Telelogic

Rhapsody

Code Generation Guide

Rhapsody®

Code Generation Guide

IBM®

Before using the information in this manual, be sure to read the “Notices” section of the Help or the PDF available from **Help > List of Books**.

This edition applies to Telelogic Rhapsody 7.4 and to all subsequent releases and modifications until otherwise indicated in new editions.

Contents

C Code Generation Overview

Welcome to Rhapsody! Rhapsody® is an award-winning, UML-compliant, systems design, application development, and collaboration platform. Rhapsody is used by systems engineers and software developers to deliver embedded or real-time systems. Rhapsody uniquely combines a graphical UML programming paradigm with advanced systems design and analysis capabilities and seamlessly links with the target implementation language, resulting in a complete model-driven development environment, from requirements capture through analysis, design, implementation, and test.

About this Guide

This guide covers code generation for the C language for the Rhapsody product. While it goes over in general the code generation process, its intent is to highlight the ways you can control the generated code. To do that, various properties are discussed in detail. Note that not every property available for C is mentioned.

For more general information about code generation in Rhapsody, see the "Code Generation" section of the *Rhapsody User Guide*.

For a hands-on tutorial that shows you how to create a model, generate code, and run animation to simulate the model you create, see the *C Tutorial for Rhapsody*.

Rhapsody in C

Rhapsody in C is a visual development system that facilitates efficient construction of real-time embedded applications in the C language. Unlike other C development systems, Rhapsody uses the most advanced software development techniques currently available. Commonly described as object-based, these techniques are standardized as the Unified Modeling Language™ (UML™). Although C is not an object-oriented language, Rhapsody emphasizes those aspects of object-based development that can be natively supported by the C programming language, yet offers the major benefits of object-based development: encapsulation, conciseness, and reusability.

Rhapsody is based on the major views defined by the UML for describing software systems: use case, static structure, collaborations (scenarios), and object behavior views. Rhapsody generates production-quality C code directly from several of these views.

You can generate code either for an entire configuration or for selected UML classes. Inputs to the code generator are the model and the code generation (`C_CG` and `CG`) properties. Outputs from the code generator are source files in the C language: specification files, implementation files, and makefiles.

Rhapsody in C generates full production C code for a variety of target platforms based on UML 2.0 behavioral and structural diagrams. The Rhapsody in C product also allows you to reverse engineer existing C code so that you re-use your intellectual property within a Model-Driven environment.

As of version 7.1, C code generation in Rhapsody is compliant with MISRA-C:1998. Note that there are justified violations, which are noted in the *Rhapsody User Guide*.

Rhapsody in C comes with a number of specialized C language profiles, such as FunctionalC and CGCompatibilityPre70C. The FunctionalC profile tailors Rhapsody in C for the C coder, allowing the user to functionally model an application using familiar constructs such as files, functions, call graphs, and flow charts. Use the CGCompatibilityPre70C profile to make the code generation backwards compatible with pre-7.0 Rhapsody models. For more information about the profiles provided for Rhapsody in C, refer to the *Rhapsody User Guide*.

For more information about code generation in Rhapsody in C, refer to the Customizing C Code Generation section in the *Rhapsody User Guide*.

About Properties

All Rhapsody products (in C, C++, Ada, and Java) provide you with a graphical user interface (GUI) so you can view and edit the features of an element easily. You access the properties through the **Properties** tab of the Features dialog box.

To open the Features dialog box, do one of the following in the Rhapsody browser or a diagram:

- Double-click an element (for example, **mins** [a variable])
- Right-click an element (for example, **Execution** [a diagram]), then select **Features**
- Select an element and press **Alt + Enter**
- Select an element and select **View** > **Features**

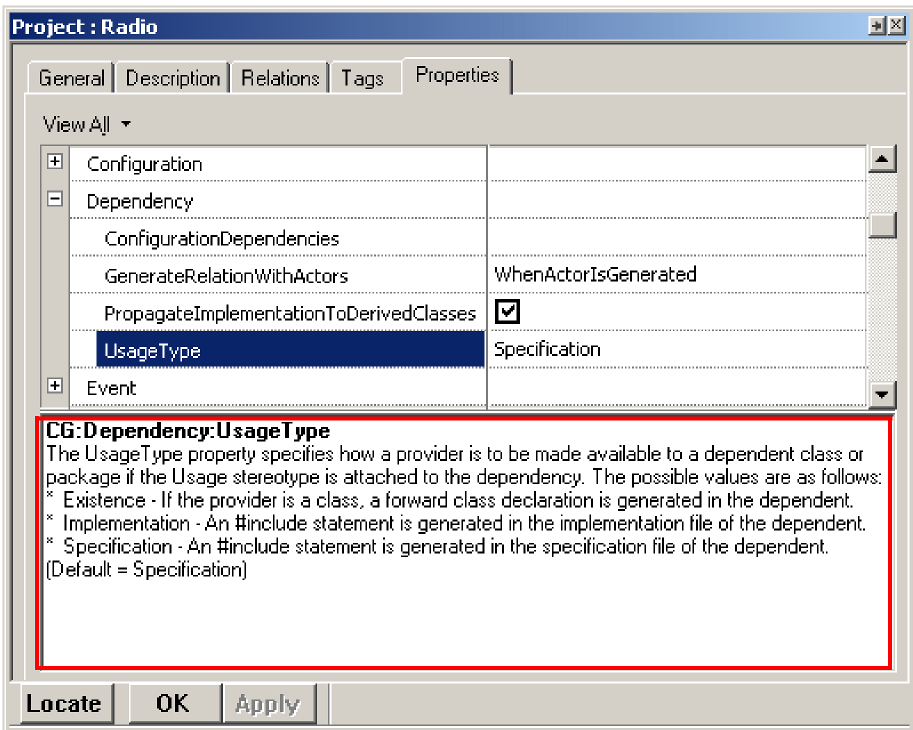
You can resize the Features dialog box and hide the tabs on this dialog box if you want. For more information about the Features dialog box, refer to the section on it in the *Rhapsody User Guide*.

Note

Once you open the Features dialog box, you can leave it open and select other elements to view their features.

The **Properties** tab lists the properties associated with the selected Rhapsody element:

- The top left column on this tab shows the metaclass and property (for example, **Dependency** and **UsageType**).
- The top right column shows the default for the selected property, if there is one (for example, **Specification**).
- The box at the bottom portion of the **Properties** tab shows the definition for the property selected in the upper left column of the tab. The definition display shows the names of the subject, metaclass, property, and the definition for the property, as shown in the following figure:



Note

Rhapsody documentation use a notation method with double colons to identify the location of a specific property. For example, for the property in the above figure, the location is `CG::Dependency::UsageType` where `CG` is the subject, `Dependency` is the metaclass, and `UsageType` is the property.

Dynamic Model-Code Associativity

Dynamic model-code associativity (DMCA) means that changes made to the model are reflected in the code and changes made to the code can be easily roundtripped back into the model. In this way, Rhapsody maintains tight relationships and traceability between the model and the code. There is no overhead from virtual machines or complex architectures. As you will see in the next sections, the code is simply another straightforward view of the model.

Dynamic model-code associativity is applicable to all versions of Rhapsody (meaning, Rhapsody in C, C++, Java, and Ada).

Special Features of Rhapsody Code

Rhapsody-generated code provides implementations of ANSI-compliant C code from design diagrams. It is possible to include and link Rhapsody-generated C code in any C++ system, with the appropriate wrapper. For example:

```
#ifdef _cplusplus
extern “C” {
#endif
/* wrapped C code */
#ifdef _cplusplus
}
#endif
```

Rhapsody-generated code supports static memory allocation where dynamic memory management is not required, and for dynamic memory allocation based on the user configuration.

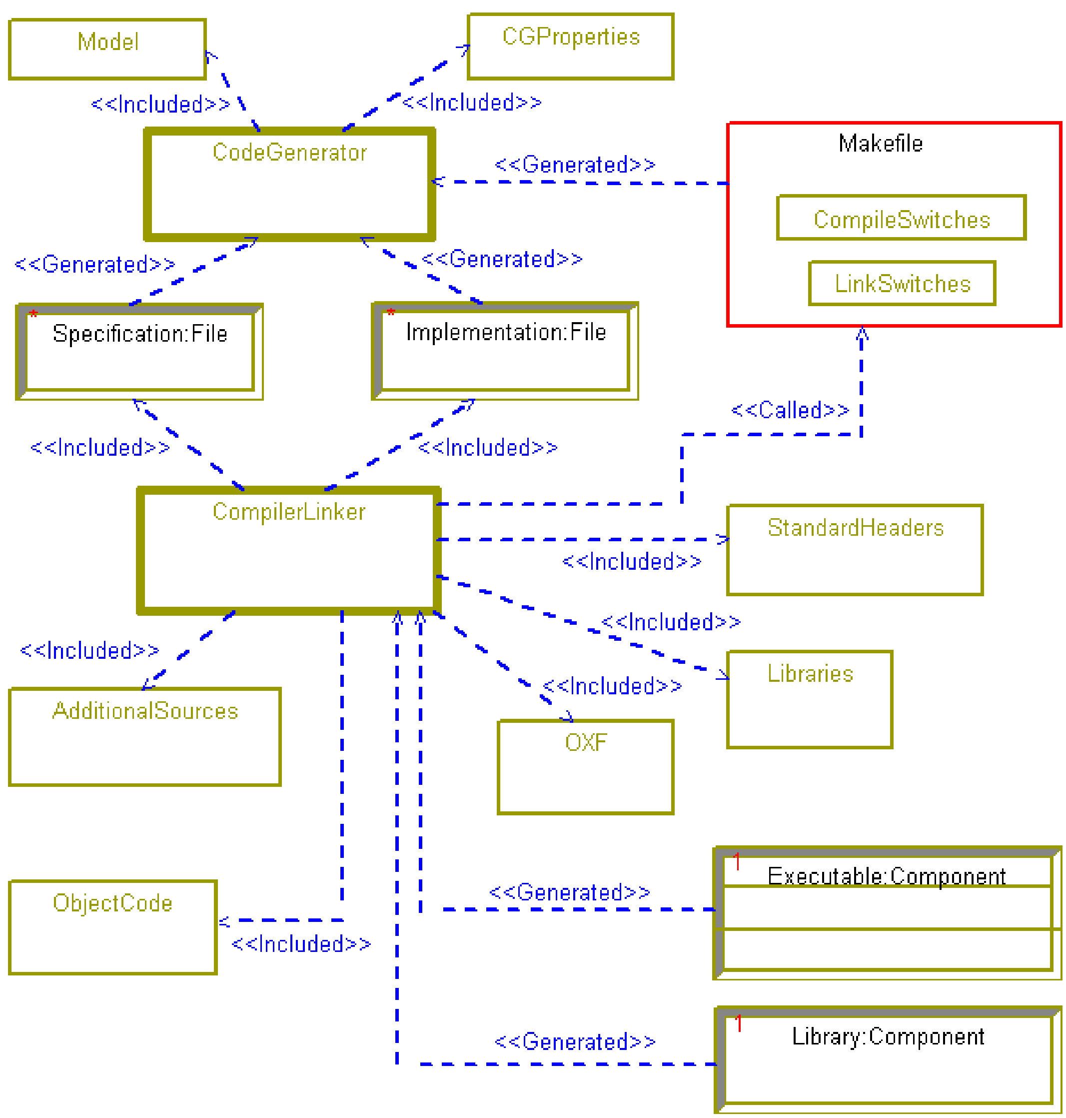
In addition, Rhapsody has features for managing source files, such as viewing, error handling, and roundtripping, which provide full associativity between the code and your model.

Code Generation Fundamentals

Inputs to the code generator are:

- Component, structural, and behavioral models built in Rhapsody
- Code generation properties set in Rhapsody

Outputs from the code generator are source files in the C language: specification files, implementation files, and makefiles. In turn, these files are used as inputs to the compiler and linker in later phases of the build process.



This object model diagram shows the elements involved in generating code, making, and finally building a component in Rhapsody. The dependency arrows indicate which files are generated and which files are included by the code generator and compiler, respectively. The thick borders around the code generator and compiler indicate that these are active objects. The executable component generated by the compiler is also an active object.

Constructive Versus Non-Constructive Views

Rhapsody generates code from the component, structural, and behavioral model elements you create using various design views:

- The *component model* consists of the components, configurations, files, and folders to which you map various modeling constructs via the browser.
- The *structural model* consists of a static view of the system created using object model diagrams (OMDs).
- The *behavioral model* consists of the life-cycle behavior of the system as defined in statecharts (SCs).

Object model diagrams and statecharts are considered *constructive* because Rhapsody generates code from them. Structural Model describes the code generated from OMDs; Behavioral Model describes the code generated from SCs.

Sequence diagrams (SDs) are only partially constructive. Rhapsody creates objects and operations from the instances and messages that you draw in them. However, the bodies of operations must be defined in the browser or a statechart.

Use case diagrams (UCDs) an activity diagrams are considered non-constructive because Rhapsody does not generate code from them. They help you analyze the system based on requirements and are useful for documentation purposes.

Structural Model

One of the major issues with object-based techniques is how to capture the logical structure of the system. Real-time systems have a static nature such that their underlying instance structure exists once the system starts because we do not want to dynamically allocate and free memory during run time. Therefore, the static structure is an instance structure rather than a class structure, the primary view in most non-real-time object-oriented (OO) systems. In Rhapsody, therefore, the instance (or *object*) is the prime concept.

The structural model consists of the objects in the system and the static relationships that exist between them. Groups of objects can be partitioned into packages or subsystems. Object model diagrams (OMDs) define the structural model. This section describes the code generated from OMDs.

Constructing Systems from Objects

Object-based modeling is applying the most fundamental engineering discipline of system construction used by system, mechanical, and hardware engineers. In other engineering disciplines, physical systems are represented as collections of parts (think of a mechanical or electrical drawing). Each part (which itself may be a collection of parts) has its own purpose and data. Early software design techniques did not follow this approach. Rather, they used *functional decomposition* because early programming languages were built around how the computers work, not how systems work.

At its core, each model is a decomposition of the system into modular, cohesive units with well-defined interfaces. Many object have an internal state that controls its behavior. Objects can be linked together (collaborate) to perform a certain task. Composite objects are constructed from simpler objects via hierarchical composition, where the composite object (or *aggregate*) owns its subobjects (or *components*). This theme follows the intuitive structure of any type of system assembly, from mechanical to electrical to software.

Re-use of services is achieved through instantiation of objects, aggregation, and delegation. *Instantiation* is the language mechanism that replicates an object type into a new object instance. By aggregating an instance of a certain component, a composite object re-uses the services provided by the component object.

Objects

Objects are the structural building blocks of a system. They form a cohesive unit of state (data) and services (behavior). Every object has a specification part (public) and an implementation part (private).

In terms of C programming, an object is implemented as a set of data members packed in a `struct`, and a set of related operations. With multiple instances, an object's data is replicated for each occurrence of the object.

For example, the following structure definition is generated in the specification file for an object `A`:

```
struct A_t {
    /* data members of A */
};
/* operations of A */
```

Some details of the implementation may differ for special types of objects (for example, see **Singleton Objects**).

Note

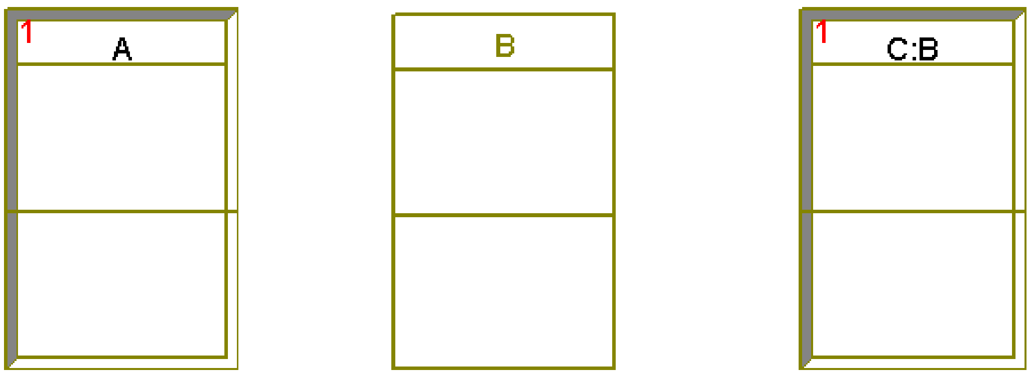
Because C structures cannot be empty, if the object has neither data nor a statechart, an `RIC_EMPTY_STRUCT` member is added as a placeholder to satisfy the C compiler. `RIC_EMPTY_STRUCT` is a macro defined in the Rhapsody in C framework.

Specifying the Type of an Object

Objects can be of either *implicit* or *explicit* type:

- Objects of implicit type are associated with their own structure.
- Objects of explicit type are defined in terms of another object type and its structure.

In the following figure, `A` is an object of implicit type, `B` is an object type, and `C` is an object that is explicitly of type `B`.



Objects of Implicit Type

Objects of implicit type are simple objects that cannot be re-used for defining other objects. Implicit types facilitate instance-based modeling. This is different from pure OO modeling, which requires every structural entity to be an instance of an existing type. This is known as the type/instance dichotomy in OO systems.

For objects of implicit type, a C structure is generated with the name of the object and the suffix "`_t`." A type is not defined for the object. For example, a C structure named `A_t` is generated in the specification file for an object `A` that is of implicit type. This object has one attribute named `att1`, which is generated as a data member of the structure, as follows:

```
struct A_t {
    /***    User-explicit entries    ***/
    int att1;/*## attribute att1 */
};
```

The object is instantiated and memory is actually allocated in the specification file for the package to which the object belongs. For example, the following statements are generated in the specification file for the `Default` package, to which the object `A` belongs:

```
struct A_t;
extern struct A_t A;
```

The first statement is a declaration of the structure `A_t`; the second is the actual definition and memory allocation for an instance `A` of `struct A_t`.

Note

The `extern` keyword indicates that `A` is declared here but defined (once) elsewhere. Any code following such a declaration can refer to `A`. If the same `extern` statement appears in different files, all of these statements refer to the same `A`.

Rhapsody automatically generates operations to handle object creation, initialization, cleanup, and destruction. These operations are analogous to what are known as constructors and destructors in C++. For example, the following operations are automatically generated for `A`:

- `A_Create()` (see **Object Creator**)
- `A_Init()`
- `A_Cleanup()`
- `A_Destroy()` (see **Object Destructor**)

Note that `Create()` and `Destroy()` operations are not generated for singletons. See **Singleton Objects** for more information.

Object Types

Object types support re-use, multiple instantiation, and dynamic instantiation. In essence, object types are abstract data types (ADTs). They specify a template of an object that can be instantiated in different contexts.

Object types are generated into C structures with their own type definitions in the specification file for the object. The type definition introduces a type alias to the `struct` representing the object. The type name consists of the name of the object type, without any suffix. For example, the following structure and type definition are generated for an object type `B`:

```
typedef struct B B;
struct B {
    /* data members of B */
};
/* operations of B */
```

Because `B` is an explicit type, other objects can be defined in terms of `B`. Both specification files and implementation files are generated for object types. Creation, initialization, cleanup, and destruction operations are all automatically generated for object types.

The type `B` is declared in the specification file for the package that owns `B`, but memory is not allocated for `B` until an object of type `B` is actually instantiated.

Object types can be instantiated either statically upon initialization of the system, or dynamically during execution (the default is dynamically). Therefore, instances of object types might have a different life span than the system. See **Dynamic Memory Allocation** for more information.

