# Improving performances of the AltaRica 3.0 stochastic simulator

Benjamin Aupetit, Michel Batteux, Antoine Rauzy, Jean-Marc Roussel

# Improving performances of the AltaRica 3.0 stochastic simulator

B. Aupetit & M. Batteux
*Technological Research Institute SystemX, Palaiseau, France**

A. Rauzy
*LGI, École Centrale Paris, France*

J-M. Roussel
*LURPA, École Normale Supérieure de Cachan, France*

ABSTRACT: This article presents the performance improvements we obtained on the AltaRica 3.0 stochastic simulator by using profiling techniques and testing it on a benchmark of models. This analysis showed that evaluating guards is one of the most time-consuming part of the execution. A selective update of the guards made it possible to improve significantly the performance.
This work is a part of the OpenAltaRica project, which aims at developing a complete set of tools for the high level modeling language AltaRica 3.0.

## 1 INTRODUCTION

### 1.1 *AltaRica in safety and reliability analyses*

The Model-Based approach for safety and reliability analysis is gradually winning the trust of engineers and is still an active domain of research. Safety engineers master "traditional" risk modeling formalism, such as Failure Mode, Effects and Criticality Analysis (FMECA), Fault Trees (FT), Event Trees (ET), Markov Processes. Efficient algorithms and tools are available. However, despite of their qualities, these formalisms share a major drawback: obtaining models from the specifications of the systems under study is time-consuming. As a consequence, models are hard to design and to maintain throughout the life cycle of systems. A small change in the specifications may require revisiting completely safety models, which is both resource consuming and error prone.

The high-level modeling language AltaRica has been created to tackle this problem (Prosvirnova et al. 2013). AltaRica models are made of hierarchies of reusable components. Graphical representations are associated with models, making them visually very close to Process and Instrumentation Diagrams.

*This research work has been carried out in the framework of the Technological Research Institute SystemX, and therefore granted with public funds within the scope of the French Program "Investissements d'Avenir"

An entirely new version of the language, named AltaRica 3.0, has been designed and a complete set of authoring and assessment tools is under development by the OpenAltaRica project (Batteux et al. 2014). These tools will be distributed as freeware, so to make them available to a wide audience.

### 1.2 *Stochastic simulation in safety and reliability analyses*

Stochastic simulation is an important tool to evaluate performances of systems (Cancela and Khadiri 1998, Baca 1993). In discrete event modeling formalisms, it is assumed that the system changes of state when and only when an event occurs. Stochastic delays are associated with events. It is therefore possible to perform Monte-Carlo simulations. The idea is to draw at pseudo-random a number of histories of the system and to make statistics on them. In this way, it is possible to evaluate the average number of times a given event has been fired, the number of times failure state has been reached, the mean time before failure state is reached, ... (see Zio 2013 for a complete monograph on Monte-Carlo simulations in the safety and reliability engineering framework).

The more histories are generated, the more accurate the statistics. It is therefore of primary importance to have a very efficient implementation of the basic simulation mechanisms.

### 1.3 *Stochastic simulation with AltaRica 3.0*

The AltaRica 3.0 stochastic simulator has been previously presented in Batteux and Rauzy 2013, which presents the performance improvements obtained by compiling a model specific simulator, and the possibility to define specific cumulative probability function when the classic ones (constant, dirac, exponential, Weibull, ...) are not sufficient to model a behavior.

The remainder of the article is organized as follows. Section 2 presents briefly the AltaRica 3.0 language and the OpenAltaRica project. Section 3 presents Guarded Transition Systems which are the underlying mathematical model of AltaRica 3.0. Section 4 describes the main features of the AltaRica 3.0 stochastic simulator and the simulation algorithm. Section 5 presents the results of preliminary performance analysis. Section 6 presents the improvements we obtained by enhancing this algorithm.

## 2 ALTARICA 3.0

### 2.1 *The OpenAltaRica project*

The OpenAltaRica project is carried out by the French Technological Research Institute SystemX with several academic and industrial partnerships.

Objectives of the OpenAltaRica project are on two ways. First, to federate a community around the AltaRica language, and more generally around Model-Based Safety Assessment (MBSA). The second objective is to make available an integrated platform based under AltaRica 3.0 and dedicated to safety and reliability analysis of complex systems. It will focus on the development of a coherent and complete set of reference tools for editing, animating and assessing AltaRica 3.0 models, on modeling methodology and on integration of safety analysis with other system engineering disciplines.

Technically the platform will contain four workshops: the AltaRica 3.0 workshop to design and assess AltaRica 3.0 models; the RAMS Open-PSA workshop to perform traditional low level safety and reliability analysis (Fault trees, Markov chains, ...); the GraphXica workshop to graphically animate and visualize AltaRica 3.0 models; and a workshop to compare models coming from different system engineering formalisms like SysML or Modelica. This future integrated platform will be made publicly available.

