Under consideration for publication in Formal Aspects of Computing

An Integrated Framework for the Performance Analysis of Asynchronous Communicating Stochastic Processes [1]

Natalia López, Manuel Núñez, and Fernando Rubio

Universidad Complutense de Madrid

Abstract. In this paper we present a design framework containing a process algebra and a concurrent functional programming language. In order to study properties of a specification written in our process algebraic notation, we provide a mechanism to automatically translate that specification into an Eden program. The functional programming language Eden is very suitable for concurrent programming. On the one hand, it presents the usual features of modern functional languages. On the other hand, it allows the execution of concurrent processes. Once we have a translation, we may use the Paradise profiling tools developed for Eden to study the performance of the (simulation of the) system. In order to add expressiveness to our design language we use a very powerful process algebra. First, we allow the specification of delays induced by random variables. The probability distribution functions associated with these variables are not restricted to be exponential. We also consider value passing. Finally, in contrast with most process algebras, the communication between concurrent processes is asynchronous. That is, a process may send data through an output channel without waiting for a process receiving the message. Actually, it is very common to find systems where communications among components must be performed in this way. In order to show the usefulness of our framework we present two examples featuring all the characteristics of our process algebraic model (in particular, communications are inherently asynchronous), we give the corresponding translations, and we provide some performance measures obtained by using an extension of the Paradise tool.

1. Introduction

It is widely recognized that engineering design languages must be based on a (semi-)formal framework. Actually, in order to build reliable complex systems, it is very important to provide a formal design to contrast the capabilities provided by the system and the expected ones. Among the several existing design formalisms, *process algebras* (see [BPS01] for a good overview of the field) are very suitable to specify concurrent systems because they allow to model them in a compositional manner. First work on process

Correspondence and offprint requests to: Dept. Sistemas Informáticos y Programación. Facultad de Infomática. Universidad Complutense de Madrid, E-28040 Madrid. Spain. e-mail:`{natalia,mn,fernando}@sip.ucm.es`

[1] Research supported in part by the Ministerio de Ciencia y Tecnología projects MASTER and AMEVA, and by the Junta de Comunidades de Castilla-La Mancha project PAC-03-001.

algebras has settled an important theoretical background for the study of concurrent and distributed systems. They were very significant, mainly to shed light on concepts and to open research methodologies. However, due to the abstraction of the complicated features, models were still far from real systems. Therefore, some of the solutions were not specific enough, for instance, those related to real time systems. Thus, researchers in process algebras have tried to bridge the gap between formal models described by process algebras and real systems. In particular, features which were abstracted before have been introduced in the models so that they allow the design of systems where not only functional requirements but also performance ones are included. The most significant of these additions are related to notions of time, probabilities, and priorities (e.g. [CH90, NS91, Han91, AM94, GSS95, DS95, NdFL95, NdF95, CDSY99, JYL01, CLN01, BM02, Núñ03]). An attempt to integrate time and probabilistic information has been given by introducing *stochastic process algebras* (e.g. [GHR93, Hil96, BG98, Her98, D'A99, HS00]) where not only information about the period of time when the actions are enabled, but also about the probability distribution associated with these time intervals is included.

In order to study systems, *model checking* [CGP99, Sti01] represents a successful technique to check whether they fulfill a property. Nevertheless, current model checking techniques cannot be used to study systems where stochastic information is included. More exactly, model checking may deal with stochastic systems where probability distributions are restricted to be exponential, by using Markov chains techniques (see e.g. [HKMS00]). Unfortunately, this is not the case if probability distributions are *general* (a preliminary step has been given in [IHK01] where semi-Markov chains are considered). So, in order to analyze this kind of systems, a different approach should be used. Another approach, that we will use in this paper, consists in the *simulation* of the designed system. Intuitively, in order to study a specification, we can *implement* it and simulate its performance. In this case, we can get *real* estimations of the *theoretical* performance of the system.

