Inference engine

Stepper & Fault-Tree generator Plugins

Version 1.0

DASSAULT SYSTEMES

Abstract

Inference engine has been the first fully operational simulator and fault-tree-generator of BPA DAS. It was made from two works:

- A step by step simulator dedicated to AltaRica made by LaBRI for Dassault Aviation. This simulator contained syntactic and semantic check of AltaRica models, models setting flat, and a simulator based upon a constraints solver.
- Inference rules engine of *Fiabex* workshop, which allows step by step simulation and tree generation.

Sylvain Lajeunesse developed this inference engine adding functionalities in order to verify completeness and consistency of models. He has openned a new way for minimal cuts computing and Sequences generation.

This chapter aims at showing different functionalities of inference engine inside BPA DAS.

Table of Contents

Introduction

Overall principle

An inference engine is a software tool that emulates the human capability to draw conclusions from a set of hypothesis. It's also artificial intelligence principle. The heart of BPA DAS's inference engine is built on this principle. Inference engine input is a set of normalised Horn clauses.

Horn clause format is:

```
Condition -> Conclusion
 where -> is the implication operator.
  Condition is a conjunction of n operations : (O1, O2, ..., On).
  Each Oi is a simple comparison operation between 2 variables or constants.
  Conclusion is a simple assignment of variables relative to
  either a variable or a constant.
```

It's equivalent to

```
if Condition then Conclusion
```

Set of Horn clauses is obtained with a set of elementary simplification from assertion and transition.

1. Transformation of events into boolean variables, and transitions into equivalent assertions
2. Creation of local variables for complex assignments
3. Transformation of If...Then...Else... into Horn clauses
4. Processing of negations
5. Transformation of conditions into minimal cartesian products
6. Simplification of Horn clauses having disjunctions on condition part.
7. Simplification of Horn clauses having multiple conclusions.

At the end of the transforation process, inference engine manages a data structure based on three sets:

- A set of possible value domains
- A set of variables
- A set of normalized Horn clauses

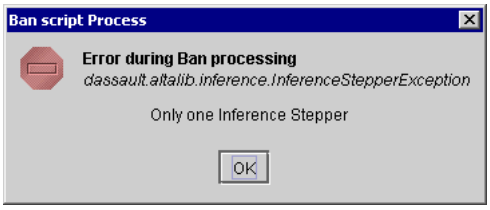
Then, inference engine will use this structure for managing simulation and algorithms (Tree generation, consistency-control, ...)

Limits/Differences

Currently, inference engine of BPA DAS can't manage discrete domains of variables. Integers, Floats, and all arithmetic operators (+, -, *, ...) generate syntax error during opening of models containing this functionnalities.

As explained in previous paragraph, inference engine considers component events as boolean variables. When an event is fired, the variable is set to true and is reset to false instaneously. It means it's not the transition which is fired, but event associated to transition, contrary to Altarica semantic. It generates behavior differences, with models having an event associated to many transitions whose guards aren't exclusive. This specificity simplifies the setting flat of broadcast synchronisations.

Inference engine is developed in C++. It's linked to BPA DAS (developed in Java) thanks to JNI. Some unstabilities in multi-thread mode has dragged a lock type architecture along. It implies it isn't possible to use two functionalities of inference engine. A step by step simulation with a FaultTree generation will display an error message:



Syntactical check (Inference)

Syntactical check verifies that component, equipment and/or system is consistent with AltaRica syntax which is limited for inference engine.

This fonction will display error if AltaRica code generated by software (for flows, states, events,...) or typed by user (guards, transitions, ...) isn't consistent with either *AltaRica syntax* or *specific limitations of inference engine*.

Inference engine only use a sub-language of AltaRica language.

Main limitations are:

- No integer or float domain (domain that could be infinite)
- No numerical operator (+, -, *, /, %, ...)