### 2.2 *AltaRica 3.0 Modeling Language*

The previous version of AltaRica modeling language, AltaRica Data-flow (Boiteau, Dutuit, Rauzy, & Signoret 2006), is a generalization of both Petri nets and block diagrams. From the former, it has imported the notions of states, events and guarded transitions whereas the latter inspired the notions of events synchronization, hierarchical description and flows. This last notion makes it possible to represent remote interactions in a simple way. It allows to easily model classic reliability mechanism, such as common cause failure or maintenance (preventive or corrective). However, located synchronizations cannot be captured and bidirectional flows circulating through a network cannot be modeled in a natural way. Moreover, it remains difficult to model looped systems. For these reasons AltaRica Data-Flow is not powerful enough to model such complex systems.

Thus, a new version of the language, so-called AltaRica 3.0, has been specified. It improves the previous version AltaRica Data-Flow into two directions. It provides new constructs to structure models (Prosvirnova & Rauzy 2014). Its semantic is based on the new underlying mathematical model: Guarded Transitions Systems. The new underlying formalism makes it possible to handle systems with instant loops and to define acausal components (components for which the input and output flows are decided at run time). It is much easier to model systems with bidirectional flows (e.g. electrical systems).

An AltaRica 3.0 model is structured in several components. Each component is a Guarded Transition System (described in next section) and exchange information with each other. To be assessed, the whole model is flattened into a unique Guarded Transition System.

## 3 GUARDED TRANSITION SYSTEMS

First introduced in Rauzy (2008), Guarded Transition Systems is a pivot formalism for Safety modeling and analyses. It generalizes classical formalisms, such as Reliability Block Diagrams, Markov chains and Petri Nets. The new semantics of instructions, presented in Prosvirnova & Rauzy (2012), makes it possible to represent components with bidirectional flows.

### 3.1 *Definition*

A Guarded Transition System, noted GTS, is a quintuple $\langle V, E, T, A, \iota \rangle$, where:

- $V = S \uplus F$ is a set of variables, divided into two disjoint sets $S$ of state variables describing the state of the system, and $F$ of flow variables allowing to exchange information between components (typically input-output links);
- $E$ is a set of symbols, called events;
- $T$ is a set of transitions denoted $e : G \to P$;
- $A$ is an assertion, i.e. an instruction built over $V$;
- $\iota$ is an assignment of variables of $V$, so-called an initial or default assignment.

A transition $e : G \to P$ is a triple $\langle e, G, P \rangle$ where $e \in E$ is an event, $G$ is a guard (i.e.: a boolean formula built over $V$) and $P$ is an instruction built over $V$, called an action or a post-condition. A transition $e$ :

$G \to P$ is said *fireable* in a given state $\sigma$ (i.e. for a given variable assignment $\sigma$) if its guard $G$ is satisfied in this state.

### 3.2 *Instructions*

Both assertion and action of transitions are described by means of instructions. An instruction can be:

- The empty instruction named $skip$;
- An assignment denoted $v := E$, where $v$ is a variable and $E$ is an expression built over variables from $V$;
- a conditional assignment denoted $if\ C\ then\ I$, where $C$ is a Boolean expression and $I$ is an instruction;
- a block of instructions denoted $\{I_1, \ldots, I_n\}$ of instructions.

State variables can occur as the left member of an assignment only in the action of a transition; whereas flow variables can only in the assertion. Instructions are interpreted in a slightly different way depending they are used in the actions or in the assertion. Let $\sigma$ be the variable assignment before the firing of the transition $e : G \to P$. Applying the instruction $P$ to the variable assignment $\sigma$ consists in calculating a new variable assignment $\tau$ as follows. We start with $\tau = \sigma$. Then,

- if $P$ is an empty instruction, then $\tau$ is left unchanged;
- if $P$ is an assignment $v := E$, then $\tau(v)$ is set to $\sigma(E)$. If the value of $v$ has been already modified and is different from the calculated one, then an error is raised;
- if $P$ is a conditional assignment $if\ C\ then\ I$ and $\sigma(C)$ is true, then the instruction $I$ is applied to $\tau$;
- if $P$ is a block of instructions $\{I_1, \ldots, I_n\}$ then instructions $I_1$, ..., $I_n$ are successively applied to $\tau$.

It is important to note that in the above mechanism, right hand side of assignments and conditional expressions are evaluated in the context of $\sigma$. Thus, the result does not depend on the order in which instructions of a block are applied. In other words, instructions of a block are applied in parallel. Let denote by $Update(P, \sigma)$ the variable assignment $\tau$ resulting from the application of the instruction $P$ to $\sigma$.