Looking for an appropriate *simulation* language, we have found the functional paradigm to be very suitable for our purpose. First, due to the absence of side effects, functional programming allows to reason equationally about programs. In particular, it is possible to apply automatic transformations. Besides, it is easier to verify or even derive programs from their formal specifications. This is specially important for safety critical applications. Among the functional languages, Haskell [PH99] can be considered to be the current *standard.* Haskell represents the result of the joint effort of the (lazy evaluation) functional community to define a standard language to be used both for research and real applications. Several Haskell compilers exist, being GHC (Glasgow Haskell Compiler) the most efficient and widely used. It is easily portable to different architectures, a COM (Component Object Model [Rog97]) interface allows to interact with programs written in other languages, and its sources are freely available.

Unfortunately, Haskell is based on a sequential evaluation strategy. Therefore, concurrent execution of programs cannot be defined. In the last years there have been some extensions of Haskell where concurrency has been included (see, e.g. [PGF96]) but usually by means of very *low-level* concurrency primitives. On the contrary, the programming language Eden [BLOP98, KLPR01] is an adequate *high-level* concurrent extension of Haskell. Eden is the result of a joint research work between two groups in Germany and Spain with the support of the Scottish team who develops GHC. The first complete Eden compiler was finished recently, and heavy experimentation has been taking place since then (e.g. [PR01, Rub03]). Eden aims at reusing the advantages of Haskell for reasoning about programs, but applying them also to both concurrent and parallel systems. The main goal of the language is to provide concurrency and parallelism at a high level of abstraction, without losing efficiency. In addition to these good properties, Eden includes a set of profiling utilities (called Paradise [HPR00]) that facilitates the analysis of programs, so that it can also be used to study the real behavior at runtime.

In this paper we build on previous work [LNR02] to provide an integrated framework that allows us to study systems containing stochastic information by simulating their behaviors as Eden programs. In addition to the operators of a usual process algebra (choice, parallel, hiding, etc) we include some complex features to ease the design of real systems. We call this language `VPASPA`. First, our language allows a designer to specify timed information given by stochastic delays. For example, a process as $\xi \,;P$ is delayed an amount of time t with probability $F_\xi(t)$, where F_ξ is the probability distribution function associated with ξ. Let us remark that deterministic delays (as presented in timed process algebras) can be specified by using Dirac distributions. The combination of the parallel operator and general distributions strongly complicates the definition of semantic models. That is why stochastic process algebras are usually based on (semi)-Markov chains (some exceptions are [BBG98, DKB98, BG02]).

In our process algebra, we also consider value passing. Let us note that value passing is not usually included in stochastic process algebras (to the best of our knowledge, [Ber99] represents the only proposal for such a stochastic language). The last non-standard feature presented in our language is an asynchronous communication mechanism (see e.g. [dBKP92, BGZ98, ABL99]). We took this decision because asynchronous communication appears in many real life situations. However, we could also add a synchronous parallel operator to our language so that the designer can choose the desired communication paradigm. In order to model asynchronous communication, there are different approaches. For example, in [BGZ98] messages are considered as active entities. Thus, when an output action is performed, the process reached can send the message at any time the communication is possible. We model asynchrony by considering queues associated with the corresponding channels. In particular, this decision produces the main conceptual difference between our approach and [BGZ98], that is, in our theory the order in which the messages are received and transmitted is the same, because we use a queue to store them. Let us consider a process as $P = (a!(v)\,;P_1) \parallel P_2$. We allow the left hand side of the parallel composition to perform its output action $a!(v)$ without synchronizing with the right hand side. The associated value v is stored in a queue associated with the channel a and the process P evolves into $P_1 \parallel P_2$. Let us consider now a process as $Q = (a?(x)\,;Q_1) \parallel Q_2$. The input action $a?(x)$ can be performed only if the queue associated with a is not empty. In this case, the value v' stored at the head of the queue is extracted, the input action $a?$ is performed, and the process Q evolves into $Q_1[x/v'] \parallel Q_2$, that is, the free occurrences of the variable x in Q_1 are replaced by the value v'. If the queue is empty then the left hand side of the process Q is blocked until a value is stored in that queue. Finally, despite the inherent complexity of our language, we would like to remark that most of the semantic details can be *hidden* to possible users of the language. Actually, a designer using `VPASPA` needs to know the intuitive meaning of the operators but a thorough knowledge of the (maybe too complicated) semantic model can be skipped.

