visually the entity in the ModelView and/or diagrams. The effect in the IDE will be that the entity is shown in the model view (if possible) and in the diagrams (if a presentation for the entity exists in a diagram, that is).

If this method is used when the IDE is not available (in a non-interactive execution environment), an `APIError` exception will be thrown.

例 751

This example locates the predefined `Boolean` type.

```
ITtdModelAccess ma = TauModelAPI.getModelAccess();
ITtdModel model = ma.createModel();
ITtdEntity b = model.findByGuid("@Predefined@Boolean");
b.locate();
```

clone

```
ITtdEntity clone(boolean preserveBindings, boolean
preserveGuids) throws APIError;
```

Creates a clone of the entity.

The recommended value for both the parameters is `false`. This means that the clone will be unbound and have new unique GUIDs (i.e. the copy of the entity itself and the copy of all contained entities will get new unique GUIDs).

If `preserveBindings` is set to `true`, the clone will have the same bindings as the original entity. In order to be able to preserve bindings of the clone, the original entity must belong to a model (i.e. getModel called on the original entity must not return `null`).

If `preserveGuids` is set to `true`, the clone will have the same GUID as the original entity.

重要！

Be careful when cloning an entity without changing GUIDs. Such a clone should not be inserted into the same model as the original entity, or GUID conflicts will arise. If a model with GUID conflicts is saved, it might not be possible to load again.

If the cloning fails for one reason or another an `APIError` exception will be thrown.

注記

It is the responsibility of the client to take care of the returned entity. It should either be inserted into the model, or deleted, to avoid a memory leak. See Memory Management for more information.

例 752

This example creates an expression by parsing textual UML syntax. The resulting expression is then cloned and unparsed. Finally it is deleted to avoid a memory leak.

```
ITtdModelAccess ma = TauModelAPI.getModelAccess();
ITtdModel model = ma.createModel();
ITtdEntity expr = model.parse("1+x*(3+z)", "Expression").get(0);
ITtdEntity clone = expr.clone(false, false);
System.out.println(clone.unparse());
clone.delete();
```

Running this example will print the output:

```
1 + x * (3 + z)
```

move

```
void move(ITtdEntity newOwner, String metafeature, int index)
throws APIError;
```

Moves the entity from its current location in the model into the context of `newOwner`. If the entity would fit in more than one metafeature of the new owner, `metaFeature` must be specified to disambiguate. The `index` argument may be specified to control the position where to move the entity when the target metafeature has non-single multiplicity.

If the move fails an `APIError` exception will be thrown.

例 753

This example creates a new package “MyPackage” in a model. The package is created as an ordinary package (located in the “OwnedMember” composition). Then the package is moved into the “Library” composition of the model. It is inserted there as the 3rd library package. Finally the names of all library packages of the model are printed.

```
ITtdModelAccess ma = TauModelAPI.getModelAccess();
ITtdEntity model = (ITtdEntity) ma.createModel();
ITtdEntity pkg = model.create("Package", false, "OwnedMember");
pkg.setValue("Name", "MyPackage", 0);
pkg.move(model, "Library", 3);
List<ITtdEntity> lst = model.getEntities("Library");
for (ITtdEntity e : lst)
{
  System.out.println(e.getValue("Name", 0));
}
```

getModel

```
ITtdModel getModel();
```

Returns the model to which the entity belongs. If the entity does not belong to a model `null` is returned. This method is often the most convenient way to get an `ITtdModel` interface from the context of an `ITtdEntity` interface. It is particularly useful when the API client works with more than one model simultaneously.

例 754

This example gets the last library in a created model, and calls `getModel` to make sure it belongs to the same model that was created (which it of course will do).

```
ITtdModelAccess ma = TauModelAPI.getModelAccess();
ITtdEntity model = (ITtdEntity) ma.createModel();
ITtdEntity library = model.getEntity("Library", 0);
if (model == library.getModel())
  System.out.println(library.getValue("Name", 0));
```

unlinkFromOwner

```
void unlinkFromOwner();
```

Unlinks the entity from its current owner. The entity will not be deleted, and can for example be inserted in another place in the model, or in another model.

例 755

This example creates two models. In the first model a class is created. Then `unlinkFromOwner` is called to unlink the class from the first model, and `setEntity` is called to insert it in the second model instead.

```
ITtdModelAccess ma = TauModelAPI.getModelAccess();
ITtdEntity m1 = (ITtdEntity) ma.createModel();
ITtdEntity m2 = (ITtdEntity) ma.createModel();
ITtdEntity cls = m1.create("Class", false, "");
cls.unlinkFromOwner();
m2.setEntity("OwnedMember", cls, 0);
System.out.println(m1.getEntities("OwnedMember").size());
System.out.println(m2.getEntities("OwnedMember").size());
```

Running this example will print the output:

```
0
1
```

Note that the move method can be used to move an entity in one single step.

---

replace

```
void replace(ITtdEntity replacementEntity) throws APIError;
```

Replaces the entity with another entity, without deleting the original entity. If the replacement is not possible to perform, an `APIError` exception will be thrown.

注記

If the entity is an identifier representing a reference it will be replaced with a clone of `replacementEntity`, rather than `replacementEntity` itself.

例 756

This example creates a model with a class. It then replaces the class with an interface. The original class is deleted to avoid a memory leak.

```
ITtdModelAccess ma = TauModelAPI.getModelAccess();
ITtdModel m = ma.createModel();
ITtdEntity cls = ((ITtdEntity) m).create("Class", false, "");
cls.replace(m.New("Interface"));
cls.delete();
ITtdEntity i = ((ITtdEntity) m).getEntity("OwnedMember", 0);
System.out.println(i.getMetaClassName());
```

Running this example will print the output:

```
Interface
```

---

getContainerMetaFeature

```
String getContainerMetaFeature();
```

Returns the name of the metafeature in which the entity is contained. If the entity is orphan an empty string is returned.

例 757

This example creates a model with an operation. It then prints the names of the container metafeatures for different parts of the operation definition.

```
ITtdModelAccess ma = TauModelAPI.getModelAccess();
ITtdModel m = ma.createModel();
ITtdEntity op = m.parse("void foo(Integer p1 = 3);",
"Definition").get(0);

class P implements ITtdMetaVisitCallback
{
  public boolean onVisitedEntity(ITtdEntity visitedEntity)
  {
    System.out.println(visitedEntity.getMetaClassName() + " is
contained in metafeature " +
visitedEntity.getContainerMetaFeature());
    return true;
  }

  public void onAfterVisitedEntity(ITtdEntity visitedEntity) { }
};

op.metaVisit(new P(), true);
```

Running this example will print the output:

```
Operation is contained in metafeature
Parameter is contained in metafeature Parameter
IntegerValue is contained in metafeature DefaultValue
```

findByName

```
ITtdEntity findByName(String strName);
```

Finds an entity by a name (possibly qualified) from the context of the entity. `strName` should be a valid UML identifier.

If no entity is found, `null` is returned.

For examples of using `findByName`, see 例 740 on page 2005 and 例 745 on page 2008.

getDescriptiveName

```
String getDescriptiveName();
```

Returns a descriptive name of the entity. The description includes the メタクラス of the entity, its name (full signature for event classes) and its location in the model.

例 758

This example locates the predefined `Real` type, and prints its descriptive name:

```
ITtdModelAccess ma = TauModelAPI.getModelAccess();
ITtdModel m = ma.createModel();
ITtdEntity real = m.findByGuid("@Predefined@Real");
System.out.println(real.getDescriptiveName());
```

Running this example will print the output:

```
DataType 'Real' in Predefined
```

ITtdResource

The `ITtdResource` interface is implemented by the Resource class of the メタモデル , which represents a resource where a UML model could be persistently stored. Typically a Resource corresponds to a `.u2` file.

注記

Since a Resource also is an Entity, a cast from `ITtdResource` to `ITtdEntity` will always succeed.

save

```
void save() throws APIError;
```

Saves the model entities that are associated with the resource on which the method is called. For the common case when the resource represents a `.u2` file, this means that the file will be saved.

If the resource cannot be saved an `APIError` exception will be thrown.

例 759

This example creates a new model, containing a package with an interface, and saves it to a `.u2` file.

```
ITtdModelAccess ma = TauModelAPI.getModelAccess();
ITtdModel model = ma.createModel();
ITtdEntity modelRoot = (ITtdEntity) model;
ITtdEntity pkg = modelRoot.create("Package", false, "OwnedMember");
ITtdEntity cls = pkg.create("Interface", false, "");
ITtdResource resource = model.createResource("C:\\temp\\test.u2");
// Insert the created package pkg as a root of the resource
((ITtdEntity) resource).setEntity("Root", pkg, 0);
resource.save();
```

ITtdPresentationElement

The `ITtdPresentationElement` interface is implemented by the PresentationElement class of the メタモデル . A presentation element is an element with a graphical appearance, for example a symbol, line or diagram.

注記

Since a PresentationElement also is an Entity, a cast from `ITtdPresentationElement` to `ITtdEntity` will always succeed.

generateEMF

```
void generateEMF(String strFileName, int maxWidth, int
maxHeight, boolean optimizeForVectorGraphics, boolean
includeFrame) throws APIError;
```

Generates an EMF file (Enhanced Meta File) for the graphical appearance of a presentation element. The presentation element will have the same appearance in this EMF file as when shown in the Tau editors.

注記

This is a deprecated function. Use generateEMFEx instead.

注記

`generateEMF` *is only available in the interactive execution environment. An attempt to call it from another execution environment will yield an* `APIError` *exception.*

generateEMFEx

```
void generateEMFEx(String strFileName, int maxWidth, int
maxHeight, boolean includeFrame, int scaleFactor) throws
APIError;
```

Generates an EMF file (Enhanced Meta File) for the graphical appearance of a presentation element. The presentation element will have the same appearance in this EMF file as when shown in the Tau editors.

注記

`generateEMFEx` *is only available in the interactive execution environment. An attempt to call it from another execution environment will yield an* `APIError` *exception.*

例 760 ——————————————————————————————

This example executes a query to find all diagrams in a model. It then generates an EMF file for the first diagram found. A JPG file is also generated using generateImage.

```
ITtdModelAccess ma = TauModelAPI.getModelAccess();
ITtdModel model = ma.createModel();
ITtdEntity query =
model.findByGuid("@TTDQuery@ExecuteQueryExpression");
LinkedList<Object> agentParameters = new LinkedList<Object>();
agentParameters.add(new LinkedList<Object>());
agentParameters.add("GetAllEntities().select(IsKindOf(\"Diagram\"))
");
model.invokeAgent(query, (ITtdEntity) model, agentParameters);
List<Object> lst = (List<Object>) agentParameters.element();
if (lst.size() > 0)
{
  ITtdPresentationElement pe = (ITtdPresentationElement)
lst.get(0);
  pe.generateEMFEx("C:\\temp\\d.emf", 0, 0, true, 0);
  pe.generateImage(ImageKind.IK_JPEG, "C:\\temp\\d.jpg");
}
```

——————————————————————————————————————

generateImage

```
void generateImage(ImageKind imgKind, String strFileName)
```

```
throws APIError;
```

Generates an image file from the graphical appearance of a presentation element. The presentation element will have the same appearance in this image file as when shown in the Tau editors.

Valid values in the `ImageKind` enumeration, and their meaning, are listed in the table below:

ImageKind	Description
IK_JPEG	Generate a JPEG image file.
IK_BMP	Generate a BMP image file.
IK_GIF	Generate a GIF image file.
IK_TIFF	Generate a TIFF image file.
IK_TARGA	Generate a TGA (Targa) image file.
IK_DIB	Generate a device independent bitmap file.
IK_PCX	Generate a PCX image file.

For an example of how to use `generateImage` see 例 760 on page 2018.

ITtdSymbol

The `ITtdSymbol` interface is implemented by the Symbol class of the メタモデル. A symbol is a presentation element with a two-dimensional graphical appearance.

注記

Since a Symbol also is a PresentationElement and an Entity, a cast from `ITtdSymbol` to `ITtdPresentationElement` or `ITtdEntity` will always succeed.

setSize

```
void setSize(int width, int height);
```

Sets the size of the symbol. The unit of the width and height is 1/10:th of a millimeter. In a non-interactive execution environment `setSize` will just update the value of the 'size' metafeature for the symbol. In the interactive execution environment, however, `setSize` can also perform additional model changes related to the resize. This could for example happen if the symbol size has a semantic significance. It is therefore recommended to always use `setSize` in order to set the size of a symbol.

For an example of how to use `setSize` see 例 761 on page 2020.

setPosition

```
void setPosition(int x, int y);
```

Sets the position of the symbol. The unit of the x and y parameters is 1/10:th of a millimeter. In a non-interactive execution environment `setPosition` will just update the value of the 'position' metafeature for the symbol. In the interactive execution environment, however,

`setPosition` can also perform additional model changes related to the repositioning. This could for example happen if the symbol position has a semantic significance. It is therefore recommended to always use `setPosition` in order to set the position of a symbol.

例 761

This example creates a new model with a class diagram containing a class symbol. It then sets the size and position of this symbol.

```
ITtdModelAccess ma = TauModelAPI.getModelAccess();
ITtdEntity model = (ITtdEntity) ma.createModel();
ITtdEntity diagram = model.create("ClassDiagram", false, "");
ITtdSymbol symbol = (ITtdSymbol) diagram.create("ClassSymbol",
true, "");
symbol.setSize(400, 200);
symbol.setPosition(350, 200);
System.out.println(((ITtdEntity) symbol).XMLEncode());
```

ITtdExpression

The `ITtdExpression` interface is implemented by the Expression class of the メタモデル. It represents an expression in the model.

注記

Since an Expression also is an Entity, a cast from `ITtdExpression` to `ITtdEntity` will always succeed.

getType

```
ITtdEntity getType();
```

Computes and returns the type of the expression. If the type cannot be computed (for example because the expression contains unbound references, or because the expression is not part of a model) `null` is returned.

注記

The returned entity is not guaranteed to be part of the model. In some cases the type is not explicitly defined in the model, and in that case a temporary entity which represents the type will be returned. Be careful not to delete such a temporary entity. getModel can be used to determine if the returned entity is part of the model.

例 762

This example creates some expressions by parsing textual UML syntax. Each expression is inserted as the default value of an attribute in the model. Then its type is computed and printed.

```
ITtdModelAccess ma = TauModelAPI.getModelAccess();
ITtdModel model = ma.createModel();
ITtdEntity attribute = ((ITtdEntity) model).create("Attribute",
false, "");

class ParseAndPrintType
{
  ParseAndPrintType(ITtdEntity attribute, String s) throws APIError
  {
```

```
    ITtdModel model = attribute.getModel();
    ITtdExpression e = (ITtdExpression) model.parse(s,
"Expression").get(0);
    attribute.setEntity("DefaultValue", (ITtdEntity) e, 0);
    ITtdEntity type = e.getType();
    if (type != null)
      System.out.println("The type of " + s + " is " +
type.getValue("Name", 0));
  }
}

new ParseAndPrintType(attribute, "1+2+3");
new ParseAndPrintType(attribute, "2.4 * -2.78");
new ParseAndPrintType(attribute, "true and false");
```

Running this example will print the output:

```
The type of 1+2+3 is Integer
The type of 2.4 * -2.78 is Real
The type of true and false is Boolean
```

---

evaluateConstantIntegralExpression

```
int evaluateConstantIntegralExpression() throws APIError;
```

Evaluates the value of the expression, which is expected to be a constant integral expression. If it is not, or the evaluation fails for some other reason, an `APIError` exception will be thrown.

例 763

This example creates an integral expression by parsing textual UML syntax. The expression is then evaluated and the result is printed.

```
ITtdModelAccess ma = TauModelAPI.getModelAccess();
ITtdModel model = ma.createModel();
ITtdExpression e = (ITtdExpression) model.parse("95-8*2",
"Expression").get(0);
int res = e.evaluateConstantIntegralExpression();
System.out.println(((ITtdEntity) e).unparse() + " = " + res);
```

Running this example will print the output:

```
95 - 8 * 2 = 79
```

---

getInstanceChildExpression

```
ITtdExpression getInstanceChildExpression(String strName)
throws APIError;
```

Use this method on an instance expression (for example a stereotype instance, or the instance expression of a named instance) to obtain the right-hand side of a contained assignment, where the left-hand side of the assignment is an identifier matching `strName` (which thus should be a valid identifier).

If no matching child expression is found, `null` is returned.

In case of an erroneous usage an `APIError` exception will be thrown.

例 764

This example creates a named instance by parsing textual UML syntax. The values of its slots (which are special kinds of assignments) are printed.

```
ITtdModelAccess ma = TauModelAPI.getModelAccess();
ITtdModel model = ma.createModel();
ITtdEntity inst = model.parse("value x : C (. a = 12, b = true,
BASE::c = \"Hello\" .);", "Definition").get(0);
ITtdExpression e = (ITtdExpression) inst.getEntity("Instance", 0);
ITtdExpression s1 = e.getInstanceChildExpression("a");
ITtdExpression s2 = e.getInstanceChildExpression("b");
ITtdExpression s3 = e.getInstanceChildExpression("BASE::c");
System.out.println(inst.getMetaClassName() + " slot values are:");
System.out.println(((ITtdEntity) s1).unparse());
System.out.println(((ITtdEntity) s2).unparse());
System.out.println(((ITtdEntity) s3).unparse());
```

Running this example will print the output:

```
NamedInstance slot values are:
12
true
"Hello"
```

ITtdMetaVisitCallback

The `ITtdMetaVisitCallback` interface is a callback interface that clients using the metaVisit method must implement.

onVisitedEntity

```
boolean onVisitedEntity(ITtdEntity visitedEntity);
```

Called when using metaVisit to traverse a model. `visitedEntity` is the currently visited entity in the model. This method is called before visiting the composition children of `visitedEntity`. If false is returned traversal will not continue with the composition children of `visitedEntity`.

See 例 749 on page 2011 for an example of how to use `onVisitedEntity`.

onAfterVisitedEntity

```
void onAfterVisitedEntity(ITtdEntity visitedEntity);
```

Called when using metaVisit to traverse a model. `visitedEntity` is the currently visited entity in the model. This function is called after the composition children of `visitedEntity` have been visited, and can thus be used in order to perform actions on the “back recursion” of the model traversal.

例 765

This example traverses a parsed expression and prints some logging before and after each entity of the expression is visited.

```
ITtdModelAccess ma = TauModelAPI.getModelAccess();
ITtdModel model = ma.createModel();
ITtdEntity e = model.parse("1 + x", "Expression").get(0);
```

```
class P implements ITtdMetaVisitCallback
{
  public boolean onVisitedEntity(ITtdEntity visitedEntity)
  {
    System.out.println("Before visiting " +
visitedEntity.getMetaClassName());
    return true;
  }
  public void onAfterVisitedEntity(ITtdEntity visitedEntity)
  {
    System.out.println("After visiting " +
visitedEntity.getMetaClassName());
  }
};

e.metaVisit(new P(), true);
```

The expression has the following representation in the model:

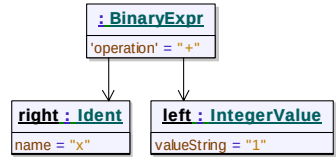


図 290: Model representation of expression 1 + x

Running this example will therefore print the output:

```
Before visiting BinaryExpr
Before visiting Ident
After visiting Ident
Before visiting IntegerValue
After visiting IntegerValue
After visiting BinaryExpr
```

Note that the order in which the composition children is determined by their order in the Tau metamodel. For example, from the above output we can conclude that the right hand side expression is defined before the left hand side expression of a binary expression.

ITtdMessageList

The `ITtdMessageList` interface represents a list of messages. It is used as a callback interface in API methods which may produce messages, for example loadProject.

addMessage

```
void addMessage(String text, MessageSeverity severity,
ITtdEntity subject) throws APIError;
```

Adds a new message to the list with the specified severity and, optionally, a subject entity. The subject entity will be associated with the message. In an interactive execution environment the subject entity can be located from the message.

Valid values in the `MessageSeverity` enumeration, and their meaning, are listed in the table below:

MessageSeverity	Description
`MS_INFORMATION`	The message is an information message.
`MS_WARNING`	The message is a warning message.
`MS_ERROR`	The message is an error message.
`MS_FATAL`	The message is a fatal error message.

See 例 724 on page 1994 for an example of using `addMessage`.

ITtdStudioAccess

The `ITtdStudioAccess` interface is the entry point for accessing functionality of the Tau IDE which is not specific to UML modeling.

注記

The `ITtdStudioAccess` interface is of no use in a non-interactive execution environment, since the Tau IDE is then not available. It can only be used when using Tau Access to access a running instance of Tau.

See Tau Access for information about how to obtain the `ITtdStudioAccess` interface.

openWorkspace

```
ITtdWorkspace openWorkspace(String path) throws APIError;
```

Opens a Tau workspace file (`.ttw` file) in the Tau IDE. All projects contained in the workspace will be loaded. If an existing workspace is open in the Tau IDE it will first be closed.

This method is the equivalent of opening a workspace using the **File** - **Open workspace** menu.

`path` is the path to the workspace to open. If it is a relative path, it will be interpreted as relative to the current working directory of Tau (which is typically the `bin` directory of the Tau installation).

If the workspace cannot be opened, an `APIError` exception will be thrown.

例 766

This example uses Tau Access to attach to an instance of Tau running on the local machine. The workspace `C:\MyWorkspace.ttw` is then loaded in that Tau instance.

```
ITtdTauAccess ta = TauAccess.getTauAccess();
ITtdStudioAccess sa =
ta.getTauApplicationAtNetworkLocation("localhost", 57000);
ITtdWorkspace wksp = sa.openWorkspace("C:\\MyWorkspace.ttw");
```

newWorkspace

```
ITtdWorkspace newWorkspace(String path) throws APIError;
```

Creates a new Tau workspace and associates it with a workspace file (`.ttw` file). If an existing workspace is open in the Tau IDE it will first be closed.

This method is the equivalent of creating a new blank workspace using the **File** - **New** menu.

`path` is the path where to save the workspace file. If it is a relative path, it will be interpreted as relative to the current working directory of Tau (which is typically the `bin` directory of the Tau installation).

If the workspace cannot be created or the workspace file cannot be saved in the specified location, an `APIError` exception will be thrown.

例 767

This example uses Tau Access to attach to an instance of Tau running on the local machine. A new workspace is then created in that Tau instance.

```
ITtdTauAccess ta = TauAccess.getTauAccess();
ITtdStudioAccess sa =
ta.getTauApplicationAtNetworkLocation("localhost", 57000);
ITtdWorkspace wksp = sa.newWorkspace("C:\\temp\\NewWksp.ttw");
```

openProject

```
ITtdProject openProject(String path) throws APIError;
```

Opens a Tau project file (`.ttp` file). The model associated with the project, if any, will be loaded.

This method is the equivalent of opening a project using the **File** - **Open** menu.

`path` is the path to the project to open. If it is a relative path, it will be interpreted as relative to the current working directory of Tau (which is typically the `bin` directory of the Tau installation). The string may contain URNs.

If the project cannot be opened, an `APIError` exception will be thrown.

例 768

This example uses Tau Access to attach to an instance of Tau running on the local machine. The project `C:\MyWorkspace.ttp` is then loaded in that Tau instance.

```
ITtdTauAccess ta = TauAccess.getTauAccess();
ITtdStudioAccess sa =
ta.getTauApplicationAtNetworkLocation("localhost", 57000);
ITtdProject prj = sa.openProject("C:\\MyProject.ttp");
```

getWorkspace

```
ITtdWorkspace getWorkspace();
```

Returns the workspace that is currently open in the Tau IDE. In case no workspace is open `null` is returned.

例 769

This example uses Tau Access to attach to an instance of Tau running on the local machine. The path of the workspace currently open in that Tau instance is printed.

```
ITtdTauAccess ta = TauAccess.getTauAccess();
ITtdStudioAccess sa =
ta.getTauApplicationAtNetworkLocation("localhost", 57000);
ITtdWorkspace currentWksp = sa.getWorkspace();
if (currentWksp != null)
  System.out.println("Current workspace is " +
currentWksp.getPath());
```

interpretTclScript

```
String interpretTclScript(String script) throws APIError;
```

Interprets a Tcl script in Tau. Which Tcl commands that are available for use in the script depends on what is loaded in Tau. As a general rule all Tcl commands prefixed with `std` are always available, while those that are prefixed with `u2` only are available when a UML model is loaded.

The result of the script interpretation is returned as a string.

In case of Tcl script interpretation errors, an `APIError` exception will be thrown.

例 770

This example uses Tau Access to attach to an instance of Tau running on the local machine. Tcl commands are executed to pop up a couple of message dialogs.

```
ITtdTauAccess ta = TauAccess.getTauAccess();
ITtdStudioAccess sa =
ta.getTauApplicationAtNetworkLocation("localhost", 57000);
String res = sa.interpretTclScript("std::MessageDialog -message
\"Cool, eh?\" -style yesno -icon question");
if (res.equals("6")) // Yes
  sa.interpretTclScript("std::MessageDialog -message \"I
agree!\"");
else if (res.equals("7")) // No
  sa.interpretTclScript("std::MessageDialog -message \"I
disagree!\"");
```

Refer to the Tcl API documentation for information about available Tcl commands.

getApplicationName

```
String getApplicationName();
```

Returns the Tau application product name. Different Tau products support different features, and this method is thus a means for a client to know which Tau features it can utilize.

例 771

This example uses Tau Access to attach to an instance of Tau running on the local machine. It then prints the name of the Tau application, as well as its PID, its version information and the name of the user running that Tau instance.

```
ITtdTauAccess ta = TauAccess.getTauAccess();
ITtdStudioAccess sa =
ta.getTauApplicationAtNetworkLocation("localhost", 57000);
System.out.println("Application name: " + sa.getApplicationName());
System.out.println("PID: " + sa.getApplicationPID());
System.out.println("Version: " + sa.getApplicationVersion());
System.out.println("User: " + sa.getApplicationUserName());
```

The output from running this example could for example be:

```
Application name: IBM Rational Tau
PID: 147260
Version: 4.1
User: mmo
```

---

getApplicationPID

```
String getApplicationName();
```

Returns the process id (PID) of the Tau instance. This method can for example be useful when there are multiple instances of Tau running on a machine, and a client wants to communicate with one particular of these instances. The PID returned by this method is then a means for distinguishing the different Tau instances.

See 例 771 on page 2026 for an example of using `getApplicationPID`.

getApplicationVersion

```
String getApplicationVersion();
```

Returns the version number of the Tau instance as a string. Different Tau versions support different features, and this method is thus a means for a client to know which Tau features it can utilize.