Let $A$ be the assertion and let $\tau$ be the variable assignment obtained after the application of the action of a transition. Applying $A$ consists in calculating a new variable assignment (of flow variables) $\pi$ as follows. We start by setting all state variables in $\pi$ to their values in $\tau$: $\forall v \in S \pi(v) = \tau(v)$. Let $D$ be a set of unevaluated flow variables, we start with $D = F$. Then,

- if $A$ is an empty instruction, then $\pi$ is left unchanged;
- if $A$ is an assignment $v := E$, then if $\pi(E)$ can be evaluated in $\pi$, i.e. all variables of $E$ have a value in $\pi$, then $\pi(v)$ is set to $\pi(E)$ and $v$ is removed from $D$. If the value of $v$ has been already modified and is different from the calculated one, then an error is raised;
- if $A$ is a conditional assignment $if\ C\ then\ I$ and $\pi(C)$ can be evaluated in $\pi$ and $\pi(C)$ is true, then the instruction $I$ is applied to $\pi$. Otherwise, $\pi$ is left unchanged;
- if $A$ is a block of instructions $\{I_1, \ldots, I_n\}$ then instructions $I_1$, ..., $I_n$ are repeatedly applied to $\pi$ until there is no more possibility to assign a flow variable.

If after applying $A$ to $\pi$ there are unevaluated variables in $D$, then all these variables are set to their default values $\forall v \in D, \pi(v) = reset(v)$ and $A$ is applied to $\pi$ in order to verify that all assignments are satisfied. If that is not true an error is raised. Let denote by $Propagate(A, \sigma)$ the variable assignment resulting from the application of the instruction $A$ to $\sigma$.

### 3.3 *Observers*

Observers are named expressions which depends on the variables of $V$ and are evaluated at each new state of the system. They cannot be used in transitions and assertion, i.e. they cannot be used to describe the behavior of a system. Rather, as their name indicates, they are quantities to be observed. Therefore, they are used to define what is useful to know in the model.

### 3.4 *Reachability graph*

Assume that $\sigma$ is the variable assignment just before the firing of a transition. Then, the firing of the transition transforms $\sigma$ into the assignment $Fire(e : G \to P, A, \sigma)$ defined as follows:

$$Fire(e : G \to P, A, \sigma) = Propagate(A, Update(P, \sigma))$$

GTS are implicit representations of Kripke structures, i.e. of graphs whose nodes are labeled by variable assignments and whose edges are labeled by events. More exactly, the semantics of a GTS $\langle V = S \uplus F, E, T, A, \iota \rangle$ is a Kripke structure, i.e. a graph $\Gamma = (\Sigma, \Theta)$, where $\Sigma$ is a set of variable assignments (also called states and representing nodes of the graph) and $\Theta$ is a set of triples $\langle s, e, q \rangle$, $s, q \in \Sigma$, $e \in E$ (representing transitions of the graph). $\Gamma$ is the smallest Kripke structure, such that the following is verified:

1. $\sigma_0$ is the initial state of the Kripke structure: i.e. $\sigma_0 = Propagate(A, \iota, \iota) \in \Sigma$;

2. if $\sigma \in \Sigma$ and $\exists t = \langle e, G, P \rangle \in T$, such that $\sigma(G) = TRUE$, then the state $\tau = Fire(P, A, \iota, \sigma)$ is in $\Sigma$ and the transition $(\sigma, e, \tau)$ is in $\Theta$,

The calculation of $\Gamma = (\Sigma, \Theta)$ may raise errors. A well designed GTS avoids this problem. The Kripke structure $\Gamma$ is also called a reachability graph.

### 3.5 *Timed/Stochastic Guarded Transition Systems*

A probabilistic time structure can be put on top of a GTS so to get timed/stochastic models. The idea is to associate to each event the following information:

- A delay that can be deterministic or stochastic and may depend on the state. When a transition labeled with the event becomes fireable at time $t$, a delay $d$ is calculated and the transition is actually fired at time $t + d$ if it stays fireable from $t$ to $t + d$;
- A weight, so called expectation, which is used to determine the probability that the transition is fired in case several transitions are fireable at the same date.

#### 3.5.1 *Timed Guarded Transition Systems*

Formally, a Timed GTS is a tuple $\langle V, E, T, A, \iota, delay \rangle$, where $\langle V = S \uplus F, E, T, A, \iota \rangle$ is a GTS and $delay : E \to \mathbb{R}_+$ is a function, that associates to each event a non-negative real number.

The semantics of Timed GTS can be defined in terms of executions. An execution is a sequence $(\sigma_0, d_0, t_0) \stackrel{e_0}{\to} (\sigma_1, d_1, t_1) \ldots (\sigma_{n-1}, d_{n-1}, t_{n-1}) \stackrel{e_{n-1}}{\to} (\sigma_n, d_n, t_n)$, where $t_i \in \mathbb{R}_+$ represents the date when the event $e_i$ occurs and the transition labeled by this event is fired. The step $(\sigma_i, d_i, t_i) \stackrel{e_i}{\to} (\sigma_{i+1}, d_{i+1}, t_{i+1})$ corresponds to the advancement of time and to the firing of the transition labeled by the event $e$.