This combination of process algebras and functional languages allows us to profit from both the good properties of the former (as specification languages) and the ones of the latter (as implementation languages where formal reasoning can be performed easier than in other paradigms). In order to make a smooth transition from the specification to the implementation, we provide a mechanism to automatically *translate* from one formalism into the other. Let us note that most of the features of functional languages are not needed to implement our specifications. Therefore, it is only necessary to have some knowledge of the functional paradigm, mainly to *optimize* the implementations that the translation produces. We have also adapted the profiling utilities of Eden to our stochastic framework. That is, we will be able to study quantitative properties of specifications by studying the behavior of the corresponding Eden programs.

Regarding related work, let us remark that the translation of process algebras into another programming language has been already done several times (e.g. considering LOTOS we have [MdM88] among many others). In the stochastic process algebras area, [D'A99] also studies the simulation and implementation of a stochastic process algebra. Even though a functional language is used (actually, Haskell) our approach differs in several points from that one. Essentially, [D'A99] computes at each point of the execution the set of all the possible next states of the whole system. On the contrary, our approach is a real concurrent implementation, where processes really evolve independently. In fact, processes run in parallel. Thus, with our approach we do not only analyze the performance properties of the specifications, but we also obtain real implementations of the specifications. Thus, we can use it both as a performance evaluation tool and as a methodology for implementing stochastic process algebras. Besides, our generated code is a usual program, so that a user with some command in functional languages may optimize it, if necessary, by hand. Let us comment that, as far as we know, there do not exist tools dealing with generalized stochastic process. We may cite TwoTowers [BCSS98] for a good tool to study performance properties of systems. However, probability distributions are restricted to be exponential. Also in the Markovian setting, in [GH02] the language LOTOS has been modified to include stochastic information. By using a combination of several tools the authors are able to study performance evaluation without the necessity of simulating the execution.

The rest of the paper is organized as follows. In Section 2 we present our value passing stochastic process algebra. We also introduce a notion of bisimulation for our language. In Section 3 we present the basic features of Eden. We will mainly concentrate on the characteristics of the language that we will use along the paper. In particular, we will explain the way processes are defined in Eden. In Section 4 we give the algorithm underlying the translation from our process algebra into Eden. In Section 5 we introduce two complete examples showing the characteristics of our method. We present their specifications, the translations into Eden, and we study some quantitative properties. Finally, in Section 6 we present our conclusions.

2. An Asynchronous Stochastic Process Algebra with Value Passing

In this section we present our language and its operational semantics. The underlying semantic model is based on [LN01] where we have introduced some important modifications. We have slightly simplified the model by removing probabilities associated with the choice operator. On the contrary, we have added some new features that add expressiveness to the language, even though they complicate the semantic model. First, we consider value passing. Second, a parallel operator is now included. This last addition leads to an involved definition of the operational semantics. However, we find extremely useful a parallel operator to model the systems that we are interested in. Finally, communication is now asynchronously performed. Processes may perform either actions for transmitting a message (output actions) or for receiving them (input actions). In addition, processes may be delayed according to a random variable. We will suppose that the sample space (that is, the domain of random variables) is the set of real numbers $\mathbb{R}$ and that random variables take positive values only in $\mathbb{R}^+$, that is, given a random variable ξ we have $\mathrm{P}(\xi \leq t) = 0$ for any $t \leq 0$. The reason for this restriction is that random variables are always associated with time distributions.

