AltaRica 3.0 in 10 Modeling Patterns

Michel Batteux

IRT SystemX, Paris-Saclay, France

Tatiana Prosvirnova

Chaire Blériot-Fabre, CentraleSupélec, Gif-sur-Yvette, France

Antoine Rauzy

Norwegian University of Science and Technology, Trondheim, Norway

Abstract: AltaRica 3.0 is an object-oriented modeling language dedicated to probabilistic risk and safety analyses. It is a prominent representative of modeling formalisms supporting the so-called model-based approach in reliability engineering. In this article, we illustrate the key features of the AltaRica 3.0 technology by presenting the implementation of ten very common modeling patterns. We demonstrate in this way the expressive power of the language as well as its elegance and simplicity of use.

Keywords: Probabilistic risk and safety assessment; modeling languages; modeling patterns; AltaRica 3.0

1 Introduction

The aim of this article is to present the key features of the object-oriented, formal, modeling language AltaRica 3.0. The AltaRica 3.0 technology is dedicated to probabilistic risk and safety analyses of complex technical systems. We shall use here the expression "probabilistic risk analysis" or equivalently "probabilistic risk assessment" in a broad sense, which encompasses processes as diverse as safety and reliability assessments, optimizations of maintenance policies, assessments of the expected production level over a given period and so on. In a word, we shall speak here about assessments of operational performance of technical systems subject to random events such as mechanical failures, operator errors…

A long journey has been made since the publication of the WASH 1400 report, [1]. Probabilistic risk and safety analyses are nowadays used on a daily basis in virtually all industries presenting significant risks for their operators, the public or the environment. Safety standards such as [2], [3] or [4] recommend this approach.

As of today, these analyses rely mainly on fault trees, event trees, reliability block diagrams or a combination of those, see e.g. [5, 6, 7] for reference books. These modeling formalisms present a good compromise between the ease of use, the accuracy of descriptions and the computational complexity of assessments. However, they have important drawbacks as well. First, they lack expressive power. Cold and warm redundancies, exclusive failures, reconfigurations, control mechanisms and many other increasingly common features of technical systems can only be approximated, leading to over-pessimism and inaccurate ranking of causes of risks, see e.g. [8]. Second, models designed with these formalisms are very distant from system specifications. Retrieving system specifications from the safety models is nearly impossible. For this reason, these models are hard to share with stakeholders and to maintain through the life-cycle of systems.

These drawbacks were compensated by advantages when systems at stake where purely mechanical, or at least when they mechanical parts were dominant. It is not the case anymore for software intensive, high integrity, critical systems. In these systems, the control (and reconfigurations) play a central role. We can characterize them as deformable: their architecture changes throughout their mission. New methods have to be developed to describe systems' behavior more accurately.

To get more expressive power, one needs to leave combinatorial (Boolean) formalisms for states/events formalisms such as Markov chains or stochastic Petri nets [9]. Explicit representations of the state space, such as Markov chains, suffer from the exponential blow-up of the number of states and transitions. Implicit representations, such as stochastic Petri nets, are thus highly preferable. They are however not sufficient in themselves to reduce the distance between system specifications and safety models. With that respect, the lack of structure of stochastic Petri nets is an important drawback, see e.g [10].

The promise of the so-called model-based risk and safety assessment approach is to provide analysts with modeling formalisms that have both a high expressive power and suitable structuring mechanisms. It is possible in this way to design models that reflect the functional and physical architectures of the system under study. Safety models are thus closer to systems specifications, which makes them both easier to share with non-specialists and to maintain. Moreover, a single model can be used to assess several safety goals.

Modeling formalisms that support this approach can be classified into three categories. The first category consists of specialized profiles of model-based systems engineering formalisms such as SysML, see e.g. [11, 12, 13, 14]. The objective here is however more to introduce a safety facet into models of system architecture than to design actual safety models. The second category consists of extensions of fault trees or reliability block diagrams so to enrich their expressive power. This category includes dynamic fault trees [15, 16], multistate systems [17, 18, 19], and some other proposals [10]. The third category, which aims at taking fully advantage of the model-based approach, consists of modeling languages such as SAML [20], Figaro [21] and AltaRica [22]. SAML is oriented towards probabilistic model checking (and compiled into PRISM descriptions [23]). Figaro has been historically the first modeling language dedicated to probabilistic risk and safety analyses. It is a rule-based systems [24, 25]. Since the very first version of AltaRica, the choice has been made to rely on the more natural and mathematically clearer notion of state automata.

AltaRica 3.0 is, as its name suggests, the third version of the language. The design of this new version started in 2012 to take advantage of more than 10 years of academic and industrial experience accumulated with the previous versions [26]. AltaRica 3.0 can be described by the following equation.

$$\mathrm{GTS} + \mathrm{S2ML} = \mathrm{AltaRica\ 3.0} \tag{1}$$

The above equation[1] summarizes an idea that goes much beyond risk and safety analyses and that can be stated as follows. Any behavioral description language is made of two parts:

a mathematical framework to describe behaviors and a set of constructs to structure models. In the case of AltaRica 3.0, the mathematical framework is the notion of guarded transition systems (GTS), see [27, 28] for in depth presentations. Guarded transition systems are state automata. They have been designed to increase as much as possible the expressive power of the language without increasing the computational cost of assessment algorithms. S2ML stands for system structure modeling language [29]. S2ML gathers in a coherent way structuring constructs stemmed from object-oriented programming, see e.g. [30], and prototype-oriented programming, see e.g. [31]. The combination of GTS and S2ML results in a powerful, versatile language which exploits in an optimum way assessment algorithms.

It is of primary importance, in order to make the modeling process efficient, to reuse as much as possible modeling components within models and between models. In languages such a Modelica [32], this goal is achieved via the design of libraries of on-the-shelf ready-to-use modeling components. Reusing components is also possible in probabilistic risk and safety analyses, but to a much lesser extent. The reason is that these analyses represent systems at a high level of abstraction. Modeling components, except for very basic ones, tend thus to be specific to each system. In AltaRica 3.0, reuse is mostly achieved by the design of modeling patterns, i.e. examples of models representing remarkable features of the system under study. Once identified, patterns can be duplicated and adjusted for specific needs, see e.g. references [33] for a preliminary study and [34] for a recent application. Patterns are pervasive in engineering. They have been developed for instance in the field of technical system architecture [35], as well as in software engineering [36]. Patterns are not only a mean to organize and to document models, but also and more fundamentally a way to reason about systems under study.

It is probably too early to design a taxonomy of modeling patterns encountered in operational performance analyses. For the time being, we classify patterns into two categories: behavioral patterns that aim at describing the behavior of a single component or a small group of components, and architectural patterns that aim at describing the whole model organization. We present and discuss here five patterns of each category.

The contribution of this article is thus twofold: first, it introduces the reader with the AltaRica 3.0 technology; second, it studies some of the most common modeling patterns encountered in probabilistic risk and safety analyses.

The reminder of this article is organized as follows. Section 2 introduces basic concepts of the language via the behavioral pattern “repairable unit” and two fundamental architectural patterns: the “structural decomposition” and “hierarchical block diagram” patterns. Section 3 describes four common behavioral patterns: the “periodically tested unit”, “warm redundancy”, “shared resource” and “common cause failure” patterns. Section 4 describes three advanced architectural patterns: the “reliability network”, “production tree”, and “monitored system” patterns. Finally, Section 5 gives a snapshot on assessment tools and concludes the article.

2 Getting started

2.1 Guarded transition systems

Guarded transition systems, introduced in [27], are at the core of AltaRica 3.0. Formally, a guarded transition system is a quintuple $\langle V, E, T, A, \iota \rangle$, where:

- V is a set of variables. Each variable v of V has a type, i.e. can take its value in a certain set of constants (Booleans, integers, reals or set of symbolic constants), called its domain and denoted by $\mathrm{dom}(v)$. V is actually the disjoint union of two subsets S and F, where S is the subset of state variables and F is the subset of flow variables.

- E is a set of events. Each event e of E can be associated with a non-decreasing invertible function $delay_e$ from $[0,1]$ to $\mathbb{R}^+ \cup \{+\infty\}$. The inverse $delay_e^{-1}$ of this function is a cumulative probability distribution.

- T is a set of transitions. A transition t is a triple $\langle e, g, a \rangle$, denoted by $g \xrightarrow{e} a$, where e is an event of E, g is a Boolean condition on variables of V called the guard of the transition, and a is an instruction, called the action of the transition, that changes the value of (some of) the state variables.

- A is an instruction, called the assertion, that calculates the values of flow variables from the values of state variables.

- Finally, ι is a function that gives the initial value of state variables and the default value of flow variables.

As an illustration, we shall consider our first behavioral pattern.

Modeling Pattern 1 (Repairable unit) *We call repairable unit a component that may fail and be repaired.*

The guarded transition system for such a unit is graphically represented Figure 1. The corresponding AltaRica 3.0 code is given Figure 2.

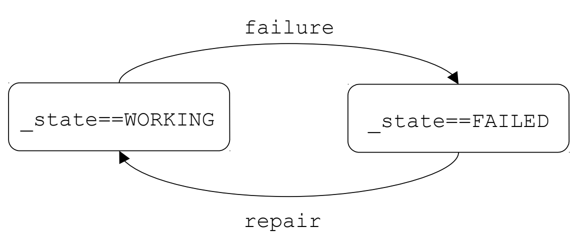


Figure 1: Graphical representation of the guarded transition system for a repairable unit.

This modeling component is reused in many different models and often many times within a model. For this reason, it is declared as a class in the code of Figure 2 (lines 3–12). A class is an on-the-shelf modeling component. It can be instantiated as many times as necessary into models.