#### 3.5.2 *Stochastic Guarded Transition Systems*

The timed interpretation of GTS does not specify how delays are calculated. Therefore, it encompasses the case where delays are stochastic. It remains however to define what stochastic means. To do so, we shall introduce the notion of oracle. The idea is to delegate all the "randomness" of a run to an oracle. In this way, the GTS stays purely deterministic but its behavior depends on the outcomes of the oracle. Oracles concentrate therefore the non-determinism of executions.

More formally, an *oracle* $o$ is an infinite sequence of real numbers comprised between 0 and 1 (included): i.e. $o : \mathbb{N} \to [0; 1] \subset \mathbb{R}$. The only operation available on an oracle is to consume its first element. This operation returns the first element and the remaining of the sequence, which is itself an oracle.

Finally, a Stochastic GTS is a tuple $\langle V = S \cup F, E, T, A, \iota, delay, expectation \rangle$, where:

- $\langle V = S \uplus F, E, T, A, \iota \rangle$ is a GTS;
- $delay$ is a function from events and oracles to non-negative real numbers.
- $expectation$ is a function from events to non-negative real numbers.

When several transitions are scheduled to be fired at the same date, one is picked at random by using the oracle and according to their expectations. The probability $p(e_k : G_k \to P_k)$ to fire the transition $e_k : G_k \to P_k$, is defined as follows:

$$p(e_k : G_k \to P_k) = \frac{expectation(e_k)}{\sum_{e_i : d_n(e_i) = 0} expectation(e_i)} \quad (1)$$

## 4 THE ALTARICA 3.0 STOCHASTIC SIMULATOR

### 4.1 *Software architecture*

The stochastic simulator is model specific to an AltaRica 3.0 model. It is obtained using compilation techniques via C++ classes. These techniques are used in order to improve the performances of simulation and were studied in Khuu 2008 and presented for this tool in Batteux and Rauzy 2013.

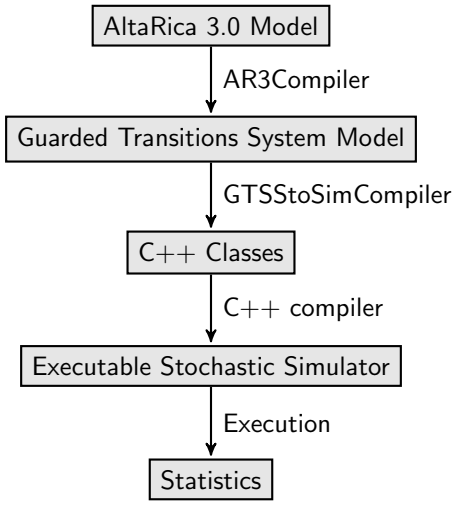


Figure 1: AltaRica 3.0 Stochastic Simulator architecture

Thus, the model specific stochastic simulator (to an AltaRica 3.0 model) is obtained through a tool chain graphically represented in figure 4.1. The different steps are:

1. The AltaRica 3.0 model is flattened in a GTS model with the AltaRica 3.0 Compiler;
2. The GTS model is then compiled with the GTS Stochastic Simulator Compiler into C++ classes, describing the model directly in the C++ language;
3. These C++ classes are then compiled, along with specific libraries, with a classic C++ compiler (e.g. gcc, visual c++, etc.), into an executable: the model specific stochastic simulator;

4. This model specific stochastic simulator can then be executed to obtain statistics on the generated histories.

### 4.2 *Statistics*

Statistics obtained from the generated histories are on events associated to transitions and observers. For transitions, the statistics are :

- The average number of times it has been fired and has been fireable per history;
- The time spent in a fireable state;
- The first date (on average) it has been fired in each history;
- The list of dates it has been fired;
- The list of dates it has been first fired for each history;
- The length of each fireable time interval.

For observers, the statistics are on the values taken during simulations:

- The average number of times this value was taken;
- The average time spend at this value;
- The first date (on average) this value was taken on each history;
- The length of each time interval this value was taken;
- The list of dates it has been first taken on each history.

### 4.3 *Simulation algorithm*

The implemented simulation algorithm takes as input the GTS model (as defined in section 3) and the implemented simulation part is summarized in Algorithm 1. A step, which correspond to the firing of a transition, is the execution of the three following parts:

1. Scheduler management (line 2 to 11)
   The scheduler retains the list of transitions and dates which are going to be fired. It must ensure that all transitions whose guard is true, and only those, are going to be fired.

