

AltaRica Extended's translators

*Syntactical check, Properties control,
Translation, ...*

Copyright © 2008 Dassault Systemes

Abstract

BPA DAS enables to imitate AltaRica model. For treatment part, it generates a AltaRica Extended-format file. Plugins whose type is translator take this file in input and generates a file in another format which is more workable for other plugins.

Currently available formats are :

- AltaRica DataFlow : This format is widely use in AltaRica environment
- AltaRica Mec5 : A model-checking tool (Made by the LaBRI : Laboratoire Bordelais de Recherche en Informatique ; 'Inventor' of AltaRica language)
- Moca12 : Stochastic simulator base on predicates Petri nets (Property of TOTAL corp.)
- Other specific format used inside BPA DAS whorkshop.

These translators are based on modules allowing reading of files in AltaRica Extended format (so it enables to verify syntax and semantics), allowing conversion of assertions to dataflow equations, allowing setting flat of a model, allowing elementary properties check, ...

Some plugins are made in order to interface the following functionalities in BPA DAS:

- Syntactical checker: Verifies that code is complying with AltaRica Extended format.
- Property Control: Verifies model semantics, and its setting flat. It also verifies elementary properties like external clauses validation or loop in assertions.
- Translation: Translate model in predetermined format (model can possibly be anonymised for confidential reasons).
- External tools: Launch an external tool for current model possibly converted by a translator.

Table of Contents

Syntactical check	4
Syntactical check launching	4
For sytems	4
For components, equipments or operators	4
Result of syntactical check	5
List of usual errors	6
Property check/Control	8
Launching of property control	8
For systems	8
For components, equipments or operators	8
Choose properties to be defined	9
List of checkable properties	10
Model translation	12
Command Translate model	12
External tools	13
Statistics	15
A. AltaRica model verification	16

Syntactical check

Syntactical check verifies that components, equipments and/or systems are consistent with AltaRica Extended's syntax.

This function will display error if AltaRica code generated by software (for flow, states, events,...) or typed by user (guard, transitions, ...) isn't correct.

Syntactical check launching

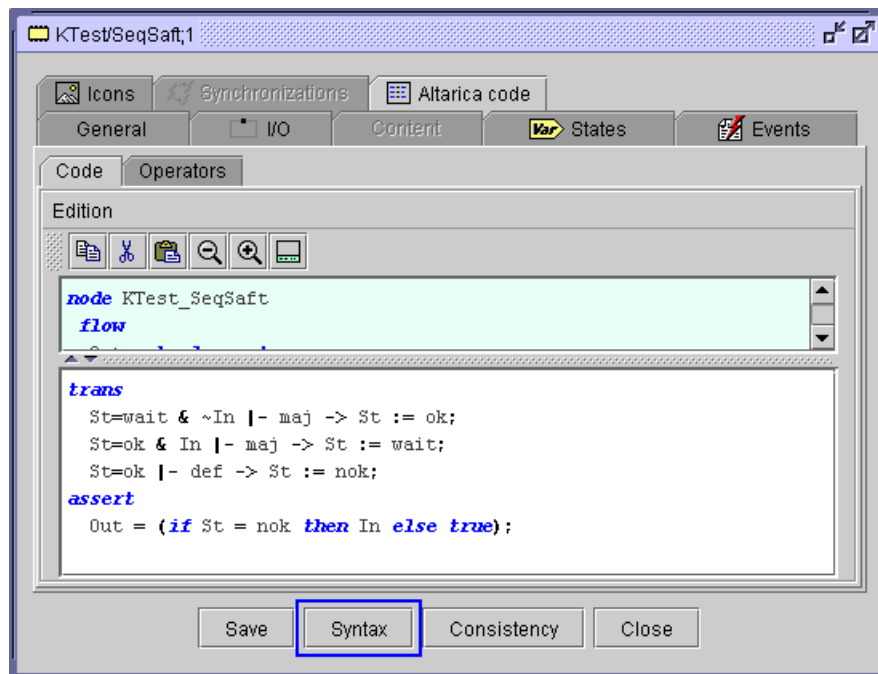
For sytems



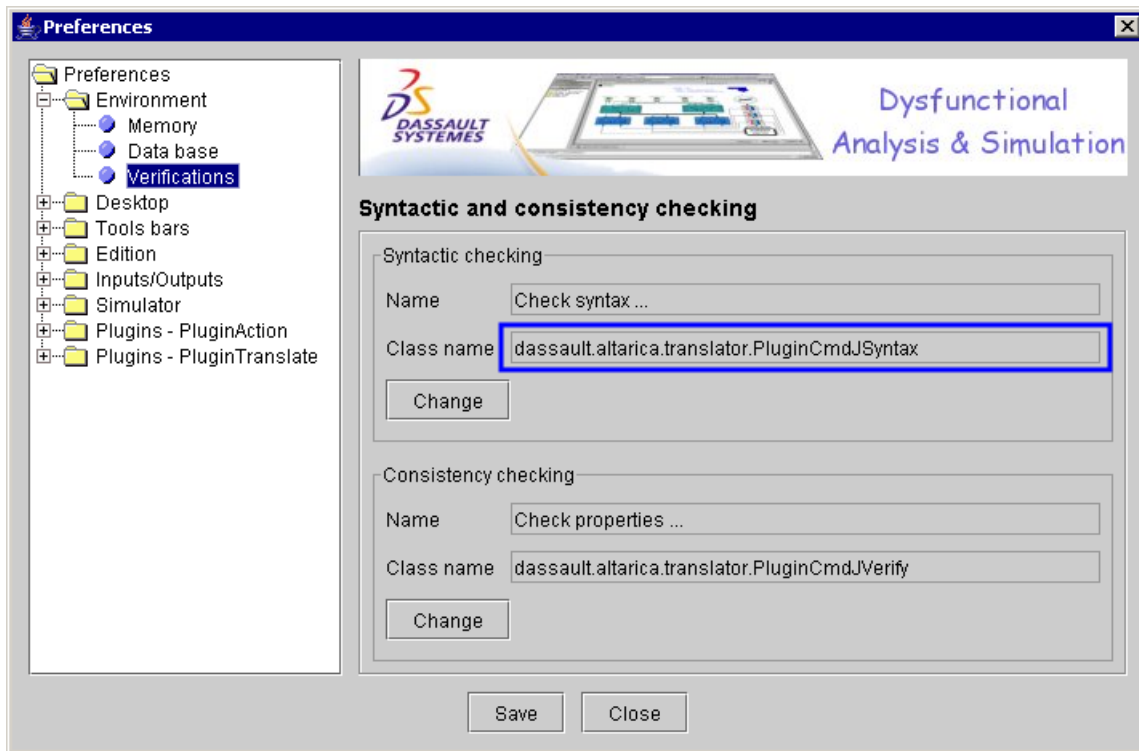
In order to launch syntactical control for current system, use the **Check syntax** command.

For components, equipments or operators

Syntactical control for components, equipments or operators is made in their edition window (button **Syntax**).



During first use, you have to verify that syntactical checker linked with this button is the one of translator. It can be made in preferences of BPA DAS (=> Menu **Options**, Command **Preferences**, Path **Preferences/Environment/Verifications**).

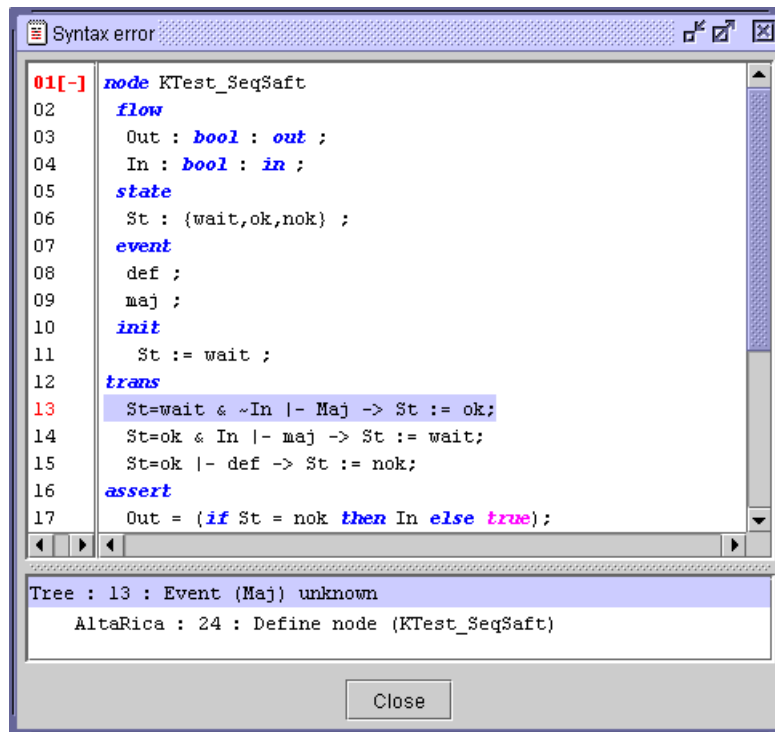


The class name of syntactical check must be `dassault.altarica.translator.PluginCmdJSyntax`. On the contrary, another syntactical checker will be used. In order to change the syntactical checker to be used for components, equipments or operators, click on button **Change** (of part **Syntactic check**).

Result of syntactical check

If there is no mistake, a message will specify it.

On the contrary, a windows **Syntax error** is displayed.



The upper part displays AltaRica code generated by BPA DAS. Every line with a number displayed in red or violet contains at least one error. The bottom part displays causes of the error. A click on the error message allows to select the line that could cause problem.

List of usual errors

1. The exact wording of identifier is defined in the upper editor which is dedicated to visualization of variables definitions (state, flow, icon, event) specified in previous tabs. A typo error in AltaRica code will imply a wrong identifier spelling which won't match exact wording.
2. Wrong syntax of transitions declaration ('=' operator for guards, ':=' operator for assignments).
3. Assertions define assignments on variables from different types (two different enumerate type, one enumerated with a boolean, ...)
4. Key words trans or assert has been forgotten.
5. A missing bracket in `if ... then ... else ...` operator. Good code indentation, in imbricate `if ... then ... else ...` expression, usually enables to avoid these issues. Each closing bracket must correspond to an opening one.

```

if ... then ... else ...
  (if <boolean-expression> then <expression1> else <expression2>)

if ... then ... else ...
  (if <boolean-expression1>
    then <expression1>
    else (if <boolean-expression2>
          then <expression2>
          else <expression3>
        )
  )

```

)
)

Property check/Control

Property check (or control) enables to validate semantics and to verify a certain number of properties on the model.

Verified properties allow to validate - a priori - that a model is compatible with tools that will be used. For example, it's possible to validate - a priori - that model doesn't use arithmetic operators which will avoid tree generation.