Objects of Explicit Type

Objects of explicit type are instances of an object type. Instances of object types obtain their structure and behavior from the object type.

Neither specification files nor implementation files are generated for objects of explicit type. Rather, an external declaration is generated in the specification file for the package to which the object belongs. For example, the following declaration is generated for an object `C` of type `B` in the specification file for the package that owns `C`:

```
struct B;
extern struct B C;
```

Multiplicity of Objects

Objects have a multiplicity that determines whether they should be implemented as a single object, an array, a list, a collection, or a map. You can modify the default implementation using the `CG::Relation::Implementation` property for the object.

Note that the `Implementation` property is under the metaclass `Relation` rather than `Class` because even objects without any visible relations have at least one relation to an object type that is hidden in the browser.

Bounded Multiplicity

Objects with bounded multiplicity (for example, 2) are allocated to an array with the same number of elements as the multiplicity. For example, for an object `B` of implicit type with a multiplicity of 2, the following array is allocated:

```
extern struct B_t B[2];
```

Unbounded Multiplicity

Objects with a multiplicity of * (unbounded) are allocated to an `RiCList` structure. For example, for an object `A` with a multiplicity of *, the following structure is allocated:

```
extern RiCList A;
```

`RiCList` is a predefined list container type provided by the Rhapsody in C framework.

Unspecified or Single Multiplicity

Objects for which no multiplicity is specified have a default multiplicity of one. Single objects are allocated to a simple structure. For example:

```
struct A_t {
    /* User explicit entries */
} A;
```

In this case, a single object `A` is allocated at the close of the `A_t struct` definition in the specification file for `A`.

Descriptions

Text entered in the **Description** field of the Features dialog box for an object is generated into a comment that appears in the specification file for the package that owns the object, *not* in the specification file for the object itself. The comment is generated immediately before the structure allocation for the object. For example, if you enter the description “A is an object of implicit type” in the Features dialog box for an object `A` that belongs to the `Default` package, the following comment line appears before the structure allocation line in the `Default.h` file:

```
/* A is an object of implicit type */
extern struct A_t A;
```

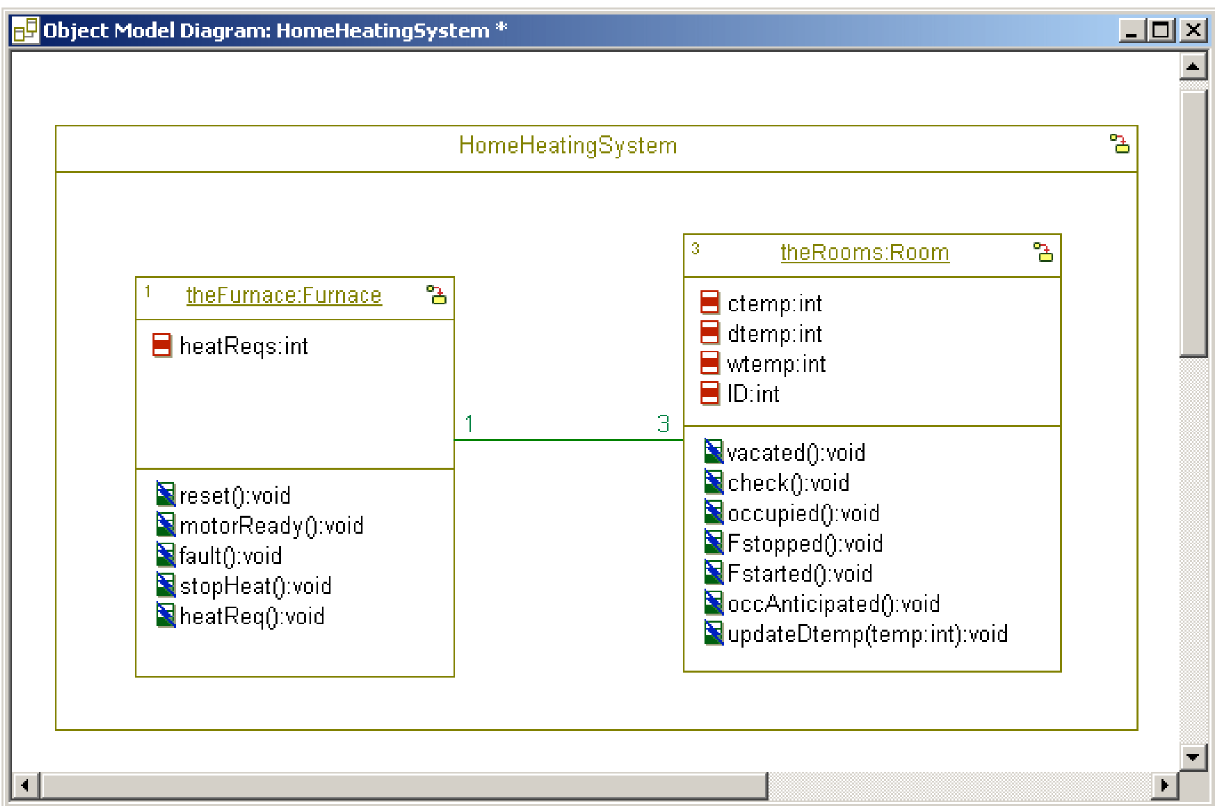
Object Interfaces

Objects provide interfaces and require interfaces. The provided interfaces are the object's signals (events and triggered operations) and services (functions). The required interfaces are realized through a set of associations and dependencies to other objects through which the object collaborates with the other objects.

The following figure shows the OMD from the home heating system (hhs) sample. It shows the provided interfaces of two objects:

- `theFurnace`—`reset()`, `motorReady()`, `fault()`, and `stopHeat()`
- `theRoom`—`vacated()`, `check()`, `occupied()`, `Fstopped()`, and `Fstarted()`

In addition, it shows the required interfaces of the two objects through the symmetric association drawn between them.



Operations

Operations are services that can be performed by an object upon request. Operations can be synchronous (such as procedure calls) or asynchronous (such as event receptions). Event receptions are special operations that process events once received from an event queue. The sending object does not wait for the processing to be completed, but “sends and forgets” the message. Typically, statecharts handle events and behavioral code scripts (written in C) define operations. However, statecharts can also handle synchronous operations, called triggered operations, and conversely, code scripts can be specified to handle events.

Implementing Operations in C

Because the C language does not directly support the concept of object operations, there are two issues that must be addressed when generating C code from object models:

- Associating an operation with a particular object
- Naming operations such that the C flat global namespace is not overloaded and contentions are avoided

Associating an Operation with an Object

Because each operation associated with an object is implemented as a global function in C, it must be provided with a context in the form of a pointer to the object on which it should operate. In C++, this context is provided in the form of an implied `this` pointer as the first argument. In C, however, the `this` pointer is not available. Therefore, in Rhapsody in C, the first argument to operations is generally a pointer to the object associated with the operation. This context pointer is conventionally called `me`. For example:

```
/*## operation close() */
void Valve_close(Valve* const me);
```

Because there is only one instance of a singleton object, the context pointer is not needed for singleton operations. See **Singleton Objects** for more information.

You can change the name generated for the first argument using the `C_CG::Operation::Me` and `C_CG::Operation::MeDeclType` properties. The `Me` property specifies the string used for the first argument (for example, “me”). The `MeDeclType` property specifies the full type declaration for the first argument. Its default value is as follows:

```
$objectName* const
```

The `objectName` variable is replaced with the name of the object type. Adding a `:i` switch to the `objectName` variable truncates the name to leave only the uppercase letters. For example, using `$objectName:i` for an object named `HomeHeatingSystem` results in the name `HHS`.

Rhapsody automatically inserts the `me` argument into code generated for operations, but it is important for you to remember to provide it when calling an operation of an object.

Naming of Operations

Because C has a flat namespace for functions, Rhapsody uses a naming convention to resolve namespace contentions. The convention used is to prefix each (public) operation with the name of the object on which it should operate. (See **Visibility of Operations** for information on different naming conventions for private operations.)

For example, the `Valve` object has two public operations: `open()` and `close()`. These operations are implemented as follows:

```
void Valve_open(struct Valve_t * const me);
void Valve_close(struct Valve_t * const me);
```

Visibility of Operations

Operations can be public or private. Private operations are those used by an object for its own internal affairs and are not part of the interface of the object. Public operations are services that the object exposes for consumption by other objects. These comprise the contract of the object and should remain stable throughout the lifecycle of the system to avoid ripple effects throughout the system. Changes to private operations (and attributes) should not impact the rest of the system.

Declarations and definitions for public and private operations can be generated in either the specification or implementation file for an object, depending on the visibility of the operation.

Note

Events and triggered operations are always public.

Operation names have different default formats, depending on whether the operation is public or private:

- Public operation names have the format `<object>_<opname>()`.
- Private operation names have the format `<opname>()`.

You can change the default format of operation names using the following properties:

- The `C_CG::Operation::PublicName` property specifies the pattern used to generate names of public operations in C.
- The `C_CG::Operation::ProtectedName` property specifies the pattern used to generate names of private operations in C.

Public Operations

Public operations are part of the object's interface. Declarations of public operations are generated in the specification file for the object, after the object's `struct` declaration. Definitions of public operations are generated in the implementation file for the object.

For example, the following declaration is generated in the specification file for the `Valve` object for the public operation `open()`:

```
/*## operation open() */
void Valve_open(Valve* const me);
```

The following definition is generated in the implementation file for the `Valve` object for the public operation `open()`:

```
void Valve_open(Valve* const me) {
    NOTIFY_OPERATION(me, &me, NULL, Valve, Valve_open, Valve_open(),
           0, Default_Valve_open_SERIALIZE);
    /*#[ operation open() */
    /*#]*/

}
```

Note that the NOTIFY_OPERATION macro is used for animation. It notifies the animator that a new operation has been called. The NOTIFY_OPERATION macro is only inserted into the code when animation is enabled.

To control the way names are generated for public operations, use the `C_CG::Operation::PublicName` property. The default value of this property, `$objectName_$opName`, prefixes the name of the operation with the name of the object. For example, the public operation to open the valve in the heating system is named `Valve_open()`.

Use the `:I` switch after `$objectName` (for example, `$objectName:I` or `$objectName:i`) to expand `$objectName` to be only uppercase letters (and digits) of the object name.

Private Operations

Private operations are not exported. Both their declaration and definition are generated in the implementation file for the object. The declarations of all of an object's private operations are grouped at the beginning of the implementation file, followed by the definitions of all the private operations.

Private operations are tagged as `static`, which allows them to be accessed by other operations in the same file.

For example, the following declaration is generated in the forward declarations section of the implementation file for the `Valve` object if the `close()` operation is made private:

```
/* Forward declaration of protected methods */
/*## operation close() *\
static void close(Valve* const me);
```

The definition of the private operation is generated later in the same file, in the methods implementation section:

```
/* Methods implementation */
static void close(Valve* const me) {
   NOTIFY_OPERATION(me, &me, NULL, Valve, close, close(),
          0, Default_Valve_close_SERIALIZE);
   /*#[ operation close() */
   /*#]*/

}
```

You can control the way names for private operations are generated using the `C_CG::Operation::ProtectedName` property. The default value of this property, `$opName`, uses the user-assigned name for the private operation, such as `myName()`.

Constructors and Destructors

Rhapsody automatically generates operations to create, initialize, clean up, and destroy objects. Object constructors include creators and initializers; object destructors include cleanup and destroy operations.

Object Creator

The object creation operation creates an object and calls its initializer. Its name has the format `<object>_Create()`.

The creator allocates memory for an object, calls the object's initializer, and returns a pointer to the object created.

For example, the following is the creator generated for the object A:

```
A * A_Create() {
    A* me = (A *) malloc(sizeof(A));
    if(me!=NULL)
            {
                    A_Init(me);
            }
    return me;
}
```

For reactive objects, a pointer to a task is added to the (end of the) creator's argument list. This pointer tells the reactive object which thread (task) it is running on. For example:

```
A * A_Create(RiCTask * p_task) {
    A* me = (A *) malloc(sizeof(A));
    if(me!=NULL)
            {
                    A_Init(me, p_task);
            }
    DYNAMICALLY_ALLOCATED(me);
    return me;
}
```

Because in C it is not possible to give an argument a default value, you can pass a value of NULL for the task to cause the instance to run in the main task.

The `C_CG::Class::AllocateMemory` property and the `C_CG::Event::AllocateMemory` property specify the string generated to allocate memory dynamically for objects or events. This string is used in the Create() operation. The default value of this property is:

```
($cname*) malloc(sizeof($cname));
```

In generated code, the `$cname` keyword is replaced with the name of the object or event for which memory is being allocated.

Dynamic Memory Allocation

You can create an object dynamically by calling its creator function. For example:

```
B *new_B;
new_B = B_create();
```

You can delete an object dynamically by calling its delete function. For example:

```
B_Destroy(new_B);
```

Object Initializer

The initialization function initializes the attributes and links of an instance. The initializer assumes that memory has previously been allocated for the object (either statically or dynamically). The object initializer name has the format `<object>_Init()`.

For example, the following is the prototype of the initializer generated for the object `A`:

```
void A_Init(struct A_t* const me);
```

The first argument is a constant pointer to the object being initialized. The `const` keyword defines a constant pointer in ANSI C. Passing a constant pointer as an argument allows the operation to change the value of the object that the pointer addresses, but not the address that the argument `me` contains.

The object initializer has the following responsibilities, which it performs in the following order:

1. Calls subobject initializer functions, if the object has subobjects.
2. Sets links for association relations.
3. Executes user code entered for the body of a constructor. This code should include initializations of the object's data.
4. Initializes aggregated framework objects (for example, `RiCTask`, `RiCReactive`, and `RiCMonitor` objects).

Subobject initialization includes calling the initializers for each subobject of a composite object. In the case of arrays, the initialization of each subobject can include the `$index` keyword.

By default, the initializer has no arguments (other than the `me` argument). If you create an initializer with arguments, you can enter initial values for the arguments in the Object dialog box. Rhapsody generates initialization code for initializers with arguments from the values entered in the Object dialog box.

Initializing Subobjects

Compositions are initialized with a call to `initRelations()` in the initializer of the parent. For example, the following initializer is generated for an object `D` that has a subobject `E`:

```
void D_Init(D* const me) {
    initRelations(me);
}
```

The `initRelations()` call in D's initializer calls the initializer for `E`:

```
        static void initRelations(D* const me) {
            E_Init(&me->E);
        }
```

If subobjects are implemented as an array (for example, because the subobject has a numeric multiplicity greater than one), the subobjects are initialized using a `while()` loop in the `initRelations()` operation. For example, if `E`'s multiplicity is two, `E` is implemented as a two-element array inside `D`. The following `while()` loop is generated in `D`'s `initRelations()` operation to initialize both instances of `E`:

```
static void initRelations(D* const me) {
    E_Init(&(me->E));
    {
            RhpInteger iter = 0;
            while (iter < 5){
                    E_Init(&((me->itsE)[iter]));
                    iter++;
            }
    }
}
```

Setting Links

If related objects are not components of a composite object, you can have the main program instantiate one of the objects by selecting it as an initial instance (in the Initialization tab for the configuration). In that object's initializer, you can create the related object explicitly and then set the link to it. For example, if an object A and an object B are related and the main() function instantiates A as an initial instance, then in the body of A's initializer you can write the following code to set its link to B:

```
B *itsB = B_Create();
A_setItsB(me, itsB);
```

Setting a link to a to-many relation involves calling the initializer for the container. In the following code, the call to RiCCollection_Init() sets the Furnace's link to three itsRooms. Passing a value of RiCTRUE to RiCCollection_setFixedSize() says that the collection is of fixed size:

```
void Furnace_Init(Furnace* const me, RiCTask * p_task) {
    RiCReactive_init(&me->ric_reactive, (void*)me,
        p_task, &Furnace_reactiveVtbl);
    RiCCollection_Init(&me->itsRoom, 3);
    NOTIFY_REACTIVE_CONSTRUCTOR(me, NULL, Furnace,
        Furnace, Furnace(), 0, Furnace_SERIALIZE);
    {
        RiCCollection_setFixedSize(&me->itsRoom,
            RiCTRUE);
    }
    initStatechart(me);
    NOTIFY_END_CONSTRUCTOR(me);
}
```

The NOTIFY_CONSTRUCTOR() and NOTIFY_END_CONSTRUCTOR() calls are instrumentation macros generated when animation is enabled. The first macro notifies the animator when the initializer has been called and creates an animation instance. The second macro notifies the animator when the initializer is about to exit.

Executing User Initialization Code

User code entered for the constructor should include initializations of the attributes of the object. You can specify the actual value for every parameter in the object constructor. The actual value will be inserted verbatim as uninterpreted text.

User code is generated between the /*#[and /*#] symbols in the code. For example, you could enter the following code in the **Implementation** field for the initializer:

```
RiCString temp;
RiCString_Init(&temp, "Hello World");
A_print(me, temp);
```

This code is implemented as follows:

```
void A_Init(struct A_t* const me) {
    NOTIFY_CONSTRUCTOR(me, NULL, A, A, A(), 0,
        A_SERIALIZE);
    me->itsB = NULL;
    {
        /*#[ operation A() */
        RiCString temp;
        RiCString_Init(&temp, "Hello World");
        A_print(me, temp);
        /*#]*/
    }
    NOTIFY_END_CONSTRUCTOR(me);
}
```

Object Cleanup

The object cleanup operation performs complementary operations to the initializer, releasing the object's links to other objects in reverse order.

Object Destructor

The destruction operation destroys an object. Its name has the format `<object>_Destroy()`.

The `Destroy()` operation calls the object's `Cleanup()` operation to clean up its links, then frees any memory allocated for the object.

For example, the following is the `Destroy()` operation generated for the object `A`:

```
void A_Destroy(A* const me) {
    if(me!=NULL)
            {
                    A_Cleanup(me);
            }
    free(me);
}
```

The `C_CG::Class::FreeMemory` property and the `C_CG::Event::FreeMemory` property specify the string generated to free memory previously allocated for objects or events. This string is used in the Destroy() operation. The default value of this property is:

```
free($meName);
```

In generated code, the `$meName` keyword is replaced with the name of the object or event for which memory is being freed.

Primitive Operations

In addition to the operations that Rhapsody automatically generates, you can define your own operations for objects. Each operation has a name and return type, and might include arguments. User-defined operations are called *primitive operations* in Rhapsody.

Object operations (as opposed to functions or global operations) are mapped to C functions with the same return type. The first argument generated for an operation is a pointer to the specific object on which the operation is to operate. Following the `me` pointer is the operation's original list of arguments, as specified in the model.

For example, the following is the prototype generated for an operation named `print()` of object type `B`:

```
void B_print(B* const me);
```

The function prototype is generated in the specification file for `B`. The only argument is a pointer to an object of type `B` called `me`.

Enter the following lines in the implementation for `B`'s `print()` operation in the model:

```
char *str;
str = "This is B";
printf("%s\n", str);
```

The following lines are added to the body of `print()` in the implementation file:

```
void B_print(B* const me) {
    NOTIFY_OPERATION(me, NULL, B, print, print(), 0,
        print_SERIALIZE);
    {
        /*#[ operation print() */
        char *str;
        str = "This is B";
        printf("%s\n", str);
        /*#]*/
    }
}
```

You can manually edit the operation between the `/*#[` and `/*#]` symbols. Roundtrip your changes back into the model by selecting **Code** > **Roundtrip** > **<configuration name>**.