See 例 771 on page 2026 for an example of using `getApplicationVersion`.

getApplicationUserName

```
String getApplicationUserName();
```

Returns the name of the user (login name) who is running the Tau instance. This method can for example be useful when there are multiple instances of Tau running on a machine with more than one user logged onto it. Using this method a client can know if a certain Tau application instance runs under the same user as the client, or under some other user.

See 例 771 on page 2026 for an example of using `getApplicationUserName`.

ITtdWorkspace

The `ITtdWorkspace` interface represents a Tau workspace. It contains methods which operate on that workspace.

注記

The `ITtdWorkspace` interface is of no use in a non-interactive execution environment, since the Tau IDE is then not available. It can only be used when using Tau Access to access a running instance of Tau.

getPath

```
String getPath();
```

Returns the full path of the workspace file where the workspace is stored.

See 例 769 on page 2026 for an example of using `getPath`.

getProject

```
ITtdProject getProject(String path);
```

Returns a project with the specified path which is contained in the workspace. If no matching project is found, `null` is returned.

注記

When searching for a matching project, paths are not normalized, nor are contained URNs expanded. A project will only be found if its path matches the argument exactly.

例 772

This example uses Tau Access to attach to an instance of Tau running on the local machine. The current workspace in that Tau instance is searched for a project stored at `C:\temp\JAPI.ttp`. If such a project is found it is made the active project of the workspace. Finally it is verified that the found project now is the active project.

```
ITtdTauAccess ta = TauAccess.getTauAccess();
ITtdStudioAccess sa =
ta.getTauApplicationAtNetworkLocation("localhost", 57000);
ITtdWorkspace wksp = sa.getWorkspace();
ITtdProject prj = wksp.getProject("C:\\temp\\JAPI.ttp");
if (prj != null)
{
  wksp.setActiveProject(prj);
  if (wksp.getActiveProject() == prj)
    System.out.println(prj.getName() + " is now the active
project!");
}
```

getActiveProject

```
ITtdProject getActiveProject();
```

Returns the currently active project of the workspace. If no project is active, `null` is returned. This can only happen in case of an empty workspace.

See 例 772 on page 2028 for an example of using `getActiveProject`.

setActiveProject

```
void setActiveProject(ITtdProject project);
```

Sets a project as the active project of the workspace. There can be at most one active project in a workspace, so if another project was active previously, it will no longer be active after the call to `setActiveProject`.

See 例 772 on page 2028 for an example of using `setActiveProject`.

ITtdProject

The `ITtdProject` interface represents a Tau project. It contains methods which operate on that project.

注記

The `ITtdProject` interface is of no use in a non-interactive execution environment, since the Tau IDE is then not available. It can only be used when using Tau Access to access a running instance of Tau.

getPath

```
String getPath();
```

Returns the full path of the project file where the project is stored.

例 773

This example uses Tau Access to attach to an instance of Tau running on the local machine. The active project in that Tau instance is found, and its name and path are printed.

```
ITtdTauAccess ta = TauAccess.getTauAccess();
ITtdStudioAccess sa =
ta.getTauApplicationAtNetworkLocation("localhost", 57000);
ITtdWorkspace wksp = sa.getWorkspace();
ITtdProject prj = wksp.getActiveProject();
if (prj != null)
{
  System.out.println("Active project is " + prj.getName());
  System.out.println("Stored at " + prj.getPath());
}
```

getName

```
String getName();
```

Returns the name of the project. This is usually the name of the project file without path.

See 例 773 on page 2029 for an example of using `getName`.

getModel

```
IUnknown getModel();
```

Returns the UML model of the project. If the project has no UML model, `null` is returned.

The interface type `IUnknown` is a common super interface for all interfaces of the Java API. It is used here instead of ITtdModel to accommodate for non-UML projects. It can be casted to ITtdModel.

例 774

This example uses Tau Access to attach to an instance of Tau running on the local machine. The active project in that Tau instance is found, and its model is located. The names of the top-level definitions in that model are then printed.

```
ITtdTauAccess ta = TauAccess.getTauAccess();
ITtdStudioAccess sa =
ta.getTauApplicationAtNetworkLocation("localhost", 57000);
ITtdWorkspace wksp = sa.getWorkspace();
ITtdProject prj = wksp.getActiveProject();
if (prj == null)
  return;

ITtdModel model = (ITtdModel) prj.getModel();
if (model == null)
  return;

List<ITtdEntity> defs = ((ITtdEntity)
model).getEntities("OwnedMember");
for (ITtdEntity d : defs)
  System.out.println(d.getValue("Name", 0));
```

62

Tau Access

This chapter describes Tau Access which is a run-time API for Tau. This API allows clients to programmatically attach to a running instance of Tau, whether that instance runs locally on the same machine as the client program, or remotely on another machine in the same network. Once a client has attached to Tau it can access its features through the standard C++ and Java APIs.

Intended readers are developers of client applications that want to integrate or interact with Tau in one way or another. A basic knowledge of either Java or C++ is assumed throughout this chapter.

参照

第 61 章「Java API」の 1989 ページ、「Java API」
第 60 章「C++ API」の 1955 ページ、「C++ API」

Introduction

Tau Access provides an entry point to the Tau C++ and Java APIs from a client application external to Tau. Similar capabilities are provided by the COM API, but while the COM API is specific for the Windows platform, Tau Access works on both Windows and Unix platforms.

Implementation Principle

The implementation of Tau Access makes use of the Tau Web サーバー and the Tcl API. When the client calls a method in the C++ or Java APIs, Tau Access translates this into one or many Tcl commands. These commands are then sent to the remote Tau instance through an HTTP request which will be processed by the web server of that Tau instance. The Tcl commands are synchronously processed in the main thread of the remote Tau instance. Finally, when execution is completed, the Tcl script result is passed back to Tau Access in the HTTP response. Tau Access then translates the response back to a piece of Java or C++ data which is made available to the client program.

Many API methods return objects identified by interface references. Such objects are represented by proxy objects in the client application. The following information is stored in each proxy object:

- The instance of Tau where the real object resides. The Tau instance is identified by the host name of the computer where it runs, and the port used by its web server.
- The remote address of the real object. This is represented by its Tcl identifier.
- Run-time type information about the object. This is an optimization to avoid having the client ask Tau for this information whenever the client needs to know about the object kind.

The Tau Access Java API is built on-top of the Tau Access C++ API through the use of JNI (Java Native Interface).

The picture below illustrates the run-time situation when a Java client uses Tau Access to get a reference to the currently loaded workspace of Tau. ‘wksp’ is a Java object, while ‘proxy’ and ‘object’ are C++ objects.

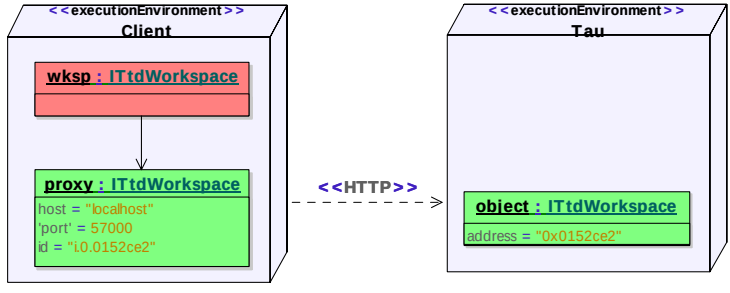


図 291: Run-time objects when using Tau Access for accessing a workspace

Calls of methods on ‘wksp’ are forwarded to ‘proxy’ and then finally, via HTTP, to ‘object’. Upon method return data flows in the opposite direction.

Using Tau Access

A client program that wants to use Tau Access should perform the following steps (depending on whether Java or C++ is used):

Java clients

1. Place the `tauaccess.jar` in the classpath. This JAR file can be found in the Tau installation at `/lib/Java`.
2. Set-up the environment variable `PATH` (`LD_LIBRARY_PATH` on Unix) so that it includes the Tau installation `bin` directory.
   Note that for Linux RedHat 5 `LD_LIBRARY_PATH` must also contain the Tau installation `bin/.rh5` directory, and this directory must be listed before the `bin` directory.

C++ clients

1. Include the header file `TauAccess.h` found in the Tau installation at `include/ToolAPI`. You may also want to include `StudioAccessInterfaces.h` and `U2ModelAccess.h` depending on which parts of the C++ API you intend to use.
2. Link with the libraries `TauAccessU.lib` and `SBL10U.lib` found in the Tau installation at `lib/<platform>`, where `<platform>` depends on the target platform (Windows / Solaris / Linux). If you have included `StudioAccessInterfaces.h` you should also link with `StudioU.lib`, and if you have included `U2ModelAccess.h` you should link with `U2DLLU.lib`.
3. Set-up the environment variable `PATH` (`LD_LIBRARY_PATH` on Unix) so that it includes the Tau installation `bin` directory.
   Note that for Linux RedHat 5 `LD_LIBRARY_PATH` must also contain the Tau installation `bin/.rh5` directory, and this directory must be listed before the `bin` directory.
4. Perform the additional standard steps for using the C++ API as described in C++ API のセットアップ .

Note that the C++ API デバッグ ユーティリティ are not available when using Tau Access.

A client using Tau Access is categorized as an interactive client, although it runs outside of the Tau executable. This is because the Tau IDE always is available in the Tau instance the client is connected to.

Like with the Tau Java API, Tau Access Java definitions are defined in the `com.telelogic.tau` package.

Like with the Tau C++ API, Tau Access C++ definitions are defined in the `u2` namespace.

API Entry Point

The entry point of using Tau Access is the `ITtdTauAccess` interface. This interface is obtained in the following way:

Java clients

```
ITtdTauAccess tauAccess = TauAccess.getTauAccess();
```

C++ clients

```
u2::ITtdTauAccess* pTauAccess = u2::GetTauAccess();
```

Object Lifetime Management

A C++ client of Tau Access must be explicit about when it no longer intends to use an interface it has obtained from the API. It does this by calling the function `DecRef()` on the interface when it no longer will use it. Internally Tau Access uses reference counting to know when a proxy object can be deleted from the memory of the client application. As a general rule the reference count for an object is incremented by the API function which returns a new interface to the client. It does this by calling the function `IncRef()` on the interface. The client is then responsible for calling `DecRef()` on the interface when it no longer needs to use it. It should also call `IncRef()` for every new reference it makes to the interface.

C++ clients that pass around interface pointers in non-trivial ways may consider using smart pointers for managing the reference count of the interface pointers. One smart pointer implementation that may be used is the `Ptr` template which is defined in the file `TauAPICommon.h`.

注記

Usually when using the Tau C++ API it is not necessary to call the `IncRef()` and `DefRef()` functions to maintain a correct reference count on objects. This is because Tau then manages the lifetime of the underlying objects. However, when using Tau Access calls of these functions are mandatory to prevent memory leaks.

A Java client of Tau Access does not need to bother about object lifetime management since Java has a garbage collector. Tau Access will automatically call `DecRef()` on an interface when it is finalized by the garbage collector. Hence both the Java and the C++ proxy objects are deleted automatically in this case.

Interface Casting

Interface casting when using Tau Access works in the same way as in traditional use of the C++ or Java APIs. See Interface Casting (Java) and インターフェイス キャスティング (C++) for more information.

Note that Tau Access keeps run-time type information in the proxy objects to allow interface casting on the client side without requiring communication with Tau.

Handling API Errors

Both the Tau Java and C++ APIs use an `APIError` exception for signalling errors that may occur when using the API. There is no difference when accessing these APIs through Tau Access. However, the client application should be prepared to handle `APIError` exceptions even for API methods which normally would never throw this exception. The reason is that with Tau Access there are always error situations that can arise from underlying HTTP or network problems regardless of the usual API-level errors.

Example

The example below shows how to use Tau Access for Java and C++ respectively, in order to attach to an instance of Tau running on the local machine (on port 57000). The active project in that Tau instance is found, and its model is located. The names of the top-level definitions in the model are then printed.

例 775

Java

```
ITtdTauAccess ta = TauAccess.getTauAccess();
ITtdStudioAccess sa =
ta.getTauApplicationAtNetworkLocation("localhost", 57000);
ITtdWorkspace wksp = sa.getWorkspace();
ITtdProject prj = wksp.getActiveProject();
if (prj == null)
  return;

ITtdModel model = (ITtdModel) prj.getModel();
if (model == null)
  return;

List<ITtdEntity> defs = ((ITtdEntity)
model).getEntities("OwnedMember");
for (ITtdEntity d : defs)
  System.out.println(d.getValue("Name", 0));
```

C++

```
using namespace u2;

ITtdTauAccess* pTA = GetTauAccess();
Ptr<ITtdStudioAccess> pSA(pTA-
>GetTauApplicationAtNetworkLocation(_T("localhost"), 57000));
Ptr<ITtdWorkspace> pWksp(pSA->GetWorkspace());
Ptr<ITtdProject> pPrj(pWksp->GetActiveProject());
if (!pPrj)
  return;

Ptr<ITtdModel> pModel(cast<ITtdModel>(pPrj->GetModel()));
if (!pModel)
  return;

std::list<ITtdEntity*> lst;
Ptr<ITtdEntity> pModelEntity(cast<ITtdEntity>(pModel.get()));
pModelEntity->GetEntities(_T("OwnedMember"), lst);
for (std::list<ITtdEntity*>::const_iterator it = lst.begin(); it !=
lst.end(); it++)
{
  Ptr<ITtdEntity> pDef(*it);
```

```
    tstring strName;
    pDef->GetValue(_T("Name"), strName);
    std::wcout << strName.c_str() << std::endl;
}
```

Note the use of the `Ptr` smart pointer to avoid explicit calls to the `IncRef()` and `DecRef()` functions.

Other API Differences

This section lists some further differences when using the Tau Java and C++ APIs through Tau Access, as compared to when using these APIs otherwise.

No callback interfaces

Tau Access does not support callback interfaces, such as for example `ITtdMetaVisitCallback` or `ITtdMessageList`. This is a consequence of the underlying HTTP protocol where all requests are initiated by the client and processed by the server (Tau). It is not possible for the server (Tau) to initiate a request to be processed by the client, as would have been required to support callback interfaces.

The workaround to this limitation is to define an agent which runs inside Tau, and which can be called by the Tau Access client. The agent can then make use of the callback interface and propagate necessary information to the client afterwards.

See エージェント for more information about agents.

Main thread serialization

All requests received by Tau from Tau Access clients will be serialized into the main execution thread. Therefore, if the main thread is blocked (for example because Tau is busy with performing some time consuming blocking operation) all Tau Access client requests will be hanging, waiting for Tau to be ready to process them.

Performance issues

Naturally the nature of sending API requests over HTTP between different applications imply some overhead, especially if the applications are running on different machines in a network. If performance becomes an issue for a Tau Access client one way to address the problem could be to refactor parts of the client implementation into an agent which is run inside Tau instead. Accessing the Tau API from an agent is significantly faster than doing the same access through Tau Access. Note, however, that currently an agent cannot be implemented in Java. Instead C++ is usually the best implementation choice for such agents.

See エージェント for more information about agents.

API Interfaces and Methods

This section describes the API interfaces and methods which are specific to Tau Access, and which are not covered by the general API documentation. See API Interfaces and Methods (Java) and API のインターフェイスと関数 (C++) for the documentation of the other parts of these APIs.

ITtdTauAccess

The `ITtdTauAccess` interface contains methods for accessing a running Tau application, or to start a new instance of Tau.

See API Entry Point to learn how to obtain the `ITtdTauAccess` interface from the client application.

GetRunningTauApplications

Java

```
List<ITtdStudioAccess> getRunningTauApplications() throws
APIError;
```

C++

```
virtual void
GetRunningTauApplications(std::list<ITtdStudioAccess*>&
instances) throw (u2::APIError) = 0;
```

Obtains a list of all Tau applications that are running on the local machine. Each Tau application is represented by an `ITtdStudioAccess` interface. Tau applications older than version 4.1 will not be part of the list.

If an error occurs while accessing the Tau applications, an `APIError` exception will be thrown.

例 776

This example prints the name and version of the Tau applications running on the local machine.

Java

```
ITtdTauAccess ta = TauAccess.getTauAccess();
List<ITtdStudioAccess> apps = ta.getRunningTauApplications();
for (ITtdStudioAccess tau : apps)
{
  System.out.println(tau.getApplicationName() + " (" +
tau.getApplicationVersion() + ")");
}
```

C++

```
using namespace u2;

ITtdTauAccess* pTA = GetTauAccess();
std::list<ITtdStudioAccess*> apps;
pTA->GetRunningTauApplications(apps);
for (std::list<ITtdStudioAccess*>::const_iterator it =
apps.begin(); it != apps.end(); it++)
{
```

```
  Ptr<ITtdStudioAccess> tau(*it, true);
  tstring strName, strVersion;
  tau->GetApplicationName(strName);
  tau->GetApplicationVersion(strVersion);
std::wcout << strName << _T(" (") << strVersion << _T(" )") <<
std::endl;
}
```

StartNewTauApplication

Java

```
ITtdStudioAccess startNewTauApplication() throws APIError;
```

C++

```
virtual ITtdStudioAccess* StartNewTauApplication() throw
(u2::APIError) = 0;
```

Starts a new instance of Tau on the local machine. An `ITtdStudioAccess` interface representing the launched Tau instance is returned.

In case multiple versions of Tau are installed on the machine, the version which matches the used Tau Access libraries will be launched. This means that a Tau Access client built against the libraries of one particular Tau version can only launch instances of that particular Tau version.

In case the launch fails, an `APIError` exception will be thrown.

例 777

This example launches a new instance of Tau on the local machine. It then prints the PID of the launched Tau application.

Java

```
ITtdTauAccess ta = TauAccess.getTauAccess();
ITtdStudioAccess sa = ta.startNewTauApplication();
System.out.println(sa.getApplicationPID());
```

C++

```
using namespace u2;

ITtdTauAccess* pTA = GetTauAccess();
Ptr<ITtdStudioAccess> pSA(pTA->StartNewTauApplication());
tstring strPID;
pSA->GetApplicationPID(strPID);
std::wcout << strPID << std::endl;
```

GetTauApplicationWithPID

Java

```
ITtdStudioAccess getTauApplicationWithPID(int pid);
```

C++

```
virtual ITtdStudioAccess* GetTauApplicationWithPID(PID pid) =
0;
```

Returns an `ITtdStudioAccess` interface representing a Tau application with the specified process ID running on the local machine. If no such Tau application exists `null` is returned.

Only Tau applications with version 4.1 or later can be found by this method.

GetTauApplicationAtNetworkLocation

Java

```
ITtdStudioAccess getTauApplicationAtNetworkLocation(String
host, int port);
```

C++

```
virtual ITtdStudioAccess*
GetTauApplicationAtNetworkLocation(const tstring& host,
unsigned int port) = 0;
```

Returns an `ITtdStudioAccess` interface representing a Tau application running on the specified host, with a web server using the specified port. If no such Tau application exists, `null` is returned.

Only Tau applications with version 4.1 or later can be found by this method.

See 例 775 on page 2045 for an example of using this method.

63

XML フレームワーク ライブラリ

この章では TTDXMLFramework ライブラリについて説明します。このフレームワークは、UML で XML 文書を使った作業を行うためのものです。このライブラリには、XML を UML にインポートするためや、UML から XML 文書を生成するための多数のエージェントが含まれています。これらのエージェントは、Tau に XML ベースのカスタム インポータやカスタム エクスポータを作成する際に、ユーティリティとしても使用できます。

XMLFramework アドインの有効化

アドインを有効にすると、TTDXMLFramework ライブラリがロードされます。以下の手順に従ってください：

1. ［ツール］メニューから［カスタマイズ］を選択します。
2. ［アドイン］タブをクリックして、［ XMLFramework］アドインをクリックします。
3. ［OK］をクリックします。

この操作を行うと、TTDXMLFramework パッケージがライブラリとしてロードされます。

XML 文書のインポート

ParseXMLFromFile エージェントを使用して XML 文書を UML 表記にインポートできます。エージェントには以下のパラメータが必要です：

- `file : Charstring`
  インポートする XML ファイル。
- `out model : ITtdEntity`
  XML 文書の UML 表記。
- `messages : ITtdMessageList[0..1]`
  メッセージ（エラー、警告など）が報告用に使用するオプションのリスト。このパラメータが省略されると、メッセージは［メッセージ］タブ（エージェントが対話式モードで実行されている場合）か、`stdout`（エージェントが対話式でない場合）に表示されます。

このエージェントのモデルコンテキストは、XML 文書の UML 表記の挿入先である「ITtdModel」である必要があります。

例 **778: XML** 文書のインポート

この Tcl スクリプトは XML ファイル、`x.xml` をインポートします：

```
set curProject [std::GetActiveProject]
set model [std::GetModels -kind U2 -project $curProject]
set agent [u2::FindByGuid $model
"@TTDXMLFramework@ParseXMLFromFile"]
set p [lappend p "x.xml" 0]
u2::InvokeAgent $model $agent $model p
```

XML 文書のモデル表現は、1 つの属性「contents」を保持するパッケージから構成されます。この属性のデフォルト値は、XML 文書の最上位のエンティティを表現する式のリストです。この最上位の式が、入れ子になった XML エンティティを表現する他の式を含んでいます。

例 **779: XML** 文書の **UML** 表記の解析

2054 ページの例 778 のスクリプトに以下の行を追加します：

```
set pkg [lindex $p 1]
output "[u2::Unparse $pkg]\n"
```

x.xml ファイルは以下のようになっています：

```
<HTML>
<HEAD><TITLE>The Title</TITLE>
</HEAD>
<BODY>
<H3>A Simple First Page</H3>
</BODY>
</HTML>
```

以下のような出力が［スクリプト］タブに表示されます：

```
package '' {
   XML::Entity contents = {HTML (."\n", HEAD (.TITLE (."The Title".), "\n".), "\n", BODY (."\n", H3 (."A Simple First Page".), "\n".), "\n".)};
}
```

UML での XML 構築子の表現の詳細については、XML の UML 表現 を参照してください。

XML 文書のエクスポート

The agent `WriteXMLToFile` エージェントを使用して XML の UML 表記を XML ファイルにエクスポートできます。エージェントのパラメータは以下のとおりです :

- `file : Charstring`
  書き込み先の XML ファイル名。
- `messages : ITtdMessageList[0..1]`
  メッセージ（エラー、警告など）が報告用に使用するオプションのリスト。このパラメータが省略されると、メッセージは［メッセージ］タブ（エージェントが対話式モードで実行されている場合）か、`stdout`（エージェントが対話式でない場合）に表示されます。このエージェントのモデルコンテキストは、ファイルに書き込む XML 文書の UML 表記である必要があります。この UML 表記は最上位のインスタンス式モデルです。

例 **780: UML** 表記からの **XML** のエクスポート

2054 ページの例 778 のスクリプトに以下の行を追加します :

```
set pkg [lindex $p 1]
set a [u2::GetEntity $pkg "OwnedMember"]
set l [u2::GetEntity $a "DefaultValue"]
set instance [u2::GetEntity $l "Expression"]
set agent [u2::FindByGuid $model
"@TTDXMLFramework@WriteXMLToFile"]
set p2 [lappend p2 "y.xml"]
u2::InvokeAgent $model $agent $I p2
```

生成されたファイル `y.xml` は、重要ではない空白の使い方を除いて、元の `x.xml` と同一である必要があります。

XML の UML 表現

1 つの XML 文書は、式を使用して UML で表現されます。最上位の XML エンティティは 1 つのリスト式で表現されます。最上位の XML エンティティごとに 1 つの式が対応します。

サポートされる XML 構築子とその UML 式としての表現は以下のとおりです。

タグ

XML タグは UML インスタンス式によって表現されます。タグの名称は、インスタンス式の UML クラス名に対応します。

タグの入れ子は UML インスタンス式の入れ子に対応します。

例 **781: XML** タグの表記

XML:

```
<HTML>
<HEAD>
</HEAD>
</HTML>
```

UML:

```
HTML (. HEAD (. .) .)
```

属性

1 つの XML 属性は割り当て式によって表現されます。たとえば、'=' を演算子とする二項式、などです。割り当ての左側値は、属性名に対応する識別子で、右側値は、属性の値に対応する文字列値です。

割り当て式はコンテナタグに対応したインスタンス式に挿入されます。

例 **782: XML** 属性の表記

XML:

```
<A HREF="foo" onClick="testSub"></A>
```

UML:

```
A (. HREF = "foo", onClick = "testSub" .)
```

テキストノード

1 つの XML テキストノードは、テキストノードのテキストを含んだ文字列値によって表現されます。

文字列値は、コンテナタグに対応したインスタンス式に挿入されます。

例 **783: XML** テキストノードの表記

XML:

```
<A HREF="foo">LinkText</A>
```

UML:

```
A (. HREF = "foo", "LinkText" .)
```

処理手順（**Processing Instruction**）

サポートされません。常に以下の初期処理手順が、生成される XML 文書に追加されます。

```
<?xml version="1.0" encoding="UTF-8"?>
```

コメント

1 つの XML コメントは括弧式で表現されます。括弧に囲まれる式は、コメントノードのテキストを含んだ文字列値です。

括弧式は、コンテナタグに対応したインスタンス式に挿入されます。

例 **784: XML** コメントの表記

XML:

```
<HTML>
<!--
This is a comment!
-->
</HTML>
```

UML:

```
HTML (. ("This is a comment") .)
```

64

Tau Web サーバー

IBM Rational Tau には Web サーバーとしての機能が備わっています。この機能によって、ブラウザなどの HTML クライアントからツールの機能をアクセスできます。本章では、Tau Web サーバーが認識できる URL とさまざまなウェブページからの使用法について説明します。

Tau Web サーバーの目的

Tau Web サーバーの目的は以下のとおりです。

- Tau（および Tau がホストする UML モデル）をネットワーク経由で使用する手段を提供します。Windows の COM API（DCOM) を使用したアクセス方法に加えて、より標準的でプラットフォーム中立な HTTP を使用する方法を提供します。
- HTML ページを Tau アドインの GUI として使用できるようになります。HTML ページは、標準の std::HtmlReport Tcl コマンドを使用して Tau の内部からオープンできます。
- Tau と他のツールとを統合できます。通信プロトコルとして HTTP を使用すると、他のツールとの間の情報のやりとりが可能になります。

Tau Web サーバーの設定

Tau Web サーバーは Tau のメイン実行形式である vcs.exe に含まれています。Tau が起動されると、Web サーバーも自動的に起動され、HTTP 要求を待ち受ける準備をします。デフォルトで、Web サーバーは TCP/IP ポート 57000 を使用します。ただし、すでに別の Tau のインスタンスが同じマシンで実行されている場合は、Web サーバーは、使用できるポートが見つかるまで、ポート番号を 1 づつ加算してゆきます。最終的に使用できるポート番号が見つからない場合は、エラーメッセージがメッセージタブに表示されます。