Launching of property control

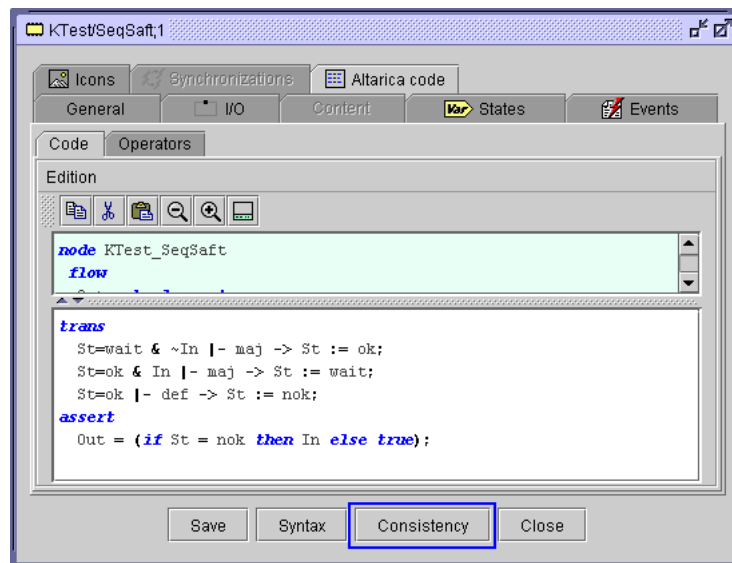
For systems



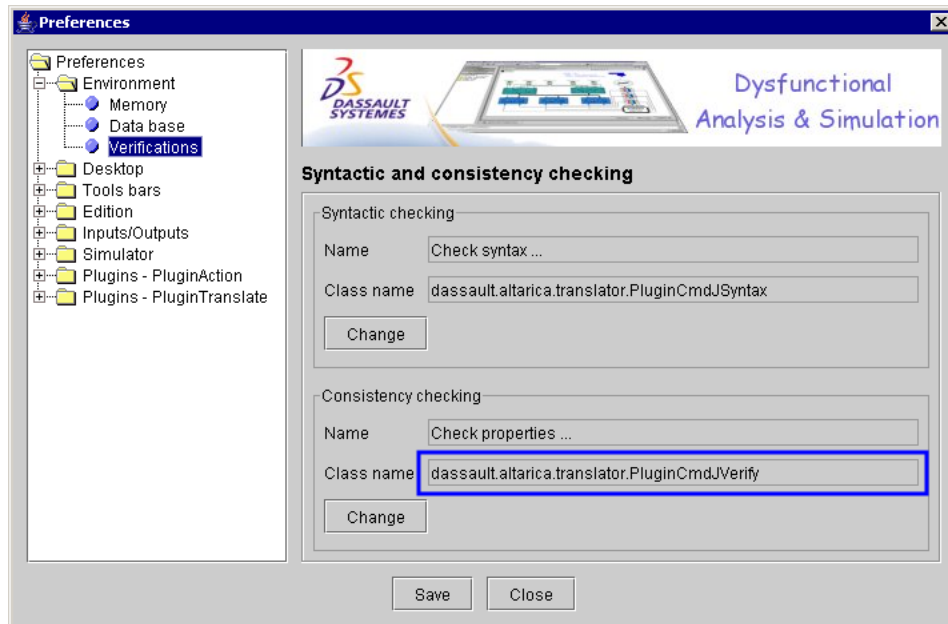
In order to verify properties of current system, use the command **Check properties ...**

For components, equipments or operators

Property control for components, equipments or operators is made in their edition windows (button **Consistency**)



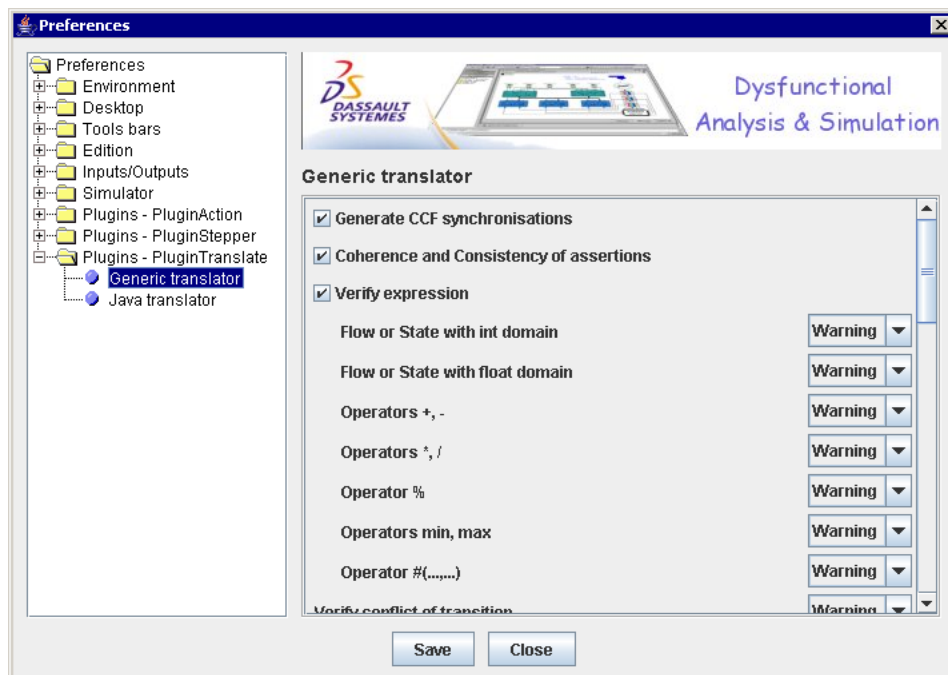
During first use, you have to verify that property control linked with this button is the one of translator. It can be made in preferences of BPA DAS (=> Menu **Options**, Command **Preferences**, Path **Preferences/Environment/Verifications**).



The class-name of consistency-control must be `dassault.altarica.translator.PluginCmdJVerify`. On the contrary, another consistency-control (or property checker) will be used. In order to change the consistency-control to be used for components, equipments or operators, click on button **Change** (in **Consistency checking** frame).

Choose properties to be defined

It's possible to define parameters for properties check in order to focus on properties having impact on treatment tools that will be used.



for each properties to be checked, it's possible:

- either to consider it's not an issue: **Ignore**.
- or to display a message: **Warning**.
- or to stop treatment: **Error**.

Inside property check, one warning or error is enough to display **Error(s) & Warning(s)** window.

In the future, select Error for a property will stop any treatment needing translator.

In order to easily define parameters for a set of properties, they are grouped in categories. To Ignore a category (uncheck and/or choose **Ignore**) enables to ignore every property/category included in it.

List of checkable properties

- Generate CCF (Common Cause Failure) synchronization: Used during translation from AltaRica Extended format to AltaRica format (e.g. the section called "Translation into 'standard' AltaRica").
- Completeness check and assertion consistency: Used during 'dataflowisation' of assertions (e.g. the section called "Convert assertions to assignments").
- Verification on expressions
 - Verification of integer variables presence
 - Verification of float variables presence
 - Verification of operators + and - presence.
 - Verification of operators * and / presence.
 - Verification of operator % presence.
 - Verification of operators min and max presence.
 - Verification of cardinal operators : # (. . .).
- Verification of warring transitions (same guard, same event)
- Verification of transition guards
 - Verification of flow presence inside transitions
 - Verification of always active guards
- Verification of synchronization type
 - Synchronization type
 - BroadCast (Diffusion) type
 - CCF (Common Cause Failure) type
 - Synchronization with events belong to same sub component
- Verify node with local simulation (e.g. the section called "Local simulation of components").
 - Presence of dynamic component
- Verification of generated Java code for Java simulator. This verification enables to confirm that code can be compile by Java compiler
- Verification of loops inside assertions
- Verification of model events
 - Verification of instantaneous event presence
 - Verification of temporized event presence
 - Verification of law presence for each event
- External clauses verification
 - External clauses verification remark
 - External clauses verification law
 - Verification of compatibility between laws with *Aralia*
 - Verification of compatibility between laws with *Moca12*
 - External clauses verification parameter
 - External clauses verification attribute
 - External clauses verification nodeproperty
 - External clauses verification priority
 - Verification of priority affectation only for temporized events.

- External clauses verification bucket
- External clauses verification preemptible
- External clauses verification observer [deprecated]
- External clauses verification predicate
- External clauses verification property
- Always flatness during Mec5 translate.

Event priority is managed in a different manner in AltaRica Extended (overall definition with integer associated to deterministic events) and in AltaRica-Mec5 (local definition with partial order of events at the component level).

In order to keep semantic equivalence between tools, a setting flat of the model must be done if the model uses instantaneous events or if there is a priority..

Model translation

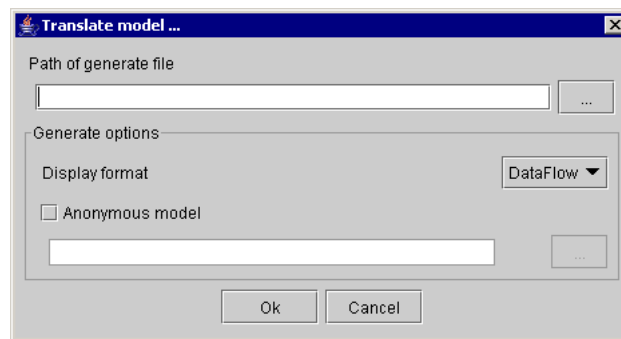
This function allows current model exportation to a file specified by the user in a specific format.

Command *Translate model ...*



In order to launch current system translation, use the **Translate model ...** command.

The following window enables to define parameters for model translation.



- Type result-file name (either directly, or with ... button).
- Select output format in the **Display format** list: **DataFlow**, **Mec5**, **Moca12**, **OTools**
- You can generated an **Anonymous model**(principally for confidential reasons) In this case, a file containing link between original model and anonymized one can be generated.

External tools ...

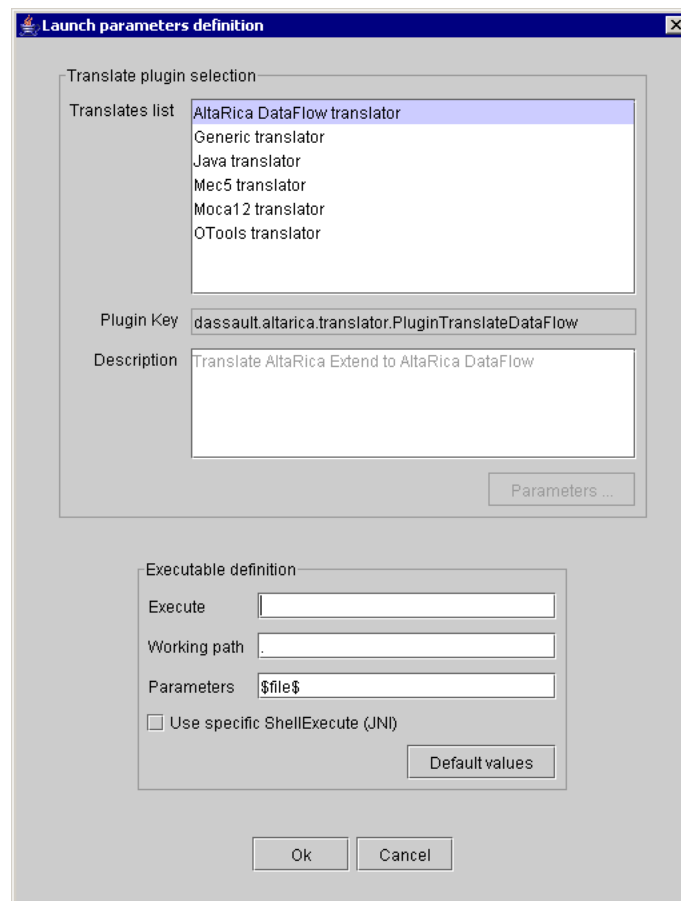
A tool outside BPA DAS can be executed with the current model in a given format.