A `SERIALIZE` macro is generated for operations (for example, `print_SERIALIZE`) if animation is enabled and the operation has no arguments that need to be animated. The `SERIALIZE` macro is used to display the operation during instrumentation. A `SERIALIZE` macro is not generated for inline operations.

Inline Operations

The `C_CG::Operation::Inline` property enables you to generate primitive operations and global functions as macros. The macro is defined in the specification file of the owner object. The operation call is replaced inline with the uninterpreted text specified for the macro during preprocessing.

Only primitive operations and global functions for which the `Inline` property is set to `in_header` can be generated as macros. The `Inline` property does not work for constructors or destructors. There is no instrumentation for inline operations.

The macro is defined in the specification file of the owner object as follows:

```
#define OperationName(ArgumentList) \
```

The operation's return type and argument types are ignored. Each generated line of the macro ends with " \". Curly braces ("{" and "}") are not generated around user code. This enables you to write short macros that return a value. The following is an example of such a macro definition:

```
(#define isEqual(arg1, arg2) (arg1)==(arg2))
```

If a macro is roundtripped, a backslash (" \") is added at the end of the line. The next code generation adds a second backslash " \ \", which will cause compilation errors. The extra backslash must be removed manually.

Error highlighting shows the line of the calling operation (macro call).

The following is an example of the code generated for a primitive operation `op()` of an object `A`. The operation's `Inline` property is set to `in_header`. The macro definition is generated in `A`'s specification file (`A.h`). This operation calls the global function `Global_F()`, which can also be generated inline, before exiting:

```
#define A_op(me)                \
    /*#[ operation op() */      \
    int i;                      \
    for(i = 0; i < 3; i++) {    \
        printf("LOOP\n");       \
    }                           \
    Global_F();                 \
/*#] */
```

Constant Operations

Constant operations cannot change the data on which they operate.

The `me` parameter of a constant operation points to a structure that is tagged as `const`. In this case, the `const` keyword comes before the data type specifier in the argument list. For example, the following is the generated code of a constant operation called `check()` that can access, but not change, the contents of `B`:

```
void B_check(const B* const me) {
    /*#[ operation check() */
    /*#]*/
}
```

Event Receptions

Events provide an asynchronous means of communication between objects. Both reactive objects and tasks can receive events. Events can trigger transitions in statecharts.

Adding an event reception to an object defines the object's ability to receive that kind of event. A comment is added to the specification file of an object to indicate that it can consume a particular kind of event. For example, if an object type `G` can receive an event `ev1`, the following comment is added to `G`'s specification file.

```
/*** Events consumed ***/
/* ev1();*/
```

All events are handled through a common interface found in `RiCReactive`.

The event is actually defined in the package file.

Triggered Operations

A *triggered operation* is a synchronous event that can return a value. It is a synchronous communication between objects that can be invoked by one object to trigger a state transition in another object. The body of the triggered operation is executed as a result of the transition. Because a triggered operation is synchronous, the sending object must wait for it to return before the sender can continue on its own thread.

The body of a triggered operation is set in the statechart of the receiving object by the action language associated with a transition. Thus, the body of the same triggered operation can be different based on the state of the object when the operation is invoked. To return a value from a triggered operation, use the `RiCREPLY(VALUE)` macro as one of the action statements associated with the triggered operation. See **Predefined Actions** for more information on the `REPLY` macro.

Invoking Operations

To invoke operations on objects, use standard C function calls in the following format:

```
<opname>(<object*>, p1,..,pn);
```

The first argument to the function must be a pointer to the object that is the target of the operation. For example, if `itsServer` is a pointer to an object that has an operation `start()`, invoke this operation with:

```
Server_start(itsServer, p1, p2);
```

If the object that is the target of the operation is a singleton, you can omit the context pointer as the first argument of the function, as follows:

```
Singleton_start(p1, p2);
```

Attributes

Attributes are variables that an object encapsulates to maintain its state. Objects encapsulate attributes as a set of data items. A data item designates a variable with a name and a type, where the type is a data type. A data item for an object is mapped to a member of the object's structure. The member's name and type are the same as those of the object data.

For example, the `isClosed` attribute of the `Valve` object type is embedded by value as a data member inside the `Valve` structure:

```
struct Valve {
    /*** User explicit entries ***/
    RiCBoolean isClosed;   /*## attribute isClosed ##*/
};
```

The `RiCBoolean` type is the C equivalent of `OMBoolean`, a Boolean data type defined in the Rhapsody in C++ framework.

An accessor operation enables you to access the data, whereas a mutator operation enables you to modify the data. The accessor is generated if the `C_CG::Attribute::AccessorGenerate` property is set to `Checked`. Similarly, the mutator is generated if the `C_CG::Attribute::MutatorGenerate` property is set to `Always`. The default `AccessorGenerate` is `Cleared`. The default for `MutatorGenerate` is `Never`.

Accessor and mutator operations are generated in the user implicit entries area of the specification file for the object type. For example, prototypes for the `_getIsClosed()` accessor operation and the `_setIsClosed()` mutator operation are generated for the `isClosed` attribute in the `Valve.h` file:

```
/*** User implicit entries ***/
RiCBoolean Valve _getIsClosed(const Valve* const me);
void Valve _setIsClosed(Valve* const me, RiCBoolean
    p_isClosed);
```

The bodies of the accessor and mutator operations are generated in the implementation file for the object type. For example, the following implementations are generated for the `_getIsClosed()` and `_setIsClosed()` operations in the `Valve.c` file:

```
/*** Methods implementation ***/
RiCBoolean Valve_getIsClosed(const Valve* const me) {
   return me->isClosed;
}
void Valve _setIsClosed(Valve* const me, RiCBoolean
    p_isClosed) {
    me->isClosed = p_isClosed;
}
```

Rhapsody generates attributes in the following order:

1. Attributes are grouped into user-defined and implicit attributes (such as relation containers).
2. The attributes in each subgroup are generated in alphabetical order.

Accessing Attributes

Attributes can be tagged as public or private. Ideally, attributes should be private to the object as part of its internal affairs. They should not be exposed as part of the object's interface. This is because attributes are an implementation issue and should not be part of the external contract of the object. In this way, the implementation can be modified to follow changing requirements without having any external impact. However, sometimes to satisfy efficiency constraints, attributes can be made public so that peer objects can access them directly.

There is no difference in the way public or private attributes are generated in C. Attributes are simply data members inside an object structure, and as such are always public.

However, when you assign public or private access to an attribute, the visibility applies to the accessor and mutator operations for the attribute, not to the attribute itself:

- Assigning public access to an attribute causes the code generator to generate public accessor and mutator operations for it.
- Assigning private access causes the code generator to generate static accessor and mutator operations for it.

Public Access

For example, the following is the accessor generated for an attribute with public access:

```
int Furnace_getHeatReqs(const Furnace* const me);
```

The `heatReqs` attribute belongs to the `Furnace` object in the home heating system sample. The prototype for the public accessor is generated in the specification file for the `Furnace`. The name of the public operation includes the name of the object that is its target, in this case `Furnace`.

The body of the public accessor is generated in the implementation file for the `Furnace`:

```
int Furnace_getHeatReqs(const Furnace* const me) {
    return me->heatReqs;
}
```

Private Access

On the other hand, the following is the accessor generated for the same attribute with private access:

```
static int getHeatReqs(const Furnace* const me);
```

The name of the static operation does not include the name of the object that is its target. Both the prototype and body for the static operation are generated in the implementation file for the `Furnace`:

```
static int getHeatReqs(const Furnace* const me) {
    return me->heatReqs;
}
```

Collaborations Between Objects

System objects collaborate by exchanging events and invoking operations. Objects can access other objects in four ways:

- **Inheritance**—Objects can inherit from one another.
- **Dependencies**—An object can directly access a global object by referencing its package namespace. A dependency from an object to a package familiarizes the object with the package namespace. See **Dependencies** for more information.
- **Composition**—Objects can access their subobjects and subobjects can access their owner objects. See **Compositions** for more information.
- **Parameters**—Objects can receive references to other objects as arguments of operations or events. This requires the definition of object types. See **Specifying the Type of an Object** for more information.
- **Links**—Objects that reside inside other objects must be accessed via a link, because they do not have a global identity. Links bind roles, which are the structural slots through which an object refers to a link. See **Links** for more information.
- **Interfaces**—An object can have an interface, which is a kind of classifier that specifies a contract consisting of a set of public services. An interface is a non-instantiable entity that is realized by a class, object, block, file, and so forth, and may be realized by any number of these entities.
- **Ports**—Objects can have ports. A *port* is a distinct interaction point between a class and its environment or between (the behavior of) a class and its internal parts.

Inheritance

Inheritance is the derivation of one class from one or more other classes. The derived class inherits the same data members and behaviors present in the parent class. It is the mechanism by which more specific elements incorporate structure and behavior of more general elements related by behavior. Inheritance is also known as generalization.

You can create inheritance by using the **Inheritance** tool for an object model diagram to draw an inheritance arrow between two classes.

Inheriting from an External Class

To inherit from a class that is not part of the model, set the `CG::Class::UseAsExternal` property for the class to `Checked`. This prevents code from being generated for the superclass.

To generate an `#include` of the class header file in the subclass, do one of the following:

- Add the external element to the scope of some component.
- Map the external element to a file in the component.
- Set the `CG::Class::FileName` property for the class to the name of its specification file (for example, `super.h`). That file is included in the source files for classes that have relations to it. If the `FileName` property is not set, no `#include` is generated.

If you need a class to import an entire package instead of a specific class, add a dependency (see **Dependencies**) with a stereotype of «`Usage`» to the external package.

Dependencies

Dependencies signify abstract links between objects. There are several types of predefined dependencies that can be tagged with stereotypes. The `Usage` stereotype is the only one that affects code generation in C. It implies a dependence on services provided by another object.

Note

The `Send` stereotype is a tag that indicates the sending of an event to another object. It has no code generation side effects.

You can also define other stereotypes for dependencies.

A dependency is different from a link. A dependency does not have any structural implications, but simply implies information that can be interpreted in several different ways. While a link has a semantic connection among multiple objects and it is an instance of an association.

The `Usage` stereotype for dependencies is constructive, in that it changes the generated code depending on the value assigned to the `CG::Dependency::UsageType` property for the dependency. The possible values for this property are as follows:

- `Specification`—An `#include` statement is generated in the specification file of the dependent.
- `Implementation`—An `#include` statement is generated in the implementation file of the dependent.
- `Existence`—A forward declaration is generated in the specification file of the dependent.

Compositions

The primary means for handling complexity in object-based systems is through object decomposition. An object can be comprised of other objects or *subobjects* (nested objects). A subobject is an object defined within a parent object. The parent object (or owner) can delegate requests to be handled by its subobjects, and the subobjects can communicate back to their parent object.

Each of the subobjects knows the `HomeHeatingSystem` as its parent, and the `HomeHeatingSystem` can access each of its subobjects by name. This view of the `HomeHeatingSystem` is called an object structure view, because it shows the internal structure of the object. The subobjects can be linked to each other or not, depending on the nature of the system.

With compositions, the parent object holds the subobjects by value rather than by pointer. The parent object is responsible for initializing and cleaning up after the subobjects. See **Initializing Subobjects** for more information.

By default, a subobject designates a single instance and is implemented as a member of the parent object's structure. The member's name and type are the same as the name and type of the subobject. In other words, subobjects are embedded by value in the parent object, rather than as pointers to objects.

When a subobject's multiplicity is specified as a number greater than one, the subobjects are implemented as an array by default. For example, `theFurnace` and `theRooms` are implemented as members of the `HomeHeatingSystem` structure. The object `theFurnace` is implemented as a single instance of type `Furnace`, and `theRooms` are implemented as an array of three `Rooms`:

```
typedef struct HomeHeatingSystem HomeHeatingSystem;
struct HomeHeatingSystem {
    RiCReactive ric_reactive;
    /***    User implicit entries    ***/
    struct Furnace theFurnace;
    struct Room theRooms[3];
};
```

If a subobject's multiplicity is not known in advance, it is implemented as a linked list. For example, if you specified multiplicity of `theRooms` as `*` rather than 3, it would be implemented as an `RiCList` as follows:

```
struct HomeHeatingSystem {
    RiCReactive ric_reactive;
    /***    User implicit entries    ***/
    struct Furnace theFurnace;
    RiCList theRooms;
};
```

You can also implement subobjects using other types of dynamic containers (such as collections). You specify how to implement concrete relations using the `CG::Relation::Implementation` property. For example, setting the `Implementation` property for `theRooms` to `UnboundedUnordered` would implement `theRooms` as an `RiCCollection` rather than as an `RiCList` or array.

The properties under the subject `RiCContainers` determine how functions are generated for various kinds of containers used to implement relations. See the definitions provided for the properties on the applicable **Properties** tab of the Features dialog box. Refer also to the *Properties Reference Manual*.

Links

An association between objects is called a *link*. An object can have links to other objects as part of its required interface. Through such links, the object can request services of or send events to another object.

Links bind roles, which are the structural slots through which an object can refer to its links. By default, a role is named `its<object>`, where `<object>` is the name of the peer on the other end of the link.

Links can be symmetric or directional. With a symmetric link, both objects know each other, implying two roles. With directional links, only one object has access to its peer via a single role. See **Symmetric Associations** and **Aggregations** for more information.

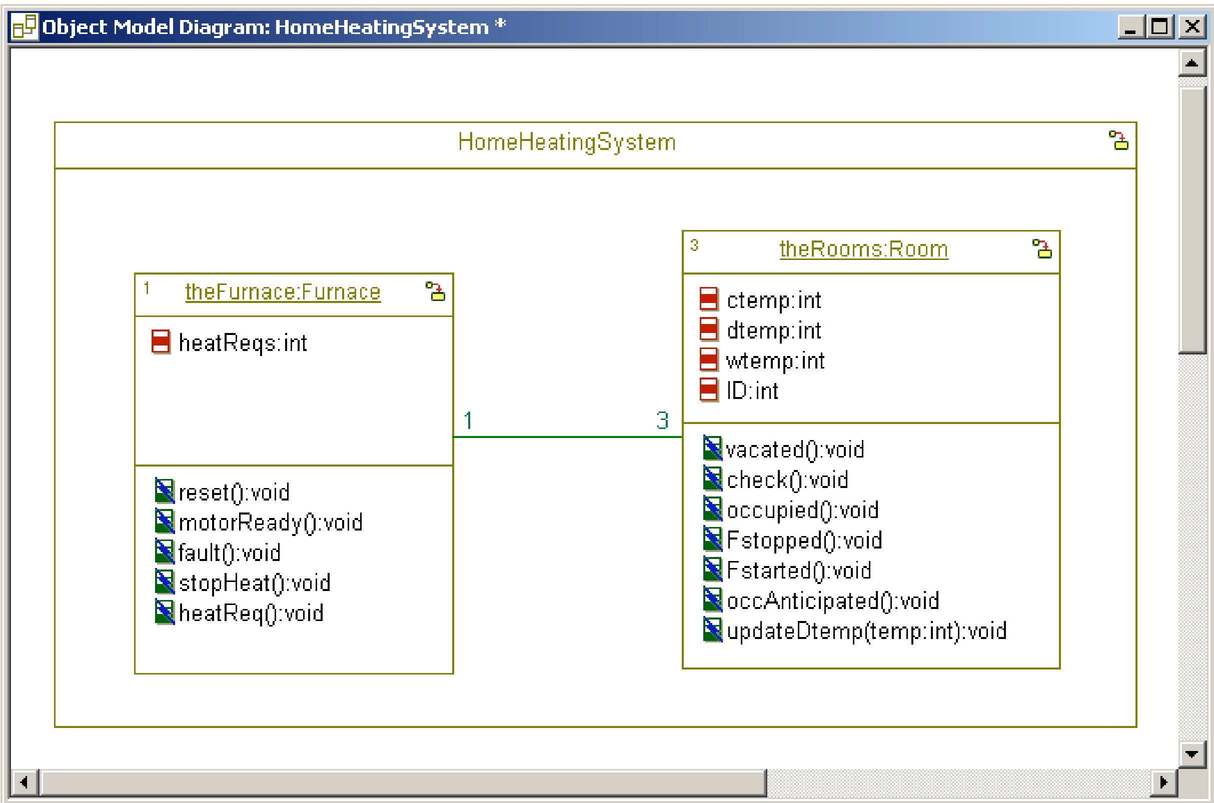
Roles have multiplicity. A multiplicity of one means that the link connects an object to only one other object. The default multiplicity is set by the `General::Relations::DefaultMultiplicity` property.

If a link connects an object to more than one other object (multiplicity greater than 1), that link is implemented by default as an array. In addition, a role can contain references in the form of pointers, facilitating access to several members within the group.

Symmetric Associations

With symmetric links, the objects on both ends of the link know each other. Thus, two roles are defined.

The sample OMD, as shown in the following figure, shows a symmetric association between `theFurnace` and `theRooms`. This is a to-many link in which one furnace services three rooms.



Roles are implemented as:

- A `struct` data member
- An accessor function
- A mutator function

Link Data Member

By default, a link is with a single instance. A link to a single instance is called *scalar.* A scalar relation is generated into a data member in the object's structure whose name is the same as the role and whose type is a pointer to the other object. For example, an `itsFurnace` member of type pointer to `Furnace` is generated as a member of the `Room` structure to represent the `Room`'s link to the `Furnace`:

```
struct Room {
    /***    User implicit entries    ***/
    struct Furnace * itsFurnace;
};
```

Link Accessor

The link accessor returns a pointer to the associated object. Its name has the format `<object>_get_<rolename>()`.

For example, the following accessor is generated for the `itsFurnace` role:

```
struct Furnace * Room_get_itsFurnace(const Room*
    const me);
```

This is the implementation of the link accessor:

```
struct Furnace * Room_get_itsFurnace(
    const Room* const me)
{
    return (struct Furnace * )me->itsFurnace;
}
```

Link Mutator

The link mutator sets a pointer to the associated object. If the link is symmetric, the mutator also sets the reciprocal link.

Up to three methods can be generated for the link mutator:

- The first is part of the object's provided interface.

  This link mutator name has the format `<object>_set<rolename>()`.
- The other two are helper functions generated only for symmetric relations that help to establish the symmetric relation without creating an infinite loop.

For example, the following mutator is generated for the `itsFurnace` role:

```
void Room_setItsFurnace(Room* const me, struct Furnace
    *p_Furnace);
```

This is the implementation of the link mutator. The link between `Furnace` and the `Room` is symmetric, so the mutator also sets the reciprocal link:

```
void Room_setItsFurnace(Room* const me, struct Furnace
    *p_Furnace) {
    if(p_Furnace != NULL)
    Furnace__addItsRoom(p_Furnace, me);
    Room__setItsFurnace(me, p_Furnace);
}
```

If the link is a symmetric relation, the first mutator calls a second that has a double underscore before the word “set” in its name:

```
void Room__setItsFurnace(Room* const me, struct Furnace
    *p_Furnace) {
    if(me->itsFurnace != NULL)
    Furnace__removeItsRoom(me->itsFurnace, me);
    Room___setItsFurnace(me, p_Furnace);
}
```

If the link is a symmetric relation, the second mutator calls a third that has a triple underscore before the word “set” in its name:

```
void Room___setItsFurnace(Room* const me,
    struct Furnace *  p_Furnace) {
    me->itsFurnace = p_Furnace;
    if(p_Furnace != NULL) {
        NOTIFY_RELATION_ITEM_ADDED(me, Room, Furnace,
            "itsFurnace", p_Furnace, FALSE, TRUE);
    }
    else
    {
        NOTIFY_RELATION_CLEARED(me, Room,"itsFurnace");
    }
}
```

Together, these three operations set the symmetric link between the `Furnace` and the `Room`.

