General Purpose Discrete Event Simulation using ♤

Pedro R. D'Argenio[1]*, Joost-Pieter Katoen[2], and Ed Brinksma[1]

[1] Dept. of Computer Science. University of Twente.
P.O.Box 217. 7500 AE Enschede. The Netherlands.
{*dargenio,brinksma*}*@cs.utwente.nl*

[2] Lehrstuhl für Informatik VII. University of Erlangen-Nürnberg.
Martensstrasse 3. D-91058 Erlangen. Germany.
katoen@informatik.uni-erlangen.de

Abstract. We discuss the use of the stochastic process algebra ♤ (*spades*) for discrete event simulation, and report on a prototype simulation algorithm that has been implemented. The use of process algebraic techniques in ♤ offers a number of advantages with respect to existing simulation techniques. The compositional nature of ♤ allows an on-the-fly construction of the simulation model, and hence only the current state need to be saved. ♤, moreover, allows us to model and simulate on a general purpose basis, as the simulation algorithm applies to any well founded process specified in ♤. Finally, specifications in ♤ are also amenable to traditional process algebraic analysis of their functional behaviour, like e.g. absence of deadlocks. We illustrate our result with a simulation example of a nontrivial system.

1 Introduction

Simulation is an important technique for analysing the behaviour and performance of systems. If the system of interest is not yet available for analysis, as is often the case during the design stage, a simulation model provides an easy way to predict the performance or compare several alternatives [15]. Computer systems, even when they have real-time behaviour, can usually be described by discrete-event models due to the fact that (the relevant part of) their state does not change continuously, but in a discrete way.

Discrete-event simulation can be carried out using general purpose languages, simulation languages, or simulation packages. Relatively little effort has been made to characterise the formal behaviour of the models described using those techniques. Exceptions are [18, 3]. The formal aspect, however, is quite important since it helps to obtain a correct specification of the system to be studied. This weakness, among others, has recently drawn the attention of many researchers into extending process algebras with stochastic and real-time features [14, 9, 11, 2, 6, . . .]. Two approaches are mainly followed: one is focussed on the purely

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

analytical study by restricting to the so-called Markovian-process algebras [14, 9, 2,. . .], the other addresses a more general case by accepting simulation as part of the analysis process [11, 16, 6, 7].

Recently, in [6], we introduced a simple extension of automata with stochastic information by borrowing ideas from timed automata [1] and generalised semi-Markov processes [8]. In addition, we introduced a stochastic process algebra for discrete event simulation, named *spades* and denoted by ♤.

In this paper we discuss the formal foundations of a simulation algorithm for stochastic automata. Since the semantics of ♤ is given in terms of stochastic automata, as an immediate consequence we can simulate any system described in ♤. In fact, there is a real advantage in implementing the algorithm directly for ♤. The simulation algorithm can generate the stochastic automaton *on-the-fly* from a given ♤ term, that is, only those parts of the stochastic automaton that are actually going to be used are generated, and moreover, in the moment they are needed and not in advance. So, since systems described in ♤ can be specified compositionally using recursive specifications, simulation of infinite-state processes is, in principle, possible.

Using ♤ as simulation modelling language has two additional advantages. On the one hand, ♤ allows, in principle, the description of any system which makes the simulation and modelling task general-purpose oriented. By "general-purpose" we mean that, in contrast to what happens with many simulation packages, ♤ does not restrict to a particular methodology such as queuing networks. On the other hand, ♤ is a stochastic process algebra and hence provides a formal verification framework which allows for the formal analysis of the functional correctness of the model to be studied.

To consolidate our results we give a (non-trivial) example of application of the simulation algorithm. We discuss the modeling of a multiprocessor mainframe with failures [13] and analyse its performance using the implementation of the simulation algorithm.

The paper is organised as follows. In Sections 2 and 3 we recall the stochastic automaton model, its semantics, and the process algebra ♤. Section 4 introduces the simulation algorithm: its foundations and a description of its (current) implementation. The multiprocessor mainframe example is studied in Section 5. In Section 6 we discuss related work and conclude the paper.

Acknowledgements. The alternative semantics for stochastic automata (see Section 2) resulted from a question posed by Mariëlle Stoelinga. We thank Victor Nicola for always answering our questions and the referees for their constructive remarks.

2 The Stochastic Automaton Model

In this section we discuss the semantic basis for our simulation algorithm. We recall the definition of stochastic automata and we give its semantics in terms of probabilistic transition systems.

Before starting and in order to fix notation we mention that $\mathbb{N}$ denotes the set of non-negative integers, $\mathbb{R}$ the set of real numbers, and $\mathbb{R}_{\geq 0}$ the set of non-negative reals. For $n \in \mathbb{N}$, we write $\mathbb{R}^n$ for the nth Cartesian product of $\mathbb{R}$.

Stochastic Automata. The stochastic automaton model was introduced in [5] and its theoretical concerns were discussed in [6]. It is an extension of the traditional automaton model and allows to describe processes with stochastic information. The basic idea is borrowed from generalised semi-Markov processes [8] and timed automata [1].

Definition 1. A *stochastic automaton* is a tuple $(\mathcal{S}, s_0, \mathcal{C}, \mathbf{A}, \longrightarrow, \kappa, F)$ where:

- $\mathcal{S}$ is a set of *locations* with $s_0 \in \mathcal{S}$ being the *initial location*.
- $\mathcal{C}$ is a (countable) set of *clocks*.
- $\mathbf{A}$ is a set of *actions*.
- $\longrightarrow \subseteq \mathcal{S} \times (\mathbf{A} \times \wp_{\text{fin}}(\mathcal{C})) \times \mathcal{S}$ is the set of *edges*. We denote the edge $(s, a, C, s') \in \longrightarrow$ by $s \xrightarrow{a,C} s'$ and call C its *trigger set*.
- $\kappa : \mathcal{S} \to \wp_{\text{fin}}(\mathcal{C})$ is the *clock-setting function*.
- $F : \mathcal{C} \to (\mathbb{R} \to [0,1])$ assigns to each clock a *distribution function* such that $F(x)(t) = 0$ for $t < 0$; we write F_x instead of $F(x)$.

Notice that each clock $x \in \mathcal{C}$ is a random variable with distribution F_x. □

As soon as a location s is entered, every clock x in $\kappa(s)$ is initialised according to its probability distribution function F_x. Once initialised, 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. The semantics of a stochastic automaton is given in terms of a probabilistic transition system (PTS for short). In the following we give the definition of PTS and define the semantics of stochastic automata.