The state of a unit is represented by a variable named `_state` (declared line 4) that takes its value in the domain `RepairableUnitState` (declared line 1). This domain consists of the two symbolic constants `WORKING` and `FAILED`. Initially, the unit is working, so the attribute `init` of the variable `_state` is set to `WORKING`. This attribute indicates also that `_state` is a state variable.

As the unit is repairable, it has two transitions: a failure transition (declared line 10) that goes from the state working to the state failed and a repair transition (declared line 11)

```
1  domain RepairableUnitState {WORKING, FAILED}
2
3  class RepairableUnit
4    RepairableUnitState _state (init = WORKING);
5    event failure (delay = exponential(lambda));
6    event repair (delay = Dirac(tau));
7    parameter Real lambda = 1.0e-4;
8    parameter Real tau = 12;
9    transition
10     failure: _state==WORKING -> _state := FAILED;
11     repair: _state==FAILED -> _state := WORKING;
12 end
```

Figure 2: AltaRica 3.0 code for the guarded transition system pictured Figure 1.

that goes the reverse way. In the code of Figure 2, events `failure` and `repair` (declared respectively lines 5 and 6) are associated with respectively a delay obeying the inverse of a negative exponential distribution of parameter `lambda` and constant delay `tau`. These parameters are declared respectively lines 7 and 8. AltaRica 3.0 provides several built-in distributions, like Dirac, exponential, Weibull as well as empirical distributions (given as a list of points).

AltaRica 3.0 comes with a standard library that declares a number of classes such as `RepairableUnit` to represent various types of components. These classes encode behavioral patterns. We shall see now examples of structural patterns.

2.2 Composition

The class `RepairableUnit` involves no flow variable and therefore no assertion. Flow variables are mainly a mean to connect components. The composition of two (or more) guarded transition systems is actually a guarded transition system.

Formally, let $M_1 : \langle V_1, E_1, T_1, A_1, \iota_1 \rangle$ and $M_2 : \langle V_2, E_2, T_2, A_2, \iota_2 \rangle$ be two guarded transition systems. Then $M_1 \otimes M_2$ is simply the guarded transition system $\langle V, E, T, A, \iota \rangle$ such that $V = V_1 \cup V_2$, $E = E_1 \cup E_2$, $T = T_1 \cup T_2$, $A = A_2 \circ A_1$ and $\iota = \iota_2 \circ \iota_1$.

This means that models can be obtained by composing smaller models.

The structural decomposition pattern is a typical example of composition of components. It is widely used to represent functional and physical breakdowns of systems, see e.g. [37]. It works as follows.

Modeling Pattern 2 (Structural decomposition) *A structural decomposition consists of:*

- *A hierarchy of components, each with its own behavior represented by means of states and transitions. Each component in the hierarchy has a number of parents (possibly none) and a number of children (possibly none).*
- *A description of interactions between parents and children components, which is represented by flow variables and assertions.*

Components are thus organized into a directed acyclic graph. Components are nodes of the graph. Links parent-child are edges of the graph. The graph is acyclic, therefore there is *no loop in the hierarchy. If each component has at most one parent, then the hierarchy is a tree.*

Fault trees are examples of models that obey the structural decomposition pattern. Consider for instance the fault tree pictured Figure 3.

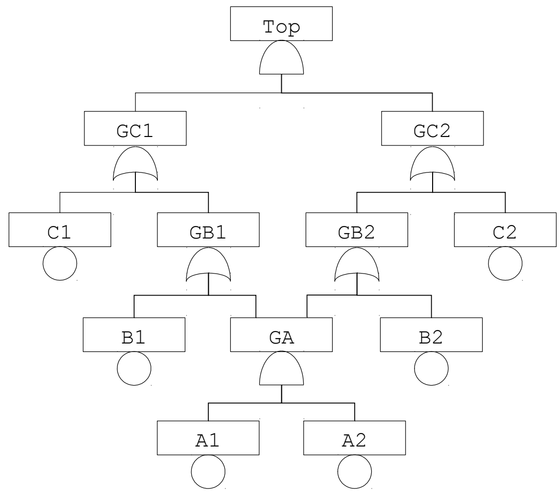


Figure 3: A fault tree.

This fault tree is made of 6 basic events: `A1`, `A2`, `B1`, `B2`, `C1` and `C2` and 6 internal events: `Top` (the top event), `GC1`, `GC2`, `GB1`, `GB2` and `GA`. Events are actually organized into a directed acyclic graph (and not just a tree because the event `GA` has two parents: `GB1` and `GB2`).

Fault trees are a degenerated case of structural decomposition: first, only leaves of the hierarchy carry out behaviors; second, the communication goes only from children to parents. For this reason, it is not necessary to represent internal events explicitly as components, assertions suffice.

Assuming basic events of our tree represent failed states of repairable units (as described above), the AltaRica 3.0 code for this fault tree could be as given in Figure 4.

This code declares first the class `BasicEventForRepairableUnit` to encode basic events (lines 1-6). This class inherits from the class `RepairableUnit` (via the `extends` clause, line 2). It means that a `BasicEventForRepairableUnit` is a `RepairableUnit` with additional properties. In this case, a Boolean flow variable `failed` is declared (line 3). Its default value is set to `false` via the attribute `reset`. This attribute indicates also that `failed` is a flow variable. Finally, the assertion line 5 tells how the value of the flow variable is calculated from the value of the state variable. The flow variable `failed` exports the state of the component and will be used to communicate this state to the parent components.

The fault tree itself is encoded as prototype, i.e. a modeling component with a unique occurrence (lines 8-22). Prototypes are introduced by the keyword `block`.

The block `FaultTree` declares as many instances of `BasicEventForRepairableUnit` as there are basic events in the fault tree (lines 9-12) and as many Boolean flow variables as there are internal events (line 13). The assertion

```
1 class BasicEventForRepairableUnit
2   extends RepairableUnit;
3   Boolean failed(reset = false);
4   assertion
5     failed := _state==FAILED;
6 end
7
8 block FaultTree
9   BasicEventForRepairableUnit A1, A2(lambda=1.0e-5);
10   BasicEventForRepairableUnit B1, B2(lambda=2.0e-5);
11   BasicEventForRepairableUnit C1, C2(lambda=1.0e-7,
12      tau=24);
13   Boolean Top, GA, GB1, GB2, GC1, GC2 (reset = false);
14   assertion
15     Top := GC1 and GC2;
16     GC1 := C1.failed or GB1;
17     GC2 := C2.failed or GB2;
18     GB1 := B1.failed or GA;
19     GB2 := B2.failed or GA;
20     GA  := A1.failed and A2.failed;
21   observer Boolean failed = Top;
22 end
```

Figure 4: AltaRica 3.0 code for the fault tree pictured Figure 3.

consists then simply in one equation per internal event, telling how the value of the internal event is calculated from the values of its children (line 15-20). Variables declared inside a component (prototype or instance of class) are accessed via the dot notation: `C1.failed` denotes the variable `failed` of the component `C1`.

In the code of Figure 4, the values of parameters `lambda` and `tau` of basic events are redefined at instantiation, when necessary. It is even possible to change the distributions themselves at instantiation. This makes it possible to design generic classes for components.

Note that it would be possible to use other types for basic events. We may for instance imagine that some of the components are not repairable. More complex schemes will be presented in the next section.

Note also that the fault tree has been declared as a prototype. There are actually little chance, if any, for this modeling component to be reused somewhere. It is definitely specific to the system under study, conversely to the components `RepairableUnit` and `BasicEventForRepairableUnit` that are reused several times in this and other models. We shall see that some constructs are only available on prototypes, making the distinction between prototypes and classes of further interest.

The code declares the Boolean observer `failed` (line 21). Observers do not play any role in the behavioral description of the model. They are updated after each transition firing. They are used to define and to calculate performance indicators. Observers can be seen as the interface of the model for the assessment tools. They make it possible to optimize the code, for instance by removing variables, without disturbing the assessment of indicators.

2.3 Semantics

The semantics of a guarded transition system $M : \langle V = S \uplus F, E, T, A, \iota\rangle$ is defined as the set of its possible executions. To define formally the executions, we need to introduce the notions of state and schedule.

A state σ of M is valuation of variables of V verifying $\sigma(F) = A(\sigma(S))$.

A schedule of M is a function from T to $\mathbb{R}^+ \cup \{+\infty\}$. A schedule Γ is compatible with a state σ and a date $d \in \mathbb{R}^+$ if for all transitions $t : G \xrightarrow{e} P$ of T, $d \leq \Gamma(t)$ if $G(\sigma) = true$ and $\Gamma(t) = +\infty$ otherwise.

Intuitively, an execution of M is a sequence:

$$\langle \sigma_0, d_0, \Gamma_0\rangle \xrightarrow{t_1} \langle \sigma_1, d_1, \Gamma_1\rangle \xrightarrow{t_2} \ldots \xrightarrow{t_n} \langle \sigma_n, d_n, \Gamma_n\rangle$$

where $n \geq 0$, the σ_i's are states of M, the d_i's are dates i.e. non negative real numbers verifying $0 = d_0 \leq d_1 \leq \ldots \leq d_n$, each Γ_i is a schedule compatible with σ_i and d_i and finally the t_i's are transitions of M.

Formally, the set of valid executions is defined recursively as follows.