Definition 2.1. Let ξ be a random variable. We define its *probability distribution function*, denoted by F_ξ, as the function $F_\xi : \mathbb{R} \longrightarrow (0,1]$ such that $F_\xi(x) = \mathrm{P}(\xi \leq x)$, where $\mathrm{P}(\xi \leq x)$ is the probability that ξ assumes values less than or equal to x.

Let ξ, ψ be random variables and $t_0 \in \mathbb{R}^+$. We say that ξ and ψ are *identically distributed up to* time t_0, denoted by $\xi \asymp_{t_0} \psi$, if for any $t \leq t_0$ we have $F_\xi(t) = F_\psi(t)$. □

The predicate $\asymp_t$ will be used to determine whether two random variables are identically distributed in the interval $[0,t]$. Regarding *communication* actions, they can be divided into *output* and *input actions.* Next, we define our alphabet of actions.

Definition 2.2. We consider a set of *communication* actions $\mathtt{Act} = \mathtt{Input} \cup \mathtt{Output}$, where we assume $\mathtt{Input} \cap \mathtt{Output} = \emptyset$. We suppose that there exists a bijection $f : \mathtt{Input} \longrightarrow \mathtt{Output}$. For any *input* action $a? \in \mathtt{Input}$, $f(a?)$ is denoted by the *output* action $a! \in \mathtt{Output}$. We consider a set of values *Val* representing the transmitted messages ($v, v', \ldots$ to range over *Val*) and a set of value variables $\mathcal{X}$ ($x, y, \ldots$ to range over $\mathcal{X}$). We define the set of *communications* as the set of input and output actions applied either to a value (output actions) or to a value variable (input actions), that is,

$$IO = \{c(x) \mid c \in \mathtt{Input}, x \in \mathcal{X}\} \cup \{c(m) \mid c \in \mathtt{Output}, m \in \mathcal{X} \cup \mathit{Val}\}$$

If there exists a message transmission between $a?$ and $a!$ we say that a is the *channel* of the communication. We denote by $\mathcal{C}_{\mathtt{Act}}$ the set of channels in $\mathtt{Act}$ ($a, b, \cdots$ to range over $\mathcal{C}_{\mathtt{Act}}$). Formally, we define the channel of a communication action $\omega = c(m) \in IO$, denoted by $ch(\omega)$, as a if $c \in \{a?, a!\}$; if $\omega \notin IO$ we will consider that $ch(\omega) = \perp \notin \mathcal{C}_{\mathtt{Act}}$. This last value indicates that this function is not defined for values $\omega \notin IO$. We consider that the function ch is extended to deal with sets in the usual way. □

We consider a denumerable set Id of process identifiers. In addition, we denote by $\mathcal{V}$ the set of random variables ($\xi, \xi', \psi, \cdots$ to range over $\mathcal{V}$). We also consider the *set of locations* $\mathrm{Loc} = \{loc_i | loc \in \{l, r\} \wedge i \in \mathbb{N}^+\}$ where l stands for left and r for right. As we pointed out in the introduction of this paper, we will use queues to store those messages that have been transmitted but they have not been yet received. In this case, we consider the usual operations for queues:

- $[\ ]$ denotes an empty queue.
- $enqueue(c, \sigma)$ denotes that the element c is added to the queue σ.
- If σ is not empty then $head(\sigma)$ returns the oldest element in σ; if the queue is empty then the undefined value $\perp$ is returned.
- If σ is not empty then $dequeue(\sigma)$ removes $head(\sigma)$ from σ; if the queue is empty then the undefined value $\perp$ is returned.

Next, we define the syntax of our process algebra.

