Specification and Analysis of Soft Real-Time Systems: Quantity and Quality

Pedro R. D'Argenio* Joost-Pieter Katoen Ed Brinksma
Dept. of Computer Science. University of Twente. P.O. Box 217. 7500 AE Enschede. The Netherlands.
{*dargenio,katoen,brinksma*}*@cs.utwente.nl*

Abstract

This paper presents a process algebra for specifying soft real-time constraints in a compositional way. For these soft constraints we take a stochastic point of view and allow arbitrary probability distributions to express delays of activities. The semantics of this process algebra is given in terms of stochastic automata, a variant of timed automata where clocks are initialised randomly and run backwards. To analyse quantitative properties, an algorithm is presented for the on-the-fly generation of a discrete-event simulation model from a process algebra specification. On the qualitative side, a symbolic technique for classical reachability analysis of stochastic automata is presented. As a result a unifying framework for the specification and analysis of quantitative and qualitative properties is obtained. We discuss an implementation of both analytic methods and specify and analyse a fault-tolerant multi-processor system.

1. Introduction

The design and analysis of various types of systems, like embedded systems or communication protocols, require insight in not only the functional, but also in the real-time and performance aspects of applications involved. Research in formal methods has recognised the need for the additional support of quantitative aspects, and various initiatives have been taken to accomplish such support. A prominent example is the treatment of real-time constraints, where specification formalisms like timed automata [2] have emerged, and impressive progress has been made in the development of efficient verification algorithms [21, 5]. This has resulted in a number of tools (model checkers) that provide interesting experimental platforms.

The real-time constraint that one considers in this setting is typically 'hard', for instance, "the system must always do a certain activity before time t". For many applications, though, real-time constraints are typically less stringent. Rather than requiring that certain activities *must always* occur before time t, in practice one is usually interested in more 'soft' real-time constraints, where a system is required to perform the activity *mostly* before t. In this paper we concentrate on such soft real-time constraints. The soft real-time requirements of systems typically have to do with their performance characteristics, and are often also referred to as their quality-of-service parameters. They are usually related to stochastic aspects of various forms of time delay, such as, for example, mean and variance of message transfer delay, service waiting times, failure rates, utilisations, etc.

Traditionally, there has been a clear separation between the functional and performance aspects of systems, and as a result different communities have constructed and analysed their own, largely unrelated models for the aspects under their responsibility. In modern systems, though, the difference between functional and performance features has become blurred, and both features are becoming of comparable interest. Thus, it would be beneficial to be able to check how changes in functionality affect performance issues, and vice versa. In addition, one would like to have a better relationship between the models that are used for qualitative and quantitative analysis, and avoid the use of different models for different aspects that are mutually incompatible. A single framework where both aspects could be defined would be highly desirable.

In this paper we take a stochastic point of view with respect to soft real-time constraints. Typical constraints that we support are of the form: "the system should perform an activity before time t in 92% of the cases". We propose a high-level specification language for soft real-time systems. Here, state changes take place at discrete points in time, but the time of occurrence of activities is controlled by random variables. In contract to most formalisms that are restricted to a particular set of probability distributions, like negative exponential or discrete distributions, we support arbitrary distributions, discrete or continuous. This makes the language more expressive and more interesting from a practical point of view. The language is based on process algebra and has been christened SPADES (Stochastic Process Algebra

*Supported by the NWO/SION project 612-33-006.

for Discrete-Event Simulation, symbolised by ⏢). The use of a process algebra facilitates the description of systems in a modular and well-structured way. The algebraic nature of the language allows reasoning about specifications in an equational way, thus facilitating step-wise design and minimisation.

Stochastic automata, a variant of timed automata [2] where clocks are initialised randomly and run backwards, are used as the underlying semantic model. These automata have an interpretation in measure theory, and due to the possibly continuous nature of probability distributions, the resulting interpretation model is infinite. We will, however, show that for checking qualitative properties, in particular reachability analysis — the key technique in checking safety properties — a symbolic algorithm at the level of finite stochastic automata suffices. Reasoning about such properties can thus take place without delving into the measure theory underlying the formalism. In fact, it turns out that such an analysis can be viewed as being carried out on ordinary labelled transition systems. Consequently, well-known techniques can be applied to reduce the complexity of the reachability analysis. In the prototype tool-implementation, partial-order reduction techniques [11] are, for instance, applied.

In addition we will show how a stochastic simulation model can be obtained from a ⏢-specification in an automatic way. This facility enables a discrete-event simulation that gathers statistics about the system specification to be carried out. An interesting aspect of this algorithm is that the modularity of ⏢ facilitates the "on-the-fly" generation of the simulation model in the sense that the state space is constructed dynamically and requires minimal storage. This means that we are not forced to construct the entire stochastic automaton a priori, as it suffices to store only the current state, and generate new states when they are needed.

A prototype implementation of the simulation and reachability algorithms has been made and several case-studies have been specified and analysed: the IEEE 1394 root-contention protocol [27], several classical queueing systems known from performance analysis, and a dynamic wavelength reconfiguration in optical networks [26].

Organisation of the paper. Section 2 introduces the process algebra ⏢ and stochastic automata. Probabilistic transition systems are presented in Section 3. Section 4 presents the discrete-event simulation. Section 5 covers the reachability analysis. A fault-tolerant multiprocessor case-study is described in Section 6. Section 7 concludes the paper.[1]

Related work. Since 1990, many extensions of process algebras have been investigated in which the delay of actions is determined by (continuous) distribution functions. In languages like TIPP [15], PEPA [17] and EMPA [3] exponential distributions are used. Due to the memoryless property of exponential distributions the semantics of these languages can be adequately described using labelled transition systems that closely resemble continuous-time Markov chains. In fact, our approach can be considered as generalising this line of research in the direction of simulation. We support arbitrary distributions and combine simulation with qualitative analysis.

Another process algebra for discrete-event simulation has been presented in [13] and applied to a cache coherency protocol in [9]. The semantic objects are infinite. To simulate a specification it is translated into `C++` and some simulation libraries are used. Although their work is related to ours, we use a different process algebra, allow nondeterminism and use the concept of adversaries [28, 24] for its resolution, and obtain (for most processes) finite stochastic automata. Recently, an alternative process algebra to derive simulation models, called GSMPA [6], was proposed. To our knowledge for this language there is no tool support available. No support for qualitative analysis is incorporated in [6, 13, 9].

Other works that relate simulation models (or languages) to process algebra are [23, 4]. In these works, the approach is different: rather than generating a simulation automatically from a process algebra specification (as we do), they use process algebra as a semantical model for simulation languages. In addition, these works do not take probabilistic timing into consideration.

2. The stochastic process algebra ⏢

Syntax. Let $\mathbf{A}$ be a set of *actions*, $\mathbf{V}$ a set of *process variables*, and $\mathcal{C}$ a set of clocks with $(x, G) \in \mathcal{C}$ for x a clock name and G an arbitrary probability distribution function satisfying $G(t) = 0$ for $t < 0$. We abbreviate (x, G) by x_G.