To use external tools, set plugins manager up. If you need more information, see the chapter *Treatment plugins manager* of user manual.

Among plugins, find the plugin named **External tools** in the library **Translator.jar**, and create an action for this plugin.

Many things can be associated with action : a label, a comment (also called tool tip), an icon and a keyboard shortcut.

You will have to add **Parameters**.



Parameter enables to specify:

- the translator plugin (and possibly its parameters) that will be used to generate the model used by the external tool,
- executable name/path
- working directory
- launching parameters: \$file\$ enables to specify location of generated file in launching parameters.
- If executable launching is made with Java methods or with platform operating system.

When action is set-up, you will have to create a plugin item either in menu or in toolbar.

Then, the external tool will be available in BPA DAS.

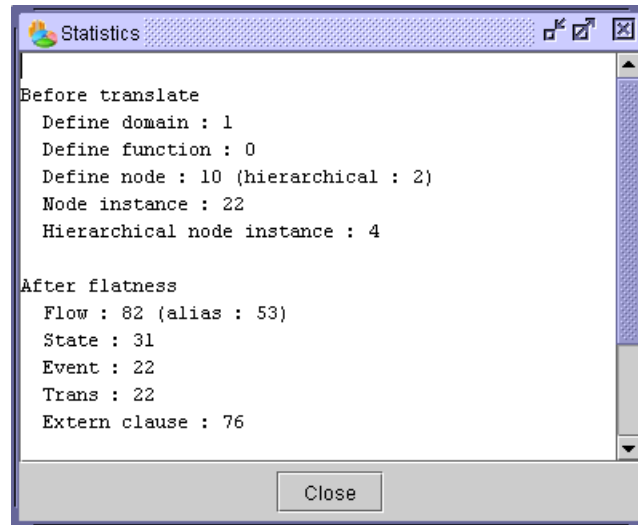
Statistics



Statistics command enables to display informations about current model size.

Statistics module relies on translation module. Syntax errors, grammatical errors, or errors of setting flat, suspend statistics display.

On the contrary, **Statistics** window is displayed.



Statistics are parted in two categories:

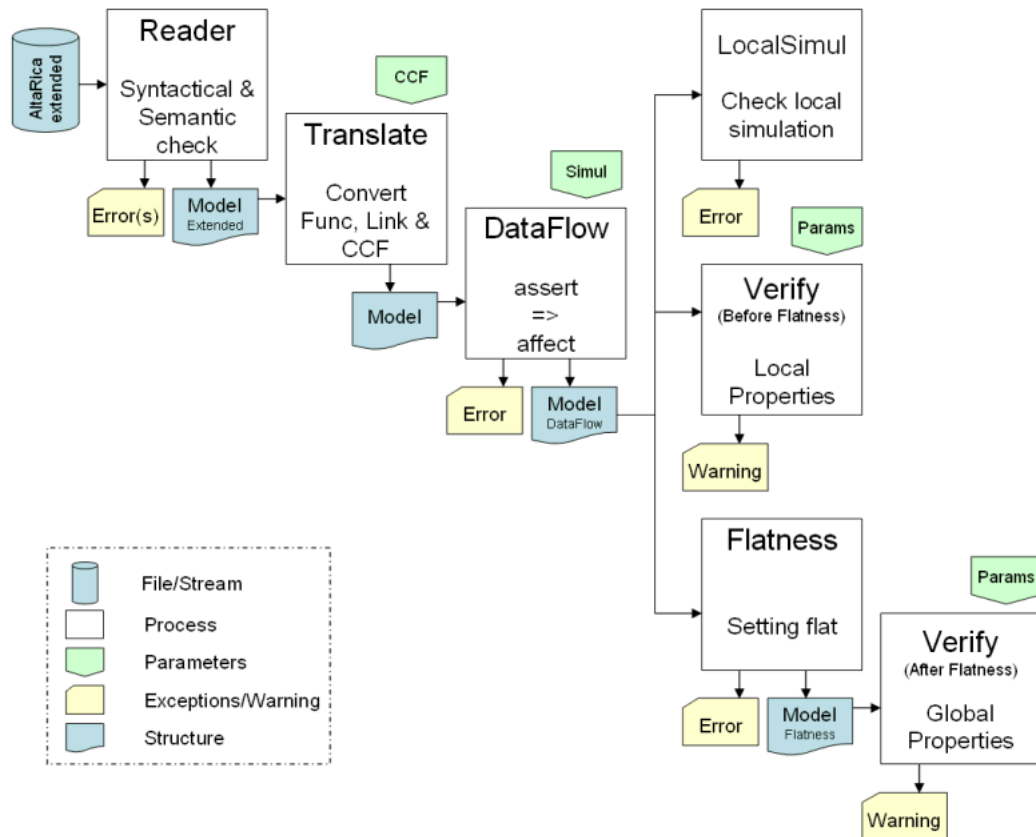
1. Before translation (Cf. the section called "Translation into 'standard' AltaRica"): Display number of high level objects which are manipulated (domains, operators/functions, nodes/components ...)
2. After setting flat (Cf. the section called "Setting flat of model"): Display number of low level objects (flow variables, state variables, events, transitions, ...)

Alias match flow variable defined with equality of type `<out> = <var>` where `out` is a flow variable and `var` a variable (flow or state).

A. AltaRica model verification

AltaRica model verification requires a certain number of steps. Each step will detect potential errors on model.

The linking of these steps is described as follows:



Overall principle consists in:

- reading file (or flow) in extended AltaRica format. Extended format adds elements to manage functions, structured links and common cause failure synchronization.
- translating extended model in 'standard' AltaRica.
- converting assertions to assignments (or convert to 'dataflow' format)
- verify node having behavior with local simulation
- setting flat a model (that's to say removing hierarchy in order to use only one component standing for the system)
- verifying properties like validity of some predefined external clauses, or like loop presence in assertions ...

Each transformation generates a data structure standing for system, if no error is detected. On the contrary case, an exception is generated so that user can modify his model. Some transformations need one or more input-parameters.

The following part deals with each transformation principle in details and all possible error messages.

Syntactical check

Syntactical check uses lexical and lexical analyzer like Lex&Yacc. AltaRica Extended's syntax is defined in BNF format (e.g. LggOcas-02-0.pdf).

If there is at least one error, a message will display line and position of the found error.


```
node syntax
state State:bool;
event Evt;
trans
  State |- Evt -> State=false;
edon;
```

```
AltaRica : 5 : Parser error on token = : syntax error
Event : 4 : Event (Evt) is orphan (No transition use it).
```

Semantic check

Semantic check aims at model consistency validation. That's to say, model uses known and defined data, data are compatible, model seems to be 'logic', ...

Errors on domains :

- Unknown domain

```
node Semantic
flow Out : Power;
edon;
```

```
UndefDomain : 2 : Domain (Power) unknown
AltaRica : 2 : Construct domain
```

- Impossible interval-domain: Min > Max

```
node Semantic
flow Out : [3,2];
edon;
```

```
UndefDomain : 2 : Domain : Min (3) > Max (2) in range
```

- Already declared domain

```
domain Power = {Pos, Null, Neg};
domain Power = [0,2];
```

```
RangeDomain : 2 : Name (Power) already used for another domain
AltaRica : 2 : Defined domain (Power)
```

Errors on structured domains ([link](#)):

- Not structured domain made of fields with structured domain

```
domain First = link
flow A,B:int;
assert
  in^A := out^A;
  in^B := out^B;
```

```
knil;

domain Second = link
  flow
    A:int;
    B:First;
  assert
    in^A := out^A;
    in^B := out^B;
knil;
```

```
Link : 11 : Link : Struct domain not allowed
AltaRica : 11 : Construct link
Link : 15 : Flow (B) unknown
AltaRica : 15 : Construct link
```

- 'inverse' and 'assert' clauses are not compatible

```
domain First = link
  flow A,B:int;
  inverse
    in^A; out^A;
  assert
    in^A := out^A;
    in^B := out^B;
knil;
```

```
Flow : 2 : inverse and assert clauses are incompatible
AltaRica : 8 : Construct link
```

- Flow variable already assigned

```
domain First = link
  flow A,B:int;
  assert
    in^A := out^A;
    in^A := out^B;
knil;
```

```
Flow : 2 : Flow (in^A) already assigned
AltaRica : 6 : Construct link
```

Errors on operators/functions (func):

- Structured domain with 'inverse' clause are not allowed in function

```
domain Connect = link
  flow A,B:int;
  inverse in^A; out^A;
  assert
    out^A := in^A;
    in^B := out^B;
knil;

func Operation
  flow
    Operation:int:out;
    Arg1:Connect:in;
```

```

    assert
      Operation = Arg1^B;
cnuf

```

```

Flow : 12 : Struct inverse not possible
AltaRica : 15 : Defined function (Operation)

```

- Output variable already defined

```

func Operation
  flow
    Operation:int:out;
    Arg1,Arg2:int:in;
    Add:bool:out;
    assert Operation =
      (if Add then Arg1+Arg2 else Arg1-Arg2);
cnuf

```

```

Flow : 5 : Out variable already exists
AltaRica : 8 : Defined function (Operation)

```

- Wrong number of argument

```

func Operation
  flow
    Operation:int:out;
    Arg1,Arg2:int:in;
    Add:bool:in;
    assert Operation =
      (if Add then Arg1+Arg2 else Arg1-Arg2);
cnuf

node Args
  flow
    In1,In2:int:in;
    Out:int:out;
    assert
      Out=Operation(In1,In2);
edon

```

```

Expr : 15 : arguments number
AltaRica : 16 : Defined node (Args)

```

- Function already declared

```

func Operation
  flow
    Operation:int:out;
    Arg1,Arg2:int:in;
    Add:bool:in;
    assert Operation =
      (if Add then Arg1+Arg2 else Arg1-Arg2);
cnuf

func Operation
  flow
    Operation:int:out;
    Arg:int:in;

```

```

    assert Operation = -Arg;
cnuf

```

```

Fct : 8 : Name (Operation) already used for another function
AltaRica : 15 : Defined function (Operation)

```

- Unknown function

```

node Node
  flow
    Out:int:out;
    In1,In2:int:in;
  assert
    Out = Fct(In1,In2);
edon

```

```

Expr : 6 : Fct (Fct) unknown
AltaRica : 6 : Construct expression
Expr : 6 : Undef expression
AltaRica : 7 : Defined node (Node)

```

Errors on nodes/components (node):

- Component already declared

```

node Node
  flow
    Out:int:out;
    In1,In2:int:in;
  assert
    Out = In1+In2;
edon

```

```

node Node
  flow
    Out:int:out;
    In:int:in;
  assert
    Out = -In;
edon

```

```

Node : 7 : Name (Node) already used for another node
AltaRica : 15 : Defined node (Node)

```

- Name-conflict with flow variable

```

node Node
  flow
    Out:int:out;
    In1,In1:int:in;
  assert
    Out = In1+In2;
edon

```

```

Flow : 4 : Name-conflict (In1) with flow variable
AltaRica : 7 : Defined node (Node)

```

- Name-conflict with state variable

```
node Node
  state Out:int;
  flow Out:int:out;
    In:int:in;
  assert
    Out = In;
edon
```

```
Flow : 3 : Name-conflict (Out) with state variable
AltaRica : 7 : Defined node (Node)
```