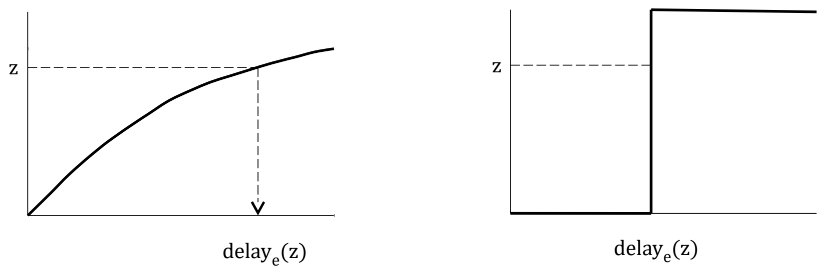
The empty execution $\langle \sigma_0, 0, \Gamma_0\rangle$ is a valid execution if $\sigma_0(S) = \iota$, $\sigma_0(F) = A(\iota)$ and the schedule Γ_0 is such that for all transitions $t : G \xrightarrow{e} P$ of T:

- $\Gamma_0(t) = delay_e(t)$ for some $z \in [0, 1]$ if $G(\iota) = true$.
- $\Gamma_0(t) = +\infty$ if $G(\iota) = false$.

Now, if $\Lambda = \langle \sigma_0, d_0, \Gamma_0\rangle \xrightarrow{t_1} \ldots \xrightarrow{t_n} \langle \sigma_n, d_n, \Gamma_n\rangle$, $n \geq 0$, is a valid execution, then so is the execution $\Lambda \xrightarrow{t_{n+1}} \langle \sigma_{n+1}, d_{n+1}, \Gamma_{n+1}\rangle$ if the following conditions hold, assuming $t_{n+1} = G_{n+1} \xrightarrow{e_{n+1}} P_{n+1}$.

- $G_{n+1}(\sigma_n) = true$.
- $\sigma_{n+1} = A(P_{n+1}(\sigma_n))$, i.e. the firing of the transition t_{n+1} is performed in two steps: first, state variables are updated by means of the action P_{n+1} of the transition, then flow variables are updated by means of the assertion A.
- $d_{n+1} = \Gamma_n(t_{n+1})$ and there is no transition t of T such that $\Gamma_n(t) < \Gamma_n(t_{n+1})$.
- Γ_{n+1} is obtained from Γ_n by applying the following rules to all transitions $t : G \xrightarrow{e} P$ of T.
  - If $G(\sigma_{n+1}) = true$, then:
    - If $G(\sigma_n) = true$ and $t \neq t_{n+1}$, i.e. if the transition was already scheduled, then $\Gamma_{n+1}(t) = \Gamma_n(t)$, i.e. the previous firing date is kept.
    - Otherwise, $\Gamma_{n+1}(t) = d_{n+1} + delay_e(z)$ for some $z \in [0, 1]$, i.e. a new firing date is chosen.
  - If $G(\sigma_{n+1}) = false$, then $\Gamma_{n+1}(t) = +\infty$.

Note that executions are fully determined by the choices of the z's.



(a) Exponential delay.

(b) Dirac delay.

Figure 5: Delays as inverse of cumulative probability distributions.

Example

In our example, there are two types of delays: exponential and Dirac. Figure 5 shows how their values are calculated from a number $z \in [0, 1]$.

A possible execution could be as follows.

At time 0, all state variables take the value `WORKING`, all the flow variables take the value `false` and the six transitions `failure` are fireable. The initial schedule could be for instance as follows.

`A1.failure`:	5849.45	`B1.failure`:	7068.84	`C1.failure`:	3629.63
`A2.failure`:	7406.81	`B2.failure`:	227.47	`C2.failure`:	1037.08

As `B2.failure` has the earliest firing date, it is fired (at 227.47). After its firing, `B2._state` takes the value `FAILED`, `B2.failed`, `GB2` and `GC2` take the value `true`, and the transition `B2.repair` gets fireable and is scheduled at $227.47 + 12 = 239.47$. The other variables and transitions stay unchanged.

`A1.failure`:	5849.45	`B1.failure`:	7068.84	`C1.failure`:	3629.63
`A2.failure`:	7406.81	`B2.repair`:	239.47	`C2.failure`:	1037.08

As `B2.repair` has the earliest firing date, it is fired (at 239.47). After its firing, all state variables take back the value `FAILED`, all flow variables take back the value `false`, and the transition `B2.failure` gets fireable again and is scheduled for instance at $239.47 + 2788.84 = 3128.21$. The other transitions stay unchanged.

`A1.failure`:	5849.45	`B1.failure`:	7068.84	`C1.failure`:	3629.63
`A2.failure`:	7406.81	`B2.failure`:	3128.21	`C2.failure`:	1037.08

And so on.

2.4 Another fundamental architectural pattern

Another very common and fundamental architectural pattern consists in representing a system as a hierarchy of communicating blocks:

Modeling Pattern 3 (Hierarchical block diagram) *A hierarchical block diagram consists of two types of blocks:*

- *Basic blocks that represent components of the system and that carry out the behavior of these components.*
- *Internal blocks that may contain other internal and basic blocks.*

Blocks may have ports that represent their interface. Ports are connected via links.

Such hierarchical block diagrams are pervasive in model-based systems engineering. They are used in modeling languages as diverse as Matlab/Simulink ([38]), Modelica ([32]), Lustre ([39]), SysML ([40]), and many others.

As an illustration, consider the system pictured Figure 6.

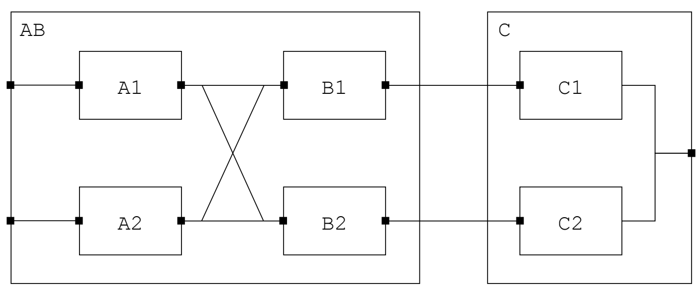


Figure 6: A hierarchical block diagram.

This system is made of the two subsystems `AB` and `C` in series. The subsystem `AB` is made of four basic units `A1`, `A2`, `B1` and `B2`. The subsystem `C` is made of two basic units `C1` and `C2`. Components are connected from left to right: the inputs of `A1` and `A2` are connected to two inputs. The inputs of `B1` and `B2` are connected to both outputs of `A1` and `A2`. The inputs of `C1` and `C2` are connected respectively to the outputs of `B1` and `B2`. Finally, the output of the system aggregates the outputs of `C1` and `C2`.

If the basic units of this hierarchical block diagram are repairable units described by our modeling pattern 1, then the above description is just a (hierarchical) reliability block diagram.

The AltaRica 3.0 code for this diagram is given Figure 7.

This code starts by declaring a class `BasicBlockForRepairableUnit` for basic blocks. This class is similar to the class `BasicEventForRepairableUnit`, except it describes the transfer function between the input of the block and its output.

The system is declared as a block (lines 8–31). The block `Plant` declares two sub-blocks `AB` and `C` (respectively lines 10–15 and 16-20). These sub-blocks declare in turn instances of the class `BasicBlockForRepairableUnit`: `A1`, `A2`, `B1` and `B2` for the sub-block `AB` (line 11), `C1` and `C2` for the sub-block `C` (line 18). The declaration of `C1` and `C2` modifies the value of the parameter `lambda`. Assertions at system and sub-block levels realize the connections.

It would have been indeed possible to declare classes for sub-blocks `AB` and `C` and then to instantiate them into the block `Plant`. However, as these sub-blocks have a unique occurrence, it is more natural to declare them as prototypes. Moreover, prototypes make it

```
class BasicBlockForRepairableUnit
  extends RepairableUnit;
  Boolean input, output(reset = false);
  assertion
    output := _state==WORKING and input;
end

block Plant
  Boolean input1, input2 (reset = true);
  block AB
    BasicBlockForRepairableUnit A1, A2, B1, B2;
    assertion
      B1.input := A1.output or A2.output;
      B2.input := A1.output or A2.output;
  end
  block C
    BasicBlockForRepairableUnit C1, C2
    (lambda = lambdaC);
    parameter Real lambdaC = 2.0e-6;
  end
  Boolean output (reset = false);
  assertion
    input1 := true;
    input2 := true;
    AB.A1.input := input1;
    AB.A2.input := input2;
    C.C1.input := AB.B1.output;
    C.C2.input := AB.B2.output;
    output := C.C1.output or C.C2.output;
  observer Boolean failed = not C.output;
end
```

Figure 7: AltaRica 3.0 code for hierarchical reliability block diagram pictured Figure 6.

possible to edit different levels of the hierarchy in the same view. For instance, it is possible to modify connections in the inner block `AB` while editing the outer block `Plant`. This is not possible with the class/instance mechanism because it is not allowed to modify a class from one of its instances.

2.5 Discussion

AltaRica 3.0 is agnostic: names are used to make clear what blocks, variables and events represent, but a model would be assessed in the same way if objects would be named X, Y, Z. Moreover, the naming conventions we used here are just a personal choice of the authors (e.g. we write symbolic constants with capital letters, prefix state variables with an underscore, capitalize names of blocks and classes…).

AltaRica 3.0 is primarily a textual language. Graphical representations, such as those of Figures 1, 3 and 6 are an excellent communication means. They made the success of languages such as Matlab/Simulink [38]. We use them as much as possible. However, as soon as the model gets complex, they cannot embed all of its details. Moreover, the same model can be looked at from different angles, therefore with different graphical representations. In other words, the text is the reference, even though modeling environments make it possible to create models by dragging and dropping graphical representations (icons). The interested reader can look at reference [41] for an interesting discussion on the pragmatics of graphical modeling.

3 Behavioral patterns

In this section, we shall review some very common patterns used to represent the behaviors of components, beside the `RepairableUnit` pattern we have already seen.

3.1 Advanced failure models

The `RepairableUnit` pattern corresponds well to continuously monitored components. In process industry, there are however many components that are only periodically inspected or tested, e.g shutdown valves that are tested by means of partial stroke. Safety standards and best practice guides like [2, 42] pay a lot of attention to this kind of components. Hence our next modeling pattern.