Definition 1. The syntax of ⏢ is defined by:

$$p \quad ::= \quad \mathbf{stop} \;\big|\; a;p \;\big|\; C{\mapsto}p \;\big|\; p+p \;\big|\; \{\!|C|\!\}p \;\big|\; p\,||_A\,p \;\big|\; p[f] \;\big|\; X.$$

where $C \subseteq \mathcal{C}$ is finite, $a \in \mathbf{A}$, $A \subseteq \mathbf{A}$, $f : \mathbf{A} \to \mathbf{A}$, and $X \in \mathbf{V}$. A *recursive specification* E is a set of recursive equations of the form $X = p$ for each $X \in \mathbf{V}$, where $p \in$ ⏢. E is called *guarded* if for every recursive equation $X = p$ in E, all process variables in p appear in a sub-term of the form $a; q$. □

The basic process $\mathbf{stop}$ cannot perform any action. The process $a; p$ can immediately perform an action a and then behaves like p. Process $C{\mapsto}p$ behaves like p after expiration of all clocks in C. Process $p + q$ behaves either as p

[1] A preliminary version was presented at the PAPM workshop (Tech. Rep., Univ. Verona, pp 85–102, 1998).

or q, but not both. During execution the fastest process, i.e. the process that is enabled first, is selected. This is known as the *race condition*. If this fastest process is not uniquely determined, a non-deterministic selection among the fastest processes is made. $\{\!|C|\!\}p$ behaves like p after all clocks in C have been initialised according to their distribution function. $||$ stands for parallel composition. In process $p||_A q$, processes p and q perform actions autonomously, but actions in A should be synchronised. Finally, the process $p[f]$ behaves like p except that actions are renamed by function f. We abbreviate $\{\!|x_G|\!\}\{x_G\}\mapsto a; P$ by $a(x_G); P$.

Stochastic automata. The semantics of our process algebra is defined in terms of stochastic automata, a model that is related to timed automata [2] and generalised semi-Markov processes (GSMPs, [10]).

Definition 2. A *stochastic automaton* is a tuple $(\mathcal{S}, s_0, \mathcal{C}, \mathbf{A}, \longrightarrow, \kappa, F)$ where $\mathcal{S}$ is a non-empty set of *locations*, $s_0 \in \mathcal{S}$ is the *initial location*, $\mathcal{C}$ is a (countable) set of *clocks*, $\mathbf{A}$ is a set of *actions*, $\longrightarrow \subseteq \mathcal{S} \times (\mathbf{A} \times \wp_{\mathrm{fin}}(\mathcal{C})) \times \mathcal{S}$ is the set of *edges*, $\kappa : \mathcal{S} \to \wp_{\mathrm{fin}}(\mathcal{C})$ is the *clock-setting function*, and $F : \mathcal{C} \to (\mathbb{R} \to [0,1])$ is the *clock-distribution function*. □

We denote $(s, a, C, s') \in \longrightarrow$ by $s \xrightarrow{a,C} s'$, use x and y to denote clocks, and abbreviate $F(x)$ by F_x. To each location s a finite set of clocks $\kappa(s)$ is associated. As soon as location s is entered any clock x in this set is initialised randomly according to its probability distribution function F_x. Once initialised, the clocks start counting down, all with the same rate. A clock expires if it has reached the value 0. The occurrence of an action is controlled by the expiration of clocks. Thus, whenever $s \xrightarrow{a,C} s'$ and the system is in location s, action a can happen as soon as all clocks in the set C have expired. The next location will then be s'.

Semantics. To associate a stochastic automaton $SA(p)$ to a given term p in 🔔 we define the different components of $SA(p)$[2]. In order to define the automaton associated to a parallel composition, we introduce the additional operation $\overline{\mathsf{ck}}$. $\overline{\mathsf{ck}}(p)$ is a process that behaves like p except that no clock is set at the very beginning. As usual in structured operational semantics, a location corresponds to a term. Thus, the set of locations equals 🔔$\cup \{\overline{\mathsf{ck}}\}$. The clock setting function κ is defined by induction on the structure of expression:

$$\begin{array}{l}
\kappa(\mathbf{stop}) = \kappa(a; p) = \kappa(\overline{\mathsf{ck}}(p)) = \varnothing \\
\kappa(C{\mapsto}p) = \kappa(p[f]) = \kappa(p) \\
\kappa(p + q) = \kappa(p \,||_A\, q) = \kappa(p) \cup \kappa(q) \\
\kappa(\{\!|C|\!\}p) = C \cup \kappa(p) \\
\kappa(X) = \kappa(p) \text{ for } X = p
\end{array}$$

[2]Here we assume that p does not contain any name clashes of clock variables. This is not a severe restriction since any term that suffers from a name clash can be properly renamed into a term without a name clash [7].

Table 1. Stochastic automata for 🔔

$$a; p \xrightarrow{a,\varnothing} p \qquad \frac{p \xrightarrow{a,C} p'}{\begin{array}{l} p + q \xrightarrow{a,C} p' \\ q + p \xrightarrow{a,C} p' \end{array}}$$

$$\frac{p \xrightarrow{a,C'} p'}{\{\!|C|\!\}p \xrightarrow{a,C'} p'} \qquad \frac{p \xrightarrow{a,C} p'}{p[f] \xrightarrow{f(a),C} p'[f]}$$

$$\frac{p \xrightarrow{a,C'} p'}{C{\mapsto}p \xrightarrow{a,C\cup C'} p'} \qquad \frac{p \xrightarrow{a,C} p'}{\overline{\mathsf{ck}}(p) \xrightarrow{a,C} p'}$$

$$\frac{p \xrightarrow{a,C} p'}{\begin{array}{l} p \,||_A\, q \xrightarrow{a,C} p' \,||_A\, \overline{\mathsf{ck}}(q) \\ q \,||_A\, p \xrightarrow{a,C} \overline{\mathsf{ck}}(q) \,||_A\, p' \end{array}} \; (a \notin A)$$

$$\frac{p \xrightarrow{a,C} p' \quad q \xrightarrow{a,C'} q'}{p \,||_A\, q \xrightarrow{a,C\cup C'} p' \,||_A\, q'} \; (a \in A)$$

The set of edges $\longrightarrow$ between locations is defined as the smallest relation satisfying the rules in Table 1. The function F is defined by $F(x_G) = G$ for each clock x in p. The other components are defined as for the syntax of 🔔.

Stochastic automata and 🔔 are equally expressive [7]. This means that for any (finitely branching[3]) stochastic automaton a corresponding (guarded recursive) term in the language can be given in which the reachable part of its stochastic automaton is identical to the stochastic automaton at hand.

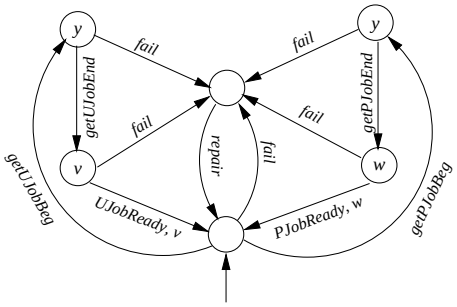
Example 3. Consider a processor that can process user jobs (like database transactions) and programmer jobs (like compilation). Jobs are queued according to the FIFO principle; there is a single queue per job type. There is no priority on the processing of jobs: if there is a job of both types to be processed, a job is selected non-deterministically. Transferring a job from the queue to the processor takes d time units with a fluctuation of ϵ, distributed uniformly. (To model this, the activity is split into two actions and a delay is incorporated between them.) After loading a job, the processor executes it. The execution time of a job is distributed according to a γ-distribution with parameters (a, a') for user jobs and with (b, b') for programmer jobs. The system is subject to failures. When a failure occurs, the processor aborts its activity. When the system is repaired, which takes a certain γ-distributed delay, the processor restarts in its initial state. A 🔔 specification of this