Tau Web サーバーが使用するポート番号を変更するには :

1. ［オプション］ダイアログを開きます（［ツール］>［オプション］)。
2. ［詳細オプションページを表示する］チェックボックスがチェックされていることを確認します。
3. ［詳細］オプションページで、［サーバー］リストボックスで Studio を選択します。したのツリー表示内で、**Studio - Settings - WebServer** を選択します。
4. **PortRangeBegin** オプションと **PortRangeEnd** オプションは、Tau Web サーバーが使用する TCP/IP ポートの範囲を指定します。ここで指定するポートがマシン上で使用可能となっていることを確認してください。マシン上で実行する Tau の最大インスタンス数を考慮して十分な大きさの範囲を指定してください。

Tau Web サーバーの使用法

Tau Web サーバーが正しく設定されて起動していることを確認するには、Web ブラウザに以下の URL を入力してアクセスします :

```
http://localhost:57000/
```

Tau Web サーバーのメインページが表示されて、サーバーが実行されていることを確認できます。このページには現在登録されている Web 要求ハンドラ もリストアップされます。

ただし、使用するポート番号を変更した場合や複数の Tau インスタンスが実行されている場合は、状況に応じた別のポート番号を指定してください。使用されているポート番号が不明の場合は、以下の手順で確認できます:

1. 新しい Tcl ファイルをオープンします（[ファイル] > [新規 ...] > [ファイル] > [Tcl file]）
2. オープンしたファイルに以下のコマンドを書き込みます。

   ```
   std::Output "[std::GetWebServerPort]"
   ```
3. このコマンドを実行します（[ツール] > [スクリプトの実行]）、Web サーバーのポート番号が［スクリプト］タブの出力域に表示されます。

以下の例では、ポート番号としてデフォルトの 57000 を使用していると仮定します。

URL 構文

Tau Web サーバーの URL の一般的な構文は以下のとおりです。

`http://localhost:`**< ポート番号 >/< ハンドラ名 >/< データ >?< パラメータ >**

ここで、< ポート番号 > は Web サーバーが使用している TCP/IP ポート番号、< ハンドラ名 > は Web 要求ハンドラ名、< データ > は要求ハンドラ向けのデータストリング、< パラメータ > は、アンパサンド（&）区切りの名前 - 値のペアから構成される通常の HTTP 要求パラメータです。

他の Web 要求ハンドラの詳細とその使用目的については、Web 要求ハンドラ を参照してください。

一般に、遠隔のマシン上で起動している Tau の Web サーバーにアクセスするには、上記の URL の “localhost” をそのマシンの IP アドレスかホスト名に置き換えてください。

Web 要求の遅延

デフォルトでは、< パラメータ > に指定した名前 - 値のペアは、選択した Web 要求ハンドラに即時に渡されます。ただし、Web サーバーそのものによって解釈される、以下の特殊なパラメータがあります。

```
_invocationDelay=<delay>
```

このパラメータが指定されていると、Web サーバーは指定した**ミリ秒**の間だけその Web 要求の処理を遅延します。Web 要求の遅延は、Web サーバーを非同期的にアクセスする場合（AJAX を使用した Web ページなど）に有用です。

Web 要求ハンドラ

Tau Web サーバーのメインページには現在登録されている Web 要求ハンドラを示す表があります。各ハンドラには、名前とサポートする URL 構文についての説明があります。Tau の状況によって、異なる要求ハンドラが表示されます。

利用できる Web 要求ハンドラを以下に説明していきます。

file（ファイル）

‘file’ web 要求ハンドラは、ファイルシステム上に格納されたデータを取得するために使用されます。典型的なデータとしては、HTML 文書、XML で記述された文書などがありますが、任意のテキストデータを取得可能です。

‘file’ web 要求ハンドラの URL 構文は以下のとおりです：

`file/<` ファイル名 `>`

< ファイル名 > は取得したいデータを持つファイルのパス名です。このパス名には Tau の URN を含めることができます。

例 **785: ‘file’ web** 要求ハンドラの使い方

Tau のユーザーアドインのディレクトリからの相対位置にある HTML ファイル、index.html、をオープンするには、以下のような URL を指定します：

```
http://localhost:57000/file/urn:u2useraddins:MyAddin/etc/index.html
```

‘file’ web 要求ハンドラは常に使用可能です。

agent（エージェント）

‘agent’ web 要求ハンドラは、Tau でエージェント機能を起動します。エージェントは現在アクティブになっているプロジェクトのモデルで起動されます。

‘agent’ web 要求ハンドラの URL 構文は以下のとおりです：

`agent/<` エージェント `id>(<` エージェントパラメータ `>)`

< エージェント id> には起動したいエージェントを指定します。指定できる値は、エージェントの GUID か完全修飾されたエージェント名です。< エージェントパラメータ > は、起動するエージェントに渡す実パラメータです。‘/’ に続くテキストは有効な U2 呼び出し式となっている必要があることに注意してください。つまり、U2 構文規則に則るために URL のこの部分の U2 識別子は、単一引用符で囲む必要があります。

エージェントの起動時には任意の U2 式を実パラメータとして使用できます。また、HTTP 要求パラメータを参照するには、パラメータ名の前に ‘$’ マークをつけます。HTTP 要求から起動されたエージェントに他の情報を渡すためのパラメータもあります。以下のような形式で指定します。

`$context::<` 変数 `>`

< 変数 > に指定できる値は以下のとおりです：

変数	説明
response	Web 要求への応答を表す out パラメータ。エージェントがこのパラメータに文字列を割り当てると、その文字列が応答テキストとなって Web クライアントに戻されます。
user_agent	HTTP サーバー変数 HTTP_USER_AGENT の値
method	HTTP サーバー変数 REQUEST_METHOD の値
accept	HTTP サーバー変数 HTTP_ACCEPT の値
accept_encoding	HTTP サーバー変数 HTTP_ACCEPT_ENCODING の値
status	Web 要求に対する HTTP 状況コードを表す out パラメータ。エージェントがこのパラメータに整数を割り当てると、その値が有効な HTTP 状況コードになります。たとえば、状況コード 404 は “Not Found” エラーを表すために使用されます。 デフォルトの状況コードは 200 であり、HTTP 要求が正しく処理されたことを示しています。
content_type	HTTP サーバー変数 HTTP_CONTENT_TYPE を表す out パラメータ。エージェントは content type 文字列をこのパラメータとして設定して、Web クライアントに対してどのようにこの文字列を処理すべきかを指示します。たとえば、応答データが HTML 文字列の場合は、content type “text/html” が使用されます。 デフォルトの content type は “text/plain” です。

例 **786: ‘agent’ web** 要求ハンドラの使い方

以下の URL は、GUID ‘@MyAgent’ を持つエージェントを起動します。‘@’ 文字があるため、GUID を単一引用符で囲む必要があります。

```
http://localhost:57000/agent/’@MyAgent’()
```

以下の URL は、パッケージ P にあるエージェント A を起動します。このエージェントは応答パラメータとして 1 つの文字列パラメータを使用しています。たとえば、この応答データはクライアントに返す HTML データなどです。

```
http://localhost:57000/agent/::P::A(‘$context::response’)
```

以下の URL は GUID ‘@MyAgent’ を持つエージェントを起動します。このエージェントは、整数値 14、ブール値リテラル false、HTTP 要求パラメータである文字列値 $par の計 3 つの実パラメータを受け取ります。par の値は “SomeValue” です。

```
http://localhost:57000/agent/’@MyGuid’(14, false,
‘$par’)?par=SomeValue
```

‘agent’ web 要求ハンドラは、最低でも 1 つの UML モデルをもつプロジェクトが開いている場合に使用できます。

例

以下に Tau Web サーバーの使用法についての例を紹介します。

例 **787: HTML** ページを生成するエージェント

Tau に ‘GeneratePage’ という名前のエージェントがあり、以下のような Tcl スクリプトで実装されているとします：

```
proc GeneratePage { triggeredBy timing context server
agentParameters } {
  upvar 1 $agentParameters ap
  set p “<html><head><h1>Page generated by
  agent!</h1></head></html>”
  set ap [lreplace $ap 0 0 $p ]
}
```

次に、以下のようなシンプルな HTML ファイル ‘MyPage.html’ を作成します：

```
<html>
<head>
<body>
<form action="/agent/::GeneratePage(‘$context::response’)"
method="get">
<input type="submit" class="button" name="pressme" value="Press
Me"/>
</form>
</body>
</html>
```

Web ブラウザに以下の URL を入力すると結果の Web ページが開きます：

```
http://localhost:57000/file/C:\MyPages\MyPage.html
```

例 **788: AJAX Web** ページから遅延 **web** 要求を送る方法

asynchroneous JavaScript (AJAX) を使った Web ページから遅延 web 要求を使用すると、Tau 内での変更に連動させて、動的に HTML ページを更新できます。まず、エージェント Let’s start by defining an agent ‘GetSelection’ を定義します。このエージェントは、Tau の［モデルビュー］ブラウザで選択されているエンティティのリストを返します：

```
proc GetSelection { triggeredBy timing context server
agentParameters } {
  upvar 1 $agentParameters ap
  set response [lindex $ap 0]
  set sel [std::GetSelection]
  set c 0
  foreach s $sel {
    if {$c != 0} {
      set response “$response, “
    }
    incr c
    if {[GetKind $s] != “U2” || $s == 0} {
      set r “(Non-U2 entity)”
    } else {
      set r [u2::GetMetaClassName $s]
```

```
    }
    set response “$response$r”
  }
  if {$response == ““} {
    set response “Nothing is selected!”
  }
  set ap [lreplace $ap 0 0 $response ]
}
```

このエージェントは、選択されているエンティティの種別についての情報を文字列としてフォーマットします。応答文字列が HTML 文字列でも XML でもなくプレーンなテキストであることに注意してください。

次に、HTML ページ ‘GetSelection.html’ を作成します。このページには、‘GetSelection’ エージェントを 1 秒間隔で定期的に起動して、起動の結果を表示する JavaScript があります。

```
<head>
<title>Ajax Selection Observer Agent</title>
<body>
<h1>Ajax Selection Observer</h1>
<script type="text/javascript">
/* Create a new XMLHttpRequest object to talk to the Web server
*/
var xmlHttp = false;
/*@cc_on @*/
/*@if (@_jscript_version >= 5)
try {
  xmlHttp = new ActiveXObject("Msxml2.XMLHTTP");
} catch (e) {
  try {
    xmlHttp = new ActiveXObject("Microsoft.XMLHTTP");
  } catch (e2) {
  xmlHttp = false;
  }
}
@end @*/
if (!xmlHttp && typeof XMLHttpRequest != 'undefined') {
  xmlHttp = new XMLHttpRequest();
}
function callServer() {
  // Build the URL to connect to
  var url =
"/agent/::GetSelection('$context::response')?_invocationDelay=10
00";
  // Open a connection to the server
  xmlHttp.open("GET", url, true);
  // Setup a function for the server to run when it's done
  xmlHttp.onreadystatechange = updatePage;
  // Send the request
  xmlHttp.send(null);
}
function updatePage() {
  if (xmlHttp.readyState == 4) {
    var response = xmlHttp.responseText;
    var resNode = document.getElementById("sel");
    if (resNode != null)
      resNode.innerHTML = response;
    callServer();
  }
```

```
}
callServer();
</script>
Selected elements in Tau:<b><p id="sel"></p></b>
</body>
</html>
```

このページがロードされると JavaScript の関数 ‘callServer’ が呼ばれます。この関数はエージェント GetSelection を起動する URL を構築して、エージェントが起動されたときに呼び出されるように関数 ‘updatePage’ を設定します。この処理は 1 秒後に行われます。これは、`_invocationDelay` パラメータに 1000 ミリ秒という値を設定しているからです。結果を得た後、ID’sel’ の DOM 要素を見つけて、内部 HTML を応答文字列に設定して、エージェントからの戻り値にします。続いて、関数 ‘callServer’ を再呼び出しして、新しい要求をスケジュールします。この仕組みによって、Tau 側で選択している要素が変わった場合に、その変更を定期的に監視して、変更にしたがって web ページの内容を更新できます。

以下のような URL でページをアクセスします :

```
http://localhost:57000/file/C:\MyPages\GetSelection.html
```

制約事項

Tau Web サーバーには以下の制約事項があります。

POST プロトコルの未サポート

Tau Web サーバーがサポートするのは、HTTP GET プロトコルのみです。

全タイプ共通のリファレンス ガイド

このセクションの各章では、Tau プロジェクトの全タイプに共通の機能のリファレンスを提供します。

65

便利なショートカット キー

このセクションでは便利なショートカット キーについて説明します。他の標準的なアプリケーションと同じようにアクセス キーを使用できます。

（UNIX）これは、Exceed ユーザーのみ適用されます。アメリカ以外のキーボードを使用している場合、アクセス キーを利用するためには Alt ボタンを正しくマッピングする必要があります。

ALT ボタンをマッピングするには

1. ［Start］ボタンをクリックし、［Programs］、［Exceed］にカーソルを合わせ、［xconfig］をクリックします。
2. 表示されたダイアログで、［Input］をダブルクックします。［Imput］ダイアログが表示されます。
3. ［Alt key］フィールドで、［To X］または［Right To Window, Left To X］を選択します。
4. ダイアログを閉じます。

注記

UNIX：X11 ウィンドウ マネージャによって、いくつかのショートカット シーケンス（ALT-X、CTRL-H など）が妨害され、このマニュアルでの説明とは異なるアクションが起動されることがあります。その場合、特定のショートカット シーケンスを妨害しないよう、X11 ウィンドウ マネージャを設定できます。ショートカットの詳細については、ウィンドウ マネージャのドキュメントを参照してください。

ワークスペースの操作

キーボード ショートカット	説明
Ctrl + N 次に Ctrl + Tab で［ワークスペース］タブに移動	新規ワークスペースを作成します。
Ctrl + O	既存のワークスペースを開きます。
テンキーのマイナス（-）記号	選択したエンティティのツリーを畳みます。
テンキーの乗算（*）記号	モデル ツリーを選択したレベルより 1 つ下に展開します。このキーを使用するたびに、さらに 1 つ下のレベルにツリーを展開できます。
テンキーのプラス（+）記号	選択を展開します。
Alt +4	モデル ビューの再構成。モデル フィルターを選択します。

プロジェクトの操作

キーボード ショートカット	説明
Ctrl + N 次に Ctrl + Tab で［プロジェクト］タブに移動	新規プロジェクトを作成します。
Ctrl + O	プロジェクトを開きます。

ファイルの操作

キーボード ショートカット	説明
Ctrl + N	新規ファイルを作成します。
Ctrl + O	ファイルを開きます。
Ctrl + P	アクティブなドキュメントを印刷します。
Ctrl + S	アクティブなドキュメントを保存します。

ファイル内での移動

キーボード ショートカット	説明
Ctrl + 下矢印	挿入ポイントを移動せずに、数行下にスクロールします。
Ctrl + End	挿入ポイントをファイルの最後に移動します。
Ctrl + Shift + G	［指定行に移動］ダイアログを開きます。
Ctrl+ Home	挿入ポイントをファイルの先頭に移動します。
Ctrl + 左矢印	挿入ポイントを 1 語左に移動します。
Ctrl + M	出力ウィンドウの［ナビゲータ］タブを開きます。
Ctrl + 右矢印	挿入ポイントを 1 語右に移動します。
Ctrl + 上矢印	挿入ポイントを移動せずに、数行上にスクロールします。
End	挿入ポイントを行の最後に移動します。
Home	挿入ポイントを行の先頭に移動します。

テキストの選択

キーボード ショートカット	説明
Ctrl + Shift + End	挿入ポイントの位置からファイルの最後までのテキストを選択します。
Ctrl + Shift + Home	挿入ポイントの位置からファイルの先頭までのテキストを選択します。
Ctrl + Shift + 左矢印	挿入ポイントの左の 1 語を選択します。
Ctrl + Shift + 右矢印	挿入ポイントの右の 1 語を選択します。
Shift + 下矢印	挿入ポイントの位置から 1 行下までのテキストを選択します。
Shift + End	挿入ポイントの位置から行の最後までのテキストを選択します。
Shift + Home	挿入ポイントの位置から行の先頭までのテキストを選択します。
Shift + 左矢印	挿入ポイントの左の 1 文字を選択します。
Shift + 右矢印	挿入ポイントの右の 1 文字を選択します。
Shift + 上矢印	挿入ポイントの位置から 1 行上までのテキストを選択します。

テキストの編集

キーボード ショートカット	説明
Ctrl + A	すべてを選択します。
Ctrl + C	コピーします。
Ctrl + F	アクティブ ファイル内で検索します。
Ctrl + H	置換します。
Ctrl + スペースバー Shift + スペースバー	名前の完成。カーソル位置までに現在の名前に合致する定義が検出された場合。複数が合致する場合は、[名前の完成] スクロール メニューが表示されます。
Ctrl + V	貼り付けます。
Ctrl + X	切り取ります。
Ctrl + Y	やり直します。
Ctrl + Z	取り消し
F1	現在の選択に関するテキスト構文のヘルプを表示します。
Shift + F8	コメントやユーザーが追加したフォーマットを破棄して、モデルからテキストを復元します。
Shift + 矢印キー	現在のテキスト選択範囲を拡大します。このためには現在テキストを選択している必要があります。
Shift + End	挿入ポイントの位置からテキスト行の最後までのテキストを選択します。
Shift + Home	テキスト行の先頭から挿入ポイントの位置までのテキストを選択します。

エディタのショートカット

キーボード ショートカット	説明
矢印キー	矢印の方向にあるシンボルを選択します。現在シンボルを選択している必要があります。
Ctrl + < ダイアグラムへのシンボル配置時にクリック >	同じタイプのシンボルを複数配置できます。シンボル ツールバーから最初にシンボルを 1 つ選択している必要があります。
Ctrl + < 状態機械 フローへのシンボル配置時にクリック >	フローにシンボルを挿入できます。フロー内の 1 つ前のシンボルまたはフローラインを選択している必要があります。

キーボード ショートカット	説明
Ctrl + < ダイアグラム内の単語をクリック >	定義に移動します。定義が含まれるダイアグラムがない場合、モデル ナビゲータが開きます。
Ctrl + < 状態機械 フロー内で選択したシンボルをダブルクリック >	フロー内の選択したシンボルから下をすべて選択します。分岐されたフロー（複数シグナル、分岐など）も選択されます。
Ctrl + < ホイールボタンを回転 >	ダイアグラムを水平方向にスクロールします（インテリマウス ポインティング デバイスが必要です）。
Ctrl + Alt + End	ダイアグラム ナビゲーション。ダイアグラム スコープ内で下に移動します。
Ctrl + Alt + Page Down Ctrl + Alt + Tab	ダイアグラム ナビゲーション。ダイアグラム スコープ内で次のダイアグラムに移動します。
Ctrl + 矢印キー	選択したシンボルを矢印の方向に 5 グリッド（目盛り単位）移動します。
Ctrl + Delete	モデルからの削除。プレゼンテーション要素と対応するモデル要素を削除します。他のプレゼンテーション要素がこのモデル要素に接続されている場合、これらのプレゼンテーション要素も削除されます。
Ctrl + テンキーの除算（/）記号	すべての操作を非表示にします。（クラス、タイマー、シグナル、インターフェイス、操作、状態機械、データ型、列挙などのシグニチャ シンボルに有効です。）
Ctrl + F3	同じモデル要素の次のプレゼンテーション要素に移動します。
Ctrl + テンキーのマイナス（-）記号	すべての属性とパラメータを非表示にします。（クラス、タイマー、シグナル、インターフェイス、操作、状態機械、データ型、列挙などのシグニチャ シンボルに有効です。）
Ctrl + テンキーの乗算（*）記号	すべての操作を表示します。（クラス、タイマー、シグナル、インターフェイス、操作、状態機械、データ型、列挙などのシグニチャ シンボルに有効です。）
Ctrl + テンキーのプラス（+）記号	すべての属性とパラメータを表示します。（クラス、タイマー、シグナル、インターフェイス、操作、状態機械、データ型、列挙などのシグニチャ シンボルに有効です。）
Ctrl + Shift + < ツールバーのシンボルをクリック >	相互作用概観図とアクティビティ図：シンボルの追加と方向の切り替えを行います。シンボルの位置は、現在選択されていない方向になります。（追加するには、シンボルが選択されている必要があります。）
Ctrl + Shift + 矢印キー	選択したシンボルを矢印の方向に 1 グリッド（目盛り単位）移動します。
CTRL + SHIFT + M	［プレゼンテーションの作成］ダイアログを開きます。
Ctrl + Tab	開いている次のダイアグラムに切り替えます。

キーボード ショートカット	説明
Ctrl + U	モデルを更新します（Active Modeler アドインが必要です）。
Ctrl+Alt + Home	ダイアグラム ナビゲーション。ダイアグラム スコープ内で上に移動します。
Ctrl + Alt + Page Up Ctrl + Alt + Shift + Tab	ダイアグラム ナビゲーション。ダイアグラム スコープ内で前のダイアグラムに移動します。
Esc、Delete < キャンバスを右クリック >	ラインの作成を中止します。
F2	選択したシンボルで編集モードに入ります。
F4	出力ウィンドウ内で次の選択に移動します。
Shift + < ツールバーのシンボルをクリック >	ダイアグラムでシンボルを作成して追加します。自動作成されたシンボルはグレー表示されます。（追加するには、シンボルが選択されている必要があります。）
Shift + 矢印キー	矢印の方向にあるシンボルを選択して、選択範囲に含めます。（現在シンボルを選択している必要があります。）
Shift + F4	出力ウィンドウウィンドウ内で前の選択に移動します。
Alt + 上矢印キー	モデル ビューで選択したノードを上に移動します。
Alt + 下矢印キー	モデル ビューで選択したノードを下に移動します。
Shift + Enter	モデル ビューで選択したダイアグラム要素のモデル要素を表示します。
F8	現在の選択を確認します。.
Ctrl + F8	現在の選択を確認しません。
Shift + スペースバー	自動作成。現在の選択に対して自動作成できる要素がすべて表示されます。自動配置を参照してください。
Ctrl + スペースバー	自動挿入。現在の選択の後ろに自動挿入できる要素がすべて表示されます。自動配置を参照してください。

比較とマージ

キーボード ショートカット	説明
Alt + 等号（=）	選択を比較します。
Alt + テンキーのプラス（+）記号	選択をマージします。

アプリケーション ビルダのショートカット

キーボード ショートカット	説明
Ctrl + Scroll Lock	ビルド プロセスを停止します。
F5	現在の構成を起動します。
SHIFT + F7	現在の構成を生成します。
F7	現在の構成をビルドします。
Shift + F5	実行を停止します。
CTRL + F7	構成を更新します。

モデル ベリファイヤ（**Model Verifier**）のショートカット

キーボード ショートカット	説明
Alt + Pause	実行の中断 コマンド
Alt + F10	ステップ ローカルコマンド
Ctrl + Shift + F5	最初からデバッグ セッションを再開します。
F5	実行
F9	選択したブレーク ポイントを挿入／削除します。
F10	ステップ オーバー コマンド
F11	ステップ イン コマンド
Shift + F10	次の遷移 コマンド
Shift + F11	ステップ アウト コマンド

ウィンドウ ナビゲーション

キーボード ショートカット	説明
Alt +1	全画面表示に切り替えます。
Ctrl + F2	カーソル位置の定義をモデル ナビゲータの［お気に入り］に入れます。
Ctrl + F4	アクティブ ウィンドウを閉じます。
Ctrl + Shift + Tab Ctrl + Shift + F6	前のウィンドウに移動します。
Ctrl + Tab Ctrl + F6	次のウィンドウに移動します。
Shift + F2	カーソル位置の定義のコンテキストにしたがって、モデル ナビゲータを表示します。

プロパティ エディタ

キーボード ショートカット	説明
Alt + Enter	プロパティ エディタを表示します。
Ctrl + BackSpace	所有者に移動。モデル ツリーのスコープを現在の選択の所有者に変更します。
Ctrl + Alt + C	コントロール ビューに切り替えます。
Ctrl + Alt + T	テキスト ビューに切り替えます。

ウィンドウとダイアログの表示／非表示

キーボード ショートカット	説明
Alt +0	ワークスペース ウィンドウを表示／非表示にします。
Alt +2	出力ウィンドウを表示／非表示にします。
Alt + Enter	プロパティ エディタを表示します。
CTRL + Q	選択に対して［クエリ］ダイアログを開きます。
F1	ヘルプを表示します。

ズーム / パン

キーボード ショートカット	説明
< ホイール ボタンを回転 >	ダイアグラムを垂直方向にスクロールします（インテリマウス ポインティング デバイスが必要です）。
< ホイール ボタンをダブルクリック >	100%表示にします。
Shift +< ホイール ボタンをダブルクリック >	エディタ ウィンドウに全体を表示します。
Shift + < ホイール ボタンを回転 >	ホイールの回転方向に従ってズーム イン／アウトします。ズーム インの中心はマウス ポインタの位置です。
CTRL + Shift + < ホイール ボタンを回転 >	1 本のラインを選択している場合、ダイアグラムがそのラインに沿ってスクロールします。どちらかのエンドポイントが表示の中央になるとスクロールが停止します（インテリマウス ポインティング デバイスが必要です）。

キーボード ショートカット	説明
テンキーのマイナス（-）記号	25% ズームアウトします。（ダイアグラムがアクティブの場合、またどの要素でもテキスト編集モードになっていない場合に有効です。）
テンキーのプラス（+）記号	25% ズームインします。（ダイアグラムがアクティブの場合、またどの要素でもテキスト編集モードになっていない場合に有効です。）
左不等号（<）	1 本のラインを選択している場合、そのラインのソース エンドポイントの方向にダイアグラムがスクロールします。
右不等号（>）	1 本のラインを選択している場合、そのラインのデスティネーション エンドポイントの方向にダイアグラムがスクロールします。

66

ツール環境の設定

このセクションでは、主として Tau と各種ツールとのインテグレーション方法について説明します。

プログラミング ツールのインポート：

- UML からのインポート
- SDL からのインポート
- XMI からのインポート

構成管理ツール：

- IBM Rational Synergy
- IBM Rational ClearCase

ディレクトリ サービス

- IBM Rational Directory Server（RDS）

インポート ウィザード

Tau は、独自の Tau モデル情報以外の他のフォーマットから、モデル データをインポートするというさまざまないくつもの可能性をサポートします。

インポート スキームは、［ファイル］メニューの［インポート］コマンドから起動します。これによってインポート ウィザードが起動されます。インポートの基本手順は以下のとおりです。