Modeling Pattern 4 (Periodically tested component) *A periodically tested component is a component:*

– *That alternates operation and test phases;*

– *Whose failures are only revealed thanks to the tests;*

It is moreover assumed that the phases are organized on a calendar base, i.e. they have a fixed duration and their chaining is decided once for all. This (possible) approximation makes it possible to consider each component independently.

The behavior of periodically tested components is conveniently described by multiphase Markov chains like the one pictured Figure 8 (indeed when occurrences of failures and repairs obey the Markovian hypothesis).

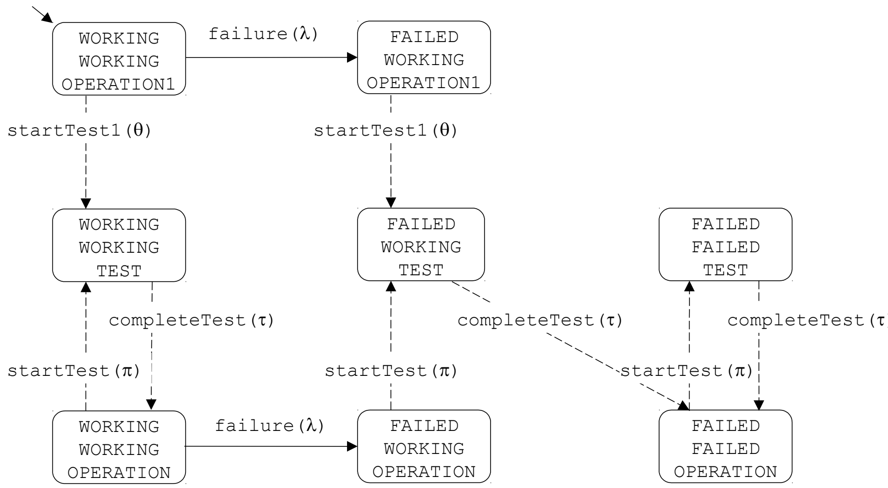


Figure 8: Multiphase Markov chain for a periodically tested component.

The component switches periodically from operation to test phases. The operation and test phases last respectively π and τ hours. A first operation phase that lasts θ is introduced so to be able to represent staggered tests when considering a multi-components system. The component fails with a failure rate λ. It is assumed that the component cannot fail during the test. The state of the component is thus represented by means of three values (represented from top to bottom on the figure): its actual state, its observed state and its phase. Stochastic transitions are represented with plain arrows while deterministic transitions are represented with dashed arrows.

The AltaRica 3.0 code for such a component is given Figure 9. This code is a direct translation of the multiphase Markov chain. It obeys the same principles as the one for repairable units, except that the state of the component is now described by means of three state variables: `_actualState`, `_observedState` and `_phase`.

The class `PeriodicallyTestedComponent` can be instantiated everywhere convenient (possibly after some adjustments), in particular in models built according to the structural decomposition pattern (pattern 2) and the hierarchical block diagram pattern (pattern 3).

More advanced models for periodically tested components are discussed in [43]. These models can be easily implemented in AltaRica 3.0 as well.

In the above pattern, the two phases alternate periodically after an initial phase. Many articles have been published that study phased mission systems, i.e. systems whose mission is decomposed into a finite number of successive phases, e.g. taxiing, take-off, cruise, landing and taxiing again for a commercial airliner. The first report work on this topic the authors could find was one by Esary and Ziehms, see [44, 45]. In phased-mission systems, a component may be active only in some phases. Moreover, the causes of failure of the system may be different from one phase to the other. It is in general assumed that if the component is failed in one phase, then it remains failed in the subsequent phases. In any cases, it is fairly easy to adjust the above pattern to represent components of phased-mission systems.

```
domain State {WORKING, FAILED}
domain Phase {OPERATION1, OPERATION, TEST}

class PeriodicallyTestedComponent
  State _actualState (init = WORKING);
  State _observedState (init = WORKING);
  Phase _phase (init = OPERATION1);
  event failure (delay = exponential(lambda));
  event startTest1 (delay = Dirac(theta));
  event startTest (delay = Dirac(pi));
  event completeTest (delay = Dirac(tau));
  parameter Real lambda = 1.0e-6;
  parameter Real theta = 2188;
  parameter Real pi = 4378;
  parameter Real tau = 2;
  transition
    failure: _actualState==WORKING and _phase!=TEST
        -> _actualState := FAILED;
    startTest1: _phase==OPERATION1 -> _phase := TEST;
    startTest: _phase==OPERATION -> _phase := TEST;
    completeTest: _phase==TEST -> {
        _phase := OPERATION;
        _observedState := _actualState;
        }
end
```

Figure 9: AltaRica 3.0 code for multiphase Markov chain pictured Figure 8.

3.2 Dependent Behaviors

So far, the patterns we proposed make “only” possible to represent combinatorial models such as fault trees or reliability block diagrams in a unified way and provide a mean to extend failure models for basic components. In a word, we stayed in the realm of combinatorial models.

Combinatorial models are not sufficient when dependencies amongst events have to be represented (although some approximations can be proposed, see e.g. [46]). Dependencies typically arise when a component is in cold or warm redundancy of another.

This leads us to our next pattern.

Modeling Pattern 5 (Warm redundancy) *A component is said in warm redundancy (of another one) if the following holds.*

- *The component is initially in standby mode. It is put in operation when there is a demand, i.e. typically when another component fails.*
- *The component may fail both in standby mode and in operation, although with different failure distributions, as a component in operation is more stressed than a component in standby.*
- *The attempt to start the component may be unsuccessful (failure on demand).*

– *The component may be repaired.*

– *The component is put back in standby mode after a repair or when it is not demanded anymore.*

The above definition is quite general and can be indeed tuned to more specific cases, such as cold redundancies where the component is assumed to be safe when it is in standby mode.

Figure 10 shows the states/transitions diagram representing the behavior of such component. Figure 11 gives the corresponding AltaRica 3.0 code.

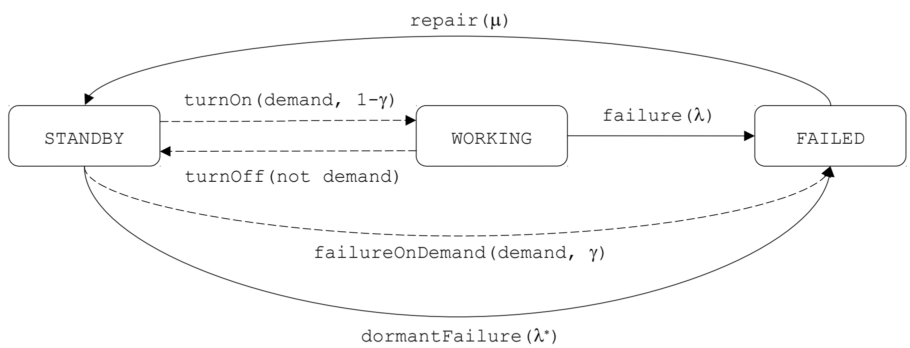


Figure 10: States/transitions diagram for a component in warm redundancy.

This model involves a new type of stochastic transitions. When the component is in standby mode and gets demanded, technically when the input flow variable `demand` becomes true, the component reacts immediately: the two transitions `turnOn` and `failureOnDemand` (defined lines 20 and 22) become fireable. As they are labeled with events associated null (Dirac) delays (defined lines 7 and 9), one of them is instantaneously fired. The choice is non-deterministic: the attribute `expectation` can be seen as a weight. Assume transitions labeled with events $e_1, \ldots e_n$ are in competition to be fired at the same exact date and that the `expectation`'s of these events are respectively $w_1, \ldots w_n$. Then, the transition labeled with the event e_i is fired with a probability $w_i / \sum_{j=1}^{n} w_j$. In our model, the weights of `turnOn` and `failureOnDemand` are determined by the parameter γ which can be seen as the probability of failure on demand.

Components in cold/warm redundancy are used in combination with other components, typically repairable components as defined by the pattern 1. They can be also used in combination to implement redundancies between subsystems, as illustrated by the code given Figure 12.

This code represents a system made of two trains: the main train and a spare train in warm redundancy. Each train is made of two components in series. The idea is then simple: the main train is demanded if it is not failed, while the spare train is demanded if the main train is failed. The same principle applies with any two subsystems.

Note that initially the two components of the main train are in standby. But, as the train is not failed, it is demanded. Therefore the two components are attempted to start.

```
domain State {STANDBY, WORKING, FAILED}

class ComponentInWarmRedundancy
  State _state (init = STANDBY);
  Boolean demand (reset = false);
  event turnOn (delay = Dirac(0),
  expectation = 1-gamma);
  event failureOnDemand (delay = Dirac(0),
    expectation =  gamma);
  event turnOff (delay = Dirac(0));
  event failure (delay = exponential(lambda));
  event dormantFailure(delay = exponential(lambdaStar));
  event repair (delay = exponential(mu));
  parameter Real gamma = 0.02;
  parameter Real lambda = 1.0e-4;
  parameter Real lambdaStar = 1.0e-6;
  parameter Real mu = 0.1;
  transition
    turnOn: _state==STANDBY and demand ->
      _state:=WORKING;
    failureOnDemand: _state==STANDBY and demand ->
      _state:=FAILED;
    turnOff: _state==WORKING and not demand ->
      _state:=STANDBY;
    failure: _state==WORKING -> _state := FAILED;
    dormantFailure: _state==STANDBY ->
      _state := FAILED;
    repair: _state==FAILED -> _state := STANDBY;
end
```

Figure 11: AltaRica 3.0 code for a component in warm redundancy.

```
class Train
  ComponentInWarmRedundancy A, B;
  Boolean demand, failed(reset = false);
  assertion
    A.demand := demand;
    B.demand := demand;
    failed := A._state==FAILED or B._state==FAILED;
end

block Plant
  Train mainTrain, spareTrain;
  Boolean failed(reset=false);
  assertion
    failed := mainTrain.failed and spareTrain.failed;
    mainTrain.demand := not mainTrain.failed;
    spareTrain.demand := mainTrain.failed;
end
```

Figure 12: AltaRica 3.0 code for trains in warm redundancy.