Aggregations

Aggregation is a strong form of association that represents a part/whole relationship, as with a car (whole) that has wheels (parts). The parts can have a life of their own, and do not necessarily come into being and die with the creation and destruction of the whole (for example, the wheels can be removed and re-used on another car before the original car is junked).

Aggregations are implemented as “shared” aggregations in Rhapsody, in that a part can be simultaneously aggregated by several wholes because it is not physically embedded inside any of them. *Composition*, on the other hand, is an even stronger form of “non-shared” aggregation, in which the part is actually embedded inside the whole and comes into begin and dies with its creation and destruction.

The rules for implementing aggregations (that are not compositions) as either pointers or containers depending on the multiplicity and ordering of the relation are the same for aggregations as for associations.

To-Many Links

The implementation described in the previous sections is for a scalar link, or a link to a single object. Rhapsody implements links to more than one object, or *to-many links*, using various kinds of containers, depending on the multiplicity and ordering of the link. Types of to-many links are as follows:

- Bounded ordered
- Bounded unordered
- Embedded fixed
- Fixed
- Qualified
- Static Array
- Unbounded ordered
- Unbounded unordered
- User-specified

Appropriate accessor and mutator operations are generated for each kind of link, depending on the container used to implement it. The defaults for implementing relations are modifiable through the properties of the role.

Ordered Links

By default, ordered links to more than one object are implemented as an `RiCList`. A to-many link is made ordered by setting its `CG::Relation::Ordered` property to `Checked`. This includes relations where the multiplicity is known (bounded ordered relations) and those where the multiplicity is not known (unbounded ordered relations).

Unordered Links

By default, unordered links to more than one object are implemented as an `RiCCollection`. A to-many link is made unordered by setting its `Ordered` property to `Cleared`. This includes relations where the multiplicity is known (bounded unordered relations) and those where the multiplicity is not known (unbounded unordered relations).

Embedded Links

Links to subobjects are implemented as an embedded data member if the subobject's multiplicity is one (embedded scalar relation), or as an array if the subobject's multiplicity has a numeric value greater than one (embedded fixed relation).

For example, the `HomeHeatingSystem` object has one subobject called `itsFurnace` and three subobjects called `itsRoom`, all embedded as components. In this case, `theFurnace` has an embedded scalar relation and `theRooms` has an embedded fixed relation to `HomeHeatingSystem`. These relations are implemented as follows:

```
struct HomeHeatingSystem {
    /***   User implicit entries   ***/
    struct Furnace theFurnace;
    struct Room theRooms[3];
};
```

You can achieve the same effect by setting the `CG::Relation::Implementation` property to `Scalar` for a scalar relation or `Fixed` for a fixed relation. These types of relation implementations should be used only under two conditions:

- The related object is inside a composite object (component relation).
- The related object is embeddable (`C_CG::Class::Embeddable` is `Checked`).

Fixed Links

By default, to-many links with a fixed multiplicity are implemented as an `RiCCollection`.

Qualified Links

By default, to-many links where a qualifier is specified on the link are implemented as an `RiCMap`.

Random Access Links

A *random access link* is a relation that has been enhanced to provide random access to the items in the container. You can give a to-many link random access by setting the `C_CG::Relation::GetAt` property for the role to `Checked`. The `C_CG::Relation::GetAtGenerate` property must also be set to `Checked`. This generates an accessor for the role that uses an appropriate `getAt()` method for the container. The `$index` keyword is passed as a parameter to the `getAt()` method to access a particular element inside the container. The default value for `$index` is `int i`.

For example, the `GetAt` property for a bounded ordered relation has the following value:

```
RiCList_getAt(&$me$cname, $index)
```

Setting the `GetAt` property for `theRooms` to `Checked` causes the following accessor to be generated in the `HomeHeatingSystem` to allow it to access a particular `Room`:

```
struct Room * HomeHeatingSystem_getTheRooms(
    const HomeHeatingSystem* const me, int i) {
    return RiCList_getAt(&me->theRooms, i);
}
```

Initializing Links within Packages

An `initRelations()` operation is generated for packages to initialize the links between the objects in a package. The name of the link initialization operation has the format `<package>_initRelations()`.

For example, if the `Default` package has an object `A` of implicit type and an object `C` of type `B`, and `A` has a directional link to type `B`, a `Default_initRelations()` operation is generated in the implementation file for the `Default` package to initialize the link between `A` and `C`, the only object of type `B`:

```
static void Default_initRelations() {
    A_Init(&A);
    B_Init(&C);
}
```

This operation calls the initialization operations for `A` and `C`, which in turn initialize the links to the respective objects.

Interfaces

Interfaces are a kind of classifier that specify a contract consisting of a set of public services. An interface is a non-instantiable entity that is realized by a class, object, block, file, and so forth, and may be realized by any number of these entities.

In terms of C programming, an interface is represented by a set of global function declarations and a structure consisting of void pointers to the global virtual functions.

For example, given some class B with the global functions `read()` and `parse()`, there exists an interface I_B with the following global declarations:

```
void I_B_parse(void * const void_me);
void I_B_read(void * const void_me);
```

and a structure as follows:

```
typedef struct I_B_Vtbl{
   size_t I_B_offset;
   RiCBoolean (*I_B_gen)(void * const void_me, RiCEvent* event,
         RiCBoolean fromISR);

   void (*I_B_parse)(void * const void_me);
   void (*I_B_read)(void * const void_me);

} I_B_Vtbl;
```

Ports

A *port* is a distinct interaction point between a class and its environment or between (the behavior of) a class and its internal parts. A port allows you to specify classes that are independent of the environment in which they are embedded. The internal parts of the class can be completely isolated from the environment and vice versa.

A port can have the following interfaces:

- **Required interfaces—**Characterize the requests that can be made from the port's class (via the port) to its environment (external objects). A required interface is denoted by a socket notation.
- **Provided interfaces—**Characterize the requests that could be made from the environment to the class via the port. A provided interface is denoted by a lollipop notation.

These interfaces are specified using a *contract*, which by itself is a provided interface.

If a port is *behavioral*, the messages of the provided interface are forwarded to the owner class; if it is *non-behavioral*, the messages are sent to one of the internal parts of the class. Classes can distinguish between events of the same type if they are received from different ports.

Partial Specification of Ports

If you specify ports without any contract (for example, an implicit contract with no provided and required interfaces), Rhapsody assumes that the port relays events using the code generator. You could link two such ports and the objects would be able to exchange events via these ports.

However, Rhapsody will notify you during code generation (with warnings or informational messages) because the specification is still incomplete.

Considerations

Ports are interaction points through which objects can send or receive messages (primitive operations, triggered operations, and events).

Ports in UML have a type, which in Rhapsody is called a *contract*. A contract of a port is like a class for an object.

If a port has a contract (for example, interface `I`), the port provides I by definition. If you want the port to provide an additional interface (for example, interface `J`), then, according to UML, `I` must inherit `J` (because a port can have only one type). In the case of Rhapsody, this inheritance is created automatically once you add `J` to the list of provided interfaces (again, this is a port with an explicit contract `I`). According to the UML standard, if `I` and `J` are unrelated, you must specify a new interface to be the contract and have this interface inherit both `I` and `J`.

Implicit Port Contracts

Some found that enforcing a specification of a special interface as the port's contract to be artificial, so Rhapsody provides the notion for an *implicit contract*. This means that if the contract is implicit, you can specify a *list* of provided and required interfaces that are not related to each other, whereas the contract interface remains implicit (no need to explicitly define a special interface to be the port's contract in the model).

Working with implicit contracts has pros and cons. If the port is connected to other ports that provide and require only subsets of its provided and required interfaces, it is more natural to work with implicit contracts. However, if the port is connected to another port that is exactly "reversed" (see the check box in the port's Features dialog box) or if other ports provide and require the same set of interfaces, it makes sense to work with explicit contracts. This is similar to specifying objects separately from the classes, or objects with implicit classes in the case when only a single object of this type or class exists in the system.

Rapid Ports

Rapid ports are ports that have no provided and required interfaces (which means that the contract is implicit, because a port with an explicit contract, by definition, provides a contract interface). These ports relay any events that come through them. The notion of rapid ports is Rhapsody-specific, and enables users to do rapid prototyping using ports. This functionality is especially beneficial to users who specify behavior using statecharts—without the need to elaborate the contract at the early stages of the analysis or design.

Components-based Development in RiC

You can do component-based developed in Rhapsody in C (RiC) because there is code generation support for interfaces and ports.

A class may realize an interface, that is, provide an implementation for the set of services it specifies (that is, operations and event receptions). You use a realization relationship to indicate that a class is realizing an interface. In addition, an interface may inherit another interface, meaning that it augments the set of interfaces the superinterface specifies. You can specify interfaces, realize them, and connect to objects via the interfaces.

RiC users can take advantage of service ports that allows the passing of operations and functions via ports, in addition to passing events. You can specify ports with provided and required interfaces. In addition, Rhapsody 7.1 provides code generation support for standard UML ports in RiC and code generation of ports supports the initialization of links via ports. For more information about ports, refer to the *Rhapsody User Guide*.

In this type of development in RiC, interfaces are treated as a specification of services (that is, operations) and **not** as inheritance of data (attributes). Also, in this type of development in RiC, realization (as opposed to inheritance) is used to distinguish between realizing an interface and inheriting an interface/class.

As of Rhapsody 7.1, code generation supports realizing interfaces in C. This means interfaces and ports specified in a C model will be implemented by the code generator. This means code generation generates:

- Code for a C interface (a class with “pure virtual operations”)
  - Virtual tables with function pointers
  - Relay code from the interface to the realizing class according to the virtual table
- The “realization code” for the realizing class
  - Aggregating the interface
  - Initializing the virtual table
- Links between objects that instantiate associations to the interface

Singleton Objects

Objects with a multiplicity of one that are tagged with the `Singleton` stereotype are instantiated only once throughout the life of the system. Singleton objects are implemented in C as a `struct` and functions. The singleton property is not enforced on the data, however.

A singleton object is declared as a `struct` in the specification file. For example:

```
struct object_0_t {
    /* attributes of object_0 */
};
```

The singleton object is instantiated as a package object in the implementation file, as follows:

```
struct object_0_t object_0;
```

Because there can be only one instance of a singleton, its operations do not include a context pointer as their first argument. For example, for a singleton object `A` with an operation `op1()` with one argument `a1`, the following function prototype is generated:

```
/*## operation op1(int) */
void A_op1(int a1);
```

If the same object were not a singleton, the following function prototype would be generated:

```
/*## operation op1(int) */
void A_op1(struct A_t* const me, int a1);
```

Initializing Singletons

`Init()` and `cleanup()` operations are generated for singletons, but `create()` and `destroy()` operations are not.

If a Rhapsody model has global instances, as in the case of singletons, something must call their `init()` function. In C++, the problem is solved using default construction. In C, however, another mechanism must be found. In the case of executable components, the `main()` function can call the initializers of global objects. But with library components, the user of the library must call the initializer before using a global object.

In Rhapsody in C, the component initializer calls the `init()` operations for all packages in the component scope. In turn, the package initializer calls the `init()` operations generated for any global objects, events, and so on, within the package scope.

External Objects

External objects are objects that are generated outside of the current Rhapsody project. They could have been created in Rhapsody or some other environment. The referencing of external objects allows you, for example, to relate to external frameworks or legacy code from within a Rhapsody model. All objects, or object types, that are read-only are assumed to be external.

You can mark an object as external by setting its `CG::Class::UseAsExternal` property to `Checked`. No assumption is made regarding implicit interfaces of external objects, such as accessors or mutators. Because they might not have been generated in Rhapsody, they are assumed to be non-instrumented.

If you override the file name of an external object via the `CG::Class::FileName` property, an `#include` statement is added to the implementation file whenever the element is added to a regular object (package, dependency, relation, and so on). It is not necessary to add a file extension because Rhapsody automatically adds the extension `.h` to the file name. For example, if you set the `FileName` property of an external object `B` to `myB`, the following `#include` directive is generated in the `.c` file for the package:

```
#include "myB.h"
```

You can also override the file name of an external object by adding the file to the component model by adding the element to a file in the model.

If any other objects in the model have `Usage` dependencies to the external object, the same `#include` directive is added to the specification files of those objects. See **Dependencies** for more information.

For the model to compile, the location of the external file must be specified as either an include path or under the compiler switches at the component or configuration level (using the Settings tab of the Features dialog box for the configuration). If you added the external object to a file with the correct path, no modification of the search path is needed.

Reactive Objects

Reactive objects are objects that can receive and process events. They typically have state-based behavior that is defined in a statechart. However, an object is considered reactive if it satisfies any one of the following conditions:

- Has a statechart
- Has an event reception

If an object is reactive, an instance of an `RiCReactive` object is embedded by value in the object's structure as a data member. For example:

```
typedef struct Furnace Furnace;
struct Furnace {
    RiCReactive ric_reactive;
    /* attributes of Furnace */
};
```

Note that `RiCReactive` is an abstract data type provided by the Rhapsody in C framework to define the event-handling behavior of reactive objects.

For every reactive object, an additional `struct` is defined in the implementation file to hold pointers to functions that are defined as part of the statechart implementation. These pointers are passed to the reactive member of an object type:

```
static const RiCReactive_Vtbl Furnace_reactiveVtbl = {
   rootState_dispatchEvent,
   rootState_entDef,
   ROOT_STATE_SERIALIZE_STATES(rootState_serializeStates),
   /* Violation of MISRA Rule 45 (Required): */
          /* 'Type casting to or from pointers shall not be used.' */
          /* The following cast is justified and is */
          /* for Rhapsody auto-generated code use only. */
          (RiCObjectDestroyMethod) Furnace_Destroy,

    NULL,
    NULL,
    NULL,
    (RiCObjectCleanupMethod) Furnace_Cleanup,
    (RiCObjectFreeMethod) FreeInstance
};
```

Note the following:

- The `RiCReactive_Vtbl` virtual function table is defined in the Rhapsody in C framework (in `RiCReactive.h`).
- The framework uses the `rootState` functions to connect to the generated statechart code. These functions are as follows:
  - The `dispatchEvent()` function consumes an event.
  - The `entDef()` function starts running a statechart. It is called by the `startBehavior()` function (see **Starting Reactive Behavior**).
  - The `serializeStates()` function passes the instrumentation information to enable visual updating of states in animated statecharts.
  - The `<object>_Destroy()` function is responsible for destroying the object and is called when a termination connector is reached.

The `dispatchEvent()`, `entDef()`, and `serializeState()` functions are implemented in the handle closer files defined in the framework (`RiCHdlCls.c`). You can define functions to perform similar tasks and link them to your project through the virtual function table, if desired. However, this topic is beyond the scope of this book.

Initialization of a reactive object and the statechart that it drives are accomplished as part of the object's initialization function. For example, the following initializer for the `Furnace` object in the HomeHeatingSystem calls `RiCReactive_init()` to initialize the reactive object, then calls `initStatechart()` to initialize the object's statechart:

```
void Furnace_Init(Furnace* const me, RiCTask * p_task) {
    RiCReactive_init(&me->ric_reactive, (void*)me,
        p_task, &Furnace_reactiveVtbl);
    /* relation initialization loop */
    initStatechart(me);
}
```

The `RiCReactive_init()` and `initStatechart()` functions are defined in the Rhapsody framework.

The second parameter to the initializer, `p_task`, is a pointer to the task, with the associated event queue, from which the reactive object processes events. If the reactive object is sequential, this task is the system thread; if the reactive object is active, this task is the object's own thread. See **Active Objects** for more information.

Concurrency Objects

Rhapsody provides several types of objects for modeling timing constraints, priorities, resource management, and performance. Rhapsody also provides facilities for allocating objects to tasks, assigning priorities, and protecting shared resources.

Logically, the Rhapsody execution model is event-driven. Therefore, there is no need to use multitasking to provide the desired system services because the underlying framework handles the dispatching of events. Task allocation results from the consideration of time constraints and handling of external outputs via polling or interrupt handlers.

To handle concurrency, Rhapsody provides two categories of objects:

- Stereotyped application objects
- Primitive concurrency and synchronization objects

Stereotyped Application Objects

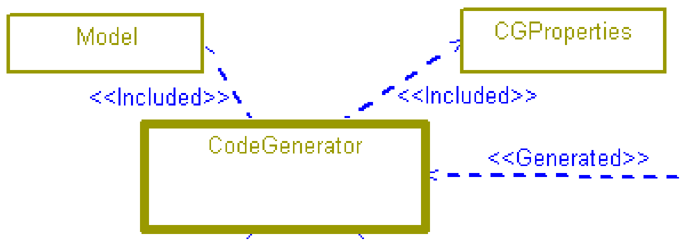
The stereotyped application objects include active objects and guarded objects (also known as protected objects, synchronized objects, or monitors).

Active Objects

Active objects are application objects that own a thread of control. Active objects have controller capabilities. Each active object owns an event queue through which it processes its incoming events. By default, subobjects share the thread (and consequently the event queue) of their parent object, unless they are also active, in which case they each own their own thread.

The counterpart to active concurrency is sequential concurrency. Sequential objects run on the system thread, allowing the system event queue to process the object’s events along with those of other sequential objects in first in, first out (FIFO) order.

Active objects are depicted similar to their sequential cousins in OMDs, but with thicker borders. In the following figure, the `CodeGenerator` is depicted as an active object with a thick border, whereas the `Model` and `CGProperties` objects are sequential and therefore have thin borders.



Rhapsody implements active objects by adding an object of a predefined type called `RiCTask` as a data member. This enables the active object to re-use the capabilities of its embedded `RiCTask` member. For example:

```
typedef struct A A;
struct A {
    RiCTask ric_task;
    /* other members of A */
};
```

Guarded Objects

Guarded objects encapsulate data shared by several active objects or tasks. They do not own their own threads, but can synchronize calls from various threads.

Operations that are protected are called *guarded operations*. A guarded operation is considered critical enough to need to enforce mutual exclusion. A guarded object is an object that owns at least one guarded operation.

One way to implement a guarded object is to give it a mutex so every operation that is explicitly set to be guarded locks the mutex at the beginning of the operation and releases it at the end.

An `RiCMonitor` member is added to the structures of guarded objects. For example:

```
typedef struct A A;
struct A {
    RiCMonitor ric_monitor;
    /* other members of A */
};
```

Note that `RiCMonitor` is a monitor type defined in the Rhapsody framework.

The `ric_monitor` subobject is used only for operations of this object that are specifically tagged as guarded. You can tag an operation as guarded using the `CG::Operation::Concurrency` property.

The guarded operation is protected inside a wrapper operation, which is responsible for the protection. The guarded operation is generated as a private operation as follows:

- The wrapper operation name is the user-assigned name for the operation `<opname>()`.
- The guarded operation name has the format `<object>_<opname>_guarded()`.