- Name-conflict with symbolic constant

```
node Node
  flow Out:{ST,SF,SB}:out;
  state ST:int;
  assert
    Out = (if ST>3 then SF else SB);
edon
```

```
State : 3 : Name-conflict (ST) with symbolic constant
AltaRica : 6 : Defined node (Node)
```

- Unknown component

```
node Unit
  flow
    Out:bool:out;
    In:bool:in;
  state OK:bool;
  assert
    Out = (if OK then In else false);
edon

node Equip
  flow
    Out:bool:out;
    In:bool:in;
  sub
    A,B:unit;
  assert
    A.In = In;
    B.In = In;
    Out = (A.In | B.In);
edon
```

```
AltaRica : 15 : Node (unit) unknown
Expr : 17 : Symbol (A.In) not found in current node
AltaRica : 20 : Defined node (Equip)
```

Errors on flow variables (flow):

- Unknown flow variable

```
domain Connect = link
  flow A,B:int;
  inverse in^C; out^A;
  assert
    out^A := in^A;
    in^B := out^B;
knil;
```

```
Link : 7 : Flow (C) unknown
AltaRica : 7 : Construct link
```

- Flow variable already declared

```
domain Connect = link
  flow A,A:int;
  inverse in^A; out^A;
  assert
    out^A := in^A;
    in^B := out^B;
knil;
```

```
Flow : 2 : Flow (A) already exists
AltaRica : 7 : Construct link
```

- Local flow variables are not allowed with structured domains having 'inverse' clauses.

```
domain Connect = link
  flow A,B:int;
  inverse in^A; out^A;
  assert
    out^A := in^A;
    in^B := out^B;
knil
```

```
node Unit
  flow
    Mem:Connect:local;
    Our:Connect:out;
  assert
    Out = Mem;
edon
```

```
Link : 12 : Flow : Inverse struct domain not allowed for local flow
Expr : 14 : Symbol (Out) not found in current node
AltaRica : 15 : Defined node (Unit)
```

Errors on state variables (state):

- State variables are not allowed with structured domains.

```
domain Connect = link
  flow A,B:int;
  inverse in^A; out^A;
  assert
    out^A := in^A;
    in^B := out^B;
knil
```

```
node Unit
  flow Out:Connect:out;
  state State:Connect;
  assert
    Out = State;
edon
```

```
Link : 11 : State : Struct domain not allowed
Expr : 13 : Symbol (State) not found in current node
AltaRica : 14 : Defined node (Unit)
```

- Unknown state variable

```
node Unit
  flow
    Out:bool:out;
    In:bool:in;
  state OK:bool;
  assert
    Out = (if OK then In else false);
  init ok := true;
edon
```

```
Init : 8 : State (ok) unknown
AltaRica : 9 : Defined node (Unit)
```

Errors on events (event):

- Unknown event

```
node Unit
  flow
    Out:bool:out;
    In:bool:in;
  state OK:bool;
  trans
    OK |- def -> OK := false;
  assert
    Out = (if OK then In else false);
  init OK := true;
edon
```

```
Tree : 7 : Event (def) unknown
AltaRica : 11 : Defined node (Unit)
```

- Event already declared

```
node Unit
  event def;
  flow
    Out:bool:out;
    In:bool:in;
  state OK:bool;
  event def;
  trans
    OK |- def -> OK := false;
  assert
```

```

    Out = (if OK then In else false);
    init OK := true;
edon

```

```

Event : 8 : Event (def) already exists
AltaRica : 13 : Defined node (Unit)

```

- Orphan event (used by none of transitions)

```

node Unit
  flow
    Out:bool:out;
    In:bool:in;
  state OK:bool;
  event def;rep;
  trans
    OK |- def -> OK := false;
  assert
    Out = (if OK then In else false);
  init OK := true;
edon

```

```

Event : 7 : Event (rep) is orphan (No transition use it).

```

Errors on sub-components (sub):

- Sub-component already declared

```

node Unit
  flow
    Out:bool:out;
    In:bool:in;
  state OK:bool;
  assert
    Out = (if OK then In else false);
edon

node Equip
  sub
    A,A:Unit;
edon

```

```

Sub : 12 : Sub (A) already exists
AltaRica : 13 : Defined node (Equip)

```

Errors on assertions/assignments (assert) :

- Always false assertion

```

node Unit
  flow True:bool:in;
  state OK:bool;
  assert
    (if OK then not(true) else false);
  init OK := true;
edon

```

```
Expr : 5 : Assert always false
AltaRica : 7 : Defined node (Unit)
```

- Assertion directly defined with function

```
func Fct
  flow
    Fct:int:out;
    Arg:int:in;
  assert
    Fct = Arg+1;
cnuf

node Unit
  flow In:bool:in;
  assert
    Fct(In);
edon
```

```
Expr : 12 : Assert not define with function
AltaRica : 13 : Defined node (Unit)
```

- Assertion without constant boolean expression

```
node Unit
  flow In:int:in;
  assert
    10 + 5;
edon
```

```
Expr : 4 : Assert with no boolean constant expression
AltaRica : 5 : Defined node (Unit)
```

Errors on expressions (assertions/guards/assignments) :

- Unknown identifier for current node (neither a variable, nor possible value of enumerate)

```
node Unit
  flow
    Out:bool:out;
    In:bool:in;
  state State:{OK,KO,SB};
  event def;rep;
  trans
    State=OK |- def -> State:=Ko;
  assert
    Out = (if State=ok then In else false);
  init State := OK;
edon
```

```
Expr : 8 : Symbol (Ko) not found in current node
AltaRica : 12 : Defined node (Unit)
```

- Non-boolean Argument(s)

```
node Unit
  flow
    Out:bool:out;
    In:bool:in;
  state State:{OK,KO,SB};
  assert
    Out = (if State then In else false);
  init State := OK;
edon
```

```
Expr : 7 : No boolean args : State
AltaRica : 9 : Defined node (Unit)
```

- Non-numeric argument(s)

```
node Unit
  flow
    Out:int:out;
    In:int:in;
  state State:{OK,KO,SB};
  assert
    Out = min(In, State);
  init State := OK;
edon
```

```
Expr : 7 : No numeric args : State
AltaRica : 9 : Defined node (Unit)
```

- Non-structured argument(s)

```
domain Connect = link
  flow A,B:int;
  assert
    in^A := out^A;
    in^B := out^B;
knil;

node Unit
  flow
    Out:bool:out;
    In1, In2:Connect:in;
  assert
    Out = (In1 != In2);
edon
```

```
Expr : 13 : No structured args : In1
AltaRica : 14 : Defined node (Unit)
```

- Equality between arguments from domains that aren't compatible

```
node Unit
  flow
    Out:bool:out;
    In1:bool:in;
    In2:int:in;
  assert
    Out = (In1 = In2);
edon
```

```
Expr : 7 : Equality args
AltaRica : 8 : Defined node (Unit)
```

- Assignment with two input flows (in = in)

```
node Unit
  flow
    Out:bool:out;
    In1, In2:int:in;
  assert
    In1 = In2;
edon
```

```
Expr : 6 : Assignment args (in = in)
AltaRica : 7 : Defined node (Unit)
```

- Assigne Assignment with enumerate domains that aren't equivalent

```
node Unit
  flow
    Out:{OK,KO,SB}:out;
    In:{OK,KO}:in;
  assert
    Out = In;
edon
```

```
Expr : 6 : Assignment enums not equivalent
AltaRica : 7 : Defined node (Unit)
```

- Assignment with two constants (cst = cst)

```
node Unit
  flow
    Out:{OK,KO,SB}:out;
    In:{OK,KO}:in;
  assert
    OK = KO;
edon
```

```
Expr : 6 : Assignment args (cst = cst)
AltaRica : 7 : Defined node (Unit)
```

- Division by zero

```
node Unit
  flow
    Out:int:out;
    In:int:in;
  state OK:bool;
  assert
    Out = (if OK then In else In/0);
edon
```

```
Expr : 7 : Division by zero
AltaRica : 8 : Defined node (Unit)
```

- Function are not allowed into transitions (guard or affectation), or into extern clause.

```
func Add
  flow
    Add:int:out;
    Arg1,Arg2:int:in;
  assert
    Add = Arg1+Arg2;
cnuf

node Unit
  flow
    Out:int:out;
    In1,In2,Chk:int:in;
  state OK:bool;
  event fail;
  trans
    OK |- fail -> OK := (Add(In1,In2)!=Chk);
  assert
    Out = OK;
edon
```

```
Expr : 16 : Function are not allowed in this context (transition or extern clause)
AltaRica : 19 : Defined node (Unit)
```

Errors on transitions (trans):

- Guard of a non-boolean transition.

```
node Unit
  flow In:int:in;
  state OK:bool;
  event chg;
  trans
    In |- chg -> OK := not(OK);
  init OK := true;
edon
```

```
Expr : 6 : Not boolean guard
AltaRica : 8 : Defined node (Unit)
```

- Guard always false

```
node Unit
  state State:{OK,KO,SB};
  event def;
  trans
    OK=KO |- def -> State := KO;
edon
```

```
Expr : 5 : Guard always false
AltaRica : 6 : Defined node (Unit)
```

- Variable already assigned in transition

```
node Unit
  state OK:bool;
  event def;
  trans
    OK |- def ->
      OK := false,
      OK := not(OK);
edon
```

```
Trans : 7 : State (OK) already assigned
AltaRica : 8 : Defined node (Unit)
```

- Domain not compatible for variable assignment

```
node Unit
  state OK:bool;
  event def;
  trans
    OK |- def -> OK := 10;
edon
```

```
Expr : 5 : Conflict domain assignment for state (OK)
AltaRica : 6 : Defined node (Unit)
```

Errors on synchronizations (sync):

- First event of a synchronization must belong to current component.

```
node Unit
  state OK:bool;
  event def;rep;
  trans
    OK |- def -> OK := false;
    not(OK) |- rep -> OK := true;
  init OK := true;
edon

node Equip
  sub A,B : Unit;
  sync <A.def ? B.def>;
edon;
```

```
Sync : 12 : First event must belong to current model
AltaRica : 13 : Defined node (Equip)
```

- Events following the first event must belong to sub-components

```
node Unit
  state OK:bool;
  event def;rep;
  trans
    OK |- def -> OK := false;
    not(OK) |- rep -> OK := true;
  init OK := true;
edon
```

```
node Equip
  sub A,B : Unit;
  event rep;
  sync <rep, A.rep, rep>;
edon;
```

```
Sync : 13 : Other event must belong to models component
AltaRica : 14 : Defined node (Equip)
```

- Inside a synchronization, two events can't belong to the same sub-component.

```
node Unit
  state OK:bool;
  event def;rep;
  trans
    OK |- def -> OK := false;
    not(OK) |- rep -> OK := true;
  init OK := true;
edon
```

```
node Equip
  sub A,B : Unit;
  event ccf;
  sync <ccf ? A.def ? A.rep>;
edon;
```

```
Sync : 13 : Two event dont must belong to same sub component
AltaRica : 14 : Defined node (Equip)
```

Errors on initializations (init):

- Impossible initialization: Usually, initial value is not compatible with the domain of state variable.

```
node Unit
  state OK:bool;
  init OK := 10;
edon
```

```
Expr : 3 : Init [OK := 10] is not possible
AltaRica : 4 : Defined node (Unit)
```

Translation into 'standard' AltaRica

AltaRica Extended language has added some constructions helping model entry. It's so useful to convert to 'standard' AltaRica.