Definition 2.3. The set of processes, denoted by $\mathtt{VPASPA}$, is given by the following EBNF:

$$P ::= \mathrm{stop} \mid \xi \,;\, P \mid a?(x) \,;\, P \mid a!(m) \,;\, P \mid P + P \mid \mathtt{if}\ e\ \mathtt{then}\ P\ \mathtt{else}\ P \mid P \parallel_M P \mid P/A \mid X := P$$

where $X \in Id$, $\xi \in \mathcal{V}$, $a?, a! \in \texttt{Act}$, $x \in \mathcal{X}$, $m \in \mathcal{X} \cup Val$, $A \subseteq \mathcal{C}_{\texttt{Act}}$, and $M \subseteq (\mathcal{V} \times \mathbb{N}^+ \times \text{Loc})$. We consider that the boolean expressions appearing as parameters of the `if` operator are of the form $m_1 = m_2$, where $m_1, m_2 \in \mathcal{X} \cup Val$. In the following, we also assume that $\texttt{Eval}(e) \in Bool$ represents the evaluation of the expression e.

The set of *communication processes*, denoted by $\texttt{VPASPA}^*$, is given by pairs of the form (P, μ), where $P \in \texttt{VPASPA}$ and μ is a set of queues over Val. □

We assume that all the random variables appearing in the definition of a process are independent. For the sake of clarity, if two delays are represented by the same random variable, we assume that these delays are associated with two independent random variables identically distributed. Next we give an intuitive explanation of each operator in our language.

- The term stop denotes a process that cannot execute any action.
- A process $\xi \,;\, P$ waits a random amount of time (determined by the probability distribution function associated with ξ) and then it behaves as P.
- The process $a?(x)\,;P$ waits until it receives a value v through the channel a. Then, the process behaves as $P[x/v]$, where $P[x/v]$ denotes the substitution of all the free occurrences of x in P by v (see forthcoming Definition 2.4). Let us note that x is bound in $a?(x) \,;\, P$.
- The process $a!(v) \,;\, P$ transmits the value v on the channel a and after that it behaves as P.
- The process $P + Q$ behaves either like P or like Q depending on which component performs the first action. Delays are an exception because they do not resolve choices. More precisely, the starting of a delay does not resolve the choice. We will explain this fact with more detail when we present our operational semantics.
- The process `if` e `then` P `else` Q behaves as P if the evaluation of the boolean expression e is `true` and as Q otherwise.
- The term $P \parallel_M Q$ can perform actions either from P or from Q. Output actions can be immediately performed while input actions must wait until a value is received through the channel, and in that moment they will be performed immediately. We explain below the meaning of the parameter M associated with the operator.
- The process P/A expresses that the actions belonging to A are hidden.
- Finally, $X := P$ denotes the definition of a (possible recursive) process.

Regarding communication processes, (P, μ) is a pair where P is a *usual* process and μ stores all the messages that have been created by P but have not been received yet. In this case, an independent queue will be used for each channel. Thus, each time a message is sent by an output action, it will be stored in the corresponding queue of μ, and it will remain there until an input action extracts it from the queue. Initially, we suppose that all the queues are empty, that is, μ contains an empty queue for each channel $a \in \mathcal{C}_{\texttt{Act}}$. We will write I to denote this initial set of queues. Along the paper we will sometimes consider the natural inclusion injection from $\texttt{VPASPA}$ into $\texttt{VPASPA}^*$ such that for any $P \in \texttt{VPASPA}$, its image in $\texttt{VPASPA}^*$ is (P, I).

Next, we formally present the definition of the substitution of all the free occurrences of a variable by a value.