2. Choice of a transition to fire (line 12)
   The choice of the next transition to fire depends on the nearest date in the scheduler: if there are several transitions scheduled at the same date, the expectation mechanism is used (see 3.5.2).

3. Firing of the transition (line 13 to 18)
   The chosen transition is removed from the scheduler. Its action is applied, updating state variables, which allow to obtain the new state of the model. Flow variables are evaluated using the assertion. The new value of each observer is then computed, using their expression and the new values of the variables.

**Algorithm 1** Simulation of a GTS model

```
 1: repeat
 2:     for all t ∈ T do
 3:         Update the guard value of t
 4:         if t fireable & t ∉ Scheduler then
 5:             Obtain a delay for the event of t from the oracle
 6:             Insert t in the Scheduler at this delay
 7:         end if
 8:         if t not fireable & t ∈ Scheduler then
 9:             Remove t from the Scheduler
10:         end if
11:     end for
12:     t ← (Next t ∈ Scheduler)
13:     Remove t from the Scheduler
14:     (e : G → P) ← t
15:     τ ← Update(P,σ)
16:     π ← Propagate(A,τ)
17:     σ ← π
18:     Compute the observers values
19: until a limit is reached
```

## 5 PERFORMANCE ANALYSIS

A set of test models was created. Table 1 indicates for each test model the different sizes: number of transitions, variables and observers. The models are of different configurations (serial, parallel, loop). The independence between the components varies (two components are dependent when a variable of one of the component is linked to the guard of a transition of the other component).

This set was then used to produce stochastic simulators specific to each model. Their executions were profiled using a profiling tool[1] to measure the time taken by each parts of the algorithm 1 during simulation. The average results in percentage of the execution time are shown table 2.

Table 1: Subset of the models used in benchmark

| Name | Trans. | State | Flow | Obs. |
|---|---|---|---|---|
| Component | 1 | 1 | 1 | 1 |
| SmallSystem | 4 | 2 | 2 | 1 |
| LandingSystem | 32 | 17 | 47 | 2 |
| EmergPowerSupply | 66 | 30 | 53 | 1 |
| Loop_10 | 22 | 11 | 22 | 1 |
| Busbar | 10 | 10 | 24 | 1 |
| MiniPlant | 8 | 4 | 8 | 1 |
| Network | 12 | 12 | 39 | 1 |
| NetworkSystem | 12 | 12 | 34 | 1 |
| SmallPumpingSys | 2 | 4 | 6 | 1 |
| PumpingSystem | 12 | 8 | 14 | 1 |
| SparePump | 10 | 3 | 6 | 1 |
| Serial_3_3 | 28 | 14 | 28 | 1 |
| Serial_10_10 | 224 | 112 | 224 | 1 |
| Plant_$n$ | $6n+2$ | $2n+1$ | $3n+1$ | 1 |

The results vary depending on the characteristics of the models.

[1] GNU gprof: https://sourceware.org/binutils/docs/gprof/

Table 2: Percentage range of execution time in each step

| Step | Percentage range |
|---|---|
| Scheduler management | 35-80% |
| including evaluation of the guards | 28-64% |
| Next transition selection | 4-5% |
| Execution of the instruction | 0.5-2% |
| Resolution of assertion | 15-35% |
| Update of the observers values | 1-5% |

The most important part in term of execution time is the management of the scheduler; more specifically the update of the values of guards. At each step of the simulation, the guard of each transition has to be evaluated. The number of transitions to check can be very high, depending on the model of the components, and the number of components in the model.

The resolution of the assertion is time consuming too: it consists of a set of conditional affectations (one for each flow variable), executed as many times as needed to obtain the value of each flow variable.

For this article, we focus on the update of the value of guards, which will be presented in the next section.

## 6 ENHANCEMENT OF THE PERFORMANCES

In order to understand how the update of the value of guards can be optimized, a model example is studied. The implementation of the optimization is then described, and its performances are measured.

### 6.1 *Model example*

In the set of models for the benchmark, Plant_$n$ is a class of models of a plant consisting of $n$ components connected in a serial way. Figure 2 is the AltaRica 3.0 model of the class Component. Each component is a repairable component (state variable vsCondition of type ComponentCondition can have 3 different values: working, failed or under repair) with an on/off switch (state variable vsMode of type PlantMode can have 2 different values: on or off). 6 transitions describe the behavior of the component: the state space with the transitions is represented figure 3. A component can fail only if it is producing, or at start (failure on demand), and it can start only if it is working. Assertion defines the condition for the component to produce : it is producing only if it is working, it is on ON and there is something to produce (i.e. there is an input).

One instance of this class (with $n = 4$) is defined in AltaRica 3.0 figure 5 and is represented figure 4. It is the model of a plant with 4 working stations: each station is modeled by a component of the class "Component". The production has to pass in each working station, from the first to the last, and there is no intermediate stock. The entry stock is infinite (the input of the first working station is always true) and the quantity to observe is the output of the plant, which is the output of the last working station (observer oOutput).

```
domain ComponentCondition {WORKING, FAILED, REPAIR}
domain PlantMode {ON, OFF}

class Component
  parameter Real lambda = 0.0001;
  parameter Real mu = 0.1;
  parameter Real nu = 5;
  parameter Real gamma = 0.01;
  ComponentCondition vsCondition (init = WORKING);
  PlantMode vsMode (init = OFF);
  Boolean vfIn (reset = false);
  Boolean vfOut (reset = false);
  PlantMode vfCommand (reset = OFF);
  event eFailure (delay = exponential(lambda));
  event eStartRepair (delay = exponential(mu));
  event eEndRepair (delay = nu);
  event eFailureOnDemand (delay = 0,
                          expectation = gamma);
  event eStart (delay = 0);
  event eStop (delay = 0);
  transition
    eFailure: vsCondition == WORKING
              and vsMode == ON
              and vfIn == true
              -> vsCondition := FAILED;
    eStartRepair: vsCondition == FAILED
                  -> vsCondition := REPAIR;
    eEndRepair: vsCondition == REPAIR
                -> vsCondition := WORKING;
    eFailureOnDemand: vsMode == OFF
                      and vfCommand == ON
                      and vsCondition == WORKING
                      -> vsCondition := FAILED;
    eStart: vsMode == OFF
            and vfCommand == ON
            and vsCondition == WORKING
            -> vsMode := ON;
    eStop: vsMode == ON
           and vfCommand == OFF
           -> vsMode := OFF;
  assertion
    vfOut := if vsCondition == WORKING
                and vsMode == ON
             then vfIn else false;
end
```

Figure 2: AltaRica 3.0 definition of a component of a plant

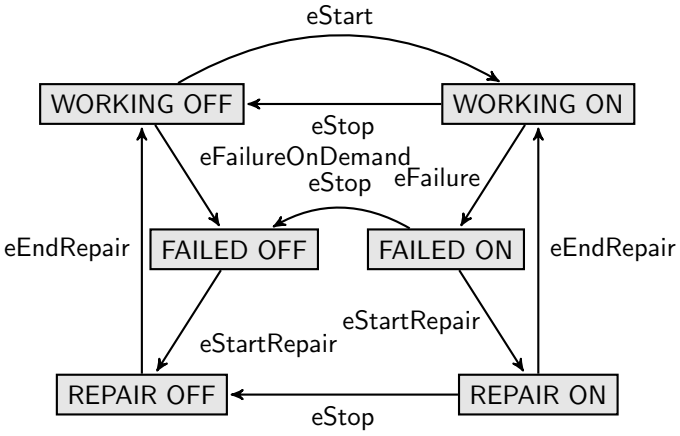


Figure 3: Component state-space with transitions

The on/off switch of each component is controlled by the plant, and correspond to the working hours of the plant (16 hours working per day).

### 6.2 *Matrix of dependencies*

At each step of the simulation (at each firing of a transition), the guard of each transition has to be evaluated.

A matrix of dependencies between guards of transi-

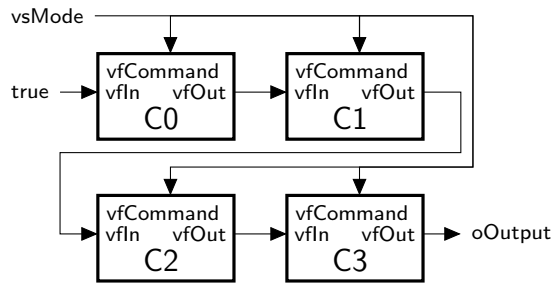


Figure 4: Representation of the model of Plant_4

```
block Plant_4
  PlantMode vsMode (init = ON);
  PlantMode vfCommand (reset = OFF);
  event eStart (delay = 57600);
  event eStop (delay = 28800);
  observer Boolean oOutput = C3.vfOut;
  Component C0(lambda = 0.0001, mu = 0.1);
  Component C1(lambda = 0.0001, mu = 0.1);
  Component C2(lambda = 0.0001, mu = 0.1);
  Component C3(lambda = 0.0001, mu = 0.1);
  transition
    eStart: vsMode == OFF -> vsMode := ON;
    eStop: vsMode == ON -> vsMode := OFF;
  assertion
    vfCommand := vsMode;
    C0.vfCommand := vfCommand;
    C1.vfCommand := vfCommand;
    C2.vfCommand := vfCommand;
    C3.vfCommand := vfCommand;
    C0.vfIn := true;
    C1.vfIn := C0.vfOut;
    C2.vfIn := C1.vfOut;
    C3.vfIn := C2.vfOut;
end
```

Figure 5: AltaRica 3.0 model of the system

tions and variables can be written for this model (figure 6). It indicates when a variable is in the guard formulae. This matrix is obviously sparse. The matrix of impact of transitions on variables (indicating when a variable is in an action of a transition) presents the same property. This remark is often true for models with dysfunctional behaviors: typically, transitions in a model for safety assessment are guarded only by variables of the corresponding component, and is affecting only this component variables and sometime some variables of close components.

The consequence is: a transition affects only a few variables that affect only a few guards. Therefore, it is not useful to update every guards at each step.

### 6.3 *Implementation*

The algorithm has been optimized to update guards only when it is useful. The simulation part of it, which takes as input the GTS model, is described in algorithm 2.

The chosen solution is to store the previous value of each variable, to compare it at the beginning of each step with the new value (line 3), and if the value has changed to mark the transition whose guard must be updated (line 4-5). The list of transitions whose guard depends of a variable is determined when the GTS model is compiled into C++ class.

| | vfCommand | C0.vsCondition | C0.vsMode | C0.vfIn | C1.vsCondition | C1.vsMode | C1.vfIn | C2.vsCondition | C2.vsMode | C2.vfIn | C3.vsCondition | C3.vsMode | C3.vfIn |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| C0.eF | 0 | **1** | **1** | **1** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| C0.eFOD | **1** | **1** | **1** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| C0.eSR | 0 | **1** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| C0.eER | 0 | **1** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| C0.Sta | **1** | **1** | **1** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| C0.Sto | **1** | 0 | **1** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| C1.eF | 0 | 0 | 0 | 0 | **1** | **1** | **1** | 0 | 0 | 0 | 0 | 0 | 0 |
| C1.eFOD | **1** | 0 | 0 | 0 | **1** | **1** | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| C1.eSR | 0 | 0 | 0 | 0 | **1** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| C1.eER | 0 | 0 | 0 | 0 | **1** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| C1.Sta | **1** | 0 | 0 | 0 | **1** | **1** | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| C1.Sto | **1** | 0 | 0 | 0 | 0 | **1** | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| C2.eF | 0 | 0 | 0 | 0 | 0 | 0 | 0 | **1** | **1** | **1** | 0 | 0 | 0 |
| C2.eFOD | **1** | 0 | 0 | 0 | 0 | 0 | 0 | **1** | **1** | 0 | 0 | 0 | 0 |
| C2.eSR | 0 | 0 | 0 | 0 | 0 | 0 | 0 | **1** | 0 | 0 | 0 | 0 | 0 |
| C2.eER | 0 | 0 | 0 | 0 | 0 | 0 | 0 | **1** | 0 | 0 | 0 | 0 | 0 |
| C2.Sta | **1** | 0 | 0 | 0 | 0 | 0 | 0 | **1** | **1** | 0 | 0 | 0 | 0 |
| C2.Sto | **1** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | **1** | 0 | 0 | 0 | 0 |
| C3.eF | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | **1** | **1** | **1** |
| C3.eFOD | **1** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | **1** | **1** | 0 |
| C3.eSR | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | **1** | 0 | 0 |
| C3.eER | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | **1** | 0 | 0 |
| C3.Sta | **1** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | **1** | **1** | 0 |
| C3.Sto | **1** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | **1** | 0 |

Figure 6: Matrix of dependencies between guards and variables

When needed, the guard value is updated if the transition is marked (and the mark is removed), or the previous value is used (line 11). If the guard has not been updated and its value is false, there is no need to check if it is in the scheduler, therefore this part of the algorithm is skipped (line 16-20).

### 6.4 *Performances*

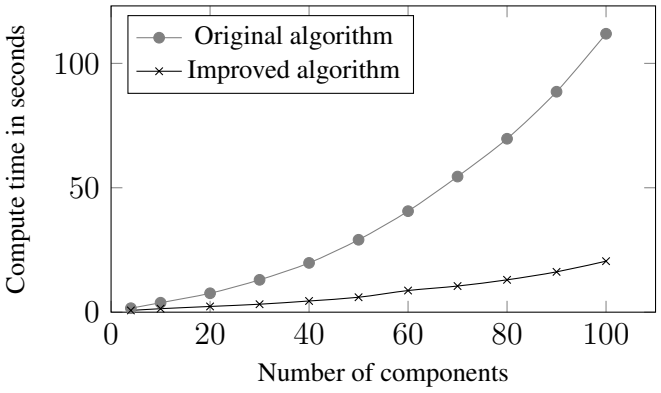


Figure 7: Performances improvement

A set of tests has been done with models of the Plant_$n$ class, with $n$ varying between 4 and 100 components. The computing time for the same amount of steps and generated histories has been measured, with the original algorithm and the improved one. The results are shown figure 7. The improvement is of about 85 percent. The performances improvement is highly dependent on the model:

**Algorithm 2** Simulation of a GTS model with optimized guard management

```
repeat
    for all v ∈ V do
        if value of v != previous value of v then
            for all t ∈ affected guard of v do
                Mark the guard of t to be updated
            end for
            previous value of v = value of v
        end if
    end for
    for all t ∈ T do
        Get the guard value of t (computed if marked)
        if t fireable & t ∉ Scheduler then
            Obtain a delay for the event of t from the oracle
            Insert t in the Scheduler at this delay
        end if
        if t updated & t not fireable then
            if t ∈ Scheduler then
                Remove t from the Scheduler
            end if
        end if
    end for
    t ← (Next t ∈ Scheduler)
    Remove t from the Scheduler
    (e : G → P) ← t
    τ ← Update(P,σ)
    π ← Propagate(A,τ)
    σ ← π
    Compute the observers values
until a limit is reached
```

- For a model whose matrix of dependencies is highly sparse (a lot of components who are independent), the computing time is highly improved;
- In contrast, for a model whose matrix of dependencies is dense (almost every transitions depends on almost every variables), this modification has almost no impact on the computing time. It is easily explained by the fact that at each step, almost every guard has to be updated, as previously;
- For a model whose transitions are mostly guarded by just one boolean variable, improvement is null: after guards to update are marked, instead of computing the new value of the guard (which is just a boolean variable), the algorithm check if the flag is present (i.e. if a boolean value is at true), which takes the same amount of time;
- For average models, with a mix of independent behavior (failure and repair) and dependent behavior (functional and common cause failure), the computing time has been improved of 30 to 60 percent, which roughly correspond to the previous step of evaluation of the guards.

## 7 CONCLUSION

Stochastic simulation is a very important tool in the safety and reliability framework. One of the key issue in stochastic simulation is to make the implementation of basic simulations as efficient as possible. The more these mechanisms are efficient, the more histories can be generated within the same computing time, and therefore the more accurate the results.

In this article, we showed how we improved the performance of the AltaRica 3.0 stochastic simulator by profiling the previous implementation. This analysis showed that the most time-consuming part of the simulation algorithm is when updating guard values. To reduce this cost, we modified the algorithm so to update only guards that are potentially impacted after each transition firing. This improvement saved from 30 to 85 percent of the computation time.

Future works include other improvements of this stochastic simulator, and the extension of this tool in order to perform stochastic model-checking.

## REFERENCES


Baca, A. (1993). Examples of monte carlo methods in reliability estimation based on reduction of prior information. *IEEE Transactions on Reliability 42*(4), 645–649.

Batteux, M. & A. Rauzy (2013). Stochastic simulation of AltaRica 3.0 models. In *Proceedings of the European Safety and Reliability Conference, ESREL 2013*.

Batteux, M., A. Rauzy, & P. Labrogère (2014, October). The OpenAltaRica project. In *Short & Tutorial Proceedings of the 4th International Symposium on Model-Based Safety Assessment, IMBSA 2014*, Munich (Germany).

Boiteau, M., Y. Dutuit, A. Rauzy, & J.-P. Signoret (2006). The AltaRica data-flow language in use: Assessment of production availability of a multistates system. *Reliability Engineering and System Safety 91*, 747–755.

Cancela, H. & M. E. Khadiri (1998). Series-parallel reductions in monte carlo network-reliability evaluation. *IEEE Transactions on Reliability 47*(2), 159–164.

Khuu, M.-T. (2008). *Contribution à l'accélération de la simulation stochastique sur des modèles AltaRica Data Flow*. Ph. D. thesis. Thèse de doctorat dirigée par Rauzy, Antoine Informatique Aix Marseille 2 2008.

Prosvirnova, T., M. Batteux, P.-A. Brameret, A. Cherfi, T. Friedlhuber, J.-M. Roussel, & A. Rauzy (2013, September). The altarica 3.0 project for model-based safety assessment. In *Proceedings of 4th IFAC Workshop on Dependable Control of Discrete Systems, DCDS 2013*, York (Great Britain). IFAC.

Prosvirnova, T. & A. Rauzy (2012, October). Système de transitions gardées : formalisme pivot de modélisation pour la sûreté de fonctionnement. In J. Barbet (Ed.), *Actes du congrès LambdaMu'18 (actes électroniques)*, Tours (France). IMdR.

Prosvirnova, T. & A. Rauzy (2014, October). The structural constructions of AltaRica 3.0. In *Actes du congrès LambdaMu'19 (actes électroniques)*, Dijon (France). IMdR.

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

Zio, E. (2013, January). *The Monte Carlo Simulation Method for System Reliability and Risk Analysis*. Springer. ISBN 978-1-4471-4588-2.