- 種類（C/C++、SDL、XMI）を選択する。
- インポート元のソース ファイルを追加する。

インポートの結果、ソース ファイル情報に基づいて、要素とダイアグラムを含んだ新規のパッケージが現在のモデル内に作成されます（該当する場合）。

参照

第 9 章「C/C++ のインポート」
第 14 章「UML インポート」
第 11 章「SDL のインポート」

構成管理

Tau では、構成管理ツールによって 2 つのインテグレーション スキームがサポートされます。

- 完全な Synergy とのインテグレーション。このインテグレーションにより、Tau ユーザー インターフェイスを使用して直接 Telelogic Synergy の機能を使用することが可能になります。
- Microsoft Source Control Integration Interface をベースとしたインテグレーション スキーム。これは、使用するソース コントロール システムが Microsoft Source Control Integration Interface をサポートしていれば、Tau でも機能することを意味します。現在、IBM Rational ClearCase とのインテグレーションがサポートされており、Microsoft Source Control Integration Interface を使用する動作が定期的に検証されています。
- 構成管理ツール内で直接選択した u2 ファイル バージョンに対して、Tau の比較とマージ操作を起動するユーティリティ。

ソース コントロール プロバイダ

［ツール］メニューの［一般］タブで［オプション］を選択すると、［ソースコントロール プロバイダ］ドロップ ダウン メニューからソース コントロール スキームを選択できます。この設定を有効にするには Tau を再起動する必要があります。

参照

ソース コントロール情報

複数の構成管理ツール

ソース コントロール コマンド

ソース コントロール情報

構成管理ツールのファイルの状態は、［ファイル ビュー］でいくつかのアイコンによって表示されます。

- **青色のチェック マーク**
  このアイコンは、ファイルまたはフォルダがチェックアウトされていることを示します。
- **オレンジのチェック マーク**
  このアイコンは、フォルダの一部がチェックアウトされていることを示します。これは、フォルダにチェックインされたファイルとチェックアウトされたファイルが存在するか、フォルダにソース コントロールに加えられていないファイルが含まれていることを示しています。
- **赤色の x**
  このアイコンは、ファイルまたはフォルダがチェックインされていることを示します。

- **アイコンなし**
  アイコンがない場合は、ファイルまたはフォルダが構成管理ツール データベースに加えられていないことを示します。

この状態はファイルのプロパティ ページにも表示されます。

Synergy インテグレーション

Synergy とのインテグレーション

このセクションでは、Telelogic Synergy と Tau とのインテグレーション方法について説明します。詳細については、Telelogic Synergy のユーザー ガイドを参照してください。

Telelogic Synergy インテグレーションを使用する場合、Synergy のタスクとオブジェクトに関する 2 つのツール バーと 1 つのメニュー［Synergy］を使用できます。ツール バーとメニューには、Tau ユーザー インターフェイスから Telelogic Synergy の機能を使用するため、以下のようなコマンドがあります。

Telelogic Synergy でのインテグレーションでは、一連の定義済み Tau ファイル タイプ定義 も提供されます。

プロジェクトの処理

- 管理対象オブジェクトのオープン
- プロジェクトの移行
- プロジェクトの履歴
- プロジェクト プロパティ
- プロジェクトのマージ
- 同期
- プロジェクトの更新

タスクの処理

- タスクの作成
- タスクの設定
- タスクの完了
- タスク プロパティ
- ［カレント タスク］ボックス

オブジェクトの処理

- オブジェクトの作成
- プロパティを表示
- 履歴
- オブジェクトのチェックアウト

- オブジェクトのチェックイン
- オブジェクトのチェックアウトの取消し

バージョンの処理

- ステータスの更新

Tau ファイル タイプ定義

2 つの定義済みファイル タイプ定義を使用できます。それらによって Telelogic Synergy の Tau プロジェクトとモデル ファイル タイプが定義されます。2 つのファイル

- tau_project.xml
- tau_model.xml

がディレクトリ 「integrations/SynergyCM」にあり、Tau プロジェクト ファイル（.ttp）と Tau モデル ファイル（.u2）の定義が含まれています。

定義は Telelogic Synergy タイプ マネージャによって関連するデータ ベースに適用されます。

タイプ定義の適用方法の詳細は、`typedef` コマンドに関する Telelogic Synergy ヘルプで説明されています。

Synergy インテグレーションのインストール

Windows

注記

インテグレーションを起動するには、別のインストーラによって Synergy インテグレーションをインストールする必要があります。

使用するソース コントロール システムはシステム レジストリで指定します。Synergy をインストールすると、このソース コントロール システムのレジストリ キーが自動的に設定されます。ただし、複数の構成管理ツールをローカルでインストールする場合は、この値を手動で編集する必要があります。Synergy の開始前に実行すべきその他の手順は 1 つだけです。

1. Tau を起動します。
2. ［ツール］メニューから［オプション］をクリックします。
3. ［一般］タブで、［ソース コントロール プロバイダ］から［Synergy インテグレーション］を選択します。
4. ［OK］をクリックします。

次回 Tau を起動すると、Synergy へのログインを実行するかどうかの確認を求められます。さらに、前述のように、2 つの新しいツールバーと、新しいメニュー［Synergy］が使用できるようになります。また、出力ウィンドウに［Synergy ］タブが表示されます。

参照

複数の構成管理ツール

Synergy へのログイン

ソース コントロール システムとして Synergy を使用し、［ソース コントロール プロバイダ ］オプションを［Synergy インテグレーション］に設定している場合、Tau の起動時に必ず Synergy サーバへログインするダイアログが表示されます。ログイン ダイアログには以下のフィールドがあります。

1. ユーザ名：Synergy で使用するユーザー ID。
2. パスワード：Synergy で使用するパスワード。
3. データベース パス：データベースの場所を示す Synergy サーバ上のパス。
4. エンジン ホスト：Synergy サーバが置かれているコンピュータの名前。
5. Synergy ホーム：Synergy ワークスペースを示すローカル マシン上のパス。

ログイン ダイアログをキャンセルした場合（または［ソース コントロール プロバイダ］オプションを設定していない場合）Tau を再起動しないと Synergy サーバにログインできません。

ログインをキャンセルすると、Tau Synergy インテグレーションも無効になります。

注記

一定時間 Synergy 機能を使用しないと、リソース確保するために Synergy サーバへの接続が一時的にシャットダウンされます。必要が生じた場合、自動的に再接続されます。

Synergy プロジェクトの処理

既存の Synergy プロジェクトを開く

Tau から Synergy データベースに格納されている UML プロジェクトに直接アクセスできます。これは、［Synergy］メニューの［管理対象オブジェクトのオープン］を使用して実行できます。

また、ローカル Synergy ワークスペース内のプロジェクトは、通常の［Open］コマンドを使用して、ローカル Synergy ワークスペースを参照し、その中の UML プロジェクトを選択して開くこともできます。

注記

Tau では、Synergy プロジェクト名と同名の Tau プロジェクト（.ttp ファイル）のみ管理対象プロジェクトと見なされます。Synergy プロジェクトに複数の Tau プロジェクトが含まれる場合、［ファイル］->［開く］を選択して、Tau プロジェクトを参照して、プロジェクトを開く必要があります。

注記

ワークスペース ファイル （`.ttw`）は Synergy の管理対象ではありません。

新規 **Synergy** プロジェクトの作成

新しい UML プロジェクトを作成し、Synergy プロジェクトに格納できます。これは、以下の方法で実行します。

- ［ファイル］->［新規］コマンドを使用してプロジェクトを作成します。
- 表示されるウィザードで、［Synergy プロジェクト］タブを選択します。使用できるプロジェクト タイプのリストが表示されます。
- プロジェクト タイプのいずれかを選択して、プロジェクトの名前を指定します。
- リリースを指定します。
- プロジェクトの種類を「目的 : ］に指定します。
- ［OK］ボタンをクリックします。

表示されたウィザードでプロジェクトの詳細を指定します。ウィザードを終了すると、新しいプロジェクトが作成されます。「Creating project <name of project>」という名前の Synergy タスクが自動的に作成されます。モデルの初期バージョンを作成してタスクを完了している場合、モデルは Synergy リポジトリに格納されます。

タスクの自動処理

Synergy インテグレーションには、Synergy タスクの処理を自動化する機能があります。オブジェクトのチェックアウトなど、タスクを必要とする操作を行う場合、タスクを作成するダイアログが表示されます。このダイアログでタスク名を定義したり、操作をキャンセルできます。

Synergy プロジェクトのコマンド

このセクションで説明するコマンドは、すべて Tau で現在アクティブなプロジェクトで使用します。

複数のコマンドを選択でき、適用したコマンドは選択したファイルに適用されます。

管理対象オブジェクトのオープン

このコマンドにより、Synergy サーバで管理されるプロジェクトのリストを含むダイアログが表示されます。Synergy プロジェクトを 1 つ選択すると、Synergy プロジェクト内で Synergy プロジェクトと同じ名前を持つ UML プロジェクトが Tau によって自動的にロードされます。Tau にワークスペースが既にある場合、プロジェクトはこのワークスペースに追加されます。ワークスペースがない場合は新しいワークスペースが作成されます。

コマンドを終了すると、UML プロジェクトに準拠するファイルが作成され、ローカルの Synergy のワークエリアおよび Tau のモデルで使用できるようになります。ローカル ワークエリアに Synergy プロジェクトがある場合、そのプロジェクトが開きます。プロジェクトがない場合、Synergy サーバのプロジェクト内のファイルがローカル ワーク エリアにコピーされます。

注記

プロジェクトが静的状態にある場合（インテグレートまたはリリース）、プロジェクトがユーザーのワーク エリアにある場合でも、“Open Managed Project” 操作によってプロジェクトをコピーできます。静的状態にあるプロジェクトを開くには、［ファイル］-> ［開く］または［ファイル］-> ［ワークスペースを開く］を選択します。

プロジェクトの移行

このコマンドにより現行プロジェクトを Synergy プロジェクトに移行します。Tau プロジェクトを構成する各ファイルのコピー（およびディレクトリ構造）が Synergy プロジェクトに作成されます。

注記

古いプロジェクトもロードされます。

［プロジェクトの移行］コマンドを選択するとダイアログが表示されます。このダイアログで UML ファイルを含む Synergy プロジェクトを指定できます。このコマンドにより、以下のアクションが実行されます。

1. Synergy タスク「Migrating <project name> to Synergy」の作成。
2. Synergy プロジェクトの作成。
3. UML プロジェクトの全ファイルの Synergy データベースへの移動。
4. タスクの完了。

注記

Synergy プロジェクトは、デフォルトで Tau プロジェクトの名前になります。したがって、既存の Synergy プロジェクトとの間で名前衝突が生じた場合、別の名前を付けるよう指示されます。これに応じて、Tau プロジェクトの名前を変更します。

プロジェクトの履歴

このコマンドにより、現行プロジェクトを含む Synergy プロジェクトの構成管理履歴を示すグラフを表示できます。グラフの詳細は、使用する Synergy のバージョンによって異なります。

プロジェクト プロパティ

このコマンドにより、現行の Synergy プロジェクトの構成管理プロパティを示すダイアログを表示できます。ダイアログの詳細は、使用する Synergy のバージョンによって異なります。

プロジェクトのマージ

並行開発を行っている場合、複数の開発者によって変更されたモデルのマージが必要なことがあります。Synergy を使用している場合、通常は以下の手法をとります。

特定リリースをベースにプロジェクト（この場合は UML プロジェクト）を開発します。

定期的に（ビルド成功後など）、プロジェクトで使用する全ファイルを含むベースライン バージョンを作成します。

それぞれの開発者には、ベースライン バージョンをベースとした Synergy の独自の開発プロジェクト バージョンと、最新の変更があります。

統合テストには、特別なプロジェクト バージョンが使用されます。これもプロジェクトのベースライン バージョンをベースにしていますが、最後のベースライン以降にチェックインされた全ファイルの新バージョンを含むように設定されています。

各開発者が自分の変更をチェックインする前に、必ずマージを行って統合プロジェクト バージョンからすべての変更を取得することを推奨します。

Tau でこのマージを行うには、ベースライン プロジェクト バージョンと統合プロジェクト バージョンへのアクセス権が必要です。以下の方法で指定します。

- ［プロジェクトのマージ］コマンドを選択します。
- 表示されたダイアログで、マージするプロジェクト バージョンと共通の世代を選択します。このダイアログで、現行タスクを選択したり、マージ操作に固有のタスクを作成することもできます。［OK］をクリックして操作を継続します。
- 相違点ダイアログでのレビュー が表示されます。このダイアログでは［世代］と［バージョン 2］があらかじめ選択されています。
- 通常のマージを行います。マージを実行してすべての競合が解決したら、タスクを完了して変更後のファイルをチェックインできます。

マージによって変更された全ファイルの履歴リンクは、実行されたマージを反映して自動更新されます。

同期

このコマンドにより、ローカル Synergy ワークスペースを Synergy サーバの対応するワークスペースと同期させることができます。

プロジェクトの更新

このコマンドにより、Synergy サーバの最新バージョンを反映して現行プロジェクト内のリソースを更新できます。

Synergy タスクのコマンド

タスクの作成

このコマンドを選択すると、新しい Synergy タスクを作成するダイアログが表示されます。このダイアログで、新しいタスクの名前を指定するよう指示されます。

タスクの設定

このコマンドを選択するとダイアログが表示されます。このダイアログで、使用可能な Synergy タスクを選択できます。

タスクの完了

このコマンドを選択すると、現行タスクが Synergy で完了したとマークされます。これは、このタスクで変更されたすべてのオブジェクトが自動的にチェックインされることを意味します。

タスク プロパティ

このコマンドを選択すると、現行の Synergy タスクのプロパティが、ダイアログに表示されます。実際に表示されるダイアログは、使用する Synergy のバージョンによって異なります。

［カレント タスク］ボックス

ツールバーの［カレント タスク］ボックスにより、現在選択しているタスクを表示できます。最近使った Synergy タスクをプルダウン メニューに表示して、そこから新しいタスクを選択することもできます。

Synergy オブジェクトのコマンド

複数のコマンドを選択でき、可能な場合、適用したコマンドは選択したファイルに適用されます。

たとえば、［チェックイン］コマンドにより、選択したすべての要素に対応する Synergy オブジェクトがチェック インされます。対応するオブジェクトがチェック アウトされていない場合は、チェック インされません。

注記

ダイアグラムで実行されたオブジェクト コマンドは、そのダイアグラムを含むファイルに適用されます。

オブジェクトの作成

このコマンドを選択すると、現在選択している UML オブジェクトを含むリソースが、現行の UML プロジェクトを含む Synergy プロジェクトに挿入されます。

プロジェクト ディレクトリ内のオブジェクト、または、プロジェクト ディレクトリ内の管理対象サブディレクトリのみ Synergy に追加できます。

プロパティを表示

このコマンドにより、現在選択している UML オブジェクトを含むリソースの構成管理プロパティを示すダイアログを表示できます。ダイアログの詳細は、使用する Synergy のバージョンによって異なります。

履歴

このコマンドにより、現在選択している UML オブジェクトを含むリソースの構成管理履歴を示すグラフを表示できます。グラフの詳細は、使用する Synergy のバージョンによって異なります。

オブジェクトのチェックアウト

このコマンドを選択すると、現在選択している UML オブジェクトを含むリソースが、Synergy サーバからチェックアウトされます。これは、チェックアウトされたリソースが編集可能になり、他のユーザーが編集できないようロックされること（Synergy ステータスに依存）を意味します。

オブジェクトのチェックイン

このコマンドを選択すると、現在選択している UML オブジェクトを含むリソースが、Synergy サーバに保存されます。これは、また、チェックアウトを取り消されたリソースが、再度チェックアウトされるまで編集できないことを意味します。

オブジェクトのチェックアウトの取消し

このコマンドを選択すると、現在選択している UML オブジェクトを含むリソースが、Synergy サーバでの直前バージョンに戻されます。これは、チェックインされたリソースが、再度チェックアウトされるまで編集できないことを意味します。

注記

このコマンドにより、チェックアウトしたオブジェクトのバージョンが削除されます。チェックアウトしたバージョンがオブジェクトの初期バージョンの場合、オブジェクトが削除されます。

Synergy のバージョン処理コマンド

ステータスの更新

このコマンドにより、アクティブ プロジェクトの各オブジェクトの Synergy ステータスを更新できます。Synergy ユーザー インターフェイスなど、Tau の外部でステータスが変更された場合、この操作が必要になります。

Synergy を使用する **UML** プロジェクトのマージ

並行開発を行っている場合、複数の開発者によって変更されたモデルのマージが必要なことがあります。Synergy を使用している場合、通常は以下の手法をとります。

特定リリースをベースにプロジェクト（この場合は UML プロジェクト）を開発します。

定期的に（ビルド成功後など）、プロジェクトで使用する全ファイルを含むベースライン バージョンを作成します。

それぞれの開発者には、ベースライン バージョンをベースとした Synergy の独自の開発プロジェクト バージョンと、最新の変更があります。

統合テストには、特別なプロジェクト バージョンが使用されます。これもプロジェクトのベースライン バージョンをベースにしていますが、最後のベースライン以降にチェックインされた全ファイルの新バージョンを含むように設定されています。

各開発者が自分の変更をチェックインする*前*に、必ずマージを行って統合プロジェクト バージョンからすべての変更を取得することを推奨します。

Tau でこのマージを行うには、ベースライン プロジェクト バージョンと統合プロジェクト バージョンへのアクセス権が必要です。最も簡単にこれを行うには、各開発者がプロジェクト バージョンに対応するローカル ワーク エリアを持つことです。また、全開発者が統合プロジェクトとベースライン プロジェクトが格納されているディレクトリを共有するというアプローチもあります。

自分のプロジェクト バージョン、ベースライン バージョン、および統合バージョンへのアクセス権があれば、以下の方法でマージを行うことができます。

- ［Tools］->［Merge］コマンドを選択します。
- マージ ダイアログで、プロジェクトのベースライン バージョンを［Ancestor］、プロジェクトの統合バージョンを［Version 2］として指定します。プロジェクトのマージを行うには、必ず .ttp ファイルを選択します。
- 通常のマージを行います。マージを実行してすべての競合が解決したら、タスクを完了して変更後のファイルをチェックインできます。

マージによって変更された全ファイルの履歴リンクは、実行されたマージを反映して自動更新されます。

参照

第 2 章「モデルの操作」の 114 ページ、「バージョンのマージ」

構成管理

複数の構成管理ツール

ジェネリック ソース コード コントロール インテグレーション

IBM Rational ClearCase とのインテグレーション

このセクションでは、Rational ClearCase と Tau のインテグレーション方法について説明します。詳細については、Rational ClearCase のユーザー ガイドを参照してください。

このインテグレーションでは、Microsoft Source Control Interface を使用します。ClearCase で使用できるコマンドについては、ソース コントロール コマンドを参照してください。

IBM Rational ClearCase インテグレーションのインストール

Windows

使用されるソース コントロール システムはシステム レジストリで指定されます。ClearCase をインストールすると、このソース コントロール システム レジストリ キーが自動的に設定されます。ただし、複数の構成管理ツールをローカルでインストールする場合は、この値を手動で編集する必要があります。

1. Tau を起動します。
2. ［ツール］メニューから［オプション］をクリックします。
3. ［一般］タブで、［ソース コントロール プロバイダ］から［ジェネリック ソース コントロール（SCC)］を選択します。
4. ［OK］をクリックします。

次回の Tau の起動からは、［プロジェクト］メニューから新しいメニュー［ソース コントロール］が使用できます。新しいツールバーも追加されます。

UNIX

レジスタ キーと環境変数を設定する必要があります。その際、正規のユーザー ID でログオンする必要があります。使用可能な複数の UNIX バージョンをネットワーク上で実行している場合は、UNIX のバージョンごとに設定が必要になります。

1. ClearCase PATH 環境変数が正しく設定されていることを確認します。
2. レジスタ キーを設定するには、次のスクリプトを実行します。

   ```
   <installationsdir>/bin/setreg_ClearCase
   ```

   ClearCase に最初にアクセスしたときだけ、このスクリプトを実行する必要があります。
3. 環境変数を設定するには、次のスクリプトを実行します。

   ```
   source <installationsdir>bin/setenv_ClearCase
   ```

   ログイン ファイルをこのパスで更新しない限り、ログインするたびにこのスクリプトの再実行が必要です。
4. Tau を起動します。
5. ［ツール］メニューから［オプション］をクリックします。
6. ［一般］タブで、［ソース コントロール プロバイダ］から［ジェネリック ソース コントロール（SCC)］を選択します。このオプションはデフォルトで設定されています。
7. ［OK］をクリックします。

次回の Tau の起動からは、［プロジェクト］メニューから新しいメニュー［ソース コントロール］が使用できます。新しいツールバーも追加されます。

注記

ソース コントロール コマンドを使用するためには、その前にプロジェクト内のファイルを ClearCase ビューに配置しなければなりません。詳細については、IBM Rational ClearCase のドキュメントを参照してください。

参照

構成管理

複数の構成管理ツール

IBM Rational ClearCase のユーザーガイド

複数の構成管理ツール

Windows

複数の構成管理（CM）ツールをインストールしている場合、使用する CM ツールを選択しなければなりません。これは次のレジストリ キーの値によって決定します。

```
[HKEY_LOCAL_MACHINE¥SOFTWARE¥SourceCodeControlProvider]
```

プロバイダを変更する場合は、このレジストリ キーの値の変更が必要です。

インストールされているプロバイダが次のレジストリ キーの値として表示されます。

```
[HKEY_LOCAL_MACHINE¥SOFTWARE¥SourceCodeControlProvider¥Installed
SCCProviders]
```

例 **789:** レジストリ キーの設定

`Microsoft SourceSafe`：

```
[HKEY_LOCAL_MACHINE¥SOFTWARE¥SourceCodeControlProvider]
    "ProviderRegKey"="Software¥Microsoft¥SourceSafe¥ccm"
```

`IBM Rational ClearCase`：

```
[HKEY_LOCAL_MACHINE¥SOFTWARE¥SourceCodeControlProvider]
    "ProviderRegKey"="Software¥Atria¥ClearCase"
```

UNIX

複数の構成管理（CM）ツールをインストールしている場合、使用する CM ツールを選択しなければなりません。

レジスタ キーと環境変数を設定する必要があります。その際、正規のユーザー ID でログオンする必要があります。

1. レジスタ キーを変更するには、使用する構成管理ツールに合ったスクリプトを実行します。

   CM Synergy：`<installationsdir>/bin/setreg_CMSynergy`

   ClearCase：`<installationsdir>/bin/setreg_ClearCase`

   CM ツールに最初にアクセスするときだけ、このスクリプトを実行する必要があります。
2. 環境変数を変更するには、使用する構成管理ツールに合ったスクリプトを実行します。

   ClearCase:
   `source <installationsdir>bin/setenv_ClearCase`

   上記のパスの 1 つで先にログイン ファイルを更新している場合は、そのファイルを編集するだけです。

ソース コントロール コマンド

CM ツールの基本的なコマンドと機能は、［プロジェクト］メニューの個々の［ソース コントロール］メニューに用意されています。これらのコマンドはソース コントロール ツールバーからも利用できます。

多くのコマンドについて、［ファイル］ダイアログが開かれ、そのダイアログからコマンドを適用する対象のファイルを選択できます。コマンドとコマンドを実行する前に選択したファイルまたはフォルダによって、ダイアログに表示されるファイルは異なります。たとえば、フォルダに対してチェックアウト コマンドを選択した場合、チェックインされているファイルだけがダイアログに表示されます。

注記

以下のコマンドは Microsoft SCC Interface ドキュメントに定義されています。すべてのコマンドが構成管理ツールで利用できるとは限りません。また、コマンドの機能が変化することもあります。構成ツールによって、その構成管理ツールに特有のコマンドとツールバー ボタンが追加されることもあります。それらのコマンドとツールバー ボタンはここには表示しません。

最新バージョンの取得

このコマンドはソース コントロール サーバからファイルの最新コピーを検索し、それをユーザーのコンピュータにコピーします。このコピーはまだ読み取り専用です。複数のチーム メンバーと一緒に 1 つのプロジェクトについて作業する場合は、ローカル コピーを頻繁に更新して他のメンバーが行った変更を取り込んでください。

1. ［ファイル ビュー］内のファイルまたはフォルダをクリックします。
2. ［プロジェクト］メニューから［ソース コントロール］をクリックし、次に［最新バージョンの取得］をクリックします。［ファイルの選択］ダイアログが開きます。

3. 更新するファイルを選択し、[OK] をクリックします。

チェックアウト

このコマンドはソース コントロール サーバからファイルまたはフォルダを検索し、ユーザーのためにそのファイルまたはフォルダを予約します。このファイルはユーザーのコンピュータにコピーされ、そのステータスは読み取り専用から読み取り／書き込み可能に変わります。複数のチェック アウトが許されていない場合、ファイルはソース コントロール サーバ上でロックされます。

コメント フィールドでの改行には Ctrl + Enter キーを使用します。

1. [ファイル ビュー] 内のファイルまたはフォルダをクリックします。
2. [プロジェクト] メニューから [ソース コントロール] をクリックし、次に [チェックアウト] をクリックします。[ファイルの選択] ダイアログが開きます。
3. チェックアウトするファイルを選択し、[OK] をクリックします。

注記

チェックアウト前にファイルまたはフォルダが最新バージョンに自動更新されるのは、[ファイルの自動更新] オプションを有効にした場合だけです。このオプションが無効になっている場合は、チェックアウト前にファイルとフォルダを手動で更新する必要があります。一部の CM システムでは、ブランチバージョンの作成などを通じて、最新版以外のバージョンのチェックアウトも可能です。

チェックイン

このコマンドは、ユーザーのローカル バージョンをアイテムの最新バージョンとしてソース コントロール サーバへコピーします。このアイテムのステータスは読み取り／書き込み可能から読み取り専用に変わります。他のメンバーがそのファイルをチェックアウトできるようになります。そのアイテムにまったく変更を加えなかった場合、そのアイテムをチェックインするのではなく、チェックアウトを元に戻すべきです。

コメント フィールドでの改行には Ctrl + Enter キーを使用します。

1. ファイルは必ず保存してください。
2. [ファイル ビュー] 内のファイルまたはフォルダをクリックします。
3. [プロジェクト] メニューから [ソース コントロール] をクリックし、次に [チェックイン] をクリックします。[ファイルの選択] ダイアログが開きます。
4. チェックインするファイルを選択し、[OK] をクリックします。