Definition 2. Let Ω be a *sample space* and $\mathcal{F}$ be a *σ-algebra* on Ω. A *probabilistic transition system* is a tuple $(\Sigma, \Omega, \sigma_0, \mathcal{L}, T, \longrightarrow)$ where

1. Σ and Ω are two disjoint sets of *states*, with the *initial state* $\sigma_0 \in \Sigma$. States in Σ are called *probabilistic* and states in Ω *non-deterministic*.
2. $\mathcal{L}$ is a set of *labels*.
3. $T : \Sigma \to (\mathcal{F} \to [0,1])$ is the *probabilistic transition relation* such that for all $\sigma \in \Sigma$, $T(\sigma)$ is a *probability measure* on $\mathcal{F}$.
4. $\longrightarrow \subseteq \Omega \times \mathcal{L} \times \Sigma$ is the *labelled (or non-deterministic) transition relation*. We denote $\sigma' \xrightarrow{\ell} \sigma$ for $\langle \sigma', \ell, \sigma \rangle \in \longrightarrow$, and $\sigma' \overset{\ell}{\not\longrightarrow}$ for $\neg\exists\sigma.\, \sigma' \xrightarrow{\ell} \sigma$. □

Notice that T is defined as a (total) function. Hence, each probabilistic state has exactly one outgoing transition.

Since our interest is to deal with time information using PTSs, the set of labels we will use is $\mathcal{L} = \mathbf{A} \times \mathbb{R}_{\geq 0}$, where $\mathbf{A}$ is a set of action names and $\mathbb{R}_{\geq 0}$ is the set of non-negative real numbers, which are intended to denote the (relative) time at which an action takes place. We usually denote $a(d)$ instead of (a, d) whenever $(a, d) \in \mathcal{L}$ and it means "action a occurs right after the system has been idle for d time units".

In the following, we define the semantics of stochastic automata. In order to study the performance of a system, it is regarded as a *closed system*, that is, a system which is considered complete by itself for which no external interaction is needed. In this kind of system one not only models the components of the intended system but also the environment with which it interacts. In this way, the activity of the whole system can take place as soon as it becomes ready to be executed since there is no external agent that may delay its execution. That is, closed systems respond to the *maximal progress* property. In [6] we referred to this interpretation as the *actual behaviour*.

A *valuation* is a function $v : \mathcal{C} \to \mathbb{R}$. Let $\mathcal{V}$ be the set of all valuations. For $d \in \mathbb{R}_{\geq 0}$, we define $v - d$ by $\forall x \in \mathcal{C}.\ (v - d)(x) \overset{\text{def}}{=} v(x) - d$.

Let $SA = (\mathcal{S}, s_0, \mathcal{C}, \mathbf{A}, \longrightarrow, \kappa, F)$ be a stochastic automaton. Let n be the cardinality of $\mathcal{C}$. We take as sample space the set $\mathcal{S} \times \mathbb{R}^n$, and as σ-algebra the respective Borel algebra $\mathcal{B}(\mathcal{S} \times \mathbb{R}^n)$. Notice that for each $s \in \mathcal{S}$ and $v \in \mathcal{V}$ there is a unique tuple $(s, v(x_1), v(x_2), \ldots, v(x_n)) \in \mathcal{S} \times \mathbb{R}^n$. To simplify notation, we denote such elements of $\mathcal{S} \times \mathbb{R}^n$ by $[s, v]$. Other elements will be irrelevant.

Definition 3. The *interpretation* of SA in the initial valuation v_0 is given by the PTS $I_{v_0}(SA) \overset{\text{def}}{=} ((\mathcal{S} \times \mathcal{V}), \mathcal{S} \times \mathbb{R}^n, (s_0, v_0), (\mathbf{A} \times \mathbb{R}_{\geq 0}), T, \longrightarrow)$, where T and $\longrightarrow$ are obtained as follows[1]

The probabilistic transition relation T is defined by

$$\textbf{Prob}\quad T(s, v) \overset{\text{def}}{=} P_v^s$$

where P_v^s is the unique probability measure on $\mathcal{B}(\mathcal{S} \times \mathbb{R}^n\mathbb{R}^n)$ induced by the following distribution functions:

$$F_0 \overset{\text{def}}{=} \mathrm{I}_s \qquad\qquad F_i \overset{\text{def}}{=} \textbf{ if } x_i \in \kappa(s) \textbf{ then } F_{x_i} \textbf{ else } \mathrm{I}_{v(x_i)}$$

with $0 < i \leq 1 + n$ and I being the *indicator function* defined by $\mathrm{I}_d(d') \overset{\text{def}}{=} \textbf{if}$ $d = d'$ **then** 1 **else** 0.

The non-deterministic transition relation $\longrightarrow$ is defined by the rule

$$\textbf{Act}\quad \frac{s \xrightarrow{a,C} s' \qquad d \in \mathbb{R}_{\geq 0} \qquad \forall x \in C.\ (v - d)(x) \leq 0 \qquad \forall d' \in [0, d).\ \forall s \xrightarrow{b,C'}.\ \exists y \in C'.\ (v - d')(y) > 0}{[s, v] \xrightarrow{a(d)} (s', (v - d))}$$

An edge $s \xrightarrow{a,C} s'$ is enabled in valuation v, notation $enabled(s \xrightarrow{a,C} s', v)$ if it induces a non-deterministic transition outgoing from $[s, v]$. In particular, notice that $s \xrightarrow{a,\emptyset} s'$ is enabled for any valuation v. □

[1] The semantics given in Definition 3 uses a different methodology from the one defined in [6]. Nonetheless, both semantics can be proven to be probabilistically bisimilar.

Notice that, for each location s and valuation v there is exactly one probabilistic transition. So, for any stochastic automaton SA and any valuation v_0, $I_{v_0}(SA)$ is indeed a PTS.

Rule **Prob** considers the setting of the clocks. Since the values of the clocks are assigned randomly, a probabilistic transition corresponds to this step. The indicator functions take care that the system stays in the same location and that the value of clocks which are not meant to be set (i.e., those *not* in $\kappa(s)$) remains unchanged. Instead, clocks in $\kappa(s)$ may randomly take a value according to their associated distribution function. Rule **Act** explains the case of triggering an edge. So, for the occurrence of an action a at time d according to an edge $s \xrightarrow{a,C} s'$, we check that all the clocks in the trigger set C have already expired at time d. This part is considered by the satisfaction of the predicate $\forall x \in C.\ (v-d)(x) \leq 0$. Moreover, it should be the case that no edge was enabled before. That is, any edge must have an active (i.e. positive) clock at any valuation "previous" to $v-d$. In this way, the edge is forced to occur as soon as it becomes enabled. The maximal progress is checked by the formula $\forall d' \in [0,d).\ \forall s \xrightarrow{b,C'}.\ \exists y \in C'.\ (v-d')(y) > 0$.