Definition 2.4. Let P be a process, $x, y \in \mathcal{X}$, $v \in Val$, and $m \in \mathcal{X} \cup Val$. We define the substitution of all the free occurrences of x in P by v, denoted by $P[x/v]$, by structural induction as:

$$\text{stop}[x/v] = \text{stop} \qquad (\xi \,;\, P)[x/v] = \xi \,;\, P[x/v]$$

$$(P + Q)[x/v] = P[x/v] + Q[x/v] \qquad (P \parallel_M Q)[x/v] = P[x/v] \parallel_M Q[x/v]$$

$$(P/A)[x/v] = P[x/v]/A \qquad (X := P)[x/v] = (X := P[x/v])$$

$$(a?(y) \,;\, P)[x/v] = \begin{cases} a?(y) \,;\, P[x/v] & \text{if } x \neq y \\ a?(y) \,;\, P & \text{if } x = y \end{cases}$$

$$(a!(m) \,;\, P)[x/v] = \begin{cases} a!(m) \,;\, P[x/v] & \text{if } (m \in \mathcal{X} \,\wedge\, x \neq m) \,\vee\, (m \in Val) \\ a!(v) \,;\, P[x/v] & \text{if } m \in \mathcal{X} \,\wedge\, x = m \end{cases}$$

$$(\texttt{if } e \texttt{ then } P \texttt{ else } Q)[x/v] = \texttt{if } e[x/v] \texttt{ then } P[x/v] \texttt{ else } Q[x/v]$$

Let e be a boolean expression, $x \in \mathcal{X}$, and $v \in \mathit{Val}$. The substitution of all the free occurrences of x by v in the expression $e = (m_1 = m_2)$, denoted by $e[x/v]$, is defined as $(m_1[x/v] = m_2[x/v])$ where

$$m_i[x/v] = \begin{cases} v & \text{if } m_i \in \mathcal{X} \ \wedge \ m_i = x \\ m_i & \text{if } (m_i \in \mathcal{X} \ \wedge \ m_i \neq x) \ \vee \ m_i \in \mathit{Val} \end{cases}$$

□

As we will show in the definition of the operational semantics, stochastic transitions are performed in two steps: start and termination. For each random variable ξ, we will denote by ξ^+ the start of the delay and by ξ^- its termination. As introduced in [BG02], this is one of the mechanisms to deal with general distributions in the context of the parallel operator. In order to avoid *undesirable* effects with some recursive processes, we assign indexes to delays. For instance, in the process $X := (\xi \,; \mathrm{stop}) \,\|_M\, (a \,; X)$ there can be several instances of the start stochastic action ξ^+ running in parallel. In order to associate each termination action ξ^- with its corresponding start, we assign an index, that will be the minimum natural number not associated to ξ in the context of the process that performs the action. Besides, we take into account the *location* where ξ^+ was performed. This information and the information of the indexes are stored in M. Initially, we suppose $M = \emptyset$. We will explain the role of this set with more detail when describing the stochastic behavior of the processes. Let us remark that indexes would not be needed if we restrict occurrences of the parallel computation in the scope of the recursive definitions. In this case, the parameter M could be removed.

The operational semantics of the language is given in Figures 1 and 2. In order to simplify the presentation, we have isolated in Figure 2 the operational rules dealing only with stochastic behavior. Besides, we will use a two-level operational semantics. In the first part of Figures 1 and 2, the first-level operational semantics is shown. This level will describe the operational behavior of processes in `VPASPA`. Meanwhile, in the second part of both figures the upper level operational semantics is presented, that is, the operational semantics of communication processes. In these definitions we have extended the alphabet with an internal action τ. Even though τ is not implicitly included in the syntax of the language, internal actions can be generated by the hiding operator. We will also consider the set of starting actions $\mathcal{V}^+ = \{\xi_i^+ \mid \xi \in \mathcal{V} \ \wedge \ i \in \mathbb{N}^+\}$ and the set of termination actions $\mathcal{V}^- = \{\xi_i^- \mid \xi \in \mathcal{V} \ \wedge \ i \in \mathbb{N}^+\}$. We will use the following conventions: $P \xrightarrow{\ \omega\ } P'$ expresses that there exists a transition from P to P' labelled by the action ω; $P \xrightarrow{\ \omega\ }$ stands for there exists $P' \in$ `VPASPA` such that $P \xrightarrow{\ \omega\ } P'$; we write $P \not\xrightarrow{\ \omega\ }$ if there does not exist $P' \in$ `VPASPA` such that $P \xrightarrow{\ \omega\ } P'$. Besides, we will use the same conventions while describing the behavior of the communication processes.