They are tree specific constructions:

1. Structured flows: they allow easy representation of complex connection between two components.
2. Operators/Functions can be considered as components without behavior (no state variable, no event). They are used directly in component assertions.
3. Synchronizations of Common Cause Failures (CCF) enrich model. They allow to consider that many events (failures) can be fired at the same time, but without deleting each event presence (as it would be done in case of broadcast synchronizations).

If syntactic and semantic check didn't display errors, this processing should not generate errors.

Convert assertions to assignments

Conversion to Dataflow format

In AltaRica, it's possible to write assertions that are not implicitly assignments. Some users use to write assertions like `if [condition] then [var]=[value]`. It's the same thing as the following boolean implication `[condition] => ([var]=[value])`.

'AltaRica DataFlow' language view assertions as assignments on output flow variables. That is to say `out = fct(ins, states);`.

Goal is to convert usually used assertions of type implication to dataflow assignments.

In order to do that, transformation algorithm need parameters which are all component assertions to generate equivalent assignments.

This algorithm is made of two steps:

A. For each flow (local or output), it search an equation like `flow = fct(flows, states>`.

This step is made of three sub-steps:

1. Is there a dataflow equation on considered flow ?
2. Is there a dataflow equation hide in clauses `if ... then ... else ...` on considered flow ?
3. On the contrary, all clauses assigning variable are retrieved in couple (condition, assignment) and equivalent equations are generated.

Consistency and completeness of assertions are checked thanks to simple flow-simulator.

B. For every assertion:

1. It verifies there is no assignment with output-variable if at least one assertion has been generated in step A.3.
2. It verifies that there is no circular definition of variables. Assignment are tidied up to avoid ambiguities.

List of A.3 errors:

- No assignment for a given variable

```
node DataFlow
  flow
    Out:bool:out;
  state
    State:{OK,KO,SB};
  /* ... */
  assert
    Out;
edon
```

Node : 9 : DataFlow(DataFlow, Out) - No assignment operation for variable : Out

- Presence of operator avoiding processing. Inside non-dataflow-equation, the only allowed operators are : implication `if [condition] then [affects]` and test-operators `if [condition] then [affects] else [affects]`; condition is boolean expressions and assignment is conjunction (operator `&`) of assignment `[Var]=[Value]`.

```
node DataFlow
  flow
    Out:bool:out;
    In:bool:in;
    icone:[1,3]:out;
  state
    State:{OK,KO,SB};
  /* ... */
  assert
    if State=OK then icone=1 | Out=true;
    if State=KO then icone=2 | Out=false;
    if State=SB then icone=3 | Out = false;
  edon
```

```
Node : 13 : DataFlow(DataFlow, Out) - Operator not dataflowisable : ((icone = 1) or (Out = true))
```

- Output flow not connected.

```
node DataFlow
  flow
    Out:bool:out;
    In:bool:in;
    icone:[1,3]:out;
  state
    State:{OK,KO,SB};
  /* ... */
  assert
    (if State=OK then Out=In else Out=false);
  edon
```

```
Node : 11 : DataFlow(DataFlow, icone) no connected variable
```

List of B errors:

- Assignment of output flow is not allowed.

```
node DataFlow
  flow
    Out:bool:out;
    In:bool:in;
    icone:[1,3]:out;
  state
    State:{OK,KO,SB};
  /* ... */
  assert
    if State=OK then icone=1;
    if State=KO then icone=2;
    if State=SB then icone=3;
    if icone=1 then Out=In else Out=false;
  edon
```

```
Node : 14 : DataFlow(DataFlow, Out) assignment with output flow invalid
```

- At the end of 'dataflowisation' process, there are still assertions that are not taken into account. It usually comes from useless assignment of a variable.


```

node DataFlow
  flow
    Out:bool:out;
    In:bool:in;
    icone:[1,3]:out;
  state
    State:{OK,KO,SB};
  /* ... */
  assert
    if State=OK then Out=In else Out=false;
    if State=OK then icone=1;
    if State=KO then icone=2;
    if State=SB then icone=3 & Out=false;
edon

```

```

Node : 14 : DataFlow(DataFlow, ???) - Already assert exist
((State = SB) => (Out = false))

```

- Set of assignments creating a loop

```

node DataFlow
  flow
    a,b,c:bool:in;
    v,w,x,y,z:bool:out;
  /* ... */
  assert
    w = (v & b);
    x = (y & z);
    y = (w & a);
    z = (v & c);
    v = (a|b|x);
edon

```

```

Node : 12 : DataFlow(DataFlow, ???) - No DAG equation.
Current loop :
v
x
y
w
v

```

Possible errors during check with simulation of flows:

- It isn't possible to verify completeness and/or consistency of a model, if one of its input flows has an infinite domain definition (integer or float)

```

node DataFlow
  flow
    Out:bool:out;
    In:int:in;
  state
    State:{OK,KO,SB};
  /* ... */
  assert
    if State=OK & In > 10 then Out=true;
    if State=KO & In < 20 then Out=false;
    if (State=SB | In <=10 | In >=20) then Out = true;
edon

```

Node : 12 : DataFlow(DataFlow, Out) - Simul - Infinity domain from variable : In

- Simulation is only possible if component isn't too-complex. On the contrary, this simulation will take too much time and too much memory.

```
node DataFlow
  flow
    Out:bool:out;
    In1,In2,In3,In4,In5,In6:[0,9]:in;
  state
    State:{OK,KO,SB};
  /* ... */
  assert
    if State=OK & (In1+In2+In3+In4+In5+In6) > 26 then Out=true;
    if State!=OK then Out=false;
edon
```

Node : 11 : DataFlow(DataFlow, Out) - Simul - Too complex component

- La simulation a engendré une affectation d'une variable en dehors de son domaine de définition

```
node DataFlow
  flow
    Out:[1,2]:out;
  state
    State:{OK,KO,SB};
  /* ... */
  assert
    if State=OK then Out=1;
    if State=KO then Out=2;
    if State=SB then Out=3;
edon
```

Node : 11 : DataFlow(DataFlow, Out) - Simul - Affectation outside domain of definition
Value 3 (Domain [1,2]) with valuation :
State = SB

- Consistency error during simulation: a valuation of input variables can generate assignments having different values.

```
node DataFlow
  flow
    Out:bool:out;
    In:bool:in;
  state
    State:{OK,KO,SB};
  /* ... */
  assert
    if State=OK | In then Out=true;
    if State=KO & In then Out=false;
    if State=KO & not(In) then Out=true;
    if State=SB then Out=false;
edon
```

Node : 13 : DataFlow(DataFlow, Out) - Simul - Incoherence error
Value (true, false) with valuation :
State = KO

```
In = true
```

- Completeness error during simulation: there is an input variable valuation that has no assignment.

```
node DataFlow
  flow
    Out:bool:out;
    In:bool:in;
  state
    State:{OK,KO,SB};
  /* ... */
  assert
    if State=OK & In then Out=true;
    if State=KO & In then Out=false;
    if State=KO & not(In) then Out=true;
    if State=SB then Out=false;
edon
```

```
Node : 13 : DataFlow(DataFlow, Out) - Simul - Uncompleteness error
No value with valuation :
State = OK
In = false
```

Local simulation of components

Local simulation of components enables detection of potentials default like:

- transition conflicts : Two transitions - having equivalent guards and associated with the same event - may be valid simultaneously.
- variable assignment with a value from a different domain of definition.
- a too much complex component (too many state variables, too many input variable) which could generate a combinative explosion with some tools.
- A dynamic component can be an issue during some fault tree generation, and for comparisons between tree generation and sequence generation. A component is dynamic if from the same initial state, 2 event-permutations lead to 2 different states.

Principle is to do a local simulation of each component having behavior (presence of transition or state variable). This simulation assumes that component input flows have the same value during simulation time. This simulation is made for every value of state variables and input flows.

Messages associated with too complex components.

- Local simulation can't be done if a variable (state or flow) has an infinite domain (Integer or Float).

```
node Test
  flow
    I : int : in ;
  state S : bool ;
  event a;
  init S := false ;
  trans
    (I > 25) & not(S) | - a -> S := true;
edon
```

```
Node : 9 : LocalSimul(Test) - Infinity domain from variable : I
```

- Local simulation can't be done if component is too complex. Otherwise, this simulation would take too many times and memory.

```
node Test
  flow
    O : bool : out ;
    I1,I2,I3,I4,I5 : [0,9] : in ;
  state S : bool ;
  event a;
  init S := false ;
  trans
    ((I1+I2+I3+I4+I5) > 25) & not(S) | - a -> S := true;
  assert
    O = (if S then I1>I2 else I3>I4);
edon
```

```
Node : 12 : LocalSimul(Test) - Too complex component
Cardinal = 200 000 [ > 100 000]
```

- Local simulation can't be done if there are too many dependent events. Otherwise there are too many permutations, and simulation takes too many time.

```
node Test
  flow
    In : [0,9] : in ;
  state S : [0,9] ;
  event a; b; c; d; e; f; g; h;
  init S := 0 ;
  trans
    S = In+0 | - a -> S := In+1;
    S = In+1 | - b -> S := In+2;
    S = In+2 | - c -> S := In+3;
    S = In+3 | - d -> S := In+4;
    S = In+4 | - e -> S := In+5;
    S = In+5 | - f -> S := In+6;
    S = In+6 | - g -> S := In+7;
    S = In+7 | - h -> S := In+8;
edon
```

```
Node : 16 : LocalSimul(Test) - Too complex component
Cardinal = 4 032 000 [ > 100 000]
```

A fault tree is a static view of a system. During fault tree generation, we assume that generated system is static. Local simulation enables static component checking. That's to say its state doesn't depend on events order leading to it.

- Example 1

```
node Test
  state S : [0,4] ;
  event a; b;
  init S := 0 ;
  trans
    S = 0 | - a -> S := 1;
    S = 1 | - b -> S := 2;
```

```

S = 0 | - b -> S := 3;
S = 3 | - a -> S := 4;
edon

```

```

Node : 10 : LocalSimul(Test) - Final state component depend on failures order fire in scenario
MemComb(a, b) => {S=2}
CurPerm(b, a) => {S=4}
from the following conditions
S = 0

```

- Example 2

```

node Test
state S : [0,3] ;
event a; b; c;
init S := 0 ;
trans
S = 0 | - a -> S := 1;
S = 1 | - b -> S := 3;
S = 0 | - c -> S := 3;
S = 0 | - b -> S := 2;
edon

```

```

Node : 10 : LocalSimul(Test) - Final state component depend on failures order fire in scenario
MemComb(a, b) => {S=3}
CurPerm(b, a?) => {S=2}
from the following conditions
S = 0

```

Two transitions are in conflict at a given time, if they have valid guards and are associated with the same event. Transition in conflict can be detected during a local simulation.

- Transition in conflict

```

node Test
flow I : [0,4] : in;
state S : [0,4] ;
event a; b;
init S := 0 ;
trans
S = 2 | - a -> S := 3;
S = 3 & I = 0 | - b -> S := 4;
S = 3 | - b -> S := 1;
edon

```

```

Node : 10 : LocalSimul(Test) - Conflict transition fire : Seq(a, b)
from the following conditions
S = 2
I = 0

```

Lors de la simulation locale, un certain nombre d'erreur peuvent survenir comme une affectation en dehors d'un domaine de définition ou une division par zéro.