[3]A stochastic automaton is finitely branching if for every location the set of outgoing edges is finite.

system is: $P ||_{\{fail, repair\}}$ *Maintain*, where

$$
\begin{aligned}
P &= getUJobBeg; (getUJobEnd(y_G); PWU + PF) \\
&\quad + getPJobBeg; (getPJobEnd(y_G); PWP + PF) \\
&\quad + PF \\
PWU &= UJobReady(v_{\gamma(a,a')}); P + PF \\
PWP &= PJobReady(w_{\gamma(b,b')}); P + PF \\
PF &= fail; repair; P \\
Maintain &= fail; repair(z_{\gamma(c,c')}); Maintain.
\end{aligned}
$$

Here G stands for a uniform distribution on $[d-\epsilon, d+\epsilon]$. The stochastic automaton corresponding to *P* is:



□

3. Probabilistic transition systems

Timed automata have a formal interpretation in terms of timed transition systems where states keep information about the current location and the values of the clocks [2]. Similarly, stochastic automata have a formal semantics in terms of a probabilistic transition system (PTS). We define the notion of a PTS and show how locations and values of clocks will come into play. A PTS is related to the alternating model of [12]. We assume some familiarity with basic measure theory; for an introduction see [20]. Let Ω be a sample space and $\mathcal{F}$ be a σ-algebra on Ω.

Definition 4. A *probabilistic transition system* (PTS) is a tuple $(\Sigma, \Omega, \sigma_0, \mathcal{L}, T, \longrightarrow)$ where Σ and Ω are disjoint sets of *states*, $\sigma_0 \in \Sigma$ is the *initial state*, $\mathcal{L}$ a set of *labels*, $T : \Sigma \to (\mathcal{F} \to [0,1])$ the *probabilistic transition relation*, and $\longrightarrow \subseteq \Omega \times \mathcal{L} \times \Sigma$ is the *labelled (or non-deterministic) transition relation*, such that

$$\forall \sigma \in \Sigma.\, T(\sigma) \text{ is a probability measure on } \mathcal{F}$$

□

Σ is the set of *probabilistic* states and Ω the set of *non-deterministic* states. Since T is defined as a (total) function, each probabilistic state has exactly one outgoing transition to a function on σ-algebras. Intuitively, in general there is a continuum of transitions each attached with a certain probability. We write $\sigma' \xrightarrow{\ell} \sigma$ for $\langle \sigma', \ell, \sigma \rangle \in \longrightarrow$, and $\sigma' \not\xrightarrow{\ell}$ for $\neg \exists \sigma.\, \sigma' \xrightarrow{\ell} \sigma$.

Since we are interested in the timing of actions, let $\mathcal{L} = \mathbf{A} \times \mathbb{R}_{\geqslant 0}$, where $\mathbf{A}$ is a set of action names and $\mathbb{R}_{\geqslant 0}$ is the set of non-negative real numbers. We denote $a(d)$ instead of (a, d) for $(a, d) \in \mathcal{L}$. The meaning of $\sigma \xrightarrow{a(d)} \sigma'$ is that action a occurs after the system has been idle for d time units in state σ, and state σ changes into σ'.

In the following we give two interpretations of stochastic automata in terms of PTSs and describe their difference.

Closed interpretation. In order to study the performance characteristics of a system, it is usually regarded as a *closed* system, i.e. a system which is complete by itself and which needs no external interaction. Typically, a closed system consists of the components of the intended system together with the environment with which it interacts. In this closed system view there is no need to delay activities any further once they are enabled, since there will be no further (external) processes that can delay their execution. Formally, this means that closed systems display the *maximal progress* property. This is made explicit in the following interpretation.

For clock x let $v(x) \in \mathbb{R}$ denote the value of x; function v is called a *valuation*. Let $\mathcal{V}$ be the set of all valuations on $\mathcal{C}$. For $d \in \mathbb{R}_{\geqslant 0}$, we define valuation $v-d$ by $(v-d)(x) \stackrel{\text{def}}{=} v(x)-d$, for all clocks x.

Let $SA = (\mathcal{S}, s_0, \mathcal{C}, \mathbf{A}, \rightarrow, \kappa, F)$ be a stochastic automaton and n the cardinality of $\mathcal{C}$. A probabilistic state is a pair consisting of a location and a valuation. The set $\mathcal{S} \times \mathbb{R}^n$ is the set of non-deterministic states and acts as the sample set for the underlying σ-algebra for which we take the Borel algebra $\mathcal{B}(\mathcal{S} \times \mathbb{R}^n)$ [20]. Notice that for any location s and valuation v there is a unique tuple $(s, v(x_1), \ldots, v(x_n)) \in \mathcal{S} \times \mathbb{R}^n$. We denote such elements simply by $[s, v]$. For convenience we use the predicate $\mathsf{exp}_d(v, C)$ which is true iff all clocks in C have expired in v after d time units, i.e.

$$\forall x \in C.\, (v-d)(x) \leqslant 0$$

and the predicate $\mathsf{mpr}_d(s, v, C)$ which is true iff there is no possibility to leave s within d time units, i.e. for all $d' \in [0, d)$ we have

$$\forall s, b, C'.\, s \xrightarrow{b,C'} \Leftrightarrow \exists y \in C'.\, (v-d')(y) > 0$$

Definition 5. The *closed interpretation of SA* in the initial valuation v_0, denoted $\mathcal{C}[\![\, SA \,]\!]^{v_0}$, is the PTS

$$(\mathcal{S} \times \mathcal{V}, \mathcal{S} \times \mathbb{R}^n, (s_0, v_0), \mathbf{A} \times \mathbb{R}_{\geqslant 0}, T, \longrightarrow)$$

where the probabilistic transition relation T is defined by:

$$T(s, v) \stackrel{\text{def}}{=} P^s_v \tag{1}$$

where P_v^s is the unique probability measure on $\mathcal{B}(\mathcal{S} \times \mathbb{R}^n)$ induced by the distribution functions $F_0 \stackrel{\text{def}}{=} \mathrm{I}_s$ and $F_i \stackrel{\text{def}}{=}$ **if** $x_i \in \kappa(s)$ **then** F_{x_i} **else** $\mathrm{I}_{v(x_i)}$, with $0 < i \leqslant n$ and I being the indicator function defined by $\mathrm{I}_d(d') \stackrel{\text{def}}{=}$ **if** $d = d'$ **then** 1 **else** 0.

For $d \in \mathbb{R}_{\geqslant 0}$, relation $\longrightarrow$ is defined by:

$$\frac{s \xrightarrow{a,C} s' \ \wedge \ \mathsf{exp}_d(v, C) \ \wedge \ \mathsf{mpr}_d(s, v, C)}{[s, v] \xrightarrow{a(d)} (s', v-d)} \tag{2}$$

□

An edge $s \xrightarrow{a,C} s'$ is enabled in valuation v, which we denote $\mathsf{enabled}(s \xrightarrow{a,C} s', v)$, if it induces a non-probabilistic transition from $[s, v]$. In particular, notice that $s \xrightarrow{a,\varnothing} s'$ is enabled for any valuation v.

Rule (1) is concerned with the setting of the clocks. Since the values of clocks are assigned randomly, a probabilistic transition corresponds to this step. Clocks in $\kappa(s)$ randomly take a value according to their associated distribution function. The indicator functions take care that the system stays in the same location and that the values of clocks that are not intended to be set (i.e. those not in $\kappa(s)$) remain unchanged.