3 The stochastic process algebra $\spadesuit$

In this section, we recall the definition and semantics of the stochastic process algebra $\spadesuit$ (read *spades*) [5, 6]. The main difference between $\spadesuit$ and the majority of other stochastic process algebras, is that $\spadesuit$ deals with any kind of probability distribution instead of restricting only to exponential distributions.

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 any probability distribution function satisfying $G(t) = 0$ for $t < 0$. We abbreviate (x, G) by x_G.

Definition 4. The syntax of $\spadesuit$ is defined by:

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

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 $p \in \spadesuit$. $\square$

Process **stop** represents inaction; it is the process that cannot perform any action. The intended meaning of $a;p$ (named *(action-)prefixing*) is that action a is immediately enabled and once it is performed the behaviour of p is exhibited. $C \mapsto p$ is the *triggering condition*; process p becomes enabled as soon as all the clocks in C expire. $p + q$ is the *choice*; it behaves either as p or q, but not both. At 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. The *clock-setting operation* $\{\!|C|\!\}p$ sets the clocks in C according to their respective distribution functions. We choose a LOTOS-like parallel composition.

Table 1. Stochastic automata for ♤ $(X = p \in E)$

$$\kappa(\mathbf{stop}) = \emptyset \qquad \kappa(C \mapsto p) = \kappa(p) \qquad \kappa(\{\!|C|\!\}p) = C \cup \kappa(p)$$

$$\kappa(a;p) = \emptyset \qquad \kappa(p[f]) = \kappa(p) \qquad \kappa(p+q) = \kappa(p) \cup \kappa(q)$$

$$\kappa(\overline{\mathsf{ck}}(p)) = \emptyset \qquad \kappa(X) = \kappa(p) \qquad \kappa(p||_A q) = \kappa(p) \cup \kappa(q)$$

$$a;p \xrightarrow{a,\emptyset} p \qquad \frac{p \xrightarrow{a,C} p'}{p+q \xrightarrow{a,C} p' \quad q+p \xrightarrow{a,C} p'} \qquad \frac{p \xrightarrow{a,C} p'}{p||_A q \xrightarrow{a,C} p'||_A\overline{\mathsf{ck}}(q) \quad q||_A p \xrightarrow{a,C} \overline{\mathsf{ck}}(q)||_A p'}\ a \notin A$$

$$\frac{p \xrightarrow{a,C'} p'}{\{\!|C|\!\}p \xrightarrow{a,C'} p'}$$

$$\frac{p \xrightarrow{a,C'} p'}{C \mapsto p \xrightarrow{a,C\cup C'} p'} \qquad \frac{p \xrightarrow{a,C} p'}{p[f] \xrightarrow{f(a),C} p'[f]} \qquad \frac{p \xrightarrow{a,C} p' \quad q \xrightarrow{a,C'} q'}{p||_A q \xrightarrow{a,C\cup C'} p'||_A q'}\ a \in A$$

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

Thus, $p||_A q$ executes p and q in parallel, and they are synchronised by actions in A. We remark that synchronisation may happen if the synchronising actions are enabled in both processes. Finally, the *renaming* operation $p[f]$ is a process that behaves like p except that actions are renamed by f. We assume the following precedence among the operations, $+ < ||_A < \{\!|C|\!\} = C\mapsto = a; < [f]$.

In Section 5 we will use the following shorthand notation. We define the *stochastic prefixing* by $a(x_G); P \overset{\text{def}}{=} \{\!|x_G|\!\}\{x_G\}\mapsto a; P$. In fact, this is the prefixing most widely adopted in stochastic process algebras.

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 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, in our case, in ♤ extended with $\overline{\mathsf{ck}}$. The clock-setting function κ and the set of edges $\longrightarrow$ are defined as the smallest relation satisfying the rules in Table 1. The function F is defined by $F(x_G) = G$ for each clock $x_G \in \mathcal{C}$. Other components are defined as for the syntax of ♤.

It turns out that stochastic automata and the language ♤ are equally expressive [6]. This means that for any (finitely branching) stochastic automaton a corresponding (guarded recursive) term in the language can be given whose reachable part of its stochastic automaton is identical to the stochastic automaton at hand. A recursive specification E is *guarded* if $X = p \in E$ implies that all variables in p appear in a context of a prefix. A stochastic automaton is finitely branching if for every location the set of outgoing edges is finite.

[2] Here we assume that p does not contain any name clashes of clock variables. This is not a severe restriction since terms that suffer from such name clash can always be properly renamed into a term without such name clash [6].

4 Discrete event simulation of ♠ processes

Given a ♠ process, the simulation algorithm returns a possible execution of it. The execution is calculated according to the probabilities and the timing of the different clocks and transitions plus an additional machinery to resolve the inherent non-determinism of the model.

In the next subsection we define the theoretical foundations of the algorithm and in the second part we give the simulation algorithm.

Foundations. Basically, an execution of a PTS is a path obtained by traversing it starting from the initial state.

Definition 5. Let $\mathcal{T} = (\Sigma, \Omega, \sigma_0, \mathcal{L}, T, \longrightarrow)$ be a PTS. An *execution* of $\mathcal{T}$ is a (finite or infinite) sequence $\sigma_0\sigma'_0\ell_1\sigma_1\sigma'_1\ell_2\sigma_2\sigma'_2\ldots$ such that, for all $i \geq 0$,

1. σ'_i is in the *support* of the distribution function F induced by $T(\sigma_i)$. That is, $\frac{\partial}{\partial\omega}F(\sigma'_i) > 0$ where $\frac{\partial}{\partial\omega}F$ can be interpreted as the density function corresponding to the distribution function F[3];
2. $\sigma'_i \xrightarrow{\ell_{i+1}} \sigma_{i+1}$;
3. if the sequence is finite, it ends in a non-deterministic state (i.e., some σ'_i).

We denote by *exec*$(\mathcal{T})$ the set of all executions of $\mathcal{T}$. If ρ is a finite execution, *last*(ρ) denotes the last (non-deterministic) state in the execution ρ. $\square$

The first restriction states that probabilistic steps should be probable ones. In this way we only consider as execution those probable paths. In a more general setting it may be interesting to consider executions with probability 0 as well. In our case, we want the simulator to generate only probable executions. The second restriction defines the non-deterministic steps in the execution.

Theoretically speaking, the notion of non-determinism is important because it tells when the choice of executing one or another activity should remain underspecified. In fact, this choice is not system dependent but architecture dependent, that is, it depends on the place where the system is running. The performance of a system does also depend on the architecture where it is executed. In other words, a PTS $\mathcal{T}$ should be understood as the semantics of the *specification* of the system. To study the performance of a system we need to consider a particular *implementation*, and hence non-determinism should be resolved in some way. To do so, an additional machinery is put on top of the system. This machinery is known, in general, as *schedulers* and in particular as *adversaries* in the context of probabilistic transition systems [21, 19]. Thus, a particular implementation of a system $\mathcal{T}$ is the pair $(\mathcal{T}, \mathcal{A})$ where $\mathcal{A}$ is a given adversary. The simulation algorithm that we are going to present allows to study the performance of a particular implementation of a given specification.