3.3 Synchronized Behaviors

In the above "warm redundancy" pattern, when the main component fails, the spare component is immediately attempted to start. The failure of the former and start (or failure on demand) of the latter occur at the same date, although in order. There are cases however where two (or more) transitions occur exactly at the same time, without any ordering among them, typically because they encode local effects of a global event.

As an illustration, consider two units `A` and `B` sharing the same resource `R`. `R` can be for instance the repair team or a spare part. When the unit `A` gets the resource `R`, `R` becomes simultaneously unavailable for `B`. `A` and `R` change of state at exactly the same time. This idea is formalized by the following pattern.

Modeling Pattern 6 (Shared Resource) *This pattern consists in two or more units sharing the same set of resources.*

- *When one of the units gets one of the resources, this resource becomes simultaneously unavailable for the other units.*
- *Symmetrically, when the unit releases the resource, the resource becomes simultaneously available for the other units.*

AltaRica 3.0 provides the powerful concept of synchronization to encode the "shared resource" pattern. This concept has been originally introduced by Arnold and Nivat to describe interactions between concurrent processes [47]. It is generalized in AltaRica 3.0.

Consider for example a system consisting in two units and a maintenance team. Assume that the maintenance team can work on only one unit at a time. If a unit fails, it must therefore wait until that the maintenance team is free to start being repaired. Once repaired, it releases the resource, i.e. the maintenance team. A possible code for this system is given Figure 13.

```
domain UnitState {WORKING, FAILED, REPAIR}

class Unit
  UnitState _state(init=WORKING);
  event failure(delay = exponential(lambda));
  event startRepair, completeRepair;
  parameter Real lambda = 1.0e-4;
  transition
    failure: _state==WORKING -> _state:=FAILED;
    startRepair: _state==FAILED -> _state:=REPAIR;
    completeRepair: _state==REPAIR -> _state:=WORKING;
end

domain MaintenanceTeamState {STANDBY, WORKING}

class MaintenanceTeam
  MaintenanceTeamState _state(init=STANDBY);
  event startJob, completeJob;
  transition
    startJob: _state==STANDBY -> _state:=WORKING;
    completeJob: _state==WORKING -> _state:=STANDBY;
end

block System
  Unit U1, U2(startRepair.hidden = true,
    completeRepair.hidden = true);
  MaintenanceTeam M(startJob.hidden = true,
    completeJob.hidden = true);
  event startRepair1, startRepair2(delay = Dirac(0));
  event completeRepair1,
    completeRepair2(delay = exponential(mu));
  parameter Real mu = 0.025;
  transition
    startRepair1:!U1.startRepair & !M.startJob;
    startRepair2:!U2.startRepair & !M.startJob;
    completeRepair1:!U1.completeRepair & !M.completeJob;
    completeRepair2:!U2.completeRepair & !M.completeJob;
end
```

Figure 13: AltaRica 3.0 code for units sharing a maintenance team.

In this code, the definitions of classes `Unit` and `MaintenanceTeam` are no surprise. The block `System` declares events (lines 29-31). The corresponding transitions (lines 34-37) synchronize transitions of components.

For instance, the transition `startRepair1` results of the simultaneous firing of transitions `startRepair` of unit `U1` and `startJob` of the repair team. The modality `!` prefixing the event name makes the corresponding transition mandatory: the global transition `startRepair1` can be fired only if local transitions `U1.startRepair` and `M.startJob` are fireable. In other words, this transition is just equivalent to the following one.

```
startRepair1: U1._state==FAILED and M._state==STANDBY
  -> { U1._state := REPAIR;
       M._state := WORKING;
      }
```

It is however much more convenient to declare local effect locally.

Instances of units and maintenance team are declared with the attribute `hidden` of their events `startRepair` and `completeRepair`, respectively `startJob` and `completeJob`, set to `true` (lines 26-28). This indicates that these events cannot be fired individually, but only through synchronizations.

We shall see now a more complex synchronization scheme, involving the modality `?`.

Common cause failures are an important contributor to the risk in many technical systems, e.g. in nuclear power plants, see e.g. [48]. There are many types of common cause failures. We shall restrict our attention to the following category (other types of common cause failures can be represented in AltaRica 3.0 as well, but with other modeling patterns).

Modeling Pattern 7 (Common Cause Failures) *A common cause failure is an event:*

– *Impacting simultaneously several basic components;*

– *Failing all impacted components that are not already failed.*

Fire or flooding are typical such events. Strictly speaking, common cause failures as defined above cannot be described at basic component level since they involve several of them. Their description in AltaRica 3.0 relies on a very important construct of the language, namely the synchronization of events.

The AltaRica 3.0 code that describes a common cause failure acting on three repairable units is given Figure 14.

```
1 block Plant
2   RepairableUnit A, B, C;
3   event ccf (delay = exponential(ccfRate));
4   parameter Real ccfRate = 1.0e-6;
5   transition
6     ccf: ?A.failure & ?B.failure & ?C.failure;
7 end
```

Figure 14: AltaRica 3.0 code describing a common cause failure to three repairable units.

The event and the transition representing the common cause failure are declared in the block that aggregates the three components (lines 3 and 6). The transition line 6 synchronizes the three events `A.failure`, `B.failure` and `C.failure`, i.e. attempts to fire these three events simultaneously. The modality `?` indicates that event it prefixes is fired only if possible (a modality `!` would indicate that event it prefixes is mandatory). The transition labeled with `ccf` is fireable if at least one the three events is fireable, i.e. if at least one of the guards of the individual transitions is satisfied in the current state. Its firing consists in performing actions of individual transitions that can be fired. Eventually, it is thus equivalent to the following code.

```
ccf: A._state==WORKING or B._state==WORKING or
  C._state==WORKING -> {
    if A._state==WORKING then A._state := FAILED;
    if B._state==WORKING then B._state := FAILED;
    if C._state==WORKING then C._state := FAILED;
    }
```

3.4 Discussion

The patterns presented in this section involve components that can be in more than two states, but that still fundamentally either working or failed. It is sometimes interesting to take into account degradation levels (and of course the transitions between these levels), as discussed in the abundant literature on so-called multistate systems, see e.g. [17, 18]. Tools like HiP-HOPS provide means to describe such systems [19]. Rauzy and Yang proposed recently a unifying algebraic framework, see [49].

Multistate systems are straightforward to represent in AltaRica 3.0 (for this reason we do not introduce here a specific pattern). Figure 15 shows for instance how levels of degradation can be composed thanks to dedicated operators.

```
domain WDFState {WORKING, DEGRADED, FAILED}

operator WDFState WDFMin(WDFState in1, WDFState in2)
  switch {
    case in1==FAILED or in2==FAILED: FAILED
    case in1==WORKING and in2==WORKING: WORKING
    default: DEGRADED
    }
end

operator WDFState WDFMax(WDFState in1, WDFState in2,
                          WDFState out)
  switch {
    case in1==FAILED and in2==FAILED: FAILED
    case in1==WORKING or in2==WORKING: WORKING
    default: DEGRADED
    }
end
```

Figure 15: AltaRica 3.0 code for aggregation operators working on the `WDFState` domain.

Once declared, operators (and functions) can be used in actions of transitions and assertions. Operators (and functions) in AltaRica 3.0 are similar to macro-instructions of programming languages. The switch expression is equivalent to a cascade of if-then-else expression: the conditions are looked at in turn and the first one which is satisfied determines the value of the expression.

We presented here binary operators for a ternary logic for the sake of the simplicity. It is of course possible to extend them to any multivalued logic (and any number of arguments), see e.g. [50] for a review.

4 Architectural Patterns

In this section, we present three common architectural patterns to illustrate the importance of this concept and the ability of AltaRica 3.0 to implement advanced models.

4.1 Reliability Networks

So far, all the models we considered are data-flow, i.e. that the information propagates only one way between components. There are however risk/safety assessment models in which the information can circulate in either directions, depending on the states of the components. Reliability networks belong to this category of models. Although much less popular than fault trees and reliability block diagrams, they have focused important research efforts, see e.g. [51, 52] for two monographs on this topics. The reason is that power and water distribution networks as well as several other types of infrastructures are typically analyzed as reliability networks.

A reliability network is a graph with two kinds of nodes: source nodes that produce something (power, information, …) and target nodes that consume and redistribute this something. Nodes of both types may fail according to some probability distribution. Edges are assumed to work perfectly.

As an illustration, consider the network pictured Figure 16. This network is made of the two source nodes `S1` and `S2` and six target nodes `T1`, …`T6`.

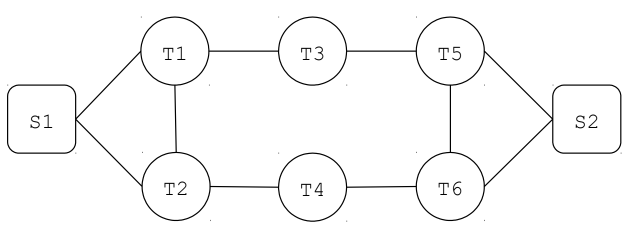


Figure 16: A reliability network.