For example, two functions are generated for a guarded operation `increase()` of an object `B`:

- `B_increase()`—The wrapper operation
- `B_increase_guarded()`—The actual guarded operation

The declaration for the wrapper operation is generated in the specification file for the object:

```
int B_increase(B* const me, int i);
```

The wrapper operation, `increase()`, obtains a lock on the `ric_monitor` object, calls the guarded operation, and finally releases the lock:

```
int B_increase(B* const me, int i) {
    int wrapper_return_value;
    RIC_OPERATION_LOCK(&me->ric_monitor);
    wrapper_return_value = B_increase_guarded(me, i);
    RIC_OPERATION_FREE(&me->ric_monitor);
    return wrapper_return_value;
}
```

Once the wrapper function obtains a lock, the guarded operation is protected and can perform its critical operations without being accessed by another object until the lock is freed:

```
static int B_increase_guarded(B* const me, int i) {
    {
        /*#[ operation increase(int) */
        return i++;
        /*#]*/
    }
}
```

Similarly, the `cleanup` operation for guarded objects is generated into a wrapper operation and a guarded operation that performs the cleanup. For example, the cleanup for a guarded object `B` first locks `B`, then calls `cleanup_guarded()`, which does the actual cleanup:

```
void B_Cleanup(B* const me) {
    RIC_OPERATION_LOCK(&me->ric_monitor);
    B_Cleanup_guarded(me);
}
void B_Cleanup_guarded(B* const me) {
    RiCMonitor_cleanup(&me->ric_monitor);
}
```

You can also use the `lock` and `free` macros directly to avoid the overhead of wrapper operations.

Primitive Concurrency and Synchronization Objects

Primitive concurrency and synchronization object types are defined outside of the system and cannot be modified. They are essentially external objects that are defined in a C framework package called `OXF`. For this reason, code is not generated for them.

Among these external objects is a set of primitive object types that support concurrency and synchronization. Such services are normally provided by common real-time operating systems. The concurrency and synchronization object types include the following:

- **Task objects**—Are distinguished from active objects. With active objects, the framework is responsible for determining how the object behaves (in terms of owning its own thread, event handler, and so on). With task objects, however, you can define how you want the task to behave.

  Typical operations on task objects include the following:

  - `start()`
  - `stop()`
  - `suspend()`
  - `resume()`

  You can provide your own implementations for these operations.

- **Message queues**—Support intertask communication between active objects.
- **Semaphores**—Protect a shared resource by allowing only a limited number of objects to hold a token (lock) on a resource at a time. Both semaphores and mutexes are RTOS entities. See the *RTOS Adapter Guide* for more information.
- **Mutexes**—Provide binary mutual exclusion for a shared resource by allowing only one object to hold the token at a time.
- **Timer objects**—Provide a timing feature that permits, for example, the output of a signal at repeatable intervals.

You create any of these object types in your model by selecting the appropriate stereotype. The primitive object types typically have an iconized representation to support easier readability of diagrams.

Packages

Packages allow partitioning of the system into functional domains. You can think of a system as a single, high-level package, with everything else in the system contained in it. A package is a collection of packages, objects and object types (in C), events, diagrams, globals, types, use cases, and actors. Because packages can be nested with other packages, they enable you to partition a system into smaller subsystems. Thus, package nesting provides a way to organize large projects into package hierarchies.

Subsystems can contain objects, object types, events, diagrams, and other logical artifacts. They can also contain basic programming constructs, such as functions and data items or variables. The elements (objects, object types, and events) that belong to a package are all declared and allocated within the context of the package file.

Packages themselves do not carry direct responsibilities or behavior—they are simply containers. Packages are not instantiable and they cannot have multiple copies.

Rhapsody generates both a specification file and an implementation file for each package. The package specification file includes forward declarations of public objects.

Global Variables

Global variable definitions are included in package implementation files after instrumentation method definitions. For example, the global variable `dT` in the home heating system sample is defined in the implementation file for the `Default` package as follows:

```
int dT;   /*## attribute dT */
```

Note

When animation is enabled, the `serializeGlobalVars()` method serializes the global variables in the model by converting them to strings so they can be displayed during instrumentation.

Instrumenting a Package

The `OM_INSTRUMENT_PACKAGE()` macro instruments the package. The third argument, `<package>_instrumentVtbl`, references a virtual function table associated with animation objects. The virtual function table allows you to create your own framework and connect it to Rhapsody.

Package Constructors and Destructors

The `<package>_OMInitializer_Init()` operation initializes the events in a package. For example, if the `Default` package contains an event `evCheck`, the package initialization operation is defined in the implementation file for the package as follows:

```
void Default_OMInitializer_Init() {
    ARC_INIT_EVENT(evCheck);
}
```

The `<package>_OMInitializer_Cleanup()` operation cleans up links between global objects when the package is destroyed if the `CG::Class::DeleteGlobalInstance` property is set for the objects.

Files

Rhapsody in C enables you to create model elements that represent files. A *file* is a graphical representation of a specification (`.h`) or implementation (`.c`) source file. This new model element enables you to use functional modeling and take advantage of the capabilities of Rhapsody (modeling, execution, code generation, and reverse engineering), without radically changing the existing files.

Note

Files are not the same as the file functionality in components that existed in previous versions of Rhapsody. To differentiate between the two, the new file is called `File in Package` and the old file element is called `File in Component`. A `File in Component` includes only references to primary model elements (package, class, object, and block) and shows their mapping to physical files during code generation.

A file element can include variables, functions, dependencies, types, parts, aggregate classes, and other model elements. However, nested files **are not** allowed.

Rhapsody supports the following modeling behavior for files:

- You can drag files onto object model diagrams and structure diagrams.
- If you use the FunctionalC profile, then the **File** tool is available on the **Drawing** toolbars for object model diagrams and structure diagrams.
- You can drag files onto a sequence diagram, or realize instance lines as files.
- A file can have a statechart or activity diagram.
- Files are implicit and always have a multiplicity of 1.
- Files are listed in the component scope and the initialization tree of a configuration. They have influence in the initialization tree only in the case of a **Derived** scope.
- Files can be defined as separate units, and can have configuration management performed on them.
- Files can be owned by packages only.

Generating Code for Files

During code generation, files produce full production code, including behavioral code. In terms of their modeling properties, modeled files are similar to implicit singleton objects.

Note the following:

- For an active or a reactive file, Rhapsody generates a public, implicit object (singleton) that uses the active or reactive functionality. The name of the singleton is the name of the file.

  Note: The singleton instance is defined in the implementation source file, not in the package source file.

- For a variable with a **Constant** modifier, Rhapsody generates a `#define` statement. For example:

  ```
  #define MAX 66
  ```

FunctionalC Profile and the File Diagram

With Rhapsody in C you can use the **FunctionalC** profile. This profile tailors Rhapsody in C for the C coder to allow you to functionally model an application using familiar constructs such as files, functions, call graphs, and flow charts.

When you use the FunctionalC profile, you can draw file diagrams, which show how files interact with one another. Typically, file diagrams show how the `#include` structure is created. File diagrams provide a graphical representation of the system structure. The Rhapsody code generator directly translates the elements and relationships modeled in file diagrams into C source code.

For a hands-on tutorial that shows you how to create a model that uses a file diagram, generate code, and run animation to simulate the model, see the *C Tutorial for Rhapsody*.

Data Types

Rhapsody provides a set of predefined data types, which you can use for defining variables, attributes of objects, and arguments to functions. You can also define your own types.

Primitive Data Types

The predefined types are defined in the `PredefinedTypesC` package (the `PredefinedTypesC.sbs` file in the `Share\Properties` directory).

Predefined types include:

- char
- char*
- double
- float
- int
- long
- long double
- short
- unsigned char
- unsigned long
- unsigned short
- void
- void *
- RiCBoolean
- RiCString
- OMString

`RiCBoolean` is a Boolean data type defined in the framework (in `RiCTypes.h`) as follows:

```
typedef unsigned char RiCBoolean;
```

`RiCString` is a string data type that is defined in the framework (in `RiCString.h`) as follows:

```
typedef struct RiCString {
    unsigned int size;  /* The current allocated size */
    unsigned int count; /* The number of characters in
                           the string (without \0) */
    char * string;      /* the string */
} RiCString;
```

The `RiCString` type has a number of operations for creating, destroying, and manipulating strings.

`OMString` is a string data type that is defined in the Rhapsody in C++ framework (in `omstring.h`). The `OMString` type provides compatibility with models created in Rhapsody in C++.

User-Defined Data Types

User-defined data types include data types that can be either enumerations or compositions of primitive data types, such as arrays, structures, or unions.

Types are generated in the specification file for the package. For example, a type `myType` could have the following declaration:

```
typedef char * myType
```

This type definition is generated verbatim in the package specification file, after the forward declarations of objects and object types:

```
typedef char * myType;
```

A semicolon is automatically appended to the line, so you do not have to include it in the declaration.

You can control the order in which types are generated in code using the Edit Type Order feature of the package. Refer to the *Rhapsody User Guide* for more information.

Structure of Generated Files

This section describes the structure of Rhapsody-generated specification (`.h`) and implementation (`.c`) files, including the main sections within each of the files. Subsequent sections provide details on how individual modeling constructs within the constructive design diagrams map to code.

Annotations

The generated source code is generously commented with annotations and, if instrumented, with instrumentation macros. *Annotations* are comment lines starting with a comment symbol and two pound signs (`/*##`). For example,

```
/*## package Default */
```

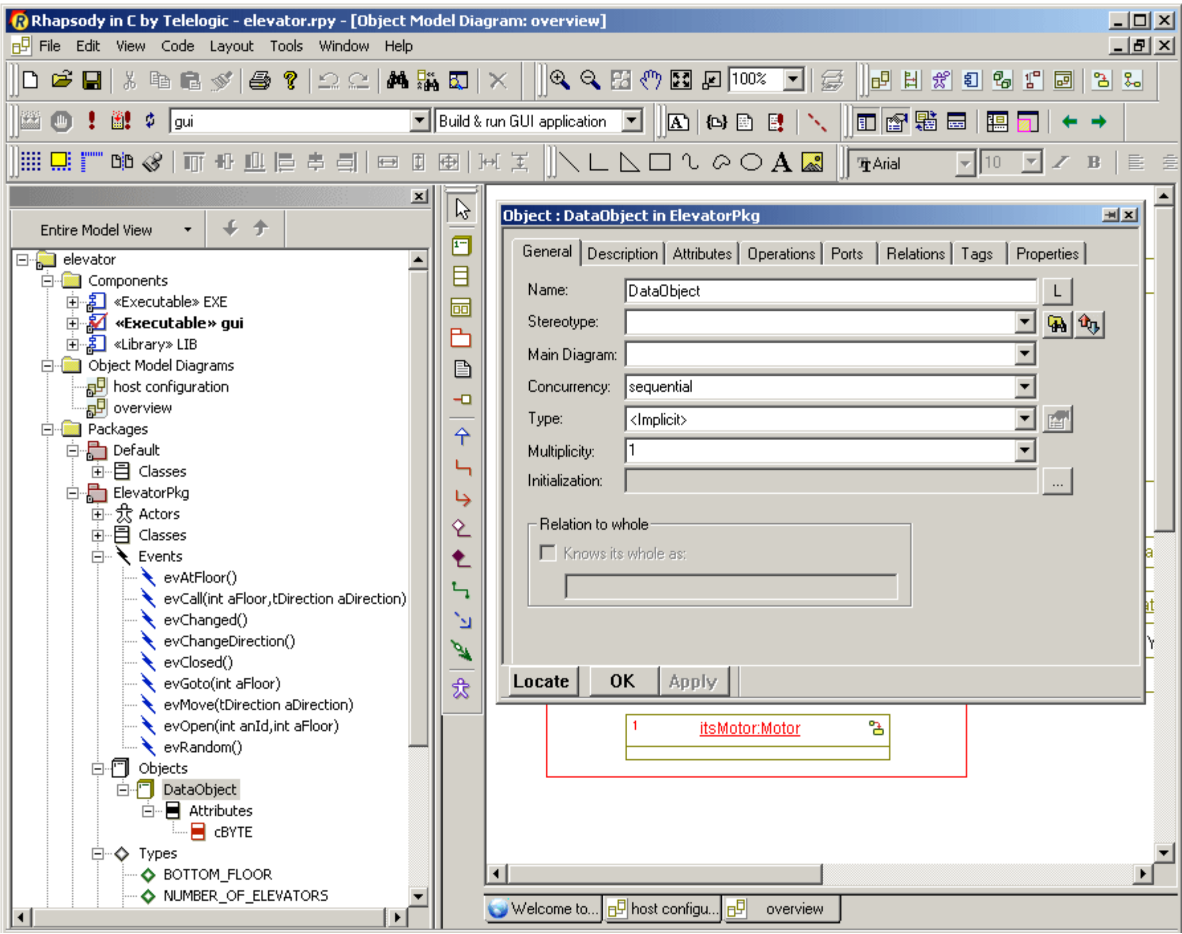
Annotations demarcate new sections in the code and therefore play an important role in tracing between design constructs and the corresponding code.

Note

Annotations are used for roundtrip and error highlighting. Do not edit or remove annotations. Doing so will hinder tracing between the model and the code and might interfere with Rhapsody's ability to animate your model.

Specification Files

When Rhapsody generates code for your project, it groups the code into predefined sections so you can easily follow it. The prolog section of the specification file can begin with a multiline header that includes the name of the generated file. The following figure shows the DataObject in the Elevator sample expanded in the Browser:



A specification file is divided into the following sections:

1. File header
2. Preprocessor directives
3. Structure declarations
4. Method declarations
5. File footer

File Header

The C_CG::File::SpecificationHeader property specifies the multiline header to be generated at the beginning of specification files.The default content for the SpecificationHeader property in C is as follows:

```
/*****************************************************
    Rhapsody in C       : $RhapsodyVersion
    Login               : $Login
    Component           : $ComponentName
    Configuration       : $ConfigurationName
    Model Element       : $FullModelElementName
//! Generated Date      : $CodeGeneratedDate
    File Path           : $FullCodeGeneratedFileName
*****************************************************/
```

Header format strings can contain any of the following keywords:

- $ProjectName for the project name.
- $ComponentName for the component name (for example, HelloWorld).
- $ConfigurationName for the configuration name (for example, HelloWorld).
- $ModelElementName for the name of the element mapped to the file. If there is more than one, this is the name of the first element.
- $FullModelElementName for the name of the element mapped to the file (for example, Default), including the full path. If there is more than one, this is the name of the first element.
- $CodeGeneratedDate for the generation date.
- $CodeGeneratedTime for the generation time.
- $RhapsodyVersion for the version of Rhapsody that generated the file (for example, 7.1).
- $Login for the user who generated the file.
- $CodeGeneratedFileName for the name of the generated file.
- $FullCodeGeneratedFileName for the full file name (for example, HelloWorld\Default.h).
- $Tag for the value of the specified element’s tag.
- $Property for the value of the element property with the specified name.

To avoid redundant compilation, Rhapsody avoids unnecessary changes to specific lines prefixed with a special string, defined by the `C_CG::File::DiffDelimiter` property. The default `DiffDelimiter` value is `//!`. The keywords are resolved in the following order:

- Predefined keywords (such as $`Name`)
- Property keywords
- Tag keywords

Note the following:

- Keyword names can be written in parentheses. For example:

  ```
  $(Name)
  ```

- If the value of a keyword is a MultiLine, each new line (except the first one) starts with the value of the `C_CG::Configuration::DescriptionBeginLine` property; each line ends with the value of the `C_CG::Configuration::DescriptionEndLine` property.

Preprocessor Directives

The preprocessor directives section of the file includes the following information:

- Element symbol check
- Include files
- Event symbols

Element Symbol Check

The `#ifndef` and `#endif` preprocessor directives check whether a symbol is defined for the element being specified. If the symbol is not already defined for the element, Rhapsody defines one. For example, a `Display_H` symbol is defined for the `Display` package.

A matching `#endif` is generated at the end of the specification file.

Include Files

The file lists the necessary include files for the project, including the appropriate framework (`oxf`) header file for the language. For example, for the Ada language, the following header file is included:

```
#include <oxf/Ric.h>
```

This file is located in the `Share\C\oxf` directory for Ada framework files. The `Ric.h` file defines certain tracer and animation symbols and includes the remaining C framework files, which provide predefined behaviors for real-time constructs such as events, event and message queues, tasks, and timers.

To specify additional include directives for header files, use the `C_CG::Class::SpecIncludes` property.

For example, if the element has dependencies to reference packages or other modules that are not part of the Rhapsody design, add the necessary include files to this property.

Event Symbols

If the element being specified is a package, it defines symbols for the events in the package.

The event symbol name has the following format:

```
<event>_<package>_id <ID number>
```

Each event has an ID number, starting with one. Event ID numbers increment based on the order in which events were added to the model during design time. They have nothing to do with the order in which events are displayed in the browser, which is generally alphabetical.

For example, if the `Foobar` package contained an `evStart` event, the following event symbol would be defined:

```
#define evStart_Foobar_id 1
```

Structure Declarations

The structure declaration section allocates memory for object types and events that belong to the package. Rhapsody names objects according to their type:

- **Implicit**—The name of an implicit object has the format `<object>`. For objects of implicit type, Rhapsody generates a C structure with the name of the object and a suffix of `_t`, which implies that the object is itself a type. For example, `Display_t`.
- **Explicit**—The name of an explicit object has the format `<object>:<object type>`.

For example, the file `Default.h` includes the following structure declarations:

```
struct Display_t;
extern struct Display_t Display;
```

Note that event structures are defined in the specification file for the package that owns the event.

If the `Default` package contains an object type A and an event `evStart`, the following structures are allocated in this section:

```
struct A;
struct evStart;
```

The A structure is defined in the specification file for A (`A.h`); the event structure is defined in the specification file for the package that owns the event.

For more information on implicit and explicit types, see **Structural Model**.

Method Declarations

The next section of the file includes declarations of methods (constructors and destructors) for packages, objects, relations, and events.

Package Methods

Two methods (operations) are generated to initialize memory when an element is created and clean up memory when the element is destroyed.

For example, the following initializer and cleanup methods are generated for the `Default` package:

```
void Default_OMInitializer_Init();
void Default_OMInitializer_Cleanup();
```

Relation Methods

Rhapsody generates a constructor to initialize relations between elements within a package. The relation initializer name has the format `<package>_initRelations()`.

For example, the following method initializes relations between the objects in the `Default` package:

```
static void Default_initRelations();
```

Applying the keyword `static` to the method allows it to be accessed by other operations in the same file.

Event Methods

Rhapsody generates the following constructors and destructors to deal with events:

- `RiC_Create_<event>()`—Creates an event. This constructor returns a pointer to the newly created event.
- `RiC_Destroy_<event>()`—Destroys an event. This destructor receives a pointer to the event that will be destroyed.
- `<event>_Init()`—Initializes memory when an event is created. The constructor points to the memory address to be allocated.
- `<event>_Cleanup()`—Cleans up memory when an event is destroyed. The destructor points to the memory address to be deallocated.

For example, Rhapsody generates the following methods for `evStart` events:

```
evStart * RiC_Create_evStart();
void RiC_Destroy_evStart(evStart* const me);
void evStart_Init(evStart* const me);
void evStart_Cleanup(evStart* const me);
```

File Footer

The specification file ends with a footer whose content is determined by the `C_CG::File::SpecificationFooter` property. The following is the default content for the `SpecificationFooter` property for C:

```
/*****************************************************
  File Path: $FullCodeGeneratedFileName
*****************************************************/
```

The variable `FullCodeGeneratedFileName` is replaced with the name of the specification file. You can change the generated footer by modifying the `SpecificationFooter` property. Footer format strings can contain any of the following keywords:

- `$ProjectName` for the project name.
- `$ComponentName` for the component name.
- `$ConfigurationName` for the configuration name.
- `$ModelElementName` for the name of the element mapped to the file. If there is more than one, this is the name of the first element.
- `$FullModelElementName` for the name of the element mapped to the file, including the full path. If there is more than one, this is the name of the first element.
- `$CodeGeneratedDate` for the generation date.
- `$CodeGeneratedTime` for the generation time.
- `$RhapsodyVersion` for the version of Rhapsody that generated the file.