[3] Because we allow arbitrary distributions, the definition of the differential operator $\frac{\partial}{\partial\omega}$ is quite involved and we prefer to omit it here. We refer to [17] for details.

An adversary is a function that schedules the next non-deterministic transition based on the past history of the system. We consider probabilistic adversaries like in [19], although our definition changes to adapt to PTSs more in the style of [10].

Definition 6. Let $\mathcal{T} = (\Sigma, \Omega, \sigma_0, \mathcal{L}, T, \longrightarrow)$ be a PTS. An *adversary* or *scheduler* is a partial function $\mathcal{A} : \mathit{exec}(\mathcal{T}) \to ((\longrightarrow) \to [0,1])$ such that for all finite executions $\rho \in \mathit{exec}(\mathcal{T})$, whenever the (countable) sample space $\rho^{\rightarrow}$ satisfies

$$\emptyset \neq \rho^{\rightarrow} \subseteq \left\{ \sigma' \xrightarrow{\ell} \sigma \mid \mathit{last}(\rho) = \sigma' \right\},$$

then $\mathcal{A}(\rho) \overset{\text{def}}{=} P$ for some discrete probability measure P on the (discrete) σ-algebra $\wp(\rho^{\rightarrow})$. □

An execution of a stochastic automaton is an execution of its underlying semantics $I_{v_0}(SA)$. Notice that, given a state $[s, v]$ in $I_{v_0}(SA)$, the transition is fully determined for this state and the outgoing edges from s. Hence, we can adapt the notion of adversaries such that their image ranges over the edges of a stochastic automaton instead of the transitions of the associated PTS.

Definition 7. Let $SA = (\mathcal{S}, s_0, \mathcal{C}, \mathbf{A}, \relbar\!\!\blacktriangleright, \kappa, F)$ be a stochastic automaton. An *adversary* or *scheduler* for SA is a partial function $\mathcal{A} : \mathit{exec}(I_{v_0}(SA)) \to ((\relbar\!\!\blacktriangleright) \to [0,1])$ such that for any finite execution $\rho \in \mathit{exec}(I_{v_0}(SA))$, whenever the (countable) sample space $\rho^{\blacktriangleright}$ satisfies

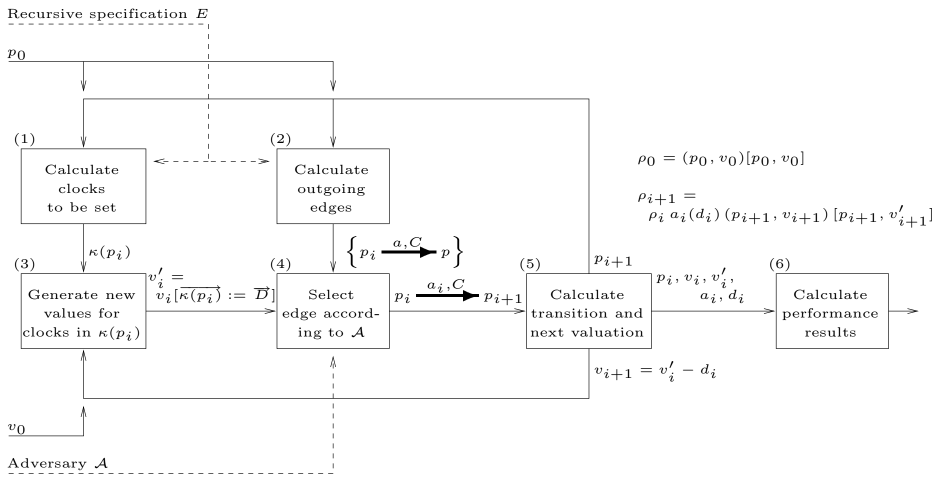
$$\emptyset \neq \rho^{\blacktriangleright} \subseteq \left\{ s \xrightarrow{a,C} s' \mid \mathit{last}(\rho) = [s, v] \wedge \mathit{enabled}(s \xrightarrow{a,C} s', v) \right\}$$

then $\mathcal{A}(\rho) \overset{\text{def}}{=} P$ for some discrete probability measure P on the (discrete) σ-algebra $\wp(\rho^{\blacktriangleright})$. □

The simulation algorithm. The simulation algorithm is a *variable time-advance procedure* [20] in which simulated time goes forward to the next time at which a transition is triggered and the intervening time is skipped. The structure of the simulation algorithm is depicted in Figure 1. The inputs of the algorithm are:

- a recursive specification E,
- an adversary $\mathcal{A}$,
- an initial process or *root* p_0, and
- an initial valuation v_0.

The root and the initial valuation are only relevant at the initial point of the algorithm since they form the initial state (p_0, v_0) of the PTS $I_{v_0}(SA(p_0))$, and hence the first state of any execution. The recursive specification E is relevant for the modules (1) and (2) which take care of generating the stochastic automaton. In fact, the stochastic automaton is not generated completely but only the required parts in order to choose the step to be simulated. The algorithm only saves the current location (or better, the current $\spadesuit$ term) p_i which is the only



Recursive specification E
p_0
(1) Calculate clocks to be set
(2) Calculate outgoing edges
$\kappa(p_i)$
$\left\{ p_i \xrightarrow{a,C} p \right\}$
(3) Generate new values for clocks in $\kappa(p_i)$
$v'_i = v_i[\overrightarrow{\kappa(p_i)} := \overrightarrow{D}]$
(4) Select edge according to $\mathcal{A}$
$p_i \xrightarrow{a_i,C} p_{i+1}$
(5) Calculate transition and next valuation
p_{i+1}
p_i, v_i, v'_i, a_i, d_i
(6) Calculate performance results
$v_{i+1} = v'_i - d_i$
v_0
Adversary $\mathcal{A}$
$\rho_0 = (p_0, v_0)[p_0, v_0]$
$\rho_{i+1} = \rho_i \; a_i(d_i)\,(p_{i+1}, v_{i+1})\,[p_{i+1}, v'_{i+1}]$

Fig. 1. Schema of the simulation algorithm

necessary information to recover the set of clock $\kappa(p_i)$ to be set (done by module (1)) and the possible outgoing edges (done by module (2)) according to the rules given in Table 1.

After calculating the structure of the next step, the next valuation should be calculated. At this point it is only necessary to save the last valuation v_i. Module (3) assigns to each clock x_G in $\kappa(p_i)$ a random value according to the distribution function G, while the rest of the clocks keep their old value. In Figure 1 this new valuation is denoted by $v'_i = v_i[\overrightarrow{\kappa(p_i)} := \overrightarrow{D}]$. $\overrightarrow{D}$ represents the randomly selected values. In fact, module (3) corresponds to the rule **Prob** given in Definition 3.