In this network, edges are bidirectional. The information can thus circulate in different directions, depending on the state of components. For instance, the node `T6` can be powered by the node `S1` via the nodes `T2` and `T4`. In case `T2` and `S2` are failed, `T4` can be powered by the node `S1` via the nodes `T1`, `T3`, `T5` and finally `T6`.

Note that we do not loose any generality by assuming edges are perfect: an imperfect edge between two nodes `A` and `B` can be represented as an imperfect node `N` and two perfect edges form `A` to `N` and from `N` to `B`.

A typical question one may ask on such a networks is “what is the probability that a specific node is a powered at time t”, i.e. what is the probability that there exists a working path from one of the operating source nodes to that particular node.

Answering this question raises difficult algorithmic issues. Specialized algorithms have been developed [53, 54]. As of today, AltaRica 3.0 is the only modeling language embedding natively mechanisms to represent and assess reliability networks, see [28] for in-depth explanations.

The solution is surprisingly simple and elegant. It relies on the following pattern.

Modeling Pattern 8 (Reliability Network) *The reliability network pattern consists in a network of components such that:*

– *Each component has a unique Boolean input and a unique Boolean output.*

– *The input of a component is defined as a coherent Boolean formula of the outputs of the other nodes.*

Whether each component is reachable, i.e. has at least one of its inputs true, depends on the global state of the network.

Recall that a Boolean formula is coherent if it is built only with “and”, “or” and “k-out-of-n” connectives.

Assuming that each node of the reliability network pictured Figure 16 is repairable, this network can be represented in AltaRica 3.0 using the class `BasicBlockForRepairableUnit` defined Section 2.4 and suitable assertions, as shown Figure 17.

```
block ReliabilityNetwork
  BasicBlockForRepairableUnit S1, S2, T1, T2, T3, T4,
    T5, T6;
  assertion
    S1.input := true;
    S2.input := true;
    T1.input := S1.output or T2.output or T3.output;
    T2.input := S1.output or T1.output or T4.output;
    T3.input := T1.output or T5.output;
    T4.input := T2.output or T6.output;
    T5.input := S2.output or T3.output or T6.output;
    T6.input := S2.output or T4.output or T5.output;
end
```

Figure 17: AltaRica 3.0 code for the reliability network of Figure 16.

The assertion of this model is “looped” i.e. that variables depends eventually on each other. AltaRica 3.0 fixpoint mechanism to update the value of flow variables after each transition firing is able to deal efficiently with such dependencies [28].

4.2 Production Systems

Many industrial processes can be represented as a hierarchical and oriented network of treatment units. Each unit treats the production of upstream units and transfers it to downstream units. The amount of products an unit is treating depends on its intrinsic capacity, on the production of upstream units and the demand of downstream units. There is thus a two-ways dependency between units.

When the organization of production units is complex, optimizing the production is also a complex problem, involving typically the determination of the maximum flow that can travel from source nodes to target nodes. Relatively efficient algorithms exists to do so, e.g. the Ford-Fulkerson algorithm [55], but calling such an algorithm after each transition firing would be probably of a prohibitive cost. When the production plant can be decomposed hierarchically, the following pattern, that has been introduced and studied in [34], makes it possible to solve the problem elegantly.

Modeling Pattern 9 (Production Tree) *A production tree consists in a hierarchy (a tree) of components such that:*

- *Each component has a certain production capacity that depends on its intrinsic capacity, its state, and the capacities of its subcomponents.*
- *Each component has a certain production objective and is in charge of determining the production objectives of its subcomponents so to reach its own objective.*
- *The production objective of a component never exceeds its production capacity.*

As an illustration, consider a sub-system `S` of a certain system consisting of two subsystems `C` and `D`. Assume that production policy consists in allocating the production objective in priority to `C`. Assume finally that `S` does not fail at the same rate when it is producing something and when it does not produce anything.

An AltaRica 3.0 code describing the system `S` is given 18.

The block `S` inherits from the class `ComponentInWarmRedundancy` (pattern 5). It composes the two blocks `C` and `D`. Finally, it declares the parameter `intrisicCapacity` and the two flow variables `capacity` and `objective` to represent the corresponding quantities.

The capacity of `S` should be 0 if `S` is failed, and the minimum of its intrinsic capacity and the sum of the capacities of `C` and `D` otherwise. This is directly translated into the assertion line 13.

`S` must split the production objective between `C` and `D`, giving the priority to `C`. This is again directly translated into the assertion line 19.

Finally, `S` must be started if its objective is not null and stopped otherwise, which is done by setting the inherited variable `demand` accordingly (line 27).

It is very easy to adjust the code of Figure 18 to represent hierarchical components that compose more than two components and/or that implement other production policies such as production with units in series, or units in parallel (and a pro-rata allocation of the production objective).

It is also worth to notice that by changing real flows to Boolean flows it is possible to use the production tree pattern to encode dynamic fault trees, see e.g. [15], as well as Boolean driven Markov process [16] applying the translation proposed in [56].

```
block S
  extends ComponentInWarmRedundancy;
  block C
    // description of C
  end
  block D
    // description of D
  end
  parameter Real intrisicCapacity = 110.0;
  Real capacity(reset=0);
  Real objective(reset=0);
  assertion
    capacity := switch {
        case _state==FAILED: 0.0
        case C.capacity+D.capacity>intrisicCapacity:
            intrisicCapacity
        default: C.capacity+D.capacity
        };
    if objective>C.capacity then {
        C.objective := C.capacity;
        D.objective := objective-C.capacity;
        }
    else {
        C.objective := objective;
        D.objective := 0;
        }
    demand := objective>0;
end
```

Figure 18: AltaRica 3.0 code implementing a priority node in a production tree.

4.3 Monitored Systems

Maintenance policies have a strong impact on operational performance of systems, for at least two reasons: first, a high reliability can only be achieved by means of maintenance interventions; second these interventions, that often require to shutdown at least partly the production, are in practice a major contributor to the system down times. One of the today's challenges of process industry is to move from calendar- or time-based maintenance policies to condition-based maintenance policies, in order to reduce down-times and beyond costs. If assessing the formers is well mastered, assessing the latter raises difficult modeling issues.

Our next and last pattern is dedicated to the analysis of monitored systems. It has been recently introduced and studied in [57].

Modeling Pattern 10 (Monitored System) *The monitored system pattern consists in organizing the model into four modules:*

- *A module dedicated to the description of the behavior of units of the system.*
- *A module that calculates the actual state of the system from the states of the units.*
- *A module that represents the diagnostic made on the state of the system.*
- *Finally, a controller that makes operational decisions, including the scheduling of maintenance interventions, based on the diagnostic and that broadcasts orders to units so that they implement these decisions.*

This pattern not only helps to organize models of monitored systems, but more fundamentally provides a methodology to reason about these systems.

As an illustration, consider a 2-out-of-3 system made of periodically tested units. The system is thus failed if at least two out of the three units are failed. In this case, the production must be stopped immediately and a maintenance intervention scheduled. There may be a significant delay between the time a maintenance intervention is scheduled and the time it starts effectively. The production is stopped during maintenance intervention. If one unit is failed, the system is degraded. In this case, the production can go on, but a maintenance intervention must be scheduled. As a unit is out of order during a test, it is dangerous to test a unit while another one is failed because only the third unit remains in service. It is however necessary to take that risk, because it is the only mean to detect that a second unit may be failed, and therefore that the production must be stopped immediately. For the same reason, it is better to stagger tests of the units.

Modeling this kind of systems is by no means easy. One has to handle intricated stochastic and deterministic events, to represent both the actual state of the system and the diagnostic made on this state, to represent decision rules for maintenance interventions and to implement them effectively in units, and so on. Additional difficulties come when there are different types of maintenance interventions, applying on different groups of units.

The implementation in AltaRica 3.0 of the condition-diagnostic-decision pattern on our example is performed into two steps.

The first step consists in augmenting the periodically-inspected-component pattern so to represent maintenance interventions. To do so, it suffices to add a phase `MAINTENANCE` to the component as well as two transitions: a transition `startMaintenance` from a phase other than `MAINTENANCE` to the phase `MAINTENANCE` and a transition `completeMaintenance` from the phase `MAINTENANCE` to the phase `OPERATION1`. The transition `completeMaintenance` may be slightly different depending on whether the component is as good as new after the maintenance or not.

The second step consists in describing the system as a whole, as shown Figures 19 and 20.

The block `Plant` declares four sub-blocks corresponding to the four modules of the pattern. We have here the illustration of a very general principle: the "good" architecture for a model does not necessarily mimics the architecture of the system, at least its physical architecture.

The block `Units` (line 4) gathers the declaration of individual units. Parameters `tau` (duration of tests) and `pi` (interval between two tests) are redefined. Parameters `theta`'s (date of the first test) are adjusted so to implement staggered tests.

The block `ActualStateCalculator` (line 12) calculates the actual state of the plant from the `_actualState`'s of the units. To do so, it aggregates these variables by means of the clause `embeds`. Aggregation should be seen as a reference to an element declared outside the aggregating block. This element is accessed by means of an absolute or a relative path. In the code of the example, it is an absolute path: the keyword `main` denotes the outer most block (or class) of the current hierarchy. Therefore, `main.Units.U1._actualState` denotes the variable `_actualState` of the unit `U1` of the block `Units` of the block `Plant`. It would have been possible to use instead the relative path `owner.Units.U1._actualState`. The keyword `owner` denotes the parent block.

The block `DiagnosticCalculator` (line 20) makes a diagnostic on the state of the plant from the `_observedState`'s of the units. It aggregates these variables.

The block `Controller` stops the production when the plant is diagnosed failed (line 15). It schedules and executes maintenance interventions. To do so, it synchronizes its own events `startMaintenance` and `completeMaintenance` with those of units (lines 18 and 24).