- `$Login` for the user who generated the file.
- `$CodeGeneratedFileName` for the name of the generated file.
- `$FullCodeGeneratedFileName` for the full file name.
- `$Tag` for the value of the specified element’s tag.
- `$Property` for the value of the element property with the specified name.

To avoid redundant compilation, Rhapsody avoids unnecessary changes to specific lines prefixed with a special string, defined by the `C_CG::File::DiffDelimiter` property. The default `DiffDelimiter` value is `//!`. The keywords are resolved in the following order:

- Predefined keywords (such as `$Name`)
- Property keywords
- Tag keywords

Note the following:

- Keyword names can be written in parentheses. For example:

```
$(Name)
```

- If the value of a keyword is a MultiLine, each new line (except the first one) starts with the value of the `C_CG::Configuration::DescriptionBeginLine` property; each line ends with the value of the `C_CG::Configuration::DescriptionEndLine` property.

Implementation Files

The implementation (`.c`) file contains implementations of operations (methods) whose prototypes are defined in the specification file. For example, when you run the HelloWorld sample, one of the generated files is `Default.c`.

An implementation file is divided into the following sections:

1. File header
2. Preprocessor directives
3. Global variables
4. Method implementations
5. File footer

File Header

As with specification files, implementation files begin with a multiline header. The C_CG::File::ImplementationHeader property determines the content of the header. By default, the value of this property is the same as the SpecificationHeader property.

Preprocessor Directives

The next section of the file lists the specification files of related packages, objects, and object types. For example, the Default.h file includes the following files:

```
#include <oxf/RiCTask.h>
#include "Display.h"
```

To include additional files, use the C_CG::Class::ImpIncludes property.

Global Variables

The next section of the implementation file defines global variables and methods for serializing global variables for instrumentation. If the implementation file is for a package, this section also defines methods to initialize the events in the package, and to clean up memory when the package is destroyed.

Method Implementations

The next section of the implementation file implements the bodies of both user-defined (explicit) and automatically generated (implicit) methods.

File Footer

As with specification files, the implementation file ends with a multiline footer whose content is determined by the C_CG::File::ImplementationFooter property. By default, the value of this property is the same as the SpecificationFooter property.

Component Model

The component model consists of the components, configurations, folders, and files to which you can map various design constructs of the software model.

Components

Components are binary-level entities that are the end result of compilation. Libraries (`.lib` files) and executables (`.exe` files) are the final output of the build process with the source files generated by Rhapsody.

In the browser, you can specify the name and location of the final component. You can also specify which elements to map to a component, the locations of any include files, and which libraries, additional sources, and standard headers to link in during compilation.

If the component is an executable, Rhapsody generates a specification file and an implementation file for it called `Main<component>.h` and `Main<component>.c`, respectively. These files are named for the active component. For example, if the active component is called `DefaultComponent` and it is an executable, the names for its source files are `MainDefaultComponent.h` and `MainDefaultComponent.c`. If the component is a library, the files are named simply `<component>.h` and `<component>.c` (without the "Main" prefix).

The component specification (`.h`) file declares the component and its initializer and cleanup methods. For example:

```
/***********************************************
.
.
.
#ifndef MainDefaultComponent_H
#define MainDefaultComponent_H
/*---------------------------------------------*/
/* MainDefaultComponent.h                      */
/*---------------------------------------------*/

/* Constructors and destructors:*/

void DefaultComponent_Init();
void DefaultComponent_Cleanup();

#endif

/***********************************************
   File Path: DefaultComponent\DefaultConfig\
MainDefaultComponent.h
***********************************************/
```

The component implementation (`.c`) file contains the main program loop. For example:

```
.
.
.
#include "MainDefaultComponent.h"
#include <oxf/Ric.h>
#include "Default.h"

/*---------------------------------------------*/
/* MainDefaultComponent.c                      */
/*---------------------------------------------*/

void DefaultComponent_Init() {
    Default_OMInitializer_Init();
}

void DefaultComponent_Cleanup() {
    Default_OMInitializer_Cleanup();
}

int main(int  argc, char*  argv[]) {
   if(RiCOXFInit(argc, argv, 6423, "", 0, 0)) {
          DefaultComponent_Init();
          {
                 /*#[ configuration
                 DefaultComponent\DefaultConfig */
                 /* your code goes here */;
                 /*#]*/
          }
          RiCOXFStart(FALSE);
          DefaultComponent_Cleanup();
          return 0;
   }
   else
          return 1;
   }

/**********************************************
   File Path: DefaultComponent\DefaultConfig\
MainDefaultComponent.c
**********************************************/
```

The component specification file includes the `Ric.h` file, in which the real-time framework for Rhapsody in C is defined.

The main program loop calls `RiCOXFInit()`, one of the functions provided by the framework. This framework initialization function performs the following operations:

- Initializes the event dispatcher
- Sets the port number and host name for instrumentation
- Initializes the tick timer
- Creates the main task
- Creates a breakpoint manager
- Takes the first step in the main task
- Takes any operating-specific actions that need to be taken after the environment is set

If `RiCOXFInit()` returns successfully, the `main()` function then executes any initialization code entered in the Initialization tab for the configuration. The `main()` function then calls the function to initialize the component (for example, `DefaultComponent_Init()`). This function in turn calls the functions to initialize any packages contained in the component.

Once the component is initialized, the `main()` function calls the `RiCOXFStart()` function, which starts the main task. By default, the generated code passes a parameter value of `FALSE` to the `OXFStart()` function. This means that the system should not fork a new task and the model should run on the main system thread.

If you are creating a GUI application and the compiled component is a library that should not interfere with the main program thread, you should pass a value of `TRUE` to `RiCOXFStart()`, thus preventing the library from taking control of the system.

Together, the `RiCOXFInit()` and `RiCOXFStart()` functions start the Rhapsody model running. They must be called before your model can start receiving events. If the component is a library that will be linked into another application (for example, a GUI application), Rhapsody does not generate a `main()` function for it. You must write the code to call these two functions, first `RiCOXFInit()` and then `RiCOXFStart()`, somewhere in the main program loop for the application to start the event processing.

Note that if your animation port number is set to any number other than the default of 6423 in your `rhapsody.ini` file, you must pass the correct port number as the third parameter to `RiCOXFInit()`.

For example, in the home heating system sample, the program entry point for the GUI application (`hhsproto` component) is defined in the `hhsprdlg.cpp` file with the following call:

```
RiCOXFInit(NULL, NULL, 6423, ””, 0, 0);
```

The third argument to `RiCOXFInit()`, 6423, is the default animation port number. If your animation port is set to a different number, you can edit this argument to match the one in use (for example, 6424). Otherwise, animation will not work.

Note

All global instances must be created before `OXFInit()` and `OXFStart()` are called. Otherwise, the application will crash.

When the last event has been processed and the model has reached a termination point, the `main()` function calls the function to clean up the component (for example, `DefaultComponent_Cleanup()`).

Component source files are generated to the configuration directory, which is under the component directory by default. For example:

```
<project_dir>\<component_dir>\<config_dir>
```

Configurations

Configurations define various flavors of a component. For example, by defining several configurations you can generate different versions of the same component for various target environments, with instrumentation enabled or disabled, and in debug or release versions.

Rhapsody generates a specification (`.h`) file, an implementation (`.c`) file, and a make (`.mak`) file for each configuration of a component. These source files are all generated to the configuration directory by default.

Folders

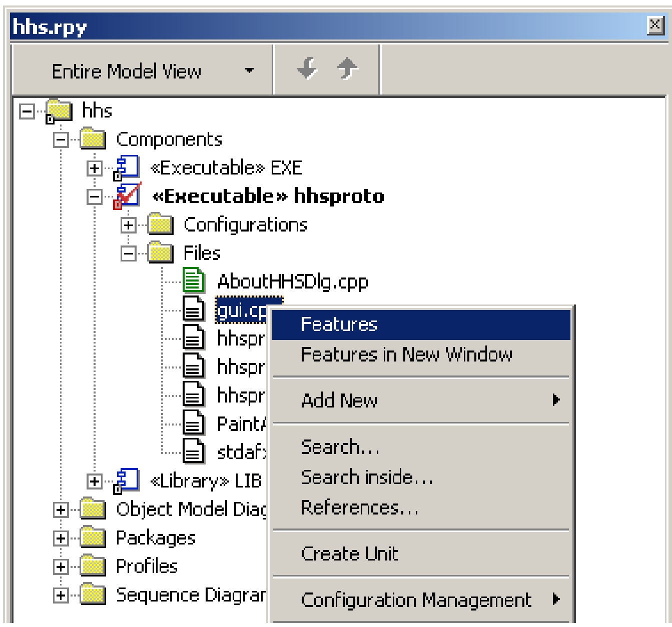
Rhapsody creates a folder with the name of the component, and under this folder is another one with the name of the active configuration. By default, generated files are mapped to the configuration folder. To map files to different folders, you can add folders to a component in the browser, and then map elements to those folders.

Files

By default, Rhapsody in C generates a specification (`.h`) file and an implementation (`.c`) file for each design element. These files have the same name as the element they represent, with different extensions. However, you can override the default file mappings and map packages and classes to files with user-specified names. In addition, you can map package files to the component file.

To add an element to a file, do the following:

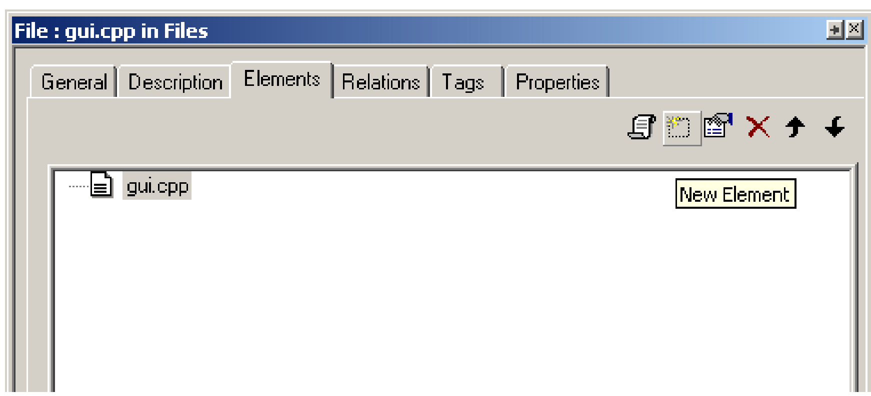
1. In the browser, right-click a file and select **Features** from the pop-up menu, as shown in the following figure. The Features dialog box opens.



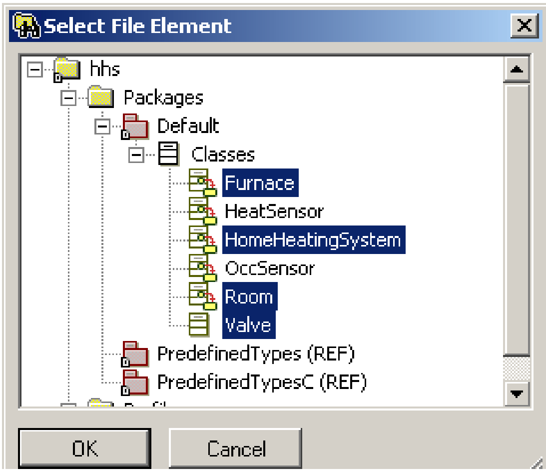
2. On the **General** tab, specify the type of file that should be generated for the elements you plan to add:

   - **Logical** generates a specification file and an implementation file containing both declaration and definition for the mapped elements. This is the default.
   - **Specification** generates only a specification file containing declaration or definition according to the mapping. Typically, a specification file includes declarations.
   - **Implementation** generates only an implementation file containing declaration or definition according to the mapping. Typically, an implementation file includes declarations.
   - **Other** generates a specification file and an implementation file, or just a specification file, or just an implementation from an included external file in a build.

3. On the **Elements** tab, as shown in the following figure, click the New Element button . The Select File Element dialog box opens.

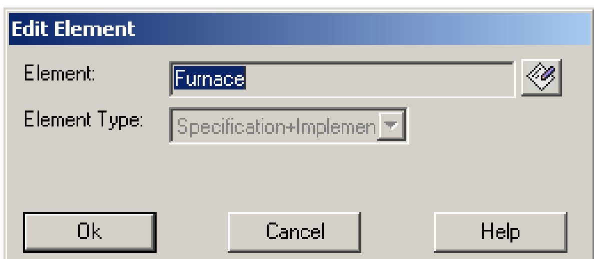


4. In the Select File Element dialog box, select the element(s) you want to add to the file, as shown in the following figure, then click **OK**.

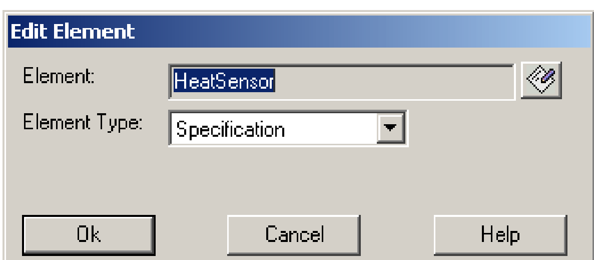


5. If you selected **Other** as your type of file or just want to see which element type is associated with a file, double-click the element on the **Elements** tab.

- If you selected **Logical** as your file of type (on the **General** tab), all your elements are set with **Specification+Implementation** as the element type by default and the **Element Type** box is disabled, as shown in the following figure.

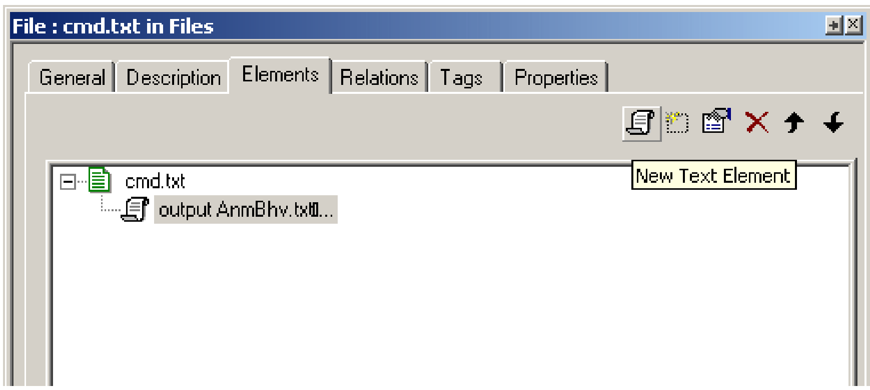


- If you selected **Specification**, all your elements are set with that element type. You can change this setting for an element if you want, as shown in the following figure.

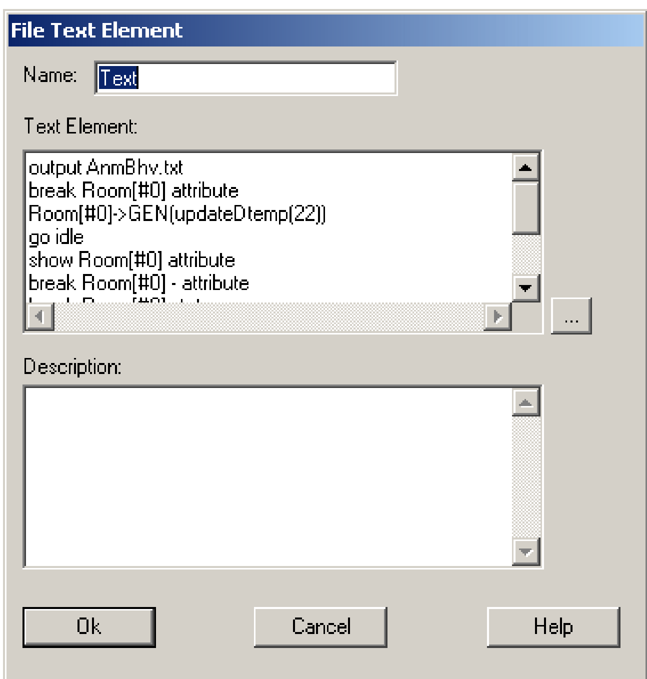


- If you selected **Implementation**, all your elements are set with that element type. You can change this for an element if you want.
- If you selected **Other**, you can set whichever setting is available from the **Element Type** drop-down list.

6. To add a text element to a file, click the New Text Element button, as shown in the following figure. The File Text Element dialog box opens.



7. Enter your text in the File Text Element dialog box, as shown in the following figure, then click **OK**.



8. Click **OK** on the Features dialog box to apply your changes.

The `CG::File::AddToMakefile` property (which supersedes the previous `GenerateInMakefileOnly` property) enables you to include an external file (when the `CG::File::Generate` property is set to `Cleared`) in a build. The `CG::File::AddToMakefile` property works in conjunction with the `CG::File::Generate` property. This technique is used in many of the Rhapsody samples with GUIs to include resources (such as dialog boxes) built with MFC in a component. The external file is included in the makefile, and therefore compiled if needed (although not generated by Rhapsody). Using this property is equivalent to adding a file as an additional source under either a component or a configuration in the browser. See the definitions provided for the properties on the applicable Properties tab of the Features dialog box. Refer also to the *Properties Reference Manual*.

Behavioral Model

To specify a system's behavior, use the use cases to determine the interactions between the system's (static structure) objects. These interactions show how the system components collaborate. Each interaction realizes one scenario within the system, typically starting with an external event generated by a system actor and terminating at a point where the desired function, or use case, is accomplished.

Sequence Diagrams

Sequence diagrams (SDs) describe message exchanges within your project. You can place messages in a sequence diagram as part of developing the software system. You can also run an animated sequence diagram to watch messages as they occur in an executing program.

Sequence diagram show scenarios of messages exchanges between roles played by objects. This functionality can be used in numerous ways, including analysis and design scenarios, execution traces, expected behavior in test cases, and so on.