The adversary $\mathcal{A}$ is used in module (4) in order to select the next edge to be executed. Module (4) takes all possible edges calculated in (2) and selects only those that are enabled (see Definition 3). In this way, the sample space $\rho_i^{\blacktriangleright}$ can be obtained, and the adversary $\mathcal{A}$ will take care of selecting only one possible edge to be "executed" according to Definition 7.

Module (5) calculates the next transition according to rule **Act** in Definition 3. Thus, it gives not only the executed action a_i and its timing d_i but also the next location p_{i+1} and the next valuation $v_{i+1} = v'_i - d_i$ which are used to determine the next step in the execution.

Module (6) is user dependent. It gathers the information from the execution determined by the simulation in order to calculate the statistics that the user may require. For examples of what can be done, we refer to the next section.

We have implemented a prototype of the simulation algorithm using the functional language Haskell 1.4 [12]. We have assumed the initial valuation to be always the constant function $v_0(x) = 0$ for every clock x. This is not an actual restriction since a specification typically does not have free clock variables in its root process p_0. Another reasonable restriction is that unguarded recursive specifications are not allowed. Unguarded recursion could yield an infinite set of

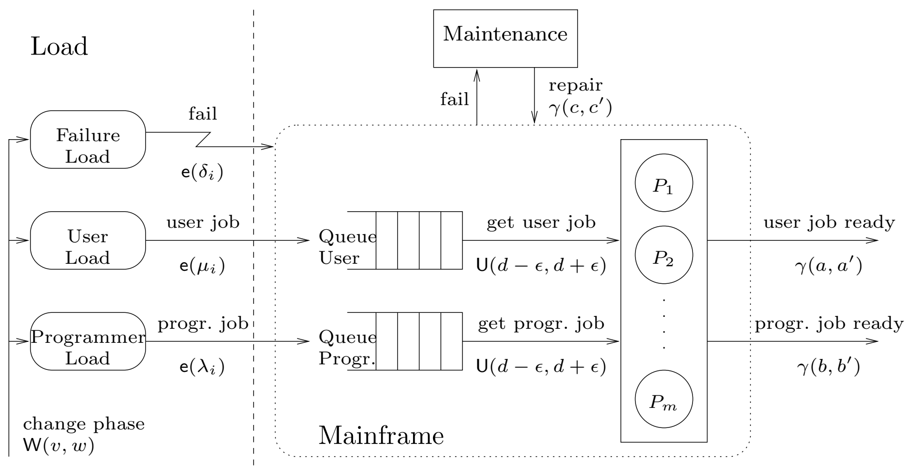


Fig. 2. Architecture of the mainframe system

outgoing edges or an infinite set of clock settings. In this way, modules (4) and (5) would never finish their computations.

Another restriction of our implementation is that it only deals with adversaries that are history independent, that is, our adversaries must satisfy

$$\mathit{last}(\rho) = \mathit{last}(\rho') \implies \mathcal{A}(\rho) = \mathcal{A}(\rho').$$

This is due to the fact that it is not a nice idea to save a complete execution which is intended to grow to the infinite. Using history independent adversaries reduces dramatically the complexity of the algorithm. We discuss improvements in the last section of the paper.

Notice that modules (3) and (4) require the use of "randomness". To that purpose, we use a random number generator which is a multiplicative linear congruential generator with modulus $m = 2^{31} - 1$ and multiplier $a = 16807$ calculated by Schrage's algorithm (see [15] for more information).

5 Example: A multiprocessor mainframe

In this section we discuss an example that we adopted from [13]. Figure 2 gives the architecture of a multiprocessor mainframe that serves two purposes. On the one hand, the system has to maintain a database and therefore has to process transactions submitted by different users. On the other hand, it is used for program development and thus it has to serve programmer's requirements such as compilation and testing. Besides, the system can be altered by the occurrence of software failures.

Using ♤ we can easily write a modular description of the system. In the next subsection we give the model description in ♤. Subsequently, we discuss the type of adversary we use and give our simulation results.

Formal specification using ♠. The system has three parts: the mainframe itself, the maintenance module that takes care of repairing failures, and the system load which basically is the environment representing the user and programmer job arrivals and the occurrence of failures.

$$\mathit{System} = \mathit{Load} \mid\mid_L (\mathit{Mainframe} \mid\mid_F \mathit{Maintain})$$

Process *Load* synchronises with the *Mainframe* each time a user or a programmer sends a job requirement or when a failure occurs. Process *Maintain* only synchronises when a failure occurs and when it is repaired. So

$$L = \{\mathit{usrJob}, \mathit{prgJob}, \mathit{fail}\} \qquad \text{and} \qquad F = \{\mathit{fail}, \mathit{repair}\}$$

Process *Load* models the user and programmer load, and the failure occurrences

$$\mathit{Load} = \mathit{PrgLoad}_1 \mid\mid_{\{\mathit{chng}\}} \mathit{UsrLoad}_1 \mid\mid_{\{\mathit{chng}\}} \mathit{FailLoad}_1 \mid\mid_{\{\mathit{chng}\}} \mathit{ChngPhase}$$

Process *ChngPhase* is intended to model the variation of the load along the time (e.g. the load during the night is different from that on peak hours.) Phases change according to a Weibull distribution function with parameters (v, w) (denoted by $\mathsf{W}(v, w)$) [4].

$$\mathit{ChngPhase} = \mathit{chng}(x_{\mathsf{W}(v,w)}); \mathit{ChngPhase}$$

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. 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. Processes *UsrLoad*, *PrgLoad*, and *FailLoad* are as follows.

$$\begin{aligned}
\mathit{UsrLoad}_1 &= \mathit{nextUsrJob}(xu_{\mathsf{e}(\mu_1)}); (\mathit{usrJob}; \mathit{UsrLoad}_1 + \mathit{reject}; \mathit{UsrLoad}_1) \\
&\quad + \mathit{chng}; \mathit{UsrLoad}_2 \\
\mathit{UsrLoad}_2 &= \mathit{nextUsrJob}(xu_{\mathsf{e}(\mu_2)}); (\mathit{usrJob}; \mathit{UsrLoad}_2 + \mathit{reject}; \mathit{UsrLoad}_2) \\
&\quad + \mathit{chng}; \mathit{UsrLoad}_3 \\
\mathit{UsrLoad}_3 &= \mathit{chng}; \mathit{UsrLoad}_1
\end{aligned}$$

$$\begin{aligned}
\mathit{PrgLoad}_1 &= \mathit{nextPrgJob}(xp_{\mathsf{e}(\lambda_1)}); (\mathit{prgJob}; \mathit{PrgLoad}_1 + \mathit{reject}; \mathit{PrgLoad}_1) \\
&\quad + \mathit{chng}; \mathit{PrgLoad}_2 \\
\mathit{PrgLoad}_2 &= \mathit{nextPrgJob}(xp_{\mathsf{e}(\lambda_2)}); (\mathit{prgJob}; \mathit{PrgLoad}_2 + \mathit{reject}; \mathit{PrgLoad}_2) \\
&\quad + \mathit{chng}; \mathit{PrgLoad}_3 \\
\mathit{PrgLoad}_3 &= \mathit{chng}; \mathit{PrgLoad}_1
\end{aligned}$$

