“D.2.2.1 - Specification of Modelica extensions and interfaces for Bayesian networks and Fault trees”

“WP2.2 - Safety analysis methods: Bayesian networks, Fault trees and hybrid stochastic models”

“WP2 - Properties modelling and Safety”

MODRIO (11004)

Version	3.0 (final deliverable)
Date	13/04/2016

Author

Marc BOUISSOU EDF R&D

Reviewer

Lena BUFFONI LIU

Executive summary

This report proposes a strategy to implement tools that could be used like plugins in Modelica tools, in order to allow the automatic generation of fault-trees or Bayesian networks from system models. Prototypes based on Dymola and OpenModelica are presented, through their use in two use cases.

The automatic generation of static reliability models such as fault trees or Bayesian networks cannot be done directly from a simulation model such as a Modelica model. This is due to the fact that Modelica models and static reliability models have completely different objectives: Modelica was designed to model deterministic, dynamic models, whereas reliability models aim at representing the probability distribution on the (possibly huge) set of all possible components malfunctions and failures combinations, resulting in a system failure. If a general, reusable form of reliability models is needed, then a domain specific language dedicated to reliability is required, just like Modelica was a good way to make physical models based on differential equations reusable.

This report shows how two existing domain specific languages, namely Figaro (developed in 1990 by EDF) and O3prm (developed in 2010, in the context of a French collaborative project on object oriented extensions of Bayesian networks) can be associated to Modelica models in order to produce automatically:

- Fault trees;
- Dynamic reliability models with discrete states (in Figaro language);
- Probabilistic Relational Models, a generalization of Bayesian networks (in O3prm language); this kind of model is suitable for performing diagnosis, with an optimal "guessing" of the failed components thanks to the exploitation of probabilistic data.

Thus, the approach described in this document, instead of proposing extensions to Modelica itself, relies only on the *binding* of a standard Modelica model to a Figaro or O3prm library containing the information specific to reliability studies. This has several advantages, going far beyond the mere saving of development efforts:

- Modelica models can be left unchanged: there is no risk of introducing bugs in validated models by trying to extend them towards dependability applications;
- The structure of the Figaro or O3prm model may be quite different from the structure of the Modelica model. This is due to the fact that different kinds of abstraction are used when passing from a real system to a functional, simulation model on the one hand, and to a dysfunctional model on the other hand;
- Different Figaro or O3prm libraries can be bound to a single Modelica model; this gives the possibility to realize various kinds of studies, all from a single initial model.

The principles given in this report are illustrated on two very different use cases: a meshed telecom network and a real thermohydraulic system of a nuclear power plant. Section 6 gives an overview of the two existing prototype tools, based on Dymola and OpenModelica.

The last section presents the O3prm editor tool. It also explains the principles of transformation of a Modelica model into an O3prm model, suitable for diagnosis.

Acknowledgements

Writing this report would have been impossible without the work of the following persons, who contributed to the prototypes implementations:

- Claire Campan, who developed the function Modelica2Figaro of the Dymola prototype
- Ahmed Bouzakaria, who worked on the Dymola prototype
- Alexander Carlqvist and Lena Buffoni, who developed the OpenModelica prototype
- Lionel Torti, who developed the O3prm editor tool.

Table of contents

1. Introduction

This report proposes a strategy to implement tools that could be used like plugins in Modelica tools, in order to allow the automatic generation of fault-trees or Bayesian networks from system models. Prototypes based on Dymola and OpenModelica are presented, through their use in two use cases.

The main idea of this strategy is to reuse as much as possible the KB3 workbench set of tools, that EDF has been developing continuously since 1990 in order to automate dependability studies of complex systems.

The automatic generation of static reliability models such as fault trees or Bayesian networks cannot be done directly from a simulation model such as a Modelica model. This is due to the fact that Modelica models and static reliability models have completely different objectives: Modelica was designed to model deterministic, dynamic models, whereas reliability models aim at representing the probability distribution on the (possibly huge) set of all possible components malfunctions and failure combinations, resulting in a system failure. If a general, reusable form of the reliability models is needed, then a domain specific language dedicated to reliability is required, just like Modelica was a good way to make physical models based on differential equations reusable.

EDF has a lot of experience in reliability modeling languages. EDF designed the Figaro modeling language in 1990. This language is the basis of KB3 which is the reference tool used for building fault trees and dynamic models for probabilistic safety analyses of nuclear power plants and all other reliability analyses at EDF.

Although powerful and general enough to be able to replace all classical reliability models, like fault trees, event trees, reliability block diagrams, Petri nets etc., the Figaro language is not well suited to replace Bayesian networks. Another domain specific language is needed for that.

EDF has participated in the SKOOB (Structuring Knowledge with Object Oriented Bayesian networks) collaborative project. In this project, a language (called O3prm) for describing Bayesian models in an object oriented paradigm, and some tools to process O3prm models have been developed. Although these tools are far less mature than those of the KB3 workbench, we think that O3prm (or something close to it) is the long term solution for linking Bayesian tools to Modelica models.

So the approach described in this document, instead of proposing extensions to Modelica itself, relies only on the *binding* of a standard Modelica model to a Figaro or O3prm library containing the information specific to reliability studies. This has several advantages, going far beyond the mere saving of development efforts, as it will be explained in Section 2.

This report is organized as follows: sections 2 to 4 give the principles of the binding and how they can solve various problems, pertaining to modeling issues as well as system engineering workflow and organization. Section 5 demonstrates these principles on two very different use cases: a meshed telecom network and a real thermohydraulic system of a nuclear power plant. Section 6 gives an overview of the two existing prototype tools, based on Dymola and OpenModelica. The last section is more futuristic: it explains how the same kind of approach could be used to pass from a Modelica model to an O3prm model, suitable for diagnosis. For the time being, O3rpm tools are not mature enough to enable an implementation of those ideas.

2. Objectives of this work

The benefits expected from the possibility to generate static[1] reliability models from Modelica design models are:

- Reducing the gap between the system design team and reliability analysts and bringing together the two communities, resulting in increased efficiency;
- **Reusing models built for system design in dependability analysis. This will save manpower, and, more importantly, ensure a perfect consistency between design**

[1] Of course, the listed benefits are also true for dynamic reliability models, but this report is dedicated to static models.

models and those used in dependability analysis;
- Reducing development and deployment costs by reusing the numerous tools already available to process Modelica models. In addition to tools, there are also large model databases, already available for system design;
- Expanding the M&S (modeling and simulation) market to the domain of system dependability analysis, thus giving new market opportunities to the M&S technology vendors.

A long term objective could be to obtain a convergence between the design tools based on Modelica and the tools such as KB3 used by reliability analysts. However, in order to be realistic, this objective would imply also a fusion of the communities of designers and reliability analysts. Only persons having both competences will be able to withstand the inevitable additional work due to the use of a multi-domain tool to perform analyses dedicated only to design or to reliability.

Further to the "social" issue described in the previous paragraph, a major impediment to this convergence exists: it is the fact that the structure (in terms of breakdown into components, hierarchy of classes, and topology of systems) of models dedicated to simulation may be too far from the structure needed for reliability analyses.

Because of the two issues described above (social, technical) we think that the best solution is to associate, **in a loose manner**, a Modelica model to:

- A Figaro model in order to build fault trees or even perform some discrete simulation studies,
- An O3prm model in order to do Bayesian inference, mainly for diagnosis purposes.

This has at least two advantages:

- The Figaro (or O3prm) tools and Modelica tools remain independent. Their maintenances and evolutions will simply go on, without any requirement for an organization change.
- It will be possible to use, without adaptation, existing Modelica simulation models built by system designers and existing Figaro (or O3prm) libraries built by reliability analysts. This will be well demonstrated on the second use case (the SRI thermohydraulic system, see section 5.2), a real system for which the Modelica model and the reliability library in Figaro were built by persons who never met!

The prototype tools described in this report, that were developed with minimal effort, will allow various experiments and, if it seems worth having such tools, they will help refine the GUI with some specific features in order to facilitate the joint use of Modelica and Figaro or O3prm.

In the present report, a detailed description is given for the case of the Figaro language, including two working prototypes based on Dymola and OpenModelica. For the O3prm language, a tool called O3prm editor has been developed during the MODRIO project, that offers a user friendly interface for the development of O3prm models and calculations based on these models. The “bridge” between Modelica and O3prm models is not yet operational, but we describe in section 7 how we intend to proceed.

Figure 1 shows how this work fits in a "global picture" including various ways to pass from a Modelica design model to dependability analyses. The box containing "Enrich the model… MCS" refers to the approach described in [1]. The first test "Hybrid?" in this flowchart means: "are there **inescapable** hybrid features of the modeled system?", in particular the fact that the failure rates of some components depend on continuous variables like the temperature or the pressure of some fluid. More generally, if there are bi-directional influences between discrete and continuous variables, it is most of the time impossible to obtain correct results with a purely discrete model like a Figaro model.

When it makes sense to create a Boolean model, an alternative to the use of a Figaro library is the method described in details in another report of workpackage 2.2: [2]. This approach is very interesting from a theoretical point of view because it relies only on existing features of the Modelica language and compilers. However, it is still very new and thus cannot benefit in a near future from all the expertise put by reliability analysts since 1990 in the Figaro language, libraries and tools.

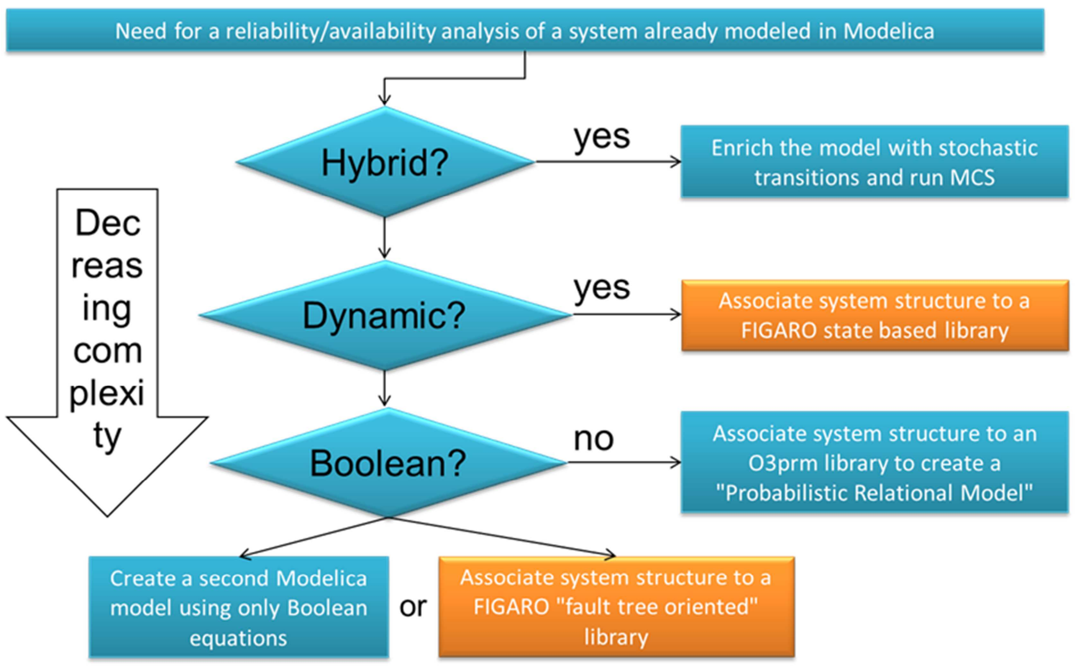

Need for a reliability/availability analysis of a system already modeled in Modelica
Hybrid?
yes
Enrich the model with stochastic transitions and run MCS
Dynamic?
yes
Associate system structure to a FIGARO state based library
Boolean?
no
Associate system structure to an O3prm library to create a "Probabilistic Relational Model"
Create a second Modelica model using only Boolean equations
or
Associate system structure to a FIGARO "fault tree oriented" library
Decreasing complexity

Figure 1. From Modelica to probabilistic models

3. The current situation

Before explaining how we propose to link Modelica to Figaro, let us begin by describing how designers and reliability analysts usually work.

Both need to start from some "real world" system. Usually, design studies are performed first. There may even be a long delay between the design studies (resulting in a precise description of the system) and the beginning of reliability studies. This is the current situation, but the MODRIO project will propose a different system engineering approach, that puts reliability studies (at least preliminary studies) much earlier.

Anyway, there are strong similarities between the workflows of design and reliability studies, as one can see on Figure 2. In both cases, the system model is built as the assembly of classes taken from an appropriate library and objects corresponding to the elements of the system.