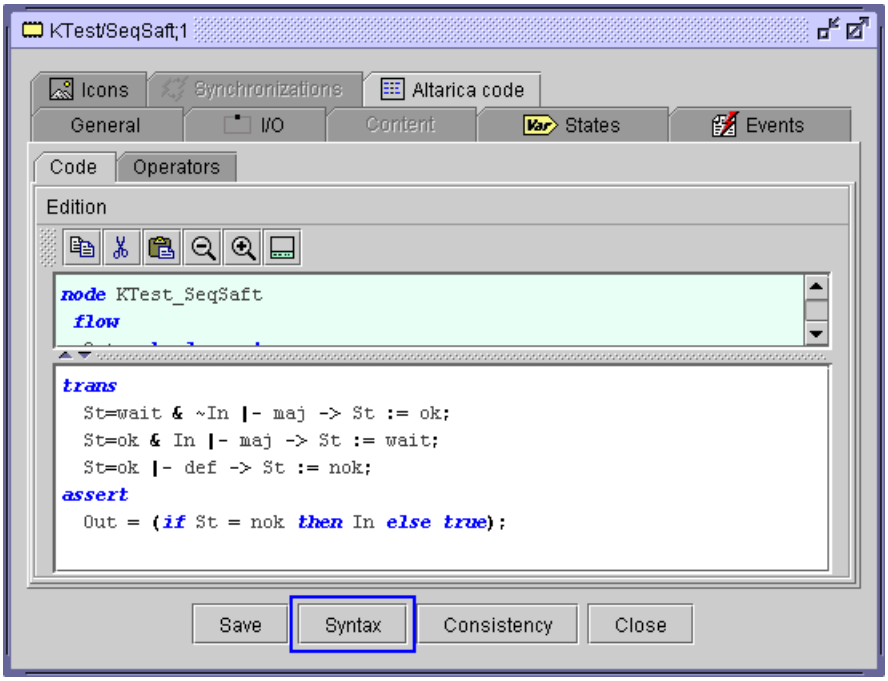
Syntactical check launching

For sytems

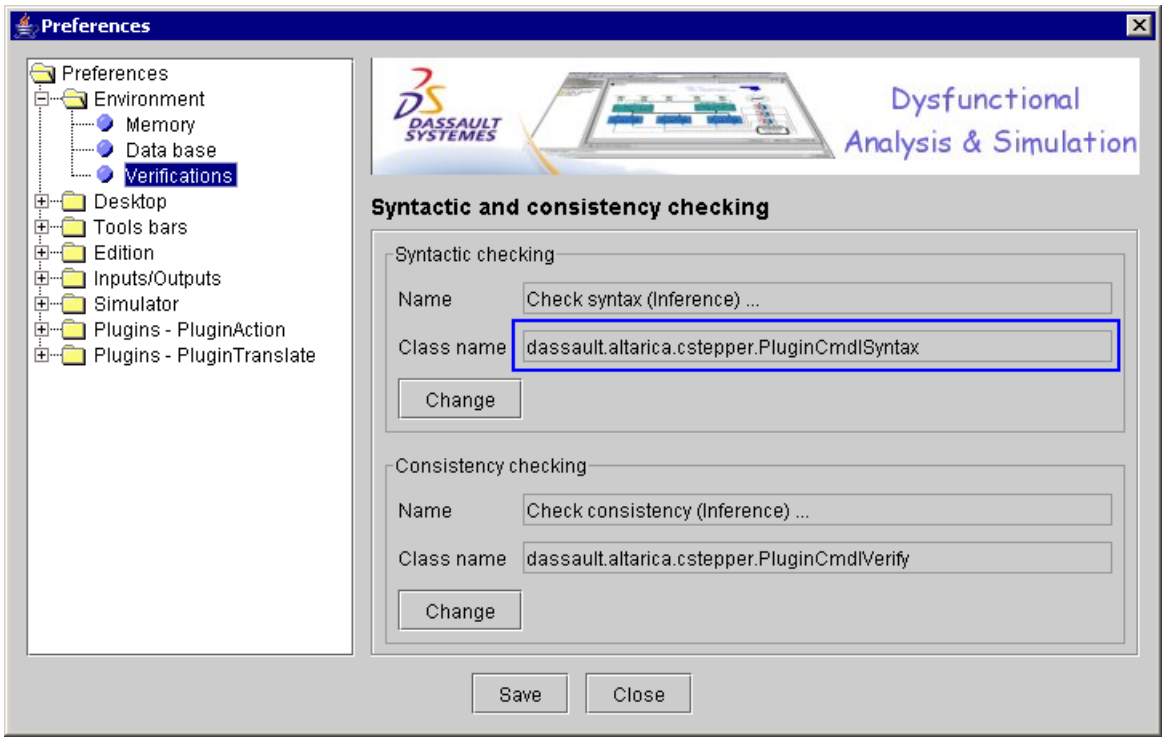
In order to launch syntactical control for current system, use the **Check syntax (Inference)** command.

For components, equipments or operators

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



During first use, you have to verify that syntactical checker linked with this button (=> **Options** Menu, **Preferences** command, **Preferences/Environment/Verifications** path).

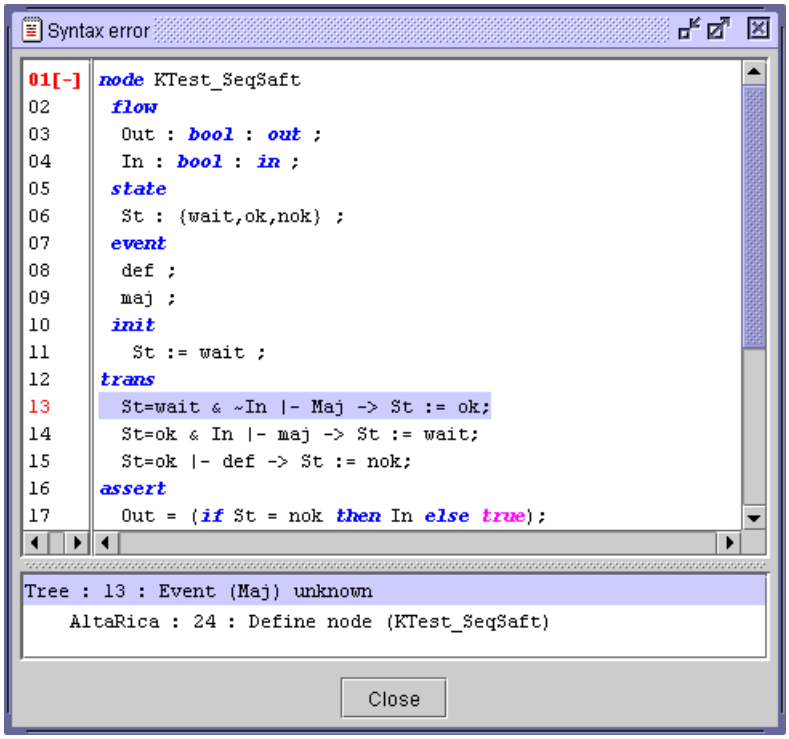


The class name of syntactical checker must be `dassault.altarica.cstepper.PluginCmdISyntax`. On the contrary, another syntactical checker will be used. In order to change the syntactical checker to be used for components, equipments or operators, click on **Change** button (of part **Syntactic checking**).