[4] Distribution functions along the example were chosen arbitrarily with the intention of showing the versatility of ♠ and the simulator.

$$\begin{aligned}
\mathit{FailLoad}_1 &= \mathit{nextFail}(\mathit{xf}_{\mathsf{e}(\delta_1)}); (\mathit{fail}; \mathit{FailLoad}_1 + \mathit{reject}; \mathit{FailLoad}_1) \\
&\quad + \mathit{chng}; \mathit{FailLoad}_2 \\
\mathit{FailLoad}_2 &= \mathit{nextFail}(\mathit{xf}_{\mathsf{e}(\delta_2)}); (\mathit{fail}; \mathit{FailLoad}_2 + \mathit{reject}; \mathit{FailLoad}_2) \\
&\quad + \mathit{chng}; \mathit{FailLoad}_3 \\
\mathit{FailLoad}_3 &= \mathit{chng}; \mathit{FailLoad}_1
\end{aligned}$$

The *Mainframe* consists of *Queues* and processors P_i. The different processes are fully synchronised with the actions *fail* and *repair*: when a failure occurs the complete system must stop until it is repaired. Besides, 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. Since we consider that transferring a job from a queue to a processor has a certain duration, we split the action into a begin part and an end part.

$$\mathit{Mainframe} = \mathit{Queues} \,||_{G \cup F}\, (P_1 \,||_F\, P_2 \,||_F\, \cdots \,||_F\, P_m)$$

$$G = \{\mathit{getUsrJobBegin}, \mathit{getPrgJobBegin}\}$$

There are two FIFO queues, one for user jobs and one for programmer jobs. They are symmetric, only varying in their length. So, we use a unique process scheme for the queue and we take advantage of the renaming operation.

$$\mathit{Queues} = \mathit{QUsr} \,||_F\, \mathit{QPrg}$$

$$\mathit{QUsr} = Q_0^{nu}[f_u] \qquad\qquad \mathit{QPrg} = Q_0^{np}[f_p]$$

$$\begin{aligned}
&f_u(\mathit{job}) = \mathit{usrJob} &\qquad& f_p(\mathit{job}) = \mathit{prgJob} \\
&f_u(\mathit{getBegin}) = \mathit{getUsrJobBegin} && f_p(\mathit{getBegin}) = \mathit{getPrgJobBegin} \\
&\text{otherwise: } f_u(a) = a && \text{otherwise: } f_p(a) = a
\end{aligned}$$

Each queue is modelled by a set of processes Q_i^n where n $(n > 0)$ is the length of the queue and i the number of queued jobs. A queue is blocking if there is a failure or if it is full. In such case, jobs are rejected.

$$\begin{aligned}
Q_0^n &= \mathit{job}; Q_1^n + \mathit{fail}; \mathit{repair}; Q_0^n \\
Q_i^n &= \mathit{job}; Q_{i+1}^n + \mathit{getBegin}; Q_{i-1}^n + \mathit{fail}; \mathit{repair}; Q_i^n \qquad 0 < i \leq n-1 \\
Q_n^n &= \mathit{getBegin}; Q_{n-1}^n + \mathit{fail}; \mathit{repair}; Q_n^n
\end{aligned}$$

The processors are symmetric. Nevertheless, we should be careful in order to avoid clock name clashes [6]. We use indices to distinguish clock names. A processor gets either a user or a programmer job from the respective queue. In any case, this process takes d units of time with an error of $\pm\epsilon$ distributed uniformly (i.e. according to $\mathsf{U}(d-\epsilon, d+\epsilon)$). Notice that actions *getUsrJobEnd* and *getPrgJobEnd* are local to the processor. We do so because we consider that this activity takes part of the service time of the processor, and not of the queue. After loading a job, the processor executes it. The execution time of a user job is distributed according to a gamma distribution with parameters (a, a') (notation $\gamma(a, a')$). The execution of a programmer job is distributed according to $\gamma(b, b')$. A failure induces the processor to abort any activity. When the system is repaired, each processor restarts from its initial state.

$$\begin{aligned}
&P_i = \mathit{getUsrJobBegin}; (\mathit{getUsrJobEnd}(y^i_{\mathsf{U}(d-\epsilon,d+\epsilon)}); \mathit{PWU}_i + \mathit{PF}_i) \\
&\qquad + \mathit{getPrgJobBegin}; (\mathit{getPrgJobEnd}(y^i_{\mathsf{U}(d-\epsilon,d+\epsilon)}); \mathit{PWP}_i + \mathit{PF}_i) \\
&\qquad + \mathit{PF}_i \\
&\mathit{PWU}_i = \mathit{usrJobReady}(v^i_{\gamma(a,a')}); P_i + \mathit{PF}_i \\
&\mathit{PWP}_i = \mathit{prgJobReady}(w^i_{\gamma(b,b')}); P_i + \mathit{PF}_i \\
&\mathit{PF}_i = \mathit{fail}; \mathit{repair}; P_i
\end{aligned}$$

Process *Maintain* simply repairs a failure each time it occurs. The reparation time is distributed according to $\gamma(c, c')$.

$$\mathit{Maintain} = \mathit{fail}; \mathit{repair}(z_{\gamma(c,c')}); \mathit{Maintain}.$$

Simulation details. In the previous subsection we presented the model of the mainframe. We notice some choices we would like to add to the specification.

(a) In the *UsrLoad* component a situation of non-determinism may arise between actions *reject* and *usrJob*. We would like 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 processes *PrgLoad* and *FailLoad*. In fact, we want that *reject* has the lowest priority amongst all actions.

(b) A failure is an arbitrary event that at any moment may disturb the normal execution of the system. For that reason, we would not like to make a failure wait if it is enabled. So, we consider that action *fail* has the highest priority.

(c) User jobs are usually short activities that have to be processed as soon as possible, such as saving a file or processing a small database transaction. Programmer jobs are more complicated tasks that may involve compilation, simulation or testing of a system. From this observation, we would like that user jobs have higher priority than programmer jobs.