Observers `dangerousState`, `safeState` and `maintenanceOnGoing` make it possible to count the number of times one of the states they describe has been encountered, to get the sojourn times in these states and other indicators of interest.

This pattern concludes our presentation of AltaRica 3.0.

5 Conclusion

In this article, we gave a snapshot of the expressive power of the AltaRica 3.0 modeling language by showing how to encode ten very common modeling patterns in probabilistic risk and safety analyses of complex technical systems. Modeling patterns are not only a very efficient means to architect and to document models, they are more fundamentally a way to reason about systems under study. We distinguished here between behavioral patterns, that stand at component level, and architectural patterns, that stand at model level. This taxonomy may be imperfect and will probably evolve in the future. Behavioral patterns are rather well mastered as they are used in a variety of contexts and with other modeling formalisms than AltaRica. On the contrary, the study of architectural patterns is still in its infancy (or its adolescence). The authors are deeply convinced that, together with the introduction of machine learning techniques to obtain degradation profile of components, they are a game changer in probabilistic risk and safety analyses.

```
domain Diagnostic {WORKING, DEGRADED, FAILED}

block Plant
  block Units
    PeriodicallyTestedUnit U1(tau=6, theta=2190-3*tau,
      pi=4380-tau);
    PeriodicallyTestedUnit U2(tau=6, theta=2190-2*tau,
      pi=4380-tau);
    PeriodicallyTestedUnit U3(tau=6, theta=2190-1*tau,
      pi=4380-tau);
  end
  block ActualStateCalculator
    embeds main.Units.U1._actualState as s1;
    embeds main.Units.U2._actualState as s2;
    embeds main.Units.U3._actualState as s3;
    Boolean failed(reset=false);
    assertion
      failed := #(s1==FAILED,s2==FAILED,s3==FAILED)>=2;
  end
  block DiagnosticCalculator
    embeds main.Units.U1._observedState as o1;
    embeds main.Units.U2._observedState as o2;
    embeds main.Units.U3._observedState as o3;
    Diagnostic diagnostic(reset=WORKING);
    assertion
      diagnostic := switch {
        case #(o1==FAILED, o2==FAILED, o3==FAILED)>=2:
          FAILED
        case #(o1==FAILED, o2==FAILED, o3==FAILED)==1:
          DEGRADED
        default: WORKING
        };
  end
  block Controller
    ...
  end
  observer Boolean dangerousState =
  ActualStateCalculator.failed and
    not Controller.productionStopped;
  observer Boolean safeState =
    Controller.productionStopped;
  observer Boolean maintenanceOnGoing =
    Controller.maintenanceOnGoing;
end
```

Figure 19: AltaRica 3.0 code implementing the condition-diagnostic-decision pattern.

```
block Controller
  embeds main.Units.U1 as U1;
  embeds main.Units.U2 as U2;
  embeds main.Units.U3 as U1;
  embeds main.DiagnosticCalculator.diagnostic as
    diagnostic;
  Boolean productionStopped(init=false);
  Boolean maintenanceOnGoing(init=false);
  event stopProduction(delay=Dirac(0));
  event startMaintenance(delay=Dirac(delta));
  event completeMaintenance(delay=Dirac(mu));
  parameter Real delta = 720;
  parameter Real mu = 48;
  transition
    stopProduction:
    diagnostic==FAILED and not productionStopped ->
        productionStopped := true;
    startMaintenance:
       !U1.startMaintenance
     & !U2.startMaintenance
     & !U3.startMaintenance
     & diagnostic!=WORKING and not maintenanceOnGoing ->
        maintenanceOnGoing := true;
    completeMaintenance:
       !U1.completeMaintenance
     & !U2.completeMaintenance
     & !U3.completeMaintenance
     & maintenanceOnGoing -> {
        maintenanceOnGoing := false;
        productionStopped := false;
        }
end
```

Figure 20: AltaRica 3.0 code implementing the controller.

An integrated modeling environment for AltaRica 3.0 (AltaRica Wizard) is currently under development as joint effort of the OpenAltaRica team at IRT-SystemX (Paris, France) and the Norwegian University of Science and Technology. Industrial partners (Airbus, Safran and Thales) support this project. A versatile set of assessment tools has been developed, which includes:

– A step by step simulator making possible to play "what-if" scenarios and to validate models. This simulator implements abstract interpretation techniques so to simulate faithfully stochastic and timed executions [58].

– A compiler of AltaRica models into fault trees. This compiler relies on advanced algorithmic techniques [59]. Fault trees are then assessed with XFTA [60], which is one of the most efficient available calculation engines.

– A compiler of AltaRica models into Markov chains. This compiler produces Markov chains that approximate the original model while staying of reasonable sizes [61]. Markov chains are then assessed with Mark-XPR, as very efficient calculation engine [62].

– A generator of critical sequences (still under development at the time we write these lines).

– A stochastic simulator. Stochastic simulation is itself a versatile tool to assess complex models, see e.g. [63].

These tools make the AltaRica 3.0 technology versatile and efficient. They make it possible cross-verification. They prefigure what will be the next generation of modeling environments for the assessment of operational performance of complex technical systems.

Tutorial material, courses for both primary and continuing education are also under development.

In a word, the AltaRica 3.0 technology is now mature or close to be. Deploying it in industry will indeed take time, but everything is ready for. With that respect, training analysts is of course a key issue. New technologies are often deployed by new generation of engineers. This will be probably the case for the model-based approach in risk and safety analyses. Using high-level modeling languages such as AltaRica 3.0 requires some familiarity with information technology in general and programming in particular. New generations of engineers have undoubtedly these skills.

Safety analyses are otfen thought as a cost. They are performed because regulation bodies require them. As we are moving from a product-centered industry to a service-oriented industry, their role is however changing: they are used not only to assess whether systems are safe and available enough to be operated, but also to establish contractual relations between providers and clients of services. In other words, they are becoming an economical driver. With that respect, we can expect that the industrial impact of new technologies such as AltaRica 3.0 goes much beyond traditional safety analyses.

Acknowledgments

The authors would like to thank the reviewers of this article for their nice and useful comments.

References

[1] Norman C. Rasmussen. Reactor Safety Study. An Assessment of Accident Risks in U.S. Commercial Nuclear Power Plants. Technical Report WASH 1400, NUREG-75/014, U.S. Nuclear Regulatory Commission, Rockville, MD, USA, October 1975.

[2] International iec standard iec61508 - functional safety of electrical/electronic/programmable safety-related systems (e/e/pe, or e/e/pes). Standard, International Electrotechnical Commission, Geneva, Switzerland, April 2010.

[3] Guidelines and methods for conducting the safety assessment process on civil airborne systems and equipment. Standard, Society of Automotive Engineers, Warrendale, Pennsylvania, USA, July 2004.

[4] Iso26262 functional safety - road vehicle. Standard, International Standardization Organization, Geneva, Switzerland, 2012.

[5] Hiromitsu Kumamoto and Ernest J. Henley. *Probabilistic Risk Assessment and Management for Engineers and Scientists*. IEEE Press, Piscataway, N.J., USA, 1996.

[6] John D. Andrews and Robert T. Moss. *Reliability and Risk Assessment (second edition)*. ASM International, Materials Park, Ohio 44073-0002, USA, 2002.

[7] Marvin Rausand and Arnljot Høyland. *System Reliability Theory: Models, Statistical Methods, and Applications, 2nd Edition*. Wiley-Blackwell, Hoboken, New Jersey, USA, January 2004.

[8] Steven Epstein and Antoine Rauzy. Can We Trust PRA? *Reliability Engineering and System Safety*, 88(3):195–205, 2005.

[9] Marco Ajmone-Marsan, Gianfranco Balbo, Giuseppe Conte, Susanna Donatelli, and Giuliana Franceschinis. *Modelling with Generalized Stochastic Petri Nets*. Wiley Series in Parallel Computing. John Wiley and Sons, New York, NY, USA, 1994.

[10] Jean-Pierre Signoret, Yves Dutuit, Jean-Pierre Cacheux, Cyrille Folleau, Stéphane Collas, and Philippe Thomas. Make your petri nets understandable: Reliability block diagrams driven petri nets. *Reliability Engineering and System Safety*, 113:61–75, 2013.

[11] Pierre David, Vincent Idasiak, and Frederic Kratz. Reliability study of complex physical systems using sysml. *Reliability Engineering & System Safety*, 95(4):431–450, 2010.

[12] Nataliya Yakymets, Yupanqui Munoz Julho, and Agnes Lanusse. Sophia framework for model-based safety analysis. In *Actes du congrès Lambda-Mu 19 (actes électroniques)*, Dijon, France, October 2014. Institut pour la Maîtrise des Risques.

[13] Pierre Mauborgne, Samuel Deniaud, Eric Levrat, Eric Bonjour, Jean-Pierre Micaëlli, and Dominique Loise. Operational and system hazard analysis in a safe systems requirement engineering process âŁ“ application to automotive industry. *Safety Science*, 87:256–268, August 2016.

[14] Faida Mhenni, Jean-Yves Choley, Nga Nguyen, and Christophe Frazza. Flight control system modeling with sysml to support validation, qualification and certification. *IFAC-PapersOnLine*, 49(3):453–458, 2016.

[15] Joanne Bechta Dugan, Salvatore J. Bavuso, and Marks A. Boyd. Dynamic fault-tree models for fault-tolerant computer systems. *IEEE Transactions on Reliability*, 41(3):363–377, September 1992.

[16] Marc Bouissou and Jean-Louis Bon. A new formalism that combines advantages of fault-trees and markov models: Boolean logic-driven markov processes. *Reliability Engineering and System Safety*, 82(2):149–163, 2003.