チェックインされて編集できない要素には、要素シンボルと要素名の間にグレー バーが表示されます。この要素が属するファイルが書き込み可能な場合、このバーは付かず、インターナルフラグのみ表示されます。

チェックアウトの取り消し

このコマンドはアイテムをソース コントロール サーバへ戻します。変更は何も保存されません。チェックアウトしたファイルに変更を加えなかった場合、このコマンドを使用します。

1. ［ファイル ビュー］内のファイルまたはフォルダをクリックします。
2. ［プロジェクト］メニューから［ソース コントロール］をクリックし、次に［チェックアウトの取り消し］をクリックします。［ファイルの選択］ダイアログが開きます。
3. 元に戻すファイルを選択し、［OK］をクリックします。

ソース コントロールに追加

このコマンドはアイテムをソース コントロール サーバに追加します。

1. ［ファイル ビュー］内のファイルまたはフォルダをクリックします。
2. ［プロジェクト］メニューから［ソース コントロール］をクリックし、次に［ソース コントロールに追加］をクリックします。［ファイルの選択］ダイアログが開きます。
3. 追加するファイルを選択し、［OK］をクリックします。

注記

プロジェクト ファイルの追加を選択した場合、プロジェクト内の全ファイルが［ファイルの選択］ダイアログに表示されます。

注記

接続しているソースコントロールプロバイダがファイルの追加に対して制限を強制することで［ソースコントロールに追加］操作が失敗することがあります。一般的な制限として考えられるのは、ファイルの位置、ファイル名、ユーザー権限などにかかわるものです。これらの詳細については、お使いのソースコントロールプロバイダのドキュメントを参照してください。

ソース コントロールから削除

このコマンドにより、ソース コントロール サーバからファイルとフォルダを削除できます。このコマンドでプロジェクト全体またはソリューション全体は削除できません。詳細については、ソース コントロールのドキュメントを参照してください。

1. ［ファイル ビュー］内のファイルまたはフォルダをクリックします。
2. ［プロジェクト］メニューから［ソース コントロール］をクリックし、次に［ソース コントロールから削除］をクリックします。［ファイルの選択］ダイアログが開きます。
3. 削除するファイルを選択し、［OK］をクリックします。

履歴の表示

構成管理ツールは、ソース コントロール サーバへ追加された全バージョンのアイテムの記録を保持します。このコマンドを使用すると、ファイルの履歴を一度に表示できます。

1. ［ファイル ビュー］内のファイルまたはフォルダをクリックします。
2. ［プロジェクト］メニューから［ソース コントロール］をクリックし、次に［履歴の表示］をクリックします。

相違点の表示

このコマンドは、アイテムのローカル コピーとソース コントロール サーバ上にあるその最新バージョンとの相違点を表示します。このコマンドは一度に 1 ファイルのみに適用できます。

1. ［ファイル ビュー］内のファイルまたはフォルダをクリックします。
2. ［プロジェクト］メニューから［ソース コントロール］をクリックし、次に［相違点の表示］をクリックします。

ソース コントロールのプロパティ

このコマンドは選択されたアイテムのプロパティを表示します。開かれるダイアログはツールによって異なります。

1. ［ファイル ビュー］内のファイルまたはフォルダをクリックします。
2. ［プロジェクト］メニューから［ソース コントロール］をクリックし、次に［ソース コントロールのプロパティ］をクリックします。構成管理ツールにアイテムのプロパティが表示されます。

ステータスの更新

このコマンドは、選択したアイテムのステータスを構成管理ツールから更新します。構成管理ツール内でファイルを直接操作した場合にこのコマンドは有用です。プロジェクト ファイル（*.ttp）を更新すると、そのプロジェクトの全ファイルのステータスも更新されます。

1. ［ファイル ビュー］内のファイルまたはフォルダをクリックします。
2. ［プロジェクト］メニューから［ソース コントロール］をクリックし、次に［ステータスの更新］をクリックします。

ソース コントロール

このコマンドは、Tau に接続されている構成管理（CM）ツールを起動します。接続される CM ツールは次のレジストリ キーの設定によって決まります。

```
[HKEY_LOCAL_MACHINE¥SOFTWARE¥SourceCodeControlProvider]
```

1. ［ファイル ビュー］内のファイルまたはフォルダをクリックします。
2. ［プロジェクト］メニューの［ソース コントロール］にカーソルを合わせ、次に［ソース コントロール］クリックします。

ソース コントロールからインポート

このコマンドを使用すると、ソース コントロール サーバに保存されていて、しかも作業中のプロジェクトに属さないファイルを見つけることができます。見つけたファイルはローカル コンピュータに追加され、ユーザーのプロジェクトに追加されます。

1. ［ファイル ビュー］内のファイルまたはフォルダをクリックします。
2. ［プロジェクト］メニューの［ソース コントロール］にカーソルを合わせ、次に［ソース コントロールからインポート］クリックします。

注記

［ソース コントロールからインポート］コマンドは ClearCase では使用されません。

ソースコード コントロール ツールからの比較とマージ

このセクションでは、ソースコード コントロール ツールのバージョン ブラウザで選択した u2 ファイルのバージョンに対して比較とマージ操作を行うために、Tau を起動する方法について説明します。

ここでは、Synergy と IBM Rational ClearCase に対する詳細な設定情報を提供します。

注記

u2 ファイルでは、比較とマージ操作のみをサポートします。

Synergy の設定

Synergy から Tau を起動できるようにするには、次の手順を行います。

- Tau u2 ファイルのファイル型定義をインストールする（Tau ファイル タイプ定義を参照）。
- Synergy クライアントの比較とマージ操作を設定する。
- Tau u2 ファイルのソース コントロールに Synergy をすでに使用している場合は、引用符で囲んだ定義を使用せずに作成していれば、既存の要素（オブジェクト）を新しいオブジェクト タイプに変えることも可能。

比較とマージ操作の設定

比較 / マージ ツールの設定は、デフォルトでクライアント インストール フォルダの etc ディレクトリにある ccm.properties ファイル（Windows）、またはホーム ディレクトリにある ccm.user.properties ファイル（UNIX）によって行います。

- Windows

```
windows.tool.compare.tau_model  =
"c:¥¥Program Files¥¥IBM¥¥Rational¥¥TAU¥¥4.3¥¥bin¥¥U2FileUtility.exe
" xcompare "%file2" "%file1"
windows.tool.merge.tau_model  =
"c:¥¥Program Files¥¥IBM¥¥Rational¥¥TAU¥¥4.3¥¥bin¥¥U2FileUtility.exe
" xmerge -base "%ancestor" -out "%outfile" "%file2" "%file1"
```

- UNIX

```
unix.tool.compare.tau_model  = .../bin/U2FileUtility xcompare
%file2 %file1
unix.tool.merge.tau_model  = .../bin/U2FileUtility xmerge -base
%ancestor -out %outfile %file2 %file1
```

パスに bin ディレクトリが含まれている場合はパスを除外できます。

注記

Windoes 版の Synergy では、区切り文字として円記号が 2 つ必要です。

既存の要素を新しいオブジェクト タイプに変更

Tau u2 ファイルをすでに Synergy に格納している場合は、そのファイルのオブジェクト タイプを変更する必要があります。これには change_type コマンドを使用します。詳細については Synergy のドキュメントを参照してください。

次のコマンドを使用して、データベースにすでに格納されている u2 ファイルのオブジェクト タイプを変更します。

```
ccm change_type file_spec -type tau_model
```

IBM Rational ClearCase の設定

Tau を ClearCase から起動できるようにするには、次の手順を行います。

- ClearCase タイプ マネージャを構成する。
- 新しい要素タイプを作成する。
- オプションとして、新しい要素タイプの ClearCase マジック ファイルを設定することも可能。
- また、Tau u2 ファイルのソース コントロールに ClearCase をすでに使用している場合は、既存の要素を新しいオブジェクト タイプに変えることも可能。

ClearCase タイプ マネージャの構成

ClearCase では、タイプ マネージャの設定は、ClearCase クライアント インストール フォルダのマップ ファイルによって行います。詳細については、「cleartool man type_manager」を参照してください。

Windows でインストールを行う場合は、次の行をマップ ファイル（ClearCase ホーム ディレクトリの directory lib¥mgrs¥map に格納）の末尾に追加します。

```
u2_manager        construct_version          ..¥..¥bin¥zmgr.exe
u2_manager        create_branch              ..¥..¥bin¥zmgr.exe
u2_manager        create_element             ..¥..¥bin¥zmgr.exe
u2_manager        create_version             ..¥..¥bin¥zmgr.exe
```

```
u2_manager        delete_branches_versions..¥..¥bin¥zmgr.exe
u2_manager        xcompare                  C:¥Program
Files¥IBM¥Rational¥TAU¥4.3¥bin¥U2FileUtility.exe
u2_manager        xmerge                    C:¥Program
Files¥IBM¥Rational¥TAU¥4.3¥bin¥U2FileUtility.exe
u2_manager        get_cont_info             ..¥..¥bin¥zmgr.exe
```

UNIX でのインストールは次の手順で行います。

- ClearCase ホームディレクトリの `lib¥mgrs` ディレクトリに移動する。
- `z_whole_copy` ディレクトリを `u2_manager` にコピーする。その際、たとえば tar を利用してディレクトリをコピーして、リンクを保持します。
- `u2_manager` に移動する。
- `compare`、`xcompare`、`merge`、`xmerge` という名前のリンクを削除する。
- `Tau` の `bin` ディレクトリ内の `xcompare` と `xmerge` への新しいシンボリック リンクを作成します。

新しい要素タイプの作成

ClearCase は mkeltype を使用して新しい要素タイプを作成します。詳細については、「cleartool man mkeltype」を参照してください。

Tau u2 ファイルの新しい要素タイプを作成するには、次の手順を行います。

- 新しい要素タイプを作成する VOB に移動する。
- 次のコマンドで新しい要素タイプを作成する。

  ```
  cleartool mkeltype -supertype file -manager u2_manager
  tau_model
  ```

`tau_model` は新しいタイプの名前です。

注記：-global フラグを使用すると、1 回の操作で複数の VOB を更新できます。

新しい要素タイプの **ClearCase** マジック ファイルを設定

mkelem コマンドで新しい要素を作成するとき、-eltype オプションを使用して要素のタイプを指定できます。マジック ファイルを使用すると、新しい要素が正しい要素タイプを自動的に取得します（-eltype オプションを使用することなく）。詳細については、「cleartool man cc.magic」を参照してください。

次の行を、default.magic ファイル（<clearcase home dir>¥config¥magic¥default.magic）に追加します。

```
tau_model text_file : -name “*.[uU]2”;
```

注記：この行は以下の行の上に挿入する必要があります。

```
# catch-all, if nothing else matches
compressed_file : -name "*" ;
```

既存の要素を新しい要素タイプに変更

Tau u2 ファイルをすでに ClearCase に格納している場合は、そのファイルの要素タイプを変更する必要があります。これには chtype コマンドを使用します。詳細については「cleartool man chtype」を参照してください。

次のコマンドを使用して、VOB にすでに格納されている u2 ファイルの要素タイプを変更します。

```
cleartool chtype -force tau_model <file>
```

ディレクトリ サーバー

Tau プロジェクトを IBM Rational Directory Server（RDS）ファシリティとして登録できます。

Tau プロジェクトを RDS ファシリティとして登録することで、以下の機能が使えるようになります。

- 外部関係のサポート。要素と RDS 統合した他ツールの間に関係が作成されます。Tau を使って RDS と統合した他ツールから作成された関係を操作できます。
- プロジェクトを IBM Rational Team WebTop で表示できるようになります。

Tau プロジェクトの公開

プロジェクトの公開は［リンク］メニューから［オプション］を選択して行います。RDS との接続を設定するオプションは、［外部関係］プロパティページにあります。

注記

ワークスペースが複数のプロジェクトを含んでいる場合、現在選択している項目によってどのプロジェクトが公開されるかが決まります。

プロパティ ページには RDS サーバーのホストとポートを入力するためのフィールドがあります。プロジェクト公開前にこれらの値を用意し、確実に設定してください。

RDS 情報を設定したら、［プロジェクトを公開する ...］ボタンを押します。新しいダイアログが表示されます。

このダイアログに入力する情報は、RDS が Tau ファシリティを見つけるために使われます。［公開されるプロジェクト名］設定はファシリティの名前です。Tau サーバーのホスト名とポートは他のツールが Tau をアクセスするために使用します。

デフォルトで、プロジェクト名がファシリティ名として使われます。Tau サーバー ポートは、デフォルトで現在実行中の Tau インスタンスのポートに設定されます。

［サーバーモード コマンドラインスイッチ］フィールドは、プロジェクトの公開後に、Tau を正しいポート上でヘッドレス モードで起動するためのコマンドライン スイッチを表示します。

［RDS 接続情報の格納］の［プロジェクト ファイル (.ttp)］をチェックすると、公開されたプロジェクト情報をプロジェクト ファイルに格納できます。この情報は同じプロジェクト ファイルをもつすべてのユーザーから利用できます。他の方法として、各ユーザーが RDS サーバー情報とプロジェクトの公開名を入力するやり方もあります。この場合情報は、プロジェクトの一部ではなく、ローカルの .u2 ファイルに格納されます。

外部関係の同期を取る

プロジェクトが Tau サーバーのファイル システムで変更されると、同期コマンドが使用できるようになり、この同期によって外部関係を持つエンティティへの変更が確実に RDS に知らされます。たとえば、外部関係を撤廃することによって、要素はプロジェクトから削除されます。

同期処理は、［リンク］メニューの［オプション］から［外部関係］プロパティページを選択して行います。［外部関係を同期する］ボタンを押すと、外部関係の同期が取られます。

67

リンクを使った作業

この章では、Tau でリンクを使った作業をどのように行うかを説明します。1 つのリンクは 2 つの要素の間の関係と言うことができます。リンクは任意の要素の間で作成できます。たとえば、モデル要素同士、モデル要素と DOORS 内の要求項目の間、モデル要素と外部のウェブページの間などがその例です。

リンクには以下の 3 つの種類があります :

- ハイパーリンク
- 依存リンク
- 外部関係

リンクの管理 の章では、リンクの作成、削除、ナビゲートについて説明します。

ハイパーリンク

ハイパーリンクは、2 つの要素を標準の Uniform Resource Location (URL) を使用して関係付けます。これは通常のウェブページ上のリンクと同じです。したがって、リンクの宛先は、URL で識別できるものであれば何でもかまいません。宛先の例は、ある特定のダイアグラム、ファイルシステム内のある特定のファイル、ある特定のウェブページなどです。

Tau URL の形式は以下のとおりです：

```
tlog://<project_path>:plugin_ident:string_ident
```

この URL は以下の部分から構成されています：

- `<project_path>` は、オブジェクトを含むプロジェクトの Windows 上のパスです。
- `plugin_ident` は、オブジェクトを所有する Tau アドインの内部的な識別子です。
- `String ident` は、所定のプラグインの所定のプロジェクトの文字列表現です。

例 **790**

以下の例は典型的な UML モデル内でのクラスの URL を示しています：

```
tlog://<C:\MyModels\LinksProj\LinksProj.ttp>:u2:iOepWITH4FFLCghYEIWspKrL
```

LinksProj という名前のプロジェクトが Class1 という名前のクラスを含んでいます。URL の u2 の部分がクラスが UML 要素であることを示しています。クラス自身は GUID、`iOepWITH4FFLCghYEIWspKrL` で表現されています。

可視化

外向きのハイパーリンクは、要素名に付いた青い下線で示されます。この下線は、ワークスペースウィンドウか、ダイアグラム内に表示されます。

内向きのハイパーリンクは、ダイアグラム内の青色の点線の下線と、ワークスペース内の要素名の隣にある小さい青色の三角印で示されます。

ハイパーリンクのナビゲーション

Tau 内部でのハイパーリンクを使ったナビゲーション

ハイパーリンクを使ってダイアグラム内をナビゲートするには、**Ctrl** キーを押しながら青色の点線下線の付いたテキストをクリックします。

外部アプリケーションでのハイパーリンクのナビゲーション

ウェブブラウザのような外部アプリケーションで与えられた URL をナビゲートすると、Tau が起動されて、そのリンクが指している要素を含むモデルがロードされます。技術的な制約のため、この動作を保証しないプラットフォームもあります。

宛先がない場合

現在ロードされているプロジェクトにリンクの宛先がない場合があります。

このリンクが Tau の要素を指している場合は、エラーメッセージが表示されます。

注記

リンクのソースはリンク情報の担い手ですので、通常は、リンクソースがないということは起こり得ません。

Tau モデルへのハイパーリンクの作成

Tau の各要素は固有の URL を割り当てられており、ウェブページのような外部ソースからリンクできます。

要素から URL を取得するには :

- ワークスペースウィンドウまたはダイアグラムで要素を選択します。
- 右クリックして、コンテキストメニューから［URL のコピー］を選択します。
- 対象のアプリケーションに URL を貼り付けます。

依存リンク

依存リンクは、UML 依存を使用して 2 つの要素を関係付けます。このリンクは任意の UML 要素間の関係付けに使用できます。依存リンクの主な利点は、UML で表現できることと、ダイアグラム中で作成、編集、削除、可視化ができることです。

注記

すべての依存関係を依存リンクで表現する、というわけではありません。ある特定のステレオタイプをもつ依存のみが依存リンクで表現できます。これについては、要求の関係セクションで説明しています。リンク コマンドを使用して任意の依存を作成できますが、要求の関係でない場合は、リンク標識はできず、リンクダイアログにも表示されません。

依存リンクは、「通常」と「シンボリック」という 2 つの異なる方法でソースや宛先を参照できます。

「通常」の場合、依存のクライアントとサプライヤは単にソースと宛先を参照します。この参照は、修飾子付きの名前や GUID を使用して設定できます。

「シンボリック」の場合、依存には、ソースと宛先へのシンボル参照（URL）から構成される補足情報の注釈が付きます。この場合、クライアントとサプライヤは、定義済みの要素、ExternalObject を参照します。シンボル参照によって、任意の UML の依存リンクや DOORS の要求のような外部要素への依存リンクを使えるようになります。

可視化

外向きの依存リンクは、ワークスペースウィンドウ内またはシンボル上の、要素名の隣の小さい紫色の三角形で示されます。

内向きの依存リンクは、ワークスペースウィンドウ内またはシンボル上の、要素名の隣の小さい橙色の三角形で示されます。

参照

要求の関係

外部関係

外部関係は、Tau プロジェクト内の 1 つの要素と Tau の外部のオブジェクト（外部オブジェクト）との間の接続形式の 1 つです。この定義は、この外部オブジェクトが同じ Tau プロジェクト内にある場合も含んでいます。

外部関係は、双方向の関係です。

可視化

外向きの外部関係は、ワークスペース ウィンドウまたはシンボル上で、要素名の横に付いた小さい紫色の三角形で表されます。

内向きの外部関係は、ワークスペース ウィンドウまたはシンボル上で、要素名の横に付いた小さい橙色の三角形で表されます。

リンクの管理

リンクは、リンク メニュー、リンク ツールバー 、リンク ダイアログ 、コンテキストメニューのリンク サブメニューから作成、編集、削除できます。リンク コマンド セクションに使用可能なコマンドのリストがあります。

依存リンクは、UML モデル中で直接編集できます。

DOORS で作業する場合、依存リンクは要素を Tau と DOORS 間でドラッグアンドドロップして作成できます。リンクの作成を参照してください。

リンクの作成

リンクを作成する方法は複数あります。ドラッグアンドドロップが最も便利ですが、複数の宛先がある場合や宛先をまだ見つけられていない場合は、手動でリンクを作成する必要があります。

ツールバーを使用してリンクを作成する

リンクツールバーを使用してリンクを作成するには：

- リンク ツールバーの［現在のリンクの種類］リストから正しいリンクのタイプを選択します。.
- ソース要素を選択します。
- リンクツールバーの［リンクの開始］を選択して、**Ctrl+K** キーを押します。

  この操作で、選択した要素はアクティブなリンクソースになり、リンクが作成されるかクリアされるまで記憶されます。

  リンクが ハイパーリンク である場合は、ハイパーリンクの挿入 ダイアログ が開き、外部のハイパーリンクが指定できます。
- 宛先の要素を選択します。
- リンクツールバーで［始点からのリンクの作成］をクリックして、**Ctrl+L** キーを押します。

リンクが、選択したソースの要素から選択した宛先の要素に向けて作成されます。

リンクのタイプが依存リンクの場合、リンクごとにステレオタイプダイアログが表示されて、リンクのタイプが指定できます。ステレオタイプダイアログでの選択は、Tau セッションの実行中維持されます。

注記

［デフォルトで、ワークスペース要素にハイパーリンクを作成する ］オプションが有効になっていると、［ハイパーリンクの挿入］ダイアログは表示されません。

ドラッグアンドドロップでリンクを作成する

ドラッグアンドドロップでリンクを作成するには：

- リンク ツールバーの［現在のリンクの種類］から現在のリンクタイプを選択します。

- ソース要素を選択します。
- 右マウスボタンを押して、要素を宛先要素にドラッグします。
- マウスボタンを離します。コンテキストメニューから［リンク］を選択します。

注記

一部のビューでは、左マウスクリックを使ったドラッグアンドドロップが特別のセマンティクスを持たないので、リンク作成に使用できます。この場合、コンテキストメニューは不要です。出力ウィンドウのタブはそのようなビューの 1 つの例です。

ダイアグラム内にリンクを作成する

依存リンクは、依存によって表現されるのでダイアグラム内に描画できます。ダイアグラム内に依存リンクを作成するには：

- リンクソースを選択して適切なラインハンドルを使用し、依存リンクを作成します。
- 必要に応じて、依存にステレオタイプを適用して、その依存を依存リンクにします。依存リンクタイプのリストは、要求の関係セクションにあります。

同一のソースから複数のリンクを作成する

リンクのソース要素をロックして、同一の要素から多数のリンクを作成できます。宛先ごとにソースを選択し直す必要はありません。

ソース要素をロックするには：

- 正しいリンク タイプをリンク ツールバーの［現在のリンクの種類］リストから選択します。
- ソース要素を選択します。
- リンク ツールバーから［複数のリンクの開始］を選択します。

この操作で、選択したオブジェクトはアクティブなリンクソースになり、［複数のリンクの開始を解除］をクリックしてクリアされるまで記憶されます。

ソース要素から 1 つまたは複数の宛先要素に向けてのリンクを作成するには：

- 宛先要素をクリックして、［始点からのリンクの作成］を選択するか、**Ctrl+L** キーを押します。

この操作を必要な回数だけ繰り返せば、同一のソースから異なる宛先向けのリンクを作成できます。ソース要素をクリアするには、［複数のリンクの開始を解除］をクリックします。

自動リンク作成

自動リンク作成と呼ばれる特殊なモードがあります。このモードを使うと、事前に選択した要素からすべての修正した要素に向けてリンクを自動的に作成できます。

このモードが有効になっていると、リンクは要素が修正されると同時に作成されます。リンク元は事前に選択されたソース要素であり、宛先は修正された要素になります。

自動リンク作成モードを有効にするには：

- ［修正されたオブジェクトとアクティブなリンク端とのリンクを自動作成］オプションを有効にします。
- リンクソースを選択して［複数のリンクの開始］をクリックします。

自動リンク作成モードを無効にするには：

- ［複数のリンクの開始を解除］をクリックします。

リンクの削除

ある要素をソースとするリンクを削除するには：

- リンクのソースまたは宛先である要素を選択します。
- リンク ツールバー の［リンクの編集］をクリックします。
- ［アクティブなリンクの種類］ドロップダウンリストから正しいリンクタイプを選択します。
- 選択した要素にしたがって［方向］を［外部へ］または［外部から］に設定します。
- リスト内の要素を選択して［削除］をクリックします。

リンクは［リンク］コンテキストメニューやメインメニューの［リンク］メニューからでも削除できます。.