Next, we briefly explain the rules appearing in Figure 1. Let us note that labels appearing in these operational transitions have the following types: $a \in \mathcal{C}_{\texttt{Act}}$, $\alpha \in \mathit{IO} \cup \{\tau\}$, and $\omega \in \mathit{IO} \cup \{\tau\} \cup \mathcal{V}^+ \cup \mathcal{V}^-$. The first four rules are quite standard. They simply indicate that if a process is prefixed by an action then the process has the capability to perform that action, besides both communication and internal actions resolve choices (as we will see, this is not true for stochastic actions). The following two rules describe the behavior of the parallel operator without considering the storage of the communication values, that will be shown in the last three rules of this figure while describing the upper level of the operational semantics. Thus, if a process of the composition can perform either a communication or an internal action, the composition can perform it. The next three rules are standard for recursion and hiding. Let us note that the rule for recursion applies to communication, internal, and stochastic actions. Let us also remark that $P[X/X := P]$ represents the substitution of all the free occurrences of X in P by $X := P$. The formal definition of this substitution is similar to that given in Definition 2.4 but taking into account that the variable X is bound in $X := P$. The following two rules express the usual meaning of a conditional operator. As in the recursion rule definition, ω ranges over the sets of communication and stochastic actions, and also over internal actions.

Finally, the last three rules present the operational behavior regarding action transitions for `VPASPA`* terms. As we said before, any process P in `VPASPA` is considered as (P, I) in `VPASPA`*, where I was defined as the set of queues such that for any $a \in \mathcal{C}_{\texttt{Act}}$, $\mu_a \in I$ and $\mu_a = [\ ]$. The first two rules define the behavior of the queues when communication actions are performed. If a process P may perform an output action $a!(v)$ then this action is performed by (P, μ), the transmitted value v is stored in the queue associated with a, that is μ_a, and the set of queues, μ, is modified accordingly (by using the function *put*). Regarding input actions, if P may perform an input action $a?(x)$ and the queue corresponding to a in μ is not empty, then this action is performed. In this case, the head of the corresponding queue is passed to the component that has performed $a?(x)$ and all the free occurrences of x are replaced by that value. Then, the set of queues is

$$\frac{}{a?(x);P \xrightarrow{a?(x)} P} \qquad \frac{}{a!(v);P \xrightarrow{a!(v)} P} \qquad \frac{P \xrightarrow{\alpha} P'}{P+Q \xrightarrow{\alpha} P'} \qquad \frac{Q \xrightarrow{\alpha} Q'}{P+Q \xrightarrow{\alpha} Q'}$$

$$\frac{P \xrightarrow{\alpha} P'}{P\|_M Q \xrightarrow{\alpha} P'\|_M Q} \qquad \frac{Q \xrightarrow{\alpha} Q'}{P\|_M Q \xrightarrow{\alpha} P\|_M Q'}$$

$$\frac{P[X/X{:=}P] \xrightarrow{\omega} P'}{X{:=}P \xrightarrow{\omega} P'} \qquad \frac{P \xrightarrow{\alpha} P' \wedge ch(\alpha) \notin A}{P/A \xrightarrow{\alpha} P'/A} \qquad \frac{P \xrightarrow{\alpha} P' \wedge ch(\alpha) \in A}{P/A \xrightarrow{\tau} P'/A}$$

$$\frac{\mathtt{Eval}(e),\ P \xrightarrow{\omega} P'}{\mathtt{if}\ e\ \mathtt{then}\ P\ \mathtt{else}\ Q \xrightarrow{\omega} P'} \qquad \frac{\neg\mathtt{Eval}(e),\ Q \xrightarrow{\omega} Q'}{\mathtt{if}\ e\ \mathtt{then}\ P