[17] Anatoly Lisnianski and Gregory Levitin. *Multi-State System Reliability*. Quality, Reliability and Engineering Statistics. World Scientific, London, England, 2003.

[18] Bent Natvig. *Multistate Systems Reliability Theory with Applications*. Wiley, Hoboken, NJ, USA, 2010.

[19] Yiannis Papadopoulos, Martin Walker, David Parker, Erich Rüde, Rainer Hamann, Andreas Uhlig, Uwe Grätz, and Rune Liend. An approach to optimization of fault tolerant architectures using hip-hops. *Journal of Engineering Failure Analysis*, 18(2):590–608, March 2011.

[20] Matthias Güdemann and Frank Ortmeier. A framework for qualitative and quantitative model-based safety analysis. In *Proceedings of the IEEE* 12^{th} *High Assurance System Engineering Symposium (HASE 2010)*, pages 132–141, San Jose, CA, USA, 2010. IEEE.

[21] Marc Bouissou, Henri Bouhadana, Marc Bannelier, and Nathalie Villatte. Knowledge modelling and reliability processing: presentation of the FIGARO language and of associated tools. In Johan F. Lindeberg, editor, *Proceedings of SAFECOMP'91 – IFAC International Conference on Safety of Computer Control Systems*, pages 69–75, Trondheim, Norway, 1991. Pergamon Press.

[22] Gérald Point and Antoine Rauzy. AltaRica: Constraint automata as a description language. *Journal Européen des Systèmes Automatisés*, 33(8–9):1033–1052, 1999.

[23] Marta Kwiatkowska, Gethin Norman, and David Parker. Prism 4.0: Verification of probabilistic real-time systems. In *Proceedings 23rd International Conference on Computer Aided Verification (CAVâŁ™11)*, volume 6806 of *LNCS*, pages 585–591, New York, NY, USA, 2011. Springer.

[24] Marc Bouissou, Sibylle Humbert, Sabine Muffat, and Nathalie Villatte. KB3 tool: feedback on knowledge bases. In *Proceedings of European Safety and Reliability conference, ESREL'2002*, pages 114–119, Lyon, France, March 2002. ISdF.

[25] Marc Bouissou and Jean-Christophe Houdebine. Inconsistency detection in KB3 models. In *Proceedings of European Safety and Reliability conference, ESREL'2002*, pages 754–759, Lyon, France, March 2002. ISdF.

[26] Tatiana Prosvirnova, Michel Batteux, Pierre-Antoine Brameret, Abraham Cherfi, Thomas Friedlhuber, Jean-Marc Roussel, and Antoine Rauzy. The altarica 3.0

project for model-based safety assessment. In *Proceedings of 4th IFAC Workshop on Dependable Control of Discrete Systems, DCDS'2013*, pages 127–132, York, Great Britain, September 2013. International Federation of Automatic Control.

[27] Antoine Rauzy. Guarded transition systems: a new states/events formalism for reliability studies. *Journal of Risk and Reliability*, 222(4):495–505, 2008.

[28] Michel Batteux, Tatiana Prosvirnova, and Antoine Rauzy. Altarica 3.0 assertions: the why and the wherefore. *Journal of Risk and Reliability*, September 2017.

[29] Michel Batteux, Tatiana Prosvirnova, and Antoine Rauzy. *System Structure Modeling Language (S2ML)*. AltaRica Association, 2015. archive hal-01234903, version 1.

[30] Mauricio Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, New-York, USA, 1998.

[31] James Noble, Antero Taivalsaari, and Ivan Moore. *Prototype-Based Programming: Concepts, Languages and Applications*. Springer-Verlag, Berlin and Heidelberg, Germany, 1999.

[32] Peter Fritzson. *Principles of ObjectâŁ"Oriented Modeling and Simulation with Modelica 3.3: A CyberâŁ"Physical Approach*. Wiley-IEEE Press, Hoboken, NJ 07030-5774, USA, 2015.

[33] Christophe Kehren, Christel Seguin, Pierre Bieber, Charles Castel, Christian Bougnol, Jean-Pierre Heckmann, and Sylvain Metge. Architecture patterns for safe design. In *Electronic proceedings of 1st Complex and Safe Systems Engineering Conference (CS2E 2004)*, Arcachon, France, 2004. AAAF.

[34] Leïla Kloul and Antoine Rauzy. Production trees: A new modeling methodology for production availability analyses. *Reliability Engineering and System Safety*, 167:561–571, November 2017.

[35] Mark W. Maier. The art of systems architecting. 2009.

[36] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley professional computing series. Addison-Wesley, Boston, MA 02116, USA, October 1994.

[37] Daniel Krob. *CESAM: CESAMES Systems Architecting Method: A Pocket Guide*. CESAMES, http://www.cesames.net, January 2017.

[38] Harold Klee and Randal Allen. *Simulation of Dynamic Systems with MATLAB and Simulink*. CRC Press, Boca Raton, FL 33431, USA, February 2011.

[39] Nicolas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous dataflow programming language lustre. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.

[40] Sanford Friedenthal, Alan Moore, and Rick Steiner. *A Practical Guide to SysML: The Systems Modeling Language*. Morgan Kaufmann. The MK/OMG Press, San Francisco, CA 94104, USA, 2011.

[41] Hauke A. L. Fuhrmann. *On the Pragmatics of Graphical Modeling*. Kiel Computer Science Series. Book on Demand, Norderstedt, Germany, 2011.

[42] Iso/tr 12489:2013 petroleum, petrochemical and natural gas industries – reliability modelling and calculation of safety systems. Standard, International Organization for Standardization, Geneva, Switzerland, November 2013.

[43] Pierre-Joseph Cacheux, Stéphane Collas, Yves Dutuit, Cyrille Folleau, Jean-Pierre Signoret, and Philippe Thomas. Assessment of the expected number and frequency of failures of periodically tested systems. *Reliability Engineering and System Safety*, 118:61–70, October 2013.

[44] Harald Ziehms. *Reliability Analysis of Phased Missions*. PhD thesis, Naval Postgraduate School, 1974.

[45] J. D. Esary and Harald Ziehms. Reliability analysis of phased missions. In Richard E. Barlow, Jerry B. Fussel, and N.D. Singpurwalla, editors, *Reliability and Fault Tree Analysis: Theoretical and Applied Aspects of System Reliability and Safety Assessment*, pages 213–236, Philadelphia, PA, USA, 1975. SIAM Press.

[46] Jussi K. Vaurio. Making systems with mutually exclusive events analyzable by standard fault tree analysis tools. *Reliability Engineering and System Safety*, 74(1):75–80, 2001.

[47] André Arnold. *Finite Transition Systems*. C.A.R Hoare. Prentice Hall, Upper Saddle River, New Jersey, USA, 1994.

[48] Ali Mosleh, Dale M. Rasmuson, and F. M. Marshall. Guidelines on modeling common-cause failures in pra. Technical Report NUREG/CR-5485, U.S. Nuclear Regulatory Commission, 1998.

[49] Antoine Rauzy and Liu Yang. Finite degradation structures. article submitted Journal of Applied Logic, 2018.

[50] Grzegorz Malinowski. Many-valued logic and its philosophy. In Dov M. Gabbay and John Woods, editors, *Handbook of the History of Logic, The Many Valued and Nonmonotonic Turn in Logic*, volume 8, pages 13–94. Elsevier, Oxford, UK, 2009.

[51] Charlie J. Colbourn. *The combinatorics of network reliability*. Oxford University Press, New York, 1987.

[52] Douglas R. Shier. *Network Reliability and Algebraic Structures*. Oxford Science Publications, Oxford, England, 1991.

[53] Jean-Christophe Madre, Olivier Coudert, Henri Fraïssé, and Marc Bouissou. Application of a New Logically Complete ATMS to Digraph and Network-Connectivity Analysis. In *Proceedings of the Annual Reliability and Maintainability Symposium, ARMS'94*, pages 118–123, Annaheim, California, 1994. IEEE.

[54] Antoine Rauzy. A New Methodology to Handle Boolean Models with Loops. *IEEE Transactions on Reliability*, 52(1):96–105, 2003.

[55] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms (Second ed.)*. The MIT Press, Cambridge, MA, USA, 2001.

[56] Antoine Rauzy. Towards a sound semantics for dynamic fault trees. *Reliability Engineering and System Safety*, 142:184–191, October 2015.

[57] Yun Zhang, Anne Barros, and Antoine Rauzy. Finite degradation structures. article submitted Journal of Risk and Reliability, 2018.

[58] Michel Batteux, Tatiana Prosvirnova, and Antoine Rauzy. Enhancement of the altarica 3.0 stepwise simulator by introducing an abstract notion of time. In *Safe Societies in a Changing World, Proceedings of European Safety and Reliability Conference (ESREL 2018)*, Trondheim, Norway, June 2018.

[59] Tatiana Prosvirnova and Antoine Rauzy. Automated generation of minimal cutsets from altarica 3.0 models. *International Journal of Critical Computer-Based Systems*, 6(1):50–79, 2015.

[60] Antoine Rauzy. Anatomy of an efficient fault tree assessment engine. In R. Virolainen, editor, *Proceedings of International Joint Conference PSAM'11/ESREL'12*, Helsinki, Finland, June 2012.

[61] Pierre-Antoine Brameret, Antoine Rauzy, and Jean-Marc Roussel. Automated generation of partial markov chain from high level descriptions. *Reliability Engineering and System Safety*, 139:179–187, July 2015.

[62] Antoine Rauzy. An experimental study on six algorithms to compute transient solutions of large markov systems. *Reliability Engineering and System Safety*, 86(1):105–115, October 2004.

[63] Enrico Zio. *The Monte Carlo Simulation Method for System Reliability and Risk Analysis*. Springer Series in Reliability Engineering. Springer London, London, England, 2013.