Abstract Executions of Stochastic Discrete Event Systems

Michel Batteux

IRT SystemX, Paris-Saclay, France
E-mail: michel.batteux@irt-systemx.fr

Tatiana Prosvirnova

ONERA/DTIS, Université de Toulouse, F-31055 Toulouse, France
E-mail: tatiana.prosvirnova@onera.fr

Antoine Rauzy

Norwegian University of Science and Technology, Trondheim, Norway
E-mail: antoine.rauzy@ntnu.no

Abstract: Stochastic discrete event systems play a steadily increasing role in reliability engineering and beyond in systems engineering. Designing stochastic discrete event systems presents however a well-known difficulty: models are hard to debug and to validate because of the existence of infinitely many possible executions, itself due to stochastic delays, which are possibly intertwined with deterministic ones.

In this article, revisiting ideas introduced in the framework of model-checking of timed and hybrid systems, we show that it is possible to abstract the time in stochastic discrete event systems, therefore alleviating considerably debugging and validation tasks. More specifically, we show that schedules of transitions can be abstracted into systems of linear inequalities and that abstract and concrete executions are bisimilar: any concrete execution can be simulated by an abstract execution and reciprocally any abstract execution corresponds to at least one concrete execution. Moreover, we propose an efficient algorithm to determine whether generated systems of linear inequalities have solutions. This algorithm takes advantage of the very specific form of inequalities.

The result presented in this article represents thus a very important step forward in quality assurance of stochastic models of complex technical systems. We illustrate the potential of the proposed approach by means of AltaRica 3.0 models.

Keywords: Stochastic Discrete Event Systems; Timed and Hybrid Automata; Abstract Intepretation; AltaRica 3.0.

Biographical notes: Michel Batteux is currently a Research Engineer at the Technological Research Institute SystemX (France). After a Master degree in Mathematics and Computer Science at the Paris Diderot University, and a PhD in Computer Science at the Paris-Sud University, he had several research engineer positions in different academic or industrial research laboratories: the Laboratory of Model driven engineering for embedded systems of CEA, the Computer Science Laboratory of the Ecole Polytechnique, Thales Research and Technology. His topics of interest are around the model-based design and assessment of complex technical systems, with a special focus on the performance

assessment.

Tatiana Prosvirnova currently works as a Research Engineer at ONERA, the French Aerospace Lab (Toulouse, France). She has a PhD in Computer Science. She is graduated from Ecole Polytechnique. She has MS of Science of Ecole Polytechnique in Systems Engineering. She has been working as a Software Developer at Dassault Systemes (the largest French software editor) for three years. Her research interests are model based safety assessment, systems engineering, formal methods, software development, system architecture modeling and model synchronisation.

Antoine B. Rauzy has currently a Full Professor position at Norwegian University of Science and Technology (NTNU, Trondheim, Norway). He is also the head of the chair Blériot-Fabre, sponsored by the group SAFRAN, at CentraleSupélec (Paris, France). During his career, he was a Researcher at French National Centrer for Scientific Research, a CEO of the start-up company ARBoost Technologies and the director of the R&D Department on Systems Engineering at Dassault Systemes. He works in the reliability engineering and system safety field for more than 20 years. He extended his research topics to systems engineering since about ten years. He published over 200 articles in international conferences and journals. He developed state-of-the-art algorithms and software for probabilistic safety analyses.

1 Introduction

Stochastic discrete event systems [1, 2] play a steadily increasing role in reliability engineering and beyond in systems engineering. They encompass a large class of modeling formalisms such as stochastic Petri nets [3] and stochastic automaton networks [4] as well as high-level modeling languages such as AltaRica 3.0 [5]. Their interest stands in their great expressive power that makes it possible to represent complex behaviors.

Models designed within these formalisms can be assessed by means of various techniques, including the compilation into lower level modeling formalisms such as fault trees [6] or Markov chains [7], as well as Monte-Carlo simulation, the swiss-knife of behavioral modeling, see e.g. [8]. They are however hard to debug and to validate because of the infinite number of possible executions, itself due to the infinitely many possible choices of firing dates for transitions. This is probably the main limiting factor to their full scale deployment, especially in the context of performance assessment of life-critical systems. Most of the analysts have experienced this frustration of waiting long minutes, if not hours, for the results of a Monte-Carlo simulation to discover eventually that these results are meaningless because of a mistake somewhere in the model.

In this article, revisiting ideas introduced in the framework of model-checking of timed and hybrid systems [9, 10], we show that it is possible to abstract the time in stochastic discrete event systems. Namely, we define an abstraction of transition schedules by means of systems of linear inequalities. These systems encode the conditions for a transition to be enabled at a given step of an execution: the transition is enabled if and only if the corresponding system has a solution.

We show that abstract and concrete executions are bisimilar in the following sense (see e.g. [11] for a reference textbook on bisimulations): any concrete execution can be simulated by a unique abstract execution and reciprocally any abstract execution corresponds to at

least one concrete execution. This property is of a great interest because abstract models can be verified with techniques developed for non-timed discrete event systems, including model-checking techniques [12, 13].

Even without entering into the model-checking framework, this property makes it possible to perform abstract interactive simulations, therefore alleviating considerably debugging and validation tasks. Interactive simulators allow the analyst to go forth and back, step by step, in sequences of events, enabling in this way to track modeling errors, unexpected behaviors and so on. With that respect, they play a similar role as debuggers like GDB or DDD [14] do for C++ programs. Without the technique we introduce here, the designers of interactive simulators face a quite unpleasant choice: either ignoring delays, which has the major drawback that some non-timed executions have no timed counterpart, or ask the analyst to enter by hand the delays associated with stochastic transitions, which is tedious and let the analyst pondering which out of the infinitely many possible delays are the most suitable for his purpose. The abstract semantics we introduce here solves this important issue. Although it "only" makes it possible to look for qualitative properties (as opposed to probabilistic ones), it proves to be extremely useful to check various scenarios of interest for the validation of the model, e.g. that firing a given sequence of events is actually possible and ends up in a state with some expected properties.

The technique presented in this article enters into the general framework of Cousot's abstract interpretation [15]. The problem at stake was to make it work for the particular case of stochastic discrete event systems. Moreover, algorithmic mechanisms implementing abstract executions had to be efficient, so to apply on-the-fly model-checking techniques [16], which are probably the best suited in an engineering context. Solving systems of linear inequalities requires in the general case linear programming methods such as the simplex algorithm or more specifically the Fourier-Motzkin elimination [17, 18]. These methods are quite complex to implement and their execution is resource consuming. However, as pointed out by Wang, Pettersson and Daniels [9], one can take advantage of the particular form of the inequalities involved to design an efficient algorithm to check for the existence of solutions.

The framework presented in this article represents thus a very important step forward in quality assurance of stochastic discrete event systems. We illustrate its potential by means of AltaRica 3.0 models. To the best of authors' knowledge, AltaRica Wizard, the AltaRica 3.0 integrated modeling environment is the first one to benefit of the techniques presented here. These techniques could however probably be implemented in other modeling environments with related objectives, e.g. Figaro [19], GRIF Workshop [20] or PRISM [21].

The remainder of this article is organized as follows. Section 2 gives a formal definition of stochastic discrete event systems and discusses their semantics. Section 3 introduces their abstract semantics in terms of systems of linear inequalities, shows bisimulation theorems and explains how systems of linear inequalities can be solved efficiently. Section 4 presents an application of this framework to AltaRica 3.0 models. Finally, Section 5 concludes the article and gives some perspectives.

2 Stochastic Discrete Event Systems

In this section, we propose a formal definition of stochastic discrete event systems. This definition is strongly inspired from the notion of guarded transition systems [22], itself generalizing formalisms like stochastic Petri nets [3].

2.1 Formal Definition

A *stochastic discrete event system* is a triple $\langle S, T, s_0 \rangle$ where:

- S is a set of *states*. S may be finite or infinite.
- T is a finite set of *transitions*. Each transition t of T is a triple $\langle g, \delta, a \rangle$ where:
  - g is a Boolean condition, i.e. a function from S to $\{0, 1\}$ (representing respectively false and true). g is called the *guard* of the transition. We say that the transition t is *enabled* in the state $s \in S$ if $g(s) = 1$.
  - δ is a function from $S \times \mathbb{R}^+$ into CDF^{-1}, where $\mathbb{R}^+$ denotes the set of non-negative real numbers and CDF^{-1} denotes the set of inverse functions of cumulative distribution functions. δ is called the *delay distribution* of the transition. We shall explain in details this notion in Section 2.3.
  - a is a function from S to S. a is called the *action* of the transition. Assume that at a given step i, the system is in the state s_i and the transition is enabled in that state. Then, *firing* the transition is making the system change from state s_i to state $s_{i+1} = a(s_i)$.
- $s_0 \in S$ is the initial state of the stochastic discrete event system.

The above definition is quite liberal regarding the definition of states. They can be virtually anything one wants, ranging from explicitly enumerated states to complex data structures. It is easy to verify that formalisms such as stochastic Petri nets [3], stochastic automaton networks [4], guarded transitions systems [22], and queuing systems [23] are special cases of stochastic discrete event systems as we defined them.

Note that it is often the case that the action of a transition involves only a small subset of the variables or data structures representing the state. Firing the transition modifies thus only these variables or data structures, the other remaining unchanged.

Note also that stochastic discrete event systems can be generated by compiling higher level descriptions. This is the principle of the AltaRica 3.0 language developed by the authors [5]. AltaRica 3.0 results of the combination of guarded transition systems with S2ML, a versatile and unified set of object-oriented and prototype-oriented constructs to structure models [24].

Both stochastic discrete event systems and timed and hybrid automata [25] define timed interpretations of state automata. In timed automata, state automata are extended with a finite set of real-valued clocks. During an execution of a timed automaton, all clock values increase at the same speed. Transitions of the automaton can be guarded (enabled or disabled) by comparisons of clock values with integers, therefore constraining its possible behaviors. Furthermore, clocks can be reset. The two classes of models are thus quite close, even though they do not emphasize the same things: stochastic behaviors for (stochastic) discrete event systems, time constraints in the design of controllers of reactive systems for timed and hybrid automata. More importantly, the objectives of these two classes of models are significantly different, which leads to very different tooling. Moreover, stochastic discrete event models daily used in industry tend to be much larger but in some sense simpler than timed and hybrid automata models proposed in the literature, which are more academic. This said, the technique we present in this article to abstract the semantics of discrete event systems is close to the one introduced by Wang, Pettersson and Daniels to perform

reachability analyses in timed automaton [9]. The latter is at the core of the model checker UPPAAL [10], which is probably one of the most mature academic research tools in its domain.

2.2 Abstract Syntax

It is convenient to give an abstract syntax to discrete event systems. In the sequel, we shall denote a transition $\langle g, \delta, a \rangle$ as $g \xrightarrow{\delta} a$, using Boolean formulas to describe guards and instructions (in pseudo-code) to describe actions.

As an illustration, assume that we want to represent the queue for a given service. The state of this system can be described by a pair (q, s) of integer variables, where q represents the number of clients in the queue, and s represents the number of clients currently served.

The set S of possible states of the system is thus (theoretically) $\mathbb{N} \times \mathbb{N}$, i.e. the set of all possible pairs of integers. This set is indeed infinite. For practical reasons, we may assume however that there are never more than 10 clients waiting and that only one client is served at a time. In this case, S is reduced to product of integer ranges $[0, 10] \times [0, 1]$, which is indeed finite.

The evolution of the system can be represented by means of three transitions:

$$\begin{aligned} &t_a : q < 10 \xrightarrow{\delta_a} q = q + 1 \\ &t_b : q > 0 \wedge s < 1 \xrightarrow{\delta_b} q = q - 1, s = s + 1 \\ &t_c : s > 0 \xrightarrow{\delta_c} s = s - 1, \end{aligned}$$

where the transition t_a represents the arrival of a client in the queue, the transition t_b represents the beginning of the service of a client, and the transition t_c represents the completion of the service of a client.

The initial state is $(0, 0)$ as there is initially no client in the service.

2.3 Delays

Let $M : \langle S, T, s_0 \rangle$ be a stochastic discrete event system. The delay distribution δ associated with a transition $t : g \xrightarrow{\delta} a$ of T associates the inverse of a cumulative distribution function with each state $s \in S$ and each date $\varphi \in \mathbb{R}^+$.

Intuitively, the delay distributions are used as follows, typically when performing a Monte-Carlo simulation. If the transition $g \xrightarrow{\delta} a$ gets enabled at a date φ in a state s, then a number z is drawn at pseuso-random uniformly between 0 and 1, the corresponding delay d is calculated as $d = \delta(s, \varphi)(z)$. If the transition remains enabled from the date φ to the date $\varphi + d$, then it is fired at $\varphi + d$.

In many practical applications, the delay distribution does not depend on the state s, nor on the date φ, i.e. $\delta(s_1, \varphi_1) = \delta(s_2, \varphi_2)$ for all $s_1, s_2 \in S$ and $\varphi_1, \varphi_2 \in \mathbb{R}^+$. We shall assume this independence in the sequel.

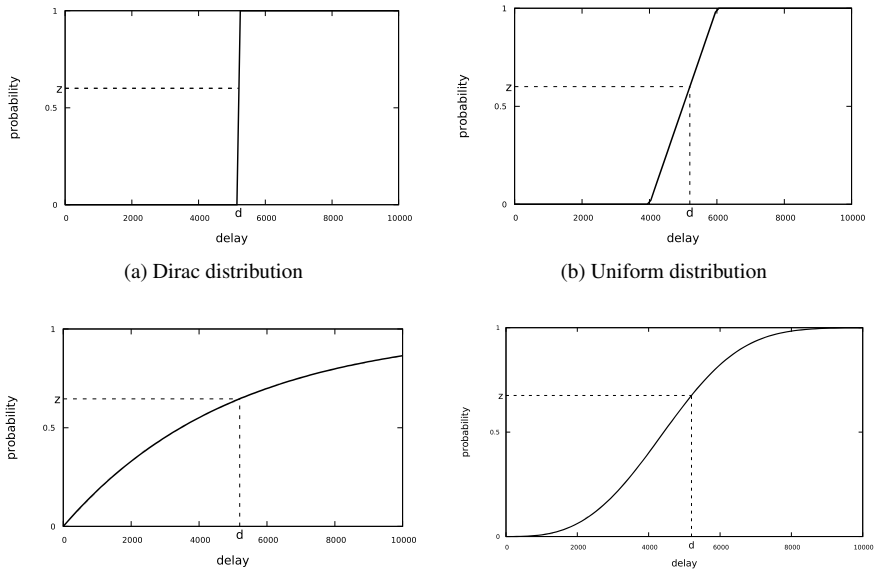
Recall that a cumulative distribution function is a function ϕ from $\mathbb{R}^+$ into $[0, 1]$ verifying the following condition.

$$\forall \varphi_1, \varphi_2 \in \mathbb{R}^+ \quad \varphi_1 \leq \varphi_2 \Longrightarrow \phi(\varphi_1) \leq \phi(\varphi_2) \tag{1}$$

In practice, the cumulative distribution functions that are used are either parametric distributions such as the negative exponential distribution, or empirical distributions

described by means of a set of points between which the value of the distribution is interpolated. Kaplan-Meier estimators [26] are typical examples of empirical distributions.

Figure 1 shows several parametric distributions that are widely used: a Dirac distribution Figure 1a, a uniform distribution Figure 1b, an exponential distribution Figure 1c, and a Weibull distribution Figure 1d.



(a) Dirac distribution

(b) Uniform distribution

(c) Exponential distribution

(d) Weibull distribution

Figure 1: Some widely used distributions

Dirac distributions correspond to deterministic delay distributions. All other delay distributions are stochastic.

In our example, the transition t_a represents the arrival of a client in the queue. The transition t_b represents the beginning of the service of a client. Finally, the transition t_c represents the completion of the service of a client. As a reasonable approximation, we can consider that they do not depend on the state of the system, nor on the current time. In the sequel, we shall assume that the delay δ_a obeys an exponential distribution, that the delay δ_b is null (the client is served as soon as possible), and finally that the delay δ_c obeys a uniform distribution between two bounds.

2.4 Semantics

Let $M : \langle S, T, s_0 \rangle$ be a stochastic discrete event system. M encodes implicitly a set of possible executions.

If we forget about delays, i.e. if we consider non-timed executions, the set of possible executions is the smallest set such that:

- s_0 is an (empty) execution.

- If $\sigma = s_0 \xrightarrow{t_1} s_1 \cdots \xrightarrow{t_{n-1}} s_{n-1}$, $n \geq 1$, is an execution, then so is $\sigma \xrightarrow{t_n} s_n$, where $s_n \in S$ and $t_n : g_n \xrightarrow{\delta_n} a_n \in T$ under the conditions that t_n verifies condition 1 and s_n verifies condition 2 given below.

Condition 1 (Fired transitions are enabled) *The transition $t_n : g_n \xrightarrow{\delta_n} a_n$ fired at step n was enabled at step $n-1$:*

$$g_n(s_{n-1}) = 1$$

Condition 2 (Next state calculation) *The state s_n at step n is obtained from the state s_{n-1} at step $n-1$ by applying the action of the fired transition $t_n : g_n \xrightarrow{\delta_n} a_n$:*

$$s_n = a_n(s_{n-1})$$

A timed execution of M is a non-timed execution of M with additional constraints due to delays. To define formally timed executions, we need to introduce the notion of schedule.

A *schedule* φ of M is a function from transitions of T to $\mathbb{R}^+ \cup \{\infty\}$ that associates a firing date with each transition t of T such that:

- $\varphi(t) \in \mathbb{R}^+$ if t is enabled in the current state.
- $\varphi(t) = \infty$ otherwise.

The set of timed executions of M is the smallest set such that:

- $\langle s_0, 0, \varphi_0 \rangle$ is a timed execution, namely the empty execution with the initial schedule φ_0 verifying the conditions 3 and 4 given below.
- If $\sigma = \langle s_0, 0, \varphi_0 \rangle \xrightarrow{t_1} \langle s_1, d_1, \varphi_1 \rangle \cdots \xrightarrow{t_{n-1}} \langle s_{n-1}, d_{n-1}, \varphi_{n-1} \rangle$, $n \geq 1$, is a timed execution, then so is $\sigma \xrightarrow{t_n} \langle s_n, d_n, \varphi_n \rangle$, where $t_n : g_n \xrightarrow{\delta_n} a_n \in T$ verifies condition 1, $s_n \in S$ verifies condition 2, $d_n = \varphi_{n-1}(t_n)$ is the date of the step n, and φ_n is a schedule verifying conditions 3-6 given below.

Condition 3 (Scheduled Transitions) *The transition t is scheduled at step $n \geq 0$ if and only if it is enabled, i.e. $\forall t : g \xrightarrow{\delta} a \in T$:*

$$g(s_n) = 1 \Longrightarrow \varphi_n(t) \in \mathbb{R}^+$$
$$g(s_n) = 0 \Longrightarrow \varphi_n(t) = \infty$$

A consequence of condition 3 is that if a transition t that was scheduled at step $n \geq 0$ is not enabled anymore at step $n+1$, then it is "de-scheduled", i.e. $\varphi_{n+1}(t) = \infty$.

Condition 4 (Newly Enabled Transitions) *If the transition t gets enabled at step $n \geq 0$, then it is scheduled, i.e. $\forall t : g \xrightarrow{\delta} a \in T$:*

$$g(s_{n-1}) = 0 \wedge g(s_n) = 1 \Longrightarrow \varphi_n(t) = d_n + \delta_n(t)$$
$$\wedge \exists z \in [0,1] \text{ s.t. } \delta_n(t) = \delta(s_n, \varphi_n)(z)$$

Condition 4 applies also:

- To the initial state, i.e. with $n = 0$ and by posing $g(s_{-1}) = 0$;

- To the transition t_n (even though $g(s_{n-1}) = 1$).

Condition 5 (Earliest Transitions Firing) *The transition t_n fired at step $n \geq 1$ is one of those with the earliest firing dates at step $n-1$, i.e. $\forall t \in T$:*

$$d_n = \varphi_{n-1}(t_n) \leq \varphi_{n-1}(t)$$

Condition 6 (Previously Enabled Transitions) *Finally, if the transition $t \neq t_n$ was enabled at step $n-1$ and is still enabled at step n, $n \geq 1$, then its schedule stays the same, i.e. $\forall t : g \xrightarrow{\delta} a \in T, t \neq t_n$:*

$$g(s_{n-1}) = 1 \wedge g(s_n) = 1 \Longrightarrow \varphi_n(t) = \varphi_{n-1}(t)$$

As an illustration consider again our queuing example.

Initially, there is no client in the queue (and indeed no client served), so $q_0 = 0$ and $s_0 = 0$. In this state, only the transition t_a is enabled. A delay $\delta_0(t_a)$ is thus calculated (i.e. drawn at random according to the delay of t_a), e.g. $\delta_0(t_a) = 3$. The firing date $\varphi_0(t_a)$ of t_a is defined as $\varphi_0(t_a) = d_0 + \delta_0(t_a) = 3$ (condition 4). Moreover, $\varphi_0(t_b) = \varphi_0(t_c) = \infty$.

As t_a is the only enabled transition at step 0, the first event that occurs in the system is the firing of this transition ($t_1 = t_a$), at the date $d_1 = \varphi_0(t_a) = 3$. The state at step 1 is thus $a_a((q_0, s_0)) = (q_1, s_1) = (1, 0)$ (condition 2).

In this state, both transitions t_a and t_b are enabled. A new delay is thus calculated for the transition t_a, e.g. $\delta_1(t_a) = 4$, and the delay $\delta_1(t_b) = 0$ is calculated for the transition t_b as this transition is deterministic and immediate. We have thus $\varphi_1(t_a) = d_1 + \delta_1(t_a) = 7$ and $\varphi_1(t_b) = d_1 + \delta_1(t_b) = 3$. Indeed, $\varphi_1(t_c) = \infty$.

At this point, both t_a and t_b are enabled, but due to the condition 5, only t_b can be fired, as $\varphi_1(t_b) = 3 < \varphi_1(t_a) = 7$. This illustrates the fact that some non-timed executions do not correspond to any timed executions.

We have thus $t_2 = t_b, d_2 = \varphi_1(t_b) = 3, q_2 = 0$ and $s_2 = 1$. In this state, both transitions t_a and t_c are enabled. As t_a was already enabled in state (q_1, s_1), its firing date remains unchanged (condition 6), i.e. $\varphi_2(t_a) = \varphi_1(t_a)$. The transition t_c was not enabled in state (q_1, s_1), so a new delay $\delta_2(t_c)$ is calculated for this transition, e.g. $\delta_2(t_c) = 2$, and its firing date is set to $\varphi_2(t_c) = d_2 + \delta_2(t_c) = 5$

Applying condition 5, we see that as $\varphi_2(t_c) = 5 < \varphi_2(t_a) = 7$, t_c must be fired at step 3.

And so on.

It is worth noticing that the semantics of stochastic discrete event systems is fully deterministic, except for the calculation of delays and the choice of the transition to fire when several transitions are enabled at the same time.

2.5 Interpretation in Terms of Timed Automata

Stochastic discrete event systems can be re-interpreted in terms of (stochastic) timed automata.

The idea consists in associating a clock c with each transition $t : g \xrightarrow{\delta} a \in T$. c is reset when t gets logically enabled at step $n \geq 0$. The guard g is extended into $g' : \ g \wedge c = \varphi_n(t)$, where $\varphi_n(t)$ is calculated as in condition 4. This makes the transition t enabled exactly at the date $\varphi_n(t)$.

In other words, stochastic discrete event systems can be seen as stochastic timed automata in which clocks are implicitly defined. The latter are thus strictly more expressive than the former. However, this gain in expressiveness comes with a significant price in terms of practical difficulty to design, to validate and to maintain models. This is probably the reason why timed automata remain, as of today, mostly used in academia, conversely to discrete event systems that are widely used in industry.

2.6 Stochastic Simulations

Stochastic discrete event systems are in general assessed by means of Monte-Carlo simulations. The idea is fairly simple: one performs a large number of executions, drawing at pseudo-random delays of stochastic transitions. Executions are stepwisely expanded until a certain mission time is reached. Along each execution, the value of some indicators (random variables) are calculated. Then, one performs statistics on these values, over the different executions.

The indicators that may be calculated belong to three categories.

First, indicators regarding transitions (e.g. the number of transitions fired during the execution, the number of times a given transition has been fired during the execution, or the first date at which a given transition has been fired during the execution).

Second, indicators relying on predicates over states. A *predicate* over states is a Boolean function over states (e.g. the first date at which a state satisfying the predicate has been reached during the execution, or the number of times a state satisfying the predicate has been reached during the execution).

Third, indicators relying on reward over states. A *reward* over states is a real-valued function over states (e.g. the minimum, maximum or mean value the reward takes during the execution).

The above list is indeed non-exhaustive, although it covers most of the needs. It can be extended at will.

In our queuing example, we may be interested in:

- The number of times the transition t_a is fired, which gives an indicator on the number of clients who joined the queue.
- The number of times the transition t_c is fired, which gives an indicator on the number of clients who have been served.
- The maximum and mean values of the reward q, which gives an indicator on the maximum and mean numbers of waiting clients.
- The mean value of the reward that takes the value 1 when $s = 0$ and the value 0 otherwise, which gives an indicator on the time the operator at the counter remains idle.

3 Abstract Semantics

3.1 Principle

Let $M : \langle S, T, s_0 \rangle$ be a stochastic discrete event system and let $\sigma = \langle s_0, d_0, \varphi_0 \rangle \xrightarrow{t_1} \cdots \xrightarrow{t_n} \langle s_n, d_n, \varphi_n \rangle$, $n \geq 1$, be a timed execution of M.

By conditions 3-6, for each firing date d_j, $j \geq n$, it exists a step i, $0 \leq i < j$, such that the transition t_j fired at step j has been scheduled at step i, i.e.

$$d_j = \varphi_i(t_j) = d_i + \delta_i(t_j) \tag{2}$$

This property can be graphically illustrated by timelines. Figure 2 shows the timeline describing a timed execution of our queuing system.

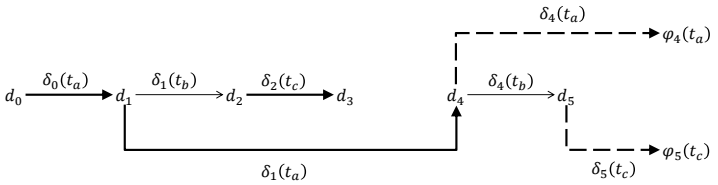


Figure 2: A timeline describing an execution of the queuing system.

This timed execution is made of the following steps.

0. Initially, i.e. at date $d_0 = 0$, there is no client in the queue and no client served. The arrival of a first client (transition t_a) is scheduled at date $\varphi_0(t_a) = d_0 + \delta_0(t_a)$.
1. At date $d_1 = \varphi_0(t_a)$, the first client arrives in the queue. A second client arrival is scheduled at date $\varphi_1(t_a) = d_1 + \delta_1(t_a)$ and the beginning of the service of the first client is scheduled at date $\varphi_1(t_b) = d_1 + \delta_1(t_b)$.
2. At date $d_2 = \varphi_1(t_b)$, the first client starts to be served. The completion of the service of the first client is scheduled at date $\varphi_2(t_c) = d_2 + \delta_2(t_c)$.
3. At date $d_3 = \varphi_2(t_c)$, the service of the first client is completed.
4. At date $d_4 = \varphi_1(t_a)$, the second client arrives in the queue. The start of the service of the second client is scheduled at date $\varphi_4(t_b) = d_4 + \delta_4(t_b)$. Moreover, the arrival of a third client is scheduled at date $\varphi_4(t_a) = d_4 + \delta_4(t_a)$.
5. At date $d_5 = \varphi_4(t_b)$, the second client starts to be served. The completion of the service of the second client is scheduled at date $\varphi_5(t_c) = d_5 + \delta_5(t_c)$.

In Figure 2, we represent stochastic transitions (t_a and t_c) by thick arrows and deterministic ones (t_b) by thin arrows. Moreover, transitions scheduled but not fired yet are represented with dashed arrows. We shall keep these conventions in the sequel.

If we consider the d_i's, the $\varphi_i(t)$'s and the $\delta_i(t)$'s as real-valued variables, each execution generates three sets of constraints:

- The equalities $\varphi_i(t) = d_i + \delta_i(t)$ which reflect the dates at which the transitions are scheduled and fired. The difference between these two dates being the delay calculated for the transition.
- The equalities $d_j = \varphi_i(t_j)$ that indicates which transition is fired at step j.
- Finally, the inequalities $d_{i-1} \leq d_i$ that reflect the chronological order of steps.

Table 1 summarizes the constraints generated by the execution depicted in Figure 2.

The key idea behind the definition of an abstract semantics for stochastic discrete event systems is thus to step-wisely generate a system of inequations for each execution, rather than concrete values for the d_i's, the $\varphi_i(t)$'s and the $\delta_i(t)$'s. The abstract execution is valid if and only if the corresponding system of inequalities has a solution. This process starts by defining abstract delays.

Table 1 Constraints generated by the execution pictured in Figure 2.

Step	Date	Chronology	Schedule
0	$d_0 = 0$		$\varphi_0(t_a) = d_0 + \delta_0(t_a)$
1	$d_1 = \varphi_0(t_a)$	$d_0 \leq d_1$	$\varphi_1(t_a) = d_1 + \delta_1(t_a),\ \ \varphi_1(t_b) = d_1 + \delta_1(t_b)$
2	$d_2 = \varphi_1(t_b)$	$d_1 \leq d_2$	$\varphi_2(t_c) = d_2 + \delta_2(t_c)$
3	$d_3 = \varphi_2(t_c)$	$d_2 \leq d_3$	
4	$d_4 = \varphi_1(t_a)$	$d_3 \leq d_4$	$\varphi_4(t_a) = d_4 + \delta_4(t_a),\ \ \varphi_4(t_b) = d_4 + \delta_4(t_b)$
5	$d_5 = \varphi_4(t_b)$	$d_4 \leq d_5$	$\varphi_5(t_c) = d_5 + \delta_5(t_c)$

3.2 Abstract Delays

If we do not put any constraint on the values of the delays, i.e. on variables $\delta_i(t)$'s, the systems of linear inequalities are actually trivially satisfiable: it suffices to set $\delta_i(t) = 0$ for all steps i and all steps t. However, we do have information on the $\delta_i(t)$'s as they are obtained by considering inverse functions of cumulative probability distributions.

The second idea upon which the abstract semantics of stochastic discrete event systems relies, consists thus in abstracting possible values of delays $\delta_i(t)$ by means of intervals of $\mathbb{R}^+ \cup \{\infty\}$. Table 2 provides the intervals associated with widely used built-in delay functions as well as the corresponding scheduling constraints.

Table 2 Intervals associated with delay functions

Distribution	Variation interval	Scheduling constraints
Dirac(d)	$[d, d]$	$\varphi_i(t) = d_i + d$
exponential(λ)	$]0, +\infty)$	$\varphi_i(t) > d_i$
Weibull(α, β)	$]0, +\infty)$	$\varphi_i(t) > d_i$
uniform(l, h)	$]l, h[$	$\varphi_i(t) > d_i + l,\ \ \varphi_i(t) < d_i + h$

In other words, delays can be split into two categories:

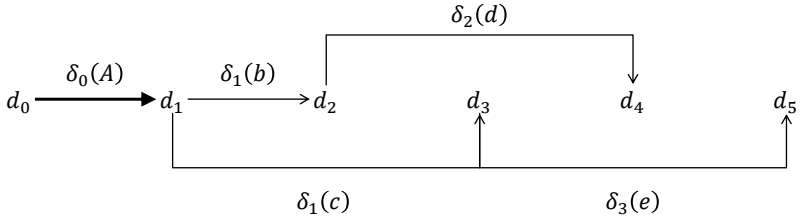
- Deterministic delays, represented by Dirac distributions, which can take a single value d, $d \geq 0$.
- Stochastic delays, represented by all other distributions, which can take any value in an interval $]l, h[$, $0 < l < h \leq \infty$, where l and h depend on the distribution.

There is however a subtlety that prevents to implement our two ideas directly. It is explained in the next section 3.3.

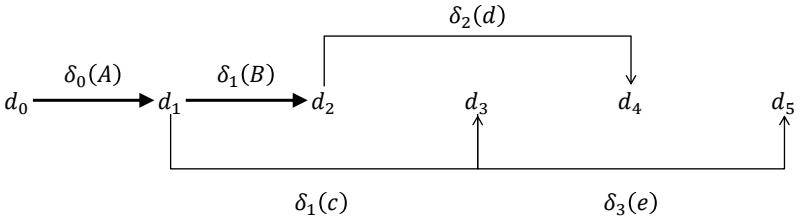
3.3 Causality Chains and Chronology Constraints

A stochastic transition can never be fired at exactly the same time as another transition. The reason is a well-known argument of the Kolmogorov axiomatic of probability theory [27]: the Lebesgue's measure of the probability of such an event is null. This has to be reflected in systems of inequalities. Namely, there are cases in which inequalities $d_{i-1} \leq d_i$, which reflect the chronology, must be strict: the date of the firing of the transition t_i can be as close as one wants to the date of the firing of the transition t_{i-1}, but not exactly the same.

Figure 3 shows the timelines of nearly identical executions of some stochastic discrete event systems, involving both deterministic and stochastic transitions. The first execution, pictured in Figure 3 (a), starts with a stochastic transition A, then continues with deterministic transitions b, c, d and e. The second execution, pictured in Figure 3 (b), is similar to the first one, except that now B is a stochastic transition. In the first execution, steps 4 and 5 can take place at the same date, i.e. $d_4 \leq d_5$, providing that $d_1(b) + d_2(d) = d_1(c) + d_3(e)$; even though the execution starts with a stochastic transition. In the second execution, steps 4 and 5 cannot take place at the same date, i.e. $d_4 < d_5$, even though both transitions d and e are deterministic. The reason is that the transition B prevents the two dates to be equal.



(a) Execution 1



(b) Execution 2

Figure 3: Timelines of two executions involving both deterministic and stochastic transitions.

To decide about the chronology constraint between dates d_{i-1} and d_i, we need to introduce the notion of causality chain. Let $M : \langle S, T \rangle$ be a stochastic discrete event system and let $\sigma = \langle s_0, d_0, \varphi_0 \rangle \xrightarrow{t_1} \cdots \xrightarrow{t_n} \langle s_n, d_n, \varphi_n \rangle$, $n \geq 1$, be a timed execution of M. The *causality chain* of the transition t_n is the sub-sequence of transitions of σ such that:

$$\langle s_0, d_0, \varphi_0 \rangle \xrightarrow{t_{i_1}} \langle s_{i_1}, d_{i_1} = \varphi_0(t_{i_1}), \varphi_{i_1} \rangle \xrightarrow{t_{i_2}} \langle s_{i_2}, d_{i_2} = \varphi_{i_1}(t_{i_2}), \varphi_{i_2} \rangle \cdots \xrightarrow{t_n} \langle s_n, d_n = \varphi_{i_k}(t_n), \varphi_n \rangle$$

By construction, the causality chain of the transition t_n exists and is unique.

In the execution pictured in Figure 3 (a), the causality chains of transitions d and e are as follows:

$$\langle s_0, d_0, \varphi_0 \rangle \xrightarrow{A} \langle s_1, d_1 = \varphi_0(A), \varphi_1 \rangle \xrightarrow{b} \langle s_2, d_2 = \varphi_1(b), \varphi_2 \rangle \xrightarrow{d} \langle s_4, d_4 = \varphi_2(d), \varphi_4 \rangle$$

$$\langle s_0, d_0, \varphi_0 \rangle \xrightarrow{A} \langle s_1, d_1 = \varphi_0(A), \varphi_1 \rangle \xrightarrow{c} \langle s_3, d_3 = \varphi_1(c), \varphi_3 \rangle \xrightarrow{e} \langle s_5, d_5 = \varphi_3(e), \varphi_5 \rangle$$

Now, any two causality chains σ_1 and σ_2 extracted from an execution σ have a largest common prefix, i.e. can be written as $\sigma_1 = \pi\tau_1$ and $\sigma_2 = \pi\tau_2$, where the suffixes τ_1 and τ_2 share no transition.

In the above example, the largest common prefix of causality chains of d and e is made of the transition A, while their respective suffixes consist of the transitions b, d in one case and c and e in the other case.

The rule to decide whether the chronology constraint between dates d_{i-1} and d_i is strict or not can be stated as follow. Let $\sigma_{i-1} = \pi\tau_{i-1}$ and $\sigma_i = \pi\tau_i$ be the causality chains of the transitions t_{i-1} and t_i, where π is their largest common prefix. Then, the chronology inequality between d_{i-1} and d_i is strict if and only if at least one of the suffix sequences τ_{i-1} and τ_i involves a stochastic transition.

In our example, the chronology inequality between d_4 and d_5 is thus not strict in the first execution (as transitions b, c, d and e are all deterministic), and strict in the second one (as the transition B is stochastic).

Now, we can define formally the abstract semantics of stochastic discrete event systems.

3.4 Formal Definition

Let $M : \langle S, T, s_0\rangle$ be a stochastic discrete event system. M encodes implicitly a set of possible abstract executions. The set of abstract executions of M is the smallest set such that:

- $\langle s_0, \Gamma_0\rangle$ is an abstract execution, namely the empty execution starting in the initial state $s_0 \in S$ with the initial system of inequalities Γ_0 containing:
  - The equality $d_0 = 0$, and
  - The scheduling constraints defined by the condition 7 given below.
- If $\sigma = \langle s_0, \Gamma_0\rangle \xrightarrow{t_1} \langle s_1, \Gamma_1\rangle \cdots \xrightarrow{t_{n-1}} \langle s_{n-1}, \Gamma_{n-1}\rangle$, $n \geq 1$, is an abstract execution, then so is $\sigma \xrightarrow{t_n} \langle s_n, \Gamma_n\rangle$, where $t_n : g_n \xrightarrow{\delta_n} a_n \in T$ verifies condition 1, $s_n \in S$ verifies condition 2, and Γ_n is obtained by adding to Γ_{n-1} the constraints:
  - $d_n = \varphi_i(t_n)$, where i is the step at which t_n has been scheduled.
  - $d_{n-1} < d_n$ or $d_{n-1} \leq d_n$ according to the rule on causality chains defined in the previous section.
  - The scheduling constraints defined by the condition 7 given below.

Condition 7 (Abstract Scheduling) *If the transition $t : g \xrightarrow{\delta} a$ gets enabled at step $n \geq 0$, then Γ_n contains the constraints:*

- *$\varphi_n(t) = d_n + d$, if $\delta(s_n)$ is a deterministic delay d.*
- *$\varphi_n(t) > d_n + l$ and $\varphi_n(t) < d_n + h$, if $\delta(s_n)$ is a stochastic delay with a lower bound l and an upper bound h.*

An abstract execution $\langle s_0, \Gamma_0\rangle \xrightarrow{t_1} \langle s_1, \Gamma_1\rangle \cdots \xrightarrow{t_n} \langle s_n, \Gamma_n\rangle$, $n \geq 0$, is *valid* if all the Γ_i are satisfiable, for $i = 1 \cdots n$.

Note that the satisfiability of Γ_n implies, by construction, the satisfiability of Γ_0, Γ_1, …, Γ_{n-1}.

3.5 Bisimulation

The key mathematical property in our case is that abstract and concrete executions are *bisimilar*: any concrete execution can be simulated by an abstract execution and reciprocally any abstract execution corresponds to at least one concrete execution.

The following two theorems capture this property.

Theorem 1: *Let $M : \langle S, T, s_0 \rangle$ be a stochastic discrete event system and let $\sigma = \langle s_0, d_0, \varphi_0 \rangle \xrightarrow{t_1} \langle s_1, d_1, \varphi_1 \rangle \cdots \xrightarrow{t_n} \langle s_n, d_n, \varphi_n \rangle$, $n \geq 0$, be a timed execution of M.*

Then, the abstract execution $\langle s_0, \Gamma_0 \rangle \xrightarrow{t_1} \langle s_1, \Gamma_1 \rangle \cdots \xrightarrow{t_n} \langle s_n, \Gamma_n \rangle$ built as described above is valid.

Proof. By construction.

Theorem 2: *Let $M : \langle S, T, s_0 \rangle$ be a stochastic discrete event system and let $\langle s_0, \Gamma_0 \rangle \xrightarrow{t_1} \langle s_1, \Gamma_1 \rangle \cdots \xrightarrow{t_n} \langle s_n, \Gamma_n \rangle$, $n \geq 0$, be a valid abstract execution of M.*

Then, there exists at least one timed execution $\sigma = \langle s_0, d_0, \varphi_0 \rangle \xrightarrow{t_1} \langle s_1, d_1, \varphi_1 \rangle \cdots \xrightarrow{t_n} \langle s_n, d_n, \varphi_n \rangle$ of M.

Proof. If the abstract execution is valid, then by definition, Γ_n is satisfiable. Let σ_n be a solution of Γ_n, i.e. a valuation of each of variables that satisfies all constraints of Γ_n. Using σ_n, we can then define concrete delays and firing dates. As concrete delays and firing dates defined in this way verify by construction conditions 3-6, we obtain a valid concrete execution.

The two above theorems have important practical consequences: it is possible to verify properties of infinitely many concrete executions by means of finitely many abstract executions. Furthermore some applications will be discussed in Section 4. But before entering into this discussion, we need to show that abstract executions can be implemented efficiently.

3.6 Constraint Solving Algorithm

Let $M : \langle S, T, s_0 \rangle$ be a stochastic discrete event system and let $\sigma : \langle s_0, \Gamma_0 \rangle \xrightarrow{t_1} \langle s_1, \Gamma_1 \rangle \cdots \xrightarrow{t_n} \langle s_n, \Gamma_n \rangle$, $n \geq 0$, be an abstract execution of M. σ is constructed step by step, so we can assume that the validity of all prefix executions of σ has been checked. It remains thus to check the satisfiability of Γ_n, or more precisely to check that the constraints introduced at step n are compatible with the constraints of Γ_{n-1}.

A first remark here is that variables d_j's, $j = 1 \ldots n$, are essentially renaming of variables $\varphi_j(t_i)$, for some $0 \leq i < j$. We can thus eliminate them when building the Γ_i's. Actually, we introduced them in the above developments only for the sake of clarity of the presentation.

We are thus left with three types of equations:

- Equations of the form $\varphi_{i-1}(t) < \varphi_i(t')$ or $\varphi_{i-1}(t) \leq \varphi_i(t')$ describing chronology constraints. We can normalize these equations respectively as, $\varphi_i(t') > \varphi_{i-1}(t) + 0$ and $\varphi_i(t') \geq \varphi_{i-1}(t) + 0$.

- Equations of the form $\varphi_j(t) = \varphi_i(t') + d$, $j > i$ describing the abstract scheduling of deterministic transitions. We can normalize these equations by introducing two inequalities: $\varphi_j(t) \leq \varphi_i(t') + d$ and $\varphi_j(t) \geq \varphi_i(t') + d$.
- Equations of the form $\varphi_j(t) > \varphi_i(t') + l$, and $\varphi_j(t) < \varphi_i(t') + h$, $j > i$ describing the abstract scheduling of stochastic transitions.

Eventually, we end up with inequalities of the form:

- $Y_j < X_i + c$ or $Y_j \leq X_i + c$, $j > i$, and
- $Y_j > X_i + c$ or $Y_j \geq X_i + c$, $j > i$.

The idea is thus to perform the Fourier-Motzkin elimination backward, i.e. starting from the $\varphi_n(t)$'s:

- From $Z_k < Y_j + d$ and $Y_j < X_i + c$, one can deduces $Z_k < X_i + d + c$,
- From $Z_k \leq Y_j + d$ and $Y_j \leq X_i + c$, one can deduces $Z_k \leq X_i + d + c$,
- and so on

Three important remarks here.

First, the form of inequalities produced by the Fourier-Motzkin elimination is the same as the form of the original inequalities. This makes it possible to optimize data structures and operations on system of inequalities.

Second, if the system contains two inequalities $Y_j < X_i + c$ and $Y_j < X_i + d$, then one is necessarily useless: the first one if $c > d$, the second otherwise. This applies indeed for all pairs of inequalities going in the same direction, e.g. $<$ and $\leq$, $\leq$ and $\leq$, etc. Again, this makes it possible to optimize data structures and operations on system of inequalities. In particular, AVL trees or similar structures can be used to store inequalities in order to be able to retrieve efficiently inequalities involving any two variables (in $\mathcal{O}(\log m)$, where m is the number of pairs of variables involved in inequalities of the system).

Third, the system is unsatisfiable if and only if, at some point of the Fourier-Motzkin process, the system contains two inequalities $Y_j < X_i + c$ and $Y_j > X_i + d$ such that $c \leq d$. Indeed, this applies for all pairs of inequalities (strict or not), except in the case $Y_j \leq X_i + c$ and $Y_j \geq X_i + c$.

This means that to check the satisfiability of Γ_n, given that Γ_{n-1} is satisfiable, it suffices to apply the Fourier-Motzkin elimination keeping only variables introduced at step n as left members of inequalities.

In our implementation, we used a few other optimizations, but the description of which goes beyond the scope of this article.

As the result of the above developments, the greedy algorithm that checks whether a given transition can be fired at step n is very efficient, i.e. of nearly constant complexity in all practical cases we have dealt with.

4 Application to AltaRica 3.0 Models

4.1 The AltaRica 3.0 Modeling Language

AltaRica 3.0 is an object-oriented modeling language dedicated to probabilistic risk and safety analyses of complex technical systems [5]. It combines guarded transition systems

[22, 28] with the structuring paradigm S2ML [24]. Guarded transition systems are a specific implementation of stochastic discrete event systems, as we defined them in this article. S2ML, which stands for System Structure Modeling Language, gathers, in an unified way, structuring constructs stemmed from object-oriented programming and prototype-oriented programming.

Several assessment tools have been developed for AltaRica 3.0, including a stepwise simulator, compilers to fault trees and Markov chains, and a stochastic simulator. These tools are provided with the freely available OpenAltaRica Platform[1]. The stepwise simulator makes it possible to perform interactive simulations of AltaRica 3.0 models. It proves to be of great interest for stakeholders to discuss the behaviors of the systems under study. It is also very useful to debug and to validate models, as we shall see in this section. The original version of the stepwise simulator did not take into account delays associated to transitions. As a consequence, it was possible to fire sequences of transitions that were impossible according to the timed semantics.

The new version of this simulator, which implements the abstract semantics presented in the previous sections, makes it fully compliant with the timed semantics of AltaRica 3.0. Furthermore, results coming from both stochastic and stepwise simulators can be compared: any simulation played in the stepwise simulator could have been generated by the stochastic simulator, and vice versa.

4.2 Illustrative example

As an illustration, we shall consider the simplified power-supply system of a farm of servers pictured in Fig. 4.

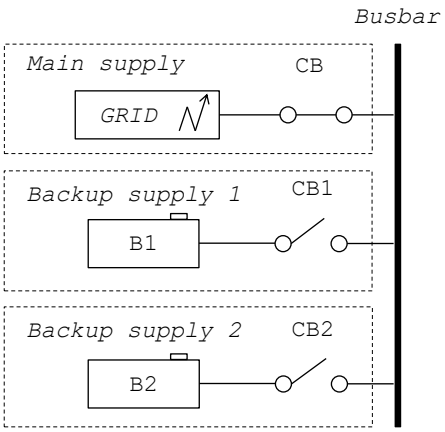


Figure 4: A power supply system

The power is delivered to the busbar via three redundant channels. The main supply channel consists of the grid `G` and a circuit breaker `CB`. The two backup supply channels consist of a battery (actually a group of batteries) `Bi`, $\texttt{i} = 1, 2$, and a circuit breaker `CBi`, $\texttt{i} = 1, 2$. All these components may fail.

In the initial state (and more generally when the main supply is working), `CB` is closed, while `CB1` and `CB2` are open. If the main supply is lost, the network is configured in order

to use the first backup supply, i.e. `CB1` is closed, while `CB` and `CB2` are opened. If the first backup supply is also lost, the network is configured in order to use the second backup supply, i.e. `CB2` is closed, while `CB` and `CB1` are opened. Network reconfigurations are assumed to be instantaneous, i.e. the corresponding transitions are associated with Dirac(0) delays.

The loss of the grid is assumed to be exponentially distributed, with a failure rate $\lambda_{\text{G}} = 5.0 \times 10^{-4}\, h^{-1}$. Moreover, it is assumed that it is recovered after at most 12 hours, i.e. to be uniformly distributed between $\alpha_{\text{G}} = 0h$ and $\beta_{\text{G}} = 12h$. The guarded transition system representing the behavior of the grid is pictured in Fig. 5a.

The batteries are normally in the standby mode. They may fail in this mode. This failure is assumed to be dormant, i.e. to remain unnoticed until the battery is actually used, and exponentially distributed with a failure rate $\lambda_{\text{B}} = 2.5 \times 10^{-5}\, h^{-1}$. In reality, periodic tests make it possible to detect these failures. However, we shall not include maintenance policies in the model presented here, in order not to overload it. When the battery is in use, it discharges. The time to a full discharge of the battery is assumed to be uniformly distributed between $\alpha_{\text{B}} = 8h$ and $\beta_{\text{B}} = 10h$. The guarded transition system representing the behavior of batteries is pictured in Fig. 5b. As for timeline, deterministic (here immediate) transitions are represented with thin arrows, while stochastic transitions are represented with thick ones. Transitions can be guarded not only by their source state, here encoded by the variable `_state`, but also by some other variables, here the Boolean variable `active` that indicates whether the battery is currently required to provide power to the busbar.

Finally, the circuit breakers may be either free to open and to close, or stuck in one of these positions. This failure is again assumed to be dormant and exponentially distributed, with a failure rate $\lambda_{\text{CB}} = 1.0 \times 10^{-6}\, h^{-1}$. As we do not take into account maintenance policies here, circuit breakers are assumed to be non repairable. The guarded transition system representing the behavior of circuit breakers is pictured in Fig. 5c.

Table 3 summarizes the probability distributions associated with failures and repairs.

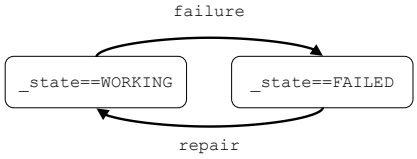
Table 3 Reliability parameters of the power supply system

Component	**Transition**	**Distribution**	**Parameters**
Grid	Failure	Exponential	Failure rate $\lambda_{\text{G}} = 5.0 \times 10^{-4}\, h^{-1}$
	Repair	Uniform	Lower bound $\alpha_{\text{G}} = 0\, h$, upper bound $\beta_{\text{G}} = 12\, h$
Batteries	Dormant failure	Exponential	Failure rate $\lambda_{\text{B}} = 2.5 \times 10^{-5}\, h^{-1}$
	Discharge	Uniform	Lower bound $\alpha_{\text{B}} = 8\, h$, upper bound $\beta_{\text{B}} = 10\, h$
Circuit breakers	Dormant failure	Exponential	Failure rate $\lambda_{\text{CB}} = 1.0 \times 10^{-6}\, h^{-1}$

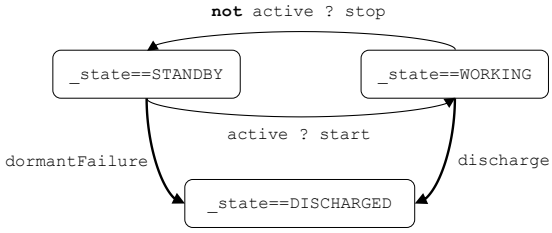
4.3 AltaRica Model

The first step in designing an AltaRica model consists usually in designing classes that encode the behaviors of the components of the system under study. Alternatively, these classes can be picked-up in libraries of on-the-shelf reusable modeling components.

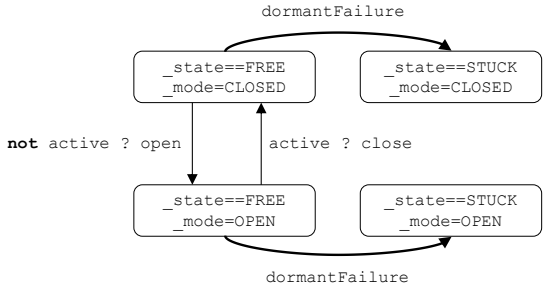
To describe (from scratch) our example, we have thus to design classes for the grid, the batteries and the circuit breakers. These classes are direct encoding of the guarded transition systems pictured in Figure 5.



(a) Grid



(b) Battery



(c) Circuit Breaker

Figure 5: State automata representing components of the power supply system.

For instance, Figure 6 shows the code for the guarded transition system pictured in Figure 5b that represents the behavior of batteries. This class involves one state variable, `_state`, and two Boolean flow variables, `active` and `outPower`. The variable `outPower` represents whether the battery actually provides power. The class `Battery` involves also four transitions, labelled respectively by events `start`, `stop`, `dormantFailure` and `discharge`. Probability distributions associated with these events are declared with parameters, to be able to change easily their values when the class is instantiated. In AltaRica 3.0, the values of state variables are modified via transitions, while the values of flow variables, that depend functionally on the former, are modified via assertions. The assertions are executed after each transition firing, in order to update the values of flow variables. In the code given in Figure 6, the assertion defines the value of `outPower`. The value of `active` is set up outside the component.

Once the behaviors of basic components described by means of classes, it is possible to describe the system under study as a whole, in a top-down way. AltaRica 3.0 provides the notion of block—prototypes in the sense of object-oriented theory—to do so. The different components of the system are then connected via assertions (or synchronizations). The reader interested in more details can refer to authors' article presenting AltaRica 3.0 [5].

Figure 7 shows the code for the power supply system.

```
domain BatteryState {STANDBY, WORKING, DISCHARGED}

class Battery
  BatteryState _state (init = STANDBY);
  Boolean active (reset = false);
  Boolean outPower (reset = false);
  event start (delay = Dirac(0));
  event stop (delay = Dirac(0));
  event dormantFailure (delay = exponential(lambda));
  event discharge (delay = uniform(alpha, beta));
  parameter Real lambda = 2.5e-5;
  parameter Real alpha = 8.0;
  parameter Real beta = 10.0;
  transition
    start: active and _state == STANDBY -> _state := WORKING;
    stop: not active and _state == WORKING -> _state := STANDBY;
    dormantFailure: _state == STANDBY -> _state := DISCHARGED;
    discharge: _state == WORKING -> _state := DISCHARGED;
  assertion
    outPower := _state == WORKING;
end
```

Figure 6: AltaRica 3.0 code for the class representing batteries

```
block PowerSupplySystem
  block MainSupply
    Grid G;
    CircuitBreaker CB;
    Boolean outPower(reset = false);
    assertion
      CB.inPower := G.outPower;
      outPower := CB.outPower;
  end
  block BackupSupply1
    Battery B;
    CircuitBreaker CB;
    Boolean isWorking(reset = false);
    Boolean active(reset = false);
    Boolean outPower(reset = false);
    assertion
      isWorking := B._state==WORKING and CB._state==FREE;
      B.active := active;
      CB.active := active;
      CB.inPower := G.outPower;
      outPower := CB.outPower;
  end
  clones BackupSupply1 as BackupSupply2;
  Boolean outPower(reset = false);
  assertion
    BackupSupply1.active := not MainSupply.outPower and BackupSupply1.isWorking;
    BackupSupply2.active := not MainSupply.outPower and not BackupSupply1.isWorking;
    outPower := MainSupply.outPower or BackupSupply1.outPower
                or BackupSupply2.outPower;
end
```

Figure 7: AltaRica 3.0 code for the power supply system

Models such as the one presented above are usually assessed by means of Monte-Carlo simulations: their highly dynamic nature prevents to assess them via the compilation into combinatorial models such as fault trees. Moreover, it is not possible to compile them into Markov chains, as they mix deterministic and stochastic transitions, the latter not always obeying Markovian hypotheses.

4.4 A Tricky Scenario

As just pointed out, the AltaRica 3.0 model presented in the previous section makes possible a fine grain analysis of the safety and the availability of the power supply system. In the actual model, maintenance policies are also taken into account, making the analysis even more accurate.

The expressive power of AltaRica 3.0 comes, however, with a price (when used fully): models must be carefully checked, so to verify that they actually encode the expected behavior of the system under study. The stepwise simulator plays a very important role to do so. We shall now illustrate our point by unwinding a tricky scenario.

In the initial state, six stochastic transitions are enabled (all exponentially distributed):

- `MainSupply.G.failure`, `MainSupply.CB.dormantFailure`,
- `BackupSupply1.B.dormantFailure`, `BackupSupply1.CB.dormantFailure`,
- `BackupSupply2.B.dormantFailure`, `BackupSupply2.CB.dormantFailure`.

In the full model, a few additional deterministic transitions representing periodic maintenance operations come in addition of the above ones.

If `MainSupply.G.failure` is fired, then the whole network needs to be reconfigured. The guards of the other failure transitions remain satisfied, but these transitions cannot be fired because of the three immediate reconfiguration transitions that are newly enabled:

- `MainSupply.CB.Open`,
- `BackupSupply1.B.start`, `BackupSupply1.CB.close`.

These immediate transitions must thus be fired prior to the firing of any other transition.

In the previous version of the stepwise simulator (that did not take into account the timed semantics), it was however possible for the analyst to fire both failure and reconfiguration transitions, making the debugging task tedious, to say the least.

After the firing of these immediate transitions, the busbar is powered by the first backup train. The following transitions are enabled:

- `MainSupply.G.repair`, `MainSupply.CB.dormantFailure`,
- `BackupSupply1.B.discharge`, `BackupSupply1.CB.dormantFailure`,
- `BackupSupply2.B.dormantFailure`, `BackupSupply2.CB.dormantFailure`.

If the transition `BackupSupply1.B.discharge` is fired, then three immediate reconfiguration transitions become enabled:

- `BackupSupply1.CB.open`,
- `BackupSupply2.B.start`, `BackupSupply2.CB2.close`.

As previously, these immediate transitions must be fired prior to the firing of any other transition. After the firing of these immediate transitions, the busbar is powered by the second backup train.

Now, five transitions have their guards satisfied:

- `MainSupply.G.repair`, `MainSupply.CB.dormantFailure`,

- `BackupSupply1.CB.dormantFailure`,
- `BackupSupply2.B.discharge`, `BackupSupply2.CB.dormantFailure`.

However, the transition `BackupSupply2.B.discharge` is not enabled. The reason is that the battery of the second backup supply has been activated after the discharge of the battery of the first backup supply. According to our reliability parameters, discharging both batteries takes at least $8 + 8 = 16$ hours. But the transition `MainSupply.G.repair`, which has been scheduled at the same date as the activation of the battery of the first supply, cannot take more than 12 hours. It follows that `MainSupply.G.repair` must be fired before `BackupSupply2.B.discharge`.

Table 4 gives a possible concrete execution corresponding to our scenario.

Table 4 A possible concrete execution of the power supply system

Step	Transition	Firing Date
1	`MainSupply.G.failure`	$0 + 1234.5 = 1234.5$
2	`MainSupply.CB.Open`	$1234.5 + 0 = 1234.5$
3	`BackupSupply1.B.start`	$1234.5 + 0 = 1234.5$
4	`BackupSupply1.CB.close`	$1234.5 + 0 = 1234.5$
5	`BackupSupply1.B.discharge`	$1234.5 + 9.2 = 1243.7$
6	`BackupSupply1.CB.open`	$1243.7 + 0 = 1243.7$
7	`BackupSupply2.B.start`	$1234.5 + 0 = 1234.5$
8	`BackupSupply2.CB.close`	$1243.7 + 0 = 1243.7$
9	`MainSupply.G.repair`	$1234.5 + 11.6 = 1246.1$

Figure 8 shows a timeline representing this execution (for the sake of simplicity, we did not represent dormant failures of circuit breakers).

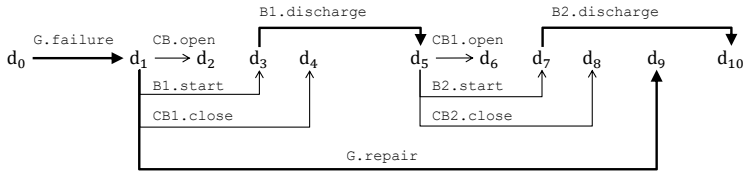


Figure 8: Timeline of the execution of the power supply system.

Note that in case the transition `BackupSupply1.CB.dormantFailure` is fired before `MainSupply.G.failure`, the scenario changes completely. Now, the first backup supply cannot be activated. Consequently, the second one is. Moreover, the discharge of its battery can occur before the grid is repaired.

On a small example, like the one presented here, it is still possible to do some bookkeeping of delays by hand. But when the model gets large, this is clearly impossible. This is the reason why, the introduction of the abstract semantics is of tremendous practical interest.

5 Conclusion

In this article, we introduced the notion of abstract execution of stochastic discrete event systems—taking a very general definition of the latter—that consists in abstracting (stochastic) delays associated with transitions into systems of linear inequalities. We showed that abstract and concrete executions are bisimilar: any concrete execution can be simulated by an abstract execution and reciprocally any abstract execution corresponds to at least one concrete execution. We introduced also the concept of timeline which proves to be very useful to reason on timed executions. Finally, we showed how to solve efficiently the generated systems of linear inequalities.

We illustrated the proposed approach via its implementation in the stepwise simulator of AltaRica 3.0. This latter tool makes it possible to debug and to validate complex behavioral models. The notion of abstract execution reconciles stochastic and stepwise simulations of AltaRica 3.0 models. We showed its practical interest by looking at tricky scenarios of an industrial case study mixing stochastic and deterministic transitions.

The introduction of abstract executions, already interesting on its own, paves the way to the design of efficient model-checking algorithms. In particular, we designed the prototype of a generator of full fledged sequences of events leading to a failure state, based on this abstract semantics. We plan to enhance this prototype, which is already available within the OpenAltaRica Platform, with model-checking functionalities.

References

[1] Christos G. Cassandras and Stéphane Lafortune. *Introduction to Discrete Event Systems*. Springer, New-York, NY, USA, 2008.

[2] Armin Zimmermann. *Stochastic Discrete Event Systems*. Springer, Berlin, Heidelberg, Germany, 2010.

[3] 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.

[4] Brigitte Plateau and William J. Stewart. Stochastic automata networks. In *Computational Probability*, pages 113–152. Kluwer Academic Press, 1997.

[5] Michel Batteux, Tatiana Prosvirnova, and Antoine Rauzy. Altarica 3.0 in 10 modeling patterns. *International Journal of Critical Computer-Based Systems*, 9(1–2):133–165, 2019.

[6] 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.

[7] 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.

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

[9] Wang Yi, Paul Pettersson, and Mats Daniels. Automatic verification of real-time communicating systems by constraint solving. In Dieter Hogrefe and Stefan Leue, editors, *Proceedings of the 7th Conference on Formal Description Techniques, FORTE'1994*, pages 223–238, Berne, Switzerland, October 1994. North–Holland.

[10] Kim G. Larsen, Paul Pettersson, and Wang Yi. Uppaal in a Nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, October 1997.

[11] Robin Milner. *Communication and Concurrency*. Prentice-Hall international series in computer science. Prentice Hall, Upper Saddle River, New Jersey, USA, 1989.

[12] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, Cambridge, MA, USA, February 2000.

[13] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. MIT Press, Cambridge, MA, USA, June 2008.

[14] Norman Matloff and Peter Jay Salzman. *The Art of Debugging with GDB, DDD, and Eclipse*. No Starch Press, San Fransisco, CA, USA, 2008.

[15] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, New York, NY, USA, 1977. ACM Press. Los Angeles, California.

[16] Gerard J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison Wesley, Boston, MA 02116, USA, September 2003.

[17] Vašek Chvàtal. *Linear Programming*. W. H. Freeman and Company, New-York, NY, USA, 1983.

[18] Alexander Schrijver. *Theory of Linear and Integer Programming*. Wiley Series in Discrete Mathematics & Optimization. John Wiley & sons, Hoboken, New Jersey, USA, 1998.

[19] 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.

[20] Jean-Pierre Signoret and Alain Leroy. *Reliability Assessment of Safety and Production Systems: Analysis, Modelling, Calculations and Case Studies*. Springer Series in Reliability Engineering. Springer, Cham, Switzerland, 2021.

[21] Marta Kwiatkowska, Gethin Norman, and David Parker. PRISM 4.0: Verification of probabilistic real-time systems. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Proceedings 23rd International Conference on Computer Aided Verification (CAV'11)*, volume 6806 of *Lecture Notes in Computer Science*, pages 585–591. Springer, April 2011.

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

[23] Kishor S. Trivedi. *Probability and Statistics with Reliability, Queuing, and Computer Science Applications*. Wiley-Blackwell, Hoboken,New Jersey, USA, 2001.

[24] Michel Batteux, Tatiana Prosvirnova, and Antoine Rauzy. From models of structures to structures of models. In *IEEE International Symposium on Systems Engineering (ISSE 2018)*, Roma, Italy, October 2018. IEEE.

[25] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[26] Edward Lynn Kaplan and Paul Meier. Nonparametric estimation from incomplete observations. *Journal of American Statistics Association*, 53(282):457–481, 1958.

[27] Andrei N. Kolmogorov. *Grundbegriffe der Wahrscheinlichkeitsrechnung*. Springer, Berlin, Germany, 1933.

[28] Michel Batteux, Tatiana Prosvirnova, and Antoine Rauzy. Altarica 3.0 assertions: the why and the wherefore. *Journal of Risk and Reliability*, 231(6):691âŁ“–700, September 2017.