Summarising, we consider the priority relation $\prec$ defined as the least (strict partial) order satisfying the following conditions

$$\begin{aligned}
\mathit{reject} \prec a &\iff a \neq \mathit{reject} \\
a \prec \mathit{fail} &\iff a \neq \mathit{fail} \\
\mathit{getPrgJobBegin} &\prec \mathit{getUsrJobBegin}
\end{aligned}$$

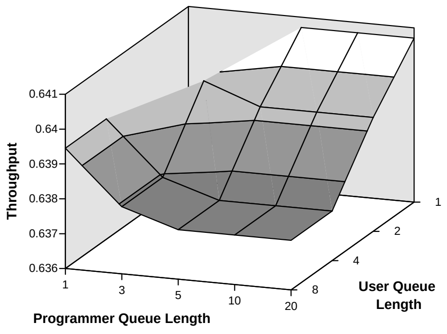
In fact these criteria are used to define the adversary that we use to simulate the multiprocessor mainframe. If after reducing the possible activities to be executed according to the defined priorities there is some nondeterminism remaining, we let the adversary to resolve it according to a (discrete) uniform probability distribution. Formally we define the adversary as follows.

The sample space $\rho\!\blacktriangleright$ is defined as the maximally possible, i.e.

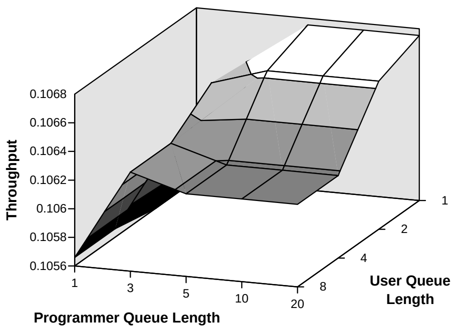
$$\rho\!\blacktriangleright \overset{\text{def}}{=} \left\{ s \xrightarrow{a,C} s' \mid \mathit{last}(\rho) = [s, v] \wedge \mathit{enabled}(s \xrightarrow{a,C} s', v) \right\}$$

Table 2. Parameters for the studied mainframe

Architecture	Load		Processing	
$m = 4$	$\mu_1 = 0.033$	$\mu_2 = 2$	$d = 0.021$	$e = 0.001$
$nu = 4$	$\lambda_1 = 0.01667$	$\lambda_2 = 0.16$	$a = 0.16667$	$a' = 0.5$
$np = 10$	$v = 300$	$w = 6$	$b = 0.16667$	$b' = 2.0$



(a) Throughput User Jobs



(b) Throughput Programmer Jobs

Fig. 3. Studying the length of the queues

We use the auxiliary operation *pri* to prune those enabled edges with lower probability than other enabled edges, that is, $pri(\rho)$ returns the maximal elements in $\rho^{\blacktriangleright}$ according to the $\prec$ order.

$$pri(\rho) \overset{\text{def}}{=} \left\{ s \xrightarrow{a,C} s' \in \rho^{\blacktriangleright} | \neg \exists s \xrightarrow{b,C'} t \in \rho^{\blacktriangleright} \wedge a \prec b \right\}$$

The adversary is then simply defined as follows

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

Simulation results. We set the values of the different parameters according to Table 2. As in [13], 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 throughputs stabilise when the user and programmer queue length are at least 4 and 5, respectively, that is, queues larger than those values do not affect the throughput (notice that the planes in the pictures become horizontal from that point on). We finally fixed the values to 4 and 10 respectively (see Table 2).

We ran different simulations changing the parameters related to failure and repairing. In all the cases we took $\delta_2 = \delta_1/2$. For the reparation time we set $c = 1$ and hence the average coincides with parameter 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 in which the mainframe is processing jobs. Figure 5 depicts the throughput of user jobs, i.e. the number of jobs that

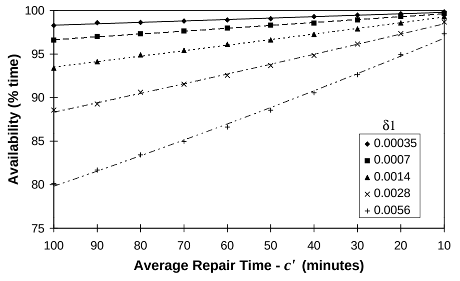


Fig. 4. Availability

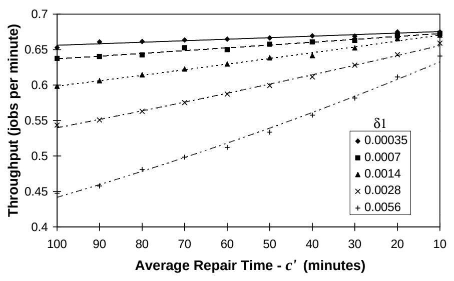


Fig. 5. Throughput User Jobs

are successfully processed per time unit. In both cases the lines are exponential functions that approximate the obtained simulation values. The functions have the shape $f(x) = re^{sx}$, where r and s are appropriate constants.

To calculate the user job throughput, we simply count the number of occurrences of action *usrJobReady* per time unit. To determine the availability we count the occurrences of the action *fail* per time unit instead, say *fpm*, and then calculate $100 \cdot (1 - fpm \cdot c')$. Since c' is the average of repair time, then $fpm \cdot c'$ is the fraction that the system is unavailable per time unit. The simulation algorithm also allows for the direct calculation of the availability, but in our case the method we followed is more precise.

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 [15]. 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%.

We mention as well that, since the system is finite, we are able to automatically check that it is deadlock free and does not have clock name clashes. The first checking can be done by simple inspection of the underlying stochastic automaton (which basically reduces to applying the expansion law) and the second one, by using the appropriate rules. The theory for these algorithms is described in [6]. We plan to include this can of checking in the near future.

6 Further discussions

Related work. A first approach to formally validate simulation models was to investigate the mapping of a high-level language for describing discrete-event simulation models into (non-stochastic) process algebras. Pooley [18] studied the translation of the process-based simulation language DEMOS (Discrete-Event Modelling On Simula) into TCCS, and Tofts and Birtwistle [3] use process algebras, basically CCS and its synchronous variant SCCS, to provide a denotational semantics of DEMOS. These works do not formalise the probabilistic information present in the original language and the role of process algebra with respect to simulation is different from ours.

Katoen et. al. [16] used generalised semi-Markov processes (GSMPs) indirectly as a semantic model: they map a non-Markovian stochastic process algebra onto event structures, and obtain for a subclass of event structures (the so-called stochastic deterministic ones) a GSMP. This translation induces a straightforward simulation algorithm. We can say that in our approach the intermediate model (the event structures) is omitted, and moreover we accept non-determinism as an inherent characteristic of our model, which is resolved when simulating. We should mention that there is no actual implementation of the algorithm presented in [16].