Rule (2) deals with the triggering of an edge. If we have an edge $s \xrightarrow{a,C} s'$, in which action a occurs at time d, and all clocks in C have expired at time d, and there is no edge that has all its clocks expire before d, then the edge is triggered. Note that maximal progress is ensured by the last predicate.

Open interpretation. In order to study reachability properties like freedom from deadlock, it is important to observe how the system behaves in an arbitrary context. That is, the interaction of a system with a certain "well-behaved" component may not induce a deadlock, while a "badly-behaved" component could take the system through an undesired path that will end in a deadlock situation. For this reason the interpretation of a stochastic automata as a closed system is not sufficient.

If we interpret a stochastic automaton as an open system we let the system interact with its environment. The environment can be a user or another system. Basically, an open system is a component of a larger system. In an open system, an action that is enabled may not be executed until the environment is also ready to execute such an action. Therefore, an activity may not take place as soon as it is enabled. In other words, the maximal progress property is no longer valid.

Definition 6. The *open interpretation of SA* in the initial valuation v_0, denoted $\mathcal{O}[\![\, SA \,]\!]^{v_0}$, is the PTS

$$(\mathcal{S} \times \mathcal{V}, \mathcal{S} \times \mathbb{R}^n, (s_0, v_0), \mathbf{A} \times \mathbb{R}_{\geqslant 0}, T, \longrightarrow)$$

where T is obtained as in Definition 5, and $\longrightarrow$ is defined for non-negative d by the rule

$$\frac{s \xrightarrow{a,C} s' \ \wedge \ \mathsf{exp}_d(v, C)}{[s, v] \xrightarrow{a(d)} (s', v-d)} \tag{3}$$

□

Notice that the only difference between the open and closed semantics is that the constraint of maximal progress is present in (2) but not in (3).

4 Discrete event simulation

A system specification in Modest contains functional and quantitative aspects. In order to understand the impact of the stochastic delays in the specification on measures of interest like throughput and response time, we consider the analysis of a Modest specification. Since arbitrary distributions are allowed in Modest, we cannot use analytical or numerical techniques as they are applicable only in restricted cases, e.g., when all delays are governed by negative exponential distributions. We therefore take a more general approach and use *simulation*, in particular discrete-event simulation, where in contrast to continuous-time simulation techniques, state changes take place at discrete points in time — but time itself is continuous. In a simulation, runs (also called sample paths) are generated, and on the basis of these runs data is gathered and analysed to determine (an estimation of) the desired measure of interest. The reliability of the estimate is given by a confidence interval. This approach will be illustrated later; here we address the problem of generating simulation runs from a stochastic automaton, expressed as a Modest specification. The main problem we must address is the resolution of possible non-determinism in a stochastic automaton. Although it is widely recognised that non-determinism is of significant importance in a step-wise design methodology for the purpose of under-specification, it must be resolved when a simulation is carried out. We discuss how non-determinism is resolved for PTSs, and for stochastic automata.

Runs and adversaries. A run of a PTS $\mathcal{T} = (\Sigma, \Omega, \sigma_0, \mathcal{L}, T, \longrightarrow)$ is a path obtained by traversing $\mathcal{T}$ starting from its initial state σ_0.

Definition 7. A *run* ρ of $\mathcal{T}$ is a (finite or infinite) sequence $\sigma_0\sigma_0'\ell_1\sigma_1\sigma_1'\ell_2 \ldots \ell_n\sigma_n\sigma_n'$ for $n \in \mathbb{N} \cup \{\infty\}$ such that, for all $0 \leqslant i < n$: (i) $\frac{\partial}{\partial\omega}F(\sigma_i') > 0$, (ii) $\sigma_i' \xrightarrow{\ell_{i+1}} \sigma_{i+1}$, and (iii) if ρ is finite and non-empty, then $\sigma_n' \in \Omega$. □

Here, $\frac{\partial}{\partial\omega}F$ can be interpreted as the density function of F.[4] Constraint (i) states that probabilistic steps should be probable ones, as we want the simulator to generate runs with

[4] Because we allow arbitrary distributions, the definition of the differential operator $\frac{\partial}{\partial\omega}$ is quite involved [20].

positive probability only. Constraint (ii) is self-explanatory, and constraint (iii) states that a non-empty and finite run should end in a non-deterministic state. If run ρ is finite, then we let $\mathit{last}(\rho)$ denote its final state. We denote the set of finite runs of $\mathcal{T}$ by $\mathit{Runs}(\mathcal{T})$.

Non-determinism is useful for under-specifying "how often" an alternative is chosen. This information is usually not available in the early steps of the design, or it is deliberately left unspecified. If we are to study the performance of such system specifications, the idea is to impose an additional mechanism — called a *scheduler* or *adversary* [28, 24] — on top of the system. If the system has reached a state in which a choice must be made between several non-deterministic possibilities, the adversary will make the choice. One thus considers a system (in our case a PTS $\mathcal{T}$), that contains a certain implementation freedom, in the context of an adversary $\mathcal{A}$. $\mathcal{T}$ can be viewed as the system specification, and $\mathcal{A}$ as the representation of the architecture on which the system is realised. The pair $(\mathcal{T}, \mathcal{A})$ is thus the entire system under consideration. Furthermore, the simulation data for $\mathcal{T}$ that is obtained should be considered with respect to the adversary $\mathcal{A}$.

Definition 8. An *adversary* $\mathcal{A}$ is a partial function $\mathit{Runs}(\mathcal{T}) \to ((\longrightarrow) \to [0,1])$ such that for all $\rho \in \mathit{Runs}(\mathcal{T})$ for which the (countable) sample space $\Omega(\rho)$ is a non-empty subset of $\left\{\sigma' \xrightarrow{\ell} \sigma \mid \mathit{last}(\rho) = \sigma'\right\}$. $\mathcal{A}(\rho) \stackrel{\text{def}}{=} P$ for some discrete probability measure P on the (discrete) σ-algebra $\wp(\Omega(\rho))$. □

This notion can be lifted to stochastic automata as follows: a run of a stochastic automaton is a run of its underlying (closed) semantics $\mathcal{C}[\![\, SA \,]\!]^{v_0}$. Notice that for a given non-deterministic state $[s, v]$, the next transition is fully determined by v and the outgoing edges from s.

Definition 9. Let $SA = (\mathcal{S}, s_0, \mathcal{C}, \mathbf{A}, \to, \kappa, F)$ be a stochastic automaton with closed interpretation $\mathcal{T} = \mathcal{C}[\![\, SA \,]\!]^{v_0}$. An *adversary* $\mathcal{A}$ for SA is a partial function $\mathit{Runs}(\mathcal{T}) \to ((\to) \to [0,1])$ such that for any run $\rho \in \mathit{Runs}(\mathcal{T})$ for which the (countable) sample space $\Omega'(\rho)$ is a non-empty subset of: $\left\{s \xrightarrow{a,C} s' \mid \mathit{last}(\rho) = [s, v] \wedge \mathsf{enabled}(s \xrightarrow{a,C} s', v)\right\}$. $\mathcal{A}(\rho) \stackrel{\text{def}}{=} P$ for some discrete probability measure P on the (discrete) σ-algebra $\wp(\Omega'(\rho))$. □

The simulation algorithm. The simulation algorithm is implemented as a *variable time-advance procedure* [25]. In this procedure time steps are of varying length and there is an event in every simulated time step. The simulation is controlled by the occurrence of "next events" and the simulation time between the occurrence of two events is "skipped".