リンクが依存リンクである場合は、モデルから直接削除できます。これには、モデルビューでそのリンクを表している依存を選択して削除キーを押します。依存がダイアグラム中で可視状態にあれば、そのラインを選択して［モデルから削除］コマンドも使用できます。

リンクのナビゲート

ワークスペースウィンドウまたはダイアグラムでリンクをナビゲートするには：

- リンクインジケータシンボルを右クリックします。
- コンテキストメニューから正しいリンクタイプを選択します。
- リンクの宛先要素か URL を選択します。

リンクのナビゲーションの結果は、リンクのタイプとインストール済みのインテグレーションによって異なります。デフォルトでは、ワークスペースウィンドウで要素を選択して、可能であれば、ダイアグラム中で強調表示されます。インテグレーションによっては、宛先要素の表示のために外部のアプリケーションが起動されます。

参照

ハイパーリンクのナビゲーション

リンク コマンド

このセクションでは、リンクについて利用可能なコマンドをすべて説明します。ただし、アドインとインテグレーションによって追加されるコマンドは説明していませんので、注意してください。コマンドは、リンク メニュー、リンク ツールバー、リンク ダイアログ からアクセスできます。

リンクの開始 (CTRL+K)

リンク作成の開始点（リンク ソース）を指定します。リンクの作成を参照してください。

複数のリンクの開始

複数のリンクを作成するための開始点（リンク ソース）を指定します。リンクの作成を参照してください。

始点からのリンクの作成 (CTRL+L)

リンクの宛先を指定してリンク作成を完了します。リンクの作成を参照してください。

URL のコピー

選択要素のハイパーリンク URL をテキストとしてクリップボードにコピーします。このテキストを貼り付けて、要素へのナビゲーション用 URL をサポートする任意のアプリケーションで使用できます。

外部へのリンクを表示

このコマンドは、選択要素から出る外向きのリンクを、出力ウィンドウのリンクタブにリスト表示します。

外部からのリンクを表示

このコマンドは、選択要素に入る内向きのリンクを、出力ウィンドウのリンクタブにリスト表示します。

リンクの編集

選択要素の内向き、外向きの両方のリンクを編集できます。リンク ダイアログを参照してください。

リンク オプション

リンクオプションの表示と編集。リンク オプション、ハイパーリンク オプション、外部関係オプションを参照してください。

リンク メニュー

リンクメニューは Tau のメインメニューの一部であり、ほとんどのリンクコマンドはここにあります。

リンク ツールバー

リンクツールバーにはほとんどのリンクコマンドがあります。リンクを操作する最も便利な方法です。

リンクツールバーの最も重要な機能は、[現在のリンクの種類] コントロールを使用してリンクの種類を指定することです。

現在のリンクの種類

リンク作成時に必要となるリンクの種類を指定します。リンクソースの指定の前に正しい種類を選択しておく必要があります。

使用可能なリンクの種類は以下のとおりです:

- ハイパーリンク
- 依存リンク
- 外部関係

注記

使用できるリンクの種類は、プラットフォームとインストールされているインテグレーションに依存します。

リンク ダイアログ

[リンク] ダイアログを使用して、選択している要素についての内向きと外向きのリンクの表示と削除ができます。このダイアログの主目的は、リンクを削除することです。

[リンク] ダイアログを表示するには:

- 要素を選択します。
- リンクツールバーで [リンクの編集] をクリックします。

[リンク] ダイアログは、選択したオブジェクトに接続しているすべてのオブジェクトを表示します。Tau にロードされていないオブジェクトについては、名前ではなく URL で参照されます。

[アクティブなリンクの種類] ドロップダウンリストは、どのタイプのリンクを表示するかを制御します。一時点で 1 つのタイプのみをアクティブにできます。

[方向] ラジオボタンは、外部へのリンクと外部からのリンクのどちらを表示するかをを制御します。

[削除] ボタンを押すと、選択したリンクは削除されます。

ハイパーリンクの挿入 ダイアログ

このダイアログでは、リンクの宛先を検索できます。以下の項目が表示されます。

- **リンク先：**リンクが外部（ウェブページやファイル）か、現在のワークスペース内かを選択できるテキストフィールド。
- **表示するテキスト：**リンク出力タブまたは［リンク］ダイアログの［名前］カラムに表示されます。
- **ハイパー リンク ターゲットの URL：**ウェブページの位置を直接指定できるテキストフィールド。
- 最近使用したファイル、ウェブページ、リンクを、**［最近使ったファイル］**、［**参照したページ**］、［**挿入したリンク**］ラジオボタンで表示します。

リンク オプション

Tau はリンク作成機能をカスタマイズする方法を提供します。［リンク］オプションを変更するには、［ツール］メニューから［オプション］を選択するか、［リンクオプションの表示］ツールバーをクリックします。

修正されたオブジェクトからアクティブなリンク端へのリンクにする

このオプションをチェックしていない場合は、下記のリンクの自動作成を使用したときに、リンクがアクティブなリンク端から修正されたモデルに向けて作成されます。

チェックしている場合は、リンクの自動作成を使用したときに、リンクが修正されたモデルからアクティブなリンク端に向けて作成されます。

修正されたオブジェクトとアクティブなリンク端とのリンクを自動作成

このオプションをチェックしている場合は、アクティブなリンク端として選択した要素と他の要素の間に、後者の要素を修正するたびにリンクが自動的に作成されます。

リンク インジケータを表示

このオプションをチェックしていると、Tau はリンクを示すマーカを表示します。

ドラッグ **&** ドロップでリンクを作成するときに、要求をターゲットとして使用

このオプションは、ドラッグアンドドロップでリンクを作成する場合のリンクの方向に影響します。

このオプションをチェックしている（デフォルト）場合、要求をモデル要素へドラッグアンドドロップすると、リンクはモデル要素から要求に向けて作成されます。通常はこの接続方向で使用することをお勧めします。

ハイパーリンク オプション

Tau は、リンク作成機能をカスタマイズする方法を提供します。カスタマイズを行うには、［ツール］メニューから［オプション］を選択するか、［リンクオプションの表示］ツールバーボタンをクリックします。

デフォルトで、ワークスペース要素にハイパーリンクを作成する

このオプションは、ハイパーリンクの挿入 ダイアログ 起動時に、［リンク先］テキストフィールドの開始設定を制御します。

このオプションをチェックすると、ワークスペース内のハイパーリンクのターゲットを選択できるようになります。チェックしないと、既存のファイルやウェブページといった他の宛先タイプを指定するように要求されます。

外部関係オプション

外部関係タイプのリンクに関連するオプションは、［ツール］または［リンク］メニューから［オプション］を選択するか、または［リンクオプションの表示］ツールバーボタンをクリックして設定できます。

外部関係サポートを有効にする

このオプションをチェックすると外部関係サポートが有効になります。

RDS サーバー情報

外部関係タイプのリンクを使用するには、IBM Rational Directory Server（RDS）との接続を設定する必要があります。

［公開されるプロジェクト名］フィールドに公開する Tau プロジェクトの名前を入力します。この名前はプロジェクトファイルに格納されるか、Tau サーバー管理者によって配布されます。.

［ホスト］と［ポート］フィールドには、RDS サーバーの位置が保持されています。

管理

RDS サーバーの管理は、通常は、Tau サーバーの管理者によって行われます。

参照

2103 ページの「ディレクトリ サーバー」

68

Visual Studio .NET インテグレーション

Tau Visual Studio .NET インテグレーションは、Microsoft Visual Studio .NET 2008 IDE とのインテグレーションを可能にします。インテグレーションは、Tau と Visual Studio .NET の両方にいくつかのコマンドを追加し、UML モデルと生成されたコード間のシームレスな双方向ナビゲーションなどを可能にします。

既存の UML プロジェクトから Visual Studio .NET プロジェクトを作成し、生成されたソース ファイルを入れることができます。

Visual Studio .NET インテグレーションは、C++ コードと C# コードの両方に対応しています。ほとんどの機能は、両方の言語に共通しています。どちらかの言語に固有の機能については、そのことをはっきり説明しています。

インテグレーションのインストール

Visual Studio .NET インテグレーションは、次の 2 つのアドインで構成されています。

- `MSVS8Integration` という名の Tau アドイン
- `TAUG2IntegrationAddin` という名の Visual Studio .NET アドイン

インテグレーションが適切に機能するには両アドインをアクティブにする必要があります。

Visual Studio アドインのアクティブ化

Visual Studio .NET で `TAUG2IntegrationAddin` アドインをアクティブにするには、まずそれをインストールする必要があります。

1. Visual Studio .NET が正しくインストールされており、実行していないことを確認します。
2. インストールを開始するには、［スタート］メニューから［すべてのプログラム］を選択し、［IBM Rational］>［IBM Rational Tools］>［IBM Rational Tau 4.3］>［Install Microsoft Visual Studio 2008 integration］を選択します。あるいは、通常、以下の場所にあるアドイン用の `Setup.exe` を実行します。

   ```
   C:¥Program Files¥IBM¥Rational¥TAU¥4.3¥¥integrations¥MSVS8
   ```
3. セットアップ プログラムの指示に従います。

次回 Visual Studio .NET を開始するときアドインがアクティブになります。アドインがアクティブになると、Visual Studio に Tau メニューが表示されます。

注記

Visual Studio .NET メニュー コントロールに接続されたコマンドは、アドインとは別の外部環境に格納されます。新しいバージョンをインストールする場合、あるいは破損したコマンド環境を復元する場合は、以下の手順を実行します。
1) ［プログラムの追加と削除］から TAUG2IntegrationAddin アドインをアンインストールする。
2) Visual Studio .NET のコマンドプロンプトから **devenv /setup** を実行する。
3) TAUG2IntegrationAddin アドインを再インストールする。

Tau アドインのアクティブ化

Visual Studio .NET と協調するすべてのプロジェクトに対して `MSVS8Integration` アドインをアクティブにする必要があります。Visual Studio .NET サポートをアクティブにするには、次の手順を行います。

1. ［ツール］メニューから［カスタマイズ］ダイアログを選択します。
2. ［アドイン］タブをクリックして、MSVS8Integration アドインをチェックします。
3. ［閉じる］をクリックします。

アドインがアクティブになると、Tau に Visual Studio .NET メニューが表示されます。

C++ コードまたは C# コード生成用に新しい Tau プロジェクトを作成したときに、`MSVS8Integration` アドインがアクティブになるように設定できます。このためには、［Developer プロジェクト］ウィザードで、［MS Visual Studio インテグレーションを有効にする］を選択します。

Tau と Visual Studio .NET の協調

Tau と Visual Studio .NET の接続

通常は、Tau と Visual Studio .NET の両方の複数のインスタンスがマシンで実行されています。インテグレーション コマンドの多くは、機能するためには Tau の 1 つの特定インスタンスと Visual Studio .NET の 1 つの特定インスタンスが接続されている必要があります。Tau または Visual Studio .NET からそのようなインテグレーション コマンドを最初に実行した際、この接続手順が自動的に実行されます。

相手のツールが実行されていなかった場合は、自動的に起動して接続が行われます。

相手のツールの少なくとも 1 つのインスタンスが実行している場合は、表示されたダイアログから接続したいツールのインスタンスを選択します。このダイアログで、ツールの新しいインスタンスに接続することもできます。

接続に失敗した場合、インテグレーションのインストールの説明どおりにインテグレーションが正しくインストールされているかどうか確認してください。

ワークフロー

Tau Visual Studio .NET インテグレーションにより、C++ コードおよび C# コードのアプリケーション開発において、以下のようなワークフローが可能になります。

- C++ コードまたは C# コード生成用に、Tau で UML モデルを作成する。モデルからソースコードを生成したら、インテグレーションにより、Visual Studio .NET プロジェクトを作成します。このプロジェクトにより、Visual Studio .NET からビルドとデバッグを行うことができます。
- Visual Studio .NET プロジェクトを作成したら、Tau または Visual Studio .NET を使用するか、あるいはこれらの環境を組み合わせて、アプリケーションの作業を行う。モデルに対して行った変更は Tau の C++ または C# コード ジェネレータによってコードに生成され、コードに対して行った変更は Tau のモデル更新（ラウンドトリップ）機能によってモデルに反映されます。
- Tau と Visual Studio .NET のナビゲーション コマンドにより、C++ または C# のソースコードと UML 間で双方向のナビゲートが可能。

また、インテグレーションにより、Visual Studio .NET から生成されたアプリケーションのデバッグを行うために役立つコマンドも提供されます。たとえば、デバッグ中の Tau 内の実行を自動的にトレースできます。

C++ の場合は、Tau UML シーケンス図によるトレースが可能です。Tau の UML デバッガでブレークポイントに到達してアプリケーションが停止した場合は、Visual Studio のデバッガに切り替えることも可能です。デバッグ中は、モデルへのナビゲーションも自動化されます。

Tau で作成する UML モデルは、最初から作成する必要はありません。C++ アプリケーションまたは C# アプリケーションの一部に、再利用するレガシーコードが使われていることがよくあります。そのようなコードを Tau にインポートして、UML モデルからそれを使用できます。C++ または C# のコードジェネレータを使用してコードを再生成するか、あるいはコードをそのままにして UML モデルからそれにアクセスできます。

Visual Studio .NET インテグレーションのナビゲーション機能は、Tau にインポートされたコードと Tau から生成されたコードの両方に機能します。

参照

第 9 章「C/C++ のインポート」

既存の C# コードのインポート

インテグレーション コマンド

Tau コマンド

Visual Studio .NET C++/C# プロジェクトの生成 / 更新

既存の UML モデルから新しい Visual Studio .NET プロジェクトを作成する場合、および UML モデルでの変更を反映して既存の Visual Studio .NET プロジェクトを作成する場合、このコマンドを使用します。

既存の UML モデルから新しい Visual Studio .NET プロジェクトを作成するには、次の手順を行います。

1. モデル（あるいはモデルの一部）から C++ コードまたは C# コードを生成します。
2. ［モデル ビュー］で、コードを生成したモデルの一部であるエンティティを選択します。
3. ［Visual Studio .NET］メニューから［Visual Studio .NET C++/C# プロジェクトの生成 / 更新］を選択します。

C++ と C# では、作成されるプロジェクトの種類が異なります。

- C++ の場合は、コンソール アプリケーションをビルドするためのプロジェクトが作成されます。
- C# の場合は、作成したいプロジェクトの種類を要求されます。開発しようとするアプリケーションに適したプロジェクトの種類を選択してください。また、新しいソリューションにプロジェクトを挿入するか、現在開いているソリューションに追加するか、選択できます。

新しい Visual Studio .NET プロジェクトには、モデル内の生成された C++/C# ファイルがすべて入ります。作成されたプロジェクトにも適切なプロジェクト設定が適用されます。

同じ Tau モデルで、C# と C++ の開発を混同してはなりません。アプリケーションの C# 部分と C++ 部分には、別々の Tau プロジェクトを使用します。

注記

選択した Visual Studio .NET C# プロジェクト ウィザードによって、不要なファイルが作成される場合があります。そのようなファイルは、プロジェクトから削除する必要があります。たとえば、モデルに静的な Main 操作が含まれている場合、ウィザードによって作成された、対応する C# Main メソッドを含むファイルを削除しなければなりません。

次回［Visual Studio .NET C++/C# プロジェクトの作成 / 更新］コマンドを実行すると、新しいソース ファイルの追加などモデル内で行われた変更を反映して、プロジェクトが更新されます。ファイルは、モデルから生成されることがなくなっても、プロジェクトから削除されることはありません。プロジェクトから削除するファイルは、手動で決定する必要があります。

Visual Studio .NET C++/C# プロジェクトを開く

このコマンドは、Tau モデルが接続されている Visual Studio プロジェクトを検索します。Visual Studio ソリューションに複数のプロジェクトが含まれている場合、あるいは Visual Studio の複数インスタンスが実行されている場合、このコマンドが役立ちます。

要素の検索

モデルの C#/C++ コードを生成した後は、選択したモデル要素に対応する Visual Studio 内のソース コードにナビゲートできます。

C# では、このコマンドは［Visual Studio プロジェクトを検索］です。

C++ 要素では、ヘッダー ファイルと実装ファイルの 2 つのファイルになることがあります。そのため、［ヘッダー ファイルを検索］と［実装ファイルを検索］の 2 つのコマンドが用意されています。

ソース ファイル内で要素が検出された場合、Visual Studio でそのファイルが開き、ファイル内の要素の場所にカーソルが置かれます。

生成された C#/C++ ソースへナビゲートするもう 1 つの方法は、［モデル ビュー］のポップアップ メニューで［ソースに移動］コマンドを使用することです。選択したモデルが生成されたコード内に少なくとも 1 つの表現を持っていれば、このコマンドを使用できます。ソース ファイルは、通常 Tau のテキストエディタで開きますが、Visual Studio インテグレーションを使用しているときは、Visual Studio でファイルが開きます。

制御をターゲットデバッガに移す

このコマンドは C++ の場合のみ使用できます。

Tau で UML デバッガを使用して C++ アプリケーションのデバッグを行う場合、Tau のデバッガがブレーク モードに入ったときに、制御を Visual Studio に移すことができます。このコマンドは、Tau の Model Verifier ツールバーのボタンとして提供されます。

生成されたコード内の次の実行文にブレークポイントを設定し、Tau のデバッガで Run コマンドを実行することによって、Visual Studio のデバッガに制御が移ります。

Visual Studio .NET コマンド

Locate in Tau

このコマンドは、Visual Studio .NET で開いている C#/C++ ソース ファイルから Tau モデルの要素にナビゲートするために使用します。

1. Visual Studio のコード エディタで、要素にカーソルを置いて選択します。
2. ［Tau］メニューから［Locate in Tau］を選択します。

モデル内で検出された要素は、Tau の［モデル ビュー］で選択されます。ダイアグラム内にプレゼンテーション要素がある場合は、そのダイアグラムも開きます。

生成されたコードのデバッグを行うときは、Tau へのナビゲーションを自動化できます。これによって、ブレーク ポイントに到達してデバッガの実行コマンド（Step または Run To Cursor など）が実行されたときに、関連するモデル要素が Tau 内で検索されるようになります。このモードを使用するためには、Visual Studio .NET で［Autolocate ］をオンにします。

Connection Status

このコマンドは、Visual Studio .NET アドインが Tau のどのインスタンスに接続しているかを表示するために使用します。接続が確立されていない場合は、接続をするためのダイアログとして使用できます。Tau と Visual Studio .NET の接続の詳細については Tau と Visual Studio .NET の接続 を参照してください。

Create/Update Tau Project

このコマンドは、C#/C++ ソースコードで行った変更で Tau プロジェクトを更新するために使用します。

注記

このコマンドは既存のファイルからのモデル更新のみをサポートしています。新しいファイルを Visual Studio .NET プロジェクトに追加した場合は、通常の手順で Tau にインポートしてください。

Autolocate

Autolocate を有効にすると、Visual Studio .NET デバッガがブレーク モードに入る度に（ブレークポイントに到達し、step/step into/step over/run to cursor コマンドを実行した後）、Tau はソース コード内の現在の位置に対応するモデル要素を強調表示します。

Autolocate モードは、Tau と Visual Studio .NET を並べて表示できる大きさのモニタを使用している場合に有用です。

Tau Trace

このコマンドは C++ の場合のみ使用できます。

Tau Trace コマンドにより、生成された C++ アプリケーションをトレースできます。トレースは、ログファイルか Tau のシーケンス図によって行われます。シーケンス図トレースの場合は、トレースを行う間 Tau が実行されている必要があります。ログファイルによってトレースを行った場合、Tau の［インポート］>［トレースのインポート］コマンドを使用して、後で Tau のシーケンス図として表示できます。

トレース機能を使用する場合、Tau から C++ コードを生成する前に、「インストルメンテーション」(デバッグ機能の配備) を有効にする必要があります。この詳細については、Enable instrumentation を参照してください。

トレース機能をアクティブにするには、次の手順を行います。

1. ［Tau］メニューから［Tau Trace］を選択します。
2. トレースの形態としてログファイルかシーケンス図を選択します。

注記

トレース機能は Visual Studio .NET 環境ではなく実行形式に埋め込まれているので、トレース設定の変更はデバッグ セッション中にのみ可能です。したがって、変更は実行がブレーク モード中に行わなければなりません。

69

印刷

この章では、ダイアグラムの各種印刷方法と印刷設定の変更方法について説明します。

プリンタの追加と削除（**UNIX**）

MainWin コントロール パネルを使用して、使用するプリンタを Tau 印刷ダイアログで有効にできます。

MainWin コントロール パネルを開くには

- ターミナル ウィンドウから以下のように入力します。
  `<installation directory>/bin/mwcontrol`

コントロール パネルが開きます。

プリンタを追加するには

1. コントロール パネルを開いたら、［Printers］をクリックします。［Printer］ウィンドウが開きます。
2. ［Add New Printer］をクリックして、表示されるプリンタの追加ウィザードの手順に従います。
3. ウィザードを完了したら［Printer］ダイアログとコントロール パネルを閉じます。
4. Tau を再起動します。

プリンタを削除するには

1. コントロール パネルを開いたら、［Printers］をクリックします。［Printer］ウィンドウが開きます。
2. 削除するプリンタを右クリックして、［Delete］をクリックします。
3. ［Printer］ダイアログとコントロール パネルを閉じます。
4. Tau を再起動します。

ダイアグラムの印刷

ダイアグラムを印刷する複数の方法があります。以下の場所から 1 つのダイアグラムを印刷できます。

- ダイアグラム自体
- モデル ビュー
- 印刷マネージャ
- ダイアグラムのプレビュー ウィンドウ

以下の場所から複数のダイアグラムを印刷できます。

- モデル ビュー
- 印刷マネージャ

注記

アイコンイメージの背景を白または透明にすると、印刷時に背景が黒になる場合があります。この問題は Windows のポストスクリプト ドライバ PostScript 言語レベル 2 に起因しています。PostScript 言語レベル 1 に変更すると、問題を解決できます。カラーの背景またはフレームを使用して、この問題を解決することもできます。

プリンタの追加と設定（**UNIX** のみ）

UNIX ホスト上の Tau から使用するプリンタの追加と設定をするには、インストール ガイドで詳しく説明されているとおりに **MainWin コントロール パネル**で必要な手順を実行します。

印刷設定

印刷の設定を変更するには

1. ［ファイル］メニューから［印刷設定］を選択します。
2. ［プリンタの設定］ダイアログで、プリンタと用紙サイズなどの選択されたプリンタに使用できるプロパティを選択します。用紙サイズと印刷の向きは、エディタでデフォルトのダイアグラム サイズを決定する際に使用されます。
3. ［OK］をクリックします。
1. ファイルを印刷します。

ファイルを印刷するには

1. 印刷するファイルを開いて、カーソルをテキスト内の任意の場所に合わせます。
2. ［ファイル］メニューから［印刷］をクリックするか、ツールバーの印刷アイコンをクリックします。
3. ［印刷］ダイアログで、必要に応じて設定を変更します。
4. ［OK］をクリックします。

印刷するダイアグラムの選択

モデル内のすべてのダイアグラムが［モデル ビュー］に表示されます。印刷マネージャを使用して、印刷するダイアグラムを選択できます。印刷マネージャを開くには、［ファイル］メニューから［印刷マネージャ］をクリックします。

［モデル ビュー］でアクティブになっているコンテナ内のダイアグラムが印刷マネージャにリストされます。［モデル ビュー］でコンテナを変更する場合は、［選択部分をトラック］ボタンを使用します。ボタンを押さないと、［印刷マネージャ］に含まれている内容が最初の選択に従ってロックされます。

図 292: 選択されていない状態の［選択部分のトラック］ボタン

［フィルタ］領域でダイアグラム タイプのチェック ボックスを選択または選択解除して、どのタイプのダイアグラムを印刷するかを決定することもできます。

印刷するページ数を計算するには、［印刷マネージャ］で［ページ］をクリックします。

ダイアグラムのプレビュー

ダイアグラムのプレビューを表示するには

1. ［モデル ビュー］でダイアグラムを選択します。
2. ［ファイル］メニューから［印刷 プレビュー］を選択します。ダイアグラムのプレビューが表示されます。
   - ［次のページ］ボタンと［前のページ］ボタンを使用して、他のダイアグラムに移動できます。

1 つのダイアグラムの印刷

ダイアグラム自体から 1 つのダイアグラムを印刷するには

1. ダイアグラムを開きます。
2. ［ファイル］メニューから［印刷］を選択します。標準の［印刷］ダイアログが表示されます。

［モデル ビュー］から 1 つのダイアグラムを印刷するには

1. ［モデル ビュー］でダイアグラムを選択します。
2. ダイアグラムを右クリックして、［印刷］を選択します。標準の［印刷］ダイアログが表示されます。

［印刷マネージャ］から 1 つのダイアグラムを印刷するには

1. ［モデル ビュー］でダイアグラムを選択します。ダイアグラム アイコンが［選択］領域に表示されます。
2. ［印刷ビュー］ボタンをクリックします。標準の［印刷］ダイアログが表示されます。

プレビュー ウィンドウから 1 つのダイアグラムを印刷するには

- ［印刷］を選択します。標準の［印刷］ダイアログが表示されます。

複数のダイアグラムの印刷

［モデル ビュー］から複数のダイアグラムを印刷するには

1. ［モデル ビュー］で複数のダイアグラムを選択します。
2. ［ファイル］メニューから［印刷マネージャ］を選択します。［印刷マネージャ］ウィンドウが表示されます。
3. ［印刷ビュー］ボタンをクリックするか、ファイル］メニューから［印刷プレビュー］を選択して、［印刷］を選択します。標準の［印刷］ダイアログが表示されます。

［印刷マネージャ］ウィンドウと［フィルタ］機能を使用すると、同じタイプのダイアグラムを同時に印刷できます。

［印刷マネージャ］から複数のダイアグラムを印刷するには

1. ［ファイル］メニューから［印刷マネージャ］を選択します。［印刷マネージャ］ウィンドウが表示されます。
2. ［モデル ビュー］で、印刷するダイアグラムを選択します。選択したダイアグラムタイプのダイアグラムとページ番号が［選択］領域に表示されます。
3. ［印刷ビュー］ボタンをクリックするか、ファイル］メニューから［印刷プレビュー］を選択して、［印刷］を選択します。標準の［印刷］ダイアログが表示されます。

70

モデルブラウザ

この章では、モデルまたはモデルの選択した部分について、ナビゲート可能な HTML ビューを生成する方法について説明します。

HTML の生成は ModelBrowser というアドインによって実行します。

HTML の生成は対話形式（「HTML の生成」参照）で行うか、またはコマンドライン（「コマンドラインの使用」参照）から実行できます。

HTML の生成

モデルの HTML ビューを生成するには、以下の手順を行います。

- ModelBrowser アドインが起動していることを確認します。
- ［モデル ビュー］で、レポートに含める要素を選択します。
- ［ツール］メニューから、［HTML の生成］を選択します。

HTML ビューが生成され、組み込まれた Web ブラウザに表示されます。詳細については、HTML ビューを参照してください。

注記

大規模なモデルの場合、HTML ビューの生成には時間がかかります。このプロセスの間、Tau の反応が遅くなります。

ModelBrowser アドインの起動

HTML 生成のサポートをアクティブにするには、以下の手順を行います。

1. ［ツール］メニューから［カスタマイズ］ダイアログを選択します。
2. ［アドイン］タブをクリックして、［ModelBrowser］アドインにチェックを付けて選択します。
3. ［OK］をクリックします。

HTML ビュー

このセクションでは、HTML ビューの表示とその内容について説明します。

内容

HTML ビューには以下の情報が含まれます。

- モデル ビューに対応する ツリー表示
- 含まれる各要素のプロパティ
- ダイアグラム

HTML ビューのメイン ページでは、3 つのフレームに情報が表示されます。

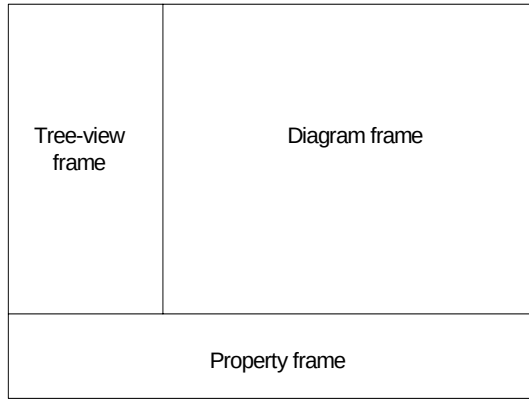


図 293: HTML ビューの表示

注記

HTML ビューの内容は定義済みなのでカスタマイズできません。アクティブなメタモデルは反映されません。

ツリー表示

ツリー表示はモデルの標準のツリー表示です。ツリー内の各ノードは 1 つの UML モデル要素を示します。

ツリー表示内のノードをクリックすると、プロパティ フレーム内に要素のプロパティが表示されます。要素が図の場合、対応する図がダイアグラム フレームに表示されます。

ノードを折りたたんだり展開すると、他のフレームの内容が変化します。

プロパティ

要素のプロパティはプロパティ フレームに表示されます。各要素で利用可能なプロパティは、そのメタクラスによって決まります。

HTML ビューに含まれる他の要素への参照はナビゲート可能です。

注記

各要素は、ModelBrowser アドインで定義されている独自のプロパティを保持します。要素のプロパティはカスタマイズできません。また、Tau に表示されるプロパティと必ずしも一致しているわけではありません。