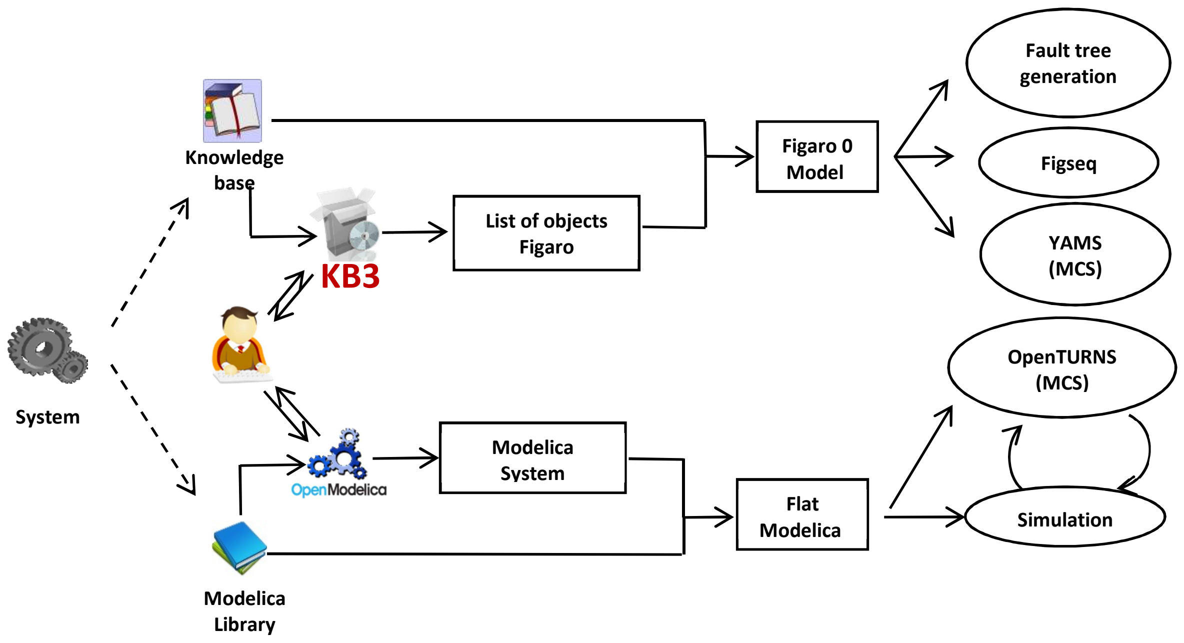


Figure 2. Figaro and Modelica workflows

In order to model a system with KB3, the reliability analyst graphically inputs the system layout using the components described in a Figaro library (more often called knowledge base by Figaro users). KB3 then "compiles" the system objects and relevant classes in a low level sub-language of Figaro called Figaro 0. This model is the input for automated dependability studies of the system, namely:

- Fault tree analysis (for static models),
- Markov analysis by Figseq (for markovian dynamic models),
- Monte Carlo simulation by YAMS (for non-markovian dynamic models).

Independently, OpenModelica (or another Modelica tool) allows modeling and simulating the same physical system using a Modelica library and a similar workflow.

The flat Modelica plays the same role of low level, easy to process model as Figaro 0 in the KB3 workbench; it can be used for a single deterministic simulation or for a set of simulations managed by the tool OpenTURNS for an uncertainty propagation study.

Unfortunately, as shown in Figure 2, there is no relationship between the two tools and some information common to the two workflows is not shared, but duplicated.

4. Bridging the gap between Modelica and Figaro tools

To benefit from the power of those two approaches from a single tool, the integration of some Figaro in Modelica (or vice-versa) is necessary. Since design studies come first, and given the existence of a large community linked to Modelica, we take the first approach. To be able to generate Figaro 0 models from a Modelica tool, all the parameters and information needed by Figaro and that do not exist in Modelica must be integrated in Modelica in such a way that they do not affect in any way the original simulation model (without Figaro). That is why they must be added as annotations or as string parameters. We decided to use parameters, because they can be inherited, and this is very important in our approach[2]. Moreover, parameters are part of the core of the Modelica language, which means that it will be possible to use the same models with *all* Modelica tools.

[2] Recent changes have led to annotations that can be inherited. The choice between parameters and annotations is simply a matter of implementation, and it can be changed later.

4.1. Adding Figaro information in a Modelica model

The Figaro library can be kept apart. It will still be managed with usual Figaro tools. The only information that needs to be added to the Modelica model is:

- Topology of the system (the topology for the Figaro model is not necessarily exactly the same as the topology of the Modelica model);
- Reliability parameters like failure and repair rates.

4.2. Correspondence between Modelica and Figaro concepts

Various concepts and notions in Figaro and their equivalent in Modelica are shown in the following table. This table is far from being exhaustive; it contains only the main concepts, and what is required to establish a correspondence between a Modelica and a Figaro model:

Figaro	Modelica	Comment
Class	Class	In Modelica it can be a class, package, model, record, block, or connector.
-	Connection	Links can be considered in Figaro as objects whereas Modelica considers them just as connections (without parameters) that can transmit information in both directions. See §4.4.
Constant	Parameter	
Object	Model	Every component created from a class is an object (instantiation)
Interaction rule	-	Modelica does not support any creation of rules for components.
Occurrence rule	-	
Interface	-	This notion in Figaro has no equivalent in Modelica. Indeed, the way connections are established between components is quite different in Modelica and Figaro: see §4.4.

4.3. Methodology

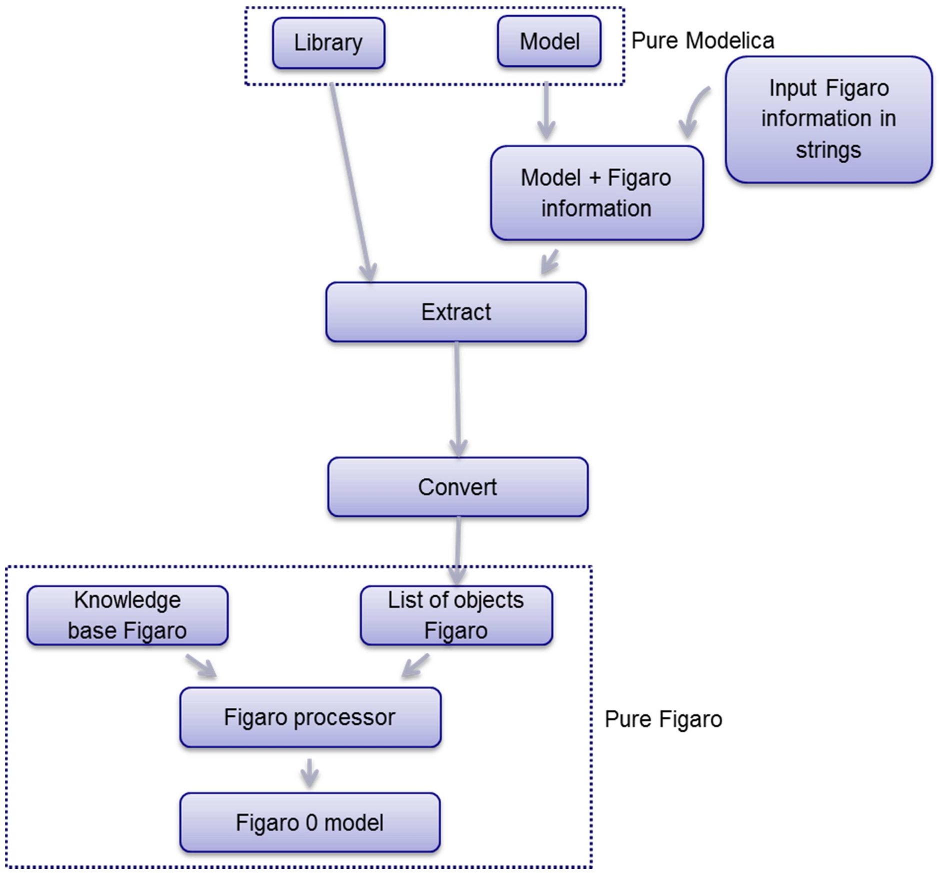


Figure 3. From Modelica to Figaro

During (or after) the creation of the Modelica model, Figaro code is added in strings, in certain components composing the model. The "augmented" model can then be used either for the usual deterministic simulation, or for dependability studies based on a Figaro model. To obtain the Figaro 0 model, Figaro additions must at first be extracted from the Modelica model and gathered into a single file containing all the Figaro objects. The mapping between the objects of the Modelica and Figaro models can be adapted to each kind of system (examples will illustrate this statement). Then the Figaro processor checks the consistency of this file in itself and with regard to the Figaro knowledge base, and creates the Figaro 0 model.

At that point, all Figaro processing tools, including, but not limited to, the fault tree generator, can be used.

In order to simplify the introduction of the Figaro strings, a simple adaptation of the Modelica library is needed. Each class C of the Modelica library that will contain Figaro strings must be declared to extend one of the two classes Figaro_Object or Figaro_Object_connector, contained in a package called FIGARO:

```
package FIGARO
  "This package contains all what is needed to manage Figaro extensions to Modelica models"
  model Figaro_Object
    "Generic structure of objects that will contain Figaro code"
    parameter String fullClassName
      "Name of the class the object will belong to in the Figaro library";
    parameter String codeInstanceFigaro
      "Figaro code specific to the current instance";
    a;
  end Figaro_Object;

  connector Figaro_Object_connector
    "Generic structure of connectors that will contain Figaro code"
    // the added parameters are the same as for Figaro_Object
    parameter String fullClassName
      "Name of the class the connector will belong to in the Figaro library";
    parameter String codeInstanceFigaro
      "Figaro code specific to the current instance";
    a;
  end Figaro_Object_connector;
  a;
end FIGARO;
```

The code is the same in these two classes, but it is necessary to duplicate it because in Modelica a connector can only inherit from a connector.

This package FIGARO must be loaded systematically when using Figaro extensions, whatever the tool used.

4.4. The problem of links with behavior

The connections between components are defined in quite different ways in Modelica and Figaro. This section explains how this problem can be solved.

4.4.1. Connections in Modelica

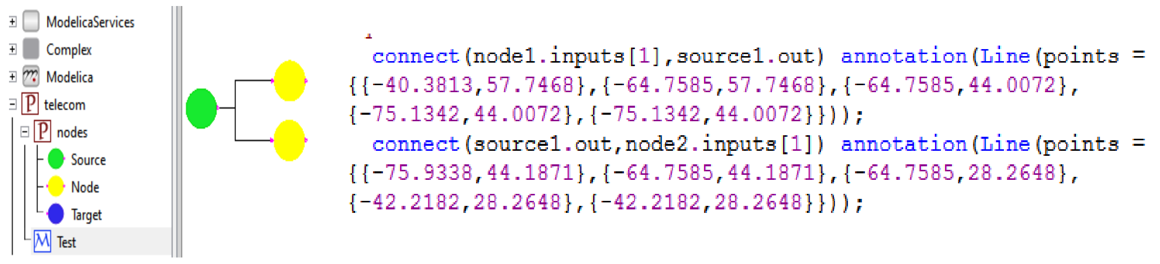
Graphical connections between objects are represented in Modelica as connect-equations in the equation part of a class, with the following syntax:

connect(variable_reference, variable_reference);

Modelica connections do not have any parameters. They can have only two predefined behaviors, depending on the type of variables they refer to:

- For variables that are not of the `flow` type, a connection just represents equality between two variables belonging to two different objects like $i_1=i_2$ or $Q_1=Q_2$ …etc.
- For `flow` variables, a connection states that the sum of the variables appearing in the connection is equal to zero (this is needed to model, for example, one of the Kirchoff's laws for electrical circuits).

In the example below, inputs[1] is the first element of a vector of variables of the same type, inputs.



```
  connect(node1.inputs[1],source1.out) annotation(Line(points =
{{-40.3813,57.7468},{-64.7585,57.7468},{-64.7585,44.0072},
{-75.1342,44.0072},{-75.1342,44.0072}}));
  connect(source1.out,node2.inputs[1]) annotation(Line(points =
{{-75.9338,44.1871},{-64.7585,44.1871},{-64.7585,28.2648},
{-42.2182,28.2648},{-42.2182,28.2648}}));
```

Figure 4. Connections in Modelica

4.4.1. Connections in Figaro

When a graphical link exists between two objects in a Figaro model, this can be interpreted in various ways. It all depends on the correspondence between the fact of drawing a link with the graphical interface of KB3 and the filling of "Interfaces" of Figaro objects.

In Figaro, an Interface of an object is a **set of objects** belonging to the same model.

For example, in a thermohydraulic system, a component can have an interface called "upstream". In the description of a particular system, the topology of the system will be given by the list of objects contained in the upstream interface of all components. In such an example, drawing a link from valve V to pump P will add V to the upstream interface of P.

Another example is given with the Telecom network Figaro knowledge base (see annex). Here, the links are not purely graphical and dedicated to the filling of interfaces of nodes. They are Figaro objects containing a behavior. For example, if we look at the description of the class "mono_dir_link":

```
CLASS mono_dir_link KIND_OF link ;
 INTERFACE
       start KIND node CARDINAL 1 EDITION NOT MODIFIABLE;
       end KIND node CARDINAL 1 EDITION NOT MODIFIABLE;
 INTERACTION
       rule1
       IF WORKING AND connected(start) AND WORKING(end)
       THEN connected(end) ;
```

We see that it contains an interaction rule and two interfaces: start and end.