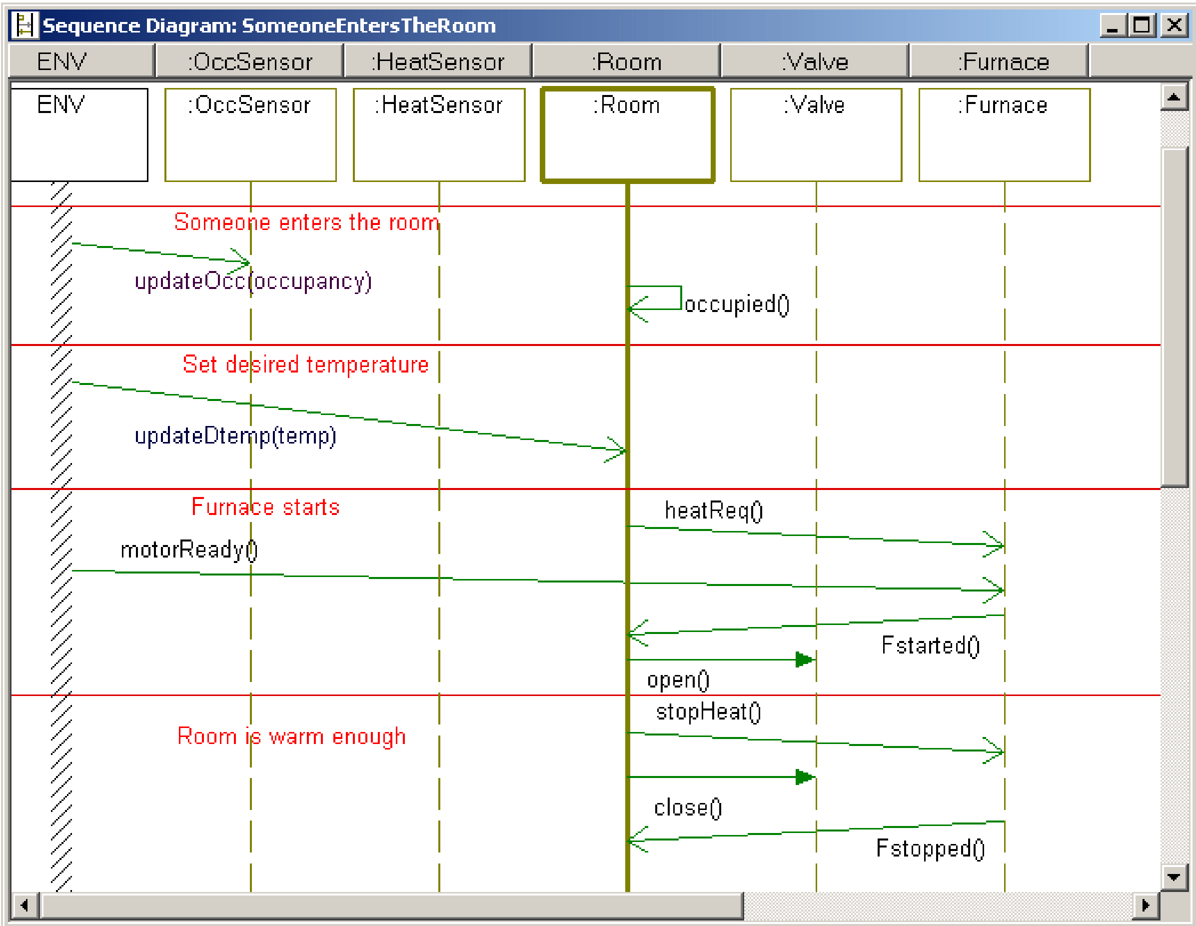
Sequence diagrams help you understand the interactions and relationships between objects by displaying the messages that they send to each other over time. In addition, they are the key tool for viewing animated execution. When you run an animated program, its system dynamics are shown as interactions between objects and the relative timing of events.

Sequence diagrams are the most common type of interaction diagrams.

Each scenario is depicted as a sequence diagram, where the system objects are depicted as columns, with each column representing the lifeline of an object throughout the scenario. Lifelines can also depict object states and timer events.

The vertical axis is the time dimension showing the exchange of messages between system objects. Messages represent the interactions between objects in the form of events or operation calls. They are depicted as arrows connecting the object lifelines.

The following sequence diagram shows the collaborations that take place within the HomeHeatingSystem once an inhabitant enters a room. The system objects are specified in the first row. Nested objects can be identified using their object path, starting from the top level object and following the hierarchy. With arrays of objects, an index indicates the instance.



The complete behavior requirement of an object is a projection of all object lifelines from each scenario. The set of lifelines in a sequence diagram forms the complete lifecycle of an object as a statechart.

Note

While executing the program with animation active in Rhapsody in C, global objects, which belong to the package, have their original names as animation instance names without the instance index. For example, the global object HomeHeatingSystem has an animation instance name of `HomeHeatingSystem` rather than `HomeHeatingSystem[0]`.

In this scenario, the following messages are passed between objects as events:

Message	Sender	Receiver	Description
`updateOcc()`	`<inhabitant>`	`OccSensor`	Someone has entered the room.
`occupied()`	`<system>`	`Room`	`Room` receives a timer.
`updateDtemp()`	`<inhabitant>`	`Room`	Inhabitant sets a desired temperature.
`heatReq()`	`Room`	`Furnace`	`Room` requests heat from `Furnace`.
`motorReady()`	`<system>`	`Furnace`	System checks whether the `Furnace`'s motor is ready to operate.
`Fstarted()`	`Furnace`	`Room`	`Furnace` tells `Room` that it has started.
`open()`	`Room`	`Valve`	`Room` tells the heating `Valve` to open.
`stopHeat()`	`Room`	`Furnace`	When the temperature is warm enough, `Room` tells `Furnace` to stop generating heat.
`close()`	`Room`	`Valve`	`Room` tells the heating `Valve` to close.
`Fstopped()`	`Furnace`	`Room`	`Furnace` tells `Room` that it has stopped.

Each of the events in the above scenario is generated into an event structure in the package specification file. Because the HomeHeatingSystem example has only one package named `Default`, the event definitions are generated in the `Default.h` file.

Events

Events provide asynchronous communication between reactive objects or tasks. Events can trigger transitions in statecharts.

In Rhapsody in C, events are implemented as objects (structures). The abstract data type and event structure are defined in the package specification file as follows:

```
typedef struct evStart evStart;

struct evStart {
   RiCEvent ric_event;
};
```

An instance of an `RiCEvent` object is embedded in the event's structure as a data member.

Note

`RiCEvent` is a predefined event type provided by the Rhapsody in C framework.

Although events are implemented as objects, they are modeled as operations. Therefore, an event does not have attributes and only has initialization and cleanup operations.

Each event is assigned a dynamic ID by default:

```
/*## package Default */
#define evStart_Default_id 1
```

The event ID can change if the same event is re-used in multiple components, for example, if the same event is used in client and server components. To avoid this situation, which can cause problems in distributed systems, you can assign a permanent ID to an event by setting its `CG::Event::Id` property.

Event Arguments

Events can have data. Although modeled as arguments, the data are implemented as members of the struct. For example, the following code is generated for an `evStart` event with an argument called `go`:

```
typedef struct evStart evStart;
struct evStart {
   RiCEvent ric_event;
   /*** User explicit entries ***/
   int go;
};
```

Event Constructors and Destructors

Constructors and destructors are defined for the event in the package specification file. For example:

```
/* Constructors and destructors: */
ev1 * RiC_Create_ev1();
void ev1_Init(ev1* const me);
void ev1_Cleanup(ev1* const me);
void RiC_Destroy_ev1(ev1* const me);
```

The names of event create and destroy operations have a slightly different pattern than names of event initialize and cleanup operations:

- Create and destroy operation names for events have the format `RiC_Create_<event>()` and `RiC_Destroy_<event>()`, respectively.
- Initialization and cleanup operation names for events have the format `<event>_Init()` and `<event>_Cleanup()`, respectively.

The implementation of the event constructors and destructors is generated in the implementation file for the package. For example:

```
ev1 * RiC_Create_ev1() {
   ev1* me = (ev1*) malloc(sizeof(ev1));
   ev1_Init(me);
   return me;
}
```

With dynamically allocated events, the creator function allocates memory for the event and initializes it via the event initializer:

```
void ev1_Init(ev1* const me) {
   RiCEvent_init(&me->ric_event, ev1_Default_id, NULL);
   me->ric_event.lId = ev1_Default_id;
}
void ev1_Cleanup(ev1* const me) {
   RiCEvent_cleanup(&me->ric_event);
}
void RiCDestroy_ev1(ev1* const me) {
   ev1_Cleanup(me);
   free(me);
}
```

Note

It is possible to statically allocate a block of memory for events at the start of run time, rather than using dynamic memory allocation during run time. See **Static Allocation of Events** for more information.

See **Sending Events** for information on generating and sending events.

Static Allocation of Events

It is possible to allocate events from a static memory pool, rather than dynamically allocating memory for events during run time, by setting the following properties under `CG::Event`:

- `AdditionalNumberOfInstances`—Specifies the number of array elements that should be added if the number of events exceeds the size of the original array
- `BaseNumberOfInstances`—Sets the initial size of the static array to be allocated for events
- `EmptyMemoryPoolCallback`—Specifies the name of the callback function that allocates more memory if the static pool is exhausted
- `EmptyMemoryPoolMessage`—Specifies whether a message is displayed when the static memory pool is empty
- `ProtectStaticMemoryPool`—Specifies whether to protect the static memory pool using an operating system mutex

See the definitions provided for the properties on the applicable **Properties** tab of the Features dialog box. Refer also to the *Properties Reference Manual*.

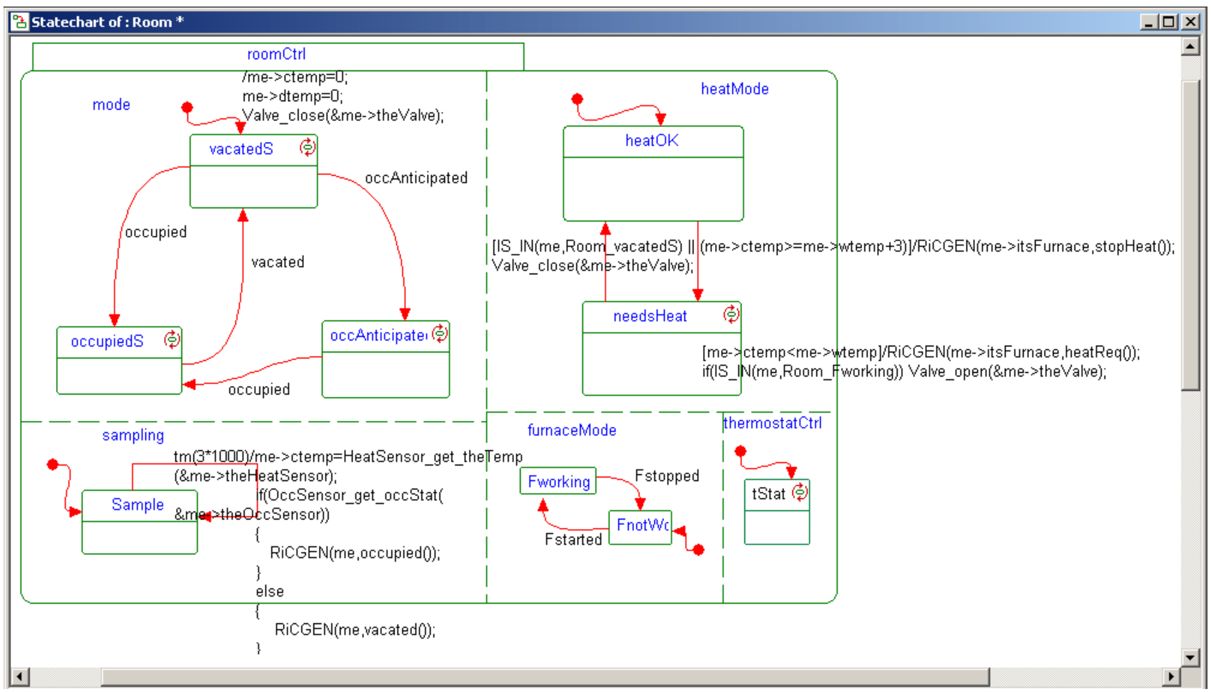
Note

In C, it is possible to allocate only events, but not user-defined objects, from static memory pools.

Statecharts

Statecharts specify the lifecycle of an object in terms of its logical states or modes, which primarily determine the object's response to external stimuli. Object states can be elicited from both the problem statement and the object lifeline in sequence diagrams.

The following figure shows the statechart of the `Room` in the HomeHeatingSystem example.



From the lifeline of the `Room` in the SomeoneEntersTheRoom sequence diagram, you can see that the `Room` polls its `occupancy` attribute to see whether it is occupied. If it is, it sends the required heat to the `Furnace` once the inhabitant has set a desired temperature on the thermostat. Once the `Room` receives a message from the `Furnace` saying that it has started, the `Room` sends a message to the `Valve` telling it to open. When the room is warm enough, the `Room` tells the `Furnace` to stop generating heat, and then closes the `Valve`. Finally, the `Room` receives an acknowledgement from the `Furnace` letting it know that the `Furnace` has stopped.

From this sequence of events, you can see that the `Room` has four regions of responsibility, or *concurrent states*:

Region	Responsibility
mode	Determine the working temperature based on the occupancy.
heatMode	Determine the need for heat.
FurnaceMode	Monitor the state of the Furnace.
sampling	Periodically sample the heat and occupancy sensors.

Accessing and Modifying Attributes

Attributes are accessed via the `me` pointer, which provides a context for the current object. Therefore, specifying conditions, assigning values, and performing calculations requires accessing attributes through this context variable.

Note

You can specify the actual name of the context variable generated as an argument to an operation using the properties for the operation.

For example, in the Room statechart, testing the condition for heat demand is expressed as follows:

```
me->ctemp < me->wtemp
```

When entered as a guard on the transition from the **heatOK** state to the **needsHeat** state in the **heatMode** region of the statechart, this comparison determines whether the `Room`'s current temperature is lower than the working temperature.

Sending Events

Events are generated via the RiCGEN() or CGEN() macro (see **Predefined Actions**). For example, the following statement sends a stopHeat() event to the Furnace:

```
RiCGEN(me->itsFurnace,stopHeat());
```

The RiC or C prefix on the CGEN() macro distinguishes this service from a similar event generation service provided by the Rhapsody framework for other languages. RiCGEN(), CGEN(), and GEN() are all convenience macros that hide the details of event generation.

The first argument of the RiCGEN() statement is the target, or the object that is to receive the event. The target can be:

- A global object that is visible to the sender.
- A subobject.
- A rolename that designates a link to a peer object. For example, the Room sends events to the Furnace by accessing the link through the itsFurnace role.
- A parameter of the current event (the one being sent).
- The current object, as when a message-to-self is sent with RiCGEN(me, event()).

The second argument of the RiCGEN() statement is the event being sent, including event arguments (if it has any). The arguments must agree with the event parameters. For example, the following statement generates an updateDtemp event and sends it to the Room, passing the desired temperature as an event parameter:

```
RiCGEN(me->itsRoom, updateDtemp(val));
```

The params Keyword

The params keyword provides access to the parameters of the consumed event. For example, in the following transition, the value of the occupancy parameter passed with the updateOcc() event received by the OccSensor object is passed as the second parameter of the set_occStat() operation:

```
updateOcc()/
OccSensor_set_occStat(me, params->occupancy);
```

In this example, the updateOcc() event is the trigger of the transition and the OccSensor_set_occStat() call is part of the action that is executed as a result.

In other words, when the OccSensor object receives an updateOcc() event with a parameter of 1, updateOcc(1), the sensor's occStat attribute is updated with the value 1 as a result.

States

Rhapsody in C supports only the Flat implementation of statecharts. In the Flat implementation, states are implemented as enumerated types. Every state that has a substate is represented as a `struct` member of the enum. For example, the statechart of the HomeHeatingSystem has only one (apparent) state, the **systemControl** state. This is implemented in the HomeHeatingSystem structure as follows:

```
struct HomeHeatingSystem {
    RiCReactive ric_reactive;
    /*states enumeration: */
    enum HomeHeatingSystem_Enum{
                HomeHeatingSystem_RiCNonState=0,
                HomeHeatingSystem_systemControl=1}
    rootState_subState,
    rootState_active;
};
```

Switch statements are used to select between the outward bound transitions from a state. The switch statements are found in the operations that implement the event processing of a statechart. These include, among others, the `takeEvent()`, `dispatchEvent()`, `serializeStates()`, and `exit()` operations generated for each state. See the following sections for more informations:

- Reactive Objects
- Taking Events
- Dispatching Events
- Exiting From a State with Exit()

Root State

Every statechart has a *root state*, which is the initial state of the statechart. The default transition leads from the (invisible) root state directly into the state that is the target of the default transition when the object starts its behavior.

A `<state>_active` pointer is generated for every component state of an **And** state. This member is the low-level active state (leaf state) used for taking events. The received event first tries to be consumed by the `<state>_active` state. If it cannot, it then tries to be consumed by the parent.

A `<state>_subState` pointer is generated for each **Or** state (parent state). This member is the active child state in the parent. It is used for exiting from the parent state. When the parent state exits, its active child state should also exit.

By default, the root state is both a component state and an **Or** state. Therefore, both `rootState_subState` and `rootState_active` members are generated for it in the object.

Transitions

Every transition is mapped to the object's private operations for implementing statecharts, with optimizations for "short" functions (see **Inlining Transition Code**). These operations set the necessary values for the current active states, execute the actions, and so on. Several outbound transitions from the same state are mapped to the same operation, and are distinguished using a `switch()` statement.

Inlining Transition Code

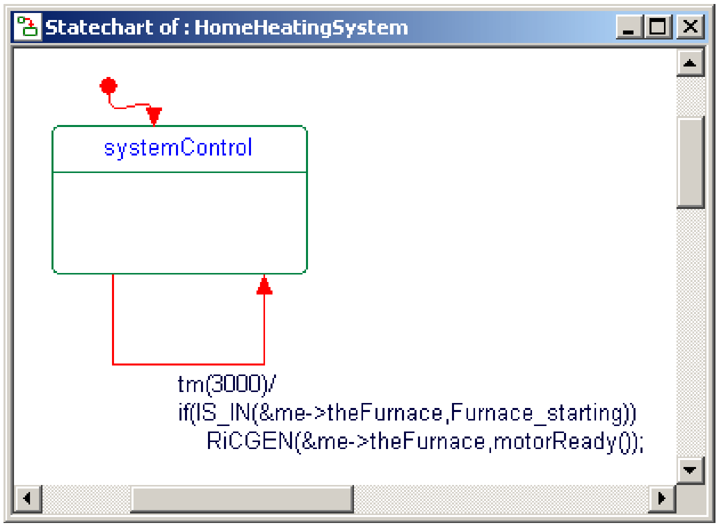
The `CG::Class::ComplexityForInlining` property specifies the upper bound for the number of lines in user code that are allowed to be inlined. The default is `3`.

"User code" is the action part of transitions in statecharts. For example, using the value of 3, all transitions with actions consisting of three lines or fewer of code are automatically inlined in the calling function.

Inlining is replacing a function call in the generated code with the actual code statements that make up the body of the function. This optimizes code execution at the expense of a slight increase in code size. For example, increasing the number of lines that can be inlined from 3 to 5 has shortened the code execution time up to 10%.

For example, in the statechart of the `HomeHeatingSystem` object, the **systemControl** state has an out transition on a timeout with the following action part:

```
if(IS_IN(&me->theFurnace,Furnace_starting))
   RiCGEN(&me->theFurnace,motorReady());
```



This action sends a `motorReady()` event from the `HomeHeatingSystem` to the `Furnace`, if the `Furnace` is in the starting state.