ダイアグラム

ダイアグラム イメージは jpeg ファイルとして生成されます。モデル要素もレポートに含まれている場合、ダイアグラムによっては、対応するシンボルがクリッカブルなものもあります。ダイアグラム内のシンボルをクリックすると、モデル要素のプロパティがプロパティフレームに表示されます。

出力

このセクションでは、生成されるファイルの構造と内容を説明します。

ファイルとフォルダの構造

モデル ブラウザは一連のファイルとフォルダを生成して、モデルを HTML で表現します。html という名前のフォルダがプロジェクト ファイルと同じ場所に作成され、すべての生成ファイルはそのフォルダに配置されます。

ファイル構成は下図のとおりです。

```
html
        images
        index.html
        intro.html
        navigation.html

        ElementGuid1
                iElementGuid1.html
                ...
                iElementGuidN.html
                dDiagramGuid1.html
                dDiagramGuid1.jpg
                ...
                dDiagramGuidN.html
                dDiagramGuidN.jpg
        ...
        ElementGuidN
```

メインページは `index.html` です。`intro.html` はダイアグラム フレームの主コンテンツであり、`navigation.html` はツリービュー フレームの主コンテンツです。

`images` フォルダには、ツリービュー用とメイン html ページ用の画像ファイルがあります。

モデル情報の大半は、`ElementGuidN` という名前の別フォルダ（モデル用に 1 つ、各パッケージごとに 1 つ）に分けて配置されます。デフォルトで、フォルダには要素の GUID と同じ名前が付けられます。名前付けの詳細は、命名方法 セクションで説明します。

各フォルダには、要素を表現したファイルが入っています。各要素には、そのプロパティをプロパティ フレームに記述したファイル（`iElementGuidN.html`）があります。ダイアグラムには、その画像をダイアグラム フレームで表現した、さらに 2 つのファイル（`dDiagramGuidN.html, DiagramGuid1.jpg`）があります。

命名方法

HTML 生成の際、デフォルトでは、ファイルとフォルダの名前には GUID が用いられます。この命名方法は、以下の方法に変更できます。

- Guid ベース
- 変換つきの Guid ベース
- 単純 Simple

命名方法の変更に関する情報は、「命名方法の変更」を参照してください。

Guid ベース

この命名方法がデフォルトです。要素の GUID がフォルダ名とファイル名に使用されます。これらのファイル名やフォルダ名は永続的であり、新しい要素を追加するなどモデルを変更しても、この名前は変わりません。このプロパティは、出力を公開する場合してレビューを行う場合などに重要になります。この方法の欠点はフォルダ名とファイル名が長くなることです。短い名前が必要な場合は、単純 Simple 手法が適しています。

この方法では GUID の値は変換されないまま用いられるため、Windows プラットフォームでは若干の問題が発生する可能性があります。GUID の大文字小文字の区別は、Tau では行われますが、Windows ではフォルダ名やファイル名を大文字小文字では区別しません。モデルが名前の違いが大文字小文字だけであるような要素を含んでいる場合、命名方法を変更する必要があります。名前の永続性が必要な場合は、変換つきの Guid ベース方法を使用し、そうでない場合は単純 Simple を使用してください。

変換つきの **Guid** ベース

この方法は Guid ベースと同じですが、GUID の値の大文字小文字の区別に関する問題を回避しています。この方法はすべてのモデルに適用できますが、ファイル名とフォルダ名が長くなるため Windows プラットフォームでは問題になる場合があります。

この方法は後方互換性のために用意されています。Guid ベース または 単純 Simple を使うことを推奨します。

単純 **Simple**

この方法では、GUID の代わりに通常の数字をファイル名やフォルダ名に使用します。この方法を使えば、名前は短く意味のあるものになります。

この方法で付けた名前は永続的ではありません。新しい要素をモデルの内部に挿入して HTML を生成すると、挿入した要素以降のすべての要素の番号が変更されます。永続名が必要な場合は、Guid ベースまたは 変換つきの Guid ベース 手法を使用してください。

命名方法の変更

特定のモデルの命名方法を変更するには、以下の手順を実行します。

- ［モデルビュー］で［Model］ノードを右クリックします。（現在のビューで［Model］が表示されない場合は、［Standard View］に切り替えます）
- ［ステレオタイプ］から、`TTDModelBrowser::ModelBrowserSettings` ステレオタイプを適用します。
- プロパティエディタから命名方法を選択します。

コマンドラインの使用

次の構文に基づいてスクリプトを vcs.exe に渡すことによって、HTML レポートをコマンドラインから生成できます。

vcs -script <etc-path>/HtmlReport.tcl <project-file>

ここで、

vcs

インストール ディレクトリの bin フォルダにある Tau 実行形式ファイル

<etc-path>

Tau インストール ディレクトリの etc フォルダへのフル パスまたは相対パス

<project-file>

プロジェクトファイルへのフル パスまたは相対パス

生成されるフォルダはプロジェクト ファイルと同じフォルダに配置されます。

例 **791** ———————————————————————————————

この例では、C:/MyModels/MyProject.ttp に格納されている MyProject という名前のプロジェクトの HTML レポートを作成する方法を示しています。

最初に、コマンド ウィンドウを開き、現在のフォルダが Tau インストール ディレクトリの bin フォルダであることを確認します。

次に、以下のコマンドラインを実行します。

```
vcs -script ../etc/HtmlReport.tcl C:/MyModels/MyProject.ttp
```

このプロジェクトの HTML レポートが生成され、プロジェクト ファイルと同じ場所に格納されます。

———————————————————————————————————

注記

コマンドライン モードを使用すると、プロジェクト全体についてのレポートが生成されます。プロジェクトの一部またはモデルについてのレポートを生成するには、対話モードを使用します。

71

各国語対応サポート

このセクションでは、Tau の各国語対応サポートについて説明します。このドキュメントの主要テーマは、中国語、日本語、韓国語（CJK）の言語処理です。

サポートされている環境

このセクションでは、システム環境の各国語対応サポートに関する特定の情報を提供します。ここで説明されていない情報は、すべての言語に共通しています。全般情報については、インストール ガイドを参照してください。

サポートされているプラットフォーム

Tau の各国語対応サポートは、Windows XP で有効です。Windows のローカル バージョンを使用して、ローカル言語用のロケールを設定することを前提にします。

構成管理

Tau は、CJK サポートの各構成管理ツールの制限外の CJK 環境をサポートしていません。

IME（Input Method Editor）

Windows にバンドルされているデフォルトの IME をサポートしています。サポートされている IME を使用して、ローカル文字をインラインで入力できます。

フォントの設定

使用する言語での表示を正しく行うには、以下の手順でその言語に合ったフォントを選択してください。

1. Tau メニューバーで、［ツール］メニューから［オプション］を選択します。
2. ［形式］タブを選択します。
3. ［カテゴリ］を選択して、フォントを指定します。
   - **Dialog fixed font**：固定幅のフォントを使用するダイアグラム向けの設定です。
   - **Developer diagram symbol font**：その他のシンボルとダイアグラムに使用するフォントを設定します。
   - **Report Windows**：出力ウィンドウのタブに使用するフォントを設定します。
   - **Output Windows**：出力ウィンドウの［メッセージ］、［モデル ベリファイヤ］および［スクリプト］タブに使用するフォントを設定します。
   - **Tcl Files**：Tau で開いた Tcl ファイルおよびテキスト ファイルに使用するフォントを設定します。
   - **C/C++ Header/Source**：Tau で開いた C/C++ ヘッダ ファイルとソース ファイルに使用するフォントを設定します。

注記

以下の手順はダイアグラムに要素を作成する前に行っておく必要があります。

ダイアグラム要素についてのフォント設定があります。

1. Tau メニューバーで、［ツール］メニューから［オプション］を選択します。
2. ［フォント設定］タブを選択します。
3. 2156 ページの「［フォント設定］タブ」を参照してフォントタイプを設定します。

注記

［ダイアグラム要素のプロパティ］ツールバーから各要素のフォント スタイルとサイズを変更することもできます。

CJK 文字を使用したモデリング

Tau は、CJK 文字を使用したモデリングをサポートしています。以下の要素などに CJK 文字を使用できます。

- すべての要素の名前
- コメント
- Charstring 型リテラル

CJK 文字は英文字と同じように入力できます。CJK 文字を使用してモデルを作成するときに特殊な操作を行う必要はありません。

CJK 文字を使用したコード生成

モデル内の要素名は生成された C/C++ ファイルで識別子の名前として使用されます。ただし、C/C++ の文法では CJK 文字を識別子名としてそのまま使用することはできません。そのため、Tau には、C/C++ コード用の正当な名前を提供する仕組みが用意されています。以下の 2 つの方法を使用できます。

自動 UTF-16 命名

要素名に CJK 文字が使用されていると、Tau は生成された C/C++ コード用の正当な名前を提供します。提供された正当な名前は、UTF-16 ビッグ エンディアン エンコード化方式による 16 進文字列で構成されています。文字列の先頭には `tau_` が使用され、終りには `_tau` が使用されます。

例 **792: UTF-16** エンコード名

MALMO は以下のようにエンコードされます。

```
tau_004D0041004C004D004F_tau
```

ansiName ステレオタイプの使用

CJK 文字を要素名に使用して、生成された C/++ コード用に正当な名前を指定する場合は、ansiName ステレオタイプを使用できます。

1. ［ツール］メニューから［カスタマイズ］ダイアログを選択します。［カスタマイズ］ダイアログの［アドイン］タブを選択します。
2. **Internationalization** アドインのチェック ボックスを選択して、ansiName ステレオタイプを現在のプロジェクトにロードします（プロジェクトごとにこれをアクティブにする必要があります)。
3. CJK 文字を使用して、要素名をダイアグラムに入力します。
4. ［モデル ビュー］で、正当な名前を指定する要素を右クリックします。
5. ショートカット メニューから［プロパティ］をクリックします。
6. ［フィルタ］ボックスから、［ansiName］を選択します。
7. コード生成用の正当な名前を［名前］フィールドに入力します。

ansiName を使用して正当な名前を指定すると、指定された名前が C/C++ コードに使用されます。

UML 定義は、すべて ansiName ステレオタイプを通じて、C または C++ コード生成用に代替名を指定できます。

ビルド ツール チェーンによって使用されるファイルとフォルダの名前

クラス シンボルとパッケージ シンボルの名前

クラス シンボルとパッケージ シンボルの名前は、生成されたファイル名およびフォルダ名として使用されます。シンボル名で CJK 文字が使用されていると、CJK 文字を使用したコード生成で説明されているコード化機構を適用して生成されたフォルダとファイルが命名されます。ファイル名とフォルダ名を指定する場合は、2143 ページの「ansiName ステレオタイプの使用」で説明されているとおりに ansiName ステレオタイプを使用します。

コメント

コメントは生成された C/C++ ファイルに表示されません。これは CJK 文字と英文字を含めて他のすべての言語に当てはまります。

ビルド ツールによって使用されるファイルのエンコード タイプ

ビルド ツール チェーンによって処理されるファイル（中間形式を保持するファイルや生成された C/C++ ファイル）のエンコード タイプの場合は、**システム ロケール**エンコード形式によってエンコードされます。たとえば、日本語版 Windows の場合、生成されたファイルは Microsoft Visual C++ でファイルの処理とコンパイルを実行できるように SHIFT-JIS を使用してエンコードされます。

したがって、適切なエンコードによってファイルを生成するために、使用する言語の正しいロケールを設定する必要があります。

1. Windows のコントロール パネルを開きます。
2. ［地域と言語のオプション］ダイアログを開いて、［地域オプション］タブを選択します。
3. 選択されているロケールが正しいことを確認します。

テキスト ファイルの処理

テキスト ファイルを Tau 内で開くことができます。Tau は、テキスト ファイルのローカル ANSI エンコーディングと UTF-8 をサポートしています。既存のテキスト ファイルを Tau で開いた場合は、Tau によってファイルが元のエンコードの状態で保存されます。テキスト ファイルを Tau で作成した場合は、デフォルトによりファイルが UTF-8 で保存されます。［名前をつけて保存］ダイアログからエンコード タイプを選択できます。

制限事項

- Shift-JIS の 0x80 ～ 0xFF で定義されている 1 バイトのカタカナと日本語文字はサポートされません。
- CJK 文字はプロジェクト名に使用できません。

- CJK 名をメッセージとメッセージ テキスト パラメータに使用すると、シーケンス図でモデル ベリファイヤ（Model Verifier）トレースを実行しているときに、元の CJK 名ではなく UTF-16 ビッグ エンディアン エンコード名が表示される場合があります。元の名前を探すには、UTF-16 エンコード文字の表示をサポートしている外部アプリケーションでデコードを行う必要があります。

参照

自動 UTF-16 命名 .

72

ダイアログ ヘルプ

このセクションでは、ダイアログのヘルプ ボタンをクリックすると表示されるヘルプ テキストについて説明します。

［新規］ウィザード

［ファイル］タブ

このダイアログで、新しいファイルを追加できます。

- ファイルの追加時に、ファイル名と場所を指定する必要があります。
- ファイルは既存のプロジェクトに追加できます。ファイルを追加するには、このプロジェクトを［ファイル ビュー］で開く必要があります。
- 新しいファイルがデスクトップに開きます。

［プロジェクト］タブ

このダイアログで、新しいプロジェクトを追加できます。

プロジェクトを追加する際、プロジェクトの使用方法を指定します。以下に示すように、選択に従って起動時に異なるアドインがロードされます。

UML（AgileC コード生成用）
AgileCApplication、ModelVerifier、および RTUtilities アドインがロードされます。

UML（C コード生成用）
CApplication、ModelVerifier、および RTUtilities アドインがロードされます。

UML（C++ コード生成用）
CppAppGen および CppTypes アドインがロードされます。

UML（モデル ベリファイ用）
ModelVerifier および RTUtilities アドインがロードされます。

UML（モデリング用）
アドインはロードされません。

後でプロジェクトのデフォルトのアドインを変更する場合は、［カスタマイズ］ダイアログの［アドイン］タブ（［ツール］メニューから［カスタマイズ］ダイアログを選択）を使用して変更します。

- プロジェクトの追加時に、プロジェクト名と場所を指定する必要があります。
- プロジェクトは現在のワークスペースに入れるか、あるいはプロジェクトのために新しいワークスペースを作成します。

参照

第 1 章「Tau 4.3 の紹介」の 20 ページ、「プロジェクトの操作」
［アドイン］タブ

UML プロジェクト - 2 ページ目

このダイアログは、モデルを含むファイル用に推奨されるファイル ディレクトリと名前を表示します。

- この推奨値は変更するか、そのまま確定できます。
- オプションとして、空のパッケージを追加できます。

UML プロジェクト - 3 ページ目

このダイアログは、プロジェクト名と関連ファイルの名前を表示します

- ［完了］ボタンをクリックして名前を確定するか、［戻る］ボタンをクリックして変更できます。
- 新しいプロジェクトがワークスペース ウィンドウに表示されます。

ワークスペース

このダイアログで、新しいワークスペースを追加できます。

- ワークスペースの追加時に、ワークスペース名と場所を指定する必要があります。
- 新しいワークスペースは、ワークスペース ウィンドウにロードされます。

参照

第 1 章「Tau 4.3 の紹介」の 18 ページ、「ワークスペースの操作」

［カスタマイズ］ダイアログ

［コマンド］タブ

このタブには、ツールバーのボタンとともにデフォルト メニューやコマンド、およびメニューが一覧表示されます。これらのコントロールはツールバーやメニューに追加できます。このタブで、ツール バーのボタンを移動、追加、削除できます。

1. ［カテゴリ］ボックスで、カスタマイズするツールバーの名前をクリックします。
2. ［ボタン］領域で、ダイアログからツールバーに項目をドラッグします。最初に項目をクリックして、特定の項目に関する情報を表示します。
3. ツール バーから項目を削除するには、ツールバーからダイアログに項目をドラッグします。

ツール バーにボタンを追加するには

1. 変更しようとするツール バーが表示されていることを確認します。
2. ［カテゴリ］ボックスに、使用可能なツール バー ボタンや項目がグループ化されています。追加するツール バー ボタンや項目があるカテゴリを選択します。
3. ボタンや項目をクリックして、機能に関する情報を表示します。
4. ［ボタン］領域からユーザー インターフェイスのツール バーにボタンや項目をドラッグします。

ツール バーからボタンを削除するには

1. 削除しようとするツール バーが表示されていることを確認します。
2. ツール バーからボタンや項目をドラッグして削除します。

デフォルトのボタンをツール バーから削除しても、［カスタマイズ］ダイアログにはそのボタンが残ります。表示をカスタマイズしたツール バーボタンを削除すると、その表示は完全になくなります。ただし、コマンドは［カスタマイズ］ダイアログの［コマンド］タブから使用可能です。

ヒント

表示をカスタマイズしたツール バー ボタンを再利用のため保存するには、未使用のボタンを格納するツール バーを作成してこのボタンを移動し、格納したツール バーを非表示にします。

［ツール バー］タブ

このタブは、標準のツールバーとカスタムのツール バーを一覧表示します。

チェック ボックスを選択するか、選択を解除して、ツール バーを表示または非表示にします。各ツール バーはデフォルトの場所、または最後にツール バーが移動された場所に表示されます。メニュー バーは非表示にできません。

ツールチップを表示
このチェック ボックスをクリックし、ツール バーのボタンまたはフィールドにカーソルを移動したときツールチップが表示されるようにします。

大きいボタン
このチェック ボックスをクリックし、ツール バーのボタンのサイズを大きくします。

新規ツール バーを作成するには

1. ［新規］をクリックします。
2. 表示されたダイアログに、ツール バーの名前を入力します。新しいツール バーがインターフェイスのツール バー領域に表示されます。
3. ［コマンド］タブで、ツール バーに追加する項目を選択します。

デフォルトのツール バー設定を復元するには

1. リストのツール バーをクリックします。
2. ［リセット］をクリックします。

ユーザーが作成したツール バーは復元できません。

ユーザーが作成したツール バーを削除するには

1. リストのツール バーをクリックします。
2. ［削除］をクリックします。

デフォルトのツール バーは削除できません。

ユーザーが作成したツール バーの名前を変更するには

1. リストのツール バーをクリックします。
2. ［ツールバー名］フィールドにツール バーの新しい名前を入力します。
3. もう一度ツール バーをクリックして、変更を保存します。

新規ツールバーの作成

新しいカスタム ツール バーの名前を入力します。大文字または小文字を使用できますが、大文字／小文字に関係なく名前は一意でなければなりません。他のツール バーと同じ名前にはできません。この名前を後で変更する場合は、［ツール バー］タブの［ツールバー名］フィールドで名前を編集できます。

ウィンドウのレイアウト

このタブでウィンドウのレイアウトをカスタマイズできます。ドッキング ウィンドウのツール バーの位置、表示、場所を保存できます。

新規レイアウトの保存

1. ［新規］ボタンをクリックします。
2. レイアウトに付ける名前を入力します。
3. ウィンドウを閉じます。

新規レイアウトを復元するには

1. 復元するレイアウトをクリックします。
2. ［復元］をクリックします。

レイアウトを削除するには

1. 削除するレイアウトをクリックします。
2. ［削除］ボタンをクリックします。

［ツール］タブ

このタブで、［ツール］メニューにコマンドを追加できます。これらのコマンドをオペレーティング システムで実行されるプログラムに関連付けることができます。この情報は、以下のディレクトリの Tools.dat というファイルに保存されます。

```
C:¥Documents and Settings¥<user>¥Application
Data¥IBM¥Rational¥Shared
```

［ツール］メニューにコマンドを追加するには

1. ［新規（挿入）］ボタンをクリックします。空の矩形で示される空白行が、［メニューの内容］ボックスに表示されます。
2. ［ツール］メニューに表示されるように、コマンドの名前を入力します。**Enter** キーを押して、名前を保存します。
3. ［コマンド］フィールドにプログラムへのパスを入力します。参照ボタンをクリックしてプログラムの場所を検索することもできます。
4. ［引数］テキストボックスに、プログラムに渡す引数を参照または手動で入力します。［引数］テキストボックスの隣のドロップダウン ボタンをクリックして、使用できる引数のリストを表示できます。
5. ［初期ディレクトリ］ボックスで、コマンドを挿入するファイル ディレクトリを参照または入力します。
6. プログラムがコンソール プログラム（Windows コマンド プロンプトなど）であれば、出力ウィンドウで実行するようにできます。このためには、［出力ウィンドウを使う］チェック ボックスを選択します。
7. コマンドを使用するたびに引数を変更できるよう設定する場合、［引数を要求する］チェック ボックスを選択します。
8. アプリケーションの出力を OEM 形式にする場合、［OEM 形式を使う］チェック ボックスを選択します。
9. ［OK］をクリックします。［ツール］メニューにコマンドが表示されます。

その他の作業

- サブメニューにコマンドを挿入するには、メニュー名とコマンド名を円記号「¥」で区切ります。たとえば、エディタ メニューのコマンド「メモ帳」は「`editor¥Notepad`」と入力します。
- アクセス キーを挿入するには、名前内で選択する文字の前にアンパサンド「&」を入力します。
- ［上に移動］ボタンと［下に移動］ボタンを使用して、メニュー内でコマンドを上下に移動します。
- コマンドの名前を変更するには、コマンドをダブルクリックして、新しい名前を入力します。

［ツール］メニューからコマンドを削除するには

1. リストのコマンドをクリックします。
2. ［削除］ボタンをクリックします。

［アドイン］タブ

このタブは、ツールの機能を拡張するために使用します。［アドイン］タブを使用して、任意の数のアドインをロードできます。

アドインを使用すると、さまざまな条件に対応した環境を簡単に作成できます。アドイン名をクリックすると、［説明］フィールドに使用法が表示されます。

- アドインをロードまたはアンロードするには、チェックボックスを選択または選択解除します。ダイアログを閉じます。
- 使用可能なアドインを変更できます。アドインをクリックして、［修正］をクリックします。表示されたダイアログで、必要に応じてアドインをカスタマイズできます。
- また、［作成］をクリックして、独自のアドインを登録することもできます。表示されたダイアログで、必要に応じてアドインを設定できます。

参照

第 55 章「Tau のカスタマイズ」の 1733 ページ、「アドインの内容と構造」

［オプション］ダイアログ

［一般］タブ

このタブで一般的なオプションを設定できます。

［ステータスバーを表示する］
Tau ユーザー インターフェイスの下部にあるステータス バーの表示、非表示を切り替えられます。

［内容の受信時に出力ウィンドウを表示する］
出力ウィンドウを閉じると、通常このウィンドウのさまざまなタブに現れる情報が表示されなくなります。しかし、このオプションを選択すると、手動でチェックを行った後など、新しい情報を表示する必要がある場合に、出力ウィンドウが自動的に開きます。

印刷マネージャで選択部分をトラックする
印刷マネージャはデフォルトでモデルビューのアクティブな選択部分をトラックします。このオプションを選択すると、このトラッキングが無効になります。

詳細オプションページを表示する
このチェック ボックスを選択すると、すべてのオプションをツリー構造で表示する［詳細］タブが表示されます。いくつかのオプションはこの詳細オプション表示でのみ表示されます。

タブ付きドキュメント
このチェックボックスを選択すると、1 つのウィンドウにドキュメントがタブとして表示されます。

ウェルカムページを起動時に表示する
このオプションはツール起動時のウェルカムページの表示 / 非表示を制御します。このオプションはウェルカムページからも設定できます。オフにした場合は、ウェルカムページは［ヘルプ］メニューから開いてください。

ソースコントロールプロバイダ
ソース コントロール システムをインストールしている場合、このチェックボックスを選択できます。このチェック ボックスを選択すると、ソース コントロール メニューとツール バーを通じた、ソース コントロール システムとの相互作用が可能になります。詳細は 構成管理 を参照してください。

ファイルの自動更新
このオプションは、ジェネリック ソースコントロールをソースコントロールプロバイダとして選択している場合に使用できます。このオプションが有効になっている場合は、ファイルは、チェックアウト前に自動的に CM システムから更新されます。

次の種類のファイルに対する外部プログラムの起動を無効にする
このフィールドでファイルの拡張子を指定し、これらのファイルを Tau から開いたとき、関連付けられている外部アプリケーションが起動しないように設定できます。たとえば、`.txt` 拡張子を追加すると、Tau からテキストファイルを開いた場合、外部のテキストエディタではなく Tau のテキストエディタでファイルが開かれます。

デフォルトのヘルプ コンテキストを選択する
IBM Rational の複数のツールをインストールしている場合、［デフォルトのヘルプ コンテキストを選択する］ボックスで、デフォルトとして使用するヘルプ ファイルを選択できます。

URN Map
URN（Universal Resource Name）Map を使用して、ファイル保存場所の短縮名を定義できます。

例：

```
home:C:¥MyHomeDir;work:C:¥MyWorkDir
```

「`home`」は `C:¥MyHomeDir` の短縮名、「`work`」は `C:¥MyWorkDir` の短縮名です。ユーザーは独自の環境の URN を定義できます。定義した短縮名は、コンポーネントがファイル、ビットマップなどのリソースを参照する際に使用されます。

「`u2`」および「`u2user`」の 2 つの名前があらかじめ用意されています。

- `u2` は、のインストール ディレクトリへマッピングされます。
- `u2user` は、ユーザー アドインのある場所へマッピングされます。

例 **793: URN** ファイル参照（**URN** の使用）

上記で説明した名前／ディレクトリのマッピングを使用すると、以下の URN ファイル参照は、

```
urn:home:mybitmap.bmp
```

次のように展開されます。

```
C:¥MyHomeDir¥mybitmap.bmp
```

以下の参照は、

```
urn:u2:etc¥TTDMetamodel.u2
```

次のように解釈されます。

```
C:¥Program Files¥IBM¥Rational¥TAU¥4.3¥¥etc¥TTDMetamodel.u2
```

インストール ディレクトリが以下の場合、

```
C:¥Program Files¥IBM¥Rational¥TAU¥4.3¥
```

これらの URN は、例としては、アドイン etc ファイルやアイコン参照の画像、およびハイパーテキスト参照のエンコードに使用できます。

保存

このタブで、Tau の保存オプションを設定できます。

ツールを実行する前に保存する
外部ツールが起動する前に未保存の作業をすべて自動保存するよう設定します。

ファイルとプロジェクトを保存する前に尋ねる
エディタを閉じる際、ファイルやプロジェクトの保存プロンプトを表示するよう設定します。

外部で修正されたファイルを自動的に再ロードする
デフォルトで、情報メッセージが表示され、外部修正したファイルを再ロードするよう促されます。このオプションを選択すると、Tau 以外のツールで修正したファイルが自動的に再ロードされます。