In our setting, the simulation algorithm requires the following inputs: (i) a specification E representing the system, (ii) an adversary $\mathcal{A}$ that resolves non-determinism in E, and (iii) the initial process p_0. It is assumed that the initial valuation v_0 equals 0 for all clocks. (For a process p_0 that does not contain free clock variables this poses no restriction.) The detailed structure of the simulation algorithm is depicted in Figure 1.

Since in our semantics (cf. Table 1) a location corresponds to a term, simulation can be carried out on the basis of expressions rather than using their semantic representation. This means that the stochastic automaton is not entirely generated a priori but only the parts that are required to choose the next step. The simulation starts in state (p_0, v_0), the initial state of the (closed) PTS underlying $SA(p_0)$. Once started, the stochastic automaton is constructed in an on-the-fly fashion on the basis of the current term p_i (i.e. location) and the input specification E. From term p_i the set of clocks $\kappa(p_i)$ to be set is determined (by module (A) in Figure 1) and the set of possible next edges is computed according to the inference rules of Table 1 (by module (B)).

To compute the next valuation we only need to keep track off the last valuation v_i. Each clock x_G in $\kappa(p_i)$ is assigned a random value according to the distribution G, while the other clocks remain unchanged (this is done by module (C)). This step corresponds to the rule (1) in Definition 5.

Given the new valuation and the set of possible edges, we now want to select an edge. From the set of possible edges (calculated by module (B)), the subset of enabled edges is selected. This step corresponds to rule (2) in Definition 5. From this set of enabled edges (if any), an edge is selected by the adversary $\mathcal{A}$ in a probabilistic fashion. This is done by module (D).

The actual traversal of the selected edge is carried out by module (E). This involves the calculation of the next step, as defined by rule (2). More precisely, module (E) determines the executed action a_i and its timing d_i plus the next state (p_{i+1}, v_{i+1}) with $v_{i+1} = v'_i - d_i$. Starting from this state the next step in the run is determined by module (B).

Finally, the measure of interest (e.g., throughput, utilisation and response time) needs to be computed from the generated simulation run. For that purpose, information is gathered from the run and analysed. This is done by module (F). This component is user-driven, since the calculations to be performed are determined by the user.

Prototype implementation. We have implemented a prototype of the simulation algorithm using the functional language Haskell 1.4 [14]. In this implementation we have confined ourselves to guarded recursive specifications. Unguarded recursion could yield an infinite set of outgoing

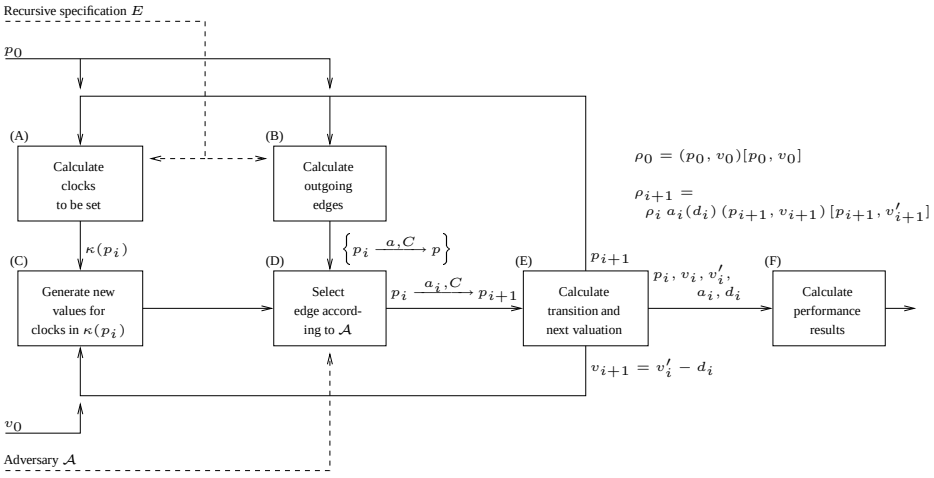


Figure 1. Schema of the simulation algorithm

edges or an infinite set of clock settings, in which case modules (D) and (E) would never finish their computations. In the current implementation we only deal with adversaries that are history-independent in the sense that if adversary $\mathcal{A}$ satisfies the property $\mathit{last}(\rho) = \mathit{last}(\rho')$ then $\mathcal{A}(\rho) = \mathcal{A}(\rho')$. Hence, we can probabilistically select an enabled next edge, using only the current location and not the locations visited before. We are therefore not required to store a complete run. This assumption reduces the space complexity of the simulation algorithm. We discuss some improvements to this treatment of adversaries in the final section of the paper. Notice that modules (C) and (D) require the use of "randomness" for which we use a random number generator. This is a multiplicative linear congruential generator with modulus $m = 2^{31} - 1$ and multiplier $a = 16807$ and is based on Schrage's algorithm [19].

5. Reachability analysis

Complementary to the quantitative analysis described above, we discuss in this section a classical analysis technique for functional correctness — reachability analysis. Reachability analysis is the key technique in proving safety properties (often characterised as properties of the type "something bad can never happen"). A typical reachability property is the absence of a deadlock, which is a state from which no further progress can be made. In order to check such properties for stochastic automata, and thus ⍾ terms, the underlying semantics in terms of probabilistic transition systems needs to be examined. However, even for finite terms, these transition systems are infinite due to the fact that distributions are continuous. We therefore consider a *symbolic* reachability analysis. Using a symbolic analysis we can avoid having to build and examine the infinite underlying PTS. Instead, we check at the level of stochastic automata. We investigate this for the open and closed interpretation of stochastic automata and confine ourselves to finite stochastic automata.

Foundations. We first define the notions of reachability in a PTS and in a stochastic automaton and investigate their correspondence.

Definition 10. Let $\mathcal{T} = (\Sigma, \Omega, \sigma_0, \mathcal{L}, T, \longrightarrow)$ be a PTS. State $\sigma' \in \Omega$ is *reachable* if and only if there exists a finite run $\rho \in \mathit{Runs}(\mathcal{T})$ such that $\mathit{last}(\rho) = \sigma'$. The set of reachable states of $\mathcal{T}$ is denoted $Reach(\mathcal{T})$. □

Definition 11. Let $SA = (\mathcal{S}, s_0, \mathcal{C}, \mathbf{A}, \rightarrow, \kappa, F)$ be a stochastic automaton. A *symbolic run* of *SA* is a finite sequence $s_0 a_1 C_1 s_1 \ldots s_{n-1} a_n C_n s_n$, $n \geqslant 0$, such that, for all $0 < i \leqslant n$, $s_{i-1} \xrightarrow{a_i, C_i} s_i$. □

Location s is reachable if there exists a symbolic run that ends in s. The set of reachable locations of *SA* is denoted $Reach(SA)$.