Result of syntactical check

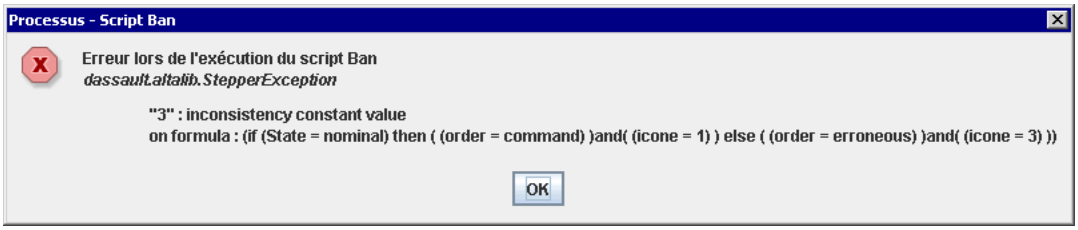
If there is no mistake, a message will specify it.

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.

In some cases, syntactical checker displays an error without line number. In this case the error message is displayed in a window without code.



List of frequent 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 ...` expressions, 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>
     )
   )
```

Consistency-control

Consistency-control validates the fact that model is considered as 'consistent' for inference engine. At first, the consistency-notion corresponds to the non-existence of a variable-valuation-combination (states or flow) that could generate 2 different values for an output flow.

Example:

```
if State = 1 then Out = true;
if State = 1 then Out = In;
```

This example (due to a typo) set a problem when State = 1 and In = false, There can be 2 values regarding the expression which is taken into account.

Afterwards, consistency-control has been extended. It controls completeness (valuation of variables which don't generate output flow). It controls some properties which set or could set a problem to inference engine.

It must be noticed that "original" consistency-control and completeness-control only have meaning if at least one component assertion is written as inference rules. That is to say with an expression `if <condition> then <assignment>` without `else` clause.

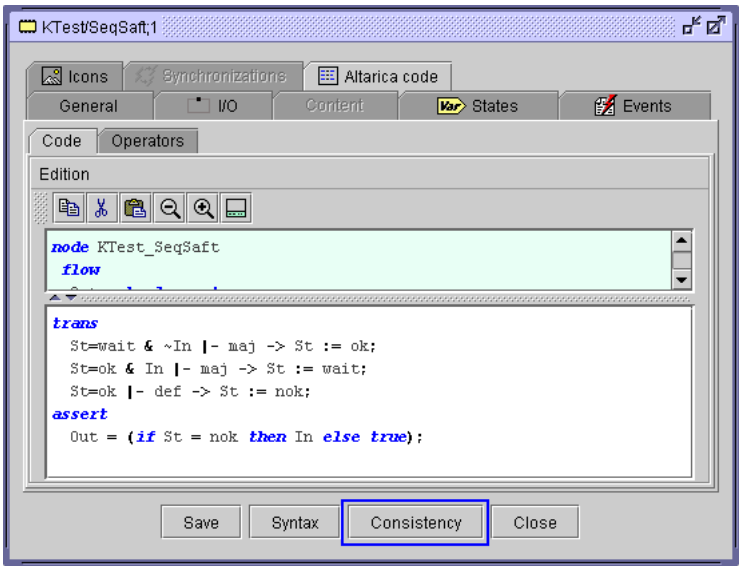
Consistency-control-launching

For systems

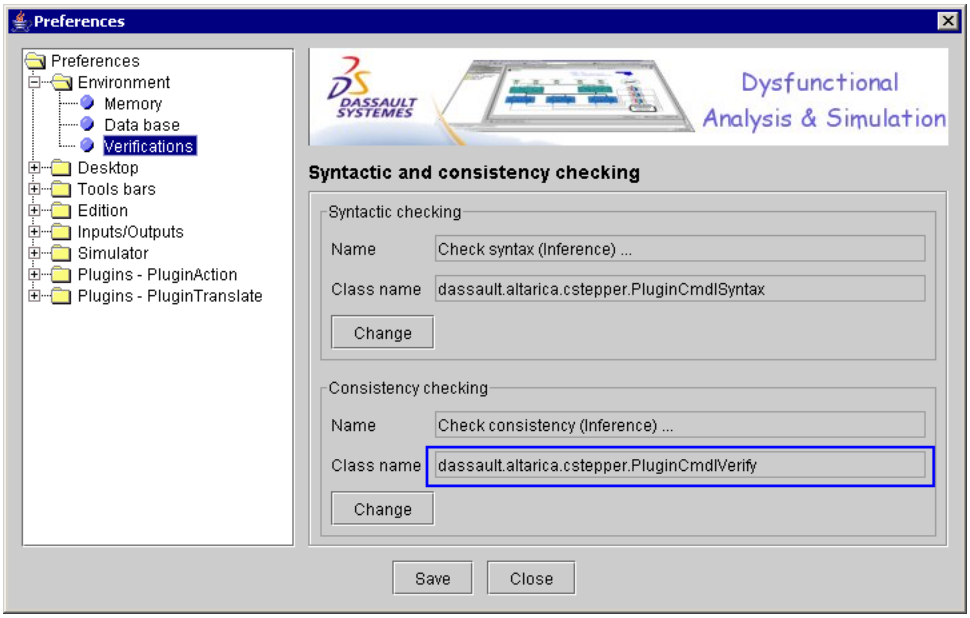
In order to launch consistency-control for current system, use the **Check consistency (Inference) ...** command.

For components, equipments or operators

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



During first use, you have to verify that consistency-control linked with this button is the one of inference engine. It can be made in preferences of BPA DAS (=> **Options** Menu, **Preferences** command, **Preferences/Environment/ Verifications** path).

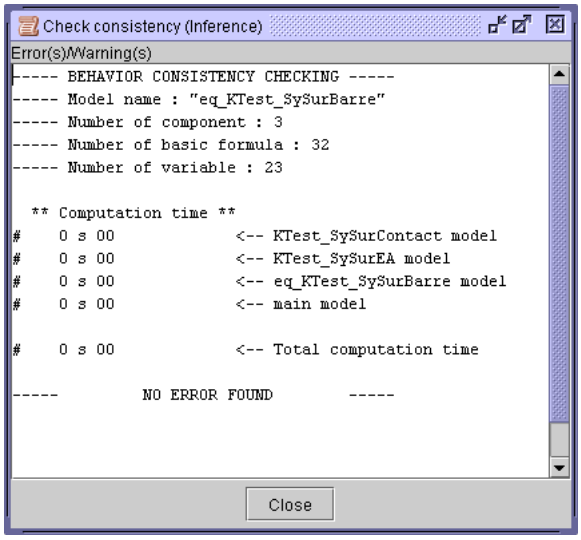


The class-name of consistency-control must be `dassault.altarica.cstepper.PluginCmdIVerify`. On the contrary, another consistency-control will be used. In order to change the consistency-control to be used for components, equipments or operators, click on **Change** button (of part **Consistency Checking**).

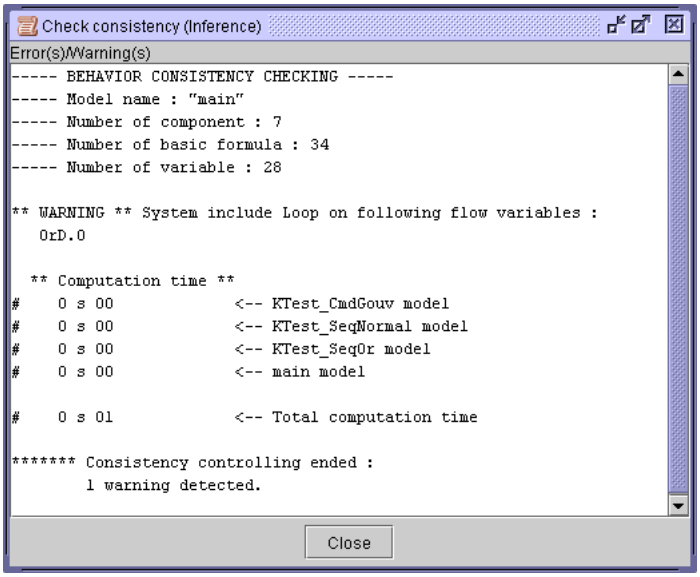
Result of consistency-control

Consistency-control uses syntactical check (Inference). If there is a syntactical error, it will be automatically displayed.

If there is neither syntactical error nor consistency-error, the following window will be displayed, it gives additional informations about the system size.



In case of consistency-error, the same window is displayed. It specifies error messages (if verified properties block inference engine or tree generation up). It also specifies warning messages (if properties may set a problem).



Errors/Warnings detected by consistency-control

1. Output flow variable never valuated in a component.

   So the model isn't complete.

   => Error

2. Output flow variable not valuated in specific condition.

   So the model isn't complete.

   A message shows for which conditions output flow isn't valuated

   => Error

3. Unconsistency for value of output flow variable.

   So the model is unconsistant : two different values assigned to one flow in identical conditions (states and input flow).

   A message shows conditions (states and input flow) allowing unconsistency explanation.

   => Error

4. Detection of unstable comportement of component.

   ```
   state = 1 |- blocking -> blocked := True;
   ```

   Message : Indication of conditions in which inconsistency appears (states, events and input flow)

   Remark : Usually, in order to correct unstability, a condition must be added in transition in order to act as a lock and to avoid new triggering.

   ```
   state = 1 and ~blocked |- blocking -> blocked := True;
   ```

=> Error

5. Combinative explosion during consistency-control.