The fact of drawing a graphical link from node N1 to node N2 will then put N1 in the interface start and N2 in the interface end of the link itself. Then the instantiation of the rule rule1 in Figaro 0 will give the following rule:

```
       IF WORKING AND connected(N1) AND WORKING(N2)
       THEN connected(N2) ;
```

Generally speaking, in Figaro, connections can be either purely graphical links, or complete objects with exactly the same kind of characteristics as those represented with icons. They have their own parameters and behavior such a failure rate, a repair rate and they can break down …etc. A given model can have many different types of connections each one with its own behavior and parameters. The telecom network example in section 5.1 perfectly illustrates this notion.

4.4.2. Problem and solution

Assuming that the Modelica and the Figaro model can have globally the same structure, most components in Figaro have their equivalent in Modelica except links which are not purely graphical connections.

To overcome this problem, the first idea seems to be to add particular annotations associated to the "connect" assumptions in Modelica. This is not viable because connections in Modelica cannot be distinguished whereas in Figaro there may be several kinds of connections with different characteristics. Therefore, the Figaro code should be written specifically for each connection, and could not inherit from default values associated to a particular kind of link. This would make the input of Figaro code cumbersome and error prone.

This is why the retained solution consists in creating a class for each kind of (Figaro) link in the Modelica library. Although this seems a very awkward solution, since it replaces a single Figaro element (the link) by three Modelica elements (the link + two connect statements), it will not be too shocking for Modelica practitioners, because this is what they already do in many situations (like for thermohydraulic systems modeled with the Thermosyspro library). Moreover, the explicit representation of connections in the Modelica model will allow simulating their failure modes.

5. Illustration on two use cases

The first example will be described in a very detailed manner, in order to show precisely all the implications of having a Modelica model which can be simulated as usual with Modelica tools, and which can also be associated to a Figaro model. This exercise may seem a bit artificial in this particular case, but on the other hand, it would not be possible to give all the details for the real use case SRI described in section 5.2 because involved libraries and models are much too large to be given *in extenso* in this report.

5.1. Connectivity in a telecom network

In a telecommunication network such as the one represented in Figure 5, the classical so-called S-T connectivity problem consists in calculating the probability that a given target (a node in blue) is connected to at least one source of information (a node in green).

Here, we make the simplest possible assumptions on the failure and repair processes of components: failure and repair times are all exponentially distributed, and components are all independent. Nodes and links can both fail.

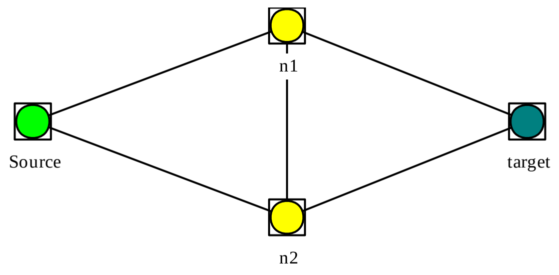


Figure 5. Telecommunication network

Despite the apparent simplicity of this example, it poses real challenges for generating fault trees: there may be loops in the topology of the system, which, without a proper treatment will lead to invalid fault trees (containing loops). Moreover, the number of links connected to a node is not known in advance and thus the knowledge base must be written in a way which is independent from the number of connections. These two problems are solved by the tools based on Figaro.

5.1.1. Description in pure Figaro

Telecom_basic is a very simple knowledge base dedicated precisely to this problem. This knowledge base is given *in extenso* in appendix A.1.

5.1.1.1. Knowledge base use

Models built with the Telecom_basic knowledge base can be used in KB3 both in simulation and to build automatically fault-trees. Of course, in a meshed network, if fault trees are built without precaution, they will contain loops, which is not acceptable. This is why it is necessary to choose the option “delete loops” of the KB3 fault tree generator to obtain valid fault trees.

This option produces large fault trees, even on small systems because it is equivalent to building all non looped paths going from a source to the considered target. It can then be untractable on large networks. This is why it is important to have alternative evaluation methods, like Monte Carlo simulation.

5.1.1.2. Hierarchy of classes

Here is the hierarchy of all the classes described in the Telecom_basic knowledge base.

```
node
   source
```

```
    target
link
  mono_dir_link
  bi_dir_link
```

The only difference between sources, targets, and regular nodes is the value of the constant “function”, which can take one of the three values 'source' 'target' 'intermediate'.

The behavior of nodes depends on the value of this constant.

5.1.1.3. Principles of the knowledge base

Nodes have a failure mode called “fail” and links have a failure mode called “interruption”.

These failure modes are inherited by the sub classes of these classes.

They are described in two different ways:

- One for simulation via occurrence rules of the group “simu_group”;
- One as basic events of fault trees, introduced via the keyword RELIABILITY_DATA; they are associated to the FT_group group of model elements.

Depending on the processing chosen by the user, it is one or the other of these representations that will be used, thanks to the selection of model elements via groups chosen when a processing is launched.

Interaction rules propagate the signal along the links in good state. They contain simple examples of Figaro language expressions including quantifiers.

5.1.1.4. Remarkable features of this knowledge base

Here is a summary of the features that make that knowledge base interesting for building test cases:

- Ability to model looped systems;
- Ability to have nodes with an arbitrary number of links attached to them;
- The fact of having two different processing modes.

5.1.2. Description in Modelica with parameters containing the Figaro code

A possible analog model of such a network in Modelica is an electrical circuit where links are represented by resistors and source nodes by generators; the other nodes are represented by pins. To protect the different components, three limiting resistors of 1 Ohm are added. They do not have Figaro string parameters; hence they disappear completely in the operations passing from the Modelica + Figaro strings model to the pure Figaro model. This shows a first example of management of structural differences between the Modelica and the Figaro model.

The model code is given in the two following sections. It relies on components of the Modelica.Electrical.Analog library.

A normal behavior of a link is represented by a resistor with a value of 10 Ohms and a failure is represented with a high value (10000 Ohms).

It is also possible to simulate a failure of a node by setting the corresponding limiting resistor to 0.

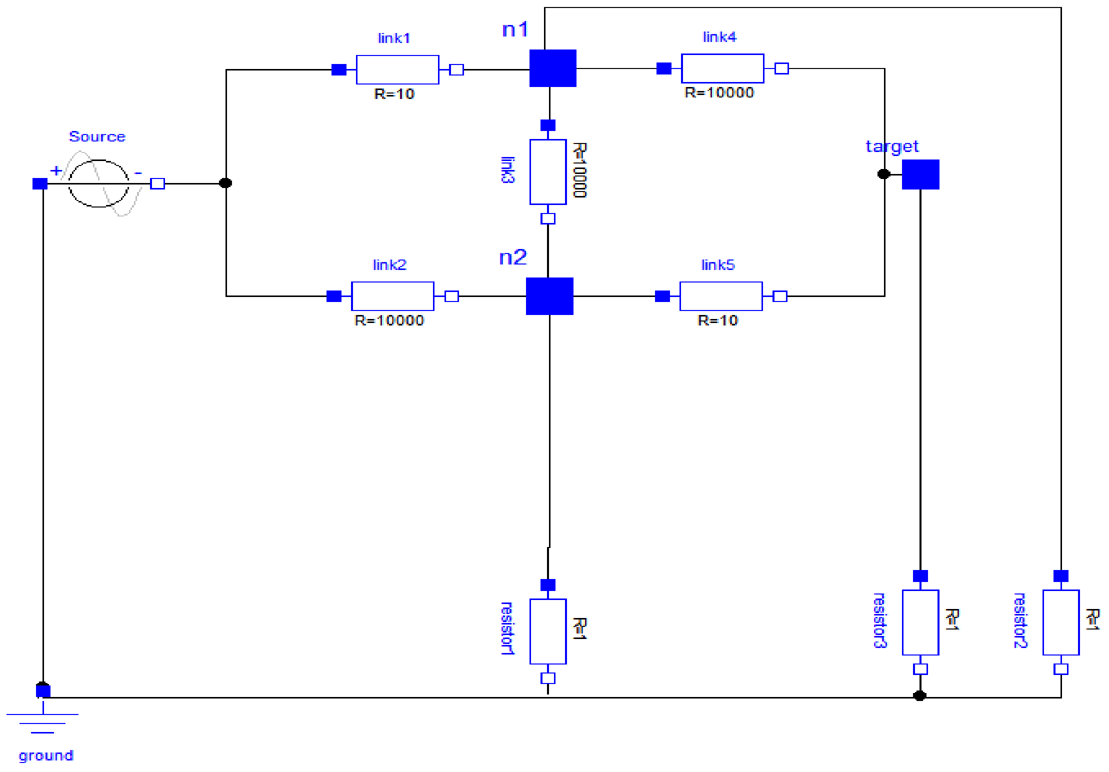


Figure 6. Telecommunication network analog model, in Modelica

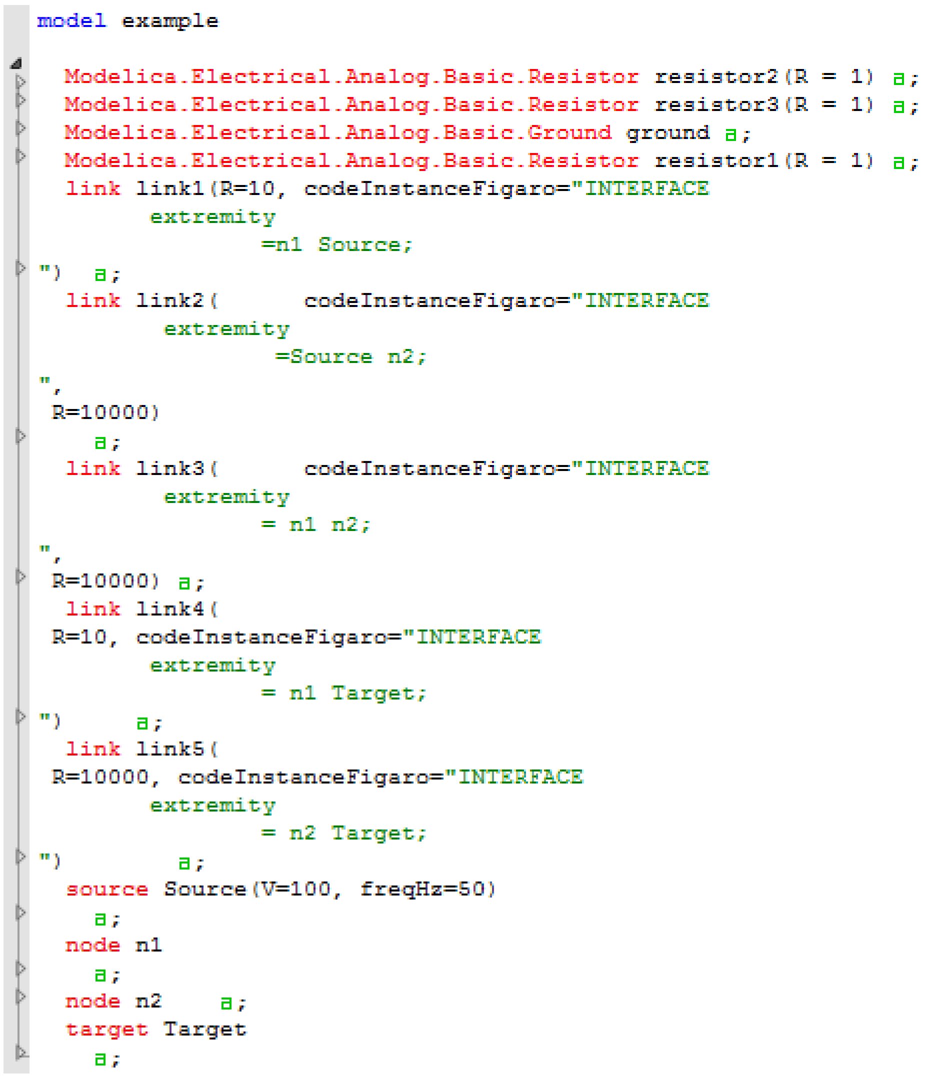
5.1.2.1. The Modelica library

```
package Network
  connector node
    extends FIGARO.Figaro_Object_connector(fullClassName="node");
    extends Modelica.Electrical.Analog.Interfaces.Pin;

    Modelica.Electrical.Analog.Interfaces.Pin pin
      a;
    a;
  end node;

  model source
    extends FIGARO.Figaro_Object(fullClassName="source");
    extends Modelica.Electrical.Analog.Sources.SineVoltage;
  end source;

  connector target
    extends Network.node(fullClassName="target")
    a;
    a;
  end target;

  model link
    extends FIGARO.Figaro_Object(fullClassName="bi_dir_link");
    extends Modelica.Electrical.Analog.Basic.Resistor;
  end link;
```

The hierarchy of classes is not the same as in the Figaro knowledge base. For example, the class "source" does not extend the class "node". Since in the electrical analog representation, all links are necessarily bi-directional, it would be meaningless to try to define a Modelica class corresponding to the mono_dir_link of the Figaro library.