Lemma 12. Let *SA* be a stochastic automaton with open interpretation $\mathcal{O}[\![\, SA \,]\!]^{v_0}$. Then:

$$s \xrightarrow{a,C} s' \;\Leftrightarrow\; \forall v \in \mathcal{V}.\, (\exists d \in \mathbb{R}_{\geqslant 0}.\, [s, v] \xrightarrow{a(d)} (s', v{-}d))$$

Stated in words: there is a transition from location s to s' if and only if in the open semantics there is a transition from non-deterministic state $[s, v]$ to probabilistic state $(s', v{-}d)$, for any valuation v. The "⇐" part of the lemma follows in a straightforward way: there can only be a transition in the semantics if there is a corresponding edge in the stochastic automaton. The "⇒" part holds, because if $s \xrightarrow{a,C} s'$ for a given valuation v, there always exists a sufficiently large d such that $v{-}d$ is at most 0 for all clocks in C (e.g. let d be the maximum clock value in v of all clocks in C). Thus,

$\mathsf{exp}_d(v, C)$ holds. Consequently there is a transition in the open semantics, due to rule (3) in Definition 6. As a result, every symbolic run of *SA* has a corresponding finite run in $\mathcal{O}[\![\, SA \,]\!]$, and vice versa.

Theorem 13.

$$s \in Reach(SA) \Leftrightarrow \exists v \in \mathcal{V}.\, [s, v] \in Reach(\mathcal{O}[\![\, SA \,]\!]^{v_0})$$

We will now consider the closed interpretation. Recall that, as opposed to the open interpretation, in the closed interpretation an edge can only be taken, if there is no earlier point in time at which the current location can be left (maximal progress). As a consequence, certain edges present in the stochastic automaton need not result in a transition in the underlying closed PTS, since there exist competitive edges that are "faster" and thus will be taken instead. Technically speaking, the "$\Rightarrow$" part of Lemma 12 does not hold anymore. Instead we have the following lemma:

Lemma 14. Let *SA* be a stochastic automaton with closed interpretation $\mathcal{C}[\![\, SA \,]\!]^{v_0}$ for $v_0 \in \mathcal{V}$. For any $v \in \mathcal{V}$:

1. if $[s, v] \xrightarrow{a(d)} (s', v{-}d)$ for some $d \in \mathbb{R}_{\geqslant 0}$ then $s \xrightarrow{a,C} s'$ for some $C \subseteq \mathcal{C}$
2. if $[s, v] \xrightarrow{a(d)}\!\!\!\!\!/\;\;$ for all $a \in \mathbf{A}$, $d \in \mathbb{R}_{\geqslant 0}$, then $s \xrightarrow{b,C}\!\!\!\!\!/\;\;$, for all $b \in \mathbf{A}$, $C \subseteq \mathcal{C}$

The first part of the lemma follows immediately from rule (2) in Definition 5. The second part can be proven by contradiction. Suppose that $s \xrightarrow{a,C} s'$. Then (as we argued above) it follows $\mathsf{exp}_d(v, C)$. For the smallest $\hat{d}$ for which $\mathsf{exp}_{\hat{d}}(v, C)$ holds, it follows $\mathsf{mpr}_{\hat{d}}(s, v, C)$, and by rule (2) it follows $[s, v] \xrightarrow{a(\hat{d})} (s', v{-}\hat{d})$. Contradiction.

As a result, every finite run of $\mathcal{C}[\![\, SA \,]\!]$ has a corresponding symbolic run of *SA* (but not the reverse).

Theorem 15.

$$s \notin Reach(SA) \Rightarrow \forall v \in \mathcal{V}.\, [s, v] \notin Reach(\mathcal{C}[\![\, SA \,]\!]^{v_0})$$

This result is e.g., sufficient to check for freedom of deadlock: if *SA* does not have a reachable deadlock state, $\mathcal{C}[\![\, SA \,]\!]$ is deadlock-free.

These results allow us to carry out reachability analysis at a purely symbolic level, i.e., without the construction of the underlying infinite probabilistic transition system and without using the clock information in the stochastic automaton. In this way we can exploit existing tools like SPIN [18] for carrying out the reachability analysis. Furthermore, given the highly expressive power of ⌂ (by which we mean the support of arbitrary distributions), this is the best one can achieve symbolically. If we want more information, like the probability of a deadlock, we can either resort to discrete-event simulation or to model checking techniques similar to those for timed automata. The latter are, however, only applicable to highly restricted classes of distribution functions [1].

Prototype implementation. The above theorems provide the formal basis for applying reachability analysis to stochastic automata. While the discrete-event simulation algorithm can be applied to any (guarded) specification in ⌂, reachability analysis can, for obvious reasons, only be applied to terms that give rise to finite stochastic automata. These terms are defined by the following syntax:

$$\begin{array}{lcl} q & ::= & \mathbf{stop} \;\big|\; a; q \;\big|\; C{\mapsto}q \;\big|\; q + q \;\big|\; \{\!|C|\!\}q \;\big|\; X \\ p & ::= & q \;\big|\; q[f] \;\big|\; p \,||_A\, p \end{array}$$

Terms constructed using the first clause are sequential processes; terms constructed using the second clause are parallel processes. The current implementation requires that the recursive specification that defines the process variables contains finitely many guarded recursive equations of the form $X = q$. That is, a specification must have only finitely many process variables and each of them must be defined by guarded sequential processes.

The reachability analysis algorithm is implemented in Haskell 1.4 [14] as part of the discrete-event simulator. The algorithm has two inputs: a parallel process (the system specification) and a characterisation of the location(s) to be checked for reachability. If the given location is reachable, the algorithm returns the symbolic execution that ends in the location. The implementation is based on a selective state space search using a partial-order reduction technique based on persistent sets [22, 11]. The advantage of this technique is that the stochastic automata needs to be constructed only when it is demanded. In this way, completely constructing the stochastic automaton for every sequential process in the specification can be avoided.

6. A fault-tolerant multiprocessor

To illustrate our approach to the specification and analysis of a soft real-time system, we consider a fault-tolerant multiprocessor system. This example is adopted from [16] where all activities are delayed according to a negative exponential distribution. Instead, we will use arbitrary distributions[5] , and analyse qualitative and quantitative properties using our prototype tool. The architecture of the multiprocessor system is presented in Figure 2.

The processors in the system are of the type described in Section 2 and are able to process two types of jobs: programmer and user jobs. These jobs are generated by load

[5] Distribution functions for the example were chosen arbitrarily with the intention of showing the versatility of ⌂ and the prototype implementation.

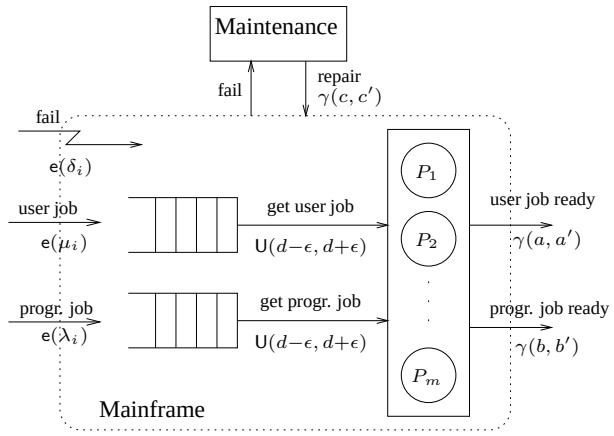


Figure 2. Architecture of the multiprocessor system

processes, one per type of job, and one load process that generates failures. The variation of the load in time — thus distinguishing between peak loads, low loads and no load — is determined by a component "change phase" that is only incorporated for modelling purposes. User and programmer jobs are queued. When a failure occurs, the system components (except the load generating components) are immediately halted. They are restarted as soon as the failure is repaired.