   Important number of state/flow variables implies a combinative explosion during consistency-control. Algorithm rely on state space saturation of associated variables.

   Current limit is rather high (maximum space of 30 000 simulation combinaisons for each component).

   Modeling of a too complexe component is not recommended. It's better to transform them in equipments and split into many elementary sub-modules.

   => Error

6. To use flow variables in a guard of transition.

   ```
   state = close in.cc=oui |- detect-cc -> state:= open ;
   ```

   Remark : the use of flow variables in a guard of transition may implies FaultTree generation problems.

   => Warning

7. Component final state depending on failure apparition order

   Remark : This information is important for fault-tree-computation, because in this case, failures are not independent.

   => Warning

8. Input flow variable (taking part in component behavior) never valuated.

   Remark : Each input port must be connected (with a direct connection or with an assertion).

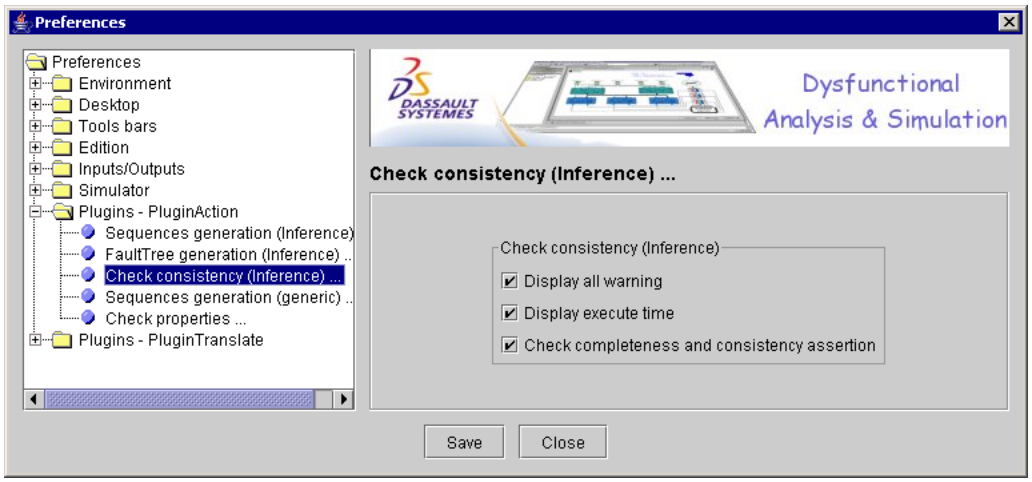
   => Error

9. Conflict on input-flow-variables for a component.

   Remark : Verify that input-port isn't connected to two output ports.

   => Error

User preferences



Specific user preferences for this plugin allow :

- to display or not to display all warning during consistency-control. If this option is checked, consistency-control will only displays errors.
- to display or not to display statistics about processing time (only for equipments and systems).
- to verify or not to verify completeness and consistency of assertions. If this option isn't checked, verification will only be made 'on the fly' during interactive simulation, during Sequences generation, and during FaultTree generation. This option has been added because this control could waste time (especially when assertions are directly written in DataFlow). Moreover, DataFlow traduction module verifies completeness and consistency of assertions, but with a more powerful manner.

FaultTree generation (Inference)

FaultTree generation consists in compiling AltaRica model into boolean formula. Compilation algorithm use a deductive approach from groups of "independant" components. Boolean formula generation is made with reachability graphs associated to each group.

Fault-trees are generated in *Aralia* format. This format contains equations (tree structure), laws, attributs associated with model events, and parameters of generated laws. Generated trees can be either used directly in *Aralia* with command scripts, or imported in BPA SSA or in another fault-tree-editor which supports *Aralia* native format.

FaultTree generation requirements

It's important to notice that Altarica model can be dynamic (reconfiguration, maintenance, ...) and that a fault-tree is an instantaneous view of a system, that's to say a static view. Therefore, an AltaRica model may not be compilable into boolean formula.

FaultTree generation implies some model restrictions. These restrictions are:

- Dynamic behavior of components

  Because fault-tree is a static model, it's impossible to compile a model whose final state depends on faults order. The two following restrictions ensure a static behavior:
  - For each fault-combination, if a fault-permutation is workable, every other permutation must be workable and must lead to the same state of the component.
  - Transitions must be independant of component flows. In other words, component reachability graph must only depends on its initial state and its local failures.

- Uncomplete or unconsistant component(s)

- Too complex component(s)

  A too complex component will generate a local combinative explosion during calcul of its reachability graph.

- Looped System

  A looped system is a system with circular definition of its flow variables. That is to say, a X variable is define with a set of AltaRica assertions which depend on variable X.

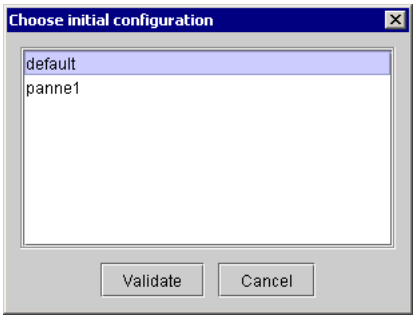
  Inference engine can - in some case - simulate such a system, but fault-tree-generator can't.

These restrictions ensure a good behavior of compilation algotithm. Consistency-control (Inference) enables to spot if model isn't conformable to these restrictions.

FaultTree generation launching

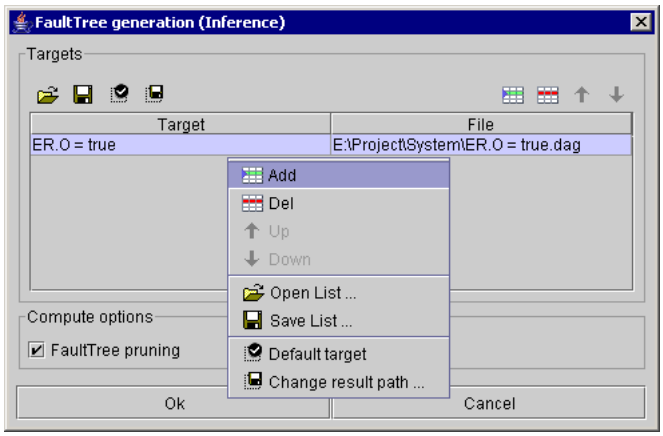
In order to launch FaultTree generation, use the **FaultTree generation (Inference)...** command.

A window is displayed when model has many initial configurations (**System** menu, **Initial states ...** command), in order to select initial configuration to take into account.



Feared event is defined thanks to a calculation-target. The target corresponds to a specific value of a model variable (If the variable is equal to the selected target the feared event happens). Calculation parameters are usually associated with a calculation-target (for exemple, the result file name).

The following dialog window enables to define one (or several) traget(s) which must be taken into account during FaultTree generation. A fault-tree will by generated for each computation target.



This window manage a list of computation-targets. When this window is validate, every typed information are validate to ensure that there is no incomplete line, no wrong line, targets correspond to a variable of processed model, file can be written ....

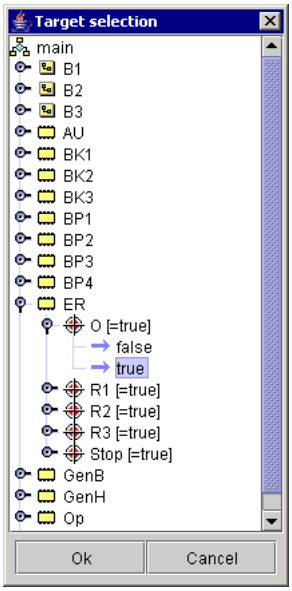
Moreover, if result-files already exist, a dialog window will ask for old file deletion.

Some icons/actions enables to modify the list of calculation targets.

	Add a calculation-target (a line in the list). The target is duplicated from a default calculation target.
	Remove selected computation target.
	Climb selected target up.
	Go selected target down.
	

	Define default computation-target from selected target. Every new target will be created using this default one.
	Save target-list in a XML file in order to re-use it.
	Load a list of computation targets.
	It enables to define result files associated to targets thanks to a pattern which take into account informations linked to calculation targets. The result file name will be define replacing generation tag %xxx% by their contents : %num% => order number in the list, %var% => variable name, %val% => value to use, ...

Double-click on an element of the list enables to edit it with an appropriate editor



Target editor opens a window and shows every model variable in a tree view. Select the value corresponding to the feared event to take into account.

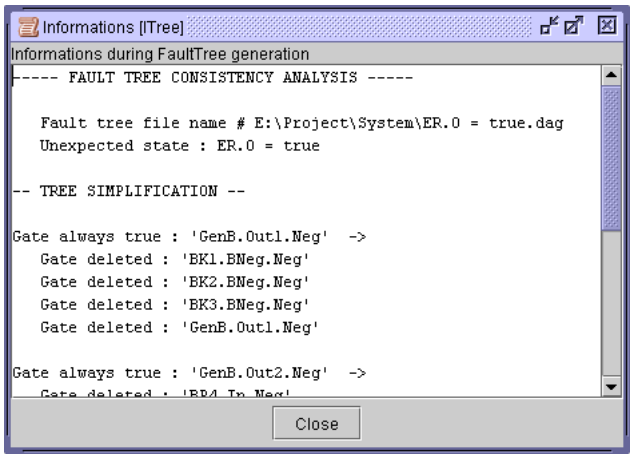
File editor displays a standard file chooser window.

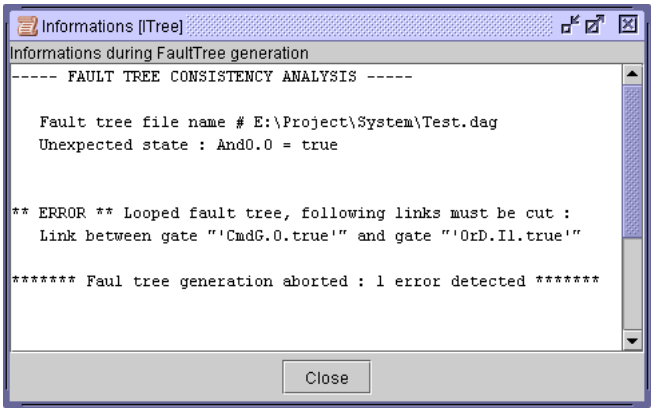
The **FaultTree pruning** option simplify the tree simplifier by deleting unuseful intermediate variables. Other options are available in user preferences.

When this window is correctly filled and validated, **ITree** task is added to the task manager of BPA DAS

Result of FaultTree generation

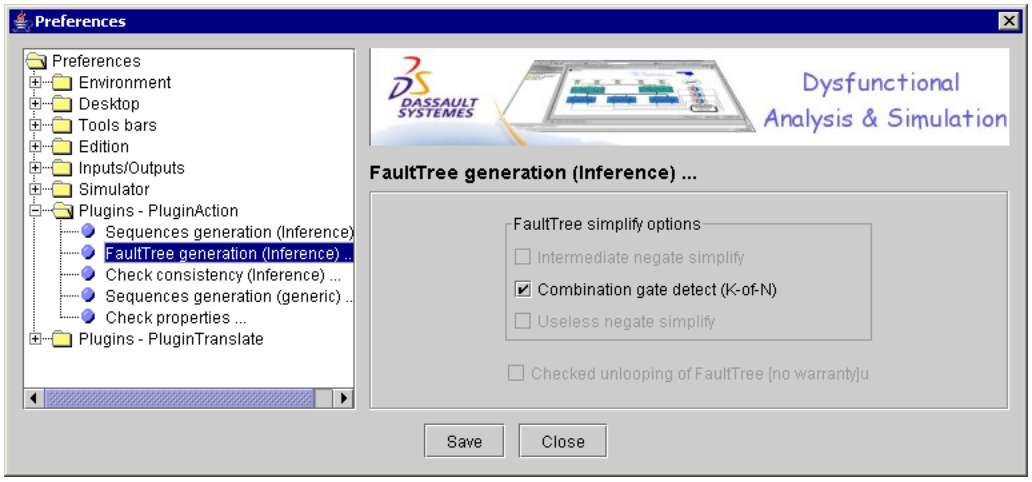
If everything is all right, a window display a summary of computation targets and possible error-messages. Fault-tree-generator based upon inference engine control the model to be processed. It can detect some errors like incompleteness, looped systems or other errors.





User preferences

User preferences allow to configure FaultTree generation. Most options aren't active for user. These options are specific options for development, they will be available soon.



Tree simplification options allow:

- To delete unuseful leaves (events). [disable - development option]
- To detect AND/OR operators structure matching a K-of-N, in order to remplace them by K-of-N operator (possibly more efficient for computation).
- To detect variable with only two allowed value (for example: boolean variables) in order to simplify *not(xxxx.true)* in *xxxx.false*. [disable - development option]

Unlooping option allows - in some cases - to generated tree from a looped system, providing that looped system can be simulated. [disable - development option]

Sequences generation (Inference)

Sequences generation consists in finding every way leading to a specific state (feared event), from an initial state of a system. Number of path to run through are exponential, strategies are set in order to explore the most interesting ones.

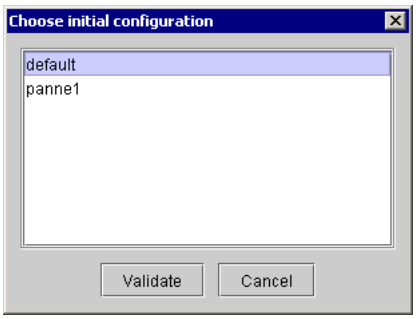
It's possible to do a Sequences generation from the moment it's possible to do a step by step simulation. This is the main avantage compared with FaultTree generation. So, it's possible to do a sequences generation on a dynamic system (and possibly with loops if the stepper can process systems with loops).

Sequences generation doesn't replace a FaultTree generation, because Sequences generation of complex systems can take a very long time.

Sequences generation launching

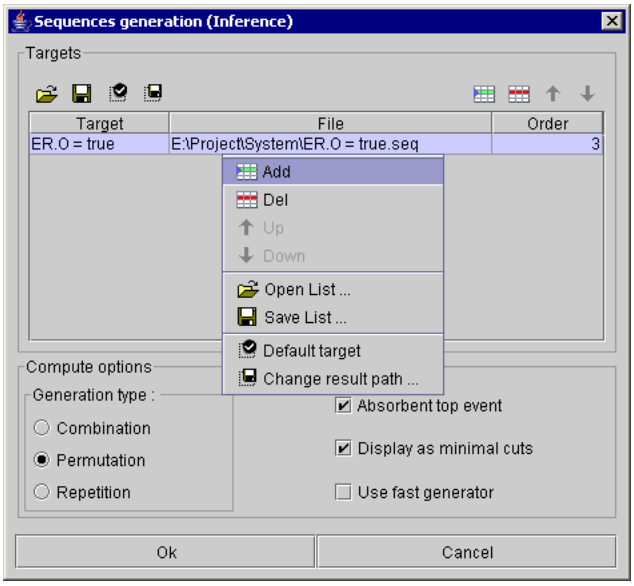
In order to launch Sequences generation, use the **Sequences generation (Inference) ...** command

A window is displayed when model has many initial configurations (**System** menu, **Initial states ...** command), in order to select initial configuration to take into account.



Feared event is defined thanks to a calculation-target. The target corresponds to a specific value of a model variable (If the variable is equal to the selected target the feared event happens). Calculation parameters are usually associated with a calculation-target (for exemple, the result file name).

The following dialog window enables to define one (or several) traget(s) which must be taken into account during Sequences generation. A result file will by generated for each computation target.



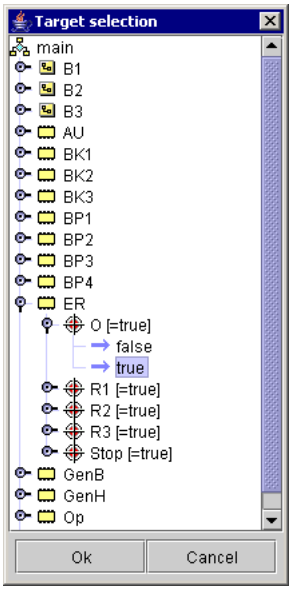
This window manage a list of computation-targets. When this window is validate, every typed information are validate to ensure that there is no incomplete line, no wrong line, targets correspond to a variable of processed model, file can be written ....

Moreover, if result-files already exist, a dialog window will ask for old file deletion.

Some icons/actions enables to modify the list of calculation targets.

	Add a calculation-target (a line in the list). The target is duplicated from a default calculation target.
	Remove selected computation target.
	Climb selected target up.
	Go selected target down.
	Define default computation-target from selected target. Every new target will be created using this default one.
	Save target-list in a XML file in order to re-use it.
	Load a list of computation targets.
	It enables to define result files associated to targets thanks to a pattern which take into account informations linked to calculation targets. The result file name will be define replacing generation tag %xxx% by their contents : %num% => order number in the list, %var% => variable name, %val% => value to use, ...

Double-click on an element of the list enables to edit it with an appropriate editor



Target editor opens a window and shows every model variable in a tree view. Select the value corresponding to the feared event to take into account.

File editor displays a standard file chooser window.

The third column enables to choice the maximum **Order** of sequences, that's to say the number of events/transitions of a sequence. The transition number doesn't take into account instantaneous transitions if their automatic firing is activated (user preferences).

The **Generation type** specifies the Sequences generation strategy.

- **Combinaison**: During the simulation phases, every combination identified is simulated once in a given order. Within a combination the same event is drawn only once.
- **Permutation**: During the simulation phases, every identified combination is simulated in all orders. Within a combination the same event is drawn only once.
- **Repetition**: Each combination is simulated in all the orders. Within a combination the same event is drawn many times.

Following options also influence sequence-generator:

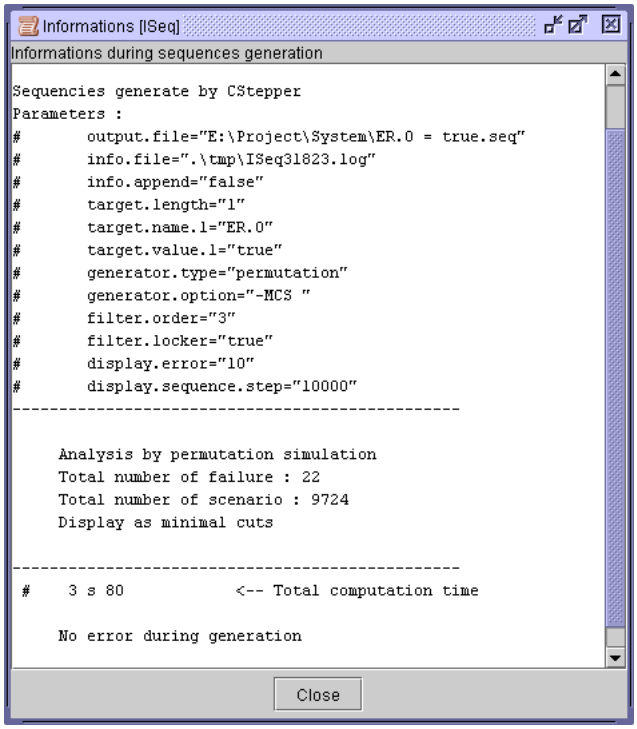
- **Absorbent top event**: If this option is checked, sequence-generator stops its exploration when top event is reached.
- **Display as minimal cuts**: If this option is checked, found sequences will be minimized assuming they are minimal cuts, that's to say without ordering notion of event appearance.
- **Use fast generator**: If this option is checked, an experimental generator will be used. It uses a pre-processing of elementary components

When this window is correctly filled and validated, **ISeq** task is added to the task manager of BPA DAS

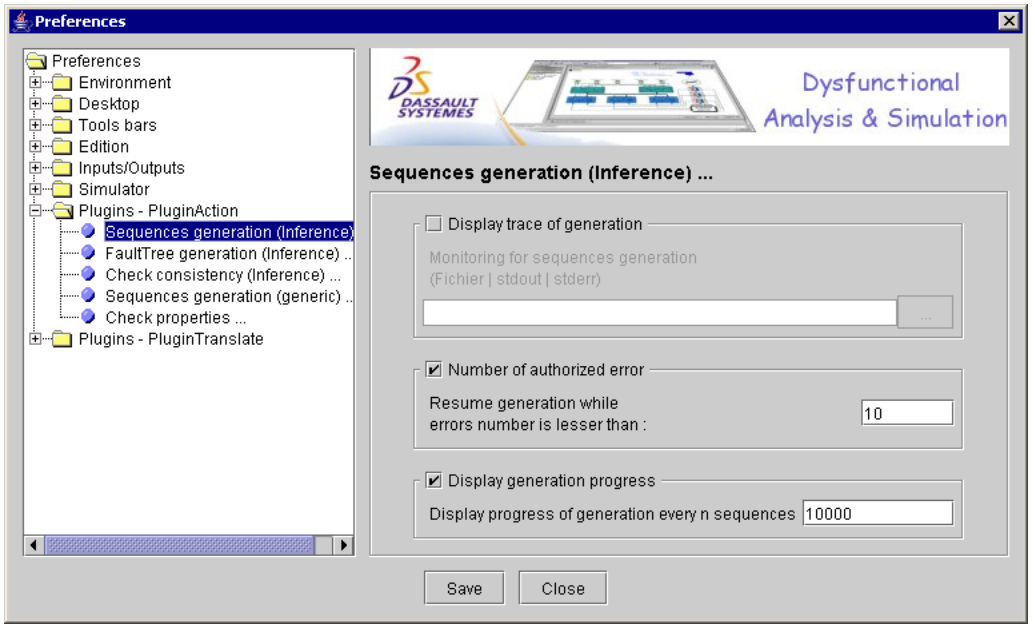
Result of Sequences generation

If everything is all right, for each computation target, a window displays a summary of computation parameters, statistics, and possible error messages.

Sequence-generator can come up against inconsistencies of the system, or instantaneous loop (that's to say a series of too many instantaneous transition)



User preferences



User preferences allow Sequences generation configuration.

- **Display trace of generation** allows you to keep trace of fired transitions. The trace can be displayed on standard output, on error output or in a specified file.

- **Number of authorized errors** enables to continue Sequences generation despite errors, assuming nummber of errors is below authorized one. This option allows not to stop generation after first error, so it's easier to debug.
- **Display generation progress** displays number of played sequences every n sequences. It enables to follow generation.