すべてのロードされたプロジェクトのアドイン状態を保存する
ロードされているアドインを、現在ロードしているすべてのプロジェクトでアクティブにします。

自動バックアップ
［有効にする］チェック ボックスを選択し、あらかじめ設定した間隔でモデルを自動保存するよう設定します。数字を入力するか上下ボタンをクリックして、任意の保存間隔（分単位）を入力できます。

［ワークスペース］タブ

このタブで、開いているワークスペースに一般的なオプションを設定できます。

起動時に前回のワークスペースを再ロードする
Tau を最後に使用した際に作業していたワークスペースを開くように設定します。

プロジェクト ファイルのステータス変更時に警告を出す
作業中のプロジェクト ファイルの状態が読み取り専用に変更された場合、警告を受け取るように設定します。これで未保存の作業を消失する危険を回避できます。

プロジェクトのデフォルトの場所
新規プロジェクトの作成時に、プロジェクト ファイルの保存場所が表示されるように設定します。このテキスト フィールドで、新規プロジェクトを保存するフォルダのパスを入力または参照できます。

［形式］タブ

このタブで、ウィンドウとファイル内のテキストと色の表示を調整できます。

カテゴリを選択すると、以下を選択できます。

- ファイルやウィンドウ内のテキストの**フォント**と**サイズ**
- 選択したカテゴリの背景色とテキストの色。デフォルトで、コントロール パネルで定義している配色が使用されます。［自動］チェック ボックスの選択を解除すると、テキストと背景色を設定できます。

［フォント設定］タブ

このタブで、新規ダイアグラムを作成する際に使用するデフォルトのフォントを選択できます。

ダイアグラムフォント設定
このフォント設定では、通常のダイアグラム要素のデフォルトの表示を決定します。

固定フォント設定
このフォント設定では、固定幅フォントで表示したほうが見映えのいいテキストを持つシンボルのデフォルトの表示を決定します。このフォント設定が適切であるシンボルの例として、テキストシンボルがあります。［有効］チェックボックスをチェックすると、この種類のシンボルを作成したときに、固定幅のフォントが適用されます。

ラベルフォント設定
このフォント設定は、ダイアグラム要素の主たるラベル以外のテキストラベルのデフォルトの表示を決定します。例としては、クラスシンボルの属性や操作のラベルがあげられます。［有効］チェックボックスをチェックすると、この種類のラベル要素にこのフォント設定が適用されます。

［リンク］タブ

このタブで、リンク作成の振る舞いをカスタマイズできます。

修正されたオブジェクトからアクティブなリンク端へのリンクにする
このオプションを選択せずに、リンクの自動作成を使用すると、アクティブなリンク端から他のモデルへのリンクを作成できます。このオプションを選択すると、他のモデルからアクティブなリンク端へのリンクが生成されます。

修正されたオブジェクトとアクティブなリンク端とのリンクを自動作成
このオプションを選択して、アクティブな 1 つのリンク端を選択すると、すべての修正内容がこのリンク端にリンクされます。

リンク インジケータを表示
このオプションを選択すると、Tau でリンク マーカーが表示されます。

ドラッグ & ドロップでリンクを作成するときに、要求をターゲットとして使用
リンクはドラッグ & ドロップ操作で作成できます。このオプションが選択されていると、リンクのターゲットは要求になります。このオプションが選択されていない場合は、要求はリンクのソースになります。

［UML 基本編集］タブ

このタブで、ダイアグラム エディタ、プロパティエディタ、モデルビューの基本的な設定を変更できます。

ダイアグラム エディタ

グリッドを表示する
このオプションを選択すると、エディタにグリッドが表示されます。このオプションはグローバルです。つまり、このオプションを選択すると、すべてのダイアグラムにグリッドが表示されるようになります。1 つのダイアグラムでグリッドの表示／非表示を切り替える場合は、ダイアグラムを右クリックしてショートカット メニューから［グリッドの表示］を選択します。

ページ区切りを表示する
このオプションを選択すると、ダイアグラムの出力時にページ区切りが表示されます。このオプションはグローバルです。つまり、このオプションを選択すると、すべてのダイアグラムでページ区切りが表示されるようになります。1 つのダイアグラムでページ区切りの表示／非表示を切り替える場合は、ダイアグラムを右クリックしてショートカット メニューから［ページ境界の表示］を選択します。

最後に参照したシンボル / ラインの削除時にモデル要素を削除する
このオプションを選択すると、ダイアグラムの最後のプレゼンテーション要素（シンボルまたはライン）が削除されたときに、モデル要素が自動的に削除されます。このオプションの選択を解除した場合、未使用のモデル要素は手動で個々に削除する必要があります。

モデル要素への最後の参照シンボル / ラインの削除の場合はダイアログを表示する
このオプションを選択すると、［最後に参照したシンボル / ラインの削除時にモデル要素を削除する］オプションの設定によってモデル要素が削除されるときに、警告が表示されます。

スクロールとズームの設定を記憶する
このオプションを選択すると、ズームのレベルとスクロール位置がダイアグラムに記憶されます。この情報は `projectname_DiagramSettings.u2x` という名前のファイルに保存されます。このファイルを削除してもモデルには影響ありません。

属性と操作ラベル上にステレオタイプを表示する
このオプションを選択すると、属性と操作に適用されているステレオタイプが、クラス系のシンボル（クラスシンボル、インターフェイスシンボルなど）のラベルに表示されます。

デフォルトのモデル ビュー フィルタ

この領域では、ワークスペースの［モデル ビュー］にデフォルトで表示させるオブジェクトを決定します。この設定はグローバルです。プロジェクトごとに設定を変更する場合、［モデル ビュー］を右クリックし、モデル ビュー フィルタにカーソルを合わせて目的のフィルタをクリックします。

定義をソートする
［定義をソートする］フィルタを有効にすると、定義（モデル要素の名前）が辞書編集上の順序（A から Z、a から z、など）で表示されます。

デフォルトのモデル ビュー

このドロップダウンリストにより、ワークスペースの［Standard View］から表示を切り替えられます。この設定はグローバルです。特定のプロジェクトの外観を変更したい場合は、［表示］>［モデルビューの再構成］を使用します。

デフォルトのプロパティビュー

このドロップダウンリストにより、デフォルトでプロパティエディタでどのようなプロパティビューを使用するかを指定できます。このオプションは設定後に開いたプロパティエディタについて有効です。すでに開いているプロパティエディタについて切り替えるには、そのエディタで［オプション ...］ボタンを押します。

デフォルトのプロパティフィルタ

このドロップダウンリストにより、デフォルトでプロパティエディタでどのフィルタを使用するかを指定できます。このオプションは設定後に開いたプロパティエディタについて有効です。すでに開いているプロパティエディタについて切り替えるには、そのエディタで［オプション ...］ボタンを押します。

デフォルトのクラスシンボルの外観

これらの設定は、クラス系のシンボル（クラスシンボルやインターフェイスシンボルなど）がエディタで描画された場合、その初期表示に影響を及ぼします。これは、新規作成されたシンボルと、［モデル ビュー］からドラッグされたシンボルの両方に影響を及ぼします。

折りたたみ
クラス シンボルの最上部入力領域のみ表示されます。

属性の表示
シンボルが折りたたまれていなければ、すべての既存属性が表示されます。

操作の表示
シンボルが折りたたまれていなければ、すべての既存操作が表示されます。

ポートの表示
すべての既存ポートが表示されます。

ステレオタイプ区画を表示する
適用されているすべてのステレオタイプがステレオタイプ区画に表示されます。

制約区画を表示する
すべての制約が制約区画に表示されます。

［UML 詳細編集］タブ

このタブで、対応するダイアグラムの詳細をグローバルに設定できます。

テキスト

インデント レベル
［自動的にインデントする］オプションを選択した場合、［インデント レベル］オプションによって「{」または「)」の次に挿入するスペース数を指定します。また、タブ間の距離も指定できます。

テキストの強調に色を使う
このオプションを選択すると、セマンティックおよび構文が色によって強調されます。色による構文の強調の詳細については、第 2 章「モデルの操作」の 57 ページ、「テキストの強調表示」を参照してください。

強調表示された名前にマウスのカーソルを合わせると、テキストの現在のステータスがツールチップに表示されます。特にある種の状況診断が表示されます。たとえば、「This reference is currently not bound, see AutoCheck log for details.」（この参照は、現在バインドされていません。オートチェックのログで詳細を確認してください。）などのメッセージが表示されます。

構文エラーのテキストを強調する
このオプションを選択すると、状況別に色分けされて構文エラーが強調されます。構文の強調の詳細については、第 2 章「モデルの操作」の 57 ページ、「テキストの強調表示」を参照してください。

強調表示された名前にマウスのカーソルを合わせると、テキストの現在のステータスがツールチップに表示されます。特にある種の状況診断が表示されます。たとえば、「This reference is currently not bound, see AutoCheck log for details.」（この参照は、現在バインドされていません。オートチェックのログで詳細を確認してください。）などのメッセージが表示されます。

自動的にインデントする
このオプションを選択すると、「{」または「)」に続く行に自動的にインデント スペースが挿入されます。挿入するスペース数は［インデント レベル］オプションで指定します。

([“および‘の自動マッチングを行う
このオプションを選択すると、開きかっこまたは引用符の直後に、閉じかっこまたは引用符が挿入されます。

編集中のモデル更新を無効化する
このオプションを選択すると、テキスト編集モードが終了した場合（ラベルまたはテキスト シンボルの外側で別のものを選択した場合など）のみモデルが更新されます。編集したテキストは編集モードを終了するまでモデルにはマージされません。これは、編集に要している時間には関係ありません。

シーケンス図

メッセージ間隔
このオプションでメッセージ間の間隔を指定します。デフォルト値は 10 ミリです。

ライフライン間隔
このオプションでライフライン間の間隔を指定します。デフォルト値は 25 ミリです。

シーケンス図トレースウィンドウをドッキング
シーケンス図トレース ウィンドウを、ドッキングされたウィンドウとして開くかどうかを指定します。デフォルトはオン（ドッキング）です。

アクティビティ図

アクティビティ図内のデフォルトのフロー方向を指定します。このオプションは自動生成されたシンボルがどのようにアクティビティ図に配置されるかを制御します。

シンボルの外観

デフォルトのシンボル色
ダイアグラム内のシンボルの色を決定します。色を変更するには、色のフィールドをクリックしてカラー パレットから変更後の色を選択するか、［Other］を選択してカスタム カラーを指定します。

ポートシンボルの矢印
ポートシンボル内に矢印を表示して、通信フローが有効なことを示します。矢印の方向はポートの位置を参照して垂直または水平の線上に表示されます。

ダイアグラムのツールチップ

シンボルとラインのツールチップを表示
このオプションを選択すると、シンボルとラインのツールチップが表示されます。このツールチップは、シンボルやラインで表現されるモデルのさまざまな情報（コメントやエラーメッセージなど）を提供します。

編集モードのツールチップを表示
このオプションを選択すると、編集モードでツールチップが表示されます。

名前の完成

'.' または '::' の入力時に名前の完成ウィンドウを表示する
このオプションは、メンバーアクセス（.）や修飾子の区切り文字 （::）の入力時に名前完成ウィンドウを開くかどうかを制御します。名前完成ウィンドウは、現在のコンテキストから参照可能な定義の名前をリストします。

名前ベースの参照の自動更新

モデルビューでの定義の移動時
このオプションを使用すると、特定の定義をモデルビュー内で移動したときに、その定義へのすべての参照を移動に伴って更新できるようになります。このオプションはデフォルトでオンです。参照を手動で更新したい場合や、そもそも移動に伴って参照を更新すべきではない場合には、無効にできます。

［UML ラインスタイル編集］タブ

このタブのオプションは、エディタでのデフォルトのラインの形状を制御します。利用可能な UML ライン種別ごとに 1 つのオプションがあります。値の意味は以下のとおりです :

自動経路選択
このオプションを選択すると、ラインの頂点は障害物を避けて配置され、ラインは直交になります。

ベジエ
ラインは曲線になります。ラインを選択すると 2 つのコントロールポイントが表示されます。このコントロールポイントを使用して曲線の形を変えることができます。

直交
ラインは常に直角になり、ラインの頂点とセグメントは移動できます。頂点はラインに追加／削除が可能です。

非直交
ラインの頂点は、無制限に移動、追加、削除できます。

注記

1 本のラインのみレイアウトを変更する場合、ラインを右クリックしてショートカット メニューからライン スタイルを選択します。現在のラインスタイルには、アクティブなラジオボタンがついています。

［UML チェック］タブ

このタブで、手動チェックおよびオートチェックのタイプを指定できます。

オートチェック

以下のオプションはモデルの編集中にどのような自動チェックを行うかを制御します。

ダイアグラムをチェックする
このオプションを選択すると、ダイアグラム内の構文エラーがレポートされます。

簡単な文法チェックを有効にする
このオプションを選択すると、簡単な文法エラーがレポートされます。

バインド エラーを表示する
このオプションを選択すると、解決できない識別子がレポートされます。

チェック

ダイアグラムのチェックを有効にする
このオプションを選択すると、構文エラーがレポートされます。モデルとプレゼンテーションの不整合についても同様にレポートされます。

参照を再バインド

このオプションをオン（デフォルト）にすると、モデルの編集中にモデル内の参照を自動的に再バインドします。これは、編集中に常にすべてのバインドが正確かつ最新になるように、必要に応じて行われます。モデルで［すべてをチェック］コマンドを使用するまで再バインドを行わないようにするには、このオプションをオフにします。サイズの大きいモデルを編集する際は、このオプションをオフにするとパフォーマンスが向上します。

［ハイパーリンク］タブ

このタブで、ハイパーリンクの振る舞いを指定できます。

デフォルトで、ワークスペース要素にハイパーリンクを作成する
このオプションを選択すると、ワークスペース内にハイパーリンク先を選択できます。チェックマークを付けないと、既存ファイルや Web ページなどの別タイプのターゲットを指定するように指示されます。

［比較 / マージ］タブ

このタブでは、比較とマージのオプションを変更できます。

外部テキスト比較 / マージ

Telelogic Synergy がインストールされている場合、このオプションのデフォルト値は、Telelogic Synergy のテキスト比較 / マージ ツールです。

外部テキスト比較 / マージ ツール パス

［相違点のレビュー］ダイアログから使用できる外部テキスト比較 / マージツールへのパスです。

コマンドライン スイッチ

比較 / マージ の使用法の種類に応じて、外部比較 / マージ ツールの呼び出しに、対応するコマンドライン スイッチ オプションが設定されます。

レビュー情報を保存

差分リストイメージ

ピクセル単位の、生成イメージの幅と高さです。

ステレオタイプインスタンスと未解析テキストのコメント

このオプションを設定すると、コメントとグラフィカル表現を持たないステレオタイプインスタンスは、生成された XML ファイル中に、テキストとして追加されます。

［詳細］タブ

このタブで、ツリーコントロールを使用してすべてのオプションの値を変更できます。ツリーは、オプションカテゴリと実際のオプションリーフから構成されます。ここには、他のオプションタブでも設定できるオプションとともに、ここでのみ設定できるオプションがあります。ツリーコントロールを展開して、変更したいオプションを検索します。オプションの値を変更するには、値を選択して F2 キーを押します。

Web server

Studio - Settings - WebServer

PortRangeBegin オプションと **PortRangeEnd** オプションは、Tau Web サーバーが使用する TCP/IP ポートの範囲を定義します。Tau を使うマシンでこれらのデフォルトのポート番号がすでに使用されている場合は、ここで値を変更します。

Proxy settings

U2 - Options - ProxySettings

Host、**Password**、**User** の各オプションは HTTP プロキシーサーバーを通して Web にアクセスする場合に設定します。設定すると Tau から URL を使って情報にアクセスする際に常にプロキシーサーバーが使用されます。たとえば、WSDL ファイルをある URL からインポートするような場合などです。Host オプションの形式は、`<address>:<port>` です。

エディタのショートカット

要素の表示

このダイアログで、既存のモデルから一括で選択したシンボルを、別のダイアグラムに追加できます。

- 要素リストのチェック ボックスをチェックして、複数の要素を選択できます。
- この要素リストには現在設定されているスコープの要素が含まれます。
- ［スコープの選択］ボタンを使用して、モデル内の任意のスコープから要素をリストに追加できます。

あらかじめ要素を選択してこのダイアログを表示した場合、リスト内でその要素がすでに選択されています。

- 新しいダイアグラムがデスクトップに開きます。

モデル

モデル ビューの再構成

使用するブラウザ モデルを選択します。あらかじめ定義されたブラウザ ビューが 2 つあります。［Standard View］は、設計の詳細を含んだロードされたモデルの総合的なビューを提供します。このビューは、デザイン指向のユーザーに適しています。

もう 1 つの［Diagram View］は、ロードされたモデルの簡単なビューを提供します。このビューは、設計指向のユーザーに適しています。

参照

第 4 章「UML 言語ガイド」の 321 ページ、「メタモデル」

その他のオプション

ステレオタイプ

要素に適用するステレオタイプを選択します。それぞれの行をクリックすると、各ステレオタイプの説明が表示されます。適用できるステレオタイプの数は選択した要素によって異なります。

参照

第 4 章「UML 言語ガイド」の 322 ページ、「ステレオタイプ」

ビルドルートの選択

ビルド ルートとして使用するクラスを選択します。

参照

第 4 章「UML 言語ガイド」の 309 ページ、「アーティファクト」

モデル ベリファイヤ（**Model Verifier**）

コンソール ウィンドウ

Model Verifier コンソール ウィンドウ

メッセージ ウィンドウ

メッセージ ウィンドウ

再実行

再実行 : ［中断］ボタンをクリックした後でクリックすると、実行が再開されます。

モデル ベリファイヤ（**Model Verifier**）の停止

停止 : クリックすると実行が停止され、モデル ベリファイヤ（Model Verifier）が終了します。

73

その他のリソース

このセクションでは、ヘルプ ファイル以外で Tau に関する知識を広げるために役立つドキュメントについて説明します。役に立つ Web へのリンクも提供しています。

リンク

IBM Rational ソフトウェア・サポートへの問い合わせ

お手持ちのリソースで、問題が解決されない場合は、IBM® Rational® ソフトウェア・サポートに連絡してください。IBM ソフトウェア・サポートでは、製品の問題解決に関する支援を行っています。

前提条件

IBM Rational ソフトウェア・サポートに問題を送信するには、有効な Passport Advantage® ソフトウェア保守契約が必要です。パスポート・アドバンテージは、IBM の包括的ソフトウェア・ライセンスおよびソフトウェア保守 (製品のアップグレードおよび技術支援) オファリングです。次のサイトからオンラインでパスポート・アドバンテージに登録できます。
lhttp://www.ibm.com/software/lotus/passportadvantage/howtoenroll.htm

- パスポート・アドバンテージについて詳しくは、パスポート・アドバンテージ FAQ (http://www.ibm.com/software/lotus/passportadvantage/brochures_faqs_quickguides.html) にアクセスしてください。
- さらに支援が必要な場合は、IBM 担当員に連絡してください。

問題をオンラインで (IBM Web サイトから) IBM Rational ソフトウェア・サポートに送信するには、さらに以下が必要です。

- IBM Support Web サイトの登録ユーザーであること。登録について詳しくは、http://www.ibm.com/software/support/ を参照してください。
- 許可された呼び出し元としてサービス要求ツールにリストされていること。

問題報告について

次のようにして、IBM Rational ソフトウェア・サポートに問題を送信します。

1. お客さまの問題のビジネス・インパクトを判別します。IBM へ問題を報告する際は、重大度レベルを問われます。そのため、報告する問題とそのビジネス・インパクトを理解して、評価する必要があります。

重大度のレベルを決めるにあたっては、下表を参照してください。

重大度	**説明**
1	問題は*危機的*なビジネス・インパクトを持ちます。プログラムを使用できず、業務に重大な影響が出ています。この状況には、即時に解決策が必要とされます。
2	問題は、*重大*なビジネス・インパクトを持ちます。プログラムは使用可能ですが、非常に限定されています。
3	問題は*部分的*なビジネス・インパクトを持ちます。プログラムは使用可能ですが、比較的重要でない (業務に大きな影響はない) 機能が利用できません。
4	問題は*わずか*なビジネス・インパクトを持ちます。問題による業務への影響がほとんどないか、問題に対する有効な回避策が実施済みです。

2. 問題を説明して、背景情報を収集します。IBM に問題を説明する際は、なるべく具体的に説明してください。IBM Rational ソフトウェア・サポートの専門家が、問題を解決するために効果的な支援をできるように、関連するすべての背景情報を含めてください。時間を節約するために、以下の質問の答えを用意してください。

   - 問題の発生時に実行していたソフトウェア（複数可）のバージョンは何ですか？

   次のオプションを使用して、正確な製品名とバージョンを判別することができます。

   - IBM Installation Manager を始動して、「ファイル」>「インストール済みパッケージの表示」を選択します。パッケージ・グループを展開し、パッケージを選択して、パッケージ名およびバージョン番号を確認します。
   - 製品を始動して、「ヘルプ」>「製品情報」をクリックし、オファリング名とバージョン番号を確認します。
   - オペレーティング・システムおよびバージョン番号 (サービス・パックまたはパッチを含む) は何ですか？
   - 問題の症状に関連するログ、トレース、およびメッセージはありますか？
   - 問題を再現できますか？再現できる場合は、問題を再現するための手順は何ですか？
   - システムに変更を加えましたか？例えば、ハードウェア、オペレーティング・システム、ネットワーキング・ソフトウェア、またはその他のシステム・コンポーネントに変更を加えましたか？
   - 現在、問題に対する何らかの回避策を使用していますか？使用している場合は、問題の報告時にその回避策も説明する準備をお願いします。

3. IBM Rational ソフトウェア・サポートに問題を送信します。次の方法で、IBM Rational ソフトウェア・サポートに問題の送信ができます。

   - **オンラインの場合：** IBM Rational ソフトウェア・サポートの Web サイト (https://www.ibm.com/software/rational/support/) にアクセスして、Rational サポート・タスク・ナビゲーターで「**サービス要求を開く (Open Service Request)**」をクリックします。エレクトロニック問題報告ツールを選択し、「問題管理レコード (PMR) (Problem Management Record (PMR))」を開き、問題についてご自身の言葉で正確に記述してください。

注記

サービス要求を開く方法について詳しくは、
http://www.ibm.com/software/support/help.html にアクセスしてください。

IBM Support Assistant を使用してオンラインのサービス要求を開くこともできます。詳しくは、http://www.ibm.com/software/support/isa/faq.html を参照してください。

- **電話の場合：**国または地域別の電話番号を調べるには、http://www.ibm.com/planetwide/ の「IBM directory of worldwide contacts」で、お住まいの国名または地域名をクリックします。
- **IBM 担当員に依頼する場合：**オンラインまたは電話で IBM Rational ソフトウェア・サポートにアクセスできない場合は、IBM 担当員に連絡してください。必要な場合は、お客さまに代わって、IBM 担当員がサービス要求を開くことができます。http://www.ibm.com/planetwide/ で、各国への詳しい連絡先情報を検索できます。

送信した問題が、ソフトウェアの障害に関するものか、資料の欠落や不正確な記述によるものである場合は、IBM Rational ソフトウェア・サポートはプログラム診断依頼書 (APAR) を作成します。APAR には、問題の詳細が記述されます。IBM ソフトウェア・サポートは可能な限り、APAR が解決されてフィックスが提供されるまでの間に実施できる回避策を提供します。IBM は、同一の問題を経験している他のユーザーが同じ解決方法を利用できるように、IBM Rational ソフトウェア・サポート Web サイトに解決済みの APAR を公開し、毎日更新しています。

UML ドキュメント

- **UML チュートリアル**

  このチュートリアルは、UML モデルの設計に沿ったシナリオでツールの使い方とモデリングの基礎を解説しています。さまざまなダイアグラムと、Tau を使用してモデルを検証する方法についても説明しています。

  チュートリアルは、インストール ディレクトリの locale\etc に tutorial.pdf の名前であります。
- **UML クイック リファレンス ガイド**

  このドキュメントでは UML で一般的に使用されているグラフィックとテキスト構成要素の例を示します。

  クリック リファレンス ガイドはインストール ディレクトリの locale\etc に quickref.pdf の名前であります。

その他のリンク

Borland C/C++

Borland ビルダでサポートされる C/C++ です。
http://www.borland.com/cbuilder

Cygwin

Cygwin の各バージョンの正確な内容については以下の Web サイトをご覧ください。
http://www.cygwin.com

GNU C/C++

GNU Compiler Collection でサポートされている C/C++ です。
http://www.gnu.org/software/gcc

ITU-T

旧 CCITT
http://www.itu.int/

Macrovision

FLEXnet または Macrovision の詳細については以下の Web サイトをご覧ください。
http://www.macrovision.com

Microsoft Visual C/C++

Microsoft Visual C++ でサポートされる C/C++ です。
http://msdn.microsoft.com/visualc

MISRA

AgileC コード ジェネレータで生成されたコードは、2004 年 10 月より、「MISRA-C:2004 Guidelines for the use of the C language in critical systems」に記述されている MISRA コーディング ルールにほぼ適合しています。以下の Web サイトをご覧ください。

http://www.misra.org.uk

OCL

OCL (Object Constraint Language）の詳細については以下の Web サイトをご覧ください。
http://www.omg.org

OMG

Object Management Group（OMG）の詳細については以下の Web サイトをご覧ください。
http://www.omg.org

PDF

PDF ファイルを読むには Adobe Acrobat Reader を使用します。
www.adobe.com

Tcl

詳細については以下の Tcl Developer の Web サイトをご覧ください。
http://tcl.activestate.com/

TTCN-3

TTCN-3 標準は以下の Web サイトからダウンロードできます。
http://www.etsi.org

XML

Extensible Markup Language（XML）の詳細については以下の Web サイトをご覧ください。
http://www.w3.org/XML

137

161

349

417

455

461

531

533

605

第章：

621

635

653

667

695

703

711

799

825

865

877

885

907

925

933

963

991

1027

1087

1129

1139

1175

1231

1249

1263

1305

1307

第章：

1403

1409

1441

第章：

1451

1455

1463

1485

1517

1531

第章：

1551

1559

1571

1581

1601

1727

1765

1767

1797

1903

1955

1989

2041

2053

2061

2071

2081

2105

2119

第章：

2127

2133

2141

2147

2189

第章：

索引

Symbols

Numerics

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

あ

い

う

え

お

か

き

く

け

こ

さ

し

す

せ

そ

た

ち

つ

て

と

な

に

は

ひ

ふ

へ

ほ

ま

め

も

や

ゆ

よ

ら

り

る

れ

ろ

わ