- Affectation outside domain of definition included assert

```

node Test
flow

```

```

    Out : [0,4] : out ;
    In : [0,2] : in ;
    state S : [0,3] ;
    event a; b;
    init S := 0 ;
    trans
      S = 0 | - a -> S := 1;
      S = 1 | - b -> S := 3;
      S = 0 | - b -> S := 2;
      S = 2 | - a -> S := 3;
    assert
      Out = In+S;
  edon

```

```

Node : 15 : LocalSimul(Test) - Affectation outside domain of definition
  Out : value=5, domain=[0,4]
  from the following conditions
    In = 2
    S = 3

```

- Affectation outside domain of definition included transition's affect

```

node Test
  state
    S : [0,3];
  event a; b;
  init S := 0;
  trans
    S = 0 | - a -> S := 1;
    S = 1 | - b -> S := 4;
    S = 0 | - b -> S := 2;
    S = 2 | - a -> S := 4;
  edon

```

```

Node : 11 : LocalSimul(Test) - Affectation outside domain of definition
  S : value=4, domain=[0,3]
  after sequence : Seq(a, b)
  from the following conditions
    S = 0

```

- Erreur lors de l'évaluation d'une expression (division par zéro)

```

node Test
  state
    div : [0,1];
  event dec;
  init div := 1;
  trans
    div > 0 | - dec -> div := div / (div-1);
  edon

```

```

Node : 8 : LocalSimul(Test) - Division by zero
  after sequence : Seq(dec)
  from the following conditions
    div = 1

```

Setting flat of model

To set a model flat is to remove hierarchy in order to use only one component standing for the system.

The principle consists in making instances recursively for sub-components, in order to add flow-variables, states, events, transitions, assertions and external clauses inside current model.

The only difficulty is the synchronizations processing.

Possible error during setting flat:

- Unknown event for a parent node

```
node Flat1
/* ... */
state OK:bool;
event chg;
trans
    OK      | - chg -> OK := false;
    not(OK) | - chg -> OK := true;
/* ... */
edon

node Flat2
sub U1,U2:Flat1;
event chg;
sync <chg, U1.chg, U2.chg>;
/* ... */
edon

node Flat3
sub
    U:Flat1;
    E:Flat2;
event chg;
sync <chg, U.chg, E.U1.chg>;
/* ... */
edon;
```

```
Node : 25 : Flatness Event (E.U1.chg) unknown for node (Flat3)
Node : 25 : Attach data
```

Indeed, model is set flat recursively, the component Flat2 will be set flat before component Flat3.

During Flat2 setting flat, the events U1.chg and U2.chg will be replaced by chg.

During Flat3 setting Flat, during processing of <chg, U.chg, E.U1.chg> synchronization, E.U1.chg event is searched, but it has disappeared and has been replaced by E.chg, that's why there is an error.

- Too complex component - Too high number of transition having to be generated.

```
node Flat1
/* ... */
state OK:[0,9];
event chg;
trans
    OK = 0 | - chg -> OK := 1;
    OK = 1 | - chg -> OK := 2;
    OK = 2 | - chg -> OK := 3;
    OK = 3 | - chg -> OK := 4;
    OK = 4 | - chg -> OK := 5;
    OK = 5 | - chg -> OK := 6;
    OK = 6 | - chg -> OK := 7;
    OK = 7 | - chg -> OK := 8;
```

```

    OK = 8 | - chg -> OK := 9;
    OK = 9 | - chg -> OK := 0;
/* ... */
edon

node Flat2
sub
    U1,U2,U3,U4:Flat1;
event chg;
sync <chg, U1.chg, U2.chg, U3.chg, U4.chg>;
/* ... */
edon;

```

```

Sync : 23 : Flatness Sync (chg) for node (Flat2)
generate so large number of transition (10 000)
Sync : 23 : Attach data

```

Properties control

Properties control consists in a local properties checking on each component or an overall properties checking on the flat model.

These properties aren't considered as AltaRica errors. Nevertheless, some tools can have difficulties to process AltaRica models having properties like float variables or looped assertions.

This processing displays possible errors/issues.

Control of 'parameter' external clause

'parameter' external clause allows to define law parameters that can be used in external clauses. These parameters are named either in a overall way or with a clause <local ID>.

Overall syntax for this external clause is: `parameter [ID] = [param]; avec [param] ::= [FLOAT] | [ID] | [FCT]([param]+)` where [ID] are identifiers, [FLOAT] is a float and [FCT] is a incertitude-propagation-law (also called propagation-law or incertitude-law) among {lognormal, uniform, normal}.

A parameter is either a float (parameter value), or a name referencing a named parameter, or an incertitude-law. In the last case, parameter of the incertitude-law can't be defined with an incertitude-law.

Possible errors are:

- Syntax error in parameter statement

```

node main
state OK:bool ;
event def; rep;
trans
    OK | - def -> OK := false;
    not(OK) | - rep -> OK := true;
init OK := true;
extern
    parameter {lbd,mu} = 1e-3;
    law <event def> = exponential(lbd);
    law <event rep> = exponential(mu);
edon;

```

```

ExternParameter : 9 : file : Syntax error for 'parameter' clause :
parameter [id] = [param];
=> parameter {lbd, mu} = 0.0010

```


- Syntax error in parameter definition

```
node main
  state OK:bool ;
  event def ;
  trans OK |- def -> OK := false;
  init OK := true;
  extern
    parameter lbd = "1e-3";
    law <event def> = exponential(lbd);
edon;
```

```
ExternParameter : 7 : file : Syntax error for term [param] used in 'law' or 'parameter' clause
:
[param] ::= [float] | [id] | fct([param]+);
=> parameter lbd = "1e-3"
```

- Unknown incertitude-function

```
node main
  state OK:bool ;
  event def;
  trans OK |- def -> OK := false;
  init OK := true;
  extern
    parameter lbd = lognormale(1e-3, 3);
    law <event def> = exponential(lbd);
edon;
```

```
ExternParameter : 7 : file : Unknown propagation function for [param] () :
fct = {lognormal|uniform|normal}
=> parameter lbd = lognormale(0.0010, 3)
```

- Recursive incertitude-function

```
node main
  state OK:bool ;
  event def;
  trans OK |- def -> OK := false;
  init OK := true;
  extern
    parameter lbd = lognormal(uniform(0.8e-3, 1.2e-3), 3);
    law <event def> = exponential(lbd);
edon;
```

```
ExternParameter : 7 : file : Recursive propagation functions are forbidden ...
=> parameter lbd = lognormal(uniform(8.0E-4, 0.0012), 3)
```

Control of 'law' external clause

The 'law' external clause allows to define delay and/or probability laws associated with events of model.

Overall syntax for this external clause is: `law <event [ID-EVT]> = [FCT]([param]+);` where [ID-EVT] is an event identifier and [FCT] is a recognized probability function (see previous paragraph).

Possible errors:

- Syntax error in law definition

```
node main
  state OK:bool ;
  event def;
  trans OK |- def -> OK := false;
  init OK := true;
  extern
    law <event def> = 1e-3;
edon;
```

```
ExternLaw : 7 : file : Syntax error for 'law' clause :
  law <event [id]> = fct([param]+>);
=> law <event def> = 0.0010
```

- Unknown law

```
node main
  state OK:bool ;
  event def;
  trans OK |- def -> OK := false;
  init OK := true;
  extern
    law <event def> = dirac(1e-3);
edon;
```

```
ExternLaw : 7 : file : Unknown function for 'law' clause : must be a known law for Aralia or
Mocal2 compute engine
=> law <event def> = dirac(0.0010)
```

- Managed Aralia laws are: exponential, constant, Weibull, Dirac, GLM, asymptotic_exponential, periodic_test.

```
node main
  state OK:bool ;
  event def;
  trans OK |- def -> OK := false;
  init OK := true;
  extern
    law <event def> = ifa(10, 100);
edon;
```

```
ExternLawAralia : 7 : file : Unknown law for Aralia
=> law <event def> = ifa(10, 100)
```

- Managed Mocal2 laws are: exponential, constant, Weibull, Dirac, ifa (planned instant), nlog, unif.

```
node main
  state OK:bool ;
  event def;
  trans OK |- def -> OK := false;
  init OK := true;
  extern
    law <event def> = GLM(0, 1e-3, 1e-2);
edon;
```

```
ExternLawMoca : 7 : file : Unknown law for Mocal2
=> law <event def> = GLM(0, 0.0010, 0.01)
```

- Number of parameters is defines for each law.

```
node main
  state OK:bool ;
  event def;
  trans OK |- def -> OK := false;
  init OK := true;
  extern
    law <event def> = exponential(1e-3, 1e-2);
edon;
```

```
ExternParameter : 7 : file : The number of law parameters isn't correct.
=> law <event def> = exponential(0.0010, 0.01)
```

Control of 'attribute' external clause

The 'attribute' external clause allows to associate attributes to events.

Overall syntax for this external clause is: `attribute [ID-ATTR](<event [ID-EVT]>) = [value];` where [ID-ATTR] is name of attribute, [ID-EVT] an event identifier and [value] is value of the attribute for the considered event.

Possible errors:

- Syntax error

```
node main
  state OK:bool ;
  event def;
  trans OK |- def -> OK := false;
  init OK := true;
  extern
    law <event def> = exponential(1e-3);
    attribute Type(<event def>) = pompe(2);
edon;
```

```
ExternAttribute : 8 : file : Syntax error for 'attribute' clause : [value]
attribute [name](<event [id]>) = [value];
=> attribute Type(<event def>) = pompe(2)
```

Control of 'nodeproperty' external clause

The 'nodeproperty' external clause allows to associate properties to components (nodes).

Overall syntax for this external clause is: `nodeproperty [ID] = [value];` where [ID] is name of property and [value] is value of the property.

```
node main
  state OK:bool ;
  extern
    nodeproperty date = format("2007/06/12");
```

```
edon;
```

```
ExternNodeProperty : 4 : file : Syntax error for 'nodeproperty' clause :
  nodeproperty [id] = [value];
  => nodeproperty date = format("2007/06/12")
```

Control of 'priority' external clause

The 'priority' external clause allows to define priority between events.

Overall syntax for this clause is: `priority <event [ID-EVT]> = [INT];` where [ID-EVT] is an event identifier and [INT] a non negative integer specifying priority level (the higher the number, the higher the event priority).

Possible errors:

- syntax error

```
node main
  flow In:bool:in;
  state Mem:bool ;
  event chg;
  trans Mem!=In |- chg -> Mem := In;
  init Mem := true;
  extern
    law <event chg> = Dirac(0);
    priority <event chg> = High;
edon;
```

```
ExternPriority : 9 : file : Syntax error for 'priority' clause :
  priority <event [id]> = [integer];
  => priority <event chg> = High
```

- Only instantaneous events can have priority.

```
node main
  flow In:bool:in;
  state Mem:bool ;
  event chg;
  trans Mem!=In |- chg -> Mem := In;
  init Mem := true;
  extern
    law <event chg> = exponential(0.001);
    priority <event chg> = 1;
edon;
```

```
ExternPriorityDirac : 8 : file : A untemporised event (chg) can not have a priority.
  => law <event chg> = exponential(0.0010)
```

Control of 'remark' external clause

The 'remark' external clause allows to document events (flow variables, state variables events, sub-components, local parameters) of an Altarica model.

