IRT Saint Exupéry LIV-S085L01-001

IRT SystemX ISX-S2C-LIV-1285

Issue 4

MBSA Modelling guide and validation report

DATE: 17/03/2023

Summary

The S2C project aims to define processes, methods and tools that allow to guarantee that safety analyses and system modelling done by system engineer are consistent. These activities are preformed, in a context of digital continuity, during all iterative development cycles of products and systems, to answer to certification constraints.

This guide aims to develop and validate a shared Model Based Safety Analysis methodology, based on AltaRica language, suitable for aeronautical developments. It presents some general principles, through a unique simple example, as well as main identified difficulties modelers may encounter such as control loop management up to mathematical theory.

Our intent is to support classical ARP4761A PSSAs and SSAs analyses with MBSA.

Author(s)	*Function(s) & name(s)*	*Research engineers*	*X. de Bossoreille* *S. Delavault* *F. Deschamps* *C. Frazza* *T. Prosirnova* *C.Seguin*
Checker(s)	*Function(s) & name(s)*	*WP leader*	*S. Delavault*
Approver	*Function & name*	*Project leader* *IRT Saint Exupéry*	*J. Perrin*

Evolutions

Version	Date	Modified §	Modification summary	Modified by
1	01/04/2021		First issue	
2	01/06/2021		Updates following comments	
3	07/04/2022		Delivery for J3	
4	17/03/2023		Delivery for J4	

The S2C team would like to thank M. Machin, E. Saez, for their contributions to the previous issues of the document.

Table of Contents

Table of figures

Table of tables

1 Introduction

1.1 *Purpose of document*

One of S2C project target is to develop and validate a shared MBSA methodology suitable for aeronautical developments.

This document provides methodological guidance for MBSA using AltaRica Data flow modelling languages and related tools. It presents general principles as well as main identified difficulties that modellers can encounter. Our intent is to provide recommendable generic practices to use MBSA for the support of classical ARP4761A PSSAs and SSAs analyses.

The recommended practices are illustrated using the AltaRica Data Flow language supported by Cecilia OCAS (Dassault Aviation), SimfiaNeo (AIRBUS Protect) tools and Open AltaRica.

1.2 *Referenced documents*

1.2.1 S2C referenced documents

Reference Number	Title	Reference
[REF 1]	State of the Art of the S2C Project	LIV-S085L01-001-V3 ISX-S2C-LIV-1285-V3
[REF 2]	MBSE-MBSA Consistency	LIV-S085L02-007-V6
[REF 3]	Structural Scoped Review	LIV-S085L02-023-V3
[REF 4]	Behavioral Scoped Review	LIV-S085L02-024-V6
[REF 5]	Behavioral Cross Check	LIV-S085L02-025-V6

1.2.2 External referenced documents

Reference Number	Title	Reference
[REF A]	Michel Batteux, Tatiana Prosvirnova & Antoine Rauzy. "AltaRica 3.0 in ten modelling patterns".	In International Journal of Critical Computer-Based Systems. Inderscience Publishers. Vol. 9, Num. 1-2, pp 133-165, 2019.
[REF B]	GUIDELINES AND METHODS FOR CONDUCTING THE SAFETY ASSESSMENT PROCESS ON CIVIL AIRBORNE SYSTEMS AND EQUIPMENT	SAE ARP4761A / EUROCAE ED-135A

1.2.3 MBSA tools configuration

Reference Number	Tool name	Tool Owner	Revision
[REF X]	Cecilia	Dassault Aviation/ Satodev	V 6.0.4
[REF Y]	SimfiaNeo	AIRBUS Protect	V1.4.2
[REF Z]	Open AltaRica	Open source	v1.2.0

2 Glossary

2.1 Abbreviations and acronyms

Abbreviation and acronyms	Definition	Reference
ASA	Aircraft Safety Assessment	[REF B]
ARP	Aerospace Recommended Practice	[REF B]
CCA	Common Cause Analysis	[REF B]
CCF	Common Cause Failures	[REF B]
CMA	Common Mode Analysis	[REF B]
COM	Command	[REF B]
DAL	Design Assurance Level	[REF B]
DD	Dependence Diagram	[REF B]
FC	Failure Condition	[REF B]
FDAL	Functional DAL	[REF B]
FFS	Functional Failure Set	[REF B]
FMEA	Failure Mode and Effects Analysis	[REF B]
FMES	Failure Mode and Effects Summary	[REF B]
FPM	Failure Propagation Model	[REF B]
FTA	Fault Tree Analysis	[REF B]
MBSA	Model Base Safety Analysis	[REF B]
MCS	Minimal Cut Set	[REF B]
MON	Monitoring	[REF B]
N/A	Not Applicable	[REF B]
PRA	Particular Risk Analysis	[REF B]
PSSA	Preliminary System Safety Assessment	[REF B]
SA	Safety Analyst	[REF 2]
SE	System Expert	[REF 2]
SFHA	System Functional Hazard Assessment	[REF B]
SFMEA	System Failure Mode and Effect Analysis	[REF B]
ZSA	Zonal Safety Analysis	[REF B]

2.2 Definitions

This chapter deals with the definition of terms introduced in this guide. These definitions are pertinent in the frame of the context of the S2C project.

The terms in bold in the definition are defined in the same table.

Terms	Definition
Assertion	An assertion is the definition of the behavior of a **modelling unit**. This definition is coded in AltaRica.
Basic event	A basic **event** is an elementary event for the system considered.
Component failure	A component failure is a combination of **basic events** that lead to the observed failure
Cut Set	A Cut Set is a unique combinations of **component failures** that can cause system failure. Specifically, a cut set is said to be a minimal cut set if, when any **basic event** is removed from the set, the remaining events collectively are no longer a cut sets.
Domain	A **domain** is a set of data of value.
Dormancy	The dormancy represents the duration when a failure is not visible.
Dynamic model	A model is called **dynamic** if it is not static i.e. it exists at least one couple of **sequences** that are constituted with the same **events** and result in different configurations
Error	An error is when a system deviates from its correct service **state**.
Event	An event is a change of state of a variable.
Failure	A loss of function or a malfunction of a system or a part thereof.
Failure Condition	A Failure condition with an effect on the aircraft and its occupants, both direct and consequential, caused or contributed to by one or more failures, considering relevant adverse operation or environmental conditions. A Failure Condition is classified in accordance to the severity of its effects.
Fault	An undesired anomaly in an item or system
Fault Tree	A Fault Tree is a failure propagation model defined as a combination of logical operators that define the combination of events leading to final state (failure condition).
Functional Failure Set	A Functional Failure Set is equivalent to a **cut set** of a failure condition but modelling the development errors instead of failures of the equipment.
Modelling unit	A modelling unit is a basic element to build a model within a tool
Node	A node is the visual representation of an **event**
Observer	An observer is a modelling artefact used to compute and report the status of a variable (**domain**) associated to a failure condition.
Sequence	A sequence is a succession of **events** in a certain order from an initial **state** and leading to a final **state**.
State	A state is the condition in which a modelled system is at one time of a **sequence**.
Static model	A model is called **static** if all the sequences of transitions which starts from the same configuration of the accessibility and which are made of permutation of a same set of **events** leads to the same configuration of the reachability graph of the model
Transition	A transition is an equation that define the change from an initial state of an **event** to the final state of the same **event** depending on a trigger or another **event**.

Terms	Definition
Trigger	A trigger is an external **event** that push an **event** to occur.
Variable	It is a symbol and placeholder for (historically) a quantity that may change, or (nowadays) any mathematical object. In particular, a variable may represent a number, a vector, a matrix, a function, the argument of a function, a set, or an element of a set. It is an abstract storage location paired with an associated symbolic name, which contains some known or unknown quantity of information referred to as a value.

3 General introduction: context and objectives

3.1 Context

The S2C project was born thanks to an industrial need to improve current development processes and to ensure consistency between safety analyses and system design, which today requires a significant effort.

Indeed, the development of systems like aircrafts is made difficult by two major factors: the inherent complexity of such system and the complexity of the organization required to build and maintain them. A large number of stakeholders are involved throughout the lifecycle. They have various concerns that are not necessarily consistent with each other. Each stakeholder has its own viewpoints and descriptions, focused on its concerns and based on the use of dedicated languages, formalisms, and tools. These viewpoints are highly interrelated, which produce overlaps. Clearly, these overlaps and all elements mentioned above may possibly introduce inconsistencies.

In the S2C project, we will focus on the interactions between two particular disciplines: systems engineering and safety analyses. The challenge is to develop methodologies to manage inconsistencies and to maintain a coherent state over the development cycle. These methodologies should be validated through a representative case study supporting by tools demonstrator in order to define tooling specification.

The main objectives are to:

- Improve confidence in safety analysis by aligning and maintaining consistency with system definition models.
- Provide safety specialists with more efficient means to understand and analyze complex systems.
- Facilitate the use of model-based approaches in compliance with the requirements of certification bodies (for the MBSA approach, there are no requirements issued by certification bodies).
- Reduce the number of iterations between system definition and related safety analyses, in order to limit costs and delays, by enabling efficient MBSE/MBSA interfaces [REF 2] or MBSE/FTA interfaces.
- Control changes during the product development cycle; reduce risks due to redesign.

In order to address these issues, the project was divided into 4 work packages:

- WP1: Definition of a global process for ensuring and maintaining system/safety consistency all along the life cycle [REF 1]
- WP2: Methods and means of implementing and maintaining system/safety consistency for the integrated system models and system levels (horizontal links) [REF 2], [REF 3], [REF 4], [REF 5]
- WP4: Modeling methodology for safety

3.2 Objectives of the document

S2C Work Package 4 aims at developing and validating a shared Model Based Safety Analysis (MBSA) methodology suitable for aeronautical developments based on members experience and best practices. The project choice is to focus on the Model Based System Analysis defined in the **ARP4761A** guidelines [REF B].

ARP4761A ([REF B])current version defines MBSA as the "[…] technique which models system content and behavior in order to provide safety analysis results. MBSA employs an analytical model called a Failure Propagation Model **(FPM)."**

In this document amongst the different processing algorithms available for Model Based Safety Analysis **[REF 1]** we choose to consider the boolean equation generation and sequence generation.

Our purpose is to provide validated recommended practices built on the experience of the S2C projects members who are amongst the main actors of the MBSA in aeronautics. This document aims to provide support and illustration of the proposed methods using AltaRica Data Flow language in Cecilia Workshop (Dassault Aviation /Satodev) and SimfiaNeo (AIRBUS Protect) tools, and in some cases by examples of AltaRica 3.0. (Open AltaRica).

This document targets the safety specialists with no MBSA background as well as MBSA advanced users. We expect readers discovering the MBSA to have a set of mind opened to programming, new reasoning and tools.

3.3 Contents and reading levels

This document provides three reading levels (Beginners, Intermediate and Advanced MBSA users) organized as described in the following table. Note that our goal is to provide the beginners with the necessary bases to become advanced users after practicing sufficiently.

The following table describes the content of the different sections of the document. It also provides the reading level they target.

Topics	Section	Section title	Reading level	Content
Modelling : **Write and understand failure propagation models**	4	Principles of AltaRica Data Flow language	For beginners users.	To understand the bases and Principles of AltaRica Data Flow language and mathematical background.
	5 , 5.5 and 7	Get started with failure propagation modelling	For beginners users.	To get started with MBSA
	8	Models characteristics impacting the simulation	For advanced users and beginners.	To provide guidance to overcome specific MBSA difficult points of modelling
Computation **: Analyse failure propagation models**	9 and 11	Principles of analysis of AltaRica Data Flow models	For advanced users and beginners.	To detail the computation principles and beyond of the AltaRica Dataflow models
	10	Specific modelling topics (remove the stone in your shoe)	For advance users.	To go further with the failure propagation models.
Assurance: **Assure the acceptability of models and analyses with respect to the user goals**	12	Verification & validation of MBSA activities	For beginners users.	To understand how to integer a MBSA process in a V&V product life cycle.
Process**: Include MBSA in development processes**	13	Using MBSA to support an industrial development and Aerospace Recommended Practices	For beginners and advanced users.	To introduce the integration of the S2C guideline inside an aeronautic industrial development.

Based on the above table, the document is introducing the reading level of each chapter by the following legend:

- B - : Beginner reading level

- I - : Intermediate reading level

- A - : Advanced reading level

The document does not proposed any further definition of those categories and let the reader define in which category, the reader is comfortable with.

3.4 Topic not addressed

The document does not address the tool performance as a modelling choice.

3.5 Introducing the MBSA interest

MBSA is the answer to several challenges from modern complex systems and limits of traditional methods such as Fault Tree Analysis.

Based on an example from the APR4761A / ED -135A [REF B][1], this section highlights some properties of MBSA, especially the failure propagation models in regards of challenges and limits.

The example is a Wheel Breaking System, considering two Failure Conditions: Loss of breaking capability and Untimely Breaking. The number of failure conditions is limited in order to illustrate the methodology. However others FC shall be considered. The graphical view of the model, provided in Figure 1, allows to illustrate the main properties of MBSA.

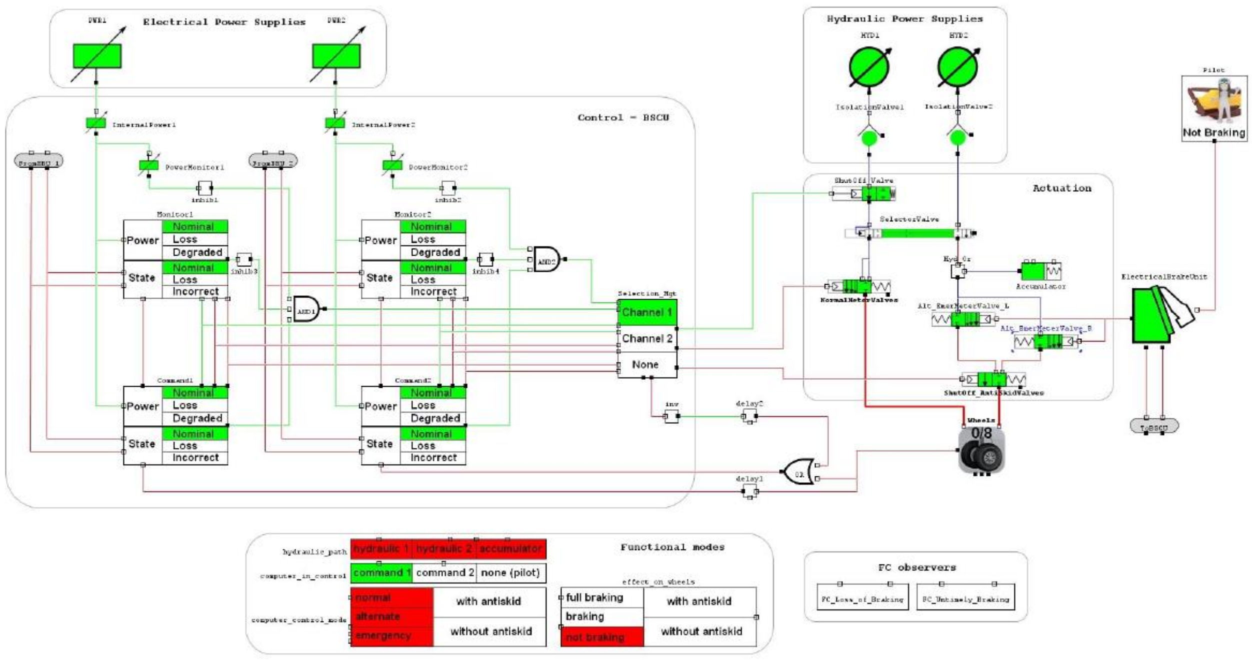
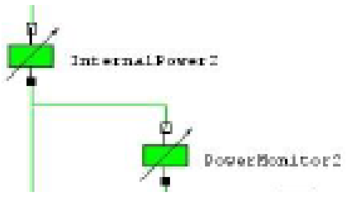


Figure 1 - Graphical view of Wheel Breaking System AltaRica model from ARP4761A/ED-135A ([REF B])

The MBSA, as defined in section §2.2, has several properties, making it efficient to perform safety analysis. These properties are listed hereafter:

- **The MBSA can address dynamic systems.**
  Usually the analyst takes into consideration dynamic aspect of a system and simplify it in order to use static methods such as fault trees. For example (See Figure 1), if the monitoring fails before the power supply dedicated equipment, the failure can remain hidden (in case of redundancy). In fault tree this situation is capture with a combination (AND gate) leading to a cut set loss of internal power and loss of monitor. The MBSA allows to distinguish between the sequence loss of power supply (which is detected) and then loss of monitoring and loss of monitoring (considered undetected) and then loss of power supply. There are a lot of other illustration of importance of the dynamic aspect (repair phase …) which are not or poorly capture by standard methods.
  The MBSA offers the possibility to address order of events in a sequence (A then B, or, B then A). This topic is detailed in chapter 8.4.

[1] This model is available here: https://satodev.com/nos-produits/cecilia-workshop/



MBSA will only select the sequence "loss of monitor and then internal power" as relevant for a failure condition hidden loss of internal power.

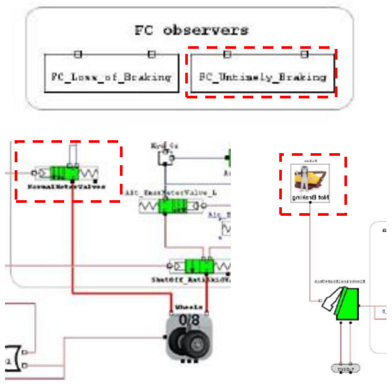
The fault tree analysis is not making any difference.

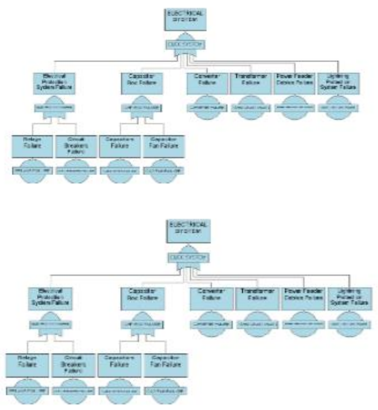
Figure 2 – Illustration of monitoring of power supply

- **The same model is designed for different Failure Conditions**
  Usually a system has between 5 and more than 50 Failures Conditions. Sometimes, when you are using Fault Tree Analysis, several failure conditions can be merged. Dozens of them still can be analyzed. With MBSA, several Failure Conditions can be analyzed with more efficiency and consistency. The Figure 3 highlights the model of the Failure Condition and the graphical needs to represent them with a standard approach (Fault Tree Analysis). A Failure Condition in the AltaRica model is only a simple Boolean equation (observer in AlatRica domain) considering two inputs: outputs of NormalMeterValve and Pilot Not breaking. Then the result is generated by the tool. For the Fault Tree analysis the complete scenario (with all contributors) has to be modelized by the safety analyst.

AltaRica model (graphical)

Fault Tree Analysis





BC_untimaly_Breaking = NormalMetterValve is OK and Pilot_Not_Breaking is OK

Figure 3 - Comparison of MBSA and FTA

- **The same model can be used for different analyses**
  The Fault Tree allows to generate cut sets and an estimation of the probability of occurrence. The same AltaRica model, if made for that purpose, can be used with several computation engines such as Boolean equation generation (Usually represented as a Fault Tree) and cut set calculation, sequence generation, probability estimation using Monte Carlo. This model is not limited to one purpose and it can be used for different usages.

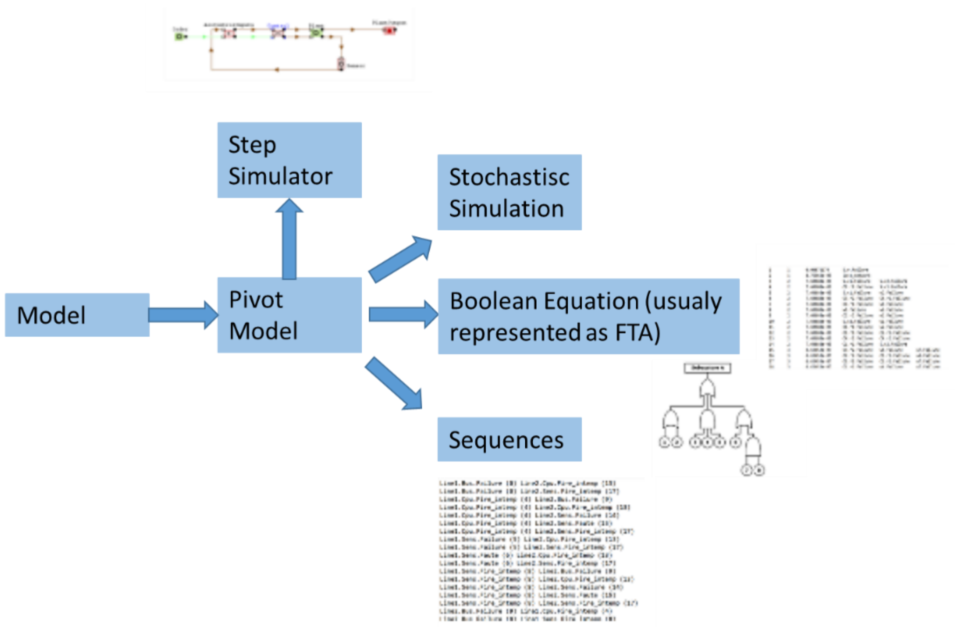


Figure 4 – Interoperability of tools with main model

- **The representation of a system similar to engineering schematics from a structural point of view.**
  This property is obvious considering Figure 1 compared to Fault Tree Presented in Figure 3 - Comparison of MBSA and FTA (right part).
  The model eases a common understanding of the system behavior by using the same baseline to represent the designed architecture and the operation of the system.
  This model is simulable with a step simulator allowing to trigger occurrence failures and results on the state of equipment or flows with appropriate figures and colors as it can be seen in Figure 5.

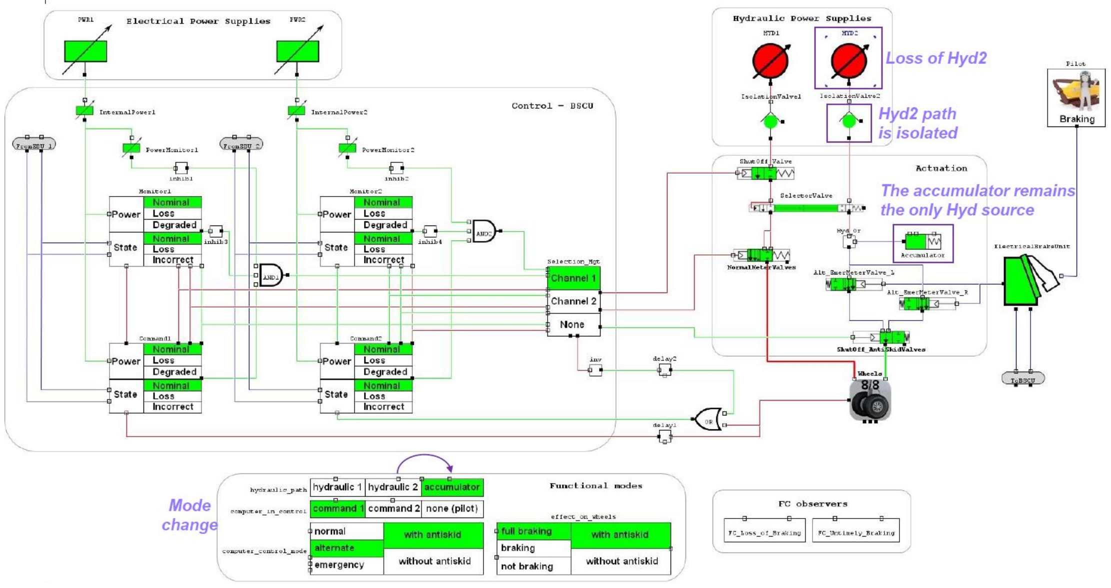


Figure 5 – Illustration of step simulator with failure at hydraulic power supply level

Moreover the model addresses also the functional behavior (not only dysfunctional) which eases discussions between the safety and the design teams. For instance, it is possible to define the appropriate functional mode (full braking, braking, not baking, with / without antiskid) and to monitor the main status of a system (loss of hydraulic, command 1 considered …) as shown in Figure 6.

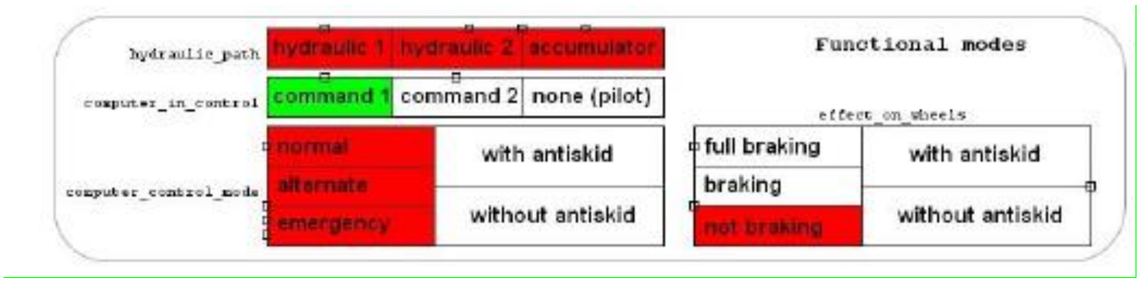


Figure 6 – Illustration of step simulator with failure at hydraulic power supply level

- **The usage recommended is to use patterns, leading to update easily the models.**
  It is highly recommended to use a library of templates in order to build the model. Same equipment are based on the same template that is why their behavior is consistent. The template is copy-pasted on the model and each equipment is a different instantiation of the template. If needed, the modification of the template is able to be propagated on the complete model, and even other models.
  Even without template approach, the update is faster than a Fault Tree analysis and more robust in case of modification of the architecture (add an item and associated probabilistic law) or identification of new failure conditions (add an observer). The analyst doesn't have to go through each FTA in order to take into consideration the modifications. This is a reason why the MBSA is used to perform quick analysis in order to make some trade off from safety perspectives between several architectural solutions.

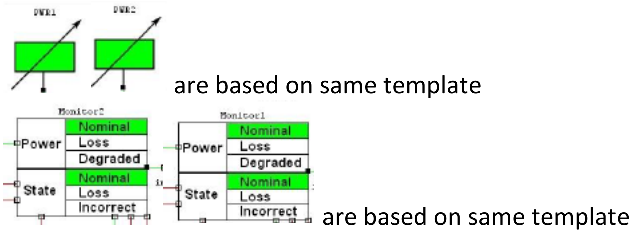


Figure 7 – Same patterns used twice in the model

- **The approach is to act smart locally, and dummy globally.**
  The effort of a behavior modelling is performed at item level. The behavior at the system level is generated automatically by using the connection ports from different items. The model takes into consideration the reconfiguration of the system when needed, while the FTA remains static. This characteristic implies homogeneous and robust results at the system level. This approach allows to analyze a complex system, as the complexity is reduced to the local behavior of items.
  One other interest of this property is the ability to connect several systems together if the interfaces are correctly specified. It is then possible to simulate very complex scenarios for very specific situations.

4 - B - AltaRica Data Flow language – general vocabulary

This section introduces the syntax (writing rules) and semantics (meaning rules) applicable to the AltaRica Data Flow models. Its intent is to provide insight and understanding of the AltaRica languages in its whole. Note that it does not present the graphical capacities of the tools that greatly reduces the coding effort.

The definitions, presented in chapter 2.2, are illustrated by a small case study: the command/monitoring pattern of safety architecture.

Such a pattern is structured as follows to compare the orders in case one fault occurs.

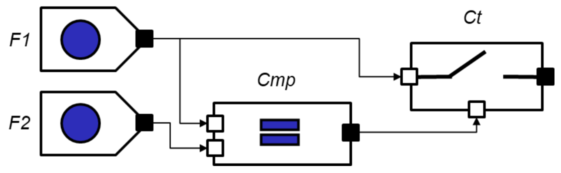


Figure 8: Command/Monitoring (COM-MON) pattern of safety architecture

It contains the following components:

- Two numerical functions ***F1*** and ***F2***
- A comparator ***Cmp*** that checks the equality of two inputs
- A contactor ***Ct*** that is closed as long as the equality check is true. When it is closed, it transmits F1 output; else, it transmits no output.

The functions have two failure modes:

- They may produce an erroneous output.
- They may produce no output at all.

The failure conditions of interest for this pattern are defined in the below table:

Failure Conditions	Text	Classification
FC_B1	erroneous output	Catastrophic
FC_B2	Loss of output	Minor

Table 1: COM/MON example – Failure Conditions definition and classification

AltaRica models are based on AltaRica formal language. They are some kind of program code: they shall be written according to rules defining the *syntax of the AltaRica language* (e.g. these kind of rules are in Cecilia WS user manual). MBSA tools can understand the AltaRica model code if and only if the model satisfies the syntax rules. Syntax errors prevent the model simulation or analysis.

Let us clarify the main syntax rules for AltaRica DataFlow models.

Two kinds of words can be used to write AltaRica codes: Identifiers and Keywords

Identifiers are names defined by the language users to refer different data (components, variables…) of their program.

A first syntax rule defines how to write valid identifiers and one rule applied by several tools is the following:

"The identifier shall start by a letter of `[a-z]or[A-Z]` and it can be followed by other letters or figures of `[0-9]` or `[_].`"

Example: `F1` satisfies this rule. It is the identifier used to refer one component of the case study.

Keywords are reserved names with a predefined meaning. They will be introduced progressively in the following.

NB: Some keywords or language writing rules may slightly change according to the language version or supporting tools. Main differences are between the AltaRica3.0 version and the former versions. Here we introduce only the syntax of concepts shared by all dataflow models and when needed, the alternative 3.0 notation is given in brackets [3.0] for the interested readers.

Example: the keyword `true` is one predefined possible value of the Boolean data.

4.1 - B - Domains

All data of an AltaRica code shall be associated with a domain.

Domains (respectively **finite domains**) are set (respectively finite set) of data values. They are attributes of the program data, which defines how users and tools have to interpret the data.

Examples:

`bool [3.0 Boolean]` is the AltaRica keyword used for the pre-defined domain `{true, false}`.

`int [3.0 integer]` is the AltaRica keyword used for the pre-defined integer domain.

Failure propagation models often model the quality and failure mode of the service provided by a function. In our case study, `D_Command` is a user finite domain of the case study. It is an **enumeration** of values declared as follows:

```
domain D_Command = {OK, LOST, ERR};
```

The kind of domain has a strong influence on the model exploitation. For instance, the use of finite domains such as Boolean and user-defined enumerations is compatible with any kind of analysis. This is not the case for integer or float values, which are currently compatible only with stochastic simulations.

In case of a re-use of an existent domain, there is not any impact on the assertions.

4.2 - B - Overview of modelling units

An AltaRica code can be structured in modelling units. The keyword used to declare a modelling unit is **node [3.0 class].**

These units are similar to classes in the object oriented programming languages: they represent reusable (« off-the-shelf ») components and they can be **instantiated** inside other units. They allow building customized libraries.

Such units have a beginning, definition fields and an end. This **unit structure is shared by all AltaRica languages versions** even if it is written with different keywords.

In this section, we use an example to give an overview of the modelling units shared structure and to clarify the nature of the various fields of this structure. The field details are given in next sections.

Example: Components F1 and F2 can be seen as two instances of the same component `Source` with the following behavior (Figure 9):

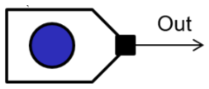


Figure 9: The "Source"

The modelling unit `Source` produces an output Out.
It may *fail*. In this case, the output Out is lost.
It may also produce *errors*. In this case, the output Out is erroneous.
Initially, the source performs the nominal function.

Table 2 shows the model of this component. The first column gives the structure of the modelling unit, which is shared by all languages. The second (respectively third column) gives the keywords for the first version of the AltaRica language (respectively 3.0 variant)

Start of unit definition	**node** Source	[3.0] **class** Source
Declarations of flow variables	**flow** Out:D_Command:out;	D_Command Out (**reset** = LOST);

Declarations and initialisations of state variables	**state**
	St:D_Source;
	init St := State_OK;
		D_Source St(**init** = OK);

Declarations of events	**event**
	fail_loss, fail_err;
	extern
	law <**event** fail_loss> = exp(1.0E-4);
	law <**event** fail_err> = exp(1.0E-5);
		event
		fail_loss(**delay**=**exponential**(1.0E-4));
		event
		fail_err(**delay**=**exponential**(1.0E-5));

Unit transitions	**trans**
	(St = State_OK) |- fail_loss -> St := State_LOST;
	(St = State_OK) |- fail_err -> St := State_ERR;
		transition
		fail_loss: (St == OK) -> St := LOST;
		fail_err: (St == OK) -> St := ERR;

Unit assertions	**assert** Out = if (St = State_LOST) then LOST else if (St = State_ERR) then ERR else OK;	**assertion** Out := St
End of unit definition	**edon**	**end**

Table 2: Modelling description of "Source"

4.3 - B - *Declarations in a modelling unit*

In this section we describe the different variable kinds and how they are declared in modelling units.

Flow variables[2] are used to model flows circulating through the model. In plain dataflow languages, they are either inputs or outputs of the node.

State variables are used to model the states of the systems. An initial value shall be assigned to any state variable.

These two kinds of variables take their values into domains. **The variable declarations link the identifier of the variable to its interpretation domain**.

Example: the output flow Out is interpreted over the domain D_Command previously defined (outside the modelling unit) i.e. Out can only have the value, OK, LOST or ERR. The output flow is declared as follows (using the keyword "out" to declare an output flow):

```
Out:D_Command:out;
```

The state variable St is interpreted over the domain D_Source, meaning it can only take the value State_OK, State_LOST and State_ERR (cf the declaration of the state in the above table). A component may have several state variables. Moreover, an initial value shall be declared for any state variable. Here, St initial value is State_OK as written below

```
init St := State_OK;
```

The **event** names are used to refer a change of values of some state variables. Their declaration may assign a **probability law** to the event occurrence. Laws available depend on the tool.

Example: the occurrence of the fail_loss event follows an exponential law with parameter lambda 10^{-4}

```
[3.0] event fail_loss(delay=exponential(1.0E-4));
```

4.4 - B - *Transitions in a modelling unit*

The graph below describes the expected dynamic behavior of the source component[3].

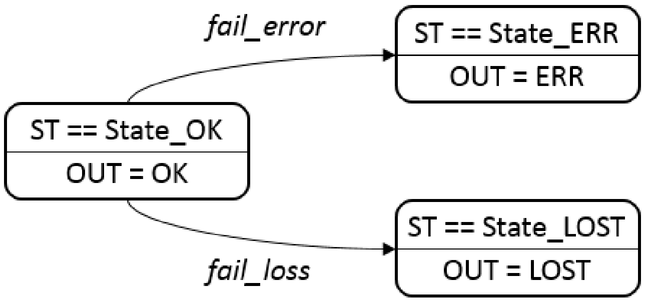


Figure 10: Transitions in the "source" modelling unit

It is worth to notice that variables change their value if and only if an event has occurred. Such changes of state variables are specified by **transition** rules.

More precisely, a **transition** (defined for a current model unit) is a triple <e, G, P>, where:

- e is a declared event name,
- G is a **guard** i.e. a Boolean condition which is necessarily true before firing a transition
- P is an **action** which assign new values to a selection of state variables after the transition firing.

[2] When many flow variables are exchanged there is the possibility to regroup them. This is called "bus" in Cecilia Workshop and "record" in SimfiaNeo. Refer to 14.1.4 and 14.2.4.

[3] In this example the output corresponds to the internal value of the state.

A transition is **enabled only if its guard is satisfied**.

***Example*:** the transitions are enabled in the initial state, when St is ok, as illustrated Figure 10.

The following keywords and notations are used to write one transition:

```
        trans <Guard> |- <Event identifier> -> <Action> ;
[3.0]   transition <Event identifier>: <Guard> -> <Action> ;
```

where

<Guard> is a <Boolean Expression> and

<Action> is at set of assignments of values to state variables such that

`<State variable identifier>:= <Expression value>;` and

`<Expression value>` is a value of the domain of the state variable

***Examples*:**

```
trans
    (St = State_OK) |- fail_loss -> St := State_LOST;
    (St = State_OK) |- fail_err -> St := State_ERR;
```

State variables are modified only by actions of transitions. Conversely, actions of transition cannot modify a flow variable.

The order of the firing transition depends on the tool in use. If a specific order of triggering is necessary, priorities have to be defined.

It is possible to modify several states with the same transition.

4.5 - B - Assertions in a modelling unit

The **combinatorial dependencies between flow and state variables** are written with an **assertion.**

***Example*:** In source, the dependency between the output flow Out and the state variable St is declared simply as follows.

assert
`  Out = if (St = State_LOST)`
`            then LOST`
`      else if (St = State_ERR)`
`            then ERR`
`      else OK;`
	`[3.0]` **assertion**
	`   Out := St;`

The expression "A=B" used in the first versions of the languages means only that A and B shall have the same value: the dependency is not explicitly oriented by the user, it will be derived by the simulator.

The expression "A:=B" means that the value of the expression B is assigned to a variable A : the dependency is oriented by the user, the value of A is derived from the evaluation of the expression B. This choice makes explicit the dataflow constraint over the assertion part.

For **dataflow model, the assertion shall define univocally the value of an output flow** according to the values of input flows and/or state variables, like complete and consistent decision tables. Univocally means that the output is defined taking into account all possible cases : *a single value* is assigned to each output flow in all possible system configurations.

A good practice is to use the following pieces of syntax:

```
<Output flow identifier> = case {
      <Expression Boolean 1> : <expression value 1>,
      <Expression Boolean 2> : <expression value 2>,
      …
      else <expression value n>};
```

Where the <Expression Boolean i> are built with conditions over input flows or state variables and

the <expression value i> provide a value of the output flow domain.

Such pieces of code have the following interpretation:

"If the expression Boolean 1 is true, then the output flow value is expression value 1

Else if the expression 2 is true then the output flow value is expression value 2

Else … the output flow value is expression value n"

It defines univocally the value of the output flow because all combination of cases are covered and are mutually exclusive.

Another way is to use the following syntax:

```
If <Expression Boolean 1> then <Output flow identifier> = <expression value 1>
Else if <Expression Boolean 2> then <Output flow identifier> = <expression value 1>
Else <Output flow identifier> = <expression value n>
```

Example**:** The logical component `Comparator` is used to decide whether two inputs have consistent values. The decision table in Figure 5 specifies the output flow of the comparison (true/false) depending on the quality (OK, LOST, ERR) of the two inputs flows. It assumes that the comparator can detect issues if and only if the input flows have different quality.

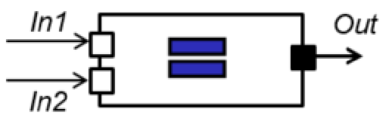
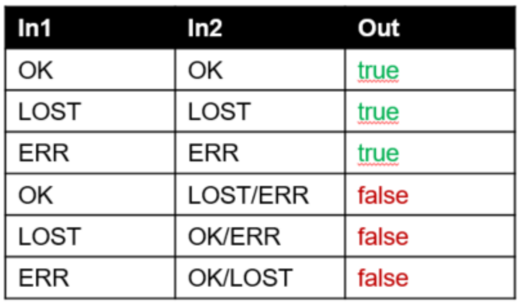


Figure 11: Flow variables of the "Comparator" modelling unit



In1	In2	Out
OK	OK	true
LOST	LOST	true
ERR	ERR	true
OK	LOST/ERR	false
LOST	OK/ERR	false
ERR	OK/LOST	false

Figure 12: : Relation between the values of the flow variables of the"Comparator" modelling unit

The truth table is an arbitrary choice.

***Example*:** The assertion of the comparator encodes the previous decision table.

```
    node Comparator
      flow
        In1:D_Command:in;
        In2:D_Command:in;
        Out:bool:out;
      assert
        Out = case {
          (In1 = In2) : true,
          else false
        };
    edon
```

```
[3.0] class Comparator
    D_Command In1, In2(reset = LOST);
    Boolean Out (reset = false);

  assertion
    Out := switch{
            case(In1 == In2) : true
            default : false
            };
end
```

The expression “A=B” used in the first versions of the languages is used both to write the Boolean condition “In1=In2” and to assign a value to output flow “Out=case{…};.

In the version 3.0, the symbol “==” is used for expressing Boolean condition whereas “:=” is reserved to the assignment of a unique value to variables.

In the following, we will mainly use the syntax 3.0 when we need to clarify semantics points. The syntax of the Data Flow version will be mainly used in association with case studies.

4.6 - B - Use and Connection of modelling units

In the following, we address the connection of the different modelling units and its associated meaning.

***Example*:** the following part of architecture can be written by reusing and connecting instances (copies) of the models of the source and comparator defined previously. The resulting code is given below.

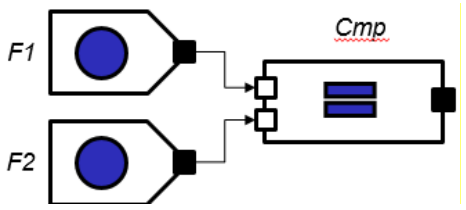


Figure 13: Connexion of two modelling units

```
node Comparator
// body of the node Comparator
 …
edon
node Source
// body of the node Source
…
edon
node main
  sub
    Cmp:Comparator;
    F1:Source;
    F2:Source;
  assert
    Cmp.In1 = F1.Out,
    Cmp.In2 = F2.Out;
 edon
```

```
[3.0]class Comparator
// body of the class Comparator
…
end
class Source
// body of the class Source
…
end
block System

    Comparator Cmp;
    Source F1, F2;

 assertion
    Cmp.In1 := F1.Out;
    Cmp.In2 := F2.Out;
end
```

The following syntax rules can be noticed.

- The reuses of model units inside other modelling unit have to be declared.

Example: `F1:Source;`

- The **names** of variables and events of **instantiated unit** are **prefixed** by the name of the instance followed by a dot.

Example: `F1.Out, Cmp.In1`

- Connections of instances are assertion linking inputs and outputs of two different instances.

Example: `assert Cmp.In1 = F1.Out`

- User comments are free text which are after the keyword **//** or between **/*** and ***/**. They can be read by users to understand the model.

Example: `// body of the node Comparator`

5 - B - Get started with failure propagation modelling

The purpose of this section is to provide the main principles and guidance's to start with AltaRica modelling without considering the tool used. In particular, we describe the different steps to follow.

The guiding thread of this section is the development of the example presented Figure 8.

Note that the models corresponding to this example, in Cecilia and SimfiaNeo tools are provided in the Appendix 10. It describes how to start, in practice, with the different MBSA tools, to simulate and to compute the cutsets and probabilities without replacing a dedicated training.

5.1 - B - Main principles and general guidance

In this section we describe the two first steps to define an AltaRica Data Flow model. Our intent is to provide high level guidances that will be detailed further in the document.

Step 1: Definition of the needs

Before starting a model, it is important to define the needs covered by this model. This is a universal good practice that strongly applies to MBSA. In other words, it is necessary to define what the model will be used for. This first step will allow defining the level of details of the model as well as the kind of modelling approach to choose.

Key questions before modelling are:

- Why are you modelling?
  - *For instance: To Support a Trade Off? To capitalize data and information? To support classical Analysis?*
- What are the analysis you want to perform?
  - ARP analysis involving probabilities computations? High level safety recommendations about the architecture functional independence? DAL analysis?
- What are the output you seek?
  - *For instance: FC quantification? Qualitative Analysis? Functional recommendations?*
- What are the input you need? What are the information available?
  - *For instance: Functions, Equipment, Electrical interfaces? Failure modes? Existing SE models with / without gateway to SA tool ?*
- Will the model evolve in size and in complexity?
  - *For instance: new functions, reconfigurations?*
  - *Is there any link with another model (for example a SE model, refer to [**REF 2**]) to be updated?*

The answers to those questions will help defining:

- ➢ If a fault propagation model is more relevant than a fault tree
- ➢ The level of information and the objects that will be used to build the model (for instance will you model functions or equipment? Will you model the interfaces in detail?)
- ➢ The information to be observed (in general at safety level we assess Failure Conditions)
- ➢ The level and granularity to be achieved (for instance you do not need a very high level of details for Trades-offs)
- ➢ The level and the way of representing the system functional behaviors in your model

Step 2: Definition of the model

In a second time, it is necessary to perform the following modelling activities.

- Definition of the perimeter of the system studied (its interfaces);
- Listing of a list of the main objects in the study perimeter: list of system components, list of failure conditions,…
- Expression of the failure conditions in relationship with the model perimeter
- Definition of the hypotheses about propagation equations inside each modelling unit or node that results both:
    - from potential error and failure modes of all components/ functions
    - from safety functions performed in the nominal case

In order to satisfy its intended use, the model has to:

- Fully cover the scope defined for the system studied;
- Enable the observation and the analysis of a set of failure conditions on this perimeter.

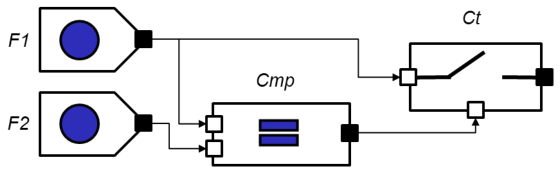
Note that the definition of the scope and of the observation allows to perform the validation of the model (refer to § 0). This preliminary work is important to avoid additional work such as additional iterations on the SA model.

5.2 - B - Modelling of a simple example: Command/Monitoring

In this section, we illustrate the proposed guidance using the COM/MON example presented in section 4.

5.2.1 - B - Description of the system

We consider the system already defined in section 4 and described in Figure 8.



Perimeter of the system studied

- General description: The purpose of the system is to send a command order F1 consolidated from two input commands. The system monitors the two orders F1 and F2. When F1 and F2 are different, an opening command is sent to the Contactor, the Contactor opens and the command is lost. When the Contactor does not receive the opening command, F1 is transmitted.
- Interfaces: Two input command F1 and F2 and one output command F1. The output assumptions are defined in chapter 5.2.3(c)(i).
- The system is composed of:
    - A comparator (Cmp)
    - A contactor (Ct)[4]

Safety requirement:

- FC1: erroneous output (Catastrophic)
- FC2: loss of output (Minor)

[4] In this example the contactor is a passive device without power supply that can alter its behavior (for instance: voltage too low to close the contactor). Then the behavior of the contactor only relies on inputs.

5.2.2 - B - Purpose and perimeter of the model (Step1)

The intent of our model is to provide the safety assessment of the architecture regarding the proposed safety requirements. We also want the system designer to understand our model in order to ease the communication and the validation activity.

We assume that we start from a blank model without existing libraries for instance that could reduce the effort but also bring some modelling constraints.

In order to perform the safety assessment of the architecture, our analysis will assess qualitatively and quantitatively (refer to the Get Started Kit to address the computation of the results) the provided Failure Conditions:

- FC1: erroneous output (Catastrophic)
- FC2: loss of output (Minor)

The targeted analyses require the computation of the CutSets and probabilities of each failure conditions.

Consequently, our model will integrate the different system components, the SFMEA failure modes and the functional reconfigurations of the system.

Because of the simplicity of the model, we choose not to model the external system sending F1 and F2. All the information is available.

5.2.3 - B - Definition of the model (Step2)

(a) Hypothesis about propagation equations inside each modelling unit

In this example, we base the definition of the local behavior inside each modelling on the hypothesis provided by the system designers. Note that this behavior could also be related and traced to the system specifications.

Our first hypotheses about the propagation equations inside each modelling unit (Cmp and Ct) are the following, considering the nominal or dysfunctional case:

- The comparator Cmp checks the equality of its two inputs :
  - when the inputs are different an isolation order is sent
  - when the inputs are not different no order is sent

- The contactor Ct
  - Is closed as long as the equality check is true. When it is closed, it transmits F1 output;
  - The contactor is open as soon as the equality check is wrong. When it is opened, it does not transmits F1 output;

Note these first hypotheses describe the model need. They require to be adapted in order to be included in the model definition.

In the following we will use the same naming that in this description.

[GP 1] General recommendation - Choose explicit names in the model including their type (Event, State, …)

In addition, in order to be able to model the system, its functional and dysfunctional behavior, we need to define the flow variables that will be used in the model and the modelling unit themselves. We base this definition on the AltaRica modelling language already defined in §4.

(b) Definition of the observer and flows

In this section we describe how we define the observers and the related flows and domains of our model.

For the FC1: erroneous output (Catastrophic) the system sends an erroneous command, for the FC2 the command is lost:

- FC1: erroneous output (Catastrophic) => the CMD at the system output is erroneous
- FC2: loss of output (Minor) => the CMD at the system output is lost

In order to be able to assess the failure conditions, we need to express these possibilities. Consequently we choose to define the command domain as `D_Command` = {OK, LOST, ERR}.

In addition, either from the documentation, or from informal exchanges, we know (or we hypothesize) that F1 and F2 in our example can provide:

- an erroneous value
- no value at all

In this example the input and the output of the system is a command with the same possible quality considering the possible failure of the F1 and F2 input data. Consequently we chose to use the same domain for F1 and F2 as for the output `D_Command`: {OK, LOST, ERR} .

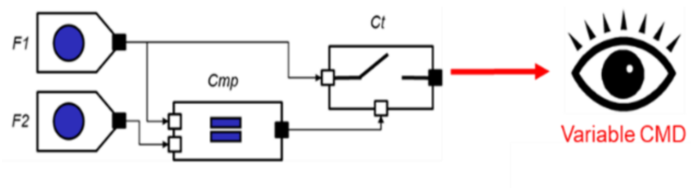


Figure 14: Observer – illustration

With the flow represented, the model became:

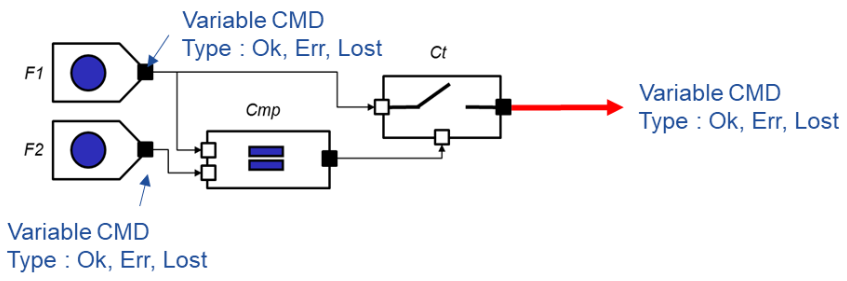


Figure 15: Illustration of the flows

In the tool modeling the observer variable CMD is replaced by two observers for each Failure Condition.

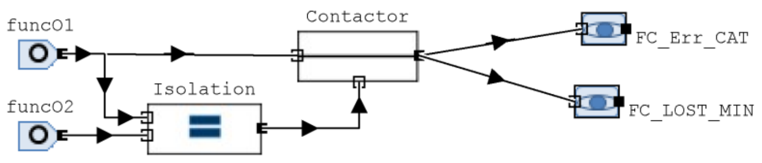


Figure 16: Observers – in the model

At this stage, we have not defined all the flows in the model. In particular, we need to define the flow from ***Cmp*** to ***Ct***.

From the description provided in (a) in case of detected difference between F1 and F2, an opening command is sent from the Comparator to the Contactor. We chose to model this command close to its functionality. This flow could be modelled as a Boolean but we choose to follow our good practice and be explicit. We define the domain of the isolation command as `D_isolation={Isolation, No_Isolation}`.

(c) Definition of the modelling unit

(i) The Sources – F1 and F2

The modelling units F1 and F2 have the same definition. Consequently, they can be seen as two instances of the same component Source. As the modelling of the source has already been presented in

Table 2: Modelling description of "Source", in this section we only concentrate on guidances and generality.

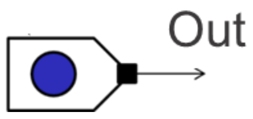


Figure 17: Illustration of the source

Definition of Input and Output:

- *Input flow:* In this particular example there is no input flow
- *Output flow*: we have defined previously that the output flow of the source domain is {OK,ERR,LOST}. The already dedicated type for a CMD is D_Command = {OK,ERR,LOST}.

Definition of the Internal State (St) of the modelling unit and Types

- The failure modes of the source are the followings : `fail_loss and fail_err`

[GP 2] Failure modes of components in MBSA are in principle not different from the failure modes in FTAs. Nevertheless, it is useful to reduce their number as much as possible in order to optimize your modelling and analysis. To do so and keep the traceability between your different analyses, it is possible in some cases to build a correspondence table between the FMES provided and the failure modes in the model.

- The internal state of the Source depends on the failure modes. For the sake of readability, we chose different names for the domain : D_Source {State_OK, State_LOST, State_ERR}

[GP 3] In some cases, it can be more efficient to use the same domain for the States and Flows. Nevertheless, this choice can make the model difficult to read for others.
[GP 4] Beginners should choose to use different types for different nature of flows, and different domains types for states and flows.

Definition of the Source modelling unit behavior

[GP 5] A table of truth linking the input, internal state and output variables is a valuable communication and validation tool regarding the modelling unit behavior

In the *Source* modelling unit the output flow value only depends on the input flows. The following provide a complete description of the modelling unit behavior.

Internal State: St	Out Type CMD Domain Ok, Lost, Err
State_OK	Ok
State_LOST	Lost
State_ERR	Err

Figure 18: Source Table of Truth

(ii) The Comparator (Cmp)

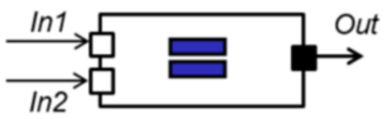


Figure 19: Illustration of the comparator Cmp

Definition of Input and Output :

- *Input flows:* there have been chosen in 0 in the domain D_Command={OK,ERR,LOST}.
- *Output flow*: it has also been chosen in 0 in the domain D_isolation={Isolation, No_Isolation}.

[GP 6] The definition of Types can allow to constrain you model : even though your model definition is the same you will not be able to connect two output and input if they do not have compatible Types and orientation (nominally it is impossible to connect two output for instance)

Definition of the Internal State (St) of the modelling unit and Types

This modelling unit has no failure consequently; there is no need to define an internal state.

Definition and Validation : Table of Truth

[GP 7] A table of truth linking the input, internal state and output variables is a valuable tool to communicate and document a modelling unit.

[GP 8] Comment the model locally providing the user with all the necessary information to understand the modelling intent

Note that in the *Comparator* modelling unit the output flow value only depends on the input flows in1 and in2.

In1 Type CMD Domain OK, Err, Lost	In2 Type CMD Domain OK, Err, Lost	Out Type boolean Domain: true/false ⇒ Comparison ⇒ isolation
Ok	**Ok**	**False**
Ok	**Lost**	**True**
Ok	**Err**	**True**
Lost	**Ok**	**True**
Lost	**Lost**	**False**
Lost	**Err**	**True**
Err	**Ok**	**True**
Err	**Lost**	**True**
Err	**Err**	**False**

Figure 20: Source Table of Truth

Table 3: Modelling description of "Cmp" provides the description of the modelling unit corresponding to its description, in AltaRica language.

Start of unit definition	*node Source*

Declarations of flow variables	***flow***
	
	Out: D_isolation:out;
	
	In1:FailType:in;
	
	In2:FailType:in;

Unit assertions	***assert*** ***// If Equal -> isolation*** ***Out = case {*** ***(In1 = In2) : false,*** ***else*** ***true***

Table 3: Modelling description of "Cmp"

(iii) The Contactor (Ct)

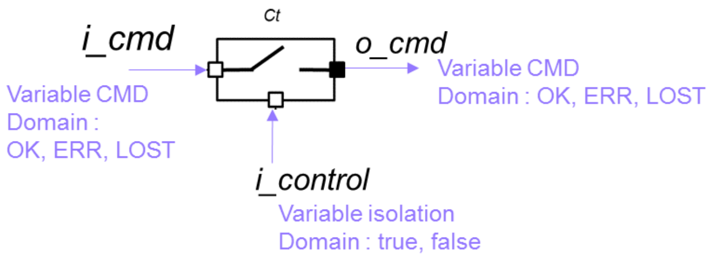


Figure 21: Illustration of the Contactor (Ct)

Definition of Input and Output :

- *Input flows:*
  - i_cmd is a Command of the same nature of F1 and F2 as discussed in 0, consequently its domain type is D_Command={OK,ERR,LOST}.
  - i_control is an isolation command received form the Comparator. As per 0 and (ii), consequently its domain type is D_isolation={Isolation, No_Isolation}.
- *Output flow*: o_comd is a Command of the same nature of in1 as discussed in 0, consequently its domain type is D_Command={OK,ERR,LOST}.

Definition of the Internal State (St) of the modelling unit and Types

- The failure modes of the contactor are the followings : `stuck_open and stuck_closed`
- The internal state of the Contactor depends on its failure modes. We define a generic domain `D_Switch={ Ok, Stuck_Opened, Stuck_Closed}`

Definition and Validation: Table of Truth

In the *Contactor* modelling unit the output flow value only depends on the input flows. The following provide a complete description of the modelling unit behavior.

Internal State OK, failed_closed failed_open	i_cmd Type CMD Domain OK, ERR, LOST	i_control Type isolation Domain true false	o_cmd Type CMD Domain OK, ERR, LOST
State_OK	Ok	False	Ok
State_OK	Ok	True	Lost
State_OK	Lost	False	Lost
State_OK	Lost	True	Lost
State_OK	Err	False	Err
State_OK	Err	True	Lost
State_Stuck_Open	Err or Lost or Ok	Err or Lost or Ok	Lost
State_Stuck_Closed	Ok	True	Ok
State_Stuck_Closed	Ok	False	Ok
State_Stuck_Closed	Lost	True	Lost
State_Stuck_Closed	Lost	False	Lost
State_Stuck_Closed	Err	True	Err
State_Stuck_Closed	Err	False	Err

Figure 22: Contactor Table of Truth

In addition the Table 4 below provides the description of the Contactor modelling unit corresponding to its description, in AltaRica language.

Start of unit definition	*node Source*

Declarations of flow variables	***flow***
	
	o_cmd: D_Command:out;
	
	i_cmd:D_Command:in;
	
	i_opening_cmd: D_isolation:in;

Declarations and initialisations of state variables	***State***
	
	St: D_Switch;
	
	Init St:=Ok;

Unit transitions	***trans***
	
	(St=Ok) |-stuck_open ->St:=Stuck_Opened;
	
	(St=Ok) |-stuck_closed ->St:=Stuck_Closed;

Unit assertions	***assert*** ***o_cmd = case {*** ***(St=Stuck_Closed): i_cmd,*** ***(St=Stuck_Opened): Lost,*** ***(i_opening_cmd=Isolation) and (St=Ok): Lost,*** ***else*** ***i_cmd*** ***}***

Table 4: Modelling description of "Ct"

5.3 - I - Flattening structured models

Before the assessment an AltaRica model is flattened "by the tool", i.e. it is transformed into a Guarded Transition System by means of rewriting rules. All the structure is removed. The flattened model is composed of a set of variables, a set of events, a set of transitions, an assertion and a default or initial assignment.

Formally, a **Guarded Transition System is a quintuple** < V, E, T, A, ι>, where

- V = S ∪ F is a disjoint set of state variables S and flow variables F;
- E is a set of events;
- T is a set of transitions, a **transition** is a triple <*e, G, P*>, where *e* is an event of E, *G* is a Boolean expression built over the variables of V, called a **guard**, and *P* is an instruction, called an **action** or a **post-condition**;
- A is an instruction, called an **assertion**;
- ι is a default or initial variable assignment.

Example of Guarded Transition System (flattened model):

A Guarded Transition System representing the COM-MON example, or in other words its flatten model, is as follows (we use the syntaxe of AltaRica 3.0 for the description of expressions and instructions):

- The set of state variables S = {F1.St, F2.St},
- The set of flow variables F = {F1.Out, F2.Out, Cmp.In1, Cmp.In2, Cmp.Out, Ct.In, Ct.Out, Ct.CloseCondition},
- The set of events E = {F1.fail_err, F1.fail_loss, F2.fail_err, F2.fail_loss},
- The set of transitions {
    1. <F1.fail_loss, F1.St==State_OK, F1.St:= State_LOST>
    2. <F1.fail_err, F1.St== State_OK, F1.St:= State_ERR>
    3. <F2.fail_loss, F2.St== State_OK, F2.St:= State_LOST>
    4. <F2.fail_err, F2.St== State_OK, F2.St:= State_ERR>}
- The assertion A =
    1. F1.Out := if (F1.St = State_LOST) then LOST else if (F1.St = State_ERR) then ERR else OK;
    2. F2.Out := if (F2.St = State_LOST) then LOST else if (F2.St = State_ERR) then ERR else OK;
    3. Cmp.In1 :=F1.Out;
    4. Cmp.In2 := F2.Out;
    5. Cmp.Out := if (Cmp.In1 == Cmp.In2) then true else false;
    6. Ct.In := F1.Out;
    7. Ct.CloseCondition := Cmp.Out;
    8. Ct.Out := if Ct.CloseCondition then Ct.In else LOST;
- The default variable assignment ι = {F1.St= State_OK, F2.St= State_OK}.

5.4 - B - *Good Practice summary*

This section aims to summarize the good practices presented in the above chapters. These good practices are not mandatory but guideline coming from the experience of the S2C project.

[GP 1] General recommendation - Choose explicit names in the model.

[GP 2] Failure modes of components in MBSA are in principle not different from the failure modes in FTAs. Nevertheless, it is useful to reduce their number as much as possible in order to optimize your modelling and analysis. To do so and keep the traceability between your different analyses, it is possible in some cases to build a correspondence table between the FMES provided and the failure modes in the model.

[GP 3] In some cases, it can be more efficient to use the same domain for the States and Flows. Nevertheless, this choice can make the model difficult to read for others.

[GP 4] Beginners should choose to use different types for different nature of flows, and different domains types for states and flows.

[GP 5] A table of truth linking the input, internal state and output variables is a valuable communication and validation tool regarding the modelling unit behavior.

[GP 6] The definition of Types can allow to constrain you model : even though your model definition is the same you will not be able to connect two outputs and input if they do not have compatible Types and orientation (nominally it is impossible to connect two output for instance).

[GP 7] A table of truth linking the input, internal state and output variables is a valuable tool to communicate and document a modelling unit.

[GP 8] Comment the model locally providing the user with all the necessary information to understand the modelling intent.

5.5 - B - *Latent, dormant or hidden failures*

As in all safety assessments in aeronautical domains, there is a need to address the latent and dormant failures.
Indeed, if one or more failed elements in the system can persist for multiple flights (latent, dormant, or hidden failures), the calculation should consider the relevant exposure times (e.g. time intervals between maintenance and operational checks/ inspections).

The MBSA based on AltaRica Dataflow allows to define a dormancy / latency for each basic event with the same characteristics than in a FTA based analysis.

6 Simulation – general definitions

One interest of the proposed MBSA approach is to provide simulation capabilities. The simulation aims at checking whether the propagation equation are complete and compliant with the knowledge about the system and its failures. It allows simulating what happens when a fault occurs or when a nominal event (e.g. a reconfiguration) is triggered.

The following section presents different aspects of the simulation modelling and use. Different examples to illustrate this capacity are provided in the document.

6.1 Basic definitions

6.1.1 - B - Simulation principles

The Simulation is defined as a sequence of trigger able events, starting from an initial safe state. In particular these events can be the cut set of chosen Failure Conditions (i.e. one triggers the event(s) of the cut). It allows observing the results and checking the model.

There are two kind of simulation used in MBSA purposes

- "step by step" or interactive, meaning there is no temporality only the order of the triggered events run the simulation. Refer to §7 for further details
- Timed or stochastic models, meaning that triggering of the event is controlled by a function from time. This aspect is discussed in §6.1.3

Different examples to illustrate this capacity are provided in the document. The impact of the model characteristics on the simulation itself is discussed in §0.

6.1.2 - I - Reachability graph

A reachability graph for AltaRica models is a formal representation of all possible configurations a given model can reach when following its defined transitions. For most exploitations of an AltaRica model, this graph is not directly computed, as its size faces a combinatorial explosion. Instead, it is gradually explored with each transition. In order to ease the discussion, in the following sections, the whole reachability graph will be displayed on our small example.

A **configuration** is an assignment of state and flow variables, each variable of V has a value of its domain.

Example of Configuration:

In the COM-MON model given Figure 8 the configuration is defined by the values of the following variables:

- State variables F1.St, F2.St,
- Flow variables F1.Out, F2.Out, Cmp.In1, Cmp.In2, Cmp.Out, Ct.In, Ct.Out, Ct.CloseCondition.

A possible assignment of state and flows variables is

$\sigma_i =$ {F1.St=OK, F2.St=OK, F1.Out=OK, F2.Out=OK, Cmp.In1=OK, Cmp.In2=OK, Cmp.Out=true, Ct.In=OK, Ct.CloseCondition=true, Ct.Out=OK}.

The **initial configuration** is calculated as follows. First, all the state variables receive their initial values from ι, then the assertion A is applied to calculate the value of the flow variables. Let denote the initial configuration

$\sigma_0 = A(\iota)$.

Example of Initial configuration:

In the initial configuration, the value of F1.St and F2.St is OK.

Then the assertion is applied to calculate the value of the flow variables:

1. F1.Out := if (F1.St = State_LOST) then LOST else if (F1.St = State_ERR) then ERR else OK;
2. F2.Out := if (F2.St = State_LOST) then LOST else if (F2.St = State_ERR) then ERR else OK;
3. Cmp.In1 :=F1.Out;
4. Cmp.In2 := F2.Out;
5. Cmp.Out := if (Cmp.In1 == Cmp.In2) then true else false;
6. Ct.In := F1.Out;
7. Ct.CloseCondition := Cmp.Out;
8. Ct.Out := if Ct.CloseCondition then Ct.In else LOST;

So, F1.Out = OK from the first assignment, F2.Out = OK from the second assignment, Cmp.In1 = OK from the third assignment, Cmp.In1 = OK from the fourth assignment, Cmp.out = true from the fifth assignment, Ct.In = OK from the 6th assignment, Ct.CloseCondition = true from the 7th assignment and Ct.Out = OK from the 8th assignment.

Finally, the initial configuration of the COM-MON model is as follows:

$\sigma_0 =$ {F1.St=State_OK, F2.St=State_OK, F1.Out=OK, F2.Out=OK, Cmp.In1=OK, Cmp.In2=OK, Cmp.Out=true, Ct.In=OK, Ct.CloseCondition=true, Ct.Out=OK}.

A transition is **enabled** in the current configuration if the value of its guard is true in the current configuration.

Example of Enabled transition.

In the model of the COM-MON there are four transitions:

1. <F1.fail_loss, F1.St==State_OK, F1.St:= State_LOST>
2. <F1.fail_err, F1.St== State_OK, F1.St:= State_ERR>
3. <F2.fail_loss, F2.St== State_OK, F2.St:= State_LOST>
4. <F2.fail_err, F2.St== State_OK, F2.St:= State_ERR>

In the initial configuration the value of F1.St is State_OK, so both F1.fail_loss and F1.fail_err are enabled;

F2.St is also State_OK, so both transitions F2.fail_loss and F2.fail_err are enabled.

A Guarded Transition System defines a **Reachability graph** $R = <\Sigma, \Theta>$, where

- Σ is a set of configurations of the AltaRica model;
- Θ is a set of edges $<\sigma_1, e, \sigma_2>$, where $\sigma_i \in \Sigma$ are configurations and $e \in E$ is an event of the AltaRica model.

The **Reachability graph** is calculated as follows.

First, the initial configuration $\sigma_0 = A(\iota)$ is calculated. It is added to the set of configurations Σ.

Second, enabled transitions are calculated in the initial configuration. To calculate next configurations, each enabled transition is fired independently.

When an enabled transition *t=<e, G, P>* is fired, then first its action is executed to calculate the new value of state variables, then the assertion is applied to calculate the new value of flow variables. Let denote the new configuration

$\sigma_i = A(P(\sigma_0))$.

The new configuration is added to the set of configurations Σ and the edge $<\sigma_0, e, \sigma_i>$ is added to the set of edges Θ. The process iterates until no new configuration is reached.

Be aware that the Reachability graph of an AltaRica model may have an infinite number of configurations.

The number of configurations of the Reachability graph is exponential on the number of state variables in the AltaRica model.

For example, considering a system made of n components, each component has a Boolean state variable, representing whether it is working or not, and a transition, making the component change its state from “working” to “failed”. The whole reachability graph of this system has 2^n configurations.

In practice, the reachability graph is never generated in full but only partially explored.

Example of Firing of an enabled transition:

Let us consider again the COM-MON example given Figure 8.

In the initial configuration, four transitions are enabled: F1.fail_loss, F1.fail_err, F2.fail_loss and F2.fail_err.

Let us fire the transition F1.fail_err. First, the action of the transition is executed and F1.St receives the value State_ERR. Second, the assertion is applied.

So, F1.Out = ERR from the first assignment, F2.Out = OK from the second assignment, Cmp.In1 = ERR from the third assignment, Cmp.In2 = OK from the fourth assignment, Cmp.Out = false from the fifth assignment, Ct.In = ERR from the 6th assignment, Ct.CloseCondition = false from the 7th assignment, Ct.Out = LOST from the 8th assignment.

Finally, the new configuration of the COM-MON model is as follows:

σ_1 ={F1.St= State_ERR, F2.St= State_OK, F1.Out=ERR, F2.Out=OK, Cmp.In1 = ERR, Cmp.In2=OK, Cmp.Out = false, Ct.In = ERR, Ct.CloseCondition=false, Ct.Out=LOST}

Example of Reachability graph:

The complete Reachability graph of the AltaRica model of the COM-MON is given in Section 14.5.

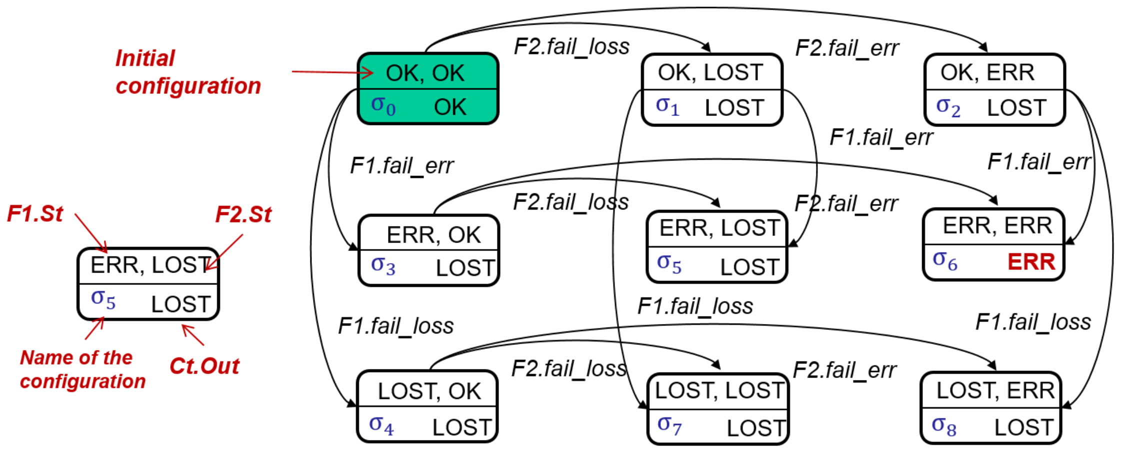


Figure 23: Reachability graph

For the sake of simplicity, for each configuration of the graph we give only the value of three variables F1.St, F2.St and Ct.Out. Each edge of the graph is labelled by an event of the model.

An **execution (or a trace)** of an AltaRica model is a succession of configurations and transitions defined recursively as follows:

- σ_0 is an empty execution (or trace),
- If $\Lambda_n = \sigma_0 \xrightarrow{t_1} \sigma_1 \xrightarrow{t_2} \dots \xrightarrow{t_n} \sigma_n$ is an execution of the model then so is $\Lambda_{n+1} = \Lambda_n \xrightarrow{t_{n+1}} \sigma_{n+1}$iff

- The transition $t_{n+1} =< e_{n+1}, G_{n+1}, P_{n+1} >$ is enabled in the configuration σ_n, i.e. the guard G_{n+1} is true in σ_n;
- The configuration $\sigma_{n+1} = A(P_{n+1}(\sigma_n))$ is obtained by firing the transition t_{n+1}.

A trace corresponds to a path in the Reachability graph, starting from the initial configuration and ending in some configuration of the model.

In the Reachability graph, we are interested by paths leading from the initial configuration to the configurations, where some conditions are satisfied (for example, a Failure condition occurs). These paths are also called sequences.

Example (Execution trace):

In the reachability graph given in §6.1, the traces leading from the initial configuration to the configurations, where Ct.Out=ERR are as follows:

$$\sigma_0 \xrightarrow{F2.fail\_err} \sigma_2 \xrightarrow{F1.fail\_err} \sigma_6$$

$$\sigma_0 \xrightarrow{F1.fail\_err} \sigma_3 \xrightarrow{F2.fail\_err} \sigma_6$$

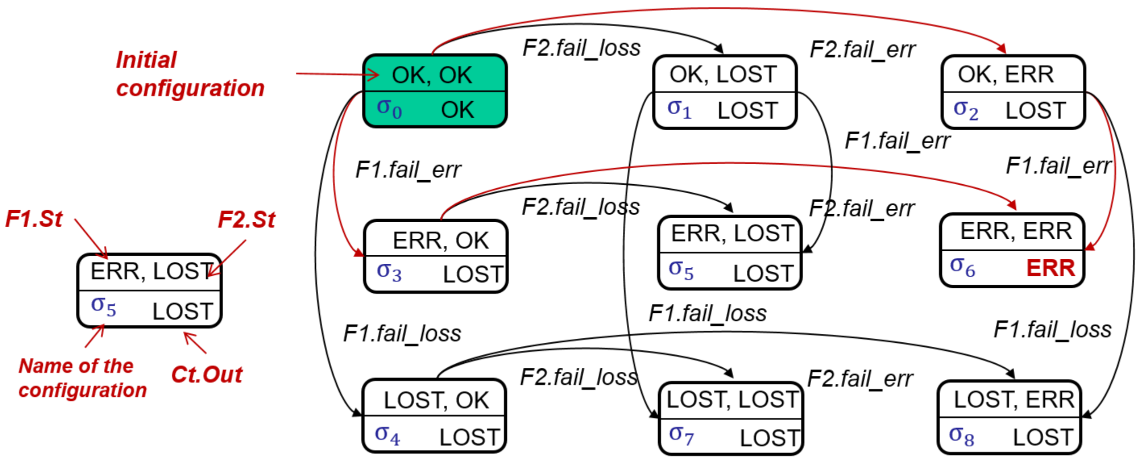


Figure 24: Traces in the reachability graph

In the reachability graph given 6.1, the execution traces leading from the initial configuration to the configurations where Ct.Out = ERR are marked in red.

The traces leading from the initial configuration to the configurations, where Ct.Out=LOST are as follows:

$$\sigma_0 \xrightarrow{F2.fail\_loss} \sigma_1$$

$$\sigma_0 \xrightarrow{F1.fail\_loss} \sigma_4$$

$$\sigma_0 \xrightarrow{F1.fail\_err} \sigma_3$$

$$\sigma_0 \xrightarrow{F2.fail\_err} \sigma_2$$

$$\sigma_0 \xrightarrow{F2.fail\_err} \sigma_3 \xrightarrow{F2.fail\_loss} \sigma_5$$

$$\sigma_0 \xrightarrow{F2.fail\_loss} \sigma_1 \xrightarrow{F1.fail\_err} \sigma_5$$

$$\sigma_0 \xrightarrow{F1.fail\_loss} \sigma_4 \xrightarrow{F2.fail\_loss} \sigma_7$$

$$\sigma_0 \xrightarrow{F2.fail\_loss} \sigma_1 \xrightarrow{F2.fail\_loss} \sigma_7$$

$$\sigma_0 \xrightarrow{F1.fail\_loss} \sigma_4 \xrightarrow{F2.fail\_err} \sigma_8$$

$$\sigma_0 \xrightarrow{F2.fail\_err} \sigma_2 \xrightarrow{F1.fail\_loss} \sigma_8$$

Example of Sequences:

In the reachability graph given in §6.1, sequences of events leading from the initial configuration to the configurations where Ct.Out = ERR are as follows:

- F2.fail_err, F1.fail_err,
- F1.fail_err, F2.fail_err.

6.1.3 - A - Timed and stochastic models

Stochastic or deterministic delays can be associated with the events of an AltaRica model.

Deterministic transitions are defined by the probability distribution named Dirac(τ), where τ is a fixed delay, a non-negative number. If a deterministic transition becomes enabled at time d, then it should be fired exactly at time d + τ.

A special case of deterministic transitions are **immediate** transitions, defined by Dirac(0). They should be fired as soon as their guards become true (before any other transition).

Stochastic transitions are defined by stochastic probability distributions (for example, exponential, Weibull, etc.). If a stochastic transition becomes enabled at time d, it should be fired at time d + τ, where τ is a random delay calculated according to the probability distribution associated with the event e.

When performing cuts or sequences generation, these computations are performed qualitatively, before then assessing each cut/sequence probability. As a consequence, the choice of a stochastic law (e.g. exponential vs. Weibull, or different numerical parameters) has no consequence on the list of cuts or sequences, but only on the associated numerical probabilities. On the contrary, numerical parameters of deterministic laws can modify the list of found sequences.

Let us denote by $delay(e)$ a delay calculated for the event e according to its probability distribution, which is a random value for stochastic events and is equal to τ for deterministic events.

Example of Stochastic and deterministic transitions:

To illustrate the use of deterministic and stochastic transitions let us consider again the COM-MON example given Figure 8. The failures of the functions are represented by stochastic events, which are exponentially distributed with given failure rates.

Now we represent a contactor with memory, i.e. if the contactor receives the order to open (the value of the variable Ct.CloseCondition becomes false) then it remains open even if it receives later the order to close again (the value of the variable Ct.CloseCondition becomes true again). To do that, we modify the previous model and define:

- A Boolean state variable Ct.Open which is false in the initial configuration,
- A deterministic event Ct.Open_ct with a delay Dirac(0),
- An immediate transition <Ct.Open_ct, not Ct.CloseCondition and not Ct.Open, Ct.Open := true>, which should be fired as soon as its guard "not Ct.CloseCondition and not Ct.Open" becomes true,
- An assertion "Ct.Out := if Ct.Open then LOST else Ct.In".

The whole Guarded Transition System is as follows (changes in comparison with the previous model are marked in red):

- The set of state variables S = {F1.St, F2.St, Ct.Open},
- The set of flow variables F = {F1.Out, F2.Out, Cmp.In1, Cmp.In2, Cmp.Out, Ct.In, Ct.Out, Ct.CloseCondition},
- The set of events E = {F1.fail_err, F1.fail_loss, F2.fail_err, F2.fail_loss, Ct.open_ct},
- The set of transitions {
    1. <F1.fail_loss (stochastic), F1.St==OK, F1.St:=LOST>
    2. <F1.fail_err (stochastic), F1.St==OK, F1.St:=ERR>
    3. <F2.fail_loss (stochastic), F2.St==OK, F2.St:=LOST>
    4. <F2.fail_err (stochastic), F2.St==OK, F2.St:=ERR>
    5. < Ct.Open_ct (Dirac(0)), not Ct.CloseCondition and not Ct.Open, Ct.Open := true >}
- The assertion A =
    1. F1.Out := F1.St;
    2. F2.Out := F2.St;
    3. Cmp.In1 :=F1.Out;
    4. Cmp.In2 := F2.Out;
    5. Cmp.Out := if (Cmp.In1 == Cmp.In2) then true else false;
    6. Ct.In := F1.Out;
    7. Ct.CloseCondition := Cmp.Out;
    8. Ct.Out := if Ct.Open then LOST else Ct.In;
- The default variable assignment ι = {F1.St=OK, F2.St=OK, Ct.open=false}.

Example of Reachability graph with stochastic and deterministic transitions:

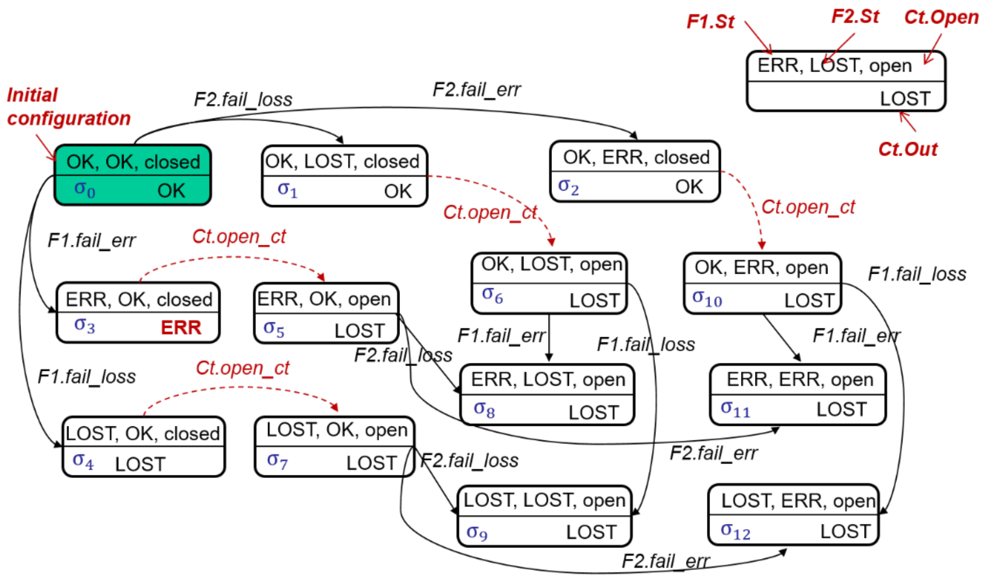


Figure 25: Reachability graph of the COM-MON with deterministic transitions

The chapter 6.1.2 shows the reachability graph of the COM-MON with deterministic (immediate) transitions. Immediate transitions are marked in red with dashed lines. As in the previous model, in the initial configuration four transitions are enabled:

- F1.fail_loss (stochastic),
- F1.fail_err (stochastic),
- F2.fail_loss (stochastic),
- F2.fail_err (stochastic).

All of them are stochastic and any of them can be fired in the initial configuration.

When the transition F1.fail_err is fired, the new configuration is calculated:

- From the action of the transition F1.St=ERR;
- After the application of the assertion Ct.Out = ERR.

In this configuration, there are three enabled transitions:

- F2.fail_loss (stochastic),
- F2.fail_err (stochastic),
- Ct.open_ct (Dirac(0)).

As Ct.open_ct is an immediate transition, it should be fired before the transitions F2.fail_loss and F2.fail_err.

It is fired first:

- From the action of the transition Ct.Open = true;
- After the application of the assertion Ct.Out = LOST.

In this new configuration, two transitions are enabled:

- F2.fail_loss (stochastic),
- F2.fail_err (stochastic).

Unlike the previous example, if the transition F2.fail_err is fired, the contactor Ct remains open and Ct.Out remains LOST even if Ct.CloseCondition becomes true again.

Execution traces leading from the initial configuration to the configurations where Ct.Out = ERR is

$$\sigma_0 \xrightarrow{F1.fail\_err} \sigma_3$$

Execution traces leading from the initial configuration to the configurations where Ct.Out = LOST are

$$\sigma_0 \xrightarrow{F1.fail\_loss} \sigma_4$$

$$\sigma_0 \xrightarrow{F1.fail\_err} \sigma_3 \xrightarrow{Ct.open\_ct} \sigma_5$$

$$\sigma_0 \xrightarrow{F1.fail\_loss} \sigma_4 \xrightarrow{Ct.open\_ct} \sigma_7$$

$$\sigma_0 \xrightarrow{F1.fail\_err} \sigma_3 \xrightarrow{Ct.open\_ct} \sigma_5 \xrightarrow{F2.fail\_loss} \sigma_8$$

$$\sigma_0 \xrightarrow{F1.fail\_err} \sigma_3 \xrightarrow{Ct.open\_ct} \sigma_5 \xrightarrow{F2.fail\_err} \sigma_{11}$$

$$\sigma_0 \xrightarrow{F1.fail\_loss} \sigma_4 \xrightarrow{Ct.open\_ct} \sigma_7 \xrightarrow{F2.fail\_loss} \sigma_9$$

$$\sigma_0 \xrightarrow{F1.fail\_loss} \sigma_4 \xrightarrow{Ct.open\_ct} \sigma_7 \xrightarrow{F2.fail\_err} \sigma_{12}$$

$$\sigma_0 \xrightarrow{F2.fail\_loss} \sigma_1 \xrightarrow{Ct.open\_ct} \sigma_6$$

$$\sigma_0 \xrightarrow{F2.fail\_loss} \sigma_1 \xrightarrow{Ct.open\_ct} \sigma_6 \xrightarrow{F1.fail\_err} \sigma_8$$

$$\sigma_0 \xrightarrow{F2.fail\_loss} \sigma_1 \xrightarrow{Ct.open\_ct} \sigma_6 \xrightarrow{F1.fail\_loss} \sigma_9$$

$$\sigma_0 \xrightarrow{F2.fail\_err} \sigma_2 \xrightarrow{Ct.open\_ct} \sigma_{10}$$

$$\sigma_0 \xrightarrow{F2.fail\_err} \sigma_2 \xrightarrow{Ct.open\_ct} \sigma_{10} \xrightarrow{F1.fail\_err} \sigma_{11}$$

$$\sigma_0 \xrightarrow{F2.fail\_err} \sigma_2 \xrightarrow{Ct.open\_ct} \sigma_{10} \xrightarrow{F1.fail\_loss} \sigma_{12}$$

Sometimes it is convenient to mask some configurations and edges in the reachability graph. It is the case of configurations with outgoing edges labelled by immediate events. Indeed, we would like to observe only the configurations without outgoing edges labelled by immediate events and only the edges labelled by non-immediate events.

For example, in the reachability graph given in §6.1.2, we would keep only the configurations without outgoing edges labelled by immediate events.

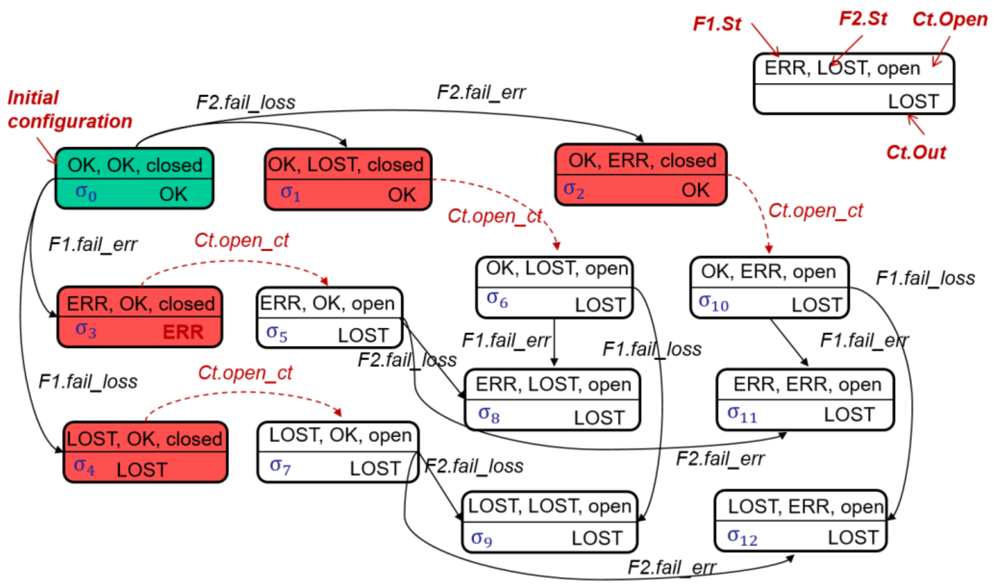


Figure 26: Masked nodes and edges

In the reachability graph given in §6.1.2, masked configurations and edges are marked in red.

Therefore, we mask the nodes $\sigma_1, \sigma_2, \sigma_3, \sigma_4$ and the outgoing edges labelled by the immediate event Ct.open_ct. In this case, the traces leading to the configurations where Ct.Out = LOST would be as follows:

$$\sigma_0 \xrightarrow{F1.fail\_err} \sigma_5$$

$$\sigma_0 \xrightarrow{F1.fail\_loss} \sigma_7$$

$$\sigma_0 \xrightarrow{F1.fail\_err} \sigma_5 \xrightarrow{F2.fail\_loss} \sigma_8$$

$$\sigma_0 \xrightarrow{F1.fail\_err} \sigma_5 \xrightarrow{F2.fail\_err} \sigma_{11}$$

$$\sigma_0 \xrightarrow{F1.fail\_loss} \sigma_7 \xrightarrow{F2.fail\_loss} \sigma_9$$

$$\sigma_0 \xrightarrow{F1.fail\_loss} \sigma_7 \xrightarrow{F2.fail\_err} \sigma_{12}$$

$$\sigma_0 \xrightarrow{F2.fail\_loss} \sigma_6$$

$$\sigma_0 \xrightarrow{F2.fail\_loss} \sigma_6 \xrightarrow{F1.fail\_err} \sigma_8$$

$$\sigma_0 \xrightarrow{F2.fail\_loss} \sigma_6 \xrightarrow{F1.fail\_loss} \sigma_9$$

$$\sigma_0 \xrightarrow{F2.fail\_err} \sigma_{10}$$

$$\sigma_0 \xrightarrow{F2.fail\_err} \sigma_{10} \xrightarrow{F1.fail\_err} \sigma_{11}$$

$$\sigma_0 \xrightarrow{F2.fail\_err} \sigma_{10} \xrightarrow{F1.fail\_loss} \sigma_{12}$$

In the new Reachability graph with masked configurations and edges, there is no traces leading from the initial configuration to the configuration where Ct.Out=ERR. Indeed, in our example the reconfiguration is perfect, only functions F1 and F2 may fail in operation. Therefore, when F1.fail_err or F2.fail_err occurs, the contactor is open and Ct.Out becomes LOST.

In order to take into account in the simulation the probability distributions associated to the events of AltaRica models, in the following we define a scheduler and a timed/stochastic execution or trace. The defined principle is used in the stochastic simulation of AltaRica models.

A **scheduler** $\mathit{\Gamma}: T \rightarrow R^{+} \cup \{\infty\}$ of an AltaRica model is a function that associates to each transition t=<e,G,P> of the set T its firing date according to the probability distribution of the event e.

At the beginning of a simulation, for each transition t =<e, G, P> from T the scheduler $\mathit{\Gamma}_0$ is calculated as follows:

- $delay(e)$, if the transition t=<e, G, P> is enabled in the initial configuration;
- ∞, otherwise.

A **timed/stochastic execution (or timed/stochastic trace)** of an AltaRica model is defined as follows:

- $< \sigma_0, d_0, \mathit{\Gamma}_0 >$ is an empty execution, with the initial configuration $\sigma_0 = A(\iota)$, the current firing date $d_0 = 0$ and the initial scheduler $\mathit{\Gamma}_0$;
- If $\mathit{\Lambda}_n = < \sigma_0, d_0, \mathit{\Gamma}_0 > \xrightarrow{t_1} < \sigma_1, d_1, \mathit{\Gamma}_1 > \xrightarrow{t_2} \dots \xrightarrow{t_n} < \sigma_n, d_n, \mathit{\Gamma}_n >$ is an execution of the model then so is $\mathit{\Lambda}_{n+1} = \mathit{\Lambda}_n \xrightarrow{t_{n+1}} < \sigma_{n+1}, d_{n+1}, \mathit{\Gamma}_{n+1} >$ iff
  1. The transition $t_{n+1} =< e_{n+1}, G_{n+1}, P_{n+1} >$ is enabled in the configuration σ_n, i.e. the guard G_{n+1} is true in σ_n, and its firing date is one of the smallest in the scheduler $\mathit{\Gamma}_n(t_{n+1}) \leq \mathit{\Gamma}_n(t)\ \forall\ t \in T$;
  2. The configuration $\sigma_{n+1} = A(P_{n+1}(\sigma_n))$ is obtained by firing the transition t_{n+1};
  3. $d_{n+1} = \mathit{\Gamma}_n(t_{n+1})$, the new current firing date equals to the firing date of the transition t_{n+1};
  4. The new scheduler is calculated as follows:
     - $\mathit{\Gamma}_{n+1}(t) = \mathit{\Gamma}_n(t)$ if the transition t was enabled and remains enabled after the firing of t_{n+1},
     - $\mathit{\Gamma}_{n+1}(t) = d_{n+1} + delay(e)$ if the transition was not enabled before and becomes enabled after the firing of t_{n+1} or $t = t_{n+1}$,
     - $\mathit{\Gamma}_{n+1}(t) = \infty$ if the transition t is not enabled.

Example of A timed execution:

Let us consider again the COM-MON pattern for safety architecture with stochastic and deterministic transitions. Here is presented one of the possible times executions of this model.

At the beginning of the simulation:

- $\sigma_0 =$ {F1.St=OK, F2.St=OK, Ct.open=false, F1.Out=OK, F2.Out=OK, Cmp.In1=OK, Cmp.In2=OK, Cmp.Out=true, Ct.In=OK, Ct.CloseCondition=true, Ct.Out=OK};
- $d_0 = 0h;$
- $\mathit{\Gamma}_0$(F1.fail_err) = 4380h, $\mathit{\Gamma}_0$(F1.fail_loss) = 6340h, $\mathit{\Gamma}_0$(F2.fail_err) = 5150h, $\mathit{\Gamma}_0$(F2.fail_loss) = 5300h, $\mathit{\Gamma}_0$(Ct.open_ct) = ∞ (the firing dates of stochastic transitions have been calculated randomly).

According to the scheduler $\mathit{\Gamma}_0$ the smallest firing date is 4380, so F1.fail_err should be fired first.

After the firing of F1.fail_err:

- $\sigma_1 =$ {F1.St=ERR, F2.St=OK, Ct.open=false, F1.Out=ERR, F2.Out=OK, Cmp.In1=ERR, Cmp.In2=OK, Cmp.Out=false, Ct.In=ERR, Ct.CloseCondition=false, Ct.Out=ERR};
- $d_1 = 4380h;$
- $\mathit{\Gamma}_1$(F1.fail_err) =∞, $\mathit{\Gamma}_1$(F1.fail_loss) = ∞, $\mathit{\Gamma}_1$(F2.fail_err) = 5150h, $\mathit{\Gamma}_1$(F2.fail_loss) = 5300h, $\mathit{\Gamma}_1$(Ct.open_ct) = $4380 + 0 = 4380h.$

According to the scheduler $\mathit{\Gamma}_1$ the smallest firing date is 4380, so Ct.open_ct should be fired.

After the firing of Ct.open_ct:

- $\sigma_2 =$ {F1.St=ERR, F2.St=OK, Ct.open=true, F1.Out=ERR, F2.Out=OK, Cmp.In1=ERR, Cmp.In2=OK, Cmp.Out=false, Ct.In=ERR, Ct.CloseCondition=false, Ct.Out=LOST};
- $d_2 = 4380h$;
- Γ_2(F1.fail_err) =∞, Γ_2(F1.fail_loss) = ∞, Γ_2(F2.fail_err) = 5150h, Γ_2(F2.fail_loss) = 5300h, Γ_2(Ct.open_ct) = ∞.

According to the scheduler Γ_2, the smallest firing date is 5150, so the next transition to fire is F2.fail_err.

After the firing of F2.fail_err:

- $\sigma_3 =$ {F1.St=ERR, F2.St=ERR, Ct.open=true, F1.Out=ERR, F2.Out=ERR, Cmp.In1=ERR, Cmp.In2=ERR, Cmp.Out=true, Ct.In=ERR, Ct.CloseCondition=true, Ct.Out=LOST};
- $d_3 = 5150h$;
- Γ_3(F1.fail_err) =∞, Γ_3(F1.fail_loss) = ∞, Γ_3(F2.fail_err) = ∞, Γ_3(F2.fail_loss)= ∞, Γ_3(Ct.open_ct) = ∞.

After the firing of F2.fail_err there is no more enabled transitions.

7 - B - Get started with model simulation

7.1 - B - *Interactive or "step by step" simulation of effects of failure modes*

The interactive simulators are applicable to all models that passed the syntax and execution ability checks.

When the operator launches an interactive simulator:

- the simulator computes the initial system configuration i.e. the values of all the model variables (flows and states). Initial values of state variables are directly declared in the model, flow variables are computed through assertions.
- the simulator put also in evidence the enabled transitions i.e. the transitions whose guards are true in the current configuration.
- From these, the user chooses and triggers one of these enabled transitions. The simulator computes the new configuration, i.e., assigns new values to state variables (defined by the effect of the transition) and computes the values of flow variable through assertions. The list of enabled transitions is updated from the new configuration.
- The user triggers another enabled transitions, and so on

Within Cecilia or SimfiaNeo environment, you can choose to trigger automatically or manually the Dirac(0) (see section 6 for the definition of this determinist law).

In the manual mode, you can choose to trigger all possible "Dirac" transitions as soon their condition is true. The "step by step" mode helps to see these transitions.

In the automatic mode, the enabled transitions following a Dirac law are triggered automatically. When there are several enabled transitions, the triggering order can be defined in the model. If it is not defined, the tool default order is used (for instance the default order can be the alphabetical order)

7.2 - B - *Modelling to support the step by step simulation*

In order to ease the model simulation visualisation, graphical representations of nodes can be added within AltaRica framework. It is possible to define how the modelling unit are displayed during simulation, according the values of their state and flow variables. For instance, colours can be associated to the flow values to ease the model review and simulation.

Example of icons and flows visualisation definitions are given §14.

Define links colors and icones

Select a color coding use always the same.

Examples:

- Ok : green, Erroneous: red, Lost : orange
- Drift high: red, Drift low: blue
- false: pink, true: blue green (turquoise)

8 - A - Models characteristics which impact the simulation

This section defines the usual characteristics of AltaRica models which impacts their simulation

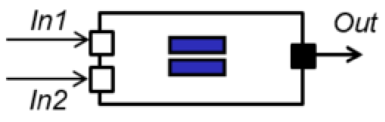
8.1 - I - Orientation of the flows and DataFlow assertions

AltaRica models can define several kind of mathematical relations between the values of the flow and state variables. Model designers often need to specify oriented propagations of values, from so called “input” flows to “output” flows. They also often need to define the values of output flows as a function of the values of a subset of input variables (flow or state variables). The concept of DataFlow model has been introduced to address these needs.

The model is called **DataFlow** if the flow variables have a constant orientation (in or out), the set of assertions assigns unique values to the output variable flows for any configuration of values of the inputs flows and state variables, and there is no circular definition of the flows variable.

Thus, the initial configuration of a DataFlow model is easily compute by the simulation tools and a unique configuration is computed after firing one enabled transition. Such a determinist step of computation accelerates the exploration of the reachability graph of the DataFlow models.

The “Comparator” modelling unit is a small example of data flow model. It specifies oriented flows: two inputs and one output. Moreover, its assertion defines uniquely the value of the flow variable “Out according to the values of the flow variables “In1” and “In2” as shown by the decision table below.



Assert Out=(In1=In2);

Figure 27: Flow variables and assertion of the "Comparator" modelling unit

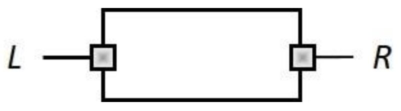
In1	In2	Out
OK	OK	true
LOST	LOST	true
ERR	ERR	true
OK	LOST/ERR	false
LOST	OK/ERR	false
ERR	OK/LOST	false

Figure 28: Relation between the values of the flow variables of the"Comparator” modelling unit

Nevertheless, model designers may also need to specify balances between flows rather than fix oriented propagations. Let us for instance consider the model of a pipe of a hydraulic circuit. The left and right extremities of the pipe can be represented by two Boolean flows L and R. L (resp. R) is true means that fluid is present at the left (resp. right) extremity. Different pumps can be activated so that the fluid goes either from left to right or from right to left**.** So, the orientation of L and R is decided at run time, according to the pump activities, the relation between L and R is called **acausal**.

Acausal relations between flows are not compatible with the dataflow constraints; they cannot be simulated by plain dataflow simulators and the simulator send a warning to the end user.

However, AltaRica 3.0 propose an extended simulation algorithm to deal both with dataflows and acausal relations.

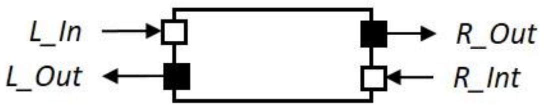


```
class Pipe_Acausal
Boolean L, R(reset = FALSE);
assertion
  L :=: R;
end
```

Figure 29: AltarIca 3.0 acausal model of a pipe

The figure above gives the AltaRica 3.0 model of a pipe. L and R orientations are not defined. The symbol “:=:” makes explicit that L and R shall have the same values without fixing an evaluation order, the simulateur shall solve the problem. When no pump is active, neither L nor R have to be true. So the simulation can either set L=R=true or L=R=false. The directive "(reset = FALSE) " defines the user preference for such a case and the AltaRica 3.0 simulator will choose L=R=false.

Finally, the following modelling approach can be used to **model acausal relations in DataFlow models**. The idea is to represent one acausal equation between L and R by two complementary dataflow equations. Since the orientation of L and R may change in time, each flow is decomposed into two flows with opposite fix orientations as shown in the figure below. Then the two complementary equations assign L_In value to R_Out and R_In value to L_out.



```
Node Pipe_DataFlow
  flow
    L_In, R_In:bool:in;
    L_Out, R_Out: bool: out
  assert
    L_Out=R_In;
    R_Out=L_In;
 edon
```

Figure 30: AltarIca dataflow model of a pipe

The model can be simulated with the data-flow tools but the price to pay is to have more flow variables and a larger set of equations in the model assertion.

8.2 - A - Verification of the correctness of DataFlow assertions

All simulation tools checks whether the model assertion is DataFlow or not. Let us see the principles used to implement this check in the AltaRica 3.0 tools.

In AltaRica 3.0 the concept of **DataFlow** assertion has been formally defined as follows. The assertion A of flow variables from F is Dataflow iff:

- Each flow variable $v \in F$ is assigned only once in the assertion A;
- There is no circular definitions of flow variables in the assertion A.

Let v and w be two variables from V, let A be an assertion built over the variables of V. We say that v **depends immediately** on w in A if $v = F(w, \dots)$ in A.

Let v and w be two variables from V, let A be an assertion built over the variables of V. We say that v **depends** on w in A if there is a variable $\in V$, such that v depends immediately on u in A and u depends on w in A.

The last definition is recursive and allows us to define the **cycles of equations**.

There is a **cycle of equations** in the assertion A if there is a flow variable $\in F$, which depends on itself in the assertion A, in other words there is a circular definition in the assertion A.

In practice, cycles of equations can be detected during a compilation thanks to the dependency graph of the assertion.

A **dependency graph** of the assertion A is a graph $G = < N, E >$, where

- N is a set of nodes, each node is labelled by a variable $v \in V$, such that v appears in the left hand side or in the right hand side of the assignment of the assertion A, v can be a flow or a state variable, let denote such a node $n(v)$;
- E is a set of edges, such that if there is an assignment in A $v_1 := F(v_2, \dots)$, then there is an edge $e =< n(v_1), n(v_2) > n(v_1), n(v_2) \in N, e \in E$.

If the dependency graph of the assertion A has cycles, then there is a cycle in the equations of the assertion A.

An example of dependency graph with cycles is given in Section 10.1.1.

Example of a dependency graph of the assertion:

Let us consider the following assertion from the example given in Section 0:

1. F1.Out := if (F1.St = State_LOST) then LOST else if (F1.St = State_ERR) then ERR else OK;
2. F2.Out := if (F2.St = State_LOST) then LOST else if (F2.St = State_ERR) then ERR else OK;
3. Cmp.In1 :=F1.Out;
4. Cmp.In2 := F2.Out;
5. Cmp.Out := if (Cmp.In1 == Cmp.In2) then true else false;
6. Ct.In := F1.Out;
7. Ct.CloseCondition := Cmp.Out;
8. Ct.Out := if (Ct.CloseCondition = No_Isolation) then Ct.In else LOST;

The corresponding dependency graph is given in Figure 31. Nodes of the graph labelled by state variables are marked in green, nodes of the graph labelled by flow variables are marked in black. This graph is an acyclic directed graph.

If there is a cycle in the dependency graph of the assertion A that means that there is a circular definition in the assertion A and the assertion A is not a Dataflow assertion.

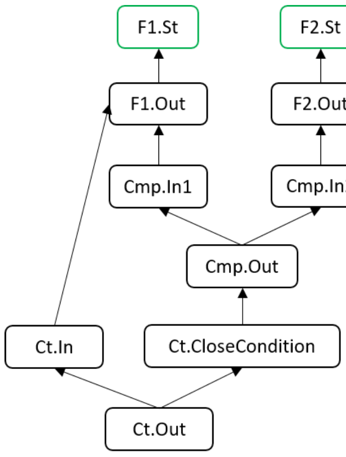


Figure 31: Dependency graph of the assertion

To verify if an assertion A is a DataFlow assertion two properties should be verified:

1. Each flow variable is assigned only once in the assertion A;
2. The dependency graph of the assertion A has no cycles.

If at least one of the properties is not verified then the assertion is not a DataFlow assertion.

8.3 - I - Determinist reachability graph

The reachability graph $R = < \Sigma, \Theta >$ of an AltaRica model is **determinist** if and only if:

- Σ is a set of configurations of the AltaRica model;
- Θ is a set of edges $< \sigma_1, e, \sigma_2 >$, where $\sigma_i \in \Sigma$ are configurations and $e \in E$ is an event of the AltaRica model;
- If two edges $< \sigma_1, e, \sigma_2 >, < \sigma_1, e, \sigma_3 >$ belong to Θ, then $\sigma_2 = \sigma_3$.

In other words, a determinist graph does not contain two edges labelled by the same event and leading from one configuration to two different configurations.

Let us illustrate the impact of this feature on simulation with a component which monitors the occurrence of a hazard and raises an alarm when the hazard occurs. This monitor can fail and generates randomly a true or false alarm. The modeling unit "Monitor" encodes this behavior and the resulting reachability graph is given in the figure below.

```
Node Monitor
  flow Alarm: bool: out;
  state                                S_Hazard:                          bool;
    S_Monitor:{ok,stuck_true, stuck_false};
  init S_Monitor:=ok; S_Hazard:=false;
  event fail_stuck ; hazard
  trans
    S_Monitor=ok |-fail_stuck-> S:=stuck_false;
    S_Monitor=ok |-fail_stuck-> S:=stuck_true;
    S_Hazard=false |-hazard-> S_Hazard:=true;
  assert
    Alarm= case{ S_Monitor=ok : S_Hazard,
                 S_Monitor=stuck_false: false,
                 else true};
 edon
```

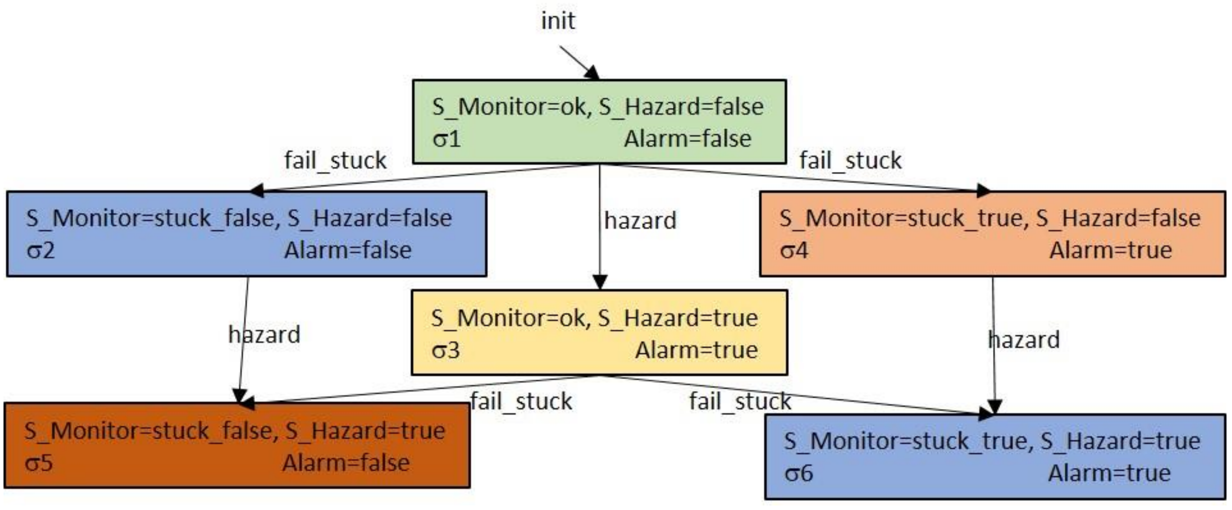

init
S_Monitor=ok, S_Hazard=false
σ1 Alarm=false
fail_stuck
fail_stuck
hazard
S_Monitor=stuck_false, S_Hazard=false
σ2 Alarm=false
S_Monitor=stuck_true, S_Hazard=false
σ4 Alarm=true
hazard
S_Monitor=ok, S_Hazard=true
σ3 Alarm=true
hazard
S_Monitor=stuck_false, S_Hazard=true
σ5 Alarm=false
fail_stuck
fail_stuck
S_Monitor=stuck_true, S_Hazard=true
σ6 Alarm=true

Figure 32: Non-determinist reachability graph of the Monitor component

This graph is clearly non determinist: starting from the initial configuration σ1, the same "fail_stuck" event leads either to σ2 or σ4. In such a case, there is no common agreement on what should be presented to a user who wants to trigger the fail_stuck event: one of the resulting configuration? Both resulting configurations? So, more often, the simulators detect the non determinist transitions, warn the end user and they do not support the simulation.

The problem can be easily solved: different event names are needed for labelling the transitions with compatible guards and divergent effects.

For instance, it is recommended to use two different event names e.g fail_stuck_true and fail_stuck_false rather than fail stuck in the Monitor model. The corrected transitions and the resulting graphs are the followings:

```
  trans
    S_Monitor=ok |-fail_stuck_false-> S:=stuck_false;
    S_Monitor=ok |-fail_stuck_true-> S:=stuck_true;
```

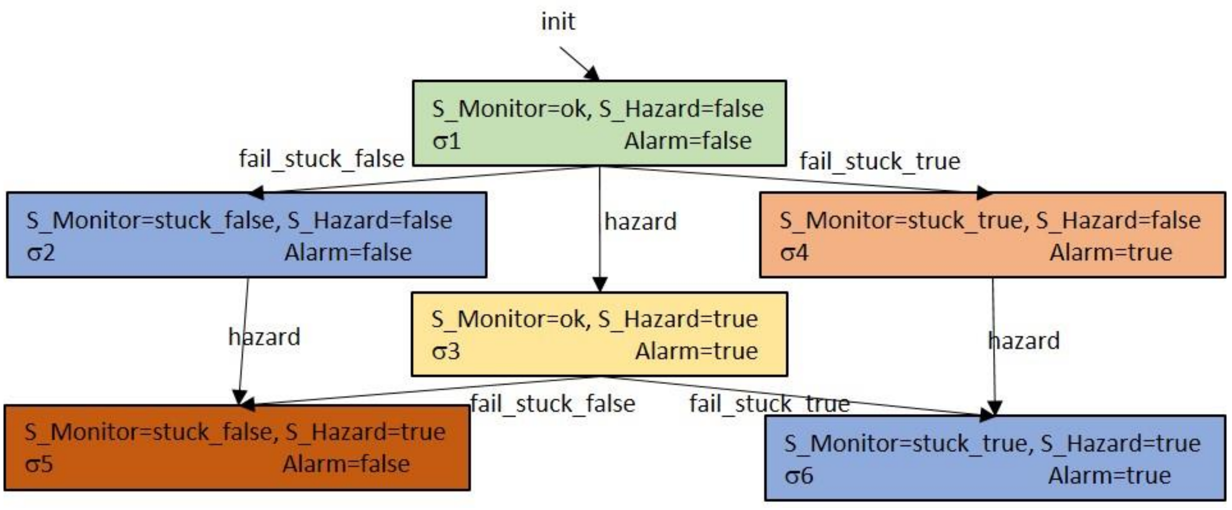

init
S_Monitor=ok, S_Hazard=false
σ1 Alarm=false
fail_stuck_false
fail_stuck_true
S_Monitor=stuck_false, S_Hazard=false
σ2 Alarm=false
hazard
S_Monitor=stuck_true, S_Hazard=false
σ4 Alarm=true
S_Monitor=ok, S_Hazard=true
σ3 Alarm=true
hazard
hazard
fail_stuck_false
fail_stuck_true
S_Monitor=stuck_false, S_Hazard=true
σ5 Alarm=false
S_Monitor=stuck_true, S_Hazard=true
σ6 Alarm=true

Figure 33: Determinist reachability graph of the Monitor component

8.4 - A - Static/dynamic models

As a reminder, a configuration is an assignment of state and flow variables. It describes the current global state of the model.

A model is called **static** if all the sequences of transitions which starts from the same configuration of the accessibility and which are made of permutation of a same set of events leads to the same configuration of the reachability graph of the model.

In other words, the occurrence order of the events has no influence on the resulting configuration of the reachability graph. For example, we consider a model with two possible events E1 and E2. Both sequences (E1, E2) and (E2, E1) result in the same configuration (Configuration D).

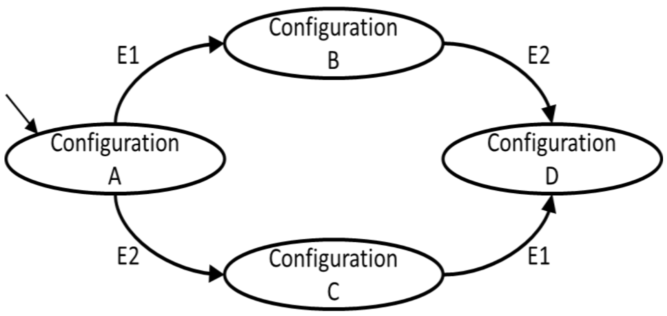


Figure 34: Example of static model

A model is called **dynamic** if it is not static i.e. it exists at least one couple of sequences that are constituted with the same events and result in different configurations. For example, in the Figure 35: Example of dynamic model, the sequences (E1, E2) and (E2, E1) are constituted with the same events and result in different configuration: (E1, E2) results in Configuration E and (E2, E1) results in Configuration F.

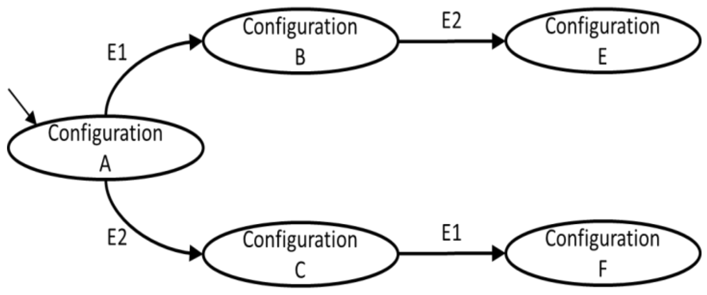


Figure 35: Example of dynamic model

9 – B- Computation of feared events contributors

In aeronautic, a **Failure Condition** is "a condition having an effect on the aircraft and/or its occupants, either direct or consequential, which is caused or contributed to by one or more failures or errors, considering flight phase and relevant adverse operational or environmental conditions, or external events (AMC 25.1309)".

Safety Analysis Methods such as Fault tree analysis or Model Based Safety Analysis aim at providing means to investigate critical scenarios which causes each failure condition of interest for a studied system.

On one side, models or fault trees make explicit the knowledge of the safety expert which is used for this investigation.

- The set of basic events of the model (system failures or errors, external adverse events, …) defines the potential root causes which may contribute to the occurrences of a set of Failure Conditions of interest.
- The equations of the model define the effects of the basic events and how they can be propagated or mitigated according to the system nature and its protections.

On the other side, associated tools extract from the model or fault tree the **critical scenarios causes of a given FC** i.e. the combinations of the model basic events (failures or external event) that lead from an initial system configuration to a configuration where a given FC holds, according to the model hypotheses.

Let us remind the COM-MON example and the two related failure conditions FC_ERR_CAT and FC_LOST_MIN which are true when the output value of the contactor Ct.O is respectively equal to ERR or LOST.

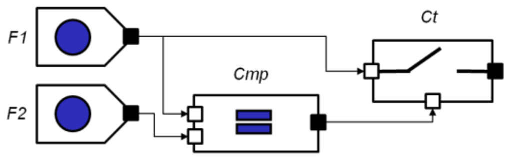


Figure 36: Overview of the COM-MON system with a reversible contactor and a perfect comparator

One version considers that only F1 et F2 may fail and that Ct can be open and then closed again if the condition disappear. The reachability graph of this model was previously defined in chapter 6.1.2. This figure is repeated below for sake of readability. Red arrows of this graph show two paths leading from the initial configuration to configurations where Ct.O =ERR:

$$\sigma_0 \xrightarrow{F2.fail\_err} \sigma_2 \xrightarrow{F1.fail\_err} \sigma_6$$
$$\sigma_0 \xrightarrow{F1.fail\_err} \sigma_3 \xrightarrow{F2.fail\_err} \sigma_6$$

Those red paths define the critical scenarios for FC_ERR_CAT.

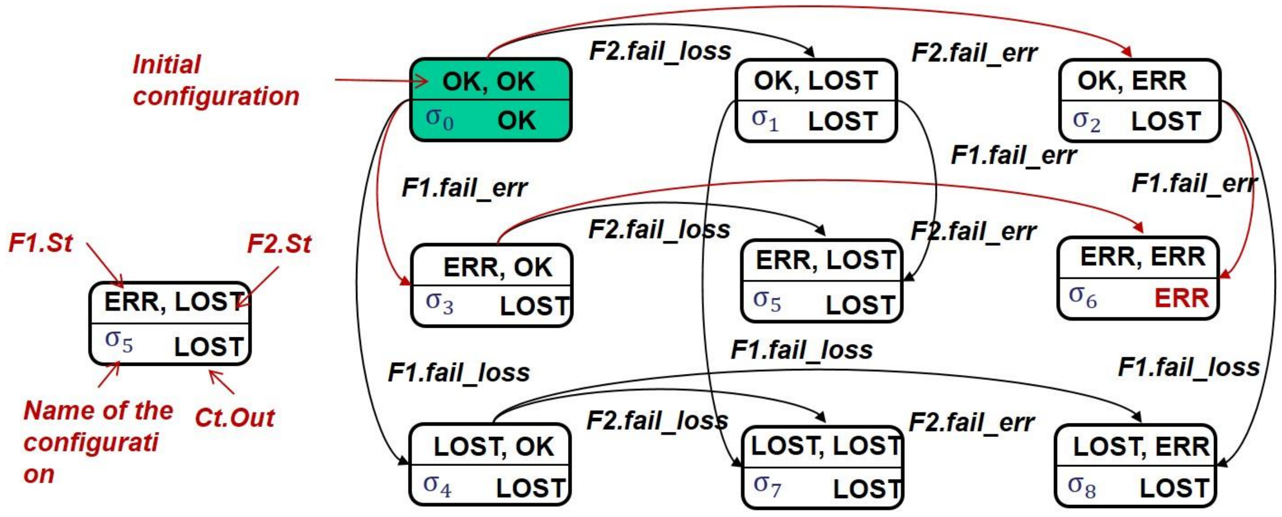


Figure 37: Links between the reachability graph and the causes of failure conditions

Critical scenarios for the FC_LOST_MIN are more numerous:

The traces leading from the initial configuration to the configurations, where Ct.Out=LOST are as follows:

$$\sigma_0 \xrightarrow{F2.fail\_loss} \sigma_1$$

$$\sigma_0 \xrightarrow{F1.fail\_loss} \sigma_4$$

$$\sigma_0 \xrightarrow{F1.fail\_err} \sigma_3$$

$$\sigma_0 \xrightarrow{F2.fail\_err} \sigma_2$$

$$\sigma_0 \xrightarrow{F2.fail\_err} \sigma_3 \xrightarrow{F2.fail\_loss} \sigma_5$$

$$\sigma_0 \xrightarrow{F2.fail\_loss} \sigma_1 \xrightarrow{F1.fail\_err} \sigma_5$$

$$\sigma_0 \xrightarrow{F1.fail\_loss} \sigma_4 \xrightarrow{F2.fail\_loss} \sigma_7$$

$$\sigma_0 \xrightarrow{F2.fail\_loss} \sigma_1 \xrightarrow{F2.fail\_loss} \sigma_7$$

$$\sigma_0 \xrightarrow{F1.fail\_loss} \sigma_4 \xrightarrow{F2.fail\_err} \sigma_8$$

$$\sigma_0 \xrightarrow{F2.fail\_err} \sigma_2 \xrightarrow{F1.fail\_loss} \sigma_8$$

Indeed, safety experts are interested by the **minimal critical scenarios** i.e. the scenarios such that all events of the scenario are necessary and sufficient to reach a configuration where FC holds.

For instance:

1) $\sigma_0 \xrightarrow{F2.fail\_err} \sigma_2 \xrightarrow{F1.fail\_err} \sigma_6$ is a minimal critical scenario for FC_ERR_CAT because all events of the trace are needed to reach a configuration where FC_ERR_CAT holds
2) $\sigma_0 \xrightarrow{F2.fail\_err} \sigma_3 \xrightarrow{F2.fail\_loss} \sigma_5$ is not a minimal critical scenario for FC_LOST_MIN because FC_LOST_MIN holds in σ_3 and thus, $\sigma_3 \xrightarrow{F2.fail\_loss} \sigma_5$ is not necessary

Safety expert also expect that the tools provide for the each FC a **complete set of minimal critical scenario** i.e. no relevant scenario is missing**.**

Finally, the safety experts need a **compact representation of the complete set of minimal critical scenarios**.

When the reachability graph is determinist and the initial configuration is unique, the **sequences of events** are compact representation sufficient to reconstruct the paths.

For instance, the sequence < F2.fail.err ; F1.fail.err> represents the path: $\sigma_0 \xrightarrow{F2.fail\_err} \sigma_2 \xrightarrow{F1.fail\_err} \sigma_6$

Moreover, if the model is static, **failure sets** are sufficient to represent sequences in a more compact way.

For instance, this version of the COM-MON example is static. < F2.fail.err ; F1.fail.err> and < F1.fail.err ; F2.fail.err> lead both to the same configuration σ_6. So, in such a case failure sets noted { F2.fail.err, F1.fail.err} is a more compact representation of the two sequences.

Let us now see how these concepts are implemented with fault tree or AltaRica models.

9.1 - B - Computation of cut sets from fault tree

Computation Inputs

A fault tree is a type of graph used to specify progressively the logical combination of causes of one failure condition. The top node of the tree is the studied failure condition. It is decomposed into some intermediate events combined by one Boolean gate (and, or, …). Intermediate events can be decomposed similarly until reaching the tree leaves i.e. the atomic system failure.

The figure below gives the tree of for the FC_ERR_CAT of the former COM-MON system

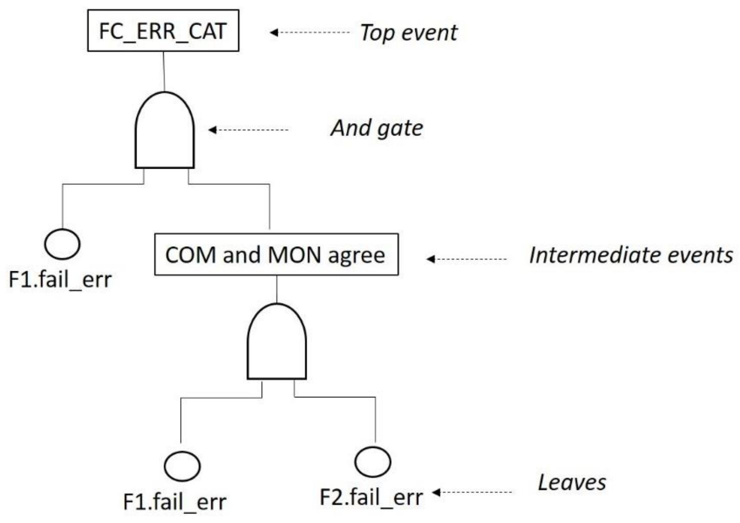


Figure 38: Fault tree of the failure condition FC_ERR_CAT of the COM-MON system

It is worth noting that the fault tree identifies which combinations of events cause the FC but it does specify the event ordering that can be find in reachability graphs.

Computation Outputs

The tools usually extract from the fault tree the minimal sets of atomic failures leading to the top event according to the FC decomposition. These sets are also called "**minimal cut sets**". The size of the cut set is called "**order**"

In our example, { F2.fail.err, F1.fail.err} is the unique cut set leading to FC_ERR_CAT. The cut set order is 2.

Properties of the available algorithms

A fault tree is a Boolean equation which encodes when the top event is true or false according to the value of the leaves.

A cut set represents an applicant of the top event i.e. if all leaves of a cut set are true, then the top event is also true.

Some algorithms can compute all the minimal cut sets of any order. Some others limit the search to cut sets of a given maximum order, especially for vey big fault tree.

9.2 - B - Extraction of boolean equations from AltaRica models

Computation Inputs

It is recommended to apply this computation to a static and determinist AltaRica model which includes observers of the studied failure conditions.

Computation Outputs

The tools synthetize a Boolean equation for each failure condition which encodes when the top event is true or false according to the value of the model basic events.

These equations can be injected in fault tree tools to compute the minimal cut sets of the failure conditions.

In our example, the Boolean equation produced for FC_ERR_CAT could be :

FC_ERR_CAT = ((F2.fail.err and F1.fail.err) or (F1.fail.err and F2.fail.err))

Properties of the available algorithms

The algorithms explore the reachability graph of the model and search the critical paths leading from the initial configurations to configurations where the studied FC is true.

Each critical path is encoded by the product (conjunction) of its event. The set of all the critical paths is encoded by a disjunction of the "path"-product (see for example the Boolean equation generated for FC_ERR_CAT above).

One expects that the computation of minimal cut sets of a FC applied to the generated Boolean equation is:

- sound i.e. there exist for each cut set an ordering of events which leads from the initial configuration to a configuration where the FC holds
- complete i.e. each minimal critical path is covered by a minimal cut sets

These properties are at least guaranteed for correct algorithms applied to static models like the former COM-MON system.

9.3 - B - Computation of sequences

Computation Inputs

This computation is applicable to any AltaRica model which includes observers of the studied failure conditions.

It is recommended (or mandatory depending on the tools) to use determinist model with a unique initial configuration.

The user shall also provide a criteria to bound the sequence search. A common bound is the maximum number of events that are in each sequence.

Computation Ouputs

The tools extract sequences of events leading from the initial configuration to configurations where the failure condition holds. The sequences may be minimal or not depending on the tool used.

In our example, <F2.fail.err ; F1.fail.err > and <F1.fail.err ; F2.fail.err > are the sequences of length 2 leading to FC_ERR_CAT.

Properties of the available algorithms

The algorithms perform an automated simulation guided by the search of configurations where the FC holds.

Starting from one initial configuration, all enabled transition are triggered, new configurations are built and the process is applied to the new configuration. The automated simulation is stopped in one configuration either because the configuration satisfies the FC or because the maximum sequences length is reached.

The successful paths are represented by sequences, some tools minimize the sequences and the generation is complete modulo the bound on the sequence length.

10 - I - Going further with Modelling

10.1 - I - *Specific modelling topics (remove the stone in your shoe)*

This section objective is to provide guidance to overcome chosen MBSA modelling difficulties. For more clarity these points are presented through dedicated examples. We highlight the reasons of the difficulty, the different possible solutions of modelling and the criteria to choose between these solutions.

For the sake of simplicity in the following, the modelling units are also called nodes. In addition we use component as a shot cut for "physical component".

10.1.1 Non DataFlow assertion in a DataFlow model

In the case of non DataFlow assertion the model assertion does not allow to assign a unique value to the flow variables. Either several values may be acceptable or there is no acceptable value.

This issue can be detected by the analysis of the graph of variable dependencies, which derives from the assertions. If this graph has cycles, the model is not dataflow: the **equations of the assertion define circularly the variable values**.

Example : Let us illustrate the case with the following model, which contain two modelling units S and R. S is a recoverable function, which may fail randomly. R is a recovery function, which can recover immediately a failed function. They are connected as shown on the Figure 39: S provides R with the S.ok status; R provides S with the R.Recover function.

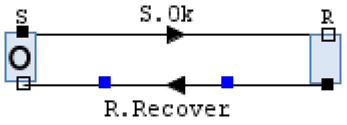


Figure 39: An example of circular definitions in the assertion

The structured model is the following

```
node S2CBlock_RecoveryPolicy_funcRecoverableRandomFault
 flow
      Recovery : bool : in;
      Ok : bool : out;
 state Failed : bool;
 event fail;
 init Failed := false;
 trans
      not Failed|- fail -> Failed:=true;
 assert
      Ok= (not Failed or Recovery);
edon


node S2CBlock_RecoveryPolicy_funcRecoverImmediate
 flow
      Ok : bool : in;
      Recovery : bool : out;
 assert
      Recovery = not Ok;
edon


node main
 sub
```

```
    S : S2CBlock_RecoveryPolicy_funcRecoverableRandomFault;
    R : S2CBlock_RecoveryPolicy_funcRecoverImmediate;
  assert
    R.Ok = S.Ok ;
    S.Recovery = R.Recovery ;
edon
```

The equations of the flat assertion are the following:

```
//equation 1
S.Ok = (not S.Failed or S.Recovery);
//equation 2
R.Recovery = not R.Ok;
//equation 3
R.Ok = S.Ok ;
//equation 4
S.Recovery = R.Recovery ;
```

The transitions are :

not S.Failed |- S.fail -> S.Failed:=true;

The corresponding graph of the variable dependencies is given Figure 40. Arrows define a “depends on ”relation. The graph illustrates that :

- The flow variable S.OK “depends on” the state variable” S.failed” and on the flow variable “S.Recovery” (*equation1*)
- The flow variable R.Recovery “depends on” the flow variable R.Ok (*equation* 2)
- The flow variable R.Ok “depends on” the flow variable S.Ok (*equation 3*)
- The flow variable S.Recovery “depends on” the flow variable R.Recovery (*equation 4*)

In addition there is an equation cycle which creates a circular definition of the variables S.Ok, R.Ok, R.Recovery and S.Recovery .

The graphs allow to shows that **S.Recovery** “depends on” R.Recovery that depends on R.Ok that depends on S.Ok, that depends on **S.Recovery.**

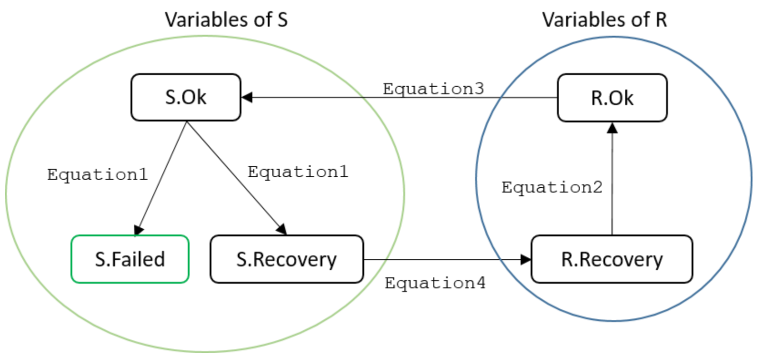


Figure 40: Dependency graph of the assertion with a cycle

In this particular case, there is no problem in the initial configuration: S.Failed is initially false, this enforces that S.Ok is true and then R.Recovery and S.Recovery are false. The initial configuration can be calculated and is unique.

σ_0 = {S.Failed=false, S.Ok=true, R.Ok=true, R.Recovery=false, S.Recovery=false}

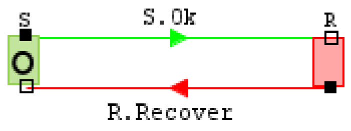


Figure 41: Simulation of the Initial configuration

The flat transition is: `not S.Failed|- S.fail -> S.Failed:=true;`

It is enabled in the initial configuration. After firing this transition, S.Failed becomes true.

Then, there is no possible assignment of values which is compatible with the flat assertion.

In general, to avoid these problems DataFlow assertion should respect the following properties:

1. Each flow variable is defined only once in the assertion;
2. There no circular definitions in the assertion, i.e. there no cycles in the dependency graph of variables of the assertion.

If these two properties are satisfied then the assertion is DataFlow and for each assignment of state variables, there is only one assignment of flow variables.

(a) - I - Cycle in the reachability graph

The reachability graph of the models may contain cycles i.e. there exist at least one sequence of events which starts and ends in the same configuration of the reachability graph.

Example: instantaneous repair policy of a system with a determinist fault

(b) - I - Equation cycle

There is a **cycle of equations** in the assertion if there is a flow variable which depends on itself in the assertion, in other words there is a circular definition in the assertion.

In practice, cycles of equations can be detected during a compilation thanks to the dependency graph of the assertion.

If the dependency graph of the assertion A has cycles, then there is a cycle in the equations of the assertion

More mathematical details are provided in the next chapter.

10.1.2 How to solve an equation cycle - Control Loop

In this section, we consider a simplified control loop example in order to illustrate our proposed guidance and solutions. More information corresponding to the different solutions and corresponding models (Cecilia, SimfiaNeo, and OAR). is provided in section 14.5.

(a) System description and Safety input data

(i) System description

In order to illustrate how to solve an equation cycle we can find when dealing with control loops, let us consider the example illustrated in Figure 42. In that example, we consider a system composed of a logical component we named **AllControlInputs** that build a command from the information provided by the **Sensor** and from the initial **Order (that can be seen as the initial** target**)**. The **Sensor** acquires the data of the Plant output. The **Control** output information is the one used to control the **Plant**.

This example is a simplified control loop, and we can easily replace the **Plant** by a valve or an actuator for instance.

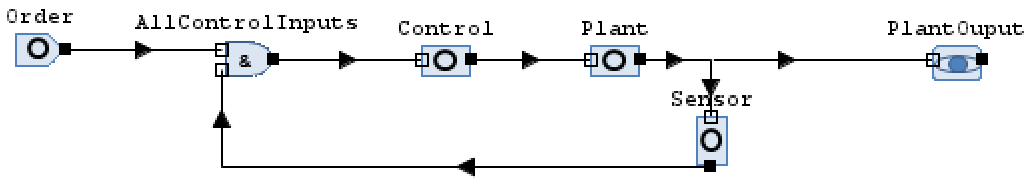


Figure 42: Control loop illustration using Cecilia

Description of the System behavior

- The **Plant** output is monitored by the **Sensor** that sends its acquired information to the **AllControInputs** node. The **Plant** ouput depends on the **Plant** input data
- The **AllControlInputs** node computes a re-evaluated order (**O**) from its two inputs, I1 (the Order) and I2 (the Sensor acquisition information) and sends it to the **Control**
- The **Control** controls the Plant based on its input data (**I**).

(ii) Safety input data

From a safety point of view, the evaluated failure conditions are the following:

- FC1 : Loss of plant output
  FC2 : Erroneous plant output

PlantOutput, on Figure 42, is a safety artefact, observator of the Failure Conditions.

The component failures and corresponding system output and effects are described below:

Control:

- *fail_loss*: leads to the loss of control and the loss of **Plant** output
- *fail_err*: leads to an erroneous command of the **Plant** and an erroneous **Plant** output.

Plant

- *fail_loss*: leads to the loss of **Plant** output
- *fail_err*: leads to an erroneous **Plant** output. The erroneous data is acquired by the **Sensor**.

Sensor

- *fail_loss*: leads to the loss of the **Sensor** acquisition sent to **AllControlInputs** leading to the loss of the **Plant** output

- *fail_err*: leads to an erroneous information from the **Sensor** acquisition, leading to an erroneous **Plant** output

The logical node **AllControlInputs** has no failure.

(b) MBSA modelling framework

In this chapter, we describe the way nodes will be modelled in the following section. Note that for the sake of simplicity and homogeneity we have chosen to use an extract of ONERA existing library. The modelling elements used in all modelling solutions are described in

Table 5. The additional modelling elements or proposed modifications will be described in the corresponding sections. Eventually, note that the modelling is directly related to chapter 5.2 descriptions.

Node/Component name	Interface (input and output flows)	Internal status and failure modes (state variables)	Transitions	Assertions

Order	Input: N/A
	output: O
	
	Type: ok, lost, err
		S ∈ {ok, err, lost}
		Initially ok
			S=ok |- fail_loss -> S:= lost;
			S=ok |- fail_err -> S:= err;
				/*The order depends on the state of the component*/
				O=S

AllControlInputs	Input: I1, I2
	output: O
	
	Type for all flows: ok, lost, err
		N/A	N/A	/*The output depends on the two inputs. If at least one is err, then the output is err,
				If not, if at least one input is lost then the output is lost. Otherwise the output is ok*/
				
				O = case {
				I1=err or I2=err : err,
				I1=lost or I2=lost :lost,
				else ok };

Control	Input: I
	output: O
	
	Type for all flows: ok, lost, err
		S ∈ {ok, err, lost}
		Initially ok
			S=ok |- fail_loss -> S:= lost;
			S=ok |- fail_err -> S:= err;
				/*The output depends on the state of the component*/
				
				O =case
				{ S=ok : I,
				S=lost : lost,
				else err };

Node/Component name	Interface (input and output flows)	Internal status and failure modes (state variables)	Transitions	Assertions

Plant	Input: I
	output: O
	
	Type for all flows: ok, lost, err
		S ∈ {ok, err, lost
		Initially ok
			S=ok |- fail_loss -> S:= lost;
			S=ok |- fail_err -> S:= err;
				/*The output depends on the state of the component*/
				
				O = case {
				S=ok : I,
				S=lost : lost,
				else err };

Sensor	Input: I
	output: O
	
	Type for all flows: ok, lost, err
		S ∈ {ok, err, lost
		Initially ok
			S=ok |- fail_loss -> S:= lost;
			S=ok |- fail_err -> S:= err;
				/*The output depends on the state of the component*/
				
				O = case {
				S=ok : I,
				S=lost : lost,
				else err };

PlantOutput	Input: I output: O Type for all flows:: ok, lost, err	N/A	N/A	O = I;

Table 5: Modelling framework for the Control Loop example

In order to facilitate the understanding of the model during simulation, icons and colours are associated to the nodes and flows. Appendix 14.2 provides a simplified description of the ONERA library colours and icons meaning used in this section.

(c) System characteristic

Even before starting the modelling, we can identify that the system modelled is a control loop: the input of the **Control** brick depends on the **Sensor** output depending themselves of the **Control** output**.**

In that case, a straightforward modelling (meaning without taking into account the characteristic of the system and just plugging the modelling bricks together), will generate an equation cycle (see §10.1.1(b)). Fortunately, this can be solved dealing with the loop or with the equation cycle.

In this section we show that a control loop can lead to an equation cycle in a fault tree. Let us model the system using a classical Fault Tree approach. On purpose, we structure the fault tree strictly following the dependencies of the different inputs and outputs of the system, as defined by the blue arrow in Figure 44 and Figure 43. The Figure 42 shows the resulting message error and the Figure 46 the fault tree. Following this modelling strategy, we generate a circular logic or in other words, an equation. This cycle is due to the dependency between the “erroneous **Sensor** output” gate and the erroneous “**AllControlInputs** outputs” gate (red square). We solve the cycle by removing the gate crossed in red, without affecting the expected resulting cut sets.

(d) Illustration of the control loop equation cycle in a Fault Tree

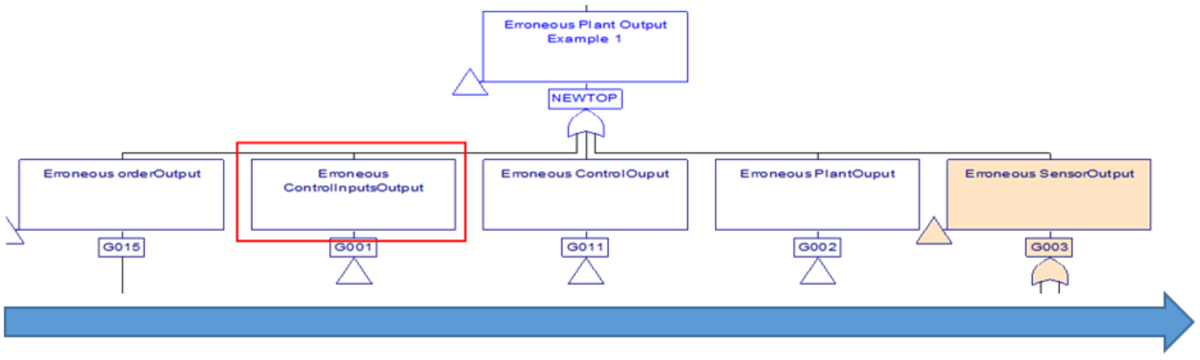


Figure 43: Strategy to analyze the control loop by fault tree- FTA

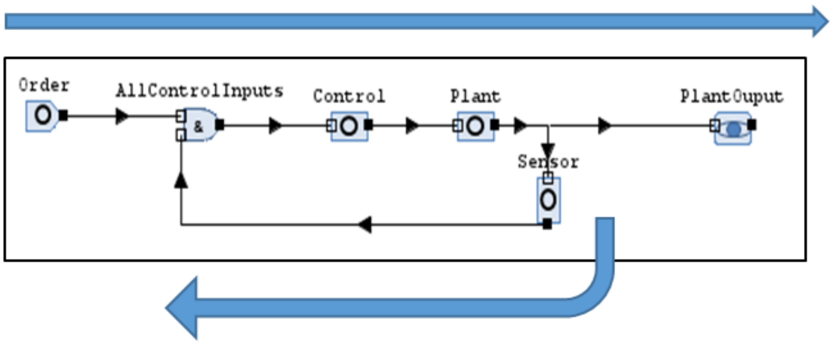
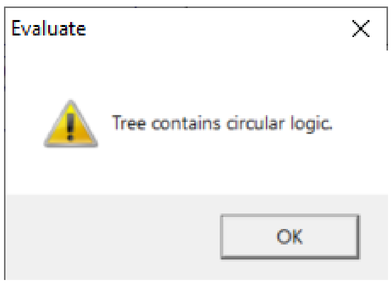


Figure 44: Strategy to analyze the control loop by fault tree - MBSA

In practice, most of the time, the equation cycles are unnoticed in fault trees because they are solved implicitly when the analysts do not choose this modelling approach (without representing the propagation in the top gates choice) or because they remove one of the independent gates without noticing it Figure 46. Nevertheless, when an equation cycle appears in a fault tree, it is always worth analysing the possible impacts of the simplification performed to solve it.


Evaluate

Tree contains circular logic.

Figure 45: Circular logic error message in Fault Tree approach

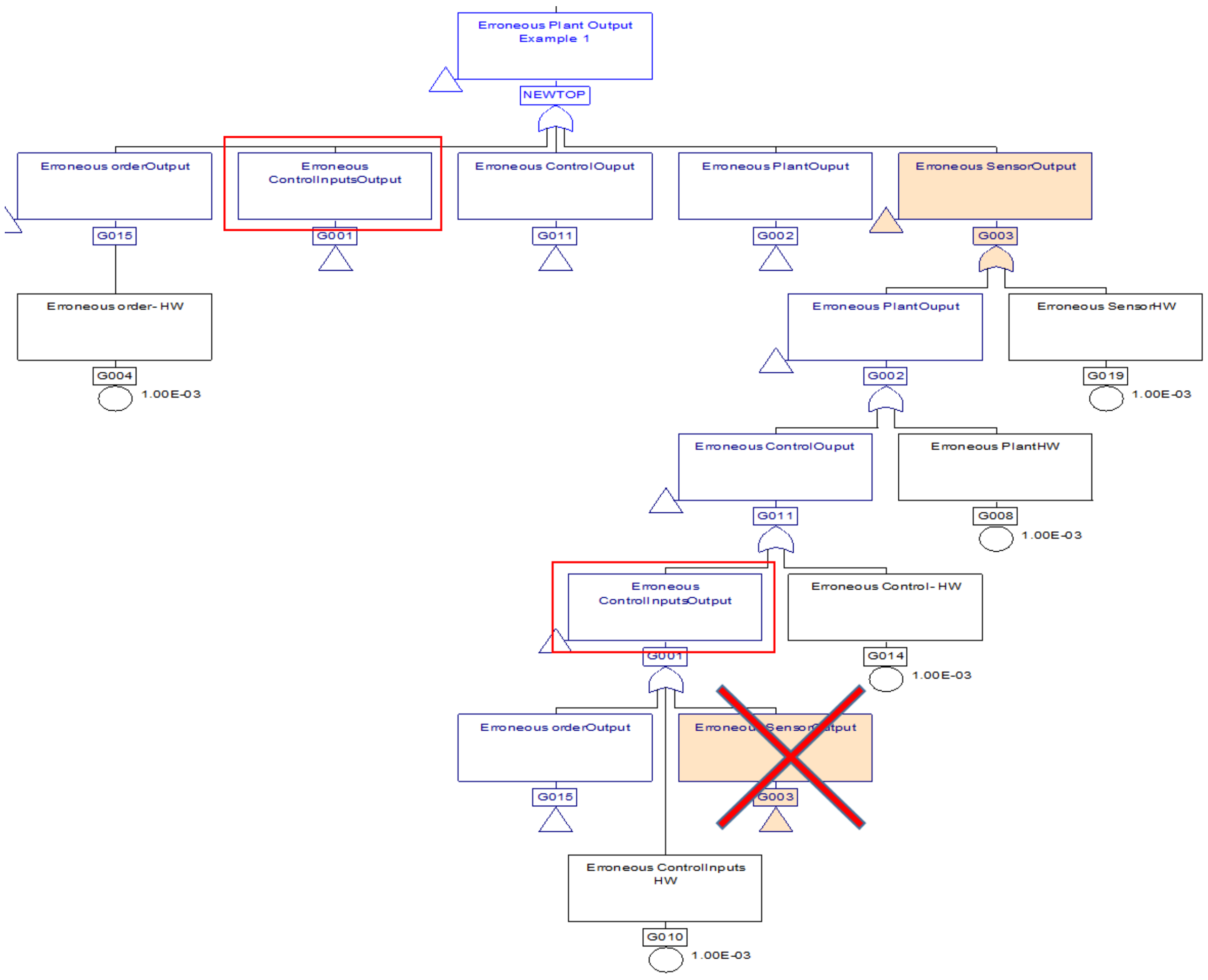


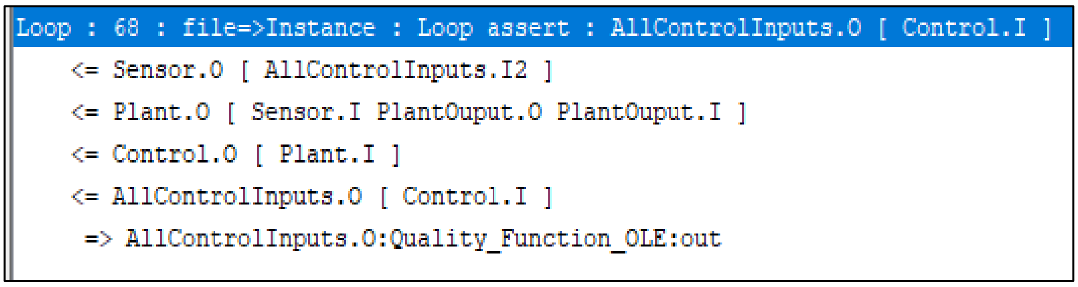
Figure 46: FT – Developed FTA and Equation cycle resolution

(e) Identification of the equation cycle

Modelling tools also provide ways to identify unresolved equation cycles due to control loops. When modelling with Cecilia or SimfiaNeo, informative messages will be displayed providing useful information.

(i) Cecilia v6 illustration of the loop "detection"

The following message is displayed explaining why the AltaRica Data Flow solver is unable to provide a solution.



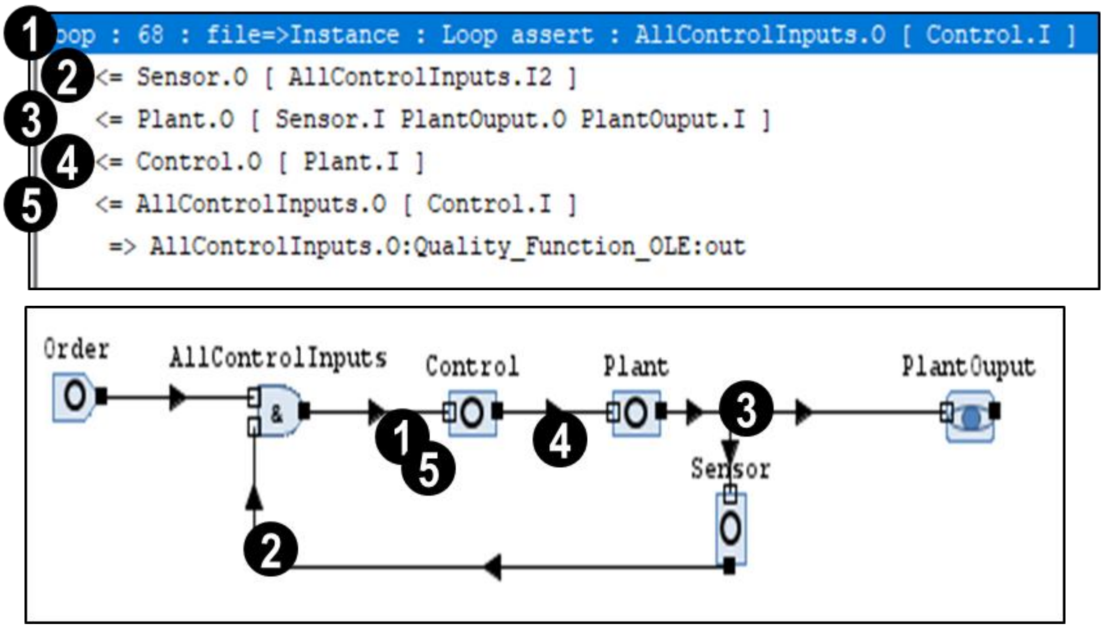
```
Loop : 68 : file=>Instance : Loop assert : AllControlInputs.O [ Control.I ]
    <= Sensor.O [ AllControlInputs.I2 ]
    <= Plant.O [ Sensor.I PlantOuput.O PlantOuput.I ]
    <= Control.O [ Plant.I ]
    <= AllControlInputs.O [ Control.I ]
     => AllControlInputs.O:Quality_Function_OLE:out
```

Figure 47: Cecilia Loop assert message

The information related to the equation cycle generated from the loop is illustrated in Figure 48. The loop is described from its start and goes from "downstream" to "upstream". The message describes that:

1. the output data of **AllControlInputs** named **O** depends on [the **Control** input named **I**]
2. the **Sensor** output named **O** depends on the input data of **AllControlInputs** called **I2**
3. the **Plant** output **O** depends on: the **Sensor** input **I**, the output of **PlantOutputO** and the input I of **PlantOutputI**
4. the **Control** output **O** depends on Plant input I
5. The **AllControlInputs** output **O** depends on the **Control** input named **I**

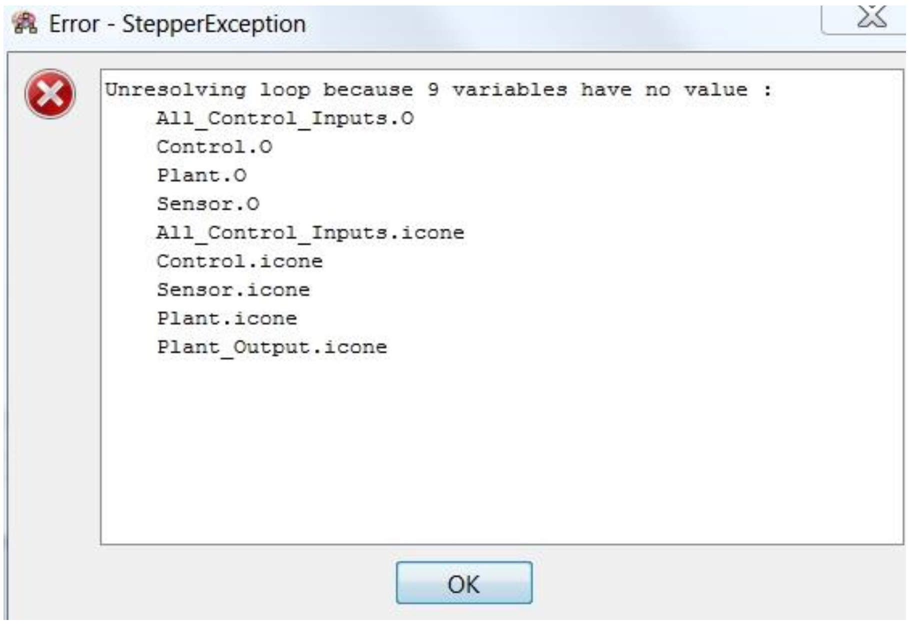
The problem as it is defined is thus insoluble using Cecilia v6 because it is not completely defined.



```
Loop : 68 : file=>Instance : Loop assert : AllControlInputs.O [ Control.I ]
    <= Sensor.O [ AllControlInputs.I2 ]
    <= Plant.O [ Sensor.I PlantOuput.O PlantOuput.I ]
    <= Control.O [ Plant.I ]
    <= AllControlInputs.O [ Control.I ]
     => AllControlInputs.O:Quality_Function_OLE:out
```

Figure 48: Cecilia Loop assert message explained

When launching the computation or the simulation, the message error Figure 49 is displayed:


Error - StepperException

```
Unresolving loop because 9 variables have no value :
    All_Control_Inputs.O
    Control.O
    Plant.O
    Sensor.O
    All_Control_Inputs.icone
    Control.icone
    Sensor.icone
    Plant.icone
    Plant_Output.icone
```

Figure 49: Cecilia Error message for the control loop equation cycle

(ii) SimfiaNeo illustration of the loop "detection"

In SimfiaNeo, the loop is identified by the tool and directly displayed textually and graphically as illustrated in Figure 50.

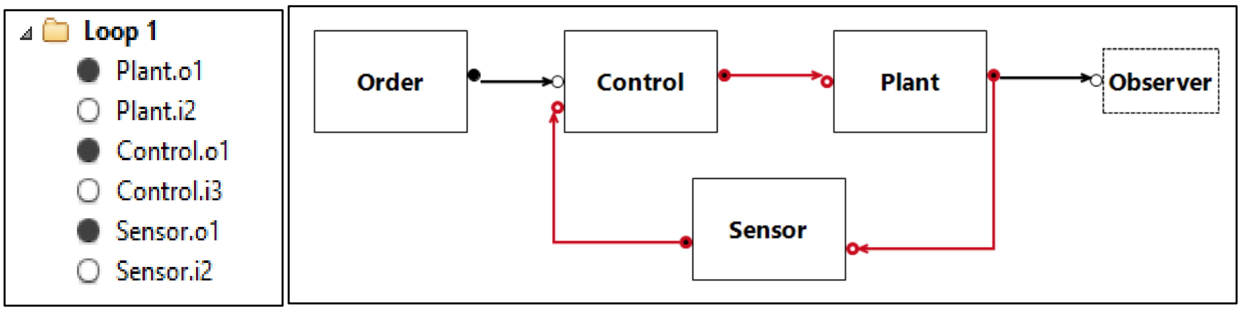


Figure 50: SimfiaNeo Loop identification

(f) The simplification solution or "Cut the Loop" solution

(i) Description of the solution

The simplification or « cut the loop » solution modifies the model in order to solve the equation cycle by "cutting" the control loop in the system, and by ensuring the safety model analysis is still representative of the system studied. Most of the time, it is necessary to add assumptions and explanations in order to achieve this goal.

For instance, instead of analyzing the control loop illustrated Figure 51 we can choose to perform the analysis on a model with a simplified control loop as illustrated Figure 52.

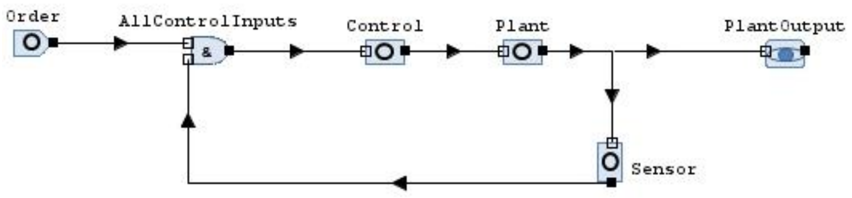


Figure 51: Control loop illustration using Cecilia

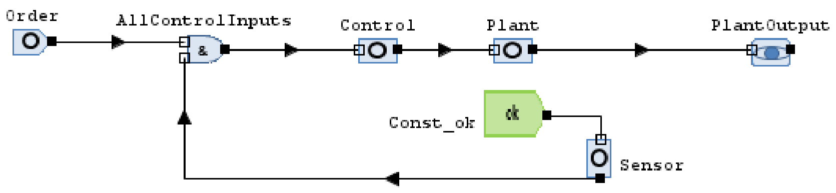


Figure 52: Illustration of the « cut the loop » solution

In our example, the system existing control loop is "cut" using a safety artefact shown in green in Figure 52. The purpose of the "Const_ok" node, or "Ok" node, is to use the same **Sensor** node than the one previously defined. It is a numerical "cap" that sends an "Ok" input to the Sensor. This is equivalent to use a **Sensor** node with no Input.

It is interesting to note that this approach can be related with the way one can "cut the loop" in the fault tree discussed in §10.1.

The key point in this solution is to ensure that the analysis performed thanks to the proposed with the simplified model is as representative as the one performed with the complete model discussed in (ii).

(ii) Validity of the Results

This solution is valid if the simplified model is representative of the system studied despite the simplification. Note that to achieve this goal, in some cases, it may be necessary to provide additional analysis to the model output.

In the example, we consider the safety analyst who decides to cut the loop will check that the cutsets obtained by "cutting the loop" are representative of the control loop described 0 (i). This is the case as shown

Figure 53.

products(MCS('Plant_Output.O.lost')) =
{'Control.fail_loss'}
{'Order.fail_loss'}
{'Plant.fail_loss'}
{'Sensor.fail_loss'}
end
 	
Cutsets for FC1 : Loss of plant output
	products(MCS('Plant_Output.O.err')) =
	{'Control.fail_err'}
	{'Order.fail_err'}
	{'Plant.fail_err'}
	{'Sensor.fail_err'}
	end
	
	Cutsets for FC2 : Erroneous plant output

Figure 53 : "cut the loop" cutsets

In particular, we check that the **Sensor** failures (fail_loss and fail_err) lead to the Failure Conditions as it is expected (loss of the **Sensor** leads to the Loss of **Plant** ouput and erroneous **Sensor** leads to an erroneous **Plant** ouput). In addition, the others components' failures effects are unchanged. Indeed the failures of the **Order**, **Control** or **Plant** (fail_loss and fail_err) directly lead to the **Plant** ouput corresponding failures.

In this example, the Sensor State ("ok" or "failed") has a direct effect. In other words, in case of an erroneous sensor, the Failure Conditions "FC2 : Erroneous **Plant** output" is directly reached.

Counterexample: In case of the addition of a consolidation between the two inputs of **AllControlInputs** (one input ok and the other erroneous leading to the loss of the output)," cutting the loop" the proposed in this example is not valid. It leads to lose the information captured by the **Sensor**. In that different case, an erroneous **Order**, **Plant** or **Control** then leads to the loss of the **Plant** output (FC1) and not to an erroneous **Plant** output (FC2). It is still possible to "Cut the loop" by linking directly the **AllControlInputs** to the **PlantOuput** and by cutting the loop right before the **Control** node, nevertheless the resulting model is very far from the initial system .

(iii) Criteria for choosing this approach

The interest of this approach is to solve the equation cycle by using a static modelling (chapter 5). Compared to dynamic modelling, static modelling enables shorter cutset computation time. In addition, some of the MBSA tools (Cecilia) are able to generate a Boolean equation from static models. In that case, this modelling choice can allow solving big models that otherwise would be too big. In particular, this is the reason why safety specialists at Dassault Aviation use this approach for MBSA models (e.g. flight Control).

Eventually, in addition, for static models the probabilities computation are straightforward.

(iv) Approach limitation

Nevertheless the simplification of the model leads to a model closer to what the safety specialist has in mind (and could write down in a FTA) than to the initial system description. Modelers can choose to make the model look like the system, for instance by adding some graphical artefacts or "empty" links. In that case, they introduce an artificial consistency between the safety and the system models that may lead to misunderstandings and future mistakes.

As illustrated in the discussion section 10.1.2(f)(ii) only the output just before the "loop cut" is affected by all the failure modes. Consequently, this approach is only valid (in terms of resulting cutset) when the control loop has only one output (here **PlantOutput**). When the control loop has several outputs (to the FCs or to the other parts of the model) some information may be missing. In our example, if **Control** output is an input for a monitoring positioned after the **Sensor**, this new node does not see the impact of the failures of **Plant**.

This approach is efficient when the loop can be "cut" before nodes (The **Sensor** here) that only affect the system when they fail (State ok or failed here). Its use is limited when the components involved are implicated in the functional description of the system (for instance in the monitoring).

(g) The "Dirac" solution

(i) Description of the solution

The "Dirac solution" introduces a safety artefact to handle the equation cycle. It allows the modelling of all dependencies between the output and input flows by introducing a state variable.

In order to solve the equation cycle in the example illustrated Figure 52, we introduce this safety artefact through the node called **FeedbackDelay**.

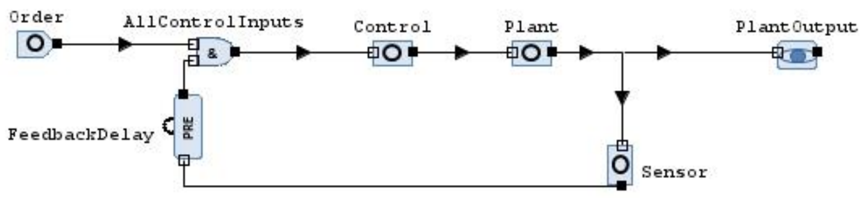


Figure 54: Illustration of the « Dirac » solution

The **FeedbackDelay** node, from the ONERA generic library, introduces:

- A state variable **prev_val** that is initially ok
- An assertion :
  O=prev_val;
- An event called "update" that follows a Dirac(0) law (see **10.1.2(g)**).
- A transition that allows:
  - To remove the direct flow dependency between the output value (O) of the node and its input value (I)
  - To introduce a dependency between the input **I** and state variable **prev_val**
    not (I = prev_val) |- update -> prev_val:=I;

  The defined transition can be read: "when the condition (input value **I** is different from the state value **prev_val)**, the determinist event **update** is instantaneously triggered (because it follows a Dirac(0) law). As a result **prev_val** is assigned to the current value of **I.** Because of the assertion the output O takes immediately the same value, resulting in the propagation of the failure mode.

The introduction of a state variable set to "ok" initializes the problem to be solved when no failure are triggered. This solves the equation cycle for the initial state (refer to 10.1.1).

At this stage it is interesting, to note the state variable introduces a "memory" effect on the transition. Indeed, the state value of **prev_val** will change only when the transition conditions are fulfilled. This is why the modelling artefact we have presented is often called a "Delay". It does not refer to quantitative time (e.g. measured in second) but to sequential time, i.e., the order in which the different updates happen.

Figure 55 and Figure 56 illustrate this "Delay", using the step by step simulation capacities of graphical MBSA tools. In the following images, the green node state is ok, while the red node state are erroneous (see Appendix 14.4.2 for more details). The two figures show consecutively the system states, before and after the manual triggering of the Dirac transition.

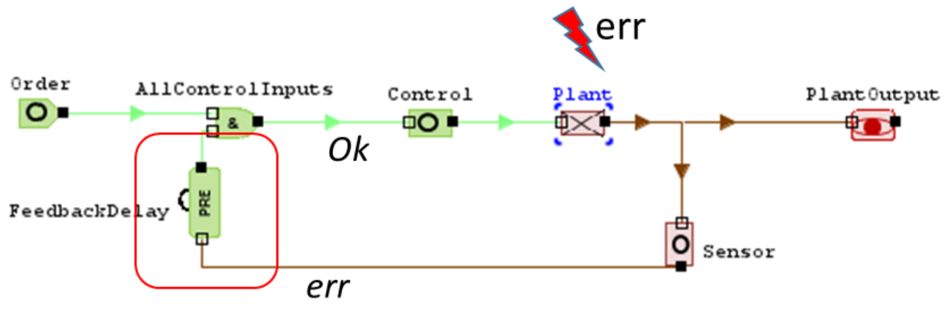


Figure 55 : Illustration of the « Dirac » solution, before Dirac update triggering

Figure 56 represents the state of the system after an erroneous **Plant** failure is triggered. Then, the output of the node **FeedbackDelay** is Ok while its input is erroneous.

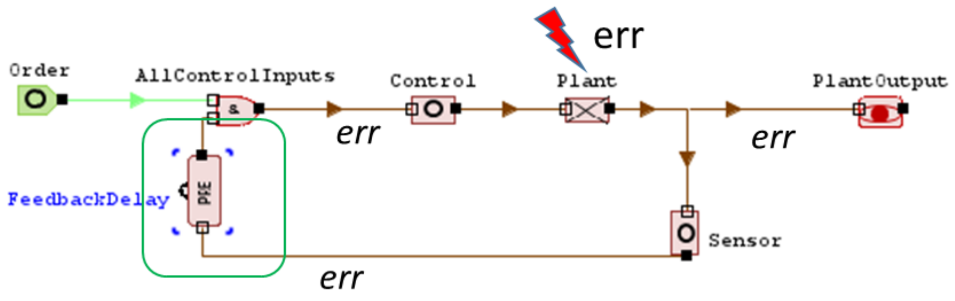


Figure 56 : Illustration of the « Dirac » solution, after Dirac update triggering

The state of the system is the following:

- prev_val =ok (default)
- I=err
- The condition not (I = prev_val) is true

Consequently, the determinist "update" transition defined above is triggered and leads to the update of the modification of prev_val value and to the correct propagation of the erroneous behavior. More precisely:

- The assignement prev_val := I leads to prev_val =err
- The assertion O=prev_val leads to O=err
- The condition not (I = prev_val) is now false, the update transition cannot be triggered in this new state of the model

The problem, with no failure triggered except for Plant, is well posed (same number of equations and unknowns) thanks to the initialization of prev_val and consecutive updates. This will be the same for all the failure combinations to be analyzed.

Note the addition of a dedicated node is a modelling choice or "good practice". It is possible to model the delay defined in this section directly within the modelling unit.

(ii) Validity of the Results

For this example, the cutsets

Figure 57 are as expected and validate the model outputs.

products(MCS('Plant_Output.O.lost')) =
{'Control.fail_loss'}
{'Order.fail_loss'}
{'Plant.fail_loss'}
{'Sensor.fail_loss'}
End
 	
Cutsets for FC1 : Loss of plant output
	products(MCS('Plant_Output.O.err')) =
	{'Control.fail_err'}
	{'Order.fail_err'}
	{'Plant.fail_err'}
	{'Sensor.fail_err'}
	end
	
	Cutsets for FC2 : Erroneous plant output

Figure 57 : "Dirac solution » cutsets

The “Dirac solution” does not require specific validation, except for the local validation of the dedicated node.

In addition, as illustrated in this section, the “step by step simulation” capacities of MBSA tools is very helpful for this validation in order to understand the transition modelled. In addition we can also outline that special care shall be taken when several Dirac are introduced.

All those validation aspects will be developed in future issues of this document.

(iii) Criteria for choosing this approach

The proposed model is very close to the system analyzed. It allows a close representation of the system control. Consequently, it will be easier to validate with system engineers. It will also be easier to use this model to communicate to others or to capture the system behavior.

(iv) Approach limitation

Introducing a determinist Dirac) law may lead to have a dynamic model. When this is the case, the tool solver will generate all the possible sequences of failures leading to the top events while for a static model it would be sufficient to generate all the combination of failures (i.e cutsets) or to solve directly a boolean equation. Consequently, the computations time becomes more important than for a static model. At worst, for very big systems this computation time can be a blocking point.

In addition, when there are several determinist transitions, their synchronization and priority of triggering need to be handled. This adds complexity to the model. Note we will develop these modelling aspects in the future issues of this document.

(h) The "Initialization" solution

(i) Description of the solution

For the proposed example the "initialization" solution allows to solve the equation cycle by initializing the **Sensor** node output to « ok », see Figure 59. This allows to have the same number of equations and unknowns (refer to 10.1.1) and thus to solve the equation cycle for the initial configuration (no failure).

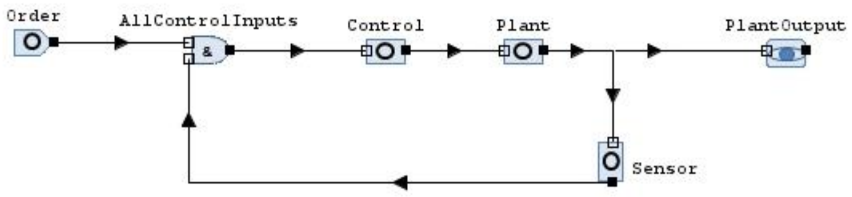
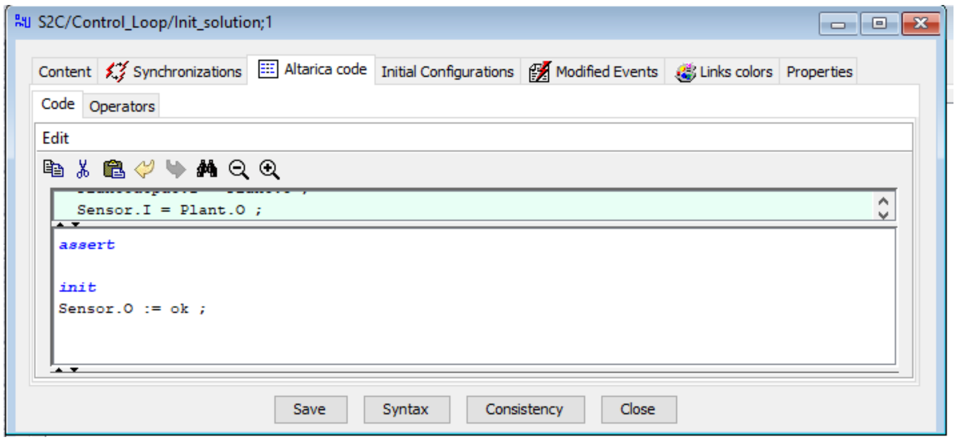


Figure 58: Illustration of the « initialisation» solution (1)



```
Sensor.I = Plant.O ;
```

```
assert

init
Sensor.O := ok ;
```

Figure 59: Illustration of the « initialisation» solution (2)

(ii) Validity of the Results

For this example, the cutsets Figure 60 are as expected and validate the model outputs.

products(MCS('Plant_Output.O.lost')) =	products(MCS('Plant_Output.O.err')) =
{'Control.fail_loss'}	{'Control.fail_err'}
{'Order.fail_loss'}	{'Order.fail_err'}
{'Plant.fail_loss'}	{'Plant.fail_err'}
{'Sensor.fail_loss'}	{'Sensor.fail_err'}
end	end
Cutsets for FC1 : Loss of plant output	*Cutsets for FC2 : Erroneous plant output*

Figure 60: "Initialisation solution" cutsets

(iii) Criteria for choosing this approach

This approach is very simple to use, as it only requires writing a few lines. The proposed model is also very close to the system analyzed. It allows a close representation of the system without safety artefacts and prevents the use of Dirac.

(iv) Approach limitation

Note this approach is only available with Cecilia tool. Since initializing a flow variable may result in a not calculable models (due to conflicts between assertions and initial values), some tools forbid initializing flow variables (SimfiaNeo), and thus do not allow the presented approach.

(i) The "Double flow" solution

(i) Description of the solution

The « double flow » solution relies on the addition of artificial flows to deal with the dependencies in the model. As shown in Figure 61, the dependencies between the variables are modelled through two different paths.

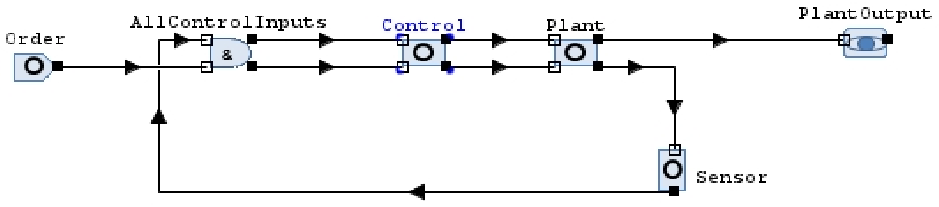


Figure 61: Illustration of the « double flow» solution

Firstly, failure modes of all components are "collected" by the flows from **Order** to **Sensor** (underneath path). In this underneath path, the output of **AllControlInputs** does not depend on the **Sensor** output. Then, **AllControlnputs** gets a second output. Each **AllControlnputs** output is related to an inputTable 6. From this point, the flows (above path) are affected by all loop failure modes.

In order to apply this approach, the nodes **AllControlInputs**, **Control** and **Plant** are modified. Table 6.shows the new definition with changes in bold font.

Node/Component name	Interface (input and output flow)	Internal status and failure modes (state variables)	Transitions	Assertions
AllControlInputs	Input: I1, I2 output: O1, **O2** Type for all flows : ok, lost, err	N/A	N/A	**O1 = I1 ;** **O2 = I2 ;**

Control :	Input: I1, **I2**
	output: O1, **O2**
	
	Type for all flows: ok, lost, err
		S Є {ok, err, lost
		Initially ok
			S=ok |- fail_loss -> S:= lost;
			S=ok |- fail_err -> S:= err;
				O1 = case {
				S=ok : I1,
				S=lost : lost,
				else err };
				
				O2 = case {
				S=ok : I2,
				S=lost : lost,
				else err };

Plant	Input: I1, **I2**
	output: O1, **O2**
	
	Type for all flows: ok, lost, err
		S Є {ok, err, lost
		Initially ok
			S=ok |- fail_loss -> S:= lost;
			S=ok |- fail_err -> S:= err;
				O1 = case {
				S=ok : I1,
				S=lost : lost,
				else err };
				O2 = case {
				S=ok : I2,
				S=lost : lost,
				else err };

Table 6: Modelling framework

When the nodes **Plant** fails, for instance in Figure 62, the path remains green (i.e. “ok”) from **Order** to **Plant**, while the failure is propagated to all nodes through the path above. The two propagations, “ok” and “erroneous” are coexisting in the model without affecting the solution computation.

For all configurations of failures in the model, the corresponding problem is correctly initialized.

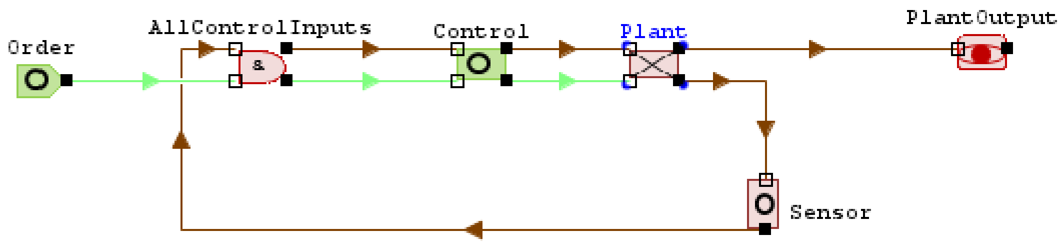


Figure 62: Illustration of the « double flow» solution – Plant failure : “erroneous”

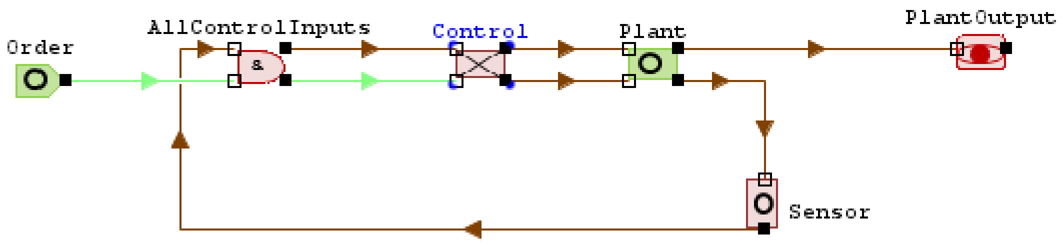


Figure 63: Illustration of the « double flow» solution –Control failure: “erroneous”

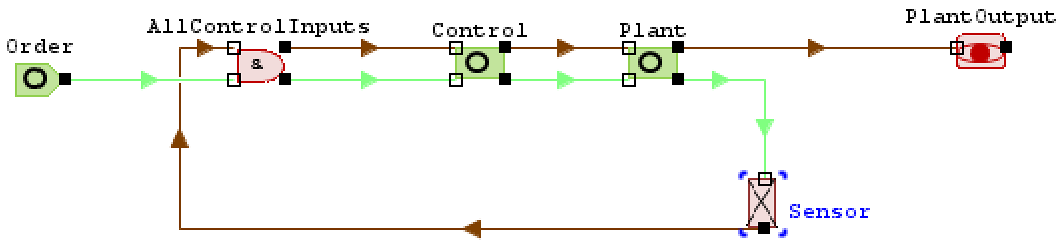


Figure 64: Illustration of the « double flow» solution –Sensor failure “erroneous”

(j) Validity of the Results

The validation of this approach is the same than the one discussed for the “cut the loop” approach. It is needed to demonstrate the modelling is representative of the modelled system. Additional analyses may be required to justify this choice.

The cutsets in the next figure are as expected and validate the model outputs.

products(MCS('Plant_Output.O.lost')) = {'Control.fail_loss'} {'Order.fail_loss'} {'Plant.fail_loss'} {'Sensor.fail_loss'} end	products(MCS('Plant_Output.O.err')) = {'Control.fail_err'} {'Order.fail_err'} {'Plant.fail_err'} {'Sensor.fail_err'} end
Cutsets for FC1 : Loss of plant output	*Cutsets for FC2 : Erroneous plant output*

Figure 65 : "Double flow solution » cutsets

(i) Criteria for choosing this approach

The interest of this approach is to solve the equation cycle using a static modelling (see definition in §2.2). As discussed, this choice can thus reduce the model computation time. In addition the probabilities computation is straightforward.

This approach is usable in case of control loops with several outputs (when several downstream components depend on the control loop output).

(ii) Approach limitation

This approach is the one requiring the more safety artefacts, making the model and the justifications heavier. As a consequence, it is mostly used for local loops, with few components involved.

(iii) Synthesis

Our simple example illustrates that there are several ways to solve an equation cycle. In order to choose between the different approaches one can try to answer the following questions:

1) Do I need to generate cutsets or is an approach that will provide me with sequences acceptable?
2) Do I want my model to represent the system's behavior as closely as possible or is it acceptable to perform simplifications?

The answers to these questions will be driven for instance by the size of the system model or by the use given to the model. Table 7 provides a first guidance to choose the solutions according to chosen criteria.

Criteria	Use of Dirac	Initialization	"Cut the loop"	Double flows
Preference for static models (lower computation time)	No	No	Ok	Ok
Control loop with several outputs	Ok	Ok	No	Ok
Large loop	Ok	Ok with a good knowledge of the loop	Ok with a good knowledge of the loop	Not adapted

Table 7: Choice criteria for the control loop equation cycle resolution

Note: the methods described above are not working whit buses (in Cecilia Workshop) or records (in SimfiaNeo) as the flows have to be isolated to apply the solutions above.

10.1.3 How to solve an event cycle

(a) A first simple example

(i) Description of the model

In order to illustrate what is an event cycle, let us consider a basic example. We concentrate on the model illustrated Figure 66 and described in the next table.

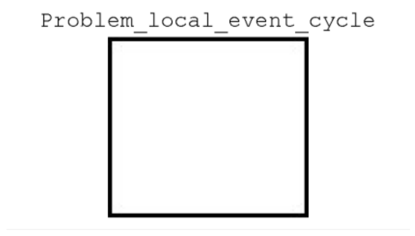


Figure 66: Event cycle illustration (using one modelling unit)

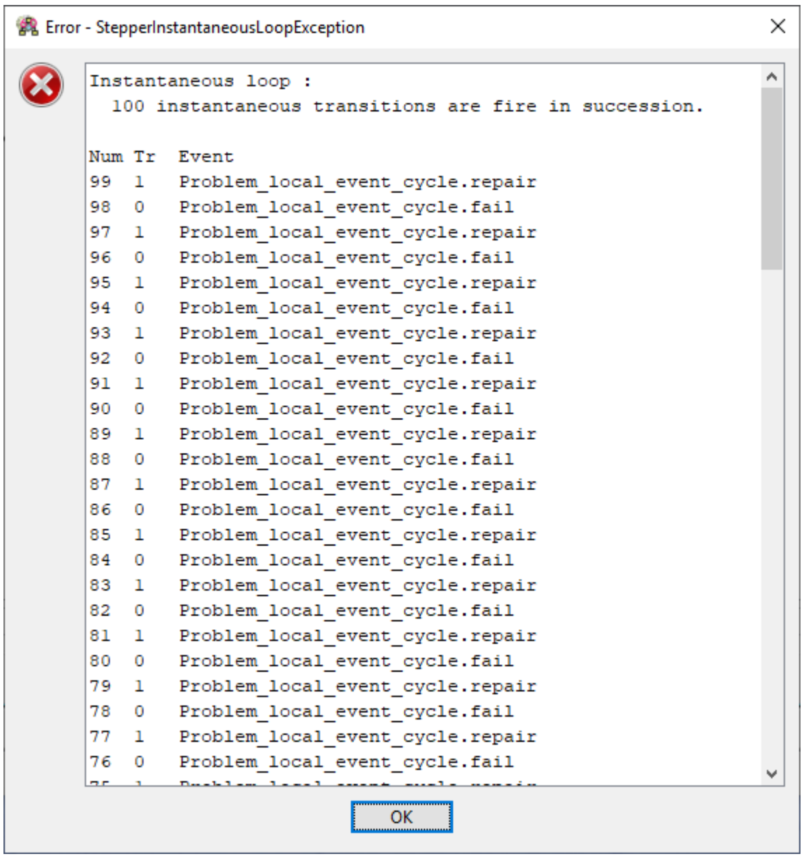
Node/Component name	Interface (input and output flow)	Internal status and failure modes (state variables)	Transitions	Assertions

Problem_local_event_cycle	None	State ∈ {ok, ko}
		Initially ok
			State_ = ok |- fail -> State_ := ko;
			State_ = ko |- repair -> State_ := ok;
			
			Repair and fail follows a Dirac(0) law
				None

Table 8: Modelling framework for the event cycle

(ii) Identification of the problem

Figure 67: Cecilia Error message for the event cycle. It indicates the solver has stopped because the limit of 100 instantaneous transitions successively triggered has been reached without allowing to find a stable configuration.


Error - StepperInstantaneousLoopException

```
Instantaneous loop :
  100 instantaneous transitions are fire in succession.

Num Tr  Event
99  1   Problem_local_event_cycle.repair
98  0   Problem_local_event_cycle.fail
97  1   Problem_local_event_cycle.repair
96  0   Problem_local_event_cycle.fail
95  1   Problem_local_event_cycle.repair
94  0   Problem_local_event_cycle.fail
93  1   Problem_local_event_cycle.repair
92  0   Problem_local_event_cycle.fail
91  1   Problem_local_event_cycle.repair
90  0   Problem_local_event_cycle.fail
89  1   Problem_local_event_cycle.repair
88  0   Problem_local_event_cycle.fail
87  1   Problem_local_event_cycle.repair
86  0   Problem_local_event_cycle.fail
85  1   Problem_local_event_cycle.repair
84  0   Problem_local_event_cycle.fail
83  1   Problem_local_event_cycle.repair
82  0   Problem_local_event_cycle.fail
81  1   Problem_local_event_cycle.repair
80  0   Problem_local_event_cycle.fail
79  1   Problem_local_event_cycle.repair
78  0   Problem_local_event_cycle.fail
77  1   Problem_local_event_cycle.repair
76  0   Problem_local_event_cycle.fail
```

Figure 67: Cecilia Error message for the event cycle

(iii) Illustration of the event cycle using the interactive simulation

We can illustrate the event cycle through an interactive simulation, using the Dirac(0) manual triggering (selected in the tool Preferences).

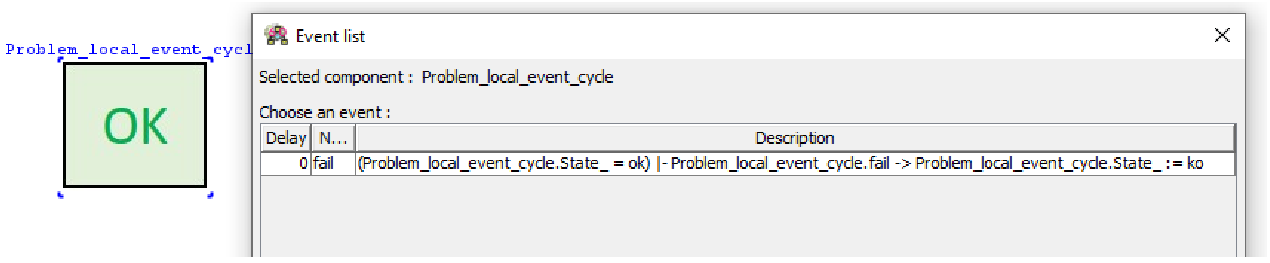
Figure 68 and Figure 69 illustrate the event cycle oscillations between two states, “ok”, and “ko”. These oscillations are due to transitions that depends on event triggered with a Dirac(0) law (as defined in

Table 8) and that become true successively and indefinitely (see §10.1.1(b)). The event cycle is due to a “re-initialization” or a “rearming” of the condition that triggers the events.

In our example, we have the two following transitions:

- State_ = ok |- fail -> State_ := ko;
- State_ = ko |- repair -> State_ := ok;

When the condition allowing the first transition is true, it allows the triggering of the event “fail” (Figure 68). This event follows an instantaneous Dirac law consequently, it is triggered. As soon as the event “fail” has been triggered the transition to trigger “repair” becomes true (Figure 69). When it is triggered, the first condition become true again, and so on (Figure 68).



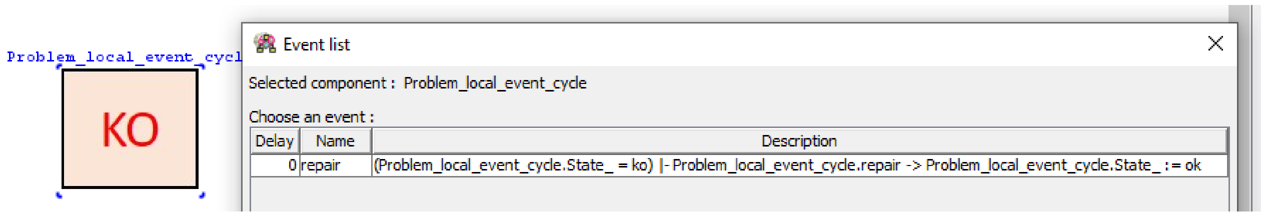
Event list

Selected component : Problem_local_event_cycle

Choose an event :

Delay	N...	Description
0	fail	(Problem_local_event_cycle.State_ = ok) |- Problem_local_event_cycle.fail -> Problem_local_event_cycle.State_ := ko

Figure 68: Event cycle illustration - OK state



Event list

Selected component : Problem_local_event_cycle

Choose an event :

Delay	Name	Description
0	repair	(Problem_local_event_cycle.State_ = ko) |- Problem_local_event_cycle.repair -> Problem_local_event_cycle.State_ := ok

Figure 69: Event cycle illustration – KO state

(b) System description and Safety input data

In this section, we model a switch in order to illustrate how to identify and solve event cycles (see §10.1.1(b)). More information corresponding to the solution is provided as well as corresponding models in their export format (Cecilia, SimfiaNeo, and OAR), see appendix 14.5 for more information.

(i) System description

Let us consider the example illustrated Figure 70. In this example, we consider a system composed of two input sources, a **Primary** and a **Backup.** Each source provides an input (an order for instance) to a **Selector** which function is to select one between the two. By default the Primary source is used. Downstream the **Selector**, a **Monitoring** is able to detect a lost output. In case of a detected failure, the Selector switches from the **Primary** to the **Backup**

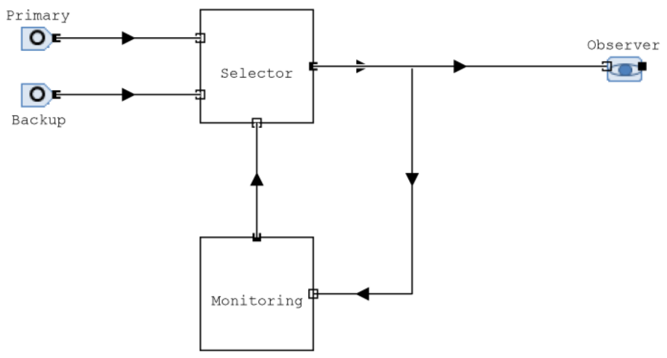


Figure 70: Switch illustration using Cecilia

Description of the System behavior

- The **Selector** output is monitored by the component called **Monitoring**
- **Monitoring** sends validity information of the **Selector** output to the **Selector** component.
- The **Selector** output depends on the validity it receives from **Monitoring.** In the nominal case (no failure detected) the **Primary** output is selected. In case the validity sent to the **Selector** is false the **Selector** selects the input from the **Backup** component**.**

(ii) Safety input data

From a safety point of view, the failure conditions analyzed are the following:

- FC1 : Loss of system output
- FC2 : Erroneous system output

The components failures and corresponding system effects are described below:

Primary:

- *fail_loss*: detected by Monitoring, leads to the use of the backup information. No system effect
- *fail_err*: first leads to an erroneous selected value, detected by **Monitoring**. As a consequence, **Selector** selects the **backup** output. No system effect.

Backup:

- *fail_loss*: leads to the loss of backup. No system effect.
- *fail_err*: leads to an erroneous backup output. No system effect.

Both **Selector** and **Monitoring** are considered fault-free.

(c) MBSA modelling framework

In this chapter, we describe the way components are modelled in presented

Table 9.

Node/Component name	Interface (input and output flow)	Internal status and failure modes (state variables)	Transitions	Assertions

Primary	Input: N/A
	output: O
	
	Type: ok, lost, err
		S Є {ok, err, lost}
		Initially ok
			S=ok |- fail_loss -> S:= lost;
			S=ok |- fail_err -> S:= err;
				O=S

Backup	Input: N/A
	output: O
	
	Type: ok, lost, err
		S Є {ok, err, lost}
		Initially ok
			S=ok |- fail_loss -> S:= lost;
			S=ok |- fail_err -> S:= err;
				O=S

Selector	Input: I1, I2, i_validity
	output: O
	
	Type for I1, I2 and O flows : ok, lost, err
	Type for i_validity: Boolean
		N/A	N/A	/*The output selection depends on the validity input. The value of the output flow depends on this selection and on the value (ok, err, lost) of the selected input
				*/
				
				O=case{
				(i_validity=true): I1,
				else
				I2 // (i_validity=false)
				};

Monitoring	Input: I
	output: o_validity
	
	Type for I flow : ok, lost, err
	Type for o_validity: Boolean
		Validity
		Type: Boolean
		Initially true
			(Validity=true)and(input=Lost)
			|-update->Validity:=false;
			
			(Validity=false)and (input!=Lost)
			|-update->Validity:=true;
				o_status_valid=Validity;

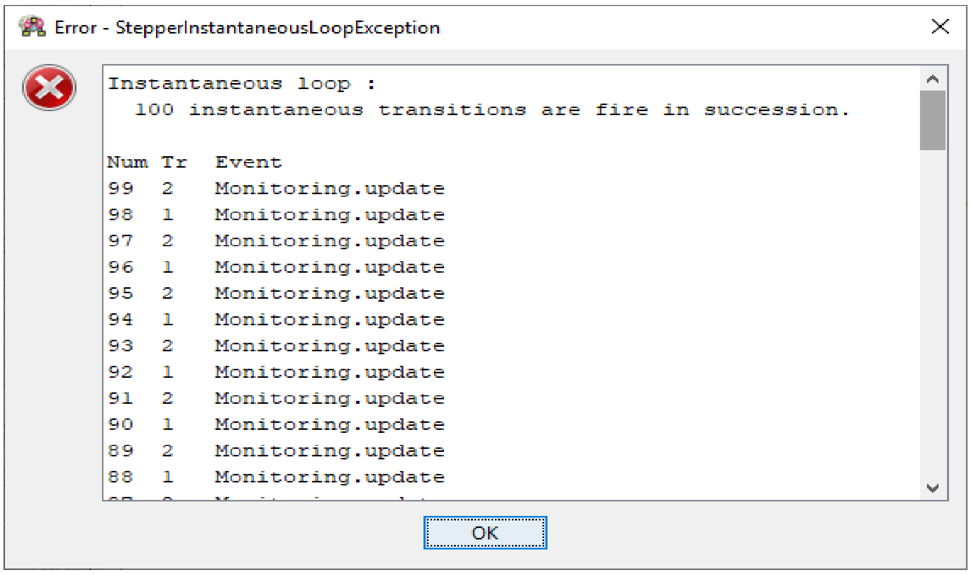
Table 9: Modelling framework for the Switch example

(d) Characteristic of the system

In this example, the equation cycle due to the control loop is solved by introducing a Dirac law in the **Monitoring** node. In addition, in order to take into account the system description 0(b)(i), the **Selection** output depends on the validity. This modelling example illustrate how an event cycle can be introduced, identified and solved.

(e) Identification of the event cycle

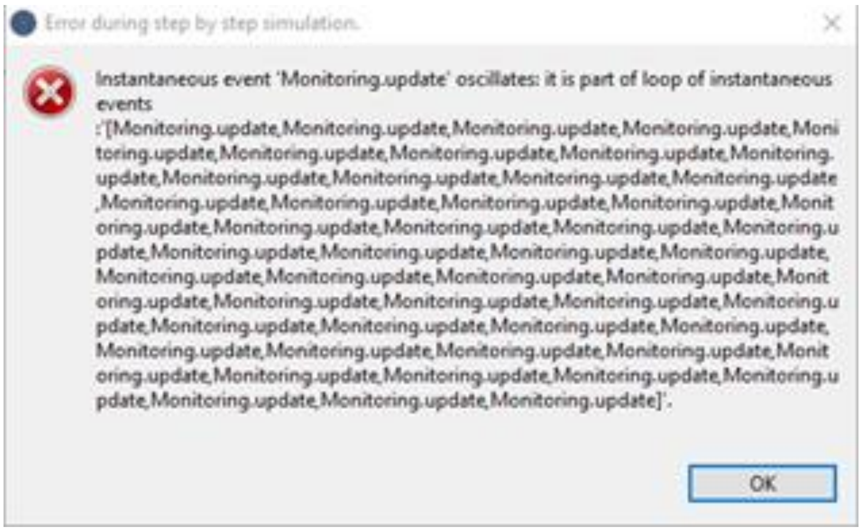
As far as event cycles are concerned, corresponding error messages of the different tools, Cecilia and SimfiaNeo, are not very explicit regarding how to solve the event cycle. Figure 71 and Figure 72 shows respectively Cecilia and SimfiaNeo messages. In Figure 71 the tool informs that the solver has stopped because the limit of 100 instantaneous transitions successively triggered has been reached without allowing to find a stable configuration (§10.1.1(b))



```
Instantaneous loop :
  100 instantaneous transitions are fire in succession.

Num Tr  Event
99  2   Monitoring.update
98  1   Monitoring.update
97  2   Monitoring.update
96  1   Monitoring.update
95  2   Monitoring.update
94  1   Monitoring.update
93  2   Monitoring.update
92  1   Monitoring.update
91  2   Monitoring.update
90  1   Monitoring.update
89  2   Monitoring.update
88  1   Monitoring.update
```

Figure 71: Cecilia Error message for the event cycle



```
Instantaneous event 'Monitoring.update' oscillates: it is part of loop of instantaneous
events
:'[Monitoring.update,Monitoring.update,Monitoring.update,Monitoring.update,Moni
toring.update,Monitoring.update,Monitoring.update,Monitoring.update,Monitoring.
update,Monitoring.update,Monitoring.update,Monitoring.update,Monitoring.update
,Monitoring.update,Monitoring.update,Monitoring.update,Monitoring.update,Monit
oring.update,Monitoring.update,Monitoring.update,Monitoring.update,Monitoring.u
pdate,Monitoring.update,Monitoring.update,Monitoring.update,Monitoring.update,
Monitoring.update,Monitoring.update,Monitoring.update,Monitoring.update,Monit
oring.update,Monitoring.update,Monitoring.update,Monitoring.update,Monitoring.u
pdate,Monitoring.update,Monitoring.update,Monitoring.update,Monitoring.update,
Monitoring.update,Monitoring.update,Monitoring.update,Monitoring.update,Monit
oring.update,Monitoring.update,Monitoring.update,Monitoring.update,Monitoring.u
pdate,Monitoring.update,Monitoring.update,Monitoring.update]'.
```

Figure 72: SimfiaNeo Error message for the event cycle

This message is displayed in simulation mode when the failure “Loss” of **Primary** is triggered. The way the system has been modelled the configuration indefinitely changes from Figure 73 to Figure 74. One can simulate these “oscillations” using the simulation “step by step” (see 6.1 for more details)

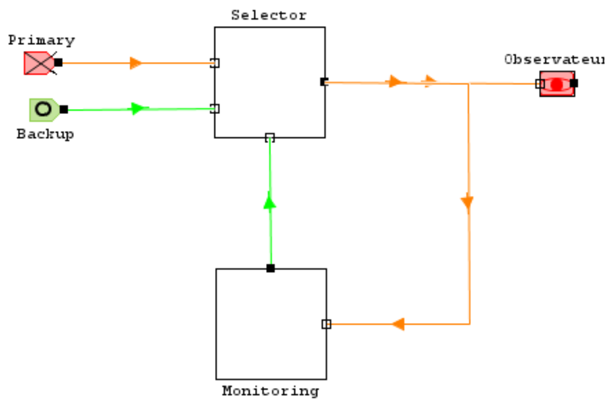


Figure 73 : Switch oscillation configuration 1

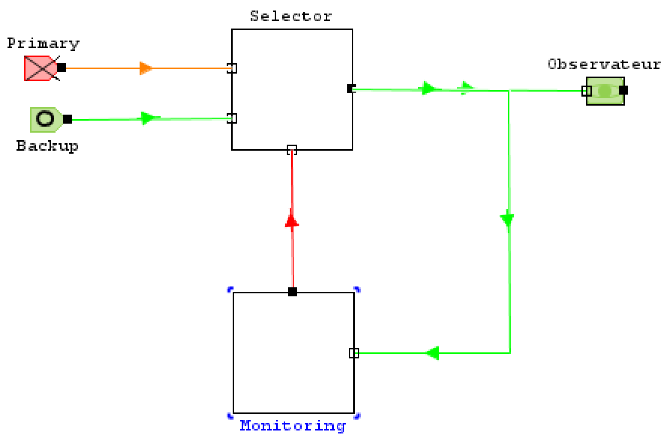


Figure 74 : Switch oscillation configuration 2

The reason of these oscillations is the fact an event cycle has been introduced due to the “re-initialization” or a “rearming” of the condition that triggered in **Monitoring**. Indeed, in order to introduce the dependency between the **Selection** output and the Validity defined in (b)(i), the two following transitions have been introduced:

(Validity=true) and (input=Lost)|-update->Validity:=false;

(Validity=false) and (input!=Lost)|-update->Validity:=true;

Figure 73 shows the model configuration when **Primary** is lost. As **Primary** is initially selected, the output of **Selector** is lost. Thus the input of **Monitoring** is lost too, making the first transition triggerable. Figure 74 illustrates the configuration after this transition. Validity output of **Monitoring** is now false (detection of the loss of the **Selector** output). **Selector** switch to its backup input and its output is then ok. In this situation, the second transition of **Monitoring** is triggerable. This transition gets the model back to the Figure 73 configuration.

As both transitions are associated to Dirac(0) event (determinist and instantaneous), these two transitions are triggered infinitely without possibility to trigger any other event. It is thus impossible to reach stable state.

(f) The solution

(i) Description of the solution

In order to solve the problem, we consider that both **Selector** and **Monitoring** cannot commute back (in practice this information needs to be discussed with the system designer in order to determine if something is actually missing in the system definition).

As a consequence, the following transition of Monitoring is deleted from the model, preventing the event cycle to happen.

(Validity=false)and (input=Lost)|-update->Validity:=true;

Only the following transition remains in order to model the initial commutation:

(Validity=true)and(input=Lost)|-update->Validity:=false;

As the Validity state variable can never go back to its initial value the only Dirac event of the model is fire able. Consequently the event cycle is prevented.

(ii) Validity of the Results

This solution is acceptable in this example, because as soon as **Primary** has failed (either erroneously or by loss), the failure mode is permanent on the whole duration of the study (for example on an aircraft, for the duration of the flight). As illustrated by Figure 73 and Figure 74, when **Primary** is lost, **Monitoring** detects it and requires the **Selector** to switch to its backup input. As **Primary** is not repairable, there is no use (from a model point of view) to enable the Selector to switch back to Primary. From a system point of view, the same reasoning shall be done to decide whether it is relevant to implement the possibility for **Selector** to switch back while **Primary** is not recoverable.

In this case, the model enables to highlight that either the system description is not sufficient (not mentioning that switching back is inhibited) or that the system has a “switch back” functionality that is, in this context, at least useless and at worst dangerous (opening the possibility of erroneous “switch back”).

(iii) Approach limitation

This approach is limited to the case of non-recoverable sources. In case of recoverable sources, and if **Selector** is able to select back Primary, the event cycle issue is much more complicated to solve. This aspect will be developed in a future issue.

(g) Synthesis

The above paragraphs sum-up the way to solve the event cycle issue. These solutions are working independently from the system modeled.

The latch solution which consist by adding a latch component in the flow is not detailed here, as we considered it could be used only if the system modelled includes the capability of latching.

11 Computation of events probability

In Aeronautics, regulations require that the probability of occurrence of a critical failure conditions per flight hour is proportionated to the failure condition severity during the whole aircraft life. However, the probability of occurrence of a failure condition is varying during the aircraft life: it is increasing with the ageing of the aircraft equipment and it decreases after an maintenance action.

So different measures (**average risk** or **maximum risk** per flight hour) have been defined in aeronautical standards such as ARP 4761 in order to address this variability. The agreed computation rules consider the **failure rates** of basic components, the **mission durations** and the **intervals of time between the components maintenance**.

The currently available algorithms exploit these reliability parameters and either the cut sets or the fault tree of the failure condition. Thus, MBSA tools can be used to generate cut sets or Boolean Formula and then other existing tools can be applied to compute the risk. These means are not specialized for MBSA model and their presentation is out of the scope of this guide. The interested reader may find more details in the appendix G of the ARP 4761.

Currently, for MBSA, the main issues are twofold:

- **Enter the parameters of interests**
- **Estimate the accuracy of the computation results**.

The parameters edition and exploitation is currently tool dependent (see for the section 14.1.6(c) in the Appendix).

The accuracy of the computation chain is model dependent. Static models do not account for event ordering, so their compilation into Boolean equations or cuts sets will not change the global accuracy level. The question is open when compiling a dynamic model, with hidden failures for instance, into a Boolean equations.

Past experiences shown that the approximation remain acceptable. Further researches are needed to estimate the conservatism of the computation and possibly design new quantification tools for assessing the reliability of dynamic systems.

12 - B - Verification & validation of MBSA activities

12.1 –B - *Assurance activities*

This section will provide guidance to ensure the validation and verification of MBSA models along time.

Validation & verification activities are distributed along the lifecycle of a model. First activities are performed before starting the creation of the model.

In order for internal or external reviewers, or for certification authorities, to be confident in the presented results, it is necessary to build confidence in the model itself. One option could be asking reviewers to analyze exhaustively the produced model. However, this would necessitates reviewers to always fully master both the modelling language and the tool, and would be highly complicated by the great variety of modelling possibilities and strategies. For these reasons, we find more suitable to take advantage of the MBSA approach specificities:

- Effort of behavior modelling is performed at item level
- Model specification should be performed before the modelling activity (refer to §5.1)

Taking advantage of these two points, verification can be performed in a structured way. The proposal of this guide is to proceed by following steps:

- Step 0: specification of the model
- Step 1: verification of the model specification
- Step 2: verification of the individual modelling units
- Step 3: verification of the global model structure
- Step 4: verification of the global model behavior

Step 0: specification of the model

As described in §5.1, preliminary activities are to be conducted prior to any modelling. These activities are used to specify the future model, including:

- List of modelling units
- List of global model inputs and outputs (e.g. resource systems, failure conditions observation, …)
- List of considered failure conditions

For each modelling unit, the following details should be given:

- What does it stand for? E.g. equipment, safety artefact, logics, …
- List of considered inputs and outputs, with their domain
- Internal states, with their domains.
- List of considered failure modes
- List of other considered transitions (e.g. “opening” and “closing” for a switch)
- For each output, the corresponding transfer function

If using an already existing library of modelling units, these details are replaced by the information of which modelling unit from the library will be used.

[GP 5] A table of truth linking the input, internal state and output variables is a valuable communication and validation tool regarding the modelling unit behavior.

Step 1: verification of the model specification

The documents produced at step 0 should be reviewed by a third-party prior to modelling. They will be used in steps 2 and 3 to check the produced model. This review aims at both validating the modelling strategy, and the correct understanding of input data, including from the design team.

Step 2: verification of the individual modelling units

Modelling units being specified and documented from step 0, they can be individually verified. If some discrepancies exist between a model unit and its specification, they can be due to:

- Error in model, needing to be fixed
- Voluntary decision to modify it, meaning the specification needs to be updated accordingly (and reviewed)

In all cases, verification should be performed by someone other than the modeler, and documented.

Step 3: verification of the global model structure

Verification of modelling units in step 2 is essential, but not enough. As models are built from assembling the modelling units, it is also necessary to verify the global model structure, according to what was specified in step 0. These verification should include:

- List of instantiated modelling units
  The list of instantiated modelling units should be the one specified in step 0. By extension, this check can additionally look at the list of failure modes. If performing quantitative analysis, failure rates should also be checked.

- Connections between modelling units
  Connections between modelling units should correspond to the input architecture. Dangling inputs are forbidden as they prevent any computation to take place. Dangling outputs are tolerated as they do not prevent computation, however they should be justified (e.g. using a generic modelling unit from library).

- Global model inputs and outputs
  Global inputs usually correspond to system inputs (e.g. resource systems). Global outputs can correspond to system outputs, or modelling artefacts to ease readability.

- Translation of Failure Conditions into observers
  Failure Conditions are branched on the model using observers. Their transfer functions should be reviewed.

Step 4: verification of the global model behavior

Last step of verification focuses on behavior of the global model.

- Individual failure modes
  Simulation of each individual failure mode can be performed. Impact of individual failure modes can usually be anticipated from a preliminary analysis. For each failure mode, its simulated impact, locally and globally, can be observed to detect discrepancies with expected behavior. If a preliminary SFMEA exists as an input of the study, it can be used to check the model behavior.
- Step-by-step simulation of scenarios
  It is usually possible to predict how the model should react to a set of scenarios. Definition of a scenario shall include the sequence of transitions and the expected values of a subset of variables from the model. These scenarios are then tested on the model, and (non-)compliance is documented.
- Cuts or Sequences computed from the model on failure conditions can also be used as an additional verification step. Presence or absence of some cuts/sequences is checked, linked to anticipated system behavior. Detailed study of orders 1 and orders 2 is usually considered as manageable.

These verification steps are performed at pre-determined milestones for the study. At each iteration of the system design, it is important to not only update the model, but also its specification, in order to update steps 1 to 4. As MBSA focuses on local modelling, it is not necessary to review the whole model at each iteration, but instead to perform logical verifications and update global simulation scenarios.

Verification needs to be documented in order to justify confidence in the model, hence in the results. These proofs are to linked to the different system and model versions.

12.2 - B - Few words about the verification of the model syntax and execution ability

Graphical editors assist the model coding and prevent some errors in the coding and in the assembling of nodes. The verification of the model syntax aims at checking whether the MBSA tools can understand the model code.

In our case, the model and all the nodes shall be compliant with the *syntax rules* of the AltaRica language. Syntax errors prevent the model simulation or analysis. The full syntax check is automated. Syntax errors are reported and they are localized (more or less accurately) in the model code. It is recommended to check the syntax of modelling unit before reusing them in hierarchical modelling unit to ease the global model debug.

The verification of the model execution ability aims at detecting whether the simulation of the model may fail.

This is dependent of the *language operational semantics* i.e. the rules that are defined to run a model syntactically correct, in our case AltaRica dataflow.

Errors can also be prevented by following recommended practices. For instance it is recommended to write a unique equation to define the value of an output flow case by case. We use “if then else” or “case” constructions to ensure a methodical consideration of the assertion. The “else” case ensures that all cases are considered while giving a fallback value.

Problems may also be raised by creating circular assertions after connecting modelling units in a system architecture with loops. This is detected by consistency checks of the global model. This kind of issues is developed in particular in section 10.1.

13 - I - Using MBSA to support industrial development in the aeronautics industry - Recommended Practices

This section will provide guidance in order to use MBSA models to support the safety activities during systems development.

13.1 - I - ARP4761A / ED-135A activities

SAE ARP4761A and its EUROCAE complement, ED-135A ([**REF B**]), present guidelines for performing safety assessments of civil aircraft, systems, and equipment. It may be used when addressing compliance with certification requirements (e.g., CS 25 for large airplanes, CS29 for large helicopters or CS-E for engines).

This document deals with both processes (Aircraft Safety Assessment (ASA), System Functional Hazard Assessment (SFHA), Preliminary System Safety Assessment (PSSA) …) and methods that may be used to conduct the processes (Fault Tree Analysis (FTA), Dependence Diagram (DD), Markov Analysis (MA), Failure Modes and Effects Analysis/Summary (FMEA/FMES) …)

The A version of SAE ARP4761 (and more particularly the appendix N) introduces the MBSA as a new method, which achieves results that are equivalent to those obtained from the classical e.g., Fault Tree Analysis (FTA) safety analysis methods.

The appendix Q describes, in detail, a contiguous example of the safety assessment process for a function on a fictitious aircraft design, the “Decelerate wheels” function, performed by the “Wheel Braking System”. More particularly, the Q9 part of the appendix gives an example of how a MBSA method may be carried out to support the safety analysis during a Preliminary System Safety Assessment (PSSA).
The latter will be quoted or referenced to illustrate the point.

13.2 - I - Generality about safety assessment with MBSA

The safety analyst should get explicit inputs from the design team or capture on his own rationales to fully understand the design characteristics. As long as the upper level requirements are not fulfilled, the design architecture is improved and refined. The MBSA model evolves accordingly.

The design architecture continues to be refined notably for definition reasons. The model will not be refined up to the same level. The stop criteria may be to have sufficient decomposition to get independent blocks regarding the effects of random failures. An other stop criteria for refining the model may be the ability to verify each safety requirement.

At each iteration, this process was followed:

1. Take into account **MBSA inputs**: Failure Conditions, refined requirements, design architecture and updates;
2. **Model** this architecture: modeling assumptions to be confirmed, acceptable simplifications;
3. Perform the **MBSA Failure Conditions Evaluation** (generation of functional failure sets and/or minimal cutsets, probability computation);
4. Assess the **PASA** (Preliminary Aircraft Safety Assessment) **and refined safety requirements compliance**;
5. Provide **MBSA outputs**: new requirements, design recommandations or even architecture suggestions to cope with non-compliant requirements.

Note that, even if it is possible to do it with a MBSA model, it is efficient to do DAL allocation and probability budgets (i.e. qualitative and quantitative objectives) with a high-level Fault Tree.

13.3 - I - FDAL/IDAL assignment

The FDAL/IDAL assignment (Functional/Item Development Assurance Level) is based on the Functional Failure Set (FFS).

The definition of a Functional Failure Set (ARP4761A ([REF B])main body § 2.2) is:
"Functional Failure Set: a set of one, or more members that are considered to be independent from one another (not necessarily limited to one system), whose development error(s) leads to a top-level failure condition."

A parallel is drawn between FFS and MCS in ARP4761A([REF B]) Appendix P § 3.2.3:
"A FFS is the equivalent to a fault tree minimal cut set, whose members represent the result of potential development errors rather than failures".

In this case, the MBSA model **"failure events"** described in Q9 § 2.2 represent development errors (for hardware as well as for software).

The FFS for each Failure Condition (FC) (e.g., Q9 § 4.3.1 and § 4.3.2) will imply DAL assignment constraints (Q9 § 4.4.1) and independence requirements. At this point, some common modes (such as common resource systems, common hardware development or common software development) have already been taken into account.

The common resource systems follow from the overall integration (Wheel Braking System, Hydraulic System, Electrical System). The hardware/software common modes are handled through Common Cause Failures (CCF) definition.

A post-processing analysis based on "Attributes", which can be assigned to each failure event, allows to identify other common modes as much as possible.

Note that in this proposition, the FDAL/IDAL assignment is optimized by engineering judgement. The model is only used to verify that DAL are correctly assigned regarding the ARP rules and the chosen safety principles.

13.4 - I - Fail-safe assessment and probability computation

The fail-safe concept (no single "random" failure should lead to a catastrophic Failure Condition) and the probability computation are based on the Minimal Cut Set (MCS).

A Minimal Cut Set is a set of one, or more members whose random failure(s) leads to a top-level failure condition.

In this case, the MBSA model **"failure events"** described in Q9 § 2.2 represent random failures (for hardware only), have their own probability laws and their own failure rates.

The MCS of the catastrophic FC should not have first order member (Table Q9-10), and each FC should meet quantitative safety objectives (Tables Q9-8 and Q9-10), depending on their criticality.

13.5 - I - CCA assessment and independence principles

This part deals with Common Causes Analyses (CCA), which cover Common Modes Analysis (CMA), Zonal Safety Analysis (ZSA) and Particular Risk Analysis (PRA).

The definition of a Common Cause (ARP4761A ([REF B])main body § 2.2) is:

"COMMON CAUSE: a single failure, error, or event that can produce undesirable effects on two or more systems, equipment, items, or functions."

Common causes issues can be addressed in several ways:

- By integrating them directly into the model
- By evaluating them through post-processing of the cuts (FFS or MCS) enriched with attributes

Hera are some examples explained:

a) Electrical power supply

<u>First method</u>: create electrical power supply attributes, with values that represent the different bus bars (AC and DC). If in a MCS of order 2, both equipment are powered with the same bus bar, the cut recalculated in relation to the attributes will be order 1.

<u>Second method</u>: create the electrical power system with its bus bars (which can also have their own common modes, for example the engines…) and integrate it with the system of interest. Then the equipment of the system of interest will include a dedicated electrical power supply input, linked to the suitable bus bar. This input will obviously impact the behavior of the equipment (through the state or the assertion).

The first method is easier to implement but allows only to assess cuts at order 1. The second method needs more work but allows to assess cuts at any order, to combine power supply system failures and the failures of the system of interest, but also to assess more precisely the impact of various failure modes (loss, erroneous…).

These methods are applicable for any power supply (hydraulic, fuel, pneumatic…)

b) Development errors

<u>First method</u>: create development errors attributes, for example "Autopilot software development".

<u>Second method</u>: use the synchronization artefacts proposed in some tools (called for instance Common Cause Failure in Cecilia). These synchronizations will create events which will appear in the cuts (FFS).

<u>Third method</u>: create nodes which will represent the development errors and which will be linked to the equipment through dedicated inputs.

The first method is easier to implement but less accurate, the second one can be time-consuming in case of many common modes to assess, the third one increases the size of the model but allows to have a graphical representation of common modes.

c) Zonal threats or particular risks

<u>First method</u>: create zonal attributes, with values that represent the different zones where equipment are installed.

<u>Second method</u>: create a zonal view, whose equipment will represent the different zones. The failure modes will represent the zonal threats (internal or external) that will impact the equipment installed in these zones. The equipment of the systems (systems of interest and power supply systems) will be linked to the zones through dedicated inputs.

The first method is easier to implement but less accurate, the second one will allow to have a graphical representation and to be more precise (which threat impacts which zone, what is the impact of each threat on each equipment behavior…).

Note: this topic is not addressed in the ARP4761A([**REF B**]) document.

14 Appendix

14.1 – B- Get Started with the tools – Cecilia

This chapter deals with the basic notions to start modelling with SATODEV Cecilia tool.

14.1.1 Introduction

Before to start it is recommended to come back to the questions to asked before starting, by referring to §5 and GetStarted kit.

The COM/MON pattern example is described in §5.2 and GetStarted.

14.1.2 Open an existing project

In Cecilia, open a model corresponds to import function. To import a file do: File> Create > Import> Import (Cecilia .xml) then select the » .xml » file to import.

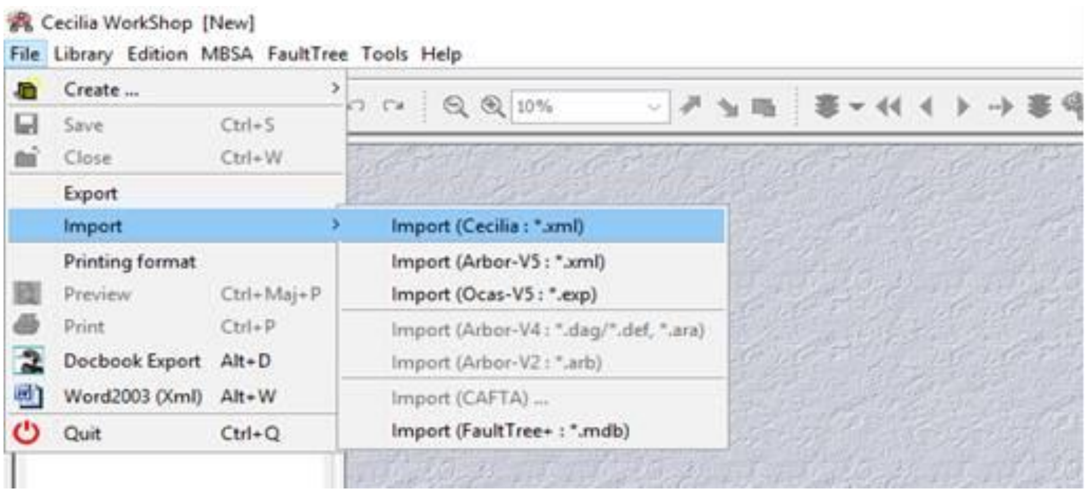


Figure 75: Cecilia GUI – open a existing project

14.1.3 How to create a new project

Open Cecilia tool by clicking on cecilia.bat.

Create a database as described below in the figure:

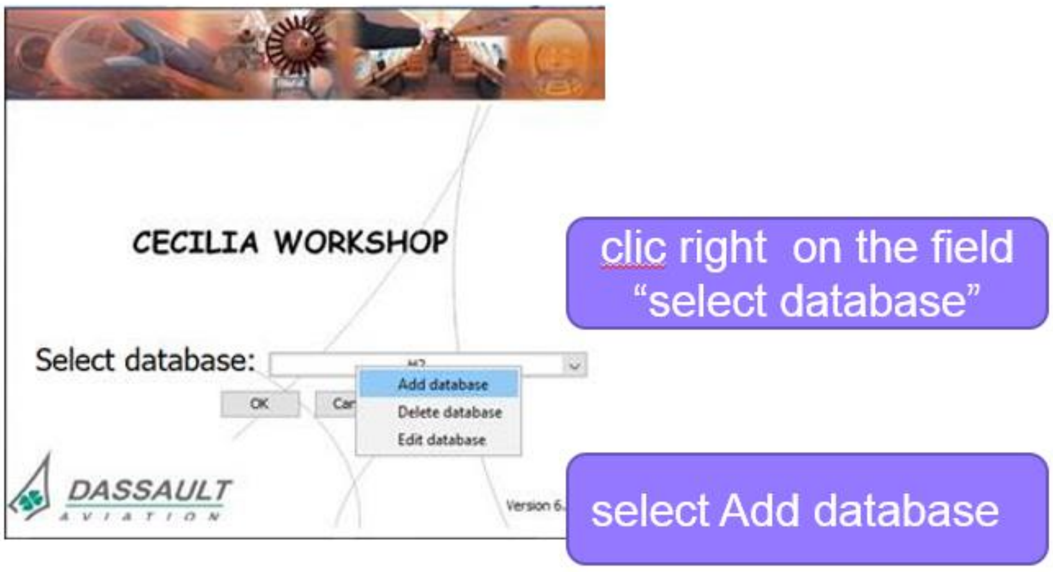


Figure 76: Cecilia GUI – database creation #1

To create a new data base: select the location and name by filling the blank field in the image below or clicking on "Create new database".

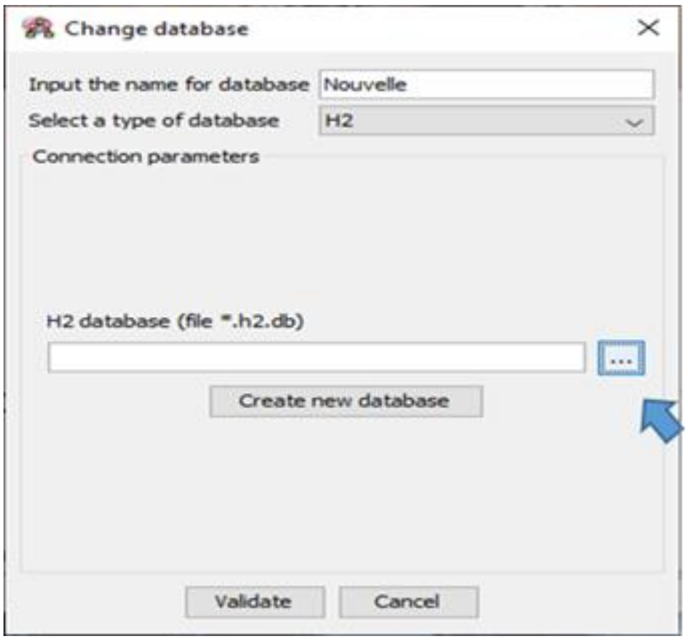


Figure 77: Cecilia GUI – database creation #2

Write the name of your database in the field "Input the name for database".

Then create the project through the add project button:

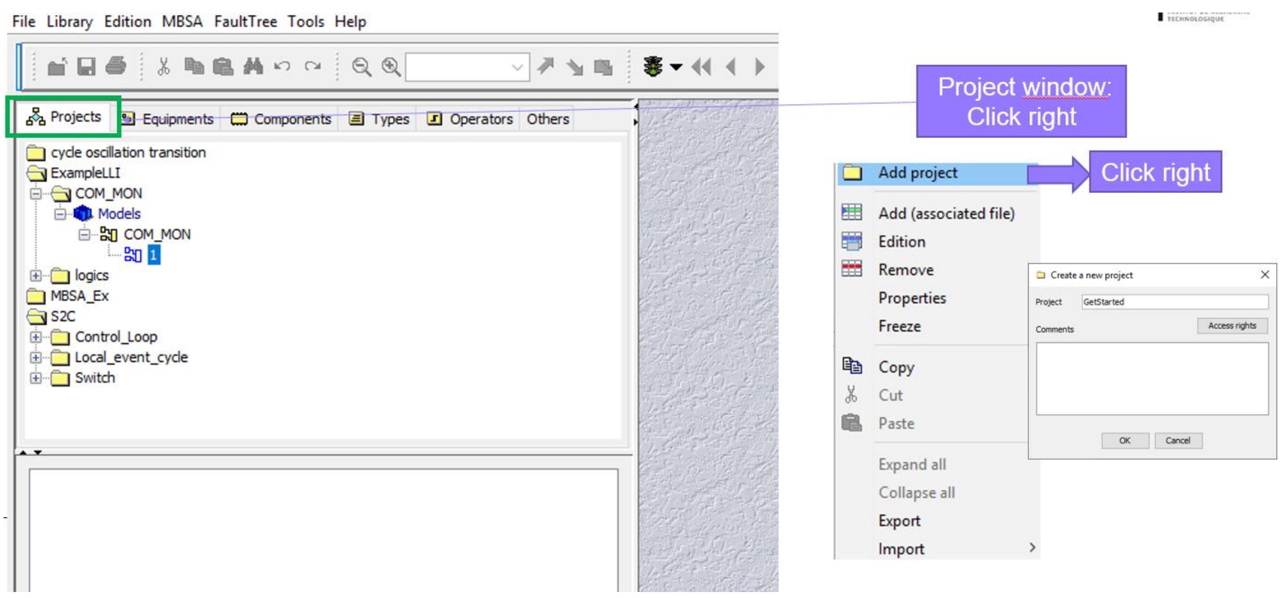


Figure 78: Cecilia GUI – create new project

Then to add the system, the first step in Cecilia consists in create and organize the model. Each model has to be divided in:

Project > System > Model > Equipment > Component > issue.

To add the system, right click on add Add button in the following menu:

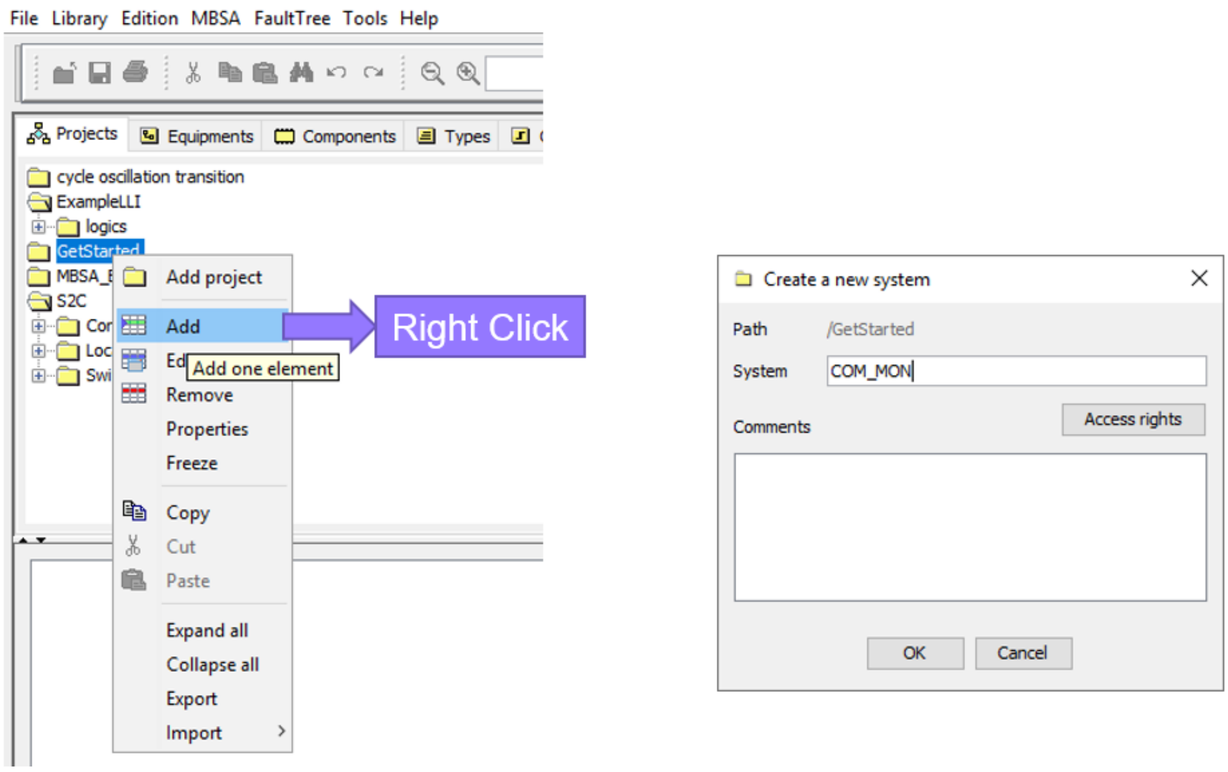


Figure 79: Cecilia GUI – adding the system to the project

Then to add the model, right click on add Add-> Model button in the following menu:

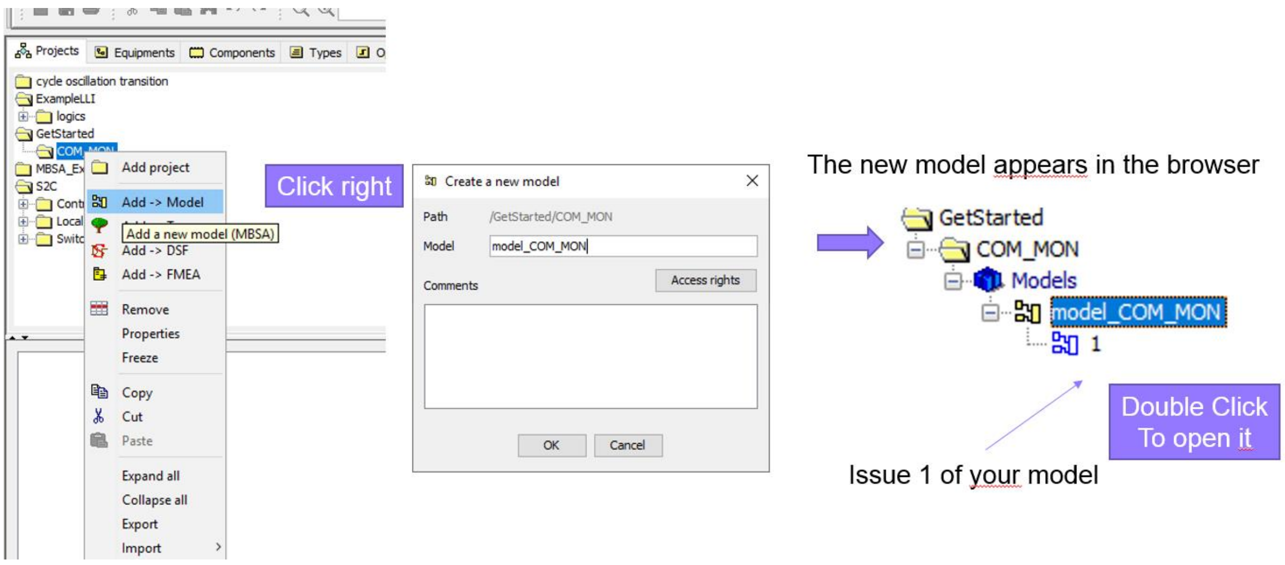


Figure 80: Cecilia GUI – adding the model to the system

Now it is possible to add component and equipment. The hierarchical order described in introduction of this chapter has to be respected (Project > System > Model > Equipment > Component > issue). The next figure shows a graphic representation of this hierarchical order.

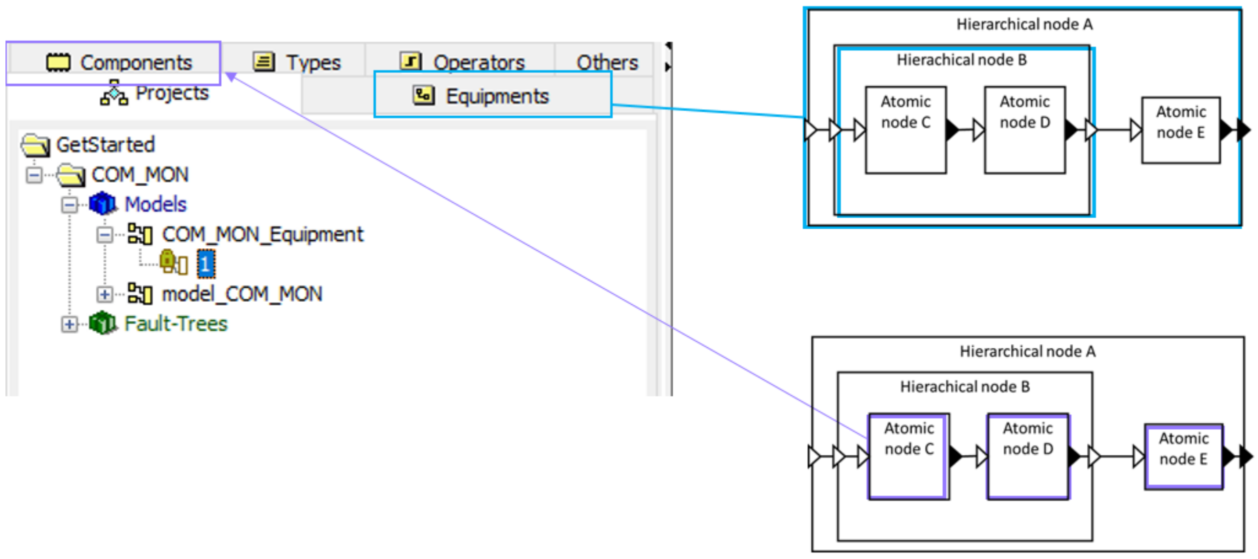


Figure 81: Cecilia GUI – adding component and equipment

Refer to §5 to create the models to create the domains suitable for your needs.

14.1.4 Create a domain (type in Cecilia)

To create a type, the following GUI has to be used:

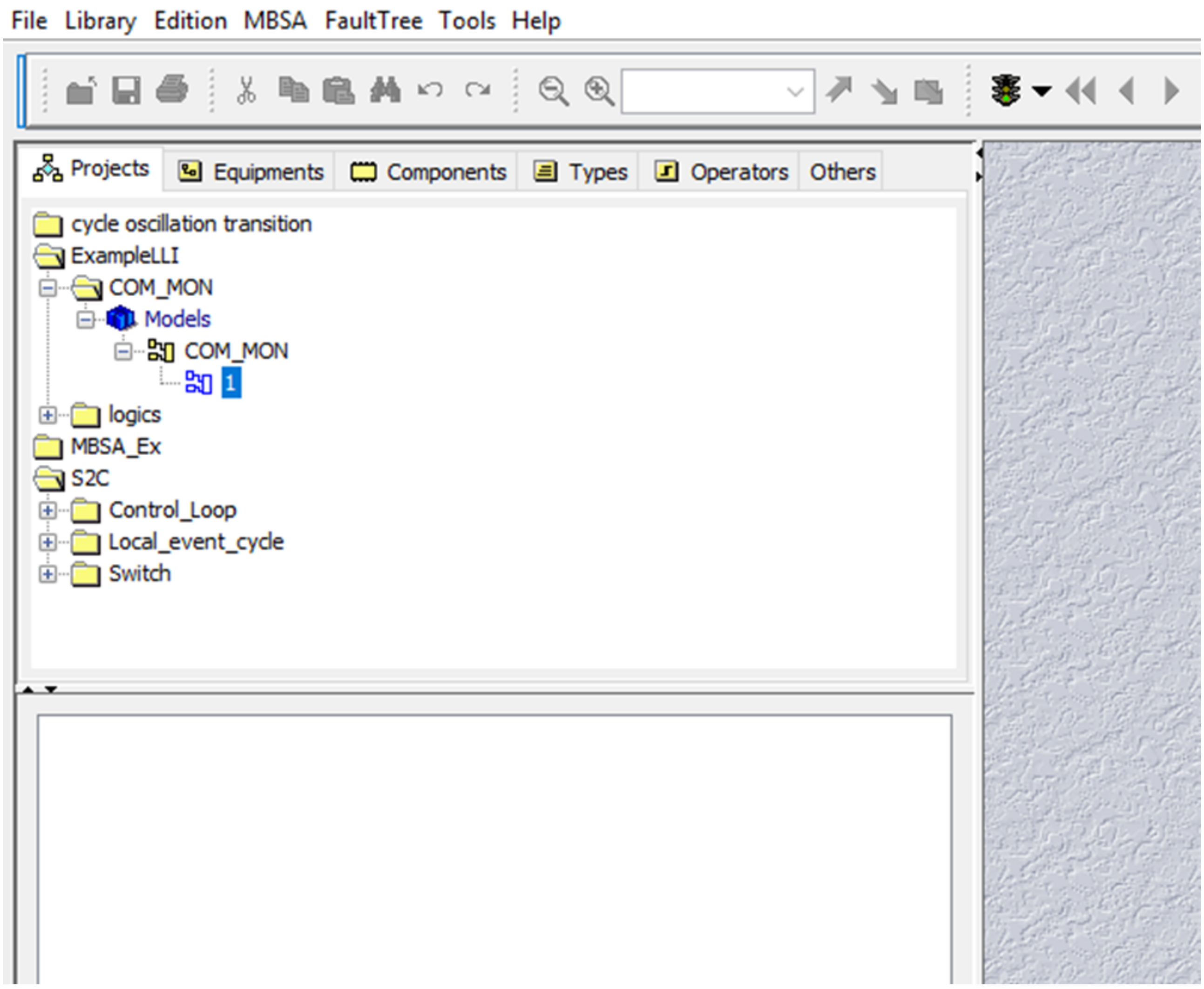


Figure 82: Cecilia GUI – domain creation #1

The type button allow user to define the domains, of the State and of the flow variable.

Then a new family has to be created. This would be the sub category of the types of the project, as described in the following figure:

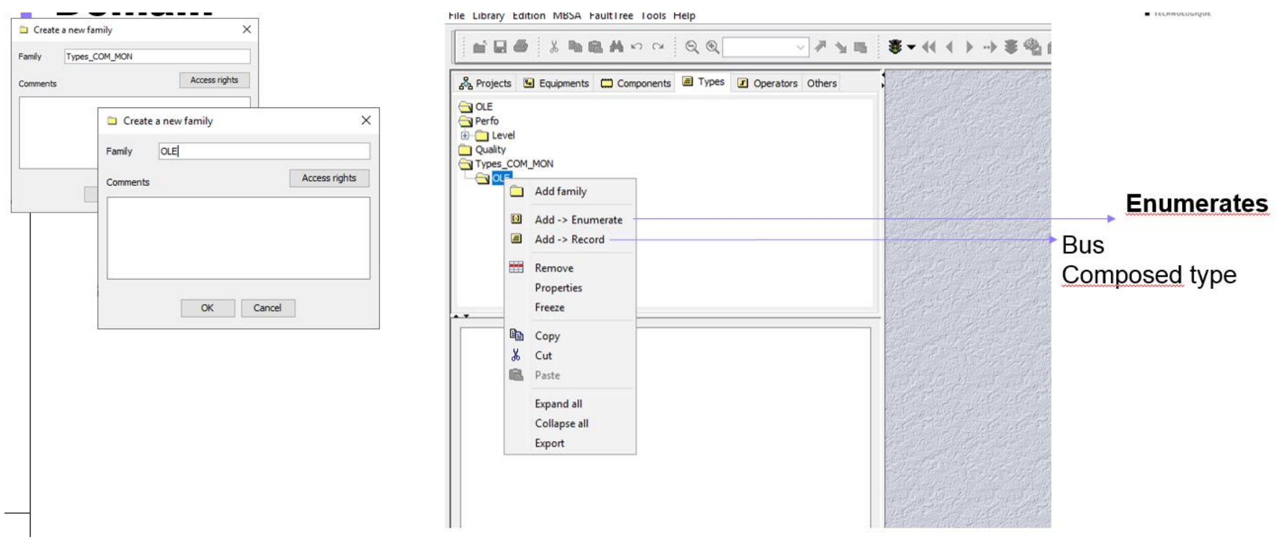


Figure 83: Cecilia GUI – domain creation #2

When creating an enumerate type, the definition of possible state domains has to be perform thanks to the following interface:

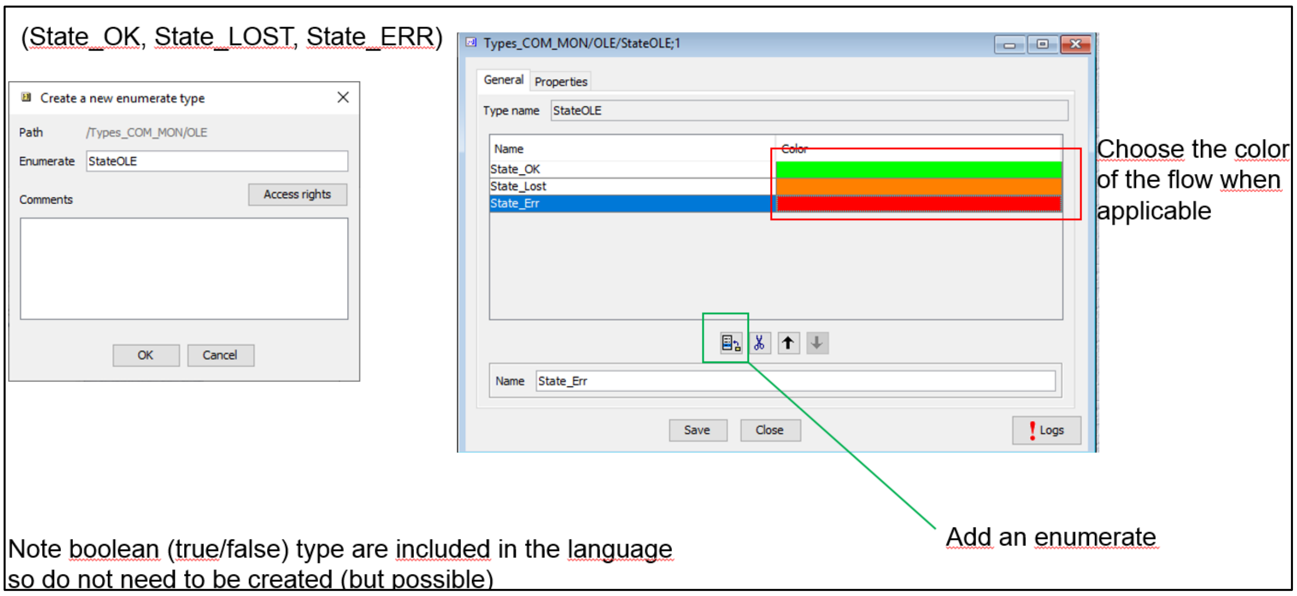
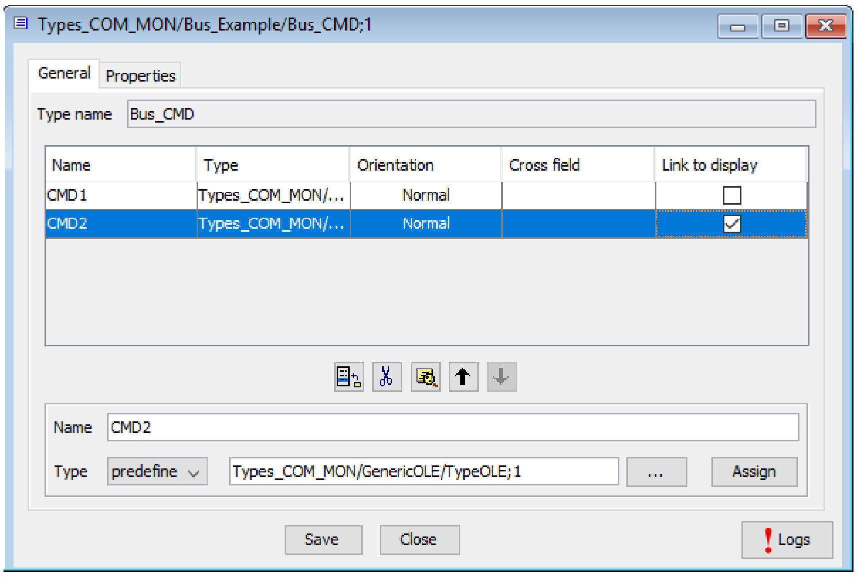

(State_OK, State_LOST, State_ERR)
Create a new enumerate type
Path
/Types_COM_MON/OLE
Enumerate
StateOLE
Comments
Access rights
OK
Cancel
Types_COM_MON/OLE/StateOLE;1
General
Properties
Type name
StateOLE
Name
Color
State_OK
State_Lost
State_Err
Name
State_Err
Save
Close
Logs
Choose the color of the flow when applicable
Add an enumerate
Note boolean (true/false) type are included in the language so do not need to be created (but possible)

Figure 84: Cecilia GUI – domain creation #3 - enumerate

When creating a record type (bus), the definition of possible state domains has to be perform thanks to the following interface:



Type name: Bus_CMD

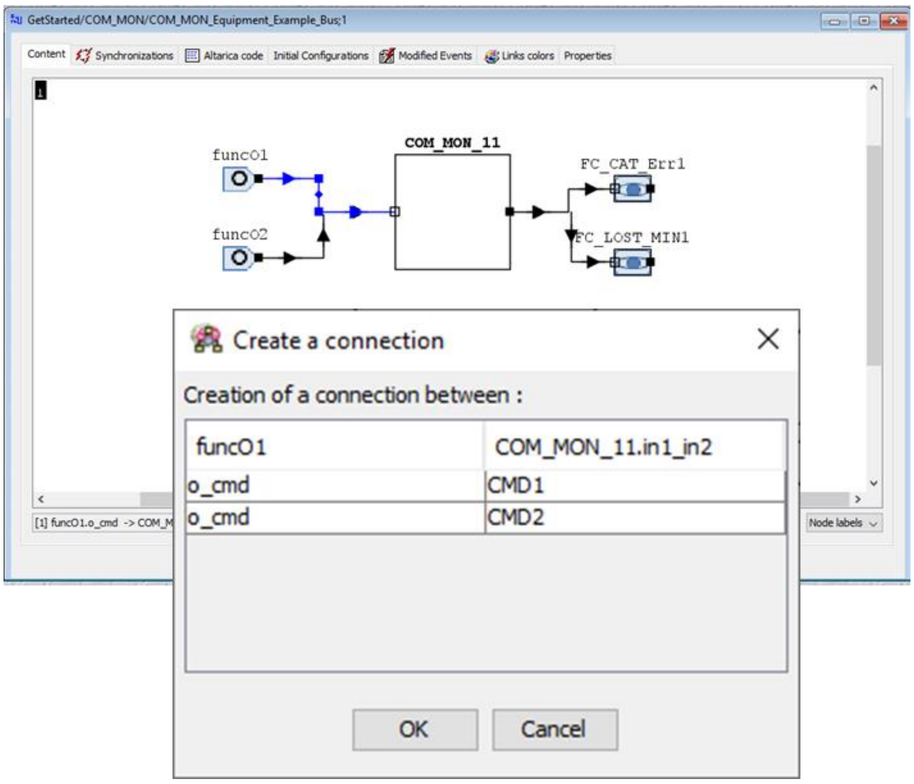
Name	Type	Orientation	Cross field	Link to display
CMD1	Types_COM_MON/...	Normal		☐
CMD2	Types_COM_MON/...	Normal		✓

Name: CMD2

Type: predefine — Types_COM_MON/GenericOLE/TypeOLE;1 — ... — Assign

Figure 85: Cecilia GUI – domain creation #4 – record

Then the connection to the data is made through the graphical interface:



Creation of a connection between :

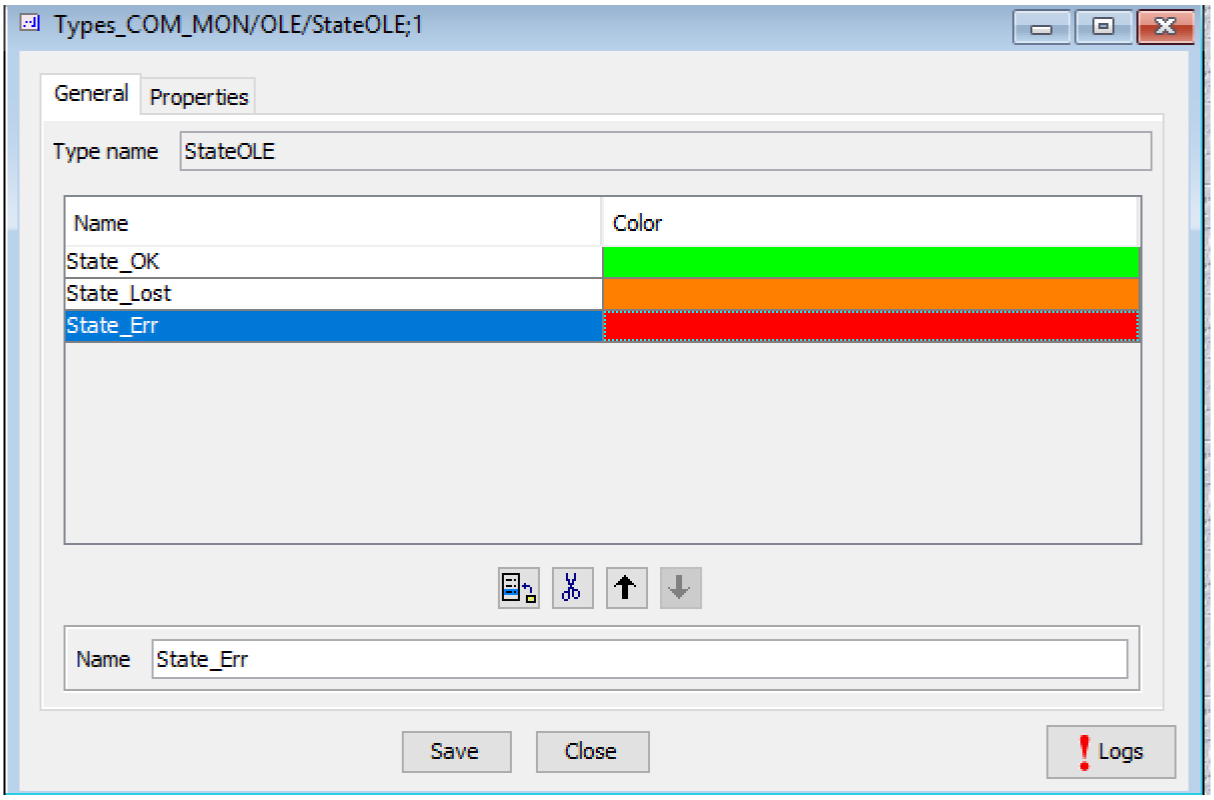
func01	COM_MON_11.in1_in2
o_cmd	CMD1
o_cmd	CMD2

Figure 86: Cecilia GUI – domain #5 – record data connection creation

The types are now created.

14.1.5 Function and color of the domains

The colors can be allocated to the domain in the below window:



Types_COM_MON/OLE/StateOLE;1

Type name: StateOLE

Name	Color
State_OK	🟩
State_Lost	🟧
State_Err	🟥

Name: State_Err

Figure 87: Cecilia GUI – domain #6 domain color

The colors for the links can be allocated in the below window:

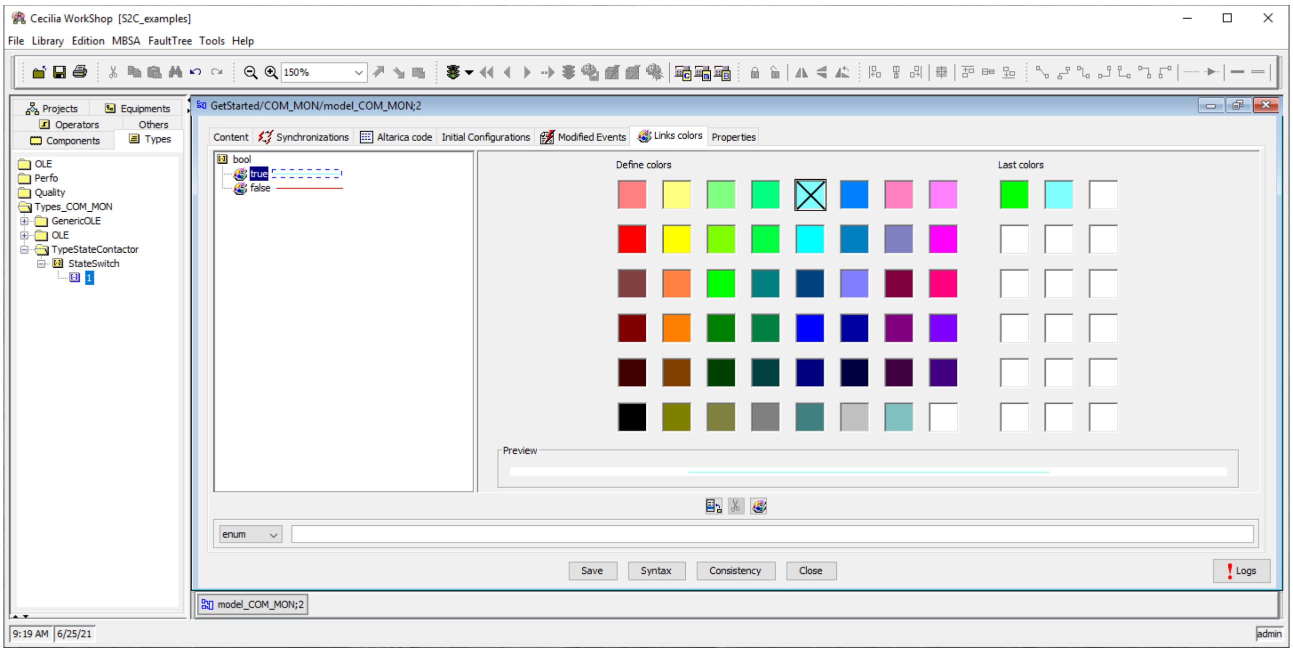


Figure 88: Cecilia GUI – domain #7 links color

14.1.6 Modelling unit creation in the tools; contactor example

This chapter deals with how to build the modelling unit with the example of the COM/MON contactor. It describes the GUI (Graphical User Interface)).

The first step in Cecilia consists in create and organize the model. Each model has to be divided in:

Project > System > Model > Equipment > Component > issue.

Then the first step to create the modelling unit has to be done at the component level:

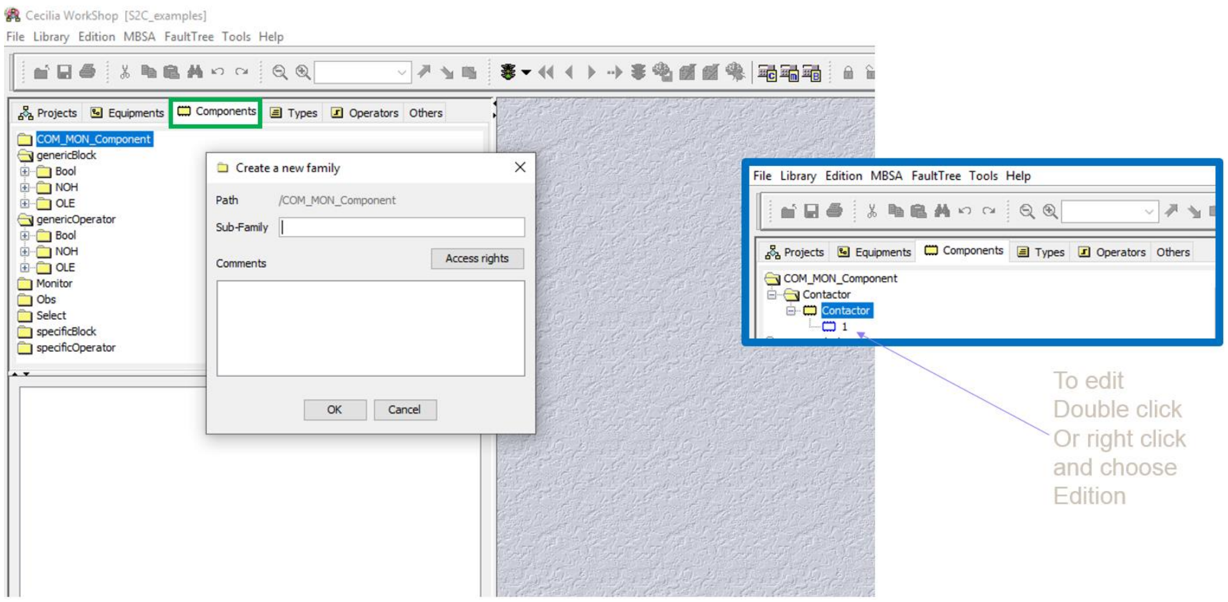


Figure 89: Cecilia GUI – modelling unit creation #1

Then right click on the component contactor to add a new issue of the instance (if there are more than on component from the same family, i.e. more than two contactors in our case):

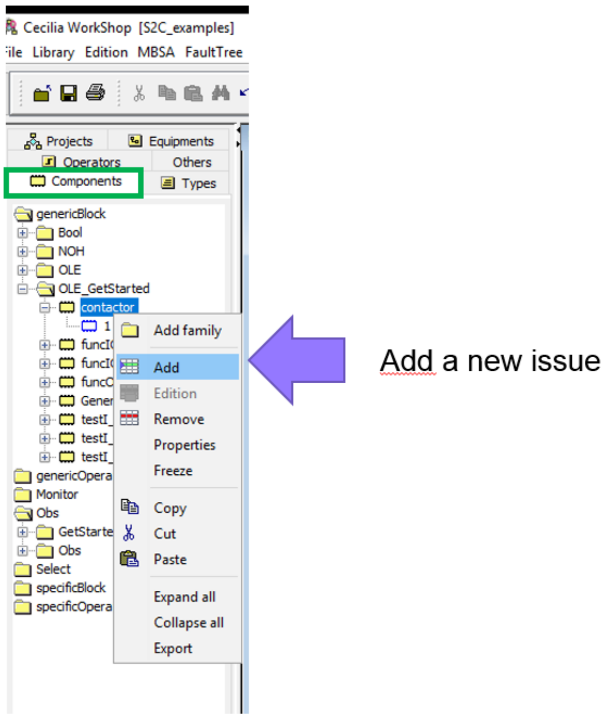
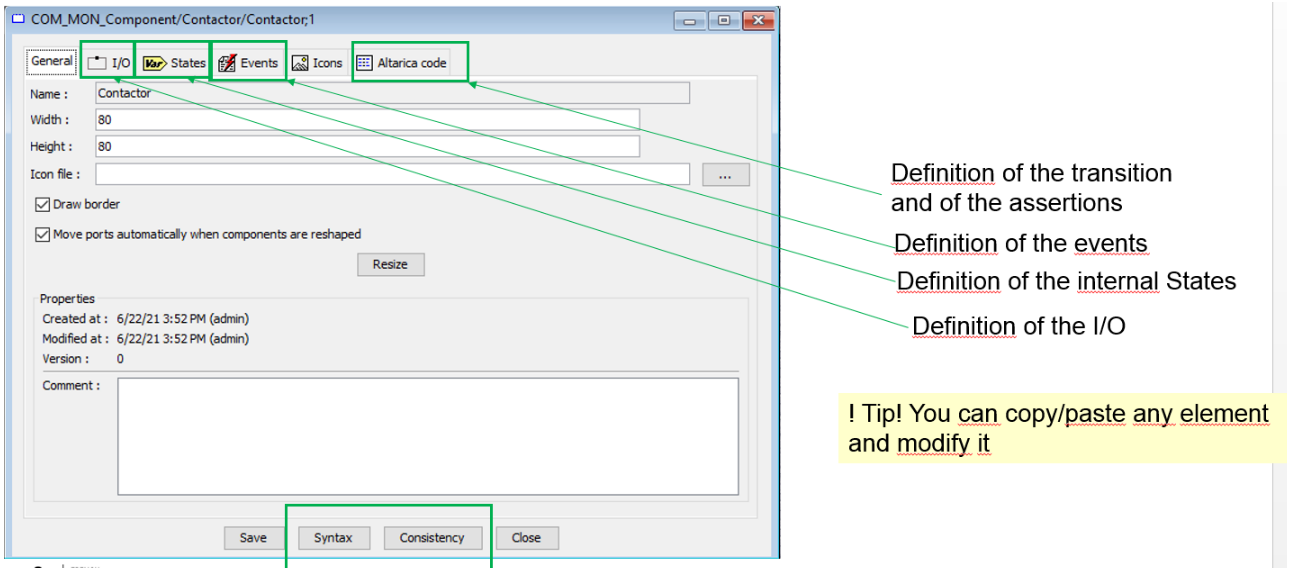


Figure 90: Cecilia GUI – modelling unit creation #2

There is the possibility to copy paste and modify and create a new issue, to replace. Then it is time to define the issue of the component: the contactor itself.


COM_MON_Component/Contactor/Contactor;1
General
I/O
States
Events
Icons
Altarica code
Resize
Save
Syntax
Consistency
Close
Definition of the transition and of the assertions
Definition of the events
Definition of the internal States
Definition of the I/O
! Tip! You can copy/paste any element and modify it

Name : Contactor
Width : 80
Height : 80
Icon file :
☑ Draw border
☑ Move ports automatically when components are reshaped

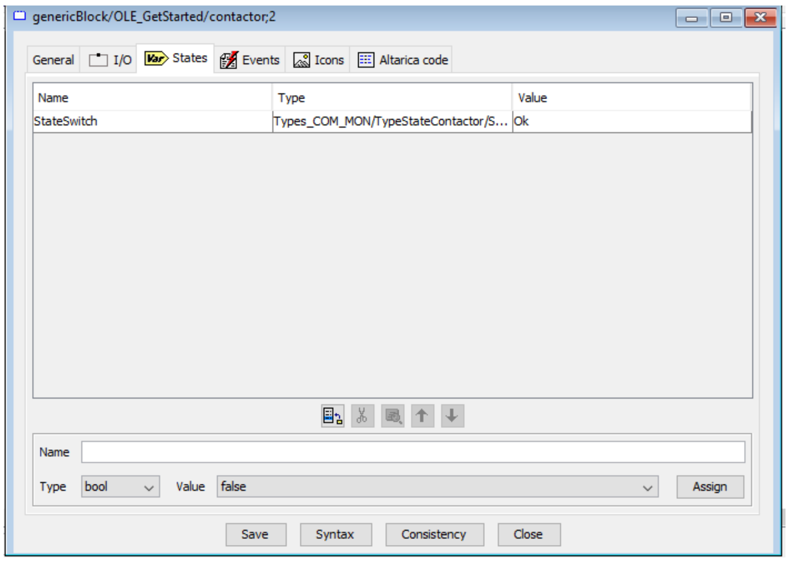
Properties
Created at : 6/22/21 3:52 PM (admin)
Modified at : 6/22/21 3:52 PM (admin)
Version : 0
Comment :

Figure 91: Cecilia GUI – modelling unit creation #3 – component defintion

The buttons highlighted in green allow to check the AltaRica code. To use the issue of the component in the model the assertion has to be define. Please refer to §(e).

(a) Create an internal state variable

To define an internal state variable of the model open the States tab:



Name	Type	Value
StateSwitch	Types_COM_MON/TypeStateContactor/S...	Ok

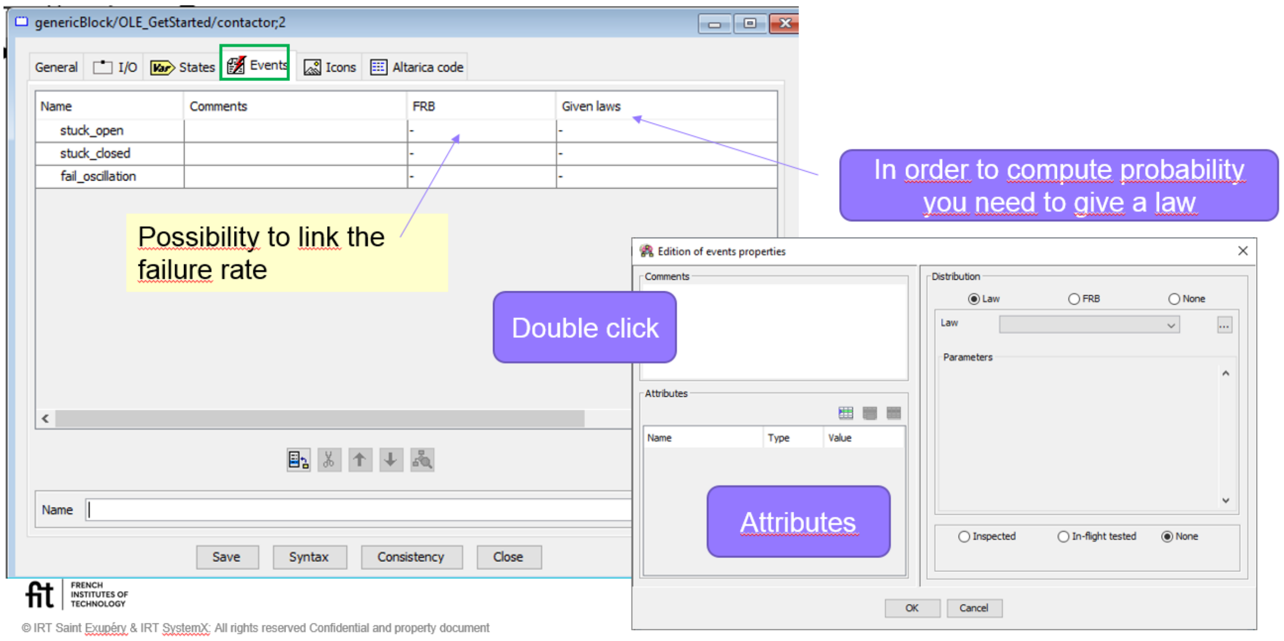
Name:
Type: bool Value: false

Figure 92: Cecilia GUI – modelling unit creation #5 – States tab

There are predefined types. If you need specific type, as Types_COM_MON in the example above go to §0 to deal with the types.

(b) Create events and transitions

To define the events that will impact the component, open the Events tab:


genericBlock/OLE_GetStarted/contactor;2
General
I/O
States
Events
Icons
Altarica code
Possibility to link the failure rate
In order to compute probability you need to give a law
Double click
Attributes
Edition of events properties
Comments
Attributes
Distribution
Law
FRB
None
Law
Parameters
Inspected
In-flight tested
None
OK
Cancel
Save
Syntax
Consistency
Close
Name
© IRT Saint Exupéry & IRT SystemX: All rights reserved Confidential and property document

Name	Comments	FRB	Given laws
stuck_open		-	-
stuck_closed		-	-
fail_oscillation		-	-

Figure 93: Cecilia GUI – modelling unit creation #6 – events definition

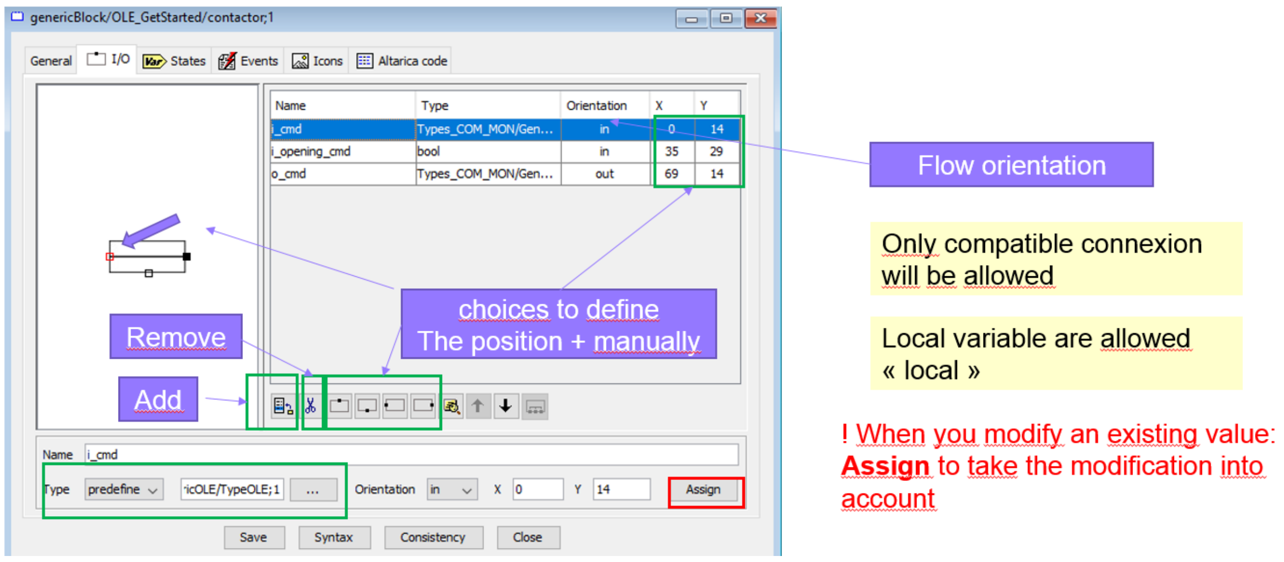
(c) Populate the probability laws

The probability laws have to be populated in the Given Law field from the Event tab in the modelling unit definition window as shown in Figure 93: Cecilia GUI – modelling unit creation #6 – events definition.

(d) Create the flows

Prior to define the flows, it is necessary to define the inputs and outputs of each component.

To define the inputs / outputs of the model open the I/O tab:



Name	Type	Orientation	X	Y
i_cmd	Types_COM_MON/Gen...	in	0	14
i_opening_cmd	bool	in	35	29
o_cmd	Types_COM_MON/Gen...	out	69	14

Name i_cmd
Type predefine ·icOLE/TypeOLE;1 ... Orientation in X 0 Y 14

Only compatible connexion will be allowed

Local variable are allowed « local »

! When you modify an existing value: **Assign** to take the modification into account

Figure 94: Cecilia GUI – modelling unit creation #4 – I/O tab

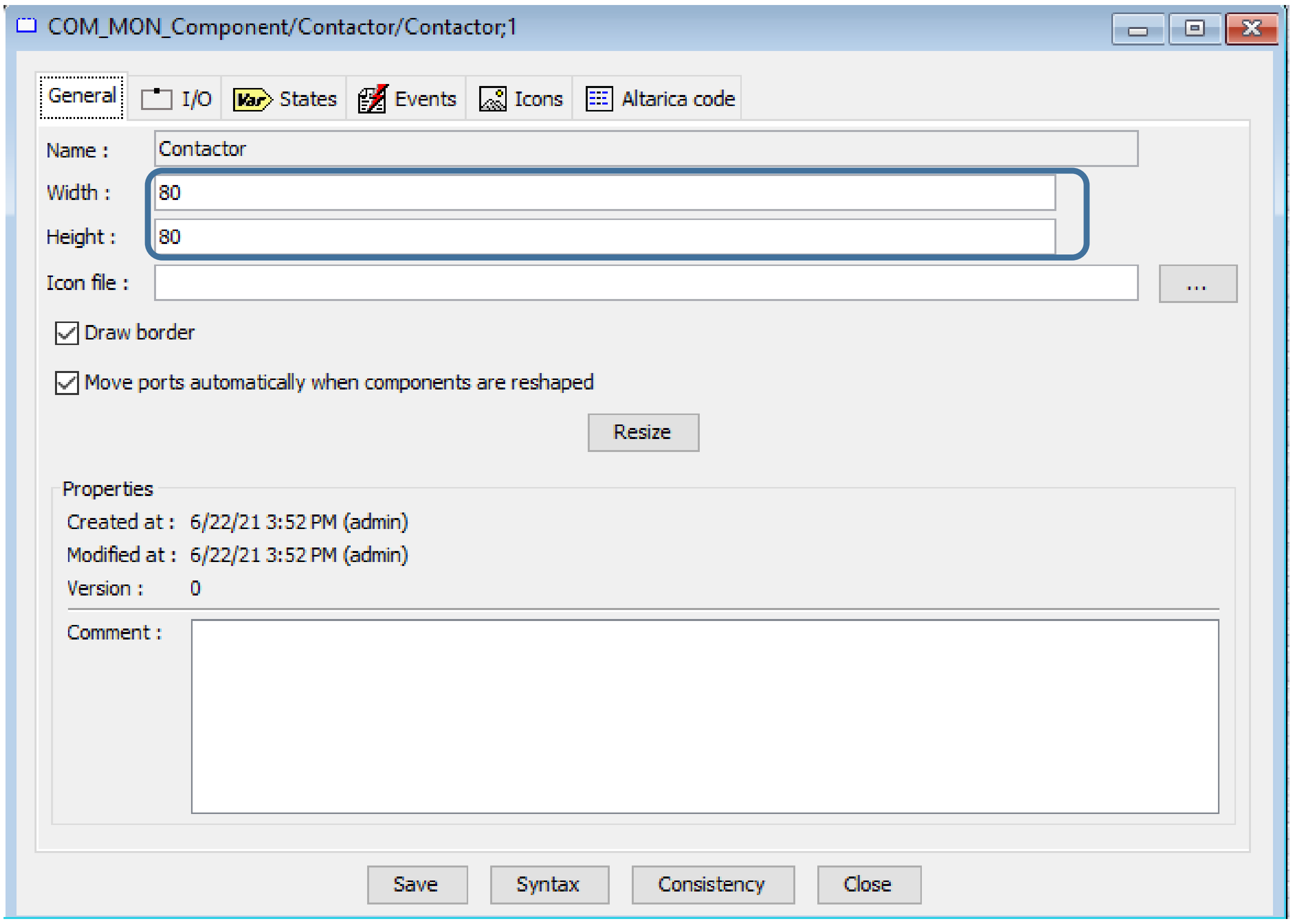
Then to define the flows, come back to the graphical view and trace the link between the ouput of the first component and the input of the second component.

(e) Create an assertion

To create an assertion, open the AltaRica code tab of the component properties window. Please refer to §4.5 to deal with assertion.

(f) Set the brick size and the flows position

The brick size has to be defined in the General tab by populating the below fields:

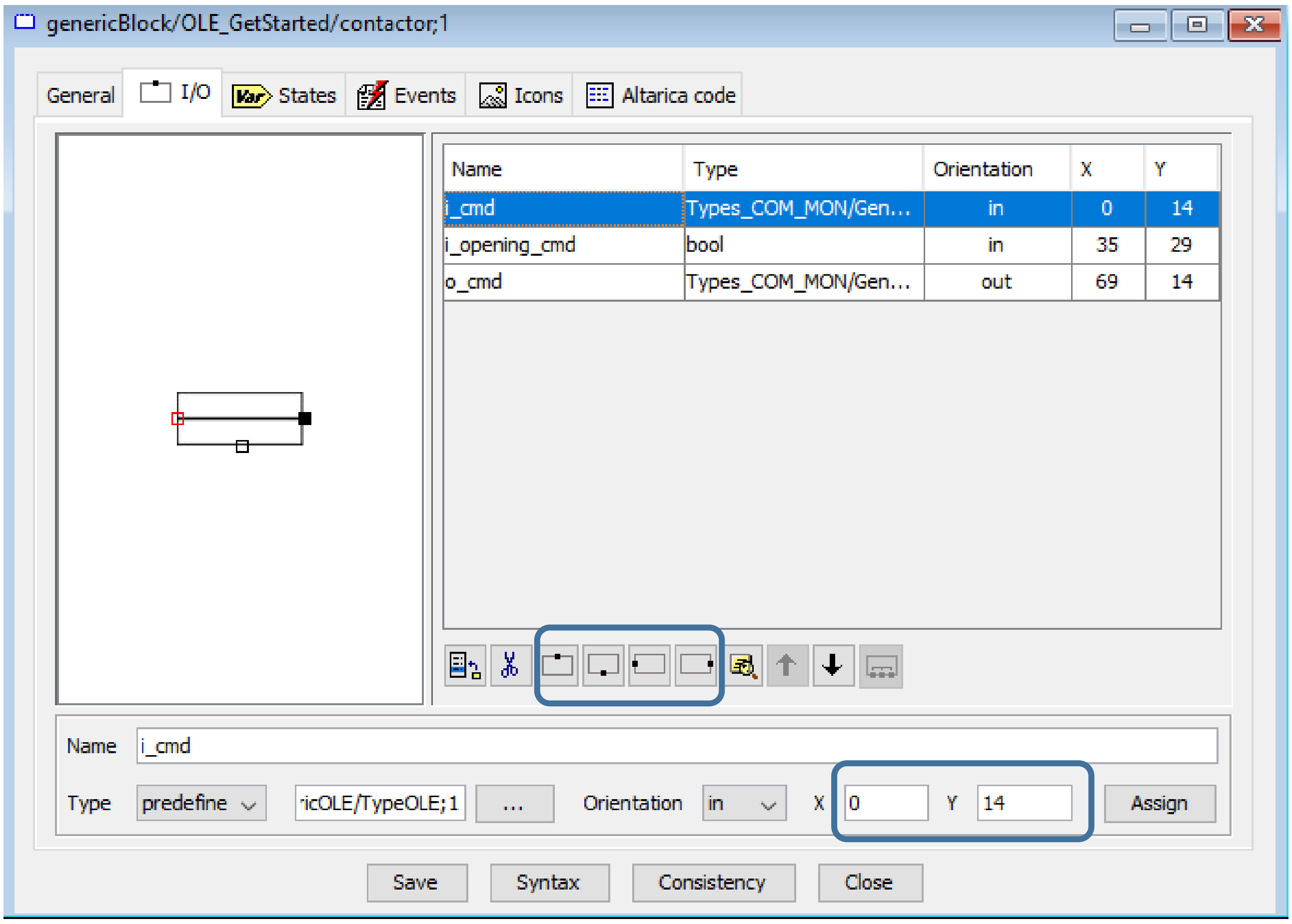

COM_MON_Component/Contactor/Contactor;1

Name : Contactor
Width : 80
Height : 80
Icon file :
☑ Draw border
☑ Move ports automatically when components are reshaped

Properties
Created at : 6/22/21 3:52 PM (admin)
Modified at : 6/22/21 3:52 PM (admin)
Version : 0
Comment :

Figure 95: Cecilia GUI – set the size of the modelling unit

The flow position has to be defined in the I/O tab:


genericBlock/OLE_GetStarted/contactor;1

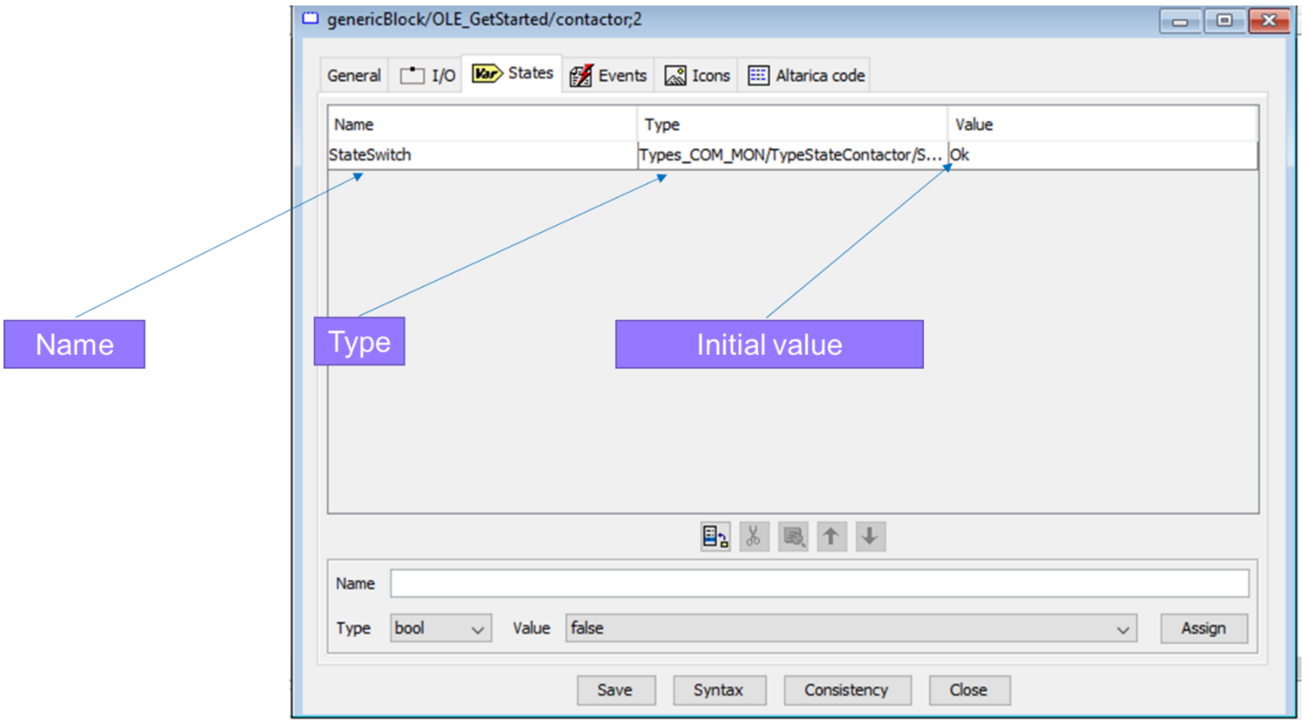
Name	Type	Orientation	X	Y
i_cmd	Types_COM_MON/Gen...	in	0	14
i_opening_cmd	bool	in	35	29
o_cmd	Types_COM_MON/Gen...	out	69	14

Name: i_cmd
Type: predefine ˅ ·icOLE/TypeOLE;1 ... Orientation: in ˅ X: 0 Y: 14 Assign

Figure 96: Cecilia GUI – flows position

The position can be set either with predefined position or by populating the x,y coordinates.

(g) Create the conditional style

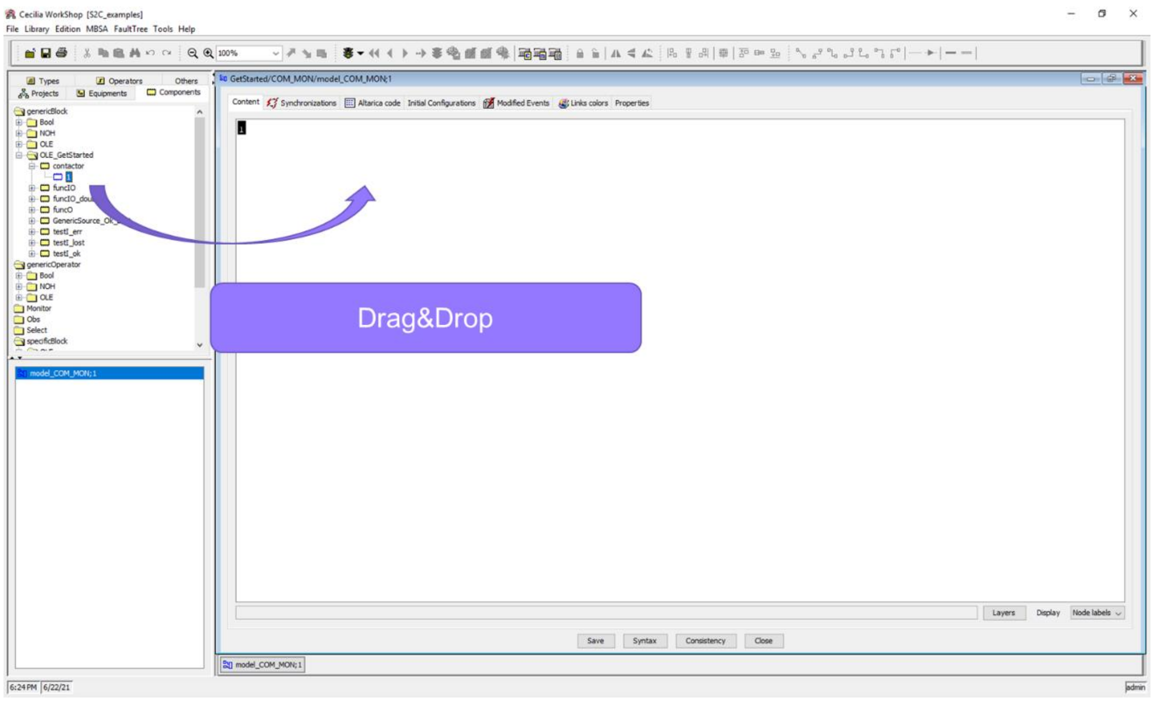


Name	Type	Value
StateSwitch	Types_COM_MON/TypeStateContactor/S…	Ok

Name:
Type: bool
Value: false

(h) Instanciate a « modelling unit »

The instantiation of the model unit is made through a drag and drop from the tree to the main window:



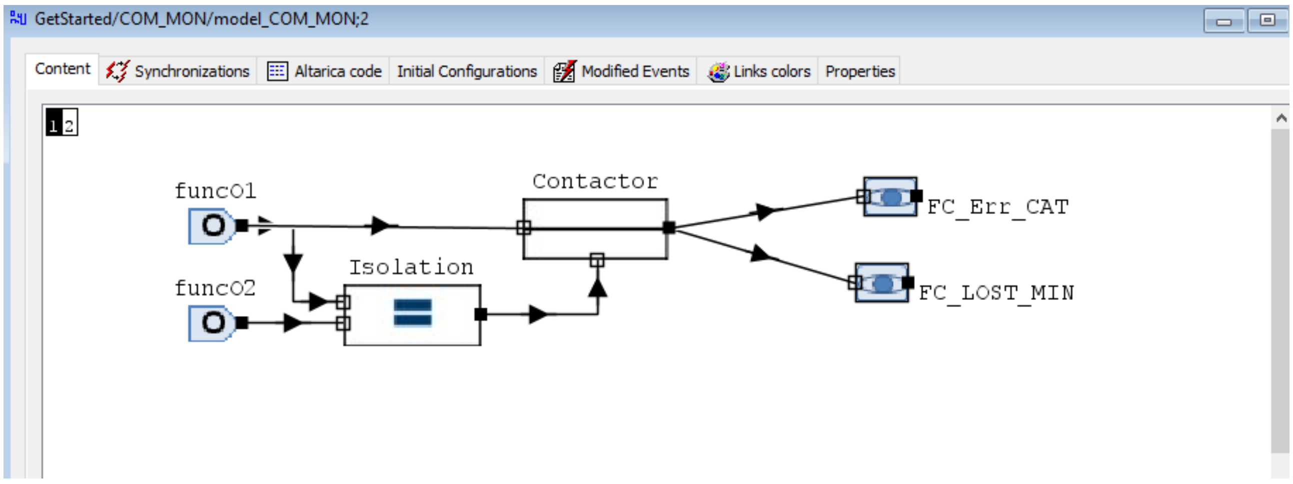
(i) Trace the links betwen « modelling unit »

Connect the modeling units, « components », « equipment » and observers

If necessary set configuration (States initial values), otherwise set to defined default values

Set synchronizations.

Note: flows types and orientation need to be compatible



(j) Create the observers (link with FC)

The observers are modeling artefacts used in the MBSA tools to calculate the Failure Conditions. It consists in a component that is observing the state of the variable associated to the failure condition. By calculating the probability of the different states of this observers, we can get the probability of the FC.

In our example the observers are defined as below:

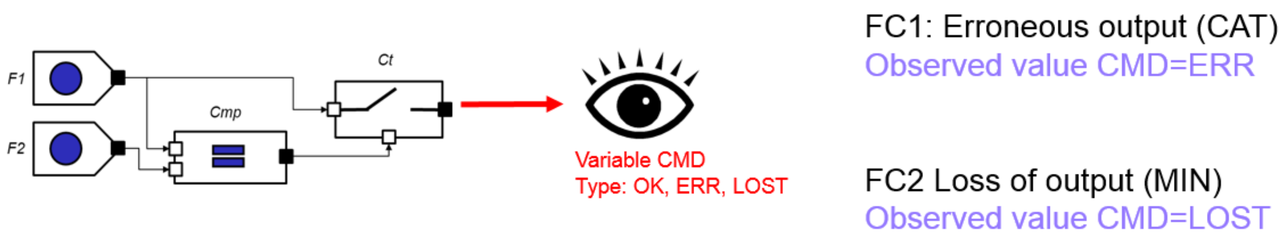


Figure 97: Observers definition

In the Cecilia tool, the observer are the below components (FC_Err_CAT and FC_LOST_MIN):

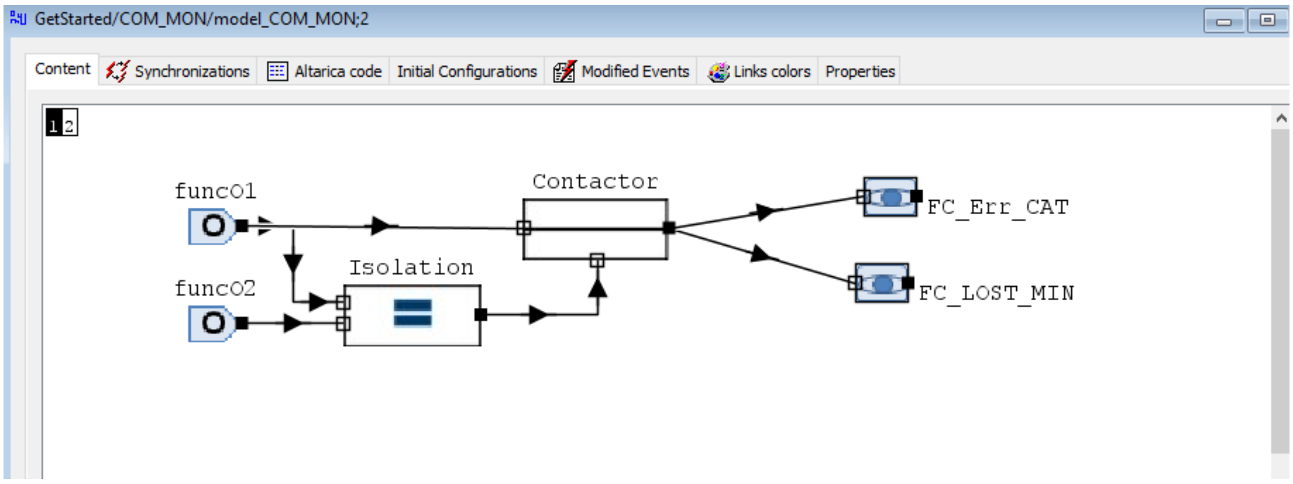


Figure 98: Cecilia GUI – observers

The code inside the observer is the equation that lead to the observed variable to be the right value depending on the failure condition. This code has to be written in AltaRica.

Refer to §5.2 to get the code.

(k) Launch a step by step simulation

To launch a step by step simulation use the following interface:

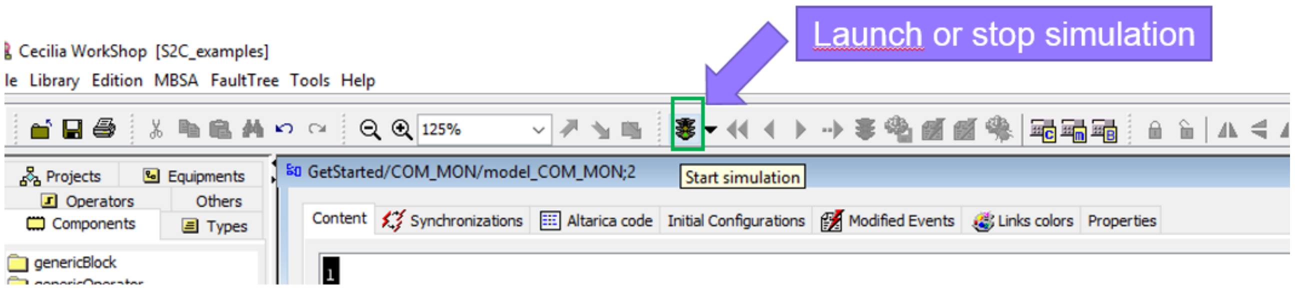
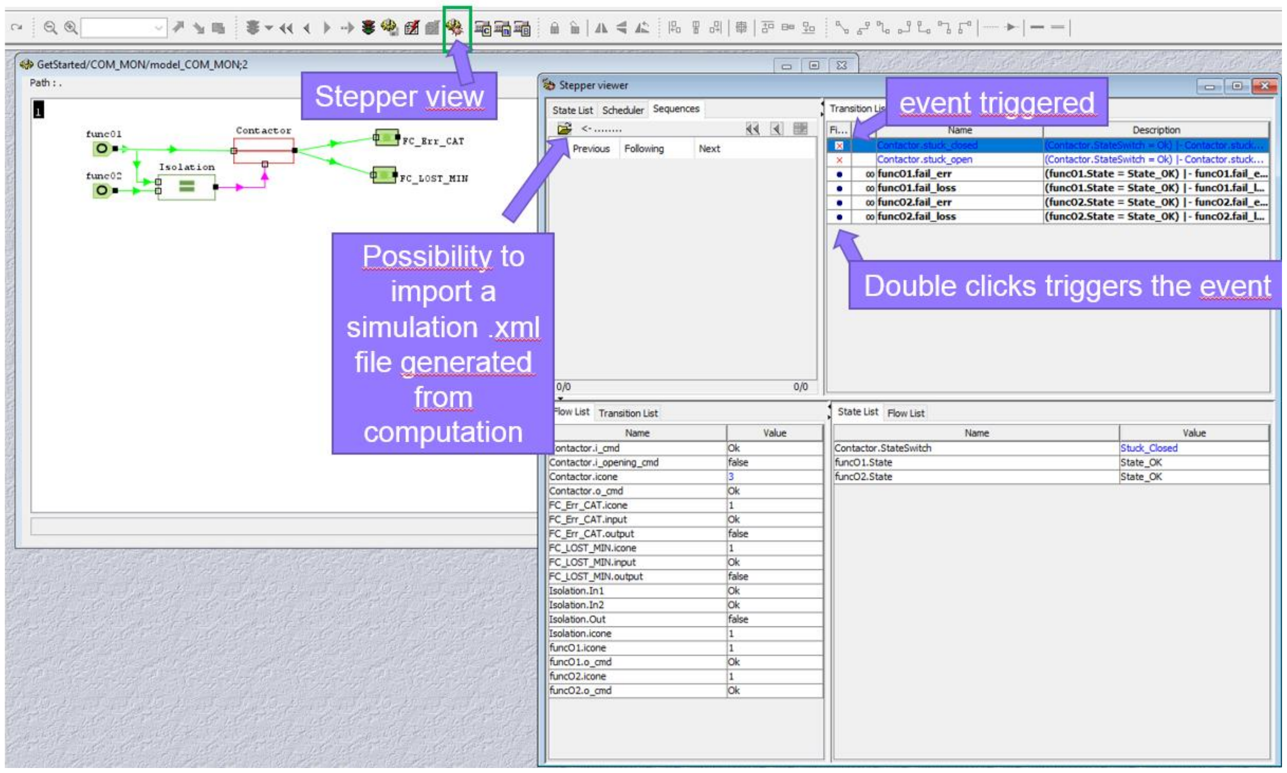


Figure 99: Cecilia GUI – launching step by step simulation

Then to analyze and configure your step by step simulation, the following window has to be use:



Fi...		Name	Description
✗		Contactor.stuck_closed	(Contactor.StateSwitch = Ok) |- Contactor.stuck...
✗		Contactor.stuck_open	(Contactor.StateSwitch = Ok) |- Contactor.stuck...
●	∞	func01.fail_err	(func01.State = State_OK) |- func01.fail_e...
●	∞	func01.fail_loss	(func01.State = State_OK) |- func01.fail_l...
●	∞	func02.fail_err	(func02.State = State_OK) |- func02.fail_e...
●	∞	func02.fail_loss	(func02.State = State_OK) |- func02.fail_l...

Flow List | Transition List

Name	Value
ontactor.i_cmd	Ok
Contactor.i_opening_cmd	false
Contactor.icone	3
Contactor.o_cmd	Ok
FC_Err_CAT.icone	1
FC_Err_CAT.input	Ok
FC_Err_CAT.output	false
FC_LOST_MIN.icone	1
FC_LOST_MIN.input	Ok
FC_LOST_MIN.output	false
Isolation.In1	Ok
Isolation.In2	Ok
Isolation.Out	false
Isolation.icone	1
func01.icone	1
func01.o_cmd	Ok
func02.icone	1
func02.o_cmd	Ok

State List | Flow List

Name	Value
Contactor.StateSwitch	Stuck_Closed
func01.State	State_OK
func02.State	State_OK

Figure 100: Cecilia GUI – analysing step by step simulation

(I) Calculate the Cut-Sets

To calculate the cut-sets the first step is to launch a sequence generation. Please use the below GUI :

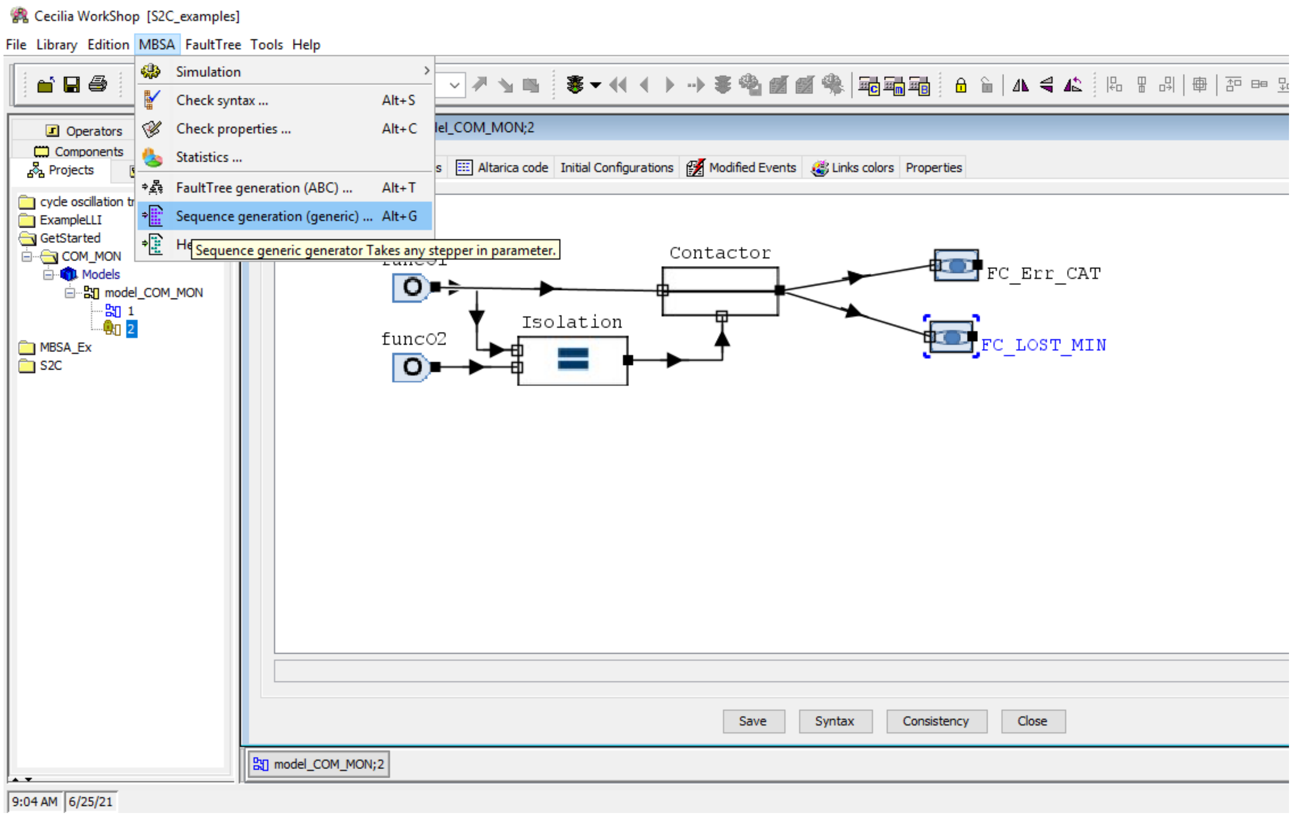


Figure 101: Cecilia GUI – calculate Cut-Sets #1 launching sequence generation

Then configure the output:

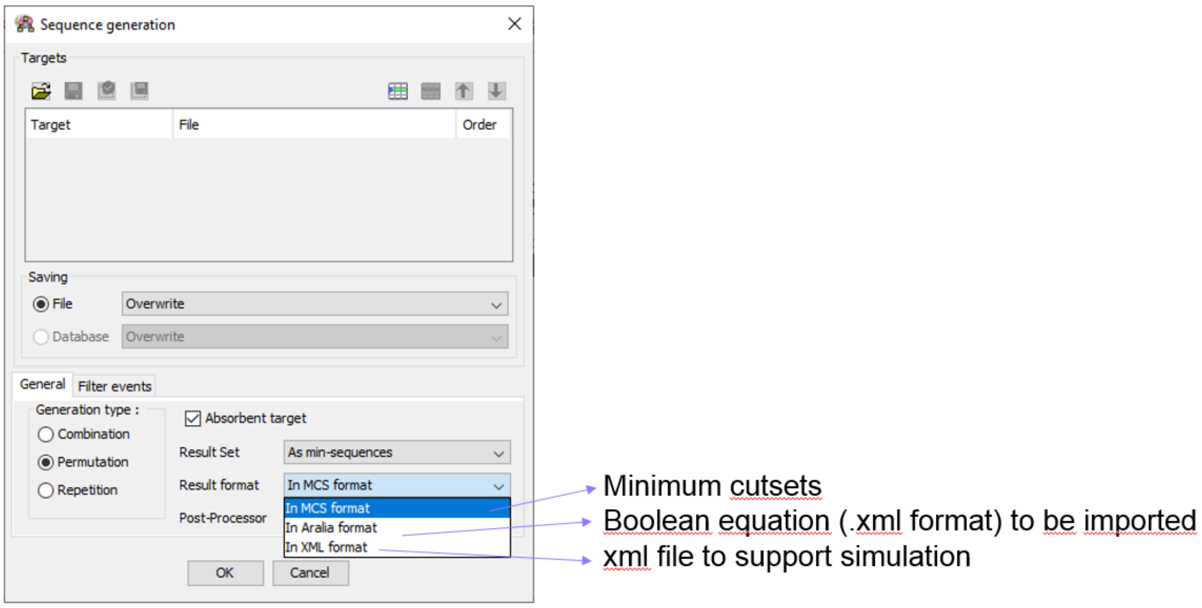
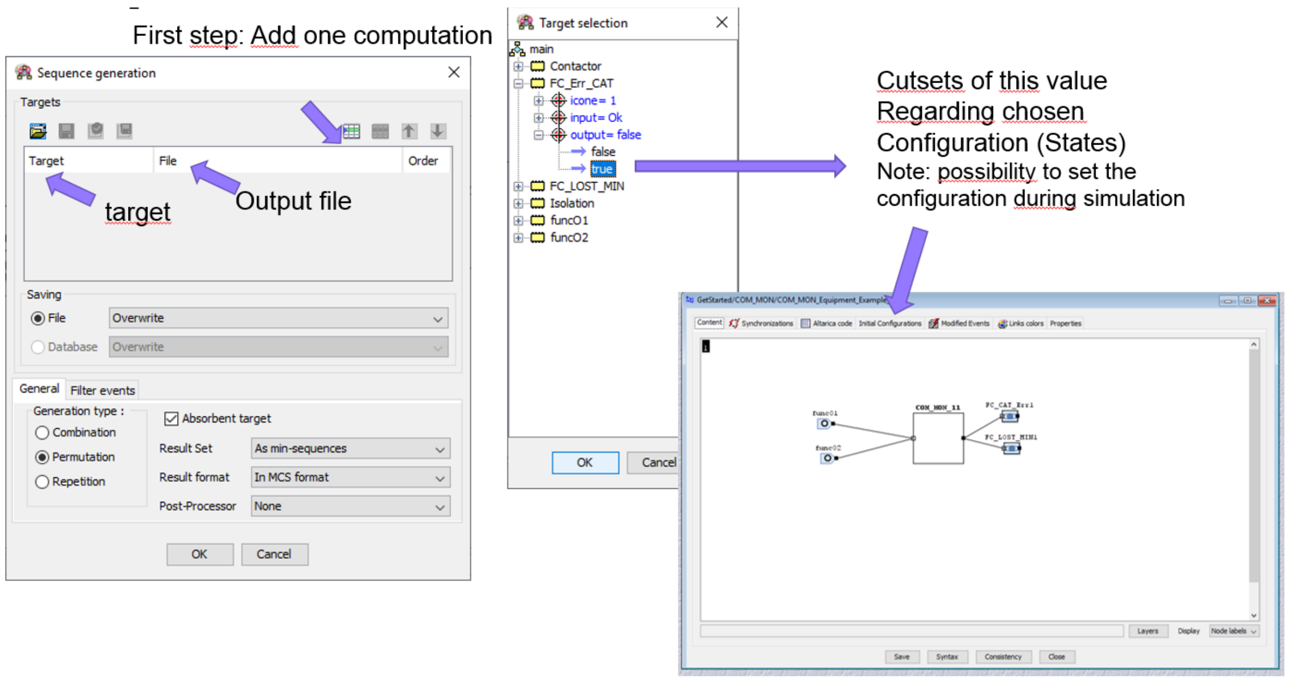


Figure 102: Cecilia GUI – calculate Cut-Sets #2 output configuration

Then the next step is to add one computation:



- First step: Add one computation

Cutsets of this value
Regarding chosen
Configuration (States)
Note: possibility to set the
configuration during simulation

Sequence generation

Targets

Target	File	Order

target

Output file

Saving
- (●) File: Overwrite
- ( ) Database: Overwrite

General | Filter events

Generation type :
- ( ) Combination
- (●) Permutation
- ( ) Repetition

[✓] Absorbent target

Result Set: As min-sequences
Result format: In MCS format
Post-Processor: None

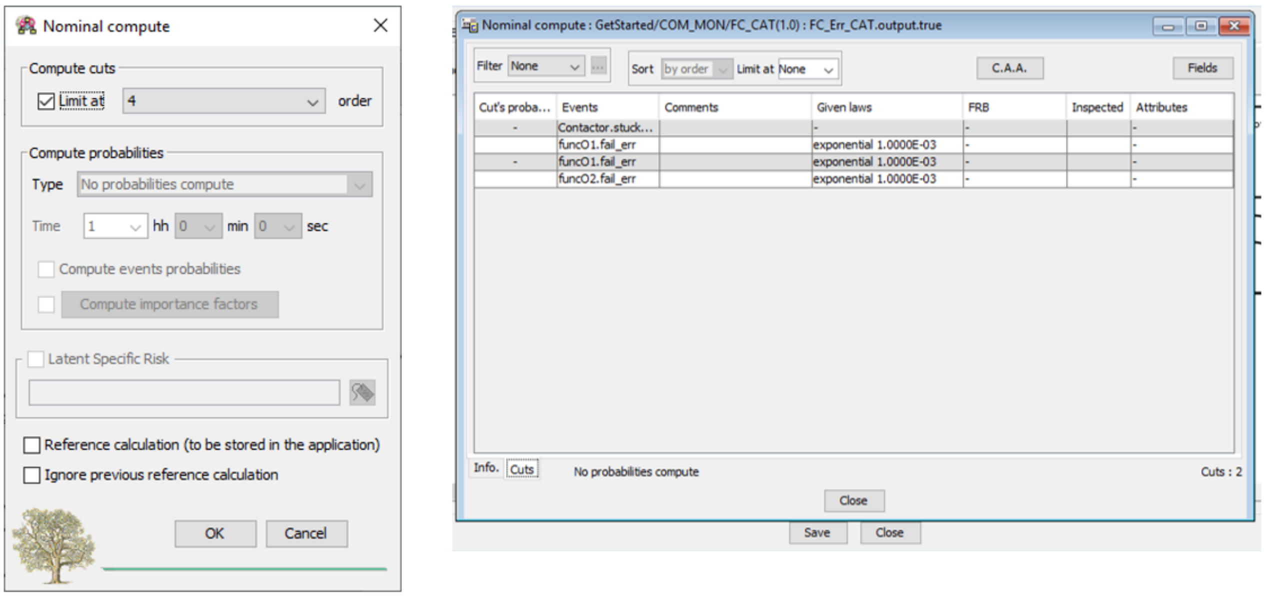
Target selection

```
main
⊞ Contactor
⊟ FC_Err_CAT
   ⊞ icone= 1
   ⊞ input= Ok
   ⊟ output= false
        false
        true
⊞ FC_LOST_MIN
⊞ Isolation
⊞ funcO1
⊞ funcO2
```

GetStarted/COM_MON/COM_MON_Equipment_Example

Figure 103: Cecilia GUI – calculate Cut-Sets #3 add on computation

Then the results are shown in the following window:



Nominal compute

Compute cuts
- [✓] Limit at 4 order

Compute probabilities
- Type: No probabilities compute
- Time: 1 hh 0 min 0 sec
- [ ] Compute events probabilities
- [ ] Compute importance factors

[ ] Latent Specific Risk

[ ] Reference calculation (to be stored in the application)
[ ] Ignore previous reference calculation

Nominal compute : GetStarted/COM_MON/FC_CAT(1.0) : FC_Err_CAT.output.true

Filter: None | Sort: by order | Limit at: None

Cut's proba...	Events	Comments	Given laws	FRB	Inspected	Attributes
-	Contactor.stuck...		-	-		-
	funcO1.fail_err		exponential 1.0000E-03	-		-
-	funcO1.fail_err		exponential 1.0000E-03	-		-
	funcO2.fail_err		exponential 1.0000E-03	-		-

No probabilities compute — Cuts : 2

Figure 104: Cecilia GUI – calculate Cut-Sets #4 Cut-Sets results

(m) Probabilities post traitment

14.1.7 Pictures import for icons

To import picture of icon you need to use the icon tab on the modelling unit window.

14.2 – B- Get Started with the tools – SimfiaNeo

This chapter deals with the basic notions to start modelling with AIRBUS PROTECT SimfiaNeo tool. Please be aware that the full SimfiaNeo User Manual, including another Getting Started, is accessible from the menu Help> User manual.

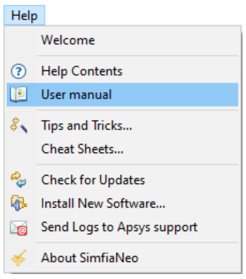


Figure 105: SimfiaNeo GUI - open user manual

14.2.1 Introduction

Before modelling, it is recommended to come back to the questions to be asked before starting, by referring to §5 and GetStarted kit.

The COM/MON pattern example is described in §5.2 and GetStarted.

14.2.2 Open an existing project

In SimfiaNeo, opening a model corresponds to the import functionality. To import a file, do: File> Import… then select General> Existing Projects into Workspace then Next>.

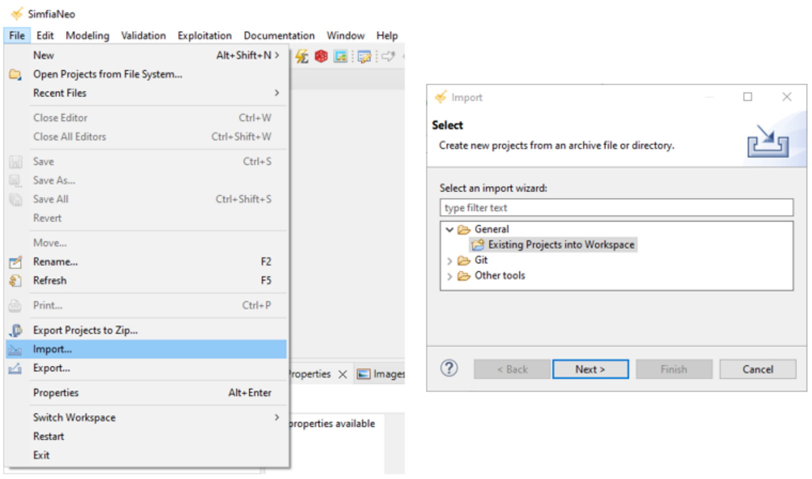
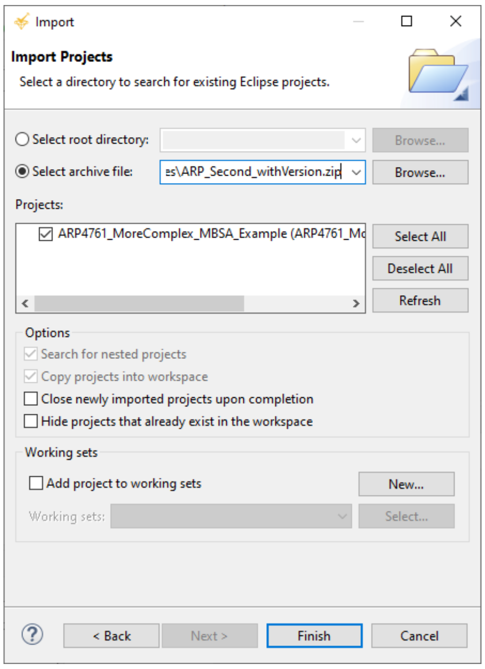


Figure 106: SimfiaNeo GUI - import project #1

Next wizard enables to choose between:

- "Select root directory": enables choosing a folder containing one or several unzipped SimfiaNeo projects

- "Select archive file": enables choosing a zipped (.zip, .tar, .tar.gz, .tgz, .jar) file containing one or several SimfiaNeo projects


Import Projects

Select a directory to search for existing Eclipse projects.

( ) Select root directory:

(●) Select archive file: ≥s\ARP_Second_withVersion.zip

Projects:

☑ ARP4761_MoreComplex_MBSA_Example (ARP4761_Mc

Options

☑ Search for nested projects

☑ Copy projects into workspace

☐ Close newly imported projects upon completion

☐ Hide projects that already exist in the workspace

Working sets

☐ Add project to working sets

Figure 107: SimfiaNeo GUI - import project #2

Once having selected the project(s), click on Finish to import it.

14.2.3 How to create a new project

Open SimfiaNeo tool by clicking on SimfiaNeo.exe. Use the menu File> New> New SimfiaNeo project.

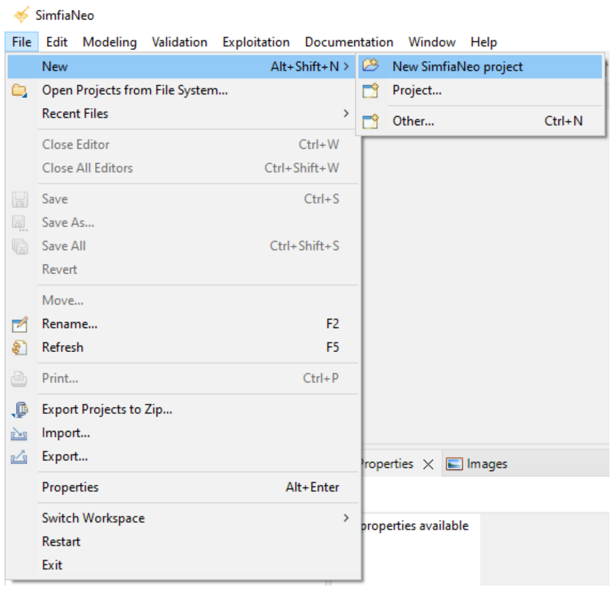
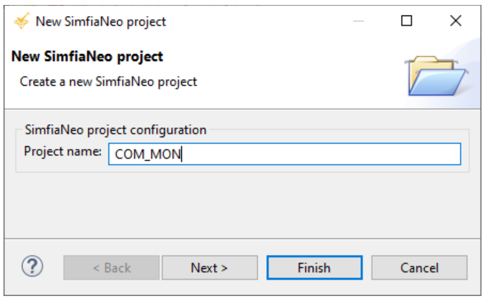


Figure 108: SimfiaNeo GUI - create new project #1

Input a name for your Project and select Finish.



Project name: COM_MON

Figure 109: SimfiaNeo GUI - create new project #2

SimfiaNeo automatically creates a Project, System and Model with the same name. They can be seen in the Model Explorer.

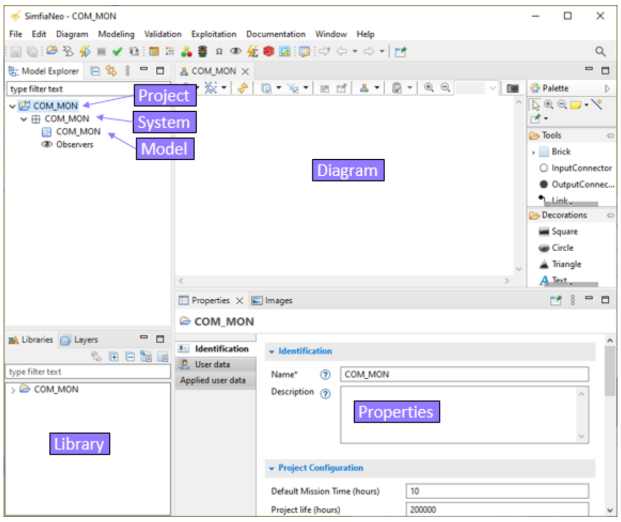


Figure 110: SimfiaNeo GUI - general overview

Top-level diagram of the Model is also automatically opened. Properties view is automatically updated depending on currently selected element in Model Explorer, Diagram, Library…

14.2.4 Create a domain

Domains are managed through the Domains table. Open the table by selecting the menu Modeling> Open domains.

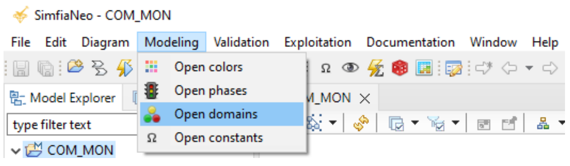
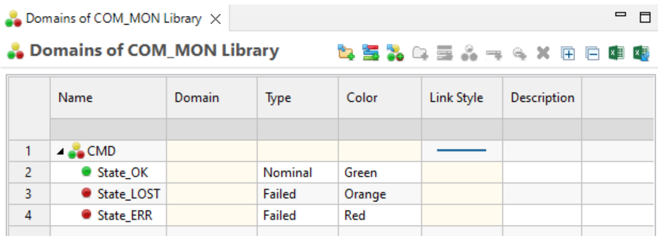


Figure 111: SimfiaNeo GUI - open domains table

Domains can be organized in folders, but this is not mandatory. Creation buttons are situated in the top-right corner of the table.


Domains of COM_MON Library

	Name	Domain	Type	Color	Link Style	Description
1	CMD				———	
2	State_OK		Nominal	Green		
3	State_LOST		Failed	Orange		
4	State_ERR		Failed	Red		

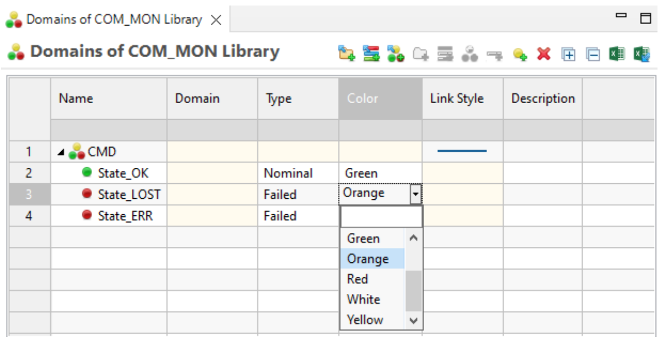
Figure 112: SimfiaNeo GUI - domains table

Names of domains and values can be customized either directly in the table or through the Properties view when the corresponding line/cell is selected in the table. Column named Type is used mainly for documentation purposes and can be ignored when getting started.

Structured domains (for connectors containing several variables) can also be defined in this table. Each variable in the structured domain is called a field.

14.2.5 Colors of the domains

Colors for Step-by-step simulation display can be chosen in the Domains table (see above) with a drop-down menu.



	Name	Domain	Type	Color	Link Style	Description
1	CMD				———	
2	State_OK		Nominal	Green		
3	State_LOST		Failed	Orange		
4	State_ERR		Failed			

Figure 113: SimfiaNeo GUI - modify domains colors

Default list of colors is automatically provided in projects. Additional colors can be defined through the Colors table. Open this table by selecting the menu Modeling> Open colors.

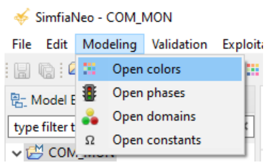
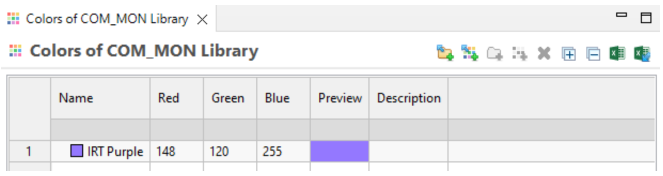


Figure 114: SimfiaNeo GUI - open colors table

Colors can be organized in folders, but this is not mandatory. Creation buttons are situated in the top-right corner of the table.



	Name	Red	Green	Blue	Preview	Description
1	IRT Purple	148	120	255	🟪	

Figure 115: SimfiaNeo GUI - define colors

Colors are defined with a name and a RGB code.

14.2.6 Modelling unit creation in the tools; contactor example

This chapter deals with how to build the modelling unit with the example of the COM/MON contactor. It describes the GUI (Graphical User Interface).

In SimfiaNeo, modelling units are called bricks, and are created directly in the model. They can then be added to the Project Library if deemed necessary for reuse.

To create a new brick, open the Model Diagram (e.g. by double-clicking on the Model level in the Model Explorer on the left side), select the Brick tool in the Palette and click in the Diagram.

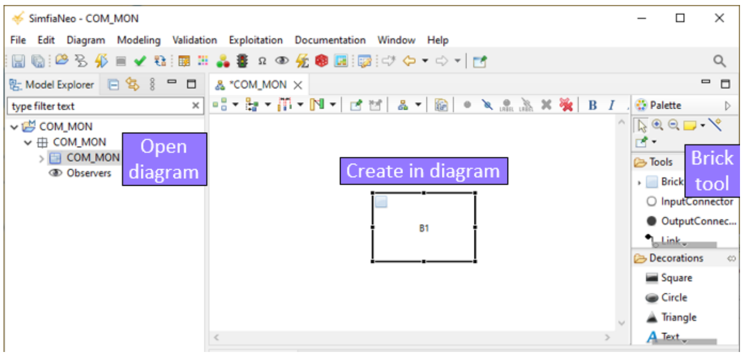


Figure 116: SimfiaNeo GUI - brick creation

Right after its creation, the new brick is automatically selected in the diagram, hence the Properties view at the bottom of the screen displays information on this brick. Modify its name to “Contactor” and its generic behavior to “Custom”.

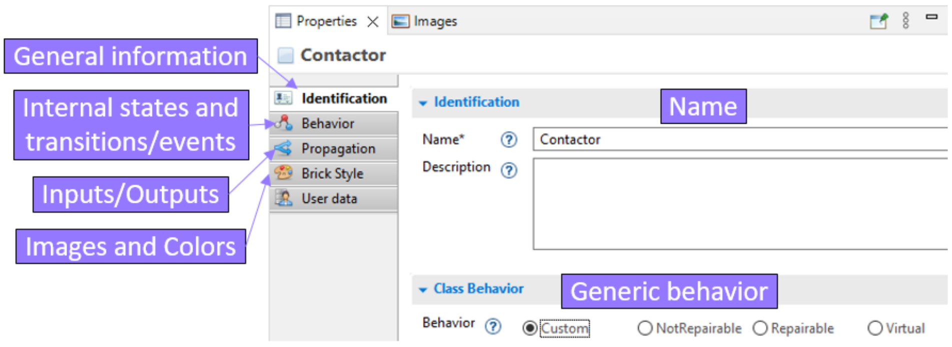
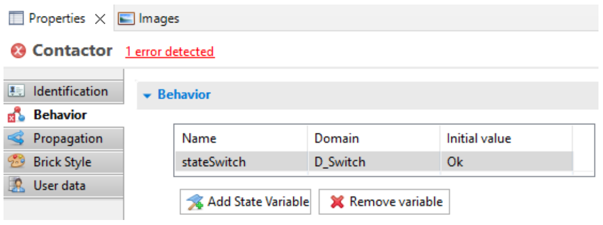


Figure 117: SimfiaNeo GUI - general bricks information

(a) Manage internal state variables

State variables are managed in the Behavior tab of the Properties view. Dedicated table and buttons enable creating/deleting variables, renaming them, switching their domain, and setting their value at initial time.



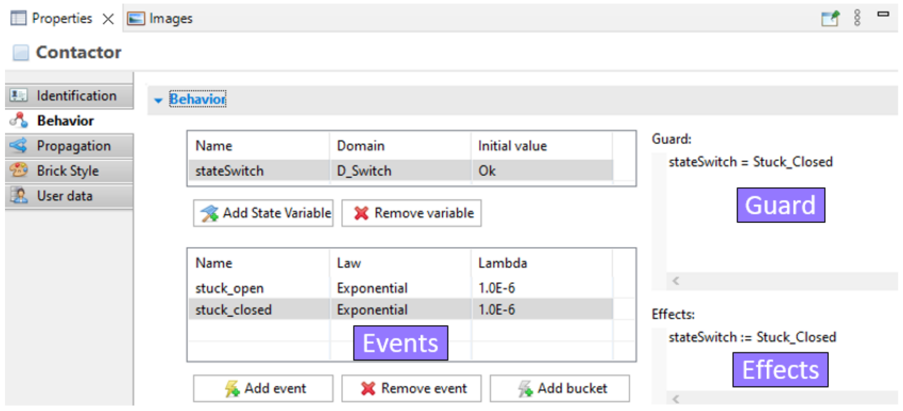
Name	Domain	Initial value
stateSwitch	D_Switch	Ok

Figure 118: SimfiaNeo GUI - state variables table

Predefined domains already exist. Domains created by the users are automatically added to the list of available domains. Switching state variables domain usually display error messages in the interface as previously defined transitions become not consistent with new domains. This is addressed in next step.

(b) Manage events and transitions

Events and transitions are managed in the Behavior tab of the Properties view. Dedicated table and buttons enable creating/deleting events, renaming them, and customizing their probability/determinist law.



Name	Domain	Initial value
stateSwitch	D_Switch	Ok

Name	Law	Lambda
stuck_open	Exponential	1.0E-6
stuck_closed	Exponential	1.0E-6

Guard:
stateSwitch = Stuck_Closed

Effects:
stateSwitch := Stuck_Closed

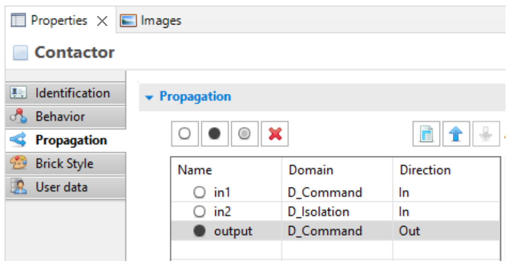
Figure 119: SimfiaNeo GUI - behavior tab

Selecting an event in the table on the left updates the contents in the fields Guard and Effects on the right. Guard field is used to input the Guard of the corresponding transition. Effects field is used to input the actions of the corresponding transition. In both fields, the shortcut Ctrl-Space enables using auto-completion feature.

Probability laws are filled directly in the Behavior tab (see above). Constants can be created in the Constants table (menu Modeling> Open constants) to have several laws share the same numerical parameters.

(c) Create the connectors and links (flows)

Prior to link creation, it is recommended to create connectors on the bricks. Connectors can be created by using the connector tools in the Palette of Diagrams, or by using the Propagation tab of the Properties view of a brick. In this tab, dedicated table and buttons enable to create or delete connectors, rename them and define their domain. White dots are input connectors while black dots are output connectors.



Name	Domain	Direction
○ in1	D_Command	In
○ in2	D_Isolation	In
● output	D_Command	Out

Figure 120: SimfiaNeo GUI - connectors table

To create the links, go in a Diagram, select the Link tool in the Palette, click on an output connector (black dot), then click on an input connector (white dot). It is also possible to directly click on bricks instead of on connectors, in which case new connectors are created.

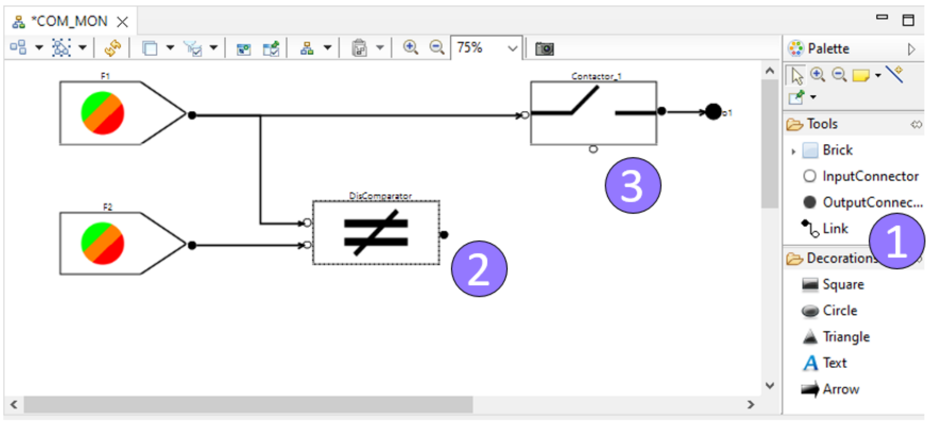
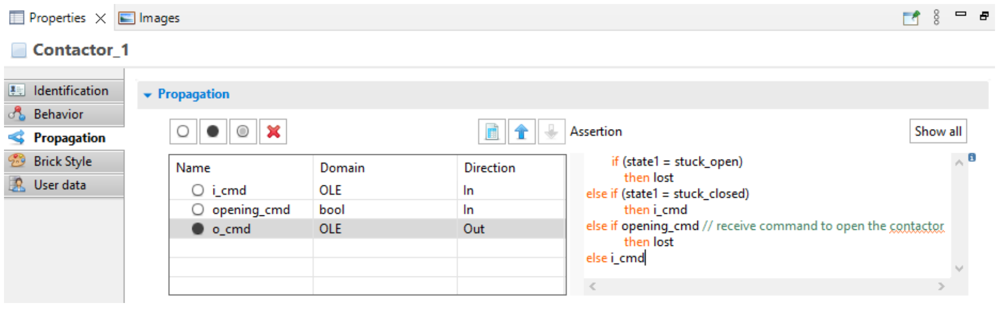


Figure 121: SimfiaNeo GUI - link creation

(d) Create an assertion

Assertions are filled in the Propagation tab of the Properties view of a brick. When selecting an output connector (black dot) in the table on the left side, the right side section is updated to display and edit the assertion.



Name	Domain	Direction
○ i_cmd	OLE	In
○ opening_cmd	bool	In
● o_cmd	OLE	Out

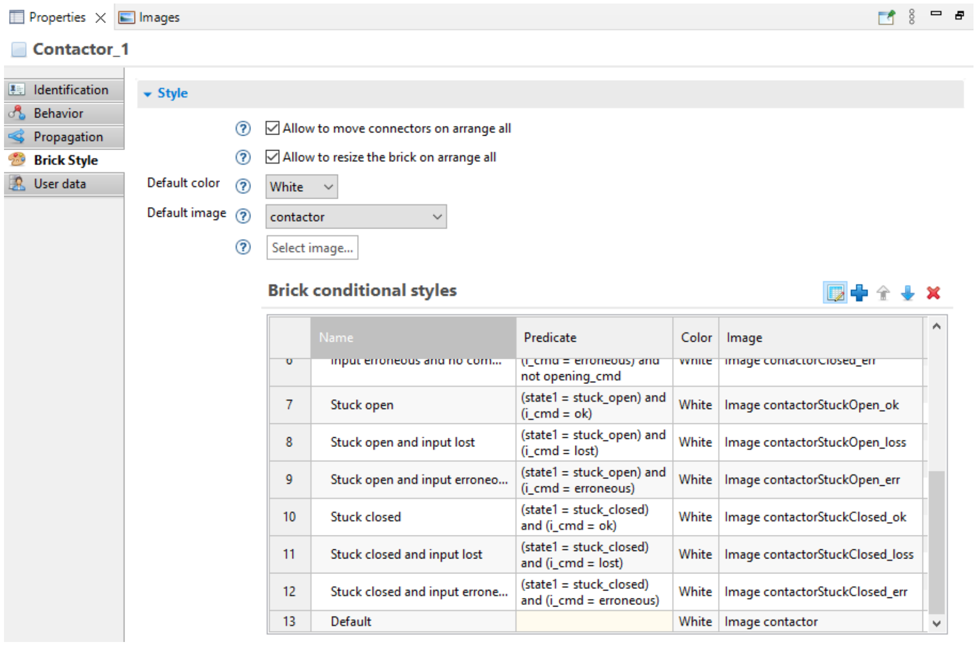
```
    if (state1 = stuck_open)
       then lost
else if (state1 = stuck_closed)
       then i_cmd
else if opening_cmd // receive command to open the contactor
       then lost
else i_cmd
```

Figure 122: SimfiaNeo GUI - assertion edition

In this field, the shortcut Ctrl-Space enables using auto-completion feature.

(e) Create the conditional style

Style of the brick is defined in the Brick Style tab of the Properties view of a brick. Default color and Default image drop-down menus enable defining style in edition mode.



Properties | Images

Contactor_1

Identification
Behavior
Propagation
Brick Style
User data

Style

☑ Allow to move connectors on arrange all
☑ Allow to resize the brick on arrange all

Default color: White
Default image: contactor
Select image...

Brick conditional styles

	Name	Predicate	Color	Image
6	Input erroneous and no com...	(i_cmd = erroneous) and not opening_cmd	White	Image contactorClosed_err
7	Stuck open	(state1 = stuck_open) and (i_cmd = ok)	White	Image contactorStuckOpen_ok
8	Stuck open and input lost	(state1 = stuck_open) and (i_cmd = lost)	White	Image contactorStuckOpen_loss
9	Stuck open and input erroneo...	(state1 = stuck_open) and (i_cmd = erroneous)	White	Image contactorStuckOpen_err
10	Stuck closed	(state1 = stuck_closed) and (i_cmd = ok)	White	Image contactorStuckClosed_ok
11	Stuck closed and input lost	(state1 = stuck_closed) and (i_cmd = lost)	White	Image contactorStuckClosed_loss
12	Stuck closed and input errone...	(state1 = stuck_closed) and (i_cmd = erroneous)	White	Image contactorStuckClosed_err
13	Default		White	Image contactor

Figure 123: SimfiaNeo GUI - style of a brick

Default conditional style for simulation is based on internal state variable. It can be customized by activating the bottom table. Predicates are user-defined Boolean formula to determine the image and/or color of the brick. In this Predicate field, the shortcut Ctrl-Space enables using auto-completion feature.

(f) Put a brick in Library

A brick that was created in a diagram can be added to the Library at any step. This is done with a right-click on a brick in a diagram and selecting Library> Store in library.

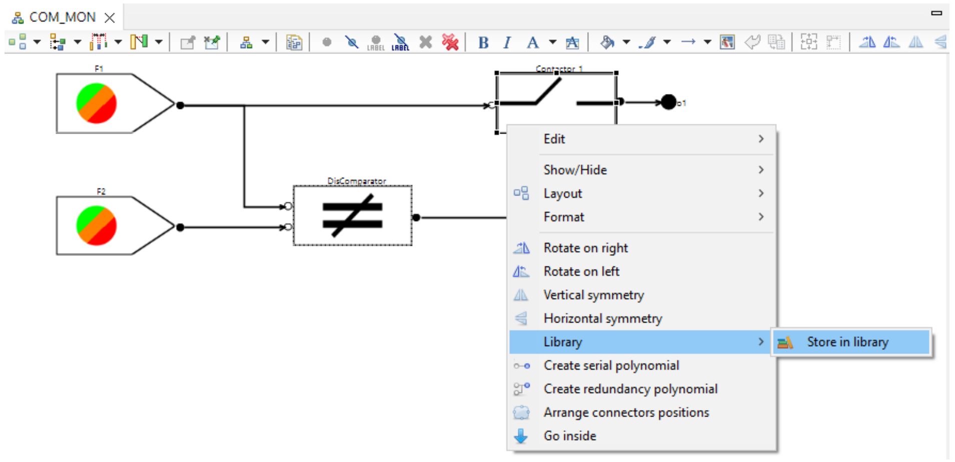


Figure 124: SimfiaNeo GUI - store brick in Library

Insert the name to use in the Library to finish. This brick is now displayed in the project Library available in the bottom-left corner.

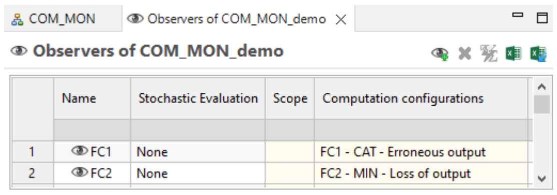
(g) Instantiate a brick

If a brick is already defined in Library, it can be instantiated in the model. This can be performed by following either of the following methods:

- Drag-and-drop the brick from the Library view (bottom-left corner) to the diagram
- Right-click in an empty space of the diagram and select Library> Instantiate existing class

(h) Create the observers (link with FC)

Observers are managed through the Observers table. Open the table by double-clicking on the Observers (eye icon) in the Model Explorer.



	Name	Stochastic Evaluation	Scope	Computation configurations
1	FC1	None		FC1 - CAT - Erroneous output
2	FC2	None		FC2 - MIN - Loss of output

Figure 125: SimfiaNeo GUI - observers table

Creation button is situated in the top-right corner of the table. Name of the observer needs to follow AltaRica variables naming rules (mainly no spaces, and no digit as the first character). Stochastic Evaluation is kept to None (this option is linked to Monte-Carlo simulation). When selecting an observer, its Boolean expression can be customized in the Properties view. This expression takes the value *true* when the feared situation is reached. In this field, the shortcut Ctrl-Space enables using auto-completion feature.

(i) Launch a step-by-step simulation

To launch the step-by-step simulation, use the menu Validation> Step by step simulation.

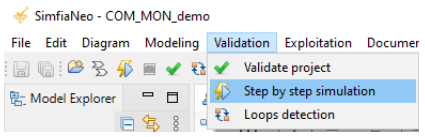


Figure 126: SimfiaNeo GUI - start step-by-step simulation

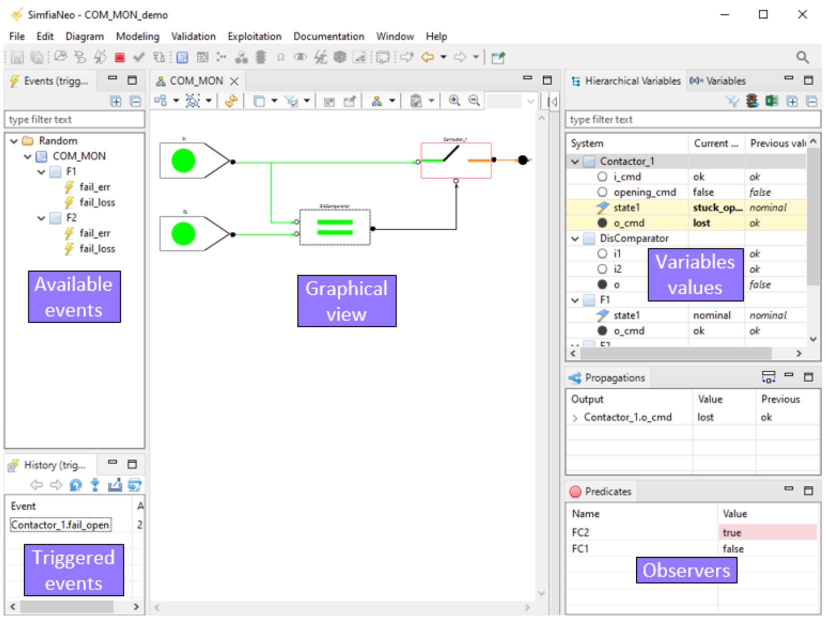


Figure 127: SimfiaNeo GUI - step-by-step simulation

Events can be triggered by double-click on the left, or with a right-click on bricks in diagrams. Simulation is exited by using the Stop simulation (red square) button.

(j) Calculate the Cut-Sets

To compute the Cut-Sets, the first step is to define the computation options. To open the corresponding table, use the menu Exploitation> Open sequences computation.

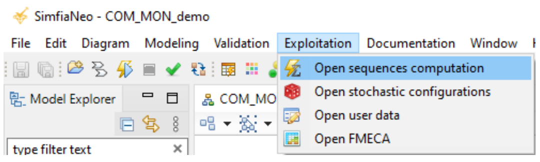
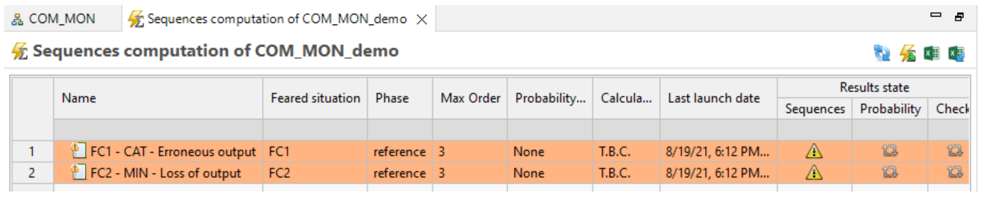

SimfiaNeo - COM_MON_demo
File Edit Diagram Modeling Validation Exploitation Documentation Window
Open sequences computation
Open stochastic configurations
Open user data
Open FMECA
Model Explorer
type filter text

Figure 128: SimfiaNeo GUI - open computation table

Creation buttons are situated in the top-right corner of the table.

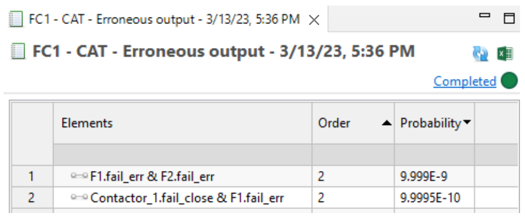


Sequences computation of COM_MON_demo

	Name	Feared situation	Phase	Max Order	Probability...	Calcula...	Last launch date	Results state		
								Sequences	Probability	Check
1	FC1 - CAT - Erroneous output	FC1	reference	3	None	T.B.C.	8/19/21, 6:12 PM...	⚠		
2	FC2 - MIN - Loss of output	FC2	reference	3	None	T.B.C.	8/19/21, 6:12 PM...	⚠		

Figure 129: SimfiaNeo GUI - computation table

After creating a computation config, go to the Options tab of the Properties view to customize the computation options. In particular, it is possible to define if you want only qualitative results or also would like the numerical probabilities. To launch the computation, right-click on the line in the table and select Execute.



FC1 - CAT - Erroneous output - 3/13/23, 5:36 PM

FC1 - CAT - Erroneous output - 3/13/23, 5:36 PM

Completed ●

	Elements	Order	Probability
1	F1.fail_err & F2.fail_err	2	9.999E-9
2	Contactor_1.fail_close & F1.fail_err	2	9.9995E-10

Figure 130: SimfiaNeo GUI - computation results

Results are stored in the project but can also be exported in Excel format.

14.3 – I – Modeling the command / monitoring pattern example in AltaRica 3.0

This part proposes to model and assess the command/monitoring pattern example (Com-Mon), introduced in part 5.2 and depicted in Figure 131, with the OpenAltaRica platform implementing the AltaRica 3.0 modeling language (the third version of AltaRica).

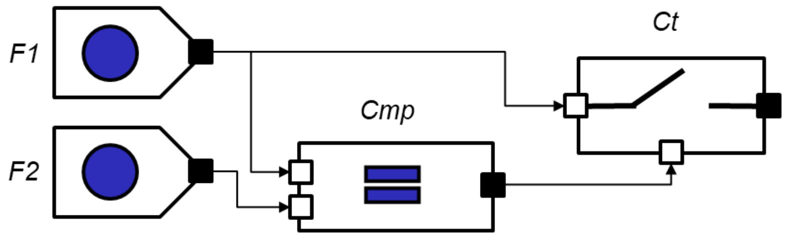


Figure 131: Com-Mon example

As previously explained, the Com-Mon is composed of two sensors F1 and F2, a comparator Cmp that checks the equality of two inputs, finally a contactor Ct that is closed as long as the equality check is true, and when it is closed, it transmits F1 output; otherwise, it transmits no output. The sensors have two failure modes: they may produce an erroneous output, or they may produce no output at all. Finally, the safety requirements of interest for this pattern are:

- Failure condition FC_B1: an erroneous output which is catastrophic.
- Failure condition FC_B2: the output loss which is minor.

14.3.1 – I - Modeling the Com-Mon

The model of the Com-Mon is divided in several parts. First, it defines a domain, depicted in Figure 132, that is used to type variables (state or flow variables). Then it defines classes representing components, depicted in Figure 133, Figure 134 and Figure 135, that are instantiated in the main block, depicted in Figure 136, corresponding to the entry point to the Com-Mon example.

```
domain FailureMode {OK, LOST, ERR}
// OK - normal behavior
// ERR - the sensor produces erroneous data
// LOST - the sensor produces no data
```

Figure 132: Definition of the domain

```
class Sensor
// definition of the state variable
FailureMode _mode (init = OK);
// definition of the output flow variable
FailureMode output (reset = LOST);
// definition of events
event failureLoss (delay = exponential(1.0E-4));
event failureErr (delay = exponential(1.0E-5));
// definition of transitions
transition
failureLoss:(_mode == OK) -> _mode := LOST;
failureErr: (_mode == OK) -> _mode := ERR;
// definition of the assertion
assertion
output := _mode;
end
```

Figure 133: Definition of the class Sensor

```
class Contactor
    // definition of flow variables
    FailureMode input, output (reset = LOST);
    Boolean closeCondition (reset = false);
    // definition of the state variable
    Boolean _open (init = false);
    // definition of the event
    event openCT (delay = Dirac(0.0)) ;
    // definition of the transition
    transition
        openCT: not _open and not closeCondition -> _open := true;
    // definition of the assertion
    assertion
        output := switch {
                        case _open : LOST
                        default : input};
end
```

Figure 134: Definition of the class Contactor

```
class Comparator
    // definition of flow variables
    FailureMode input1, input2 (reset = LOST);
    Boolean output (reset = false);
    // definition of the state variable
    Boolean _working (init = true);
    // definition of the event
    event failure (delay = exponential(1.0e-5));
    // definition of the transition
    transition
        failure: _working -> _working := false;
    // definition of the assertion
    assertion
        output := if _working then (input1 == input2) else true;
end
```

Figure 135: Definition of the class Comparator

```
block ComMon
    // components of the Com-Mon
    Sensor F1, F2;
    Comparator Cmp;
    Contactor Ct;
    // definition of connections between components
    assertion
        Ct.input := F1.output;
        Ct.closeCondition := Cmp.output;
        Cmp.input1 := F1.output;
        Cmp.input2 := F2.output;
    // definition of failure conditions
    observer Boolean FC_B1 = (Ct.output == ERR);
    observer Boolean FC_B2 = (Ct.output == LOST);
end
```

Figure 136: Definition of the main block ComMon

14.3.2 – I - Assess the Com-Mon

Within the OpenAltaRica platform, this AltaRica 3.0 model of the Com-Mon example is assessed by using the generator of critical sequences. There are two parts to realize this assessment: a first one compiling the mode, and a second one realizing the generation of the critical sequences.

(a) Compilation of the model

The compilation of the model produces only one main block with only atomic elements inside: parameters, state and flow variables, events, transitions and the assertion, named a GTS (Guarded Transition System, see [REF A] for more explanations in the AltaRica 3.0 jargon. More precisely the compiler first instantiates all classes, then it flats the hierarchy, and finally it produces the GTS model. Some checks and optimizations are also realized during this process. The compiled model, meaning the GTS model, of the AltaRica 3.0 model of the Com-Mon example is depicted Figure 137.

```
domain FailureMode {OK, LOST, ERR}


block ComMon
  Boolean Cmp._working (init = true);
  FailureMode Cmp.input1 (reset = LOST);
  FailureMode Cmp.input2 (reset = LOST);
  Boolean Cmp.output (reset = false);
  Boolean Ct._open (init = false);
  Boolean Ct.closeCondition (reset = false);
  FailureMode Ct.input (reset = LOST);
  FailureMode Ct.output (reset = LOST);
```

```
    FailureMode F1._mode (init = OK);
    FailureMode F1.output (reset = LOST);
    FailureMode F2._mode (init = OK);
    FailureMode F2.output (reset = LOST);
    event Cmp.failure (delay = exponential(1e-05));
    event Ct.openCT (delay = Dirac(0.0));
    event F1.failureErr (delay = exponential(1e-05));
    event F1.failureLoss (delay = exponential(0.0001));
    event F2.failureErr (delay = exponential(1e-05));
    event F2.failureLoss (delay = exponential(0.0001));
    observer Boolean FC_B1 = Ct.output == ERR;
    observer Boolean FC_B2 = Ct.output == LOST;
    transition
      Cmp.failure: Cmp._working -> Cmp._working := false;
      Ct.openCT: not Ct._open and not Ct.closeCondition
                -> Ct._open := true;
      F1.failureLoss: F1._mode == OK -> F1._mode := LOST;
      F1.failureErr: F1._mode == OK -> F1._mode := ERR;
      F2.failureLoss: F2._mode == OK -> F2._mode := LOST;
      F2.failureErr: F2._mode == OK -> F2._mode := ERR;
    assertion
      Cmp.output :=  if Cmp._working then (Cmp.input1 == Cmp.input2)
                     else true;
      Ct.output :=  if Ct._open then LOST else Ct.input;
      F1.output := F1._mode;
      F2.output := F2._mode;
      Ct.input := F1.output;
      Ct.closeCondition := Cmp.output;
      Cmp.input1 := F1.output;
      Cmp.input2 := F2.output;
  end
```

Figure 137: GTS model of the Com-Mon

(b) Generation of critical sequences

Two computations are realized: a first one to get all sequences of events leading to the value 'true' of the observer 'FC_B1', depicted Figure 138, and a second one to get all sequences of events leading to the value 'true' of the observer 'FC_B2', depicted Figure 139.

```
Cmp.failure F1.failureErr
Cmp.failure F2.failureLoss F1.failureErr
Cmp.failure F2.failureErr F1.failureErr
```

Figure 138: Critical sequences for the observer 'FC_B1'

```
Cmp.failure F1.failureLoss
Cmp.failure F2.failureLoss F1.failureLoss
Cmp.failure F2.failureErr F1.failureLoss
F1.failureLoss Ct.openCT
F1.failureErr Ct.openCT
F2.failureLoss Ct.openCT
F2.failureErr Ct.openCT
```

Figure 139: Critical sequences for the observer 'FC_B2'

14.4 – B- The ONERA library

In order to ease the reader understanding, we use the generic ONERA library for all the presented simple examples in this document. This section aims to provide information about the modelling unit of the library we use, in particular their icons meaning.

The ONERA library is available here: https://forge.onera.fr/projects/mbsa

14.4.1 Quick introduction

The ONERA library contains two families of nodes

- "Blocks" model nodes which may fail
- "logical blocks" model nodes which represent logical functions or connectors of flows.

Blocks and operators aim at

- Producing a value
- Observing / testing a value
- Selecting a value.

Three types of values can be handled

- Type Bool : {true, false}.
- Type OLE : {ok, lost err}
- Type NOH : {null, ok, high}

14.4.2 Main icons description

Icons are associated to model modelling units to ease the understanding of the model during simulation

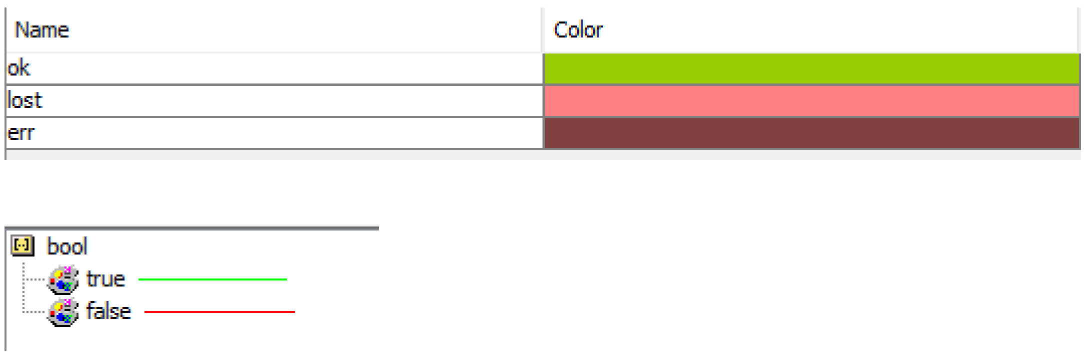
Figure 140 describes the general philosophy of the colour chart for the icons used in this document (from the ONERA library).

Description	Example	Meaning
Green	🟩	(State=ok) and (Input=ok)
Dark red	🟥	(State =ok) and (Input =lost)
Light red	▭	(State =ok) and (Input =err)
Crossed and dark red	☒	State =lost
Crossed and light red	☒	State =err

Figure 140 Information about ONERA icons colour chart

14.4.3 Flows colours

In simulation mode, the colour chart of the flows used in this document is the described Figure 141:



Name	Color
ok	🟩
lost	🟥
err	🟫

bool
- true —
- false —

Figure 141 Information about ONERA types colour chart

14.5 The document examples description

This section aims to describe the examples used to support the document.

The COM/MON pattern example that is used to support the present document

Refer to chapter 5.2 for description of the example and to 14.1 and 14.2 for modeling in the tools.

End of document