Compositional specification. The system consists of three parts: the mainframe itself, the maintenance module, and the system load. Accordingly:

$$\mathit{System} = \mathit{Load} \,||_L\, (\mathit{Mainframe} \,||_F\, \mathit{Maintain})$$

where $L = \{usrJob, prgJob, fail\}$ and $F = \{fail, repair\}$. Process *Load* models the user and programmer load, and the failure occurrences:

$$\mathit{Load} = \mathit{PLd}_1 \,||_{\{cg\}}\, \mathit{ULd}_1 \,||_{\{cg\}}\, \mathit{FLd}_1 \,||_{\{cg\}}\, \mathit{ChPhase}$$

Process *ChPhase* models the variation of the load (action cg) in time. Phases change according to a Weibull distribution function with parameters (v, w) (denoted by $\mathsf{W}(v, w)$).

$$\mathit{ChPhase} = cg(x_{\mathsf{W}(v,w)}); \mathit{ChPhase}$$

There are three phases. In the first phase, user jobs arrive according to an exponential distribution with rate μ_1 (notation $\mathsf{e}(\mu_1)$); in the second phase, arrivals are distributed according to $\mathsf{e}(\mu_2)$, and in the third phase no user job is generated. In any case, if a job cannot be queued because either the queue is full or the system has failed, the job is simply rejected (action rj). Similarly, programmer jobs arrive according to $\mathsf{e}(\lambda_1)$ and $\mathsf{e}(\lambda_2)$ in the first and second phase, and failures originate according to $\mathsf{e}(\delta_1)$ and $\mathsf{e}(\delta_2)$, respectively. We model the occurrence of a system failure regardless how many errors induce that failure. Process *ULd* is specified as follows; the processes *PLd* and *FLd* are defined in a similar way.

$$\begin{aligned}
\mathit{ULd}_1 &= nxtUJob(xu_{\mathsf{e}(\mu_1)}); (UJob; \mathit{ULd}_1 + rj; \mathit{ULd}_1) \\
&\quad + cg; \mathit{ULd}_2 \\
\mathit{ULd}_2 &= nxtUJob(xu_{\mathsf{e}(\mu_2)}); (UJob; \mathit{ULd}_2 + rj; \mathit{ULd}_2) \\
&\quad + cg; \mathit{ULd}_3 \\
\mathit{ULd}_3 &= cg; \mathit{ULd}_1
\end{aligned}$$

The *Mainframe* consists of *Queues* and processors P_i. The different processes are synchronised with the actions *fail* and *repair*: when a failure occurs the complete system must stop until it is repaired. Each processor is defined as in Section 2. In addition, the *Queues* communicate with the processors each time the processors get either a user or programmer job from the queue in order to process it.

$$\mathit{Mainframe} = \mathit{Queues} \,||_{G \cup F}\, (P_1 \,||_F \cdots ||_F\, P_m)$$

where $G = \{getUJobBeg, getPJobBeg\}$. The queues for storing user jobs and programmer jobs are simple FIFO queues and are defined in a standard way. The definitions are omitted here.

Obtaining an adversary. We observe the following:

(*a*) In process *ULd* non-determinism may arise between actions *UJob* and *rj*. We prefer not to reject a user job if there exists a chance that the mainframe may become available at the same moment. The same consideration applies to *PLd* and *FLd*.

(*b*) A failure is an arbitrary event that at any moment may disturb the normal execution of the system. For that reason, failures are handled as soon as possible.

(*c*) User jobs are usually short activities, such as saving a file or processing a small database transaction, that have to be processed as soon as possible. Programmer jobs are more complicated tasks that may involve compilation, simulation or testing of a system.

Given these observations we define a priority relation $\prec$ as the least (strict partial) order satisfying:

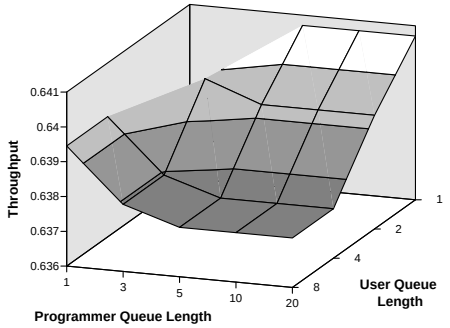
$$\begin{array}{lcl}
rj \prec a & \Leftrightarrow & a \neq rj \\
a \prec fail & \Leftrightarrow & a \neq fail \\
getPJobBeg & \prec & getUJobBeg
\end{array}$$

These priority relations are used to define an adversary. If non-determinism remains after reducing the possible activities to be executed according to the defined priorities, then the adversary resolves it according to a (discrete) uniform probability distribution. Formally, the sample space $\Omega'(\rho)$ is defined as:

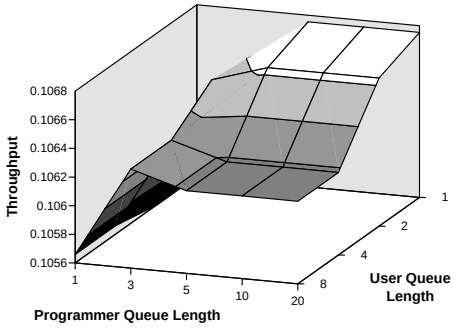
$$\left\{ s \xrightarrow{a,C} s' \mid \mathit{last}(\rho) = [s, v] \wedge \mathsf{enabled}(s \xrightarrow{a,C} s', v) \right\}$$

Table 2. Parameters for the simulation

System	Load		Processing	
$m = 4$	$\mu_1 = .033$	$\mu_2 = 2$	$d = .021$	$e = .001$
$nu = 4$	$\lambda_1 = .0167$	$\lambda_2 = .16$	$a = .167$	$a' = .5$
$np = 10$	$v = 300$	$w = 6$	$b = .167$	$b' = 2.0$



(a) Throughput User Jobs



(b) Throughput Programmer Jobs

Figure 3. Studying the length of the queues

Let *pri*(ρ) be the set of maximal elements in $\Omega'(\rho)$ according to the order $\prec$, i.e., *pri*(ρ) is:

$$\left\{ s \xrightarrow{a,C} s' \in \Omega'(\rho) \mid \neg\exists s \xrightarrow{b,C'} s'' \in \Omega'(\rho) \wedge a \prec b \right\}$$

The adversary is then simply defined as follows:

$$\mathcal{A}(\rho)(e) \stackrel{\text{def}}{=} \textbf{if } e \in \textit{pri}(\rho) \textbf{ then } \frac{1}{\#\textit{pri}(\rho)} \textbf{ else } 0$$

Simulation results. We set the values of the different parameters according to Table 2. As in [16], we studied the behaviour of the system with different queue lengths. We ran several simulations changing the length of the queues (with $\delta_1 = 0.0007$, $\delta_2 = 0.00035$, and $c' = 100$). We can see in Figure 3 that both user and programmer job throughput stabilise when the user and programmer queue length

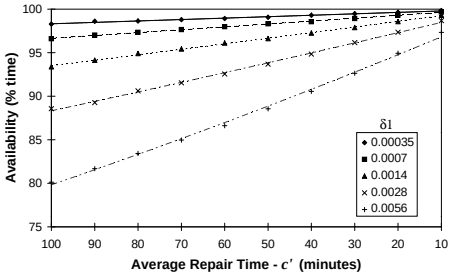


Figure 4. Availability

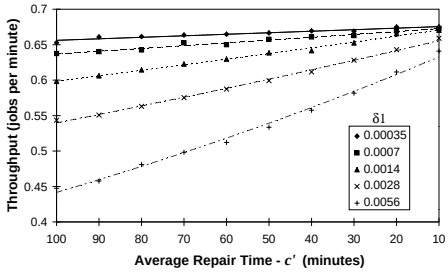


Figure 5. Throughput User Jobs

are at least 4 and 5, respectively (notice that the planes Figure 3 become horizontal from that point on). As queues of larger capacity do not affect the throughput, we take $nu = 4$ and $np = 10$ (see Table 2).