Overall syntax for this clause is: `remark [OBJ] = [STRING];` where [OBJ] is an element of the model. (`<event [ID]>|<flow [ID]>|<state [ID]>|<sub [ID]>|<local [ID]>`) and [STRING] is string (between quotation marks).

Possible errors:

- Syntax error

```
node main
  state OK:bool ;
  event def;
  trans OK |- def -> OK := false;
  extern
    remark <event def> = defaillance;
edon;
```

```
ExternRemark : 6 : file : Syntax error for 'remark' clause :
  remark [obj] = "<String>"; with [obj] ::= <event [id]>|<flow [id]>|<state [id]>|<sub
[id]>|<local [id]>
=> remark <event def> = defaillance
```

Control of 'preemptible' external clause

The 'preemptible' external clause allows to define events such as preemptible events.

Overall syntax for this clause is: preemptible { (<event [ID-EVT]>)+ }; where [ID-EVT] are events.

```
node main
  flow Call:bool:in;
  state St:{OK,KO,SB} ;
  event Def; Rep; SOK; SSB;
  trans
    St=OK          |- Def -> St := KO;
    St=KO          |- Rep -> St := SB;
    St=SB & Call   |- SOK -> St := OK;
    St=OK & ~Call  |- SSB -> St := SB;
  extern
    law <event SOK> = Dirac(0);
    law <event SSB> = Dirac(0);
    preemptible {<event Rep>} = false;
edon;
```

```
ExternPreemptible : 13 : file : Syntax error for 'preemptible' clause :
  preemptible '{' (<event [id]>)+ '}'
=> preemptible {<event Rep>} = false
```

Control of 'bucket' external clause

The 'bucket' external clause allows to associate events in order to consider that there are disjunctive in probability (can't appear at the same time). It allows to imitate trigger with realistic solicitation.

Overall syntax for this clause is: bucket { (<event [ID-EVT]>)+ }; where [ID-EVT] are events.

Possible errors:

- Syntax error

```
node main
  flow Call:bool:in;
  state St:{OK,KO,SB} ;
  event Def; Rep; SOK; SKO; SSB;
  trans
```

```

St=OK          |- Def -> St := KO;
St=KO          |- Rep -> St := SB;
St=SB & Call   |- SOK -> St := OK;
St=SB & Call   |- SKO -> St := KO;
St=OK & ~Call  |- SSB -> St := SB;
extern
  law <event Def> = exponential(0.001);
  law <event Rep> = exponential(0.01);
  law <event SOK> = constant(0.98);
  law <event SKO> = constant(0.02);
  law <event SSB> = Dirac(0);
  bucket {<event SOK>, <event SKO>} = false;
edon;

```

ExternBucket : 17 : file : Syntax error for 'bucket' clause :
 bucket '{' (<event [id]>)+ '}'
 => bucket {<event SOK>, <event SKO>} = false

- Each event must be associated with one and only one transition.

```

node main
  flow Call:bool:in;
  state St:{OK,KO,SB} ;
  event Def; Rep; SOK; SKO;
  trans
    St=OK          |- Def -> St := KO;
    St=KO          |- Rep -> St := SB;
    St=SB & Call   |- SOK -> St := OK;
    St=SB & Call   |- SKO -> St := KO;
    St=OK & ~Call  |- SOK -> St := SB;
  extern
    law <event Def> = exponential(0.001);
    law <event Rep> = exponential(0.01);
    law <event SOK> = constant(0.98);
    law <event SKO> = constant(0.02);
    bucket {<event SOK>, <event SKO>} = true;
edon;

```

ExternBucket : 16 : file : The event (SOK) is used at different transition.
 => bucket {<event SOK>, <event SKO>} = true

- The guards of transition must be equal.

```

node main
  flow Call:bool:in;
  state St:{OK,KO,SB} ;
  event Def; Rep; SOK; SKO; SSB;
  trans
    St=OK          |- Def -> St := KO;
    St=KO          |- Rep -> St := SB;
    St=SB & Call   |- SOK -> St := OK;
    St=KO & Call   |- SKO -> St := KO;
    St=OK & ~Call  |- SSB -> St := SB;
  extern
    law <event Def> = exponential(0.001);
    law <event Rep> = exponential(0.01);
    law <event SOK> = constant(0.98);
    law <event SKO> = constant(0.02);
    bucket {<event SOK>, <event SKO>} = true;
edon;

```

```
[warning] : 9 : file : Transition guards of events (SOK, SKO) are not equal (exactly).
=> ((St = KO) and Call) |- SKO -> St := KO
```

- Law associated to each event must be a constant law with parameter between 0 and 1.

```
node main
  flow Call:bool:in;
  state St:{OK,KO,SB} ;
  event Def; Rep; SOK; SKO; SSB;
  trans
    St=OK      |- Def -> St := KO;
    St=KO      |- Rep -> St := SB;
    St=SB & Call |- SOK -> St := OK;
    St=SB & Call |- SKO -> St := KO;
    St=OK & ~Call |- SSB -> St := SB;
  extern
    law <event Def> = exponential(0.001);
    law <event Rep> = exponential(0.01);
    law <event SOK> = constant(2);
    law <event SKO> = constant(0.02);
    law <event SSB> = Dirac(0);
    bucket {<event SOK>, <event SKO>} = true;
  edon;
```

```
ExternBucket : 8 : file : Gamma not between 0 and 1 : Event (SOK)
=> ((St = SB) and Call) |- SOK -> St := OK
```

- The sum of parameters for event-laws must be equal to 1.

```
node main
  flow Call:bool:in;
  state St:{OK,KO,SB} ;
  event Def; Rep; SOK; SKO; SSB;
  trans
    St=OK      |- Def -> St := KO;
    St=KO      |- Rep -> St := SB;
    St=SB & Call |- SOK -> St := OK;
    St=SB & Call |- SKO -> St := KO;
    St=OK & ~Call |- SSB -> St := SB;
  extern
    law <event Def> = exponential(0.001);
    law <event Rep> = exponential(0.01);
    law <event SOK> = constant(0.998);
    law <event SKO> = constant(0.02);
    law <event SSB> = Dirac(0);
    bucket {<event SOK>, <event SKO>} = true;
  edon;
```

```
ExternBucket : 17 : file : Sum of gamma not equal 1.
=> bucket {<event SOK>, <event SKO>} = true
```

Control of 'observer' external clause

The 'observer' external clause allowed to define statistic observers that can be use with Combava stochastic simulator and with Moca12.

The 'property' and 'predicate' external clauses replace it.

A 'deprecated' message is display if this clause is used.

```
node main
  state St:int;
  event chg;
  trans
    St >= 0 | - chg -> St := St*4/5;
  init St := 100;
  extern
    observer EndValueOfSt = <term (St)>;
edon;
```

```
ExternObserver : 8 : file : 'observer' clause is deprecated.
=> observer EndValueOfSt = <term (St)>
```

Control of 'predicate' external clause

The 'predicate' external clause allows to define boolean observers that can be use with Combava tools and with Moca12.

Right syntax for this external clause is: `predicate [ID] = <term ([boolean-term])>` where [ID] is an identifier and [boolean-term] is an AltaRica boolean expression between brackets.

An error is display if expression is not boolean.

```
node main
  state cpt:int ;
  extern
    predicate failed = <term (cpt*2)>;
edon;
```

```
ExternPredicate : 4 : file : Syntax error for 'predicate' clause :
  predicate [id] = <term ([boolean-term])>;
=> predicate failed = <term ((cpt * 2))>
```

Control of 'property' external clause

The 'property' external clause allows to define numeric (integer ou real) observers that can be use with Combava tools and with Moca12.

Right syntax for this external clause is: `property [ID] = <term ([numeric-term])>` where [ID] is an identifier and [numeric-term] is an AltaRica numeric expression between brackets.

An error is display if expression is not numeric.

```
node main
  state S : {ok, ko, hs} ;
  extern
    property failed = <term (S)>;
edon;
```

```
ExternProperty : 4 : file : Syntax error for 'property' clause :
  property [id] = <term ([numeric-term])>;
=> property failed = <term (S)>
```

Other controls

Instead of `<event [ID-EVT]>`, there usually can be a list of events. External clause is defined for every event in the list. In this case, the list mustn't be empty and mustn't have double (two times the same event).

List of possible errors in a definition of set of events.

- Empty list

```
node main
  state OK:bool;
  event def;
  trans OK |- def -> OK := false;
  extern
    law def = exponential(0.001);
edon;
```

```
ExternLaw : 6 : file : No define event for current clause
=> law def = exponential(0.0010)
```

- List with double

```
node main
  state OK:bool;
  event def;
  trans OK |- def -> OK := false;
  extern
    law {<event def>, <event def>} = exponential(0.001);
edon;
```

```
[warning] : 6 : file : Clause set with redefine event.
=> law {<event def>, <event def>} = exponential(0.0010)
```

AltaRica event are either instantaneous or temporized or stochastic. Tools may not manage instantaneous or temporized events.

- Some tools may not manage instantaneous events.

```
node main
  flow In:bool:in;
  state OK:bool;
  event chg;
  trans OK & In |- chg -> OK := false;
  extern law <event chg > = Dirac(0);
edon;
```

```
EventInstantaneous : 6 : file : Event (chg) has instantaneous (Dirac(0)).
=> law <event chg> = Dirac(0)
```

- Some tools may not manage temporized events.

```
node main
  flow In:bool:in;
  state OK:bool;
  event chg;
  trans OK & In |- chg -> OK := false;
  extern law <event chg > = Dirac(10);
edon;
```

```
EventTemporised : 6 : file : Event (chg) has time delay (Dirac(x)).
```

```
=> law <event chg> = Dirac(10)
```

- Some tools consider that events without law are instantaneous events. Existence of events without law must be checked.

```
node main
  state OK:bool;
  event def;
  trans OK |- def -> OK := false;
edon;
```

```
EventLaw : 4 : file : Event (def) has nothing define law.
=> def
```

Possible errors on guards and transitions:

- FaultTree generation with inference engine isn't safe when there are flows in guard of a transition.

```
node main
  flow In:bool:in;
  state OK:bool;
  event def;
  trans OK&In |- def -> OK := false;
edon;
```

```
GuardWithFlow : 5 : file : Guard of transition with flow variable (In).
=> (OK and In) |- def -> OK := false
```

- Besides this particular cases, it is inadvisable to have transitions that are always valid. In the case below, when failure happens, event must be no more fireable. The guard of this transition have to be modified.

```
node main
  state OK:bool;
  event def;
  trans true |- def -> OK := false;
edon;
```

```
GuardTrue : 4 : file : Always valid transition (guard always true)
=> true |- def -> OK := false
```

- Two transitions - having equivalent guards and associated with the same event - may be in conflict.

Because they have equivalent guards, they always be valid in the same time. Because they are associated with the same event, they will be fireable in the same time.

Currently, two guards are considered as equivalent if they are strictly equal (same order in arguments of operators) $A+B+C$ doesn't equals $C+B+A$.

```
node main
  state Etat:{Ouvert,Ferme};
  flow CC:bool:in;
  event chg;
  trans
```

```
Etat=Ouvert & CC |- chg -> Etat:=Ferme;
Etat=Ouvert & CC |- chg -> Etat:=Ouvert;
edon
```

```
TransConflict : 7 : file : Warring transitions (same guard, same event) for (chg) event.
=> ((Etat = Ouvert) and CC) |- chg -> Etat := Ouvert
```

Some type of synchronization are not fully compatible with some tools. For example, tree generation of type: inference engine, works properly only with CCF synchronization. In addition, it is possible to use synchronization (other CCF type) with events belong to same sub component. This synchronization can generate affectation conflict (Two transition who affect some state variable with different value).