Field et. al. [7] use the stochastic process algebra of Harrison & Strulo [11] to model and simulate a cache coherency protocol. In order to simulate the protocol they compile the specification into C++ code and make use of some simulation libraries. Although their approach is somehow similar to ours, we use a different stochastic process algebra and we use adversaries to resolve possible non-determinism.

Conclusions and Further Work. In this work we gave an overview of the foundations and showed how to simulate specifications written in ♤. The advantages of using ♤ as modelling language are many-fold. First, the fact that ♤ is a process algebra allows for formal verification of the functional correctness of the model to be studied. In our example we are able to (automatically) check that the system does not have any deadlock. A second advantage is that, in contrast to many simulation packages, ♤ allows, in principle, for the description of any process without restricting to a particular method, which makes our simulation and modelling a general-purpose task. Finally, the compositional nature of ♤ together with the inherent ideas of simulation allows for on-the-fly construction of the simulation model. This implies that simulation of systems with recursive specifications containing infinite equations (such as $P_i = a_i; P_{i+1}$) is possible.

We have used our simulation algorithm to analyse different kinds of (single) queues systems specified in ♤, including infinite queues. We checked the obtained results against the analytical solutions and the results have been satisfactory. Besides, we have "self-checked" the results obtained in the mainframe example by obtaining quite tight confidence intervals.

In its current state, we are using the program as a library to be compiled together with the ♤ specification and the module to calculate the performance results (number (6) in Figure 1). The adversary can be either selected from a given library or explicitly defined. The ♤ specification is given in terms of a Haskell data type, which closely resembles the ♤ notation.

One of our plans is to incorporate the adversaries as part of the syntax. The idea is to encode the probabilities of taking a transition from one or the other operand into the parallel composition and summation. We will base our ideas on results presented in [4]. This encoding might also be dynamic, in the sense that a transition may induce different values for the probabilities in the next step.

As we mentioned, the simulator is still in a prototype version. To turn it into a serious tool, much work has to be done. First, we know that our algorithm is optimal in state space requirements since only the current symbolic state and the valuations of the active clocks are saved. However time complexity has to be addressed by searching for optimal ways of calculating κ, $\longrightarrow$, and the selection of the transition to be triggered. In second place, we need to reduce the "extra work" the user has to do. This has two implications. On the one hand, we should try to automatise as much as possible the analysis performed by module (6) and to eliminate the Haskell overhead in the given ♤ specification. Thus, the simulation would become a program by itself that would take two scripts as inputs: one containing the ♤ specification (including the adversaries encoded in the term), and the other, the instructions for the desired analysis. On the other hand, we should provide libraries of commonly used process and macros, such as queues or "work loaders" (see Section 5).

Finally, we should mention our intention of extending ♤ with data. This would definitely facilitate the task of modelling and hence broadening the scope of ♤ to specify and analyse, for instance, mobile systems and protocols which include significant data transfer.

References

1. R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.
2. M. Bernardo and R. Gorrieri. A tutorial on EMPA: A theory of concurrent processes with nondeterminism, priorities, probabilities and time. *Theoretical Computer Science*, 202:1–54, 1998.
3. G.M. Birtwistle and C. Tofts. Process semantics for simulation. Technical report, University of Swansea, 1996.
4. P.R. D'Argenio, H. Hermanns, and J.-P. Katoen. On geneartive parallel composition. In *Proc. of PROBMIV'98 (Preliminary),* Indianapolis, USA, Tech. Rep. CSR-98-4, pp. 105–121. School of Comp. Sci., University of Birmingham, 1998.
5. P.R. D'Argenio, J.-P. Katoen, and E. Brinksma. A stochastic automata model and its algebraic approach. In E. Brinksma and A. Nymeyer, eds., *Proc. of PAPM'97,* Enschede, The Netherlands, Tech. Rep. CTIT 97-14, pp. 1–16. University of Twente, 1997.
6. P.R. D'Argenio, J.-P. Katoen, and E. Brinksma. An algebraic approach to the specification of stochastic systems (extended abstract). In D. Gries and W.-P. de Roever, editors, *Proc. of PROCOMET'98,* Shelter Island, USA, IFIP Series, pp. 126–147. Chapman & Hall, 1998.
7. A.J. Field, P.G. Harrison, and K. Kanani. Automatic generation of verifiable cache coherence simulation models from high-level specifications. In *Proc. of CATS'98. Australian Comp. Sci. Comm.*, 20(3):261–275, 1998.
8. P.W. Glynn. A GSMP formalism for discrete event simulation. *Proceedings of the IEEE*, 77(1):14–23, 1989.
9. N. Götz, U. Herzog, and M.Rettelbach. TIPP - Introduction and application to protocol performance analysis. In H. König, ed., *Formale Beschreibungstechniken für verteilte Systeme*, FOKUS series. Saur Publishers, 1993.

10. H.A. Hansson. *Time and Probability in Formal Design of Distributed Systems*, volume 1 of *Real–Time Safety Critical Systems*. Elsevier, 1994.
11. P. Harrison and B. Strulo. Stochastic process algebra for discrete event simulation. In F. Bacelli, A. Jean-Marie, and I. Mitrani, eds., *Quantitative Methods in Parallel Systems*, Esprit Basic Research Series, pp. 18–37. Springer-Verlag, 1995.
12. Report on the programming language Haskell: A non-strict, purely functional language (Version 1.4), April 1997. URL: `http://haskell.org/`.
13. U. Herzog and V. Mertsiotakis. Stochastic process algebras applied to failure modelling. In *Proc. of PAPM'94*. University of Erlangen, 1994.
14. J. Hillston. *A Compositional Approach to Performance Modelling*. Distinguished Dissertation in Computer Science. Cambridge University Press, 1996.
15. R. Jain. *The Art of Computer Systems Performance Analysis*. John Wiley & Sons, 1991.
16. J.-P. Katoen, E. Brinksma, D. Latella, and R. Langerak. Stochastic simulation of event structures. In M. Ribaudo, ed., *Proc. of PAPM'96,* Torino, Italy, pp. 21–40. Università di Torino, 1996.
17. S. Lang. *Real and Functional Analysis*, volume 142 of *Graduate Texts in Mathematics*. Springer-Verlag, 3rd edition, 1993.
18. R.J. Pooley. Integrating behavioural and simulation modelling. In *Quantitative Evaluation of Computing and Communication Systems*, LNCS 977, pp. 102–116. Springer-Verlag, 1995.
19. R. Segala and N. Lynch. Probabilistic simulations for probabilistic processes. *Nordic Journal of Computing*, 2(2):250–273, 1995.
20. G.S. Shedler. *Regenerative Stochastic Simulation*. Academic Press, 1993.
21. M.Y. Vardi. Automatic verification of probabilistic concurrent finite state programs. In 26th *Annual Symposium on FOCS, Portland, Oregon*, pp. 327–338. IEEE Computer Society Press, 1985.