If the `ComplexityForInlining` property is set to `0` (the default value), the transition code is generated in the `takeEvent()` operation of the **systemControl** state of the `HomeHeatingSystem` object as follows:

```
int HomeHeatingSystem_systemControl_takeEvent(
HomeHeatingSystem* const me, short id) {
    int res = eventNotConsumed;
    if(id == Timeout_id)
    {
          if(RiCTimeout_getTimeoutId((RiCTimeout*)
                 me->ric_reactive.current_event) ==
                 HomeHeatingSystem_Timeout_systemControl_id)
          {
                 NOTIFY_TRANSITION_STARTED(me,HomeHeatingSystem,
                 "1");
                 HomeHeatingSystem_systemControl_exit(me);
                 {
                 /*#[ transition 1 */
                 if(IS_IN(&me->theFurnace,Furnace_starting))
                 RiCGEN(&me->theFurnace, motorReady());
                 /*#]*/
                 }
                 systemControl_entDef(me);
                 NOTIFY_TRANSITION_TERMINATED(me,
                 HomeHeatingSystem,"1");
                 res = eventConsumed;
          }
    }
    return res;
}
```

The `dispatchEvent()` operation of the `rootState` of the **HomeHeatingSystem** object calls the `takeEvent()` operation as follows:

```
static int rootState_dispatchEvent(
    void * const void_me, short id) {
    HomeHeatingSystem * const me =
          (HomeHeatingSystem *)void_me;
          int res = eventNotConsumed;
          switch (me->rootState_active) {
                 case HomeHeatingSystem_systemControl:
                 {
                 res =
                 HomeHeatingSystem_systemControl_takeEvent(
                 me, id);
                 break;
                 };
          default:
                 break;
    };
    return res;
}
```

However, if `ComplexityForInlining` is set to `3`, for example, because the action code is less than three lines, it is generated directly in the `dispatchEvent()` operation of the `rootState`, replacing the `takeEvent()` call as follows:

```
static int rootState_dispatchEvent(void * const void_me,
    short  id) {
    HomeHeatingSystem * const me = (HomeHeatingSystem *)
            void_me;
    int res = eventNotConsumed;
    switch (me->rootState_active) {
            case HomeHeatingSystem_systemControl:
            {
                    if(id == Timeout_id)
                    {
                    if(RiCTimeout_getTimeoutId(
                    (RiCTimeout*) me
                    ->ric_reactive.current_event) ==
                    HomeHeatingSystem_Timeout_systemControl_id)
                    {
                    NOTIFY_TRANSITION_STARTED(me,HomeHeatingSystem,
                    "1");
                    RiCTask_unschedTm(me->ric_reactive.myTask,
                    HomeHeatingSystem_Timeout_systemControl_id,
                    &me->ric_reactive);
                    NOTIFY_STATE_EXITED(me,HomeHeatingSystem,
                    "ROOT.systemControl");
                    {
                    /*#[ transition 1 */
                    if(IS_IN(&me->theFurnace, Furnace_starting))
                    RiCGEN(&me->theFurnace,motorReady());
                    /*#]*/
                    }
                    }
                    /* rest of dispatchEvent() */
}
```

Starting Statecharts

Rhapsody generates two operations to initialize statecharts and start reactive behavior:

- `initStatechart()`
- `startBehavior()`

Initializing Statecharts

The `initStatechart()` operation initializes a reactive object’s statechart. For example, the following `initStatechart()` operation, generated in the implementation file for the `HomeHeatingSystem`, initializes the HomeHeatingSystem’s statechart:

```
static void initStatechart(HomeHeatingSystem* const me) {
    me->rootState_subState = HomeHeatingSystem_RiCNonState;
    me->rootState_active = HomeHeatingSystem_RiCNonState;
}
```

This routine initializes the `rootState_subState` and `rootState_active` pointers for the **HomeHeatingSystem** object to `<object>_RiCNonState` (the default state is `0`) when the object is created.

Starting Reactive Behavior

The `startBehavior()` operation starts the behavior of reactive objects:

- The `<package>_startBehavior()` operation starts the behavior of the reactive objects in a package.
- The `<object>_startBehavior()` operation starts the behavior of an individual object.

Note that `startBehavior()` should not be called from within the constructor.

Operations on States

Rhapsody automatically generates functions to handle state-based operations, including:

- Entering a state
- Taking events
- Checking whether an object is in a particular state
- Exiting from a state

Note

The `CG::Class::ImplementStatechart` property must be set to `Checked` for these operations to be generated.

These operations are generated in the Framework Entries section of the specification file for an object.

Entering a State

The `enter()` operation allows an object to enter a state after the object has successfully received a trigger and any possible guard condition has been passed. The `enter()` operation also executes any user-defined action on entry for the state. The `enter()` operation name has the following format:

```
<object>_<state>_enter(<object*> const <me>)
```

For example, the following `enter()` operation is generated for the **systemControl** state of the HomeHeatingSystem:

```
void HomeHeatingSystem_systemControl_enter(
   HomeHeatingSystem* const me);
```

The `enter()` operation sets the `<state>_subState` and `<state>_active` members of the state being exited (based on the statechart) to the one being entered. For example, the `enter()` operation for the **systemControl** state of the HomeHeatingSystem sets these two members of the **rootState** (the previous state) to the **systemControl** state (the one being entered), as follows:

```
void HomeHeatingSystem_systemControl_enter(
    HomeHeatingSystem* const me) {
    NOTIFY_STATE_ENTERED(me, HomeHeatingSystem,
            "ROOT.systemControl");
    me->rootState_subState = HomeHeatingSystem_systemControl;
    me->rootState_active = HomeHeatingSystem_systemControl;
    RiCTask_schedTm(me->ric_reactive.myTask, 3000,
            HomeHeatingSystem_Timeout_systemControl_id,
                    &me->ric_reactive, "ROOT.systemControl");
}
```

Note

An `enter()` operation is not generated for the root state.

Taking Events

The `takeEvent()` operation takes an event off the event queue and evaluates whether that event is valid to trigger a transition of the object out of its current state. The `takeEvent()` operation name has the following format:

```
<object>_<state>_takeEvent(<object>* const <me>,
<event ID>)
```

The event ID is the identification number generated for an event at the top of the package specification file.

For example, for the **systemControl** state of the HomeHeatingSystem, the following `takeEvent()` operation is generated:

```
int HomeHeatingSystem_systemControl_takeEvent(
    HomeHeatingSystem* const me, short id);
```

This operation has the following implementation:

```
int HomeHeatingSystem_systemControl_takeEvent(
    HomeHeatingSystem* const me, short id) {
    int res = eventNotConsumed;
    if(id == Timeout_id)
     {
           if(RiCTimeout_getTimeoutId((RiCTimeout*)
                  me->ric_reactive.current_event) ==
                  HomeHeatingSystem_Timeout_systemControl_id)
        {
                  NOTIFY_TRANSITION_STARTED(me,
                  HomeHeatingSystem, "1");
                  HomeHeatingSystem_systemControl_exit(me);
                  {
                  /*#[ transition 1 */
                  if(IS_IN(&me->theFurnace,Furnace_starting))
                  RiCGEN(&me->theFurnace,motorReady());
                  /*#]*/
                  }
                  systemControl_entDef(me);
                  NOTIFY_TRANSITION_TERMINATED(me,
                  HomeHeatingSystem, "1");
                  res = eventConsumed;
           }
    }
    return res;
}
```

Note

A `takeEvent()` operation is not generated for the root state.

Dispatching Events

The `dispatchEvent()` operation uses a `switch` statement to process the outbound transitions from the states of an object. For example, the `dispatchEvent()` operation generated for the operating state of the `Furnace` in the HomeHeatingSystem sample, uses roughly the following `switch` statement to process the out transitions from the idle, shutting, working, and starting substates of the operating orthogonal state:

```
static int operating_dispatchEvent(Furnace* const me,
short id) {
    int res = eventNotConsumed;
    switch (me->operating_active) {
            case Furnace_idle:
            {
                    /* process out transitions from idle state */
                    res = eventConsumed;
                    break;
            };

            case Furnace_shutting:
            {
                    /* process out transitions from shutting
                    state */
                    res = eventConsumed;
                    break;
            };
            case Furnace_starting:
            {
                    /* process out transitions from starting
                       state */
                    res = eventConsumed;
                    break;
            };
            case Furnace_working:
            {
                    /* process out transitions from working
                    state */
                    res = eventConsumed;
                    break;
            };
            default:
            break;
    };
    return res;
}
```

Checking an Object's State with IN()

The `IN()` operation checks whether or not an object is in a particular state. The `IN()` operation name has the following format:

```
<object>_<state>_IN(<object>* const <me>)
```

It returns `True` if the object is in the state, and `False` otherwise.

For example, for the **systemControl** state in the HomeHeatingSystem, the following `IN()` operation is generated:

```
/*systemControl:*/
int HomeHeatingSystem_systemControl_IN(
    HomeHeatingSystem* const me);
```

This operation has the following implementation:

```
int HomeHeatingSystem_systemControl_IN(
    HomeHeatingSystem* const me) {
    return me->rootState_subState ==
            HomeHeatingSystem_systemControl;
}
```

Note the following:

- An `IN()` operation is also generated for the root state.
- You can use either the `IN()` operation generated for the state or the `RiC_IS_IN()` macro for the object to determine whether an object is in a particular state. See **RiCIS_IN() or IS_IN()** for more information on this macro.

Exiting From a State with Exit()

The `exit()` operation allows an object to exit from a state. It also executes any user-defined action on exit for the state. The `exit()` operation name has the following format:

```
<object>_<state>_exit(<object*> const <me>)
```

For example, the following `exit()` operation is generated for the **systemControl** state in the HomeHeatingSystem:

```
void HomeHeatingSystem_systemControl_exit(
    HomeHeatingSystem* const me);
```

This operation has the following implementation:

```
void HomeHeatingSystem_systemControl_exit(
    HomeHeatingSystem* const me) {
            RiCTask_unschedTm(me->ric_reactive.myTask,
                    HomeHeatingSystem_Timeout_systemControl_id,
                    &me->ric_reactive);
            NOTIFY_STATE_EXITED(me, HomeHeatingSystem,
                    "ROOT.systemControl");
}
```

Note

An `exit()` operation is generated for the root state.

Predefined Actions

Rhapsody provides several predefined action statements that you can use in addition to native statements in the programming language anywhere you write code in Rhapsody.

For example, you can use predefined action statements in actions on transitions or in bodies of triggered operations in statecharts. The action statements are defined in the real-time framework (in `RiCReactive.h`) as macros to minimize their impact on the generated source code.

When generating events, note the following:

- If you are generating an event in the action part of a transition, the event name must include parentheses. For example, if you are generating an event `ev1`, use `ev1()` instead of `ev1` as the name of the event to be generated.
- If the name of the instance that is the target of the event is not a pointer, use the address operator `&` with the instance name as an argument to the event generation statement. For example, when sending an event to `itsRoom`, where `itsRoom` is defined as an instance of `Room`, use the address operator `&itsRoom` rather than `itsRoom (pointer)` as an argument.

RiCIS_IN() or IS_IN()

The `IS_IN()` statement determines whether an object is in a particular state. `RICIS_IN()` has the same effect as `IS_IN()`. This statement takes a pointer to an object and the name of the state being checked as arguments. The name of the state has the format `<object>_<state>`.

For example, to make sure that a **Furnace** object is not in the **faultS** state before it transitions from one state to another, you can use the following `IS_IN()` statement as a guard on a transition in the statechart for the **Furnace**:

```
[!IS_IN(me,Furnace_faultS)]
```

The definition of `IS_IN()` is as follows:

```
#define IS_IN(me, state) state##_IN((me))
```

This macro calls the `IN()` operation generated for the state. See **Checking an Object's State with IN()**.

When referencing states, you must use the generated state name. This can be tricky when referencing sibling states that have the same name. For example, if an object `A` has an **And** state `B` with concurrent states `B1` and `B2`, and each of these has a substate `C`, the following enumerated values are generated for these states:

```
/*states enumeration: */

enum A_Enum{ A_RiCNonState=0, A_B=1, A_B2=2,
   A_B2_C=3, A_B1=4, A_C=5 }
```

The generated name of substate `C` of `B1` is `A_C`. Therefore, the proper macro call to see whether `A` is in `C` of `B1` would be `IS_IN(me, A_C)`, not `IS_IN(me, A_B1_C)`.

RiCGEN() or CGEN()

The `RiCGEN()` statement generates an event and sends it to a particular instance. `RiCGEN()` has the same effect as `CGEN()`.

For example, to send an `Fstarted()` event to an instance `itsRoom[1]`, add the following code to the action part of a transition:

```
RiCGEN(me->itsRoom[1], Fstarted());
```

The definition of `RiCGEN()` is as follows:

```
#define RiCGEN(INSTANCE,EVENT)                              \
{                                                           \
   if ((INSTANCE) != NULL) {                                \
         RiCReactive * reactive = &((INSTANCE)->ric_reactive);\
         RiCEvent * event = &(RiC_Create_##EVENT->ric_event); \
         RiCReactive_gen(reactive, event, RiCFALSE);         \
   }                                                        \
}
```

RiCGEN_BY_GUI() or CGEN_BY_GUI()

The `RiCGEN_BY_GUI()` statement generates an event from a GUI application and sends the event to an instance. `RiCGEN_BY_GUI()` has the same effect as `CGEN_BY_GUI()`.

For example, to send a `fault()` event to an instance `GtheFurnace` from a GUI application, use:

```
RiCGEN_BY_GUI(GtheFurnace, fault());
```

The definition of `RiCGEN_BY_GUI()` is as follows:

```
#define RiCGEN_BY_GUI(INSTANCE,EVENT)                              \
{\
    if ((INSTANCE) != NULL) {                                       \
            RiCReactive * reactive = &((INSTANCE)->ric_reactive);\
            RiCEvent * event = &(RiC_Create_##EVENT->ric_event); \
            RiCReactive_genBySender(reactive, event, RiCGui);    \
    }                                                               \
}
```

`RiCGEN_BY_GUI()` uses the framework routine `RiCReactive_genBySender()` rather than `RiCReactive_gen()` to actually send the event. With GUI applications, the GUI items are not part of the Rhapsody model and the sender of the event can therefore not be known. `RiCReactive_genBySender()` can identify a GUI item as the sender of the event.

RiCGEN_BY_X() or CGEN_BY_X()

The `RiCGEN_BY_X()` statement generates an event and sends it to an instance while identifying the sender of the event. `RiCGEN_BY_X()` has the same effect as `CGEN_BY_X()`. Either statement can be useful for sending events from within global functions.

`RiCGEN_BY_X()` uses the `RiCReactive_genBySender()` framework routine to send the event because it identifies a particular object as the sender of the event.

For example, to send a `fault()` event to a `Furnace[1]` instance while identifying the sender of the event as `Room[2]`, use:

```
RiCGEN_BY_X(Furnace[1], fault(), Room[2], Room);
```

The last argument, in this case `Room`, identifies the type of the sender.

Use `GEN_BY_X` only in very special cases when you know which `AOMAnimationItem` is sending the message, but Rhapsody cannot figure this out for itself. For example, you can create an application with some GUI classes, `GUI1` and `GUI2`, and some classes that do things, `Huey` and `Louey`. You create all the classes in Rhapsody, so the animation shows instances of all four.

Now associate some GUI with classes `GUI1` and `GUI2`. Because GUIs are more easily created with MFC wizards than with Rhapsody, use the wizards. The constructor of `GUI1` constructs a modeless dialog with some buttons.

Next, configure each of the buttons to generate an event. For example:

```
void myDialog::OnButtonXPushed() {
   myHuey->GEN(E);
}
```

This is fine, except the animation does not know where the event came from. Instead, use `GEN_BY_GUI`, as follows:

```
void myDialog::OnButtonXPushed() {
   myHuey->GEN_BY_GUI
```

The animation output window displays the following message:

```
event E generated by GUI
```

If the class `myDialog` had a method `GUI1 *myOwner` that pointed to the instance of `GUI1` to which it belongs, you could write:

```
void myDialog::OnButtonXPushed() {
   myHuey->GEN_BY_X(E,myOwner);
}
```

In this case, the animation (output window, event queue, and sequence diagrams) would display `E` as coming from the correct `GUI1` object. This is especially useful if the GUI and its dialogs are test harnesses that create some real classes that are not yet written.

The definition of `RiCGEN_BY_X()` is as follows:

```
#define RiCGEN_BY_X(INSTANCE,EVENT,SENDER,theClass)        \
{                                                          \
   if ((INSTANCE) != NULL) {                               \
         RiCReactive * reactive = &((INSTANCE)->ric_reactive);\
         RiCEvent * event = &(RiC_Create_##EVENT->ric_event); \
         RiCReactive_genBySender(reactive, event,             \
               aomX2Item(SENDER,aomc##theClass));                 \
   }                                                       \
}
```

RiCGEN_ISR() or CGEN_ISR()

The `RiCGEN_ISR()` statement generates an event from an interrupt service routine. `RiCGEN_ISR()` has the same effect as `CGEN_ISR()`.

The problem with generating events from interrupt service routines is that in some operating systems (such as VxWorks), you are not allowed to allocate memory, delete memory, or block on a resource (for example, `lock()` on a semaphore). Therefore, `RiCGEN_ISR()` does not allocate new events, but uses a pointer to an event that you must supply.

There are two ways to use `RiCGEN_ISR()`:

- Initialize your own event pool and use it to manage the supply of events to `RiCGEN_ISR`. For example:

  ```
  RiCGEN_ISR(myEventPool[theNextFreeEvent]);
  ```

  To do this, you must set the `CG::Event::DeleteAfterConsumption` property to `False`.

- Use static memory management on events supplied by Rhapsody.

  To do this, you must set the following static memory management properties under `CG::Event`:

  - `BaseNumberOfInstances`—Set to the number of events in the pool.
  - `AdditionalNumberOfInstances`—Set to 0.
  - `ProtectStaticMemoryPool`—Set to `Cleared`. This means that the event memory pool is not multi-thread safe.
  - `DeleteAfterConsumption`—Set to either `False` or `Default`.

  The call to `RiCGEN_ISR()` is as follows:

  ```
  RiCGEN_ISR(RiC_Create_ev());
  ```

The definition of `RiCGEN_ISR()` is as follows:

```
#define RiCGEN_ISR(INSTANCE,EVENT)
   RiCReactive_gen(&((INSTANCE)->ric_reactive),
   (RiCEvent*)EVENT, RiCTRUE)
```

RiCREPLY() or CREPLY()

The `RiCREPLY()` statement returns a value from a triggered operation. `RiCREPLY()` has the same effect as `CREPLY()`.

For example, both of the following calls returns a value from a triggered operation:

```
count = 2;
RiCREPLY(count);
```

or

```
RiCREPLY(2);
```

The definition of `RiCREPLY()` is as follows:

```
#define RiCREPLY(retVal) params->ric_reply = (retVal)
```

RiCSETPARAMS() or CSETPARAMS()

The `RiCSETPARAMS()` statement sets the parameters of an event. `RiCPARAMS()` has the same effect as `CSETPARAMS()`. You do not need to manually write `RiCSETPARAMS()` in code—it is automatically generated in the dispatchEvent() routine of any event that has arguments.

When the event queue is ready to take an event, it calls `RiCSETPARAMS()` to allocate a variable `params` as a pointer to the event. This macro enables you to write the following statement in the guard or action part of a transition to access an argument of the event without repeating the name of the event:

```
params-><argument>
```

For example, for a transition on an event `ev1` with an argument `arg1`, you can check whether `arg1` is equal to 4 before taking the transition using the following call:

```
ev1[params->arg1 == 4]
```

The definition of `RiCSETPARAMS()` is as follows:

```
#define RiCSETPARAMS(me,type)type * params = \
    (type *)((me)->ric_reactive.current_event)
```

DYNAMICALLY_ALLOCATED()

The DYNAMICALLY_ALLOCATED macro is used in the Create() operation to distinguish between dynamically allocated and statically allocated instances. This difference allows the use of termination connectors in the statecharts of statically allocated instances.

The definition of DYNAMICALLY_ALLOCATED() is as follows:

```
#define DYNAMICALLY_ALLOCATED(object) {RiCReactive_setshouldDelete(&object-
>ric_reactive,
   RiCTRUE);
}
```

Index

Symbols

A

B

C

D

E

F

G

H

I

P

Q

R

S

T

U

V

W