- Presence of Synchronization of type "synchronization".

```
node Unit
  event def;
  state OK:bool;
  init OK := true;
  trans
    OK |- def -> OK := false;
edon;

node main
  sub A,B:Unit;
  event synk;
  sync <synk, A.def, B.def> ;
edon;
```

```
SyncSync : 12 : file : Synchronization type of synk is Synchronization.
=> <synk , A.def , B.def>
```

- Presence of synchronization of type Diffusion (BroadCast)

```
node Unit
  event def;
  state OK:bool;
  init OK := true;
  trans
    OK |- def -> OK := false;
edon;

node main
  sub A,B:Unit;
  event synk;
  sync <synk | A.def | B.def> ;
edon;
```

```
SyncDiff : 12 : file : Synchronization type of synk is Diffusion (BroadCast).
=> <synk : A.def or B.def>
```

- Presence of synchronization of type CCF (Common Cause Failure)

```
node Unit
  event def;
  state OK:bool;
  init OK := true;
  trans
```

```

    OK |- def -> OK := false;
edon;

node main
  sub A,B:Unit;
  event synk;
  sync <synk ? A.def ? B.def> ;
edon;

```

SyncCCF : 12 : file : Synchronization type of synk is CCF (Common Cause Failure).
=> <synk : A.def or B.def>

- Presence of synchronization with events belong to same sub component

```

node Unit
  event chg1; chg2;
  flow I:bool:in;
  state
    OK:bool;
    Mem:[0,2];
  init
    OK := true;
    Mem := 0;
  trans
    OK & I |- chg1 -> OK := false, Mem := 1;
    OK & ~I |- chg2 -> OK := false, Mem := 2;
edon;

node main
  sub A,B:Unit;
  event synk;
  sync <synk | A.chg1 | B.chg1 | A.chg2 | B.chg2> ;
edon;

```

SyncSomeSub : 18 : file : Synchronization synk with events (A.chg1, A.chg2) belong to same sub component.
=> <synk : A.chg1 or B.chg1 or A.chg2 or B.chg2>

Most step by step simulator and some tools can't manage systems with loops in their assertions.

- Loop presence in assertions

```

node Unit
  flow
    Out:bool:out;
    In :bool:in;
  event def;
  state OK:bool;
  init OK := true;
  trans
    OK |- def -> OK := false;
  assert
    Out = (if OK then In else false);
edon;

node main
  sub A,B:Unit;
  assert
    B.In = A.Out;
    A.In = B.Out;
edon;

```

```

Loop : 3 : file=>Instance : Loop assert : A.Out
  <= A.In
  <= B.Out
  <= B.In
  <= A.Out
  => A.Out:bool:out

```

Altatica code entered by user can be too complex to be generated into a compilable Java language. Actually, in some cases, the generated java code contains too large methods, so Java compiler can not manage compilation.

- Code généré java trop important pour être compilable

```

node complex
flow
  Ssw:bool:in;
  FmAct:bool:in;
  DefRv:bool:in;
  In1:bool:in;
  In2:bool:in;
  In3:bool:in;
  In4:bool:in;
  Val1:bool:in;
  Val2:bool:in;
  Val3:bool:in;
  Val4:bool:in;
  Rv:bool:out;
assert
  Rv = case {
    (((Val1 and Val2) and Val3) and Val4) and (((In1 and In2) and In3) and In4)) : true,
    (((Val1 and Val2) and Val3) and Val4) and (not (((In1 and In2) and In3) and In4)) and
    (((In1 and In2) and In3) and (not In4))) : true,
    (((Val1 and Val2) and Val3) and Val4) and (not (((In1 and In2) and In3) and In4)) and (not
    (((In1 and In2) and In3) and (not In4))) and (((In1 and In2) and (not In3)) and In4)) : true,
    (((Val1 and Val2) and Val3) and Val4) and (not (((In1 and In2) and In3) and In4)) and (not
    (((In1 and In2) and In3) and (not In4))) and (not (((In1 and In2) and (not In3)) and In4))
    and (((In1 and In2) and (not In3)) and (not In4)) and (not FmAct)) : DefRv,
    (((Val1 and Val2) and Val3) and Val4) and (not (((In1 and In2) and In3) and In4)) and (not
    (((In1 and In2) and In3) and (not In4))) and (not (((In1 and In2) and (not In3)) and In4))
    and (((In1 and In2) and (not In3)) and (not In4)) and FmAct) : true,
    (((Val1 and Val2) and Val3) and Val4) and (not (((In1 and In2) and In3) and In4)) and (not
    (((In1 and In2) and In3) and (not In4))) and (not (((In1 and In2) and (not In3)) and In4))
    and (not (((In1 and In2) and (not In3)) and (not In4))) and (((In1 and (not In2)) and In3)
    and In4)) : true,
    (((Val1 and Val2) and Val3) and Val4) and (not (((In1 and In2) and In3) and In4)) and (not
    (((In1 and In2) and In3) and (not In4))) and (not (((In1 and In2) and (not In3)) and In4))
    and (not (((In1 and In2) and (not In3)) and (not In4))) and (not (((In1 and (not In2)) and
    In3) and In4)) and (((In1 and (not In2)) and In3) and (not In4)) and (not FmAct)) : DefRv,
    ((not (((Val1 and Val2) and Val3) and Val4)) and (not (((Val1 and Val2) and Val3) and (not
    Val4))) and (not (((Val1 and Val2) and (not Val3)) and Val4)) and (((Val1 and (not Val2))
    and Val3) and Val4) and (not ((In1 and In3) and In4)) and (not ((In1 and In3) and (not In4)))
    and (not ((In1 and (not In3)) and In4)) and ((In1 and (not In3)) and (not In4)) and FmAct and
    Ssw) : In1,
    ((not (((Val1 and Val2) and Val3) and Val4)) and (not (((Val1 and Val2) and Val3) and (not
    Val4))) and (not (((Val1 and Val2) and (not Val3)) and Val4)) and (((Val1 and (not Val2))
    and Val3) and Val4) and (not ((In1 and In3) and In4)) and (not ((In1 and In3) and (not In4)))
    and (not ((In1 and (not In3)) and In4)) and ((In1 and (not In3)) and (not In4)) and FmAct and
    (not Ssw)) : DefRv,
    ((not (((Val1 and Val2) and Val3) and Val4)) and (not (((Val1 and Val2) and Val3) and (not
    Val4))) and (not (((Val1 and Val2) and (not Val3)) and Val4)) and (((Val1 and (not Val2))
    and Val3) and Val4) and (not ((In1 and In3) and In4)) and (not ((In1 and In3) and (not In4)))
    and (not ((In1 and (not In3)) and In4)) and (not ((In1 and (not In3)) and (not In4))) and (((not
    In1) and In3) and In4)) : true,

```

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

```

and (not Val4))) and (not (((Val1 and (not Val2)) and (not Val3)) and Val4)) and (not (((not
Val1) and Val2) and Val3)) and (not (((not Val1) and Val2) and (not Val3)) and Val4)) and
(not (((not Val1) and (not Val2)) and Val3) and Val4)) and (not (((Val1 and (not Val2))
and (not Val3)) and (not Val4))) and (not (((not Val1) and Val2) and (not Val3)) and (not
Val4))) and (((not Val1) and (not Val2)) and Val3) and (not Val4)) and (not Ssw) : DefRv,
  ((not (((Val1 and Val2) and Val3) and Val4)) and (not (((Val1 and Val2) and Val3) and (not
Val4))) and (not (((Val1 and Val2) and (not Val3)) and Val4)) and (not (((Val1 and (not
Val2)) and Val3) and Val4)) and (not (((not Val1) and Val2) and Val3) and Val4)) and (not
(((Val1 and Val2) and (not Val3)) and (not Val4))) and (not (((Val1 and (not Val2)) and Val3)
and (not Val4))) and (not (((Val1 and (not Val2)) and (not Val3)) and Val4)) and (not (((not
Val1) and Val2) and Val3)) and (not (((not Val1) and Val2) and (not Val3)) and Val4)) and
(not (((not Val1) and (not Val2)) and Val3) and Val4)) and (not (((Val1 and (not Val2))
and (not Val3)) and (not Val4))) and (not (((not Val1) and Val2) and (not Val3)) and (not
Val4))) and (not (((not Val1) and (not Val2)) and Val3) and (not Val4))) and (((not Val1)
and (not Val2)) and (not Val3)) and Val4) and Ssw) : In4,
  ((not (((Val1 and Val2) and Val3) and Val4)) and (not (((Val1 and Val2) and Val3) and (not
Val4))) and (not (((Val1 and Val2) and (not Val3)) and Val4)) and (not (((Val1 and (not
Val2)) and Val3) and Val4)) and (not (((not Val1) and Val2) and Val3) and Val4)) and (not
(((Val1 and Val2) and (not Val3)) and (not Val4))) and (not (((Val1 and (not Val2)) and Val3)
and (not Val4))) and (not (((Val1 and (not Val2)) and (not Val3)) and Val4)) and (not (((not
Val1) and Val2) and Val3)) and (not (((not Val1) and Val2) and (not Val3)) and Val4)) and
(not (((not Val1) and (not Val2)) and Val3) and Val4)) and (not (((Val1 and (not Val2))
and (not Val3)) and (not Val4))) and (not (((not Val1) and Val2) and (not Val3)) and (not
Val4))) and (not (((not Val1) and (not Val2)) and Val3) and (not Val4))) and (((not Val1)
and (not Val2)) and (not Val3)) and Val4) and (not Ssw) : DefRv,
  else DefRv
};
edon

```

```

GenerateJaval : 103 : file : Likely error during Java compilation
The 'assert' AltaRica Code of 'complex' component is too big.
=> node complex
...
edon

```

Currently, float/integer variables and some operators are not always supported by Altarica model processing tools.

- Integer variable presence

```

node Expr
  flow
    Out:int:out;
  event def;
  state Prod:int;
  init Prod := 100;
  trans
    Prod>0 |- def -> Prod := 0;
  assert
    Out = Prod;
edon;

```

```

ExprInt : 3 : file : Variable with integer domain : Out
=> Out:int:out
ExprInt : 5 : file : Variable with integer domain : Prod
=> Prod:int

```

- Float variable presence

```

node Expr
  flow
    Out:float:out;

```

```

event def;
state Prod:float;
init Prod := 100;
trans
  Prod>0 |- def -> Prod := 0;
assert
  Out = Prod;
edon;

```

```

ExprFloat : 3 : file : Variable with float domain : Out
=> Out:float:out
ExprFloat : 5 : file : Variable with float domain : Prod
=> Prod:float

```

- Presence of unwanted operators

```

node KOf3
  flow
    Out:bool:out;
    In1,In2,In3:bool:in;
  state K:[1,3];
  init K := 2;
  assert
    Out = #(In1,In2,In3)>=K;
edon;

```

```

ExprCrd : 8 : file : Operator of type : #(...)
=> #(In1, In2, In3)

```