Different simulations have been carried out while changing the parameters related to failure and repairing. In all cases we took $\delta_2 = \delta_1/2$. For the repair time we take $c = 1$. Hence, the average repair time equals c'. The simulation results are depicted in Figures 4 and 5. Figure 4 represents the availability of the system, that is, the percentage of time the mainframe is processing jobs. Figure 5 depicts the throughput of user jobs, i.e. the number of jobs that are processed successfully per time unit.

To calculate the user job throughput, we simply count the number of occurrences of action *UJobReady* per time unit. To determine the availability we count the occurrences of the action *fail* per time unit, say *fpm*, and then calculate $100 \cdot (1 - \textit{fpm} \cdot c')$. Since c' is the average repair time, $\textit{fpm} \cdot c'$ is the fraction that the system is unavailable per time unit.

The simulations have been carried out using the method of *batch means*. It consists of running a long simulation run, discarding the initial transient interval, and dividing the remainder of the run into several batches or subsamples [19]. We took 20 subsamples, each one of approximately 150000 minutes length. The values in the figures are the overall averages. In every case, we calculated the respective confidence interval. The (proportionally) widest confidence interval was obtained for $\delta_1 = 0.0056$ and $c' = 100$ in the case of the throughput: $0.4473999 \pm 5.468 \cdot 10^{-4}$ with 99% of confidence. In the case of counting failures, the widest confidence interval was for $\delta_1 = 0.00035$ and $c' = 10$:

$1.63726 \cdot 10^{-4} \pm 5.44 \cdot 10^{-9}$ with confidence 99%.

Reachability analysis. Since the multiprocessor system is finite, we are able to automatically check reachability properties. We checked using the prototype that the process *Mainframe* is deadlock-free and does not have clock name clashes.

7. Concluding remarks

In this paper we presented a high-level description language for soft (i.e., stochastic) real-time systems that is based on process algebra. The compositional nature of the language facilitates the description of such systems in a modular and well-structured way. We have presented a discrete-event simulation algorithm that allows to gather statistics about the system specification. The simulation algorithm takes as input a process algebra specification and an adversary to resolve non-determinism, and automatically generates simulation runs. This quantitative analysis technique is complemented by an on-the-fly reachability analysis algorithm. As a result, a unifying algebraic framework for the specification and analysis of quantitative and qualitative properties is obtained. This has been illustrated by treating a fault-tolerant multiprocessor system. Future work will address extensions and refinement of our methods, e.g. specification of adversaries (along the lines of [8]), model checking, and extending the current tool implementation.

References

[1] R. Alur, C. Courcoubetis, and D. Dill. Model-checking for probabilistic real-time systems. *Automata, Languages and Programming*, LNCS 510, pp 115–126, 1991.

[2] R. Alur and D. Dill. A theory of timed automata. *Th. Comp. Sc.*, **126**:183–235, 1994.

[3] M. Bernardo and R. Gorrieri. A tutorial on EMPA: A theory of concurrent processes with nondeterminism, priorities, probabilities and time. *Th. Comp. Sc.*, **202**(1-2):1–54, 1998.

[4] G.M. Birtwistle and C. Tofts. Operational semantics for process-based simulation languages. Part 2: μDemos. *Trans. of The Society for Comp. Simulation*, **11**(4):303–336, 1994.

[5] A. Bouajjani, S. Tripakis and S. Yovine. On-the-fly symbolic model-checking for real-time systems. In *RTSS*, 1997.

[6] M. Bravetti, M. Bernardo and R. Gorrieri. Towards performance evaluation with general distributions in process algebras. In *CONCUR*, LNCS 1466, pp 405–422, 1998.

[7] P.R. D'Argenio, J.-P. Katoen, and E. Brinksma. An algebraic approach to the specification of stochastic systems (extended abstract). In *PROCOMET*, pp 126–147, 1998.

[8] P.R. D'Argenio, H. Hermanns, and J.-P. Katoen. On generative parallel composition. *El. Notes in Th. Comp. Sc.*, **22**, 1999.

[9] A.J. Field, P.G. Harrison, and K. Kanani. Automatic generation of verifiable cache coherence simulation models from high-level specifications. *Australian Comp. Sc. Comms.*, **20**(3):261–275, 1998.

[10] P.W. Glynn. A GSMP formalism for discrete event simulation. *Proc. of the IEEE*, 77(1):14–23, 1989.

[11] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems*, LNCS 1032, 1996.

[12] H. Hansson and B. Jonsson. A calculus for communicating systems with time and probability. In *RTSS*, pp 278–287, 1990.

[13] P. Harrison and B. Strulo. Stochastic process algebra for discrete event simulation. In *Quantitative Methods in Parallel Systems*, pp 18–37. Springer, 1995.

[14] Report on the programming language Haskell: A non-strict, purely functional language (Version 1.4), 1997. URL: `http://haskell.org/`.

[15] H. Hermanns, U. Herzog, and V. Mertsiotakis. Stochastic process algebras – between LOTOS and Markov chains. *Comp. Netw. and ISDN Sys.*, **30**(9/10): 901–924, 1998.

[16] U. Herzog and V. Mertsiotakis. Stochastic process algebras applied to failure modelling. In *Process Algebra and Performance Modelling, PAPM'94*, pp 107–126, 1994.

[17] J. Hillston. *A Compositional Approach to Performance Modelling*. Cambridge Univ. Press, 1996.

[18] G. Holzmann. The model checker SPIN. IEEE Trans. on Softw. Eng. **23**(5), 279–295, 1997.

[19] R. Jain. *The Art of Computer Systems Performance Analysis*. John Wiley & Sons, 1991.

[20] S. Lang. *Real and Functional Analysis*, volume 142 of *Graduate Texts in Mathematics*. Springer-Verlag, 1993.

[21] K.G. Larsen, P. Pettersson and Y. Wang. Compositional symbolic model checking of real-time systems. *RTSS*, pp 76–87, 1995.

[22] W.T. Overmans. *Verification of Concurrent Systems: Functions and Timing*. PhD thesis, UCLA, 1981.

[23] R.J. Pooley. Integrating behavioural and simulation modelling. In *Quantitative Evaluation of Computing and Communication Systems*, LNCS 977, pp 102–116, 1995.

[24] R. Segala and N. Lynch. Probabilistic simulations for probabilistic processes. *Nordic J. of Computing*, **2**(2):250–273, 1995.

[25] G.S. Shedler. *Regenerative Stochastic Simulation*. Academic Press, 1993.

[26] F.N.C. Slothouber, S.M. Heemstra de Groot and I.G.M.M. Niemegeers. Performance impact of dynamic wavelength reconfiguration in WDM access networks. *Conf. on Optical Network Design and Modeling*, pp 380–393, 1999.

[27] M.I. Stoelinga and F.W. Vaandrager. Root contention in IEEE 1394. In *Formal Methods for Real-Time and Probabilistic Systems*, LNCS 1601, pp 53–74, 1999.

[28] M.Y. Vardi. Automatic verification of probabilistic concurrent finite state programs. *Found. of Comp. Sc.*, pp 327–338, 1985.