5.1.2.2. Description of the system

Here is the textual description (with hidden annotations) of the system graphically displayed in Figure 6:



```
model example

  Modelica.Electrical.Analog.Basic.Resistor resistor2(R = 1) a;
  Modelica.Electrical.Analog.Basic.Resistor resistor3(R = 1) a;
  Modelica.Electrical.Analog.Basic.Ground ground a;
  Modelica.Electrical.Analog.Basic.Resistor resistor1(R = 1) a;
  link link1(R=10, codeInstanceFigaro="INTERFACE
        extremity
                =n1 Source;
")  a;
  link link2(      codeInstanceFigaro="INTERFACE
         extremity
                 =Source n2;
",
 R=10000)
    a;
  link link3(      codeInstanceFigaro="INTERFACE
         extremity
                = n1 n2;
",
 R=10000) a;
  link link4(
 R=10, codeInstanceFigaro="INTERFACE
        extremity
                = n1 Target;
")      a;
  link link5(
 R=10000, codeInstanceFigaro="INTERFACE
        extremity
                = n2 Target;
")         a;
  source Source(V=100, freqHz=50)
    a;
  node n1
    a;
  node n2     a;
  target Target
    a;
```

Once the library described in section 5.1.2.1 is available, the input of a system can be done graphically as usual, and the **only** information that needs to be added to the Modelica model in order to allow the full range of Figaro tools application is the text written in the parameters "codeInstanceFigaro". The effort is really minimal, and it could even be made easier for the user by a specific dialogue in the GUI.

5.1.3. Simulation and results with OpenModelica

The model is simulated with the following parameters:

Source: sinusoidal voltage with amplitude = 100 V and frequency = 50 Hz.

Limitation resistors = 1 Ω.

The resistance of other resistors is:

10 Ω to simulate a normal state,

10^4 Ω to simulate a failure.

Simulation time = 1 second.

5.1.3.1. Normal mode

In this mode all links are considered working (R=10Ω)

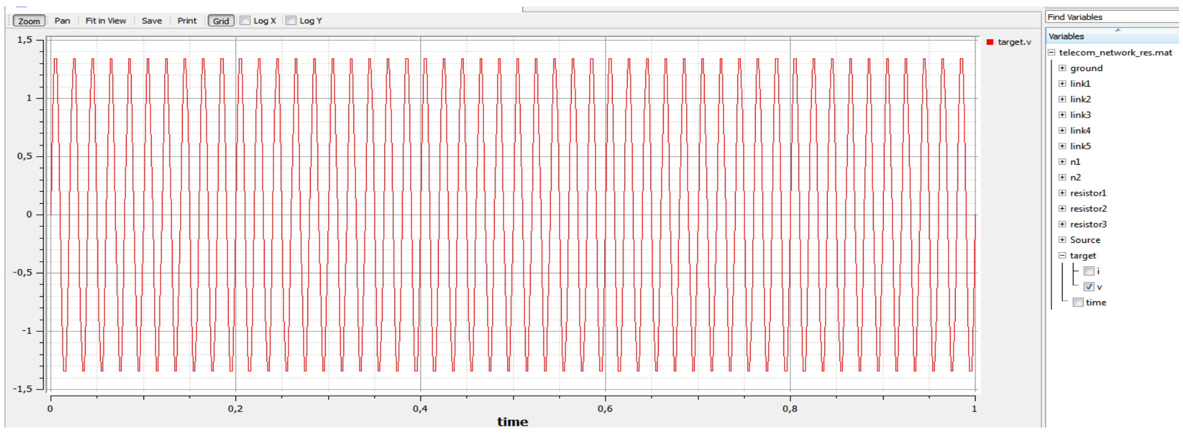


Figure 7. Normal behavior simulation result

The amplitude of the signal obtained at Target is about 1.34 V; there is a "good communication" between Source and Target.

5.1.3.2. Degraded mode

In this mode a set of links are not working but they did not affect the target (the system still works), Links 1 and 4 are working and links 2, 3 and 4 are failed. The signal still has an acceptable amplitude at node target (in red), but is very attenuated at node n2 (in blue).

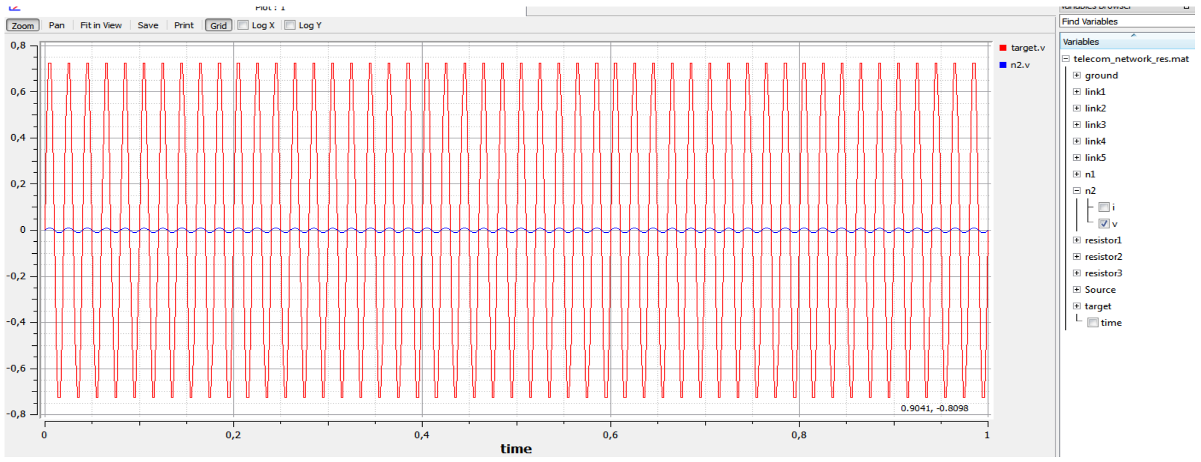


Figure 8. Degraded mode simulation result in OpenModelica

In the abstraction of the electrical network provided by the Figaro model, it is quite obvious, thanks to an interactive simulation in KB3, with a visualization of failed links by a red color, that in this situation, the node n2 is no longer connected to the source, but the target is still connected.

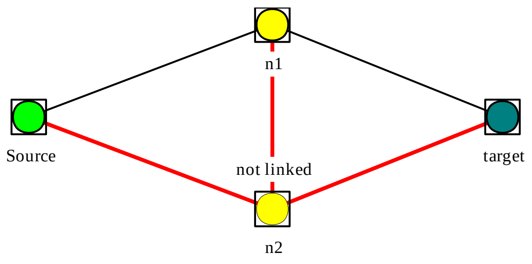


Figure 9. Degraded mode simulation in KB3

5.1.3.3. Failure of the system

A set of failed components constituting a cut set is considered in this mode. The main mission is not accomplished and the information does not reach the target (the system breaks down).

For example: links 1 and 5 are working and 2, 3 and 4 are not.

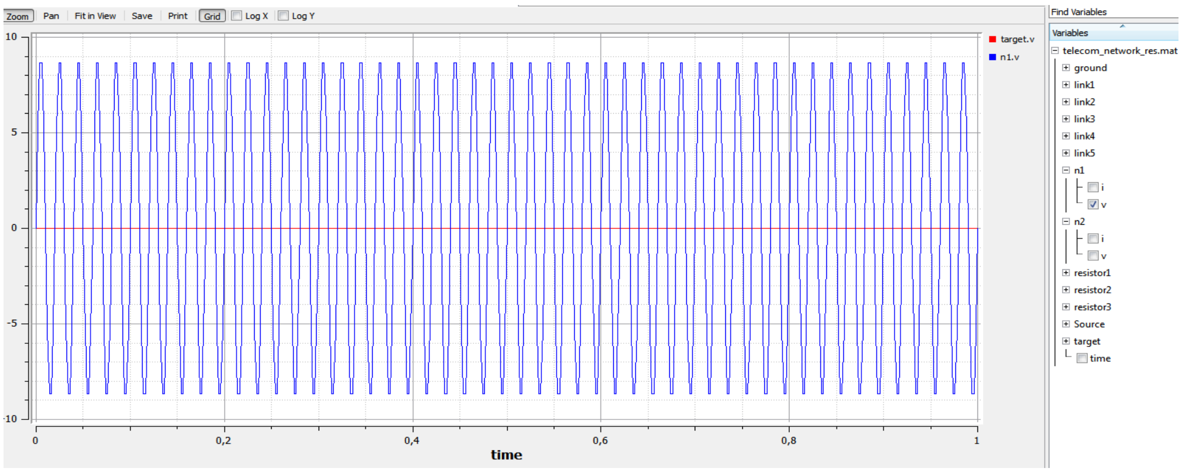


Figure 10. System failure simulation in Modelica

In this case the target does not receive the information because all the paths between Source and target are cut. Figure 10 shows also that node “n1” (blue curve) receives the information, which is expected (through link 1).

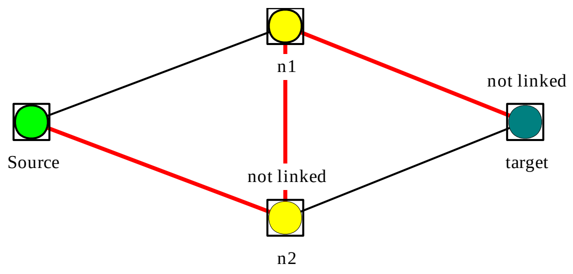


Figure 11. System failure simulation in KB3

5.1.4. Generation of a fault tree

Thanks to the Figaro processor, it is possible to automatically generate the fault tree of Figure 12.

This fault tree represents exhaustively the combinations of failures of nodes and links leading to the absence of path between the source and target of the telecom network.

In fact, the fault tree is produced as an XML file with more than 500 lines. This file is so long because it includes detailed information tracing the origin of gates in terms of interaction rules in the Figaro 0 model. Another graphical representation of the fault tree, less conventional but more readable, is given in appendix A.4.

From the fault tree, it is easy to obtain the exhaustive list of minimal cut sets, and to get the probability of failure of the system at various mission times, the importance of each component etc. These calculations use the default values of failure and repair rates of components defined in the Figaro library.

To change these values, all what is needed is to write some additional instructions in the codeInstanceFigaro strings of the Modelica model.

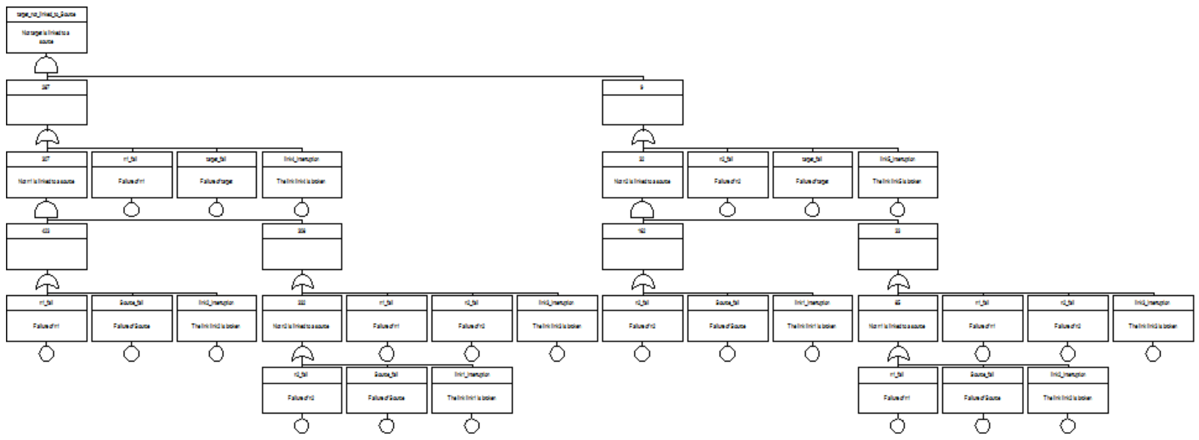


Figure 12. Fault tree generated by the Figaro processor

For example, to set the failure rate of the component Source to 5.e-3, it is sufficient to add the following instruction in the parameter codeInstanceFigaro of the component Source:

CONSTANT lambda = 5.e-3;

5.2. Thermohydraulic system

The SRI system (Intermediate Cooling System) is a system designed to ensure the cooling of important equipment of a nuclear power plant (Pumps, alternator … etc).

For this second example, we have both a real system and a real situation, in which the Modelica and Figaro libraries have been developed independently, without any intent to make them communicate.

The Modelica library used for this case is Thermosyspro.

On the Figaro side, several thermohydraulic knowledge bases have been developed, with different objectives. In order to avoid confidentiality issues, we decided to use a knowledge base called Skelbo. It is a simplified version of a knowledge base used to produce fault trees needed in probabilistic safety assessments of nuclear power plants.

Using the principles described in section 4 and following the same steps as for the telecom example, this library could be associated to a Modelica model including Figaro additions and used to build automatically a fault tree, taking as top event the absence (or incorrect temperature or pressure) of flow at the converging connection at the output of the three pumps.

5.2.1. SRI description in Modelica

This model was built using the Thermosyspro version 3.1 library. Fortunately (in the objective of a generation of a Figaro model), with this library, components in "T" must be represented explicitly. Therefore the translation from Modelica to Figaro objects is quite straightforward.

As it can be seen in the screenshot of the SRI in Dymola in Figure 13, only 9 classes had to be defined as extensions of the FIGARO classes, so the adaptation of the library was minimal.

A "fluid tester" is needed in the Figaro model to define the top event. Since there is no physical component that can play this role in the Modelica model, the fluid tester is added to the Figaro code of the volume situated at the extreme right of the figure, like this:

```
"INTERFACE

        Upstream = ClapetDP0_2 ClapetDP0_1 ClapetDP0_3;

       Downstream = VolumeD3 Bache1;

OBJECT test_loss_fluid_1 IS_A test_loss_fluid;

  INTERFACE component_to_be_tested = VolumeA1;"
```

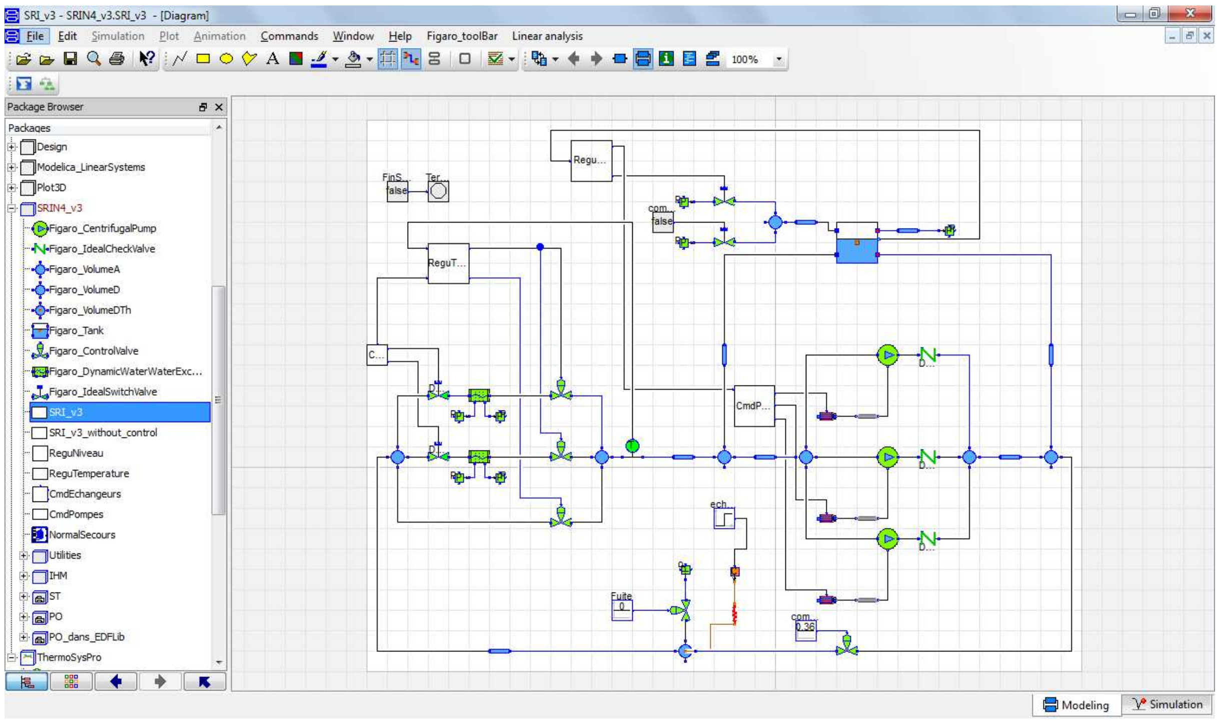


Figure 13. SRI system in Dymola

5.2.2. Description in pure Figaro

Rather than giving the textual description of the Figaro topology deduced from the Modelica model, we simply give the layout of the Figaro model as it would be displayed by KB3 in Figure 14. It makes it quite obvious that the Figaro list of objects is a sub-set of the Modelica list of objects (except for the fluid tester). We are then in a favorable position for this kind of systems, provided they are modeled with the Thermosyspro library. It might be more difficult starting from another Modelica library for fluid systems.

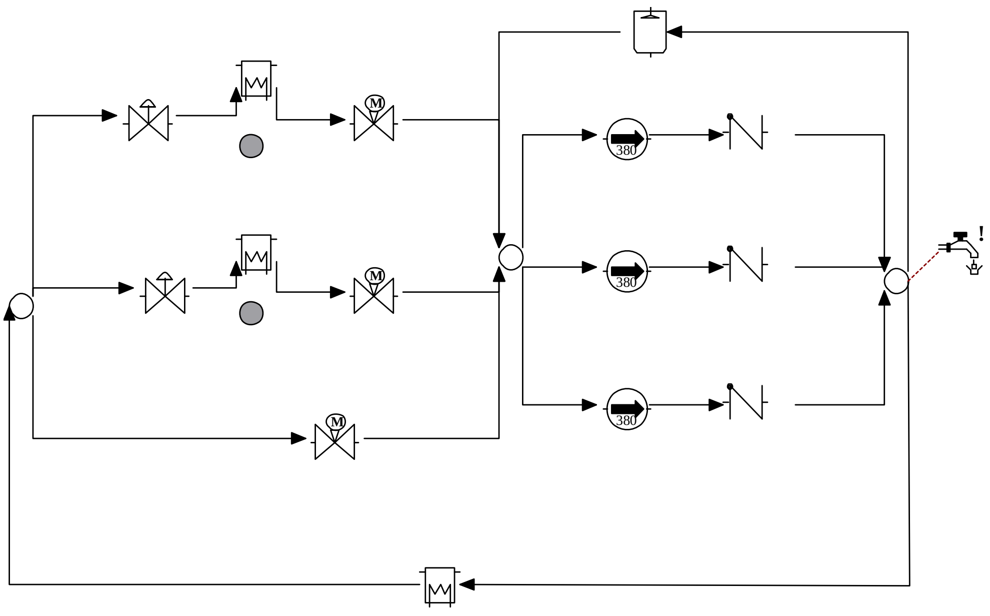


Figure 14. SRI system in KB3

The links in KB3 are purely graphical: their only purpose is to fill the upstream and downstream interfaces of components corresponding to nodes.

All the instrumentation and control part has disappeared: since a fault tree is a static model, it can only represent the functioning in permanent regime.

In the Boolean abstraction, the fluid tester (represented as a tap) has a variable called "loss". The top of the fault tree is defined by the fact that this variable takes the value true. This can mean that the flow is unsufficient (e.g. because at least two pumps out of three are failed) or that it is too hot (because heat exchangers do not work properly).

5.2.1. Fault tree generation

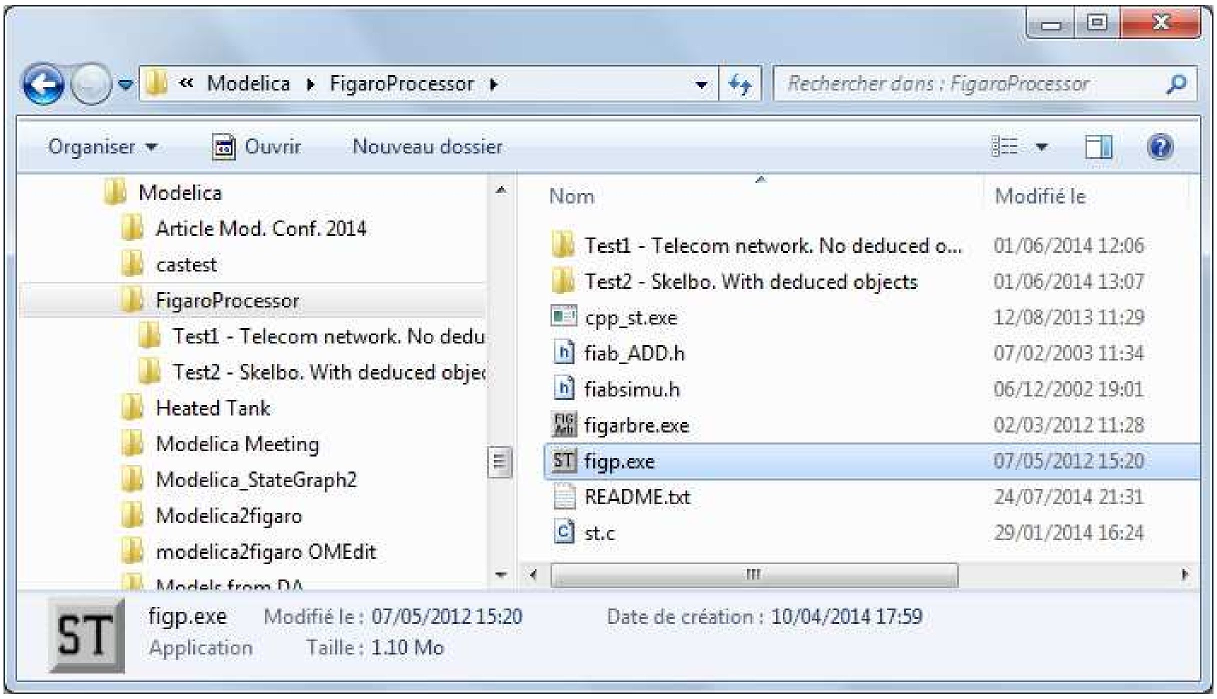
The Skelbo knowledge base can only be used to generate fault trees. It is not like the Telecom_basic knowledge base, which can also be used for interactive simulation, or processing by the tools Figseq and YAMS.

The fault tree obtained for the SRI has 72 basic events and 30 gates. These numbers could change to a large extent if another Figaro library was used, because the list of basic events for each component could be different.

6. Prototypes based on Dymola and OpenModelica

The principles described in section 4 have been implemented in two tools. The files of these prototypes and of the two use cases described in the previous section are available on the SVN server of the MODRIO project. A file "Modelica2Figaro Getting started.docx" and video tutorials explaining the use of the prototypes are also available on the SVN.

The only common part of the two prototypes is the file FIGARO.mo containing the high level Figaro classes (see section 4.3) and the files constituting the Figaro processor.


« Modelica ▸ FigaroProcessor ▸
Rechercher dans : FigaroProcessor
Organiser
Ouvrir
Nouveau dossier

Modelica
Article Mod. Conf. 2014
castest
FigaroProcessor
Test1 - Telecom network. No dedu
Test2 - Skelbo. With deduced objec
Heated Tank
Modelica Meeting
Modelica_StateGraph2
Modelica2figaro
modelica2figaro OMEdit
Models from DA

Nom	Modifié le
Test1 - Telecom network. No deduced o...	01/06/2014 12:06
Test2 - Skelbo. With deduced objects	01/06/2014 13:07
cpp_st.exe	12/08/2013 11:29
fiab_ADD.h	07/02/2003 11:34
fiabsimu.h	06/12/2002 19:01
figarbre.exe	02/03/2012 11:28
figp.exe	07/05/2012 15:20
README.txt	24/07/2014 21:31
st.c	29/01/2014 16:24

figp.exe Modifié le : 07/05/2012 15:20 Date de création : 10/04/2014 17:59
Application Taille : 1.10 Mo

Figure 15. The files of the Figaro processor

The two directories Test1 and Test2 contain example files (in pure Figaro) and explanations showing how to use the Figaro processor (figp.exe) and the fault tree generator (figarbre.exe). Figarbre.exe is invoked through figp.exe.

The needed inputs are a Figaro library and a list of objects in Figaro syntax.

All that had to be implemented in the tools Dymola and OpenModelica was the creation of this second file and the possibility to launch the Figaro processings from a menu added to the GUI.

The solutions adopted to this end were quite different, because of the constraints linked to the tools and because they were developed by different persons.

6.1. Implementation in Dymola

The developers of this solution are: Claire Campan from Dassault aviation, Ahmed Bouzakaria who did an internship at EDF and Marc Bouissou from EDF.

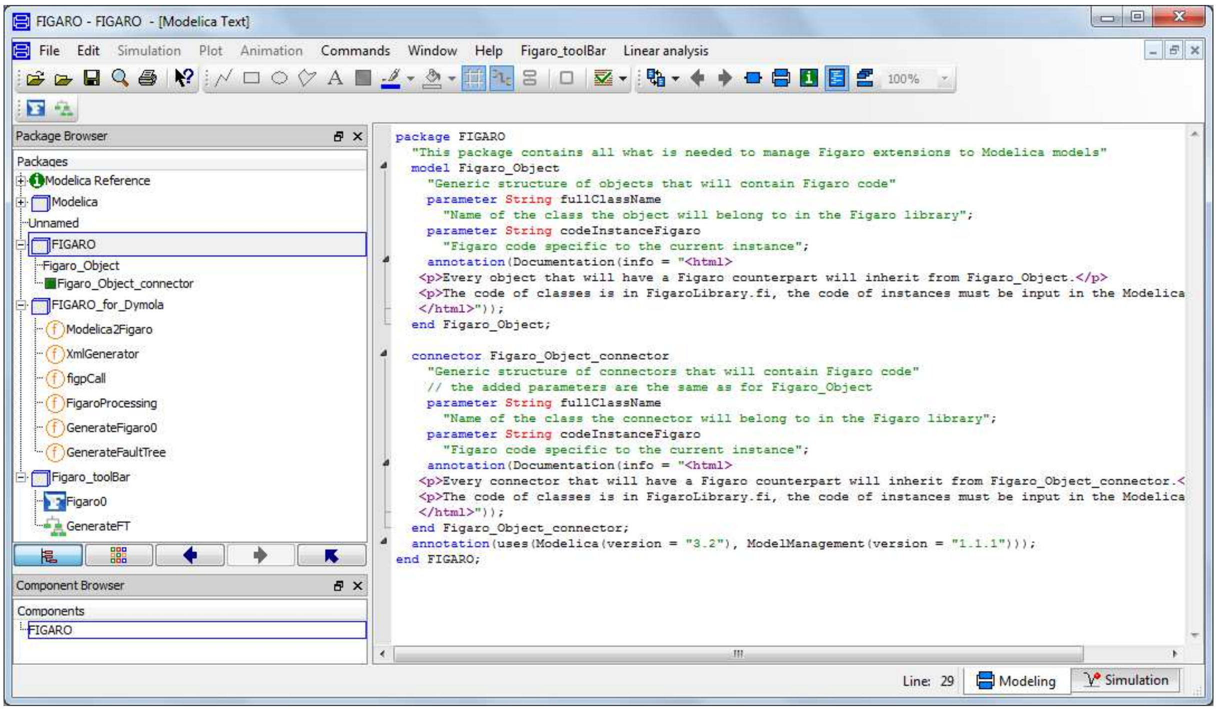
Except for a little C program of a few lines used simply to launch figp.exe, everything could be developed in Modelica. All the elements of the prototype are visible in the left part of Figure 16.

The essential part is the function Modelica2Figaro. It uses a (commercial) library: ModelManagement. Thanks to this library it is possible to explore a Modelica model containing Figaro extensions in order to gather those extensions and to write them in a file called FigaroObjects.fi. The function Modelica2Figaro contains only 100 lines of code.

The other functions are all extremely simple.

The Figaro_toolbar package adds two buttons to the GUI.

The first button calls the function GenerateFigaro0 and the second one calls the function GenerateFaultTree. Each of these functions first creates an XML file containing commands for the Figaro processor (using the function XmlGenerator) then launches it via the function figpCall.



```
package FIGARO
  "This package contains all what is needed to manage Figaro extensions to Modelica models"
  model Figaro_Object
    "Generic structure of objects that will contain Figaro code"
    parameter String fullClassName
      "Name of the class the object will belong to in the Figaro library";
    parameter String codeInstanceFigaro
      "Figaro code specific to the current instance";
    annotation(Documentation(info = "<html>
   <p>Every object that will have a Figaro counterpart will inherit from Figaro_Object.</p>
   <p>The code of classes is in FigaroLibrary.fi, the code of instances must be input in the Modelica
   </html>"));
  end Figaro_Object;

  connector Figaro_Object_connector
    "Generic structure of connectors that will contain Figaro code"
    // the added parameters are the same as for Figaro_Object
    parameter String fullClassName
      "Name of the class the connector will belong to in the Figaro library";
    parameter String codeInstanceFigaro
      "Figaro code specific to the current instance";
    annotation(Documentation(info = "<html>
   <p>Every connector that will have a Figaro counterpart will inherit from Figaro_Object_connector.<
   <p>The code of classes is in FigaroLibrary.fi, the code of instances must be input in the Modelica
   </html>"));
  end Figaro_Object_connector;
  annotation(uses(Modelica(version = "3.2"), ModelManagement(version = "1.1.1")));
end FIGARO;
```

Figure 16. The prototype based on Dymola

The advantages of the implementation in Dymola are the following:

- the code is very simple, short and written in Modelica;
- it will be possible to maintain it independently from the evolutions of Dymola;
- the call of Figaro functions is easy thanks to the addition of buttons in the GUI;
- thanks to a standard feature of Dymola, it is possible to edit the string parameters containing the Figaro code in a mini multi-line text editor.

The only drawback of this implementation is the fact that it uses a commercial library (ModelManagement).

6.2. Implementation in OpenModelica

The developer of this solution is Alexander Carlqvist, a Master student at LIU, who was supervised by Lena Buffoni.

He has written specific code which:

- adds a “Figaro” rubric to OpenModelica options. This is where the user specifies the path to the Figaro tools executable files (they are compiled independently from OpenModelica), the Figaro library to be used, and the options needed to generate a fault tree.
- adds the “ExportFigaro” command in the contextual menu of the Modelica model to be transformed into Figaro.

In this prototype, all functions equivalent to those described for Dymola are written in Modelica, in a single file: OpenModelica/Compiler/Script/Figaro.mo.

The code is longer than in the Dymola prototype because it does not rely on a library such as ModelManagement. The file Figaro.mo has a bit more than a thousand lines of code.

A few other files of the OpenModelica project have been modified in order to make these functions accessible via the GUI.

The advantages of the implementation in OpenModelica are the following:

- it is an open source solution that does not rely on any commercial library;
- error messages sent by the Figaro processor are handled and appear in the console;
- the ExportFigaro feature is now a part of the standard installation of OpenModelica.

7. Link between Modelica and O3prm

Up to this point of the report, we have described existing prototypes that can be used to carry out real studies of complex systems. This last section is more turned towards future, as no really operational toolchain from Modelica to O3prm models exists today. But the most important part of the toolchain, the O3prm editor tool, has been developed during the MODRIO project.

In this chapter we will first present shortly the O3prm language and the associated tool, and then we will explain how we intend to create a link between Modelica and O3prm.

7.1. Bayesian networks and O3prm

The O3prm language is an object oriented language dedicated to the representation of PRM: Probabilistic Relational Models [4].

PRM can be considered as the result of hybridization of Bayesian networks and relational databases. A PRM is composed of classes containing some generic knowledge and objects corresponding to a particular system.

A complete model of a system is thus made of a set of classes and a set of objects. Once the generic probabilistic relations described in classes have been instantiated on objects, **the result is equivalent to a Bayesian network**. The advantages of PRM over Bayesian networks are:

- When modeling a series of similar systems, Bayesian networks do not allow reuse of parts of models, whereas with PRM, the classes can be reused, and the only part of the model which is specific for each system is the list of objects with their connections;
- The size of Bayesian networks is limited both for practical reasons in their construction (graphs become entangled and illegible) and because of performance issues in their processing. With PRM, very large systems (with thousands of components) can be described, because the probabilistic inference starts by selecting the sub-set of objects and attributes that are really needed to answer a particular question, and this sub-set is usually small enough to allow good performance of the inference;
- Last but not least, it will be much easier to bind a Modelica model to a PRM than to a Bayesian network, thanks to the fact that Modelica models are also structured in terms of classes and objects.

7.2. Using Bayesian inference for diagnosis

When a complex system does not function as expected, it may be difficult to find which component(s) is (are) faulty. Two quite different approaches can be used to spot them:

- Use identification techniques and physical equations to estimate the largest possible amount of variable states of the system and infer from this which components do not work properly;
- Use a priori knowledge on failure probabilities and an abstract, simplified model of the system functioning to "guess" which components are more likely to be failed. It is this second approach that we want to implement thanks to PRM.

7.3. Correspondence between Modelica, O3prm and Figaro concepts

O3prm main concepts are very close to those of Figaro; this is why the following table reproduces the table of section 4.2 with an additional column for O3prm. A full definition of O3prm (syntax and semantics) can be found in the PhD dissertation of Lionel Torti [5].

From this table, it is easy to see that the method described in section 4.3 can be applied to produce an O3prm model from a few indications added to a Modelica model.

O3prm	Figaro	Modelica	Comment
Class	Class	Class	In Modelica it can be a class, package, model, record, block, or connector.
-	-	Connection	Links can be considered in Figaro or O3prm as objects whereas Modelica considers them just as connections (without parameters) that can transmit information in both directions. See §4.4.
Constant	Constant	Parameter	
Object	Object	Model	Every component created from a class is an object (instantiation)
-	Interaction rule	-	Modelica does not support any creation of rules for components.
-	Occurrence rule	-	
Conditional probability table	-	-	Modelica does not support any probabilistic concept.
Reference	Interface	-	A Reference in O3prm is used exactly like an Interface in Figaro. These notions have no equivalent in Modelica. Indeed, the way connections are established between components is quite different in Modelica and Figaro (or O3prm): see §4.4.

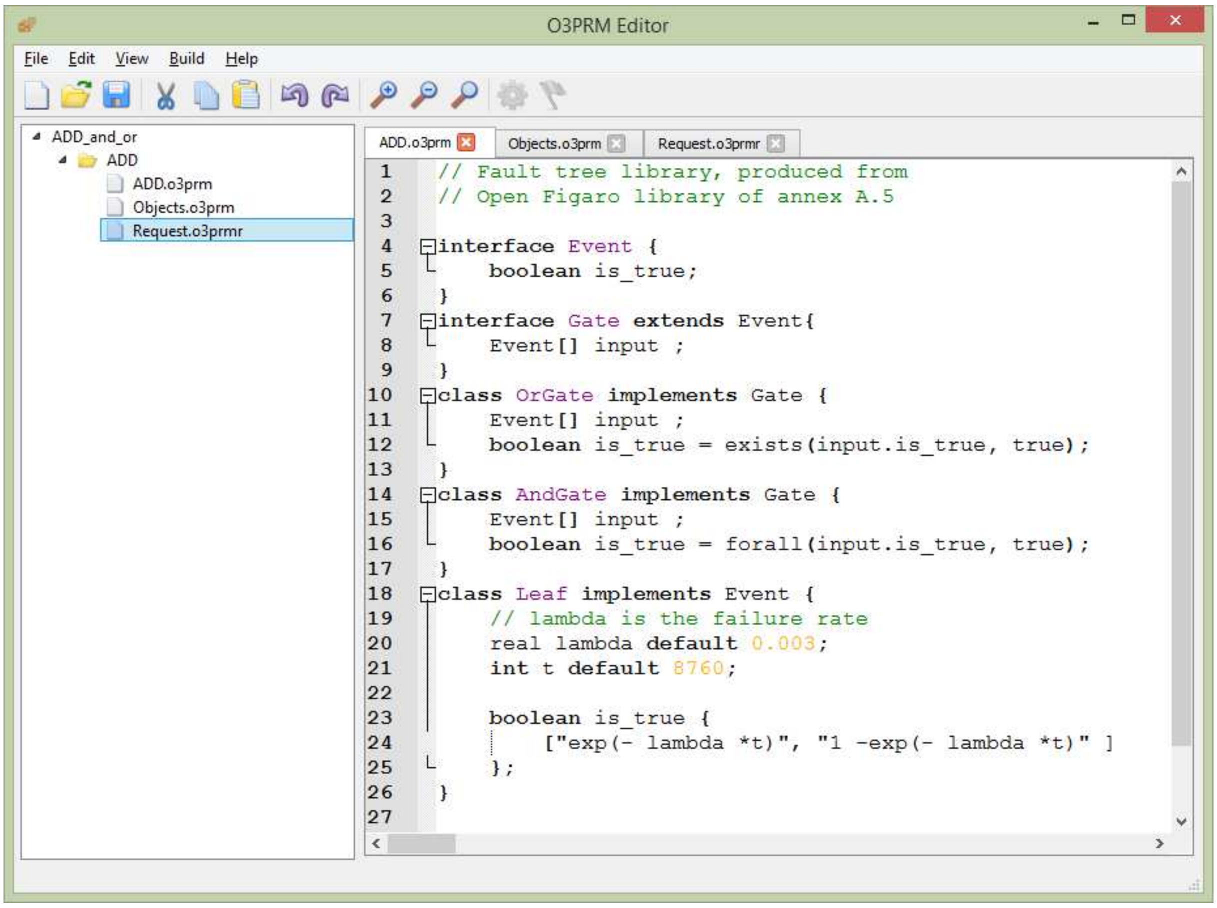
As one can see from this table, it is possible to unify the Figaro and O3prm languages because they have the same set of concepts, except for the way they describe interactions between objects: this is done via rules in Figaro, and via conditional probability tables in O3prm.

EDF is working on this unification, thanks to a new language called Open Figaro. The syntax of Open Figaro will be very close to the syntax of Figaro; it will be extended in order to include conditional probability tables and the possibility to have detail levels in a model. An example of an Open Figaro library describing the concepts of simple fault trees is given in appendix A.5. At the beginning of the library, two group names (figaro and o3prm) are declared: the interactions declared in the "interaction" section belong to only one of these groups. A translator can then automatically extract either pure Figaro or pure O3prm libraries from this Open Figaro library.

7.1. The O3prm tool

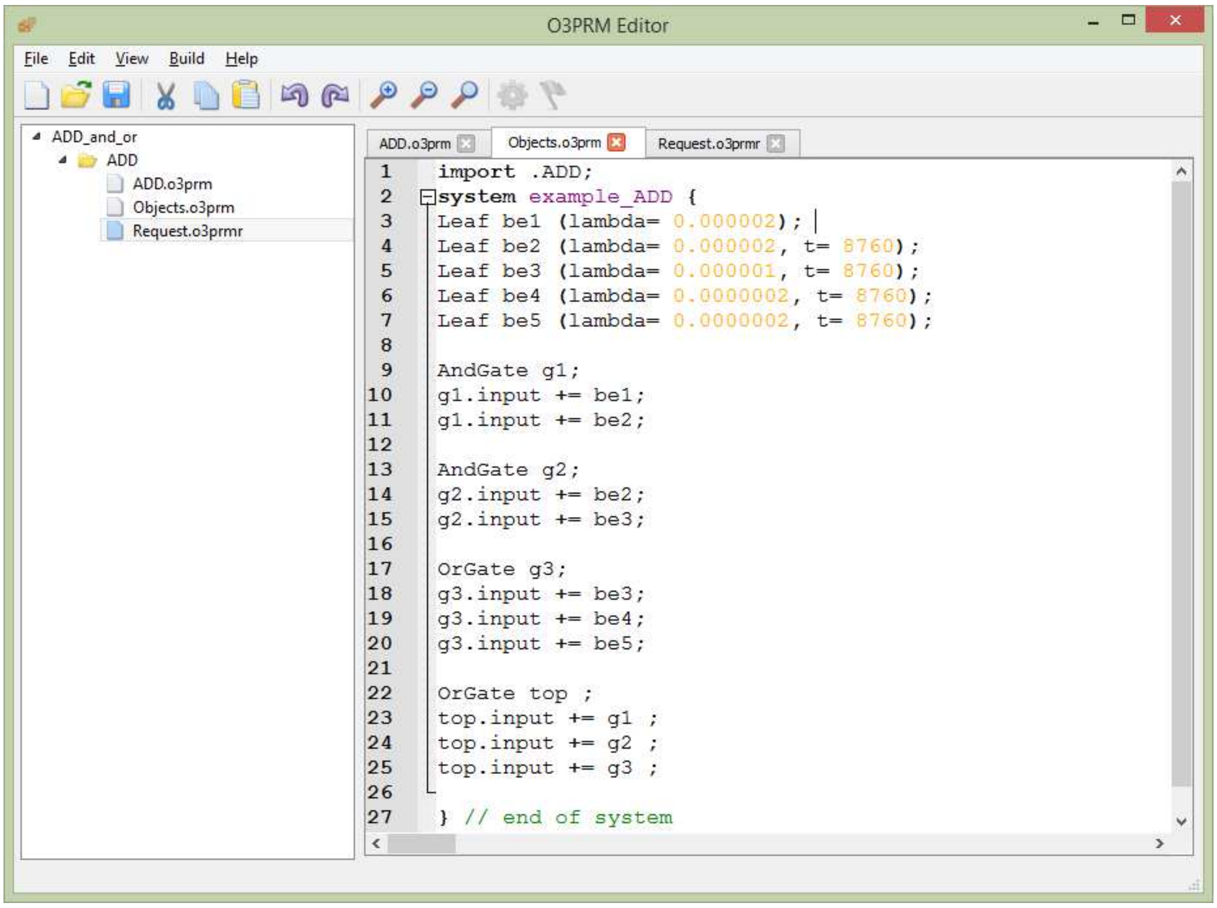
This open source tool was developed at Lip6 (computer science lab of ParisVI university), on the basis of Lionel Torti's Phd work [5] on inference algorithms in PRMs.

The tool "O3prm editor" looks like a text editor dedicated to the edition of O3prm models. It can manage "O3prm projects", composed of files belonging to three categories: libraries, system descriptions and requests. Once a project is ready, the tool can check the syntax of all files and, if it is correct, execute the requests. They consist in a sequence of observations (a given variable of a given object has a given value) and questions on the probability distributions of variables which are not observed. The figures below show the user interface of the O3prm editor, taking as example the description of a fault tree by means of its gates and leaves. Figure 19 shows how this fault tree can be used to perform a diagnosis thanks to Bayesian inference. After entering as an observation the fact that the top event of the fault tree is true, the inference can update the probability of the leaves, thus showing the components whose failure is most probable, given that the top event is realized.



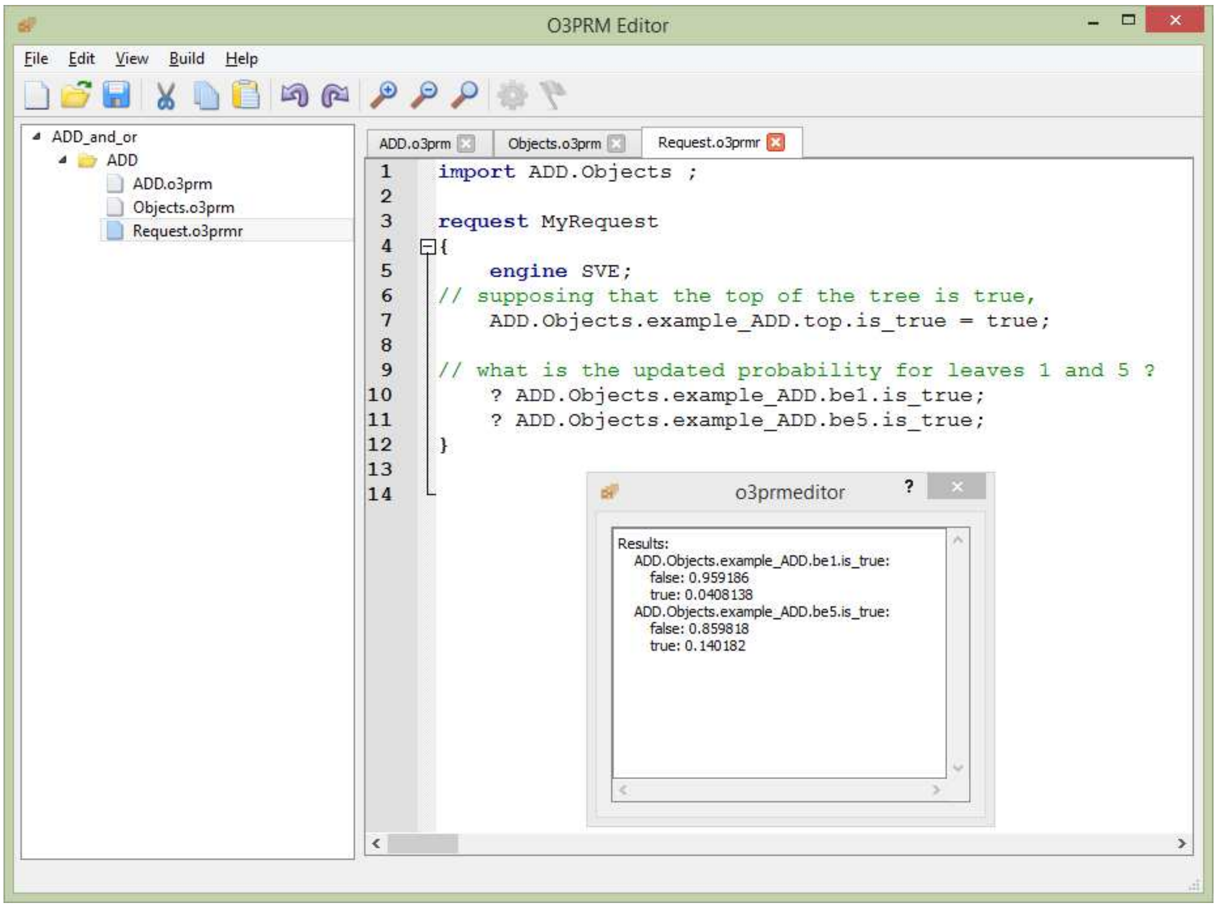
```
// Fault tree library, produced from
// Open Figaro library of annex A.5

interface Event {
     boolean is_true;
}
interface Gate extends Event{
     Event[] input ;
}
class OrGate implements Gate {
     Event[] input ;
     boolean is_true = exists(input.is_true, true);
}
class AndGate implements Gate {
     Event[] input ;
     boolean is_true = forall(input.is_true, true);
}
class Leaf implements Event {
     // lambda is the failure rate
     real lambda default 0.003;
     int t default 8760;

     boolean is_true {
          ["exp(- lambda *t)", "1 -exp(- lambda *t)" ]
     };
}
```

Figure 17. The O3prm user interface: editing a library



```
import .ADD;
system example_ADD {
Leaf be1 (lambda= 0.000002);
Leaf be2 (lambda= 0.000002, t= 8760);
Leaf be3 (lambda= 0.000001, t= 8760);
Leaf be4 (lambda= 0.0000002, t= 8760);
Leaf be5 (lambda= 0.0000002, t= 8760);

AndGate g1;
g1.input += be1;
g1.input += be2;

AndGate g2;
g2.input += be2;
g2.input += be3;

OrGate g3;
g3.input += be3;
g3.input += be4;
g3.input += be5;

OrGate top ;
top.input += g1 ;
top.input += g2 ;
top.input += g3 ;

} // end of system
```

Figure 18. The O3prm user interface: editing a system



```
import ADD.Objects ;

request MyRequest
{
    engine SVE;
// supposing that the top of the tree is true,
    ADD.Objects.example_ADD.top.is_true = true;

// what is the updated probability for leaves 1 and 5 ?
    ? ADD.Objects.example_ADD.be1.is_true;
    ? ADD.Objects.example_ADD.be5.is_true;
}
```

```
Results:
  ADD.Objects.example_ADD.be1.is_true:
     false: 0.959186
     true: 0.0408138
  ADD.Objects.example_ADD.be5.is_true:
     false: 0.859818
     true: 0.140182
```

Figure 19. The O3prm user interface: requests and answers

7.2. Implementation of a link from Modelica to O3prm

The prototypes described in section 6 could be used with only cosmetic changes to export a list of objects from a Modelica model. These objects could then be associated to a library of Open Figaro classes to constitute a complete Open Figaro model. Then this model could be translated, according to the objectives of the user, into a pure Figaro model or a pure O3prm model. The needed developments will be done after the MODRIO project. They will pave the way towards innovative uses of Open Figaro models, that will exploit jointly the two facets (Figaro and O3prm) of this new modeling language.

8. Conclusion

This document has described the principles of tools that allow combining the power of Modelica for system analog simulation and Figaro for various reliability analyses, including automatic fault tree generation.

The use of prototypes of such tools, based on Dymola and OpenModelica, has been demonstrated, both on a simple, easy to understand telecommunication network system and on one of the real systems chosen among the use cases of the MODRIO project: the SRI thermohydraulic system.

These prototypes are good enough for testing this approach and evaluating its benefits on real system studies. However if an industrial use emerges, some simple enhancements in the tool support will be needed in order to automate the transformation from Modelica to Figaro. Unlike the current prototypes which are completely generic, a fully automatic translator will have to be tailored to the chosen Modelica and Figaro libraries.

Another objective of WP2.2 was to generate Bayesian networks from Modelica models. We have shown that a better solution would be to do a translation towards PRMs (Probabilistic Relational Models). We have presented the brand new O3prm tool developed for the creation and manipulation of this kind of models.

As an after-MODRIO perspective, we have introduced a new modeling language called Open Figaro that will be an extension of Figaro allowing also to describe PRMs. The definition of this new language is a first step towards innovative applications that will exploit jointly the two facets (Figaro and O3prm) of the models. Another advantage is that the existing tools that transform Modelica models into Figaro models will also work, practically without any change, with Open Figaro.

9. References

[1] Bouissou, M., Elmqvist, H., Otter, M. & Benveniste, A., "Efficient Monte Carlo simulation of stochastic hybrid systems," in Proceedings of 'Modelica Conference 2014'.

[2] Claire Campan, "Boolean models in Modelica," presentation at IMdR meeting, Paris 3rd April 2014.

[3] Alexander Carlqvist, "OpenModelica Support for Figaro Extensions Regarding Fault Analysis," Master thesis. LIU-IDA/LITH-EX-G--14/042—SE. June 2014.

[4] Daphne Koller, "Probabilistic Relational Models," in Proceedings of 9th International Workshop on Inductive Programming, ILP-99. Bled, Slovenia, June 24-27, 1999

[5] Lionel Torti, "Structured Probabilistic Inference in Object-Oriented Bayesian Networks," PhD Thesis, Université Paris 6, 2012.

[6] Marc Bouissou, X. de Bossoreille, "From Modelica models to dependability analysis", in proc. DCDS 2015, Cancun.

A. Appendix

A.1. The Telecom network knowledge base in Figaro

NB: the capitalized words (and only those) are Figaro keywords.

```
(* English comments after French comments*)

(* Petite base de connaissances pédagogique *)
(* Elle permet de décrire des réseaux de télécommunication
   maillés avec des défaillances (réparables) sur les noeuds et les liens.
   Elle a deux utilisations possibles : simulation (avec le groupe
   de règles groupe_simu) et génération d'arbres de
   défaillances (avec le groupe de règles groupe_add) *)
(* Auteur : Marc Bouissou, 20 déc 2007 *)

(* Small  pedagogical knowledge base*)
(* It is used to describe meshed telecommunications networks
     with (repairable) failures on nodes and links.
   It has two possible uses: simulation (with the simu_group
      rules group) and generating fault trees
     (with the FT_group rules group) *)
(* Author: Marc Bouissou, December 20, 2007 *)

GROUP_NAMES
simu_group ;
FT_group ;

CLASS node ;
 CONSTANT
       function DOMAIN 'source' 'target' 'intermediate' DEFAULT 'intermediate' ;
       lambda DOMAIN REAL DEFAULT 1e-5 ROLE DESIGN;
       mu DOMAIN REAL DEFAULT 0.1 ROLE DESIGN;
 FAILURE fail LABEL "Failure of %OBJECT"
       RELIABILITY_DATA
       GROUP FT_group
       MODEL_GLM
       GAMMA 0.
       LAMBDA lambda
       MU mu ;
 EFFECT connected LABEL "%OBJECT is linked to a source" ;
 OCCURRENCE
       GROUP simu_group
       MAY_OCCUR
       FAULT fail
       DIST EXP(lambda);

       GROUP simu_group
       MAY_OCCUR
       REPAIR rep REPAIRS fail
       DIST EXP(mu);
 INTERACTION
       rule1
       IF WORKING AND function = 'source'
       THEN connected;

CLASS source KIND_OF node ;
 CONSTANT
       function DEFAULT 'source';

CLASS target KIND_OF node ;
 CONSTANT
       function DEFAULT 'target';

CLASS link ;
 CONSTANT
       link_lambda DOMAIN REAL DEFAULT 1e-5 ROLE DESIGN ;
       link_mu DOMAIN REAL DEFAULT 1 ROLE DESIGN ;
 FAILURE
       interruption LABEL "The link %OBJECT is broken"
       RELIABILITY_DATA
       GROUP FT_group
       MODEL_GLM
       GAMMA 0.
       LAMBDA link_lambda
       MU link_mu ;
 OCCURRENCE
       GROUP simu_group
       MAY_OCCUR
       FAULT interruption
       DIST EXP(link_lambda);

       GROUP simu_group
       MAY_OCCUR
       REPAIR rep REPAIRS interruption
       DIST EXP(link_lambda);

CLASS mono_dir_link KIND_OF link ;
 INTERFACE
```

```
       start KIND node CARDINAL 1 EDITION NOT MODIFIABLE;
       end KIND node CARDINAL 1 EDITION NOT MODIFIABLE;
 INTERACTION
       rule1
       IF WORKING AND connected(start) AND WORKING(end)
       THEN connected(end) ;

CLASS bi_dir_link KIND_OF link ;
 INTERFACE
        extremity KIND node CARDINAL 2 EDITION NOT MODIFIABLE;
 INTERACTION
        rule1
        IF WORKING AND
        (FOR_ANY x AN extremity WE_HAVE WORKING(x)) AND
        IT_EXISTS x AN extremity SUCH_THAT connected OF x
        THEN FOR_ALL z AN extremity DO connected(z);

CLASS fail_counter ;
(* Compteur des défaillances présentes dans le système (noeuds et liens) *)
(* Count of failures in the system (nodes and links) *)
 ATTRIBUTE
       nb_failures DOMAIN INTEGER DEFAULT 0;
 INTERACTION
       rule1
       GROUP simu_group
       THEN nb_failures <-- SUM FOR_ALL x AN OBJECT OF_CLASS node OF_TERMS (1 * fail(x)) +
       SUM FOR_ALL y AN OBJECT OF_CLASS link OF_TERMS (1 * interruption(y)) ;

(* Un OBJET_SYSTEME est un objet global, qui peut accéder aux variables de tous les objets
   faisant partie du système *)
(* A SYSTEM_OBJECT is a global object that can access the variables of all objects
    forming part of the system *)
SYSTEM_OBJECT Failure_counter IS_A fail_counter ;
```

A.2. The Telecom network system in Figaro (list of objects)

```
OBJECT Source IS_A source;
OBJECT n1 IS_A node;
OBJECT n2 IS_A node;
OBJECT target IS_A target;
OBJECT bidir_1 IS_A bi_dir_link;
       INTERFACE
         extremity
              = n1 Source;
OBJECT bidir_2 IS_A bi_dir_link;
```

```
       INTERFACE
         extremity
              = Source n2;
OBJECT bidir_3 IS_A bi_dir_link;
       INTERFACE
         extremity
              = n1 n2;
OBJECT bidir_4 IS_A bi_dir_link;
       INTERFACE
         extremity
              = n1 target;
OBJECT bidir_5 IS_A bi_dir_link;
       INTERFACE
         extremity
              = n2 target;
```

A.3. The Telecom network system in Figaro 0

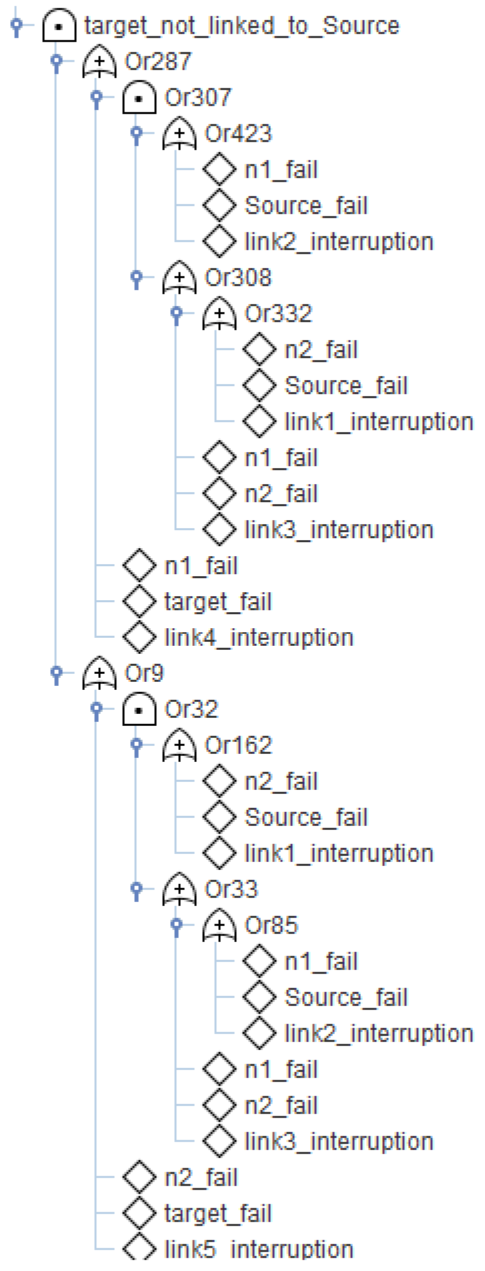
This section shows representative excerpts of the generated Figaro 0 (version for simulation):

```
…
OBJECT Source IS_A source;
       CONSTANT
         function
              DOMAIN 'source' 'target' 'intermediate'
              = 'source';
         lambda
              DOMAIN REAL
              = 1e-005;
         mu
              DOMAIN REAL
              = 0.1;
       (*FAILURE*)
       ATTRIBUTE
         fail
              LABEL "Failure of %OBJECT"
              DOMAIN BOOLEAN
              = FALSE;
       ATTRIBUTE
         connected
              LABEL "%OBJECT is linked to a source"
              DOMAIN BOOLEAN
              REINITIALISATION FALSE;
       INTERACTION
         rule1
              STEP default_step
```

```
               IF fail OF Source = FALSE
               THEN connected OF Source <-- TRUE;
       OCCURRENCE
         xx1
               GROUP simu_group
               IF fail OF Source = FALSE
               MAY_OCCUR
                 FAULT fail
                 DIST EXP (1e-005)
                 INDUCING fail OF Source <-- TRUE;
         xx2
               GROUP simu_group
               IF fail OF Source = TRUE
               MAY_OCCUR
                 REPAIR rep
                 DIST EXP (0.1)
                 INDUCING fail OF Source <-- FALSE;

OBJECT n1 IS_A node;
       CONSTANT
         function
               DOMAIN 'source' 'target' 'intermediate'
               = 'intermediate';
         lambda
               DOMAIN REAL
               = 1e-005;
         mu
               DOMAIN REAL
               = 0.1;
       (*FAILURE*)
       ATTRIBUTE
         fail
               LABEL "Failure of %OBJECT"
               DOMAIN BOOLEAN
               = FALSE;
       ATTRIBUTE
         connected
               LABEL "%OBJECT is linked to a source"
               DOMAIN BOOLEAN
               REINITIALISATION FALSE;
       OCCURRENCE
         xx1
               GROUP simu_group
               IF fail OF n1 = FALSE
               MAY_OCCUR
                 FAULT fail
                 DIST EXP (1e-005)
                 INDUCING fail OF n1 <-- TRUE;
         xx2
               GROUP simu_group
               IF fail OF n1 = TRUE
               MAY_OCCUR
                 REPAIR rep
                 DIST EXP (0.1)
                 INDUCING fail OF n1 <-- FALSE;
OBJECT bidir_1 IS_A bi_dir_link;
       INTERFACE
         extremity
               = n1 Source;
       CONSTANT
         link_lambda
               DOMAIN REAL
               = 1e-005;
         link_mu
               DOMAIN REAL
               = 1;
       (*FAILURE*)
       ATTRIBUTE
         interruption
               LABEL "The link %OBJECT is broken"
               DOMAIN BOOLEAN
               = FALSE;
       INTERACTION
         rule1
               STEP default_step
               IF ((interruption OF bidir_1 = FALSE) AND ((fail OF n1 = FALSE)
                 AND (fail OF Source = FALSE))) AND (connected OF n1 OR
                 connected OF Source)
               THEN connected OF n1 <-- TRUE,
                 connected OF Source <-- TRUE;
       OCCURRENCE
         xx3
               GROUP simu_group
               IF interruption OF bidir_1 = FALSE
               MAY_OCCUR
                 FAULT interruption
                 DIST EXP (1e-005)
                 INDUCING interruption OF bidir_1 <-- TRUE;
```

```
          xx4
                GROUP simu_group
                IF interruption OF bidir_1 = TRUE
                MAY_OCCUR
                  REPAIR rep
                  DIST EXP (1e-005)
                  INDUCING interruption OF bidir_1 <-- FALSE;
...
```

A.4. The fault tree produced for the telecom network system



```
target_not_linked_to_Source
  Or287
    Or307
      Or423
        n1_fail
        Source_fail
        link2_interruption
      Or308
        Or332
          n2_fail
          Source_fail
          link1_interruption
        n1_fail
        n2_fail
        link3_interruption
    n1_fail
    target_fail
    link4_interruption
  Or9
    Or32
      Or162
        n2_fail
        Source_fail
        link1_interruption
      Or33
        Or85
          n1_fail
          Source_fail
          link2_interruption
        n1_fail
        n2_fail
        link3_interruption
    n2_fail
    target_fail
    link5_interruption
```

A.4. An Open Figaro library describing fault tree concepts

```
(* A simple description of fault trees with AND and OR gates *)
(* and non repairable components with constant failure rate. *)

group_names
  figaro;
  o3prm;

class Gate ;
	attribute
		is_true domain boolean reinitialisation false ;
	reference
		input kind Gate cardinal 0 to infinity ;



class OrGate extends Gate ;
	interaction
		group figaro
			if it_exists x an input such_that is_true of x
			then is_true ;
		group o3prm
			cpt boolean is_true = exists(input.is_true, true);



class AndGate extends Gate ;
	interaction
		group figaro
			if for_any x an input we_have is_true of x
			then is_true ;
		group o3prm
			cpt is_true = forall(input.is_true, true);

class Leaf extends Gate ;
	failure
		fail;
	attribute
		Lambda domain real default 0.003;
		t domain integer default 8760;
	reference
		input kind Gate cardinal 0 ;
	interaction
	    group figaro
		 if fail
		 then is_true ;
```

```
              group o3prm
                     cpt is_true
                      raw
                        "exp(- Lambda *t)",
                        "1 -exp(- Lambda *t)"
                       ;
    occurrence
       may_occur
              fault fail
                     label "%OBJECT failed in operation"
               dist exp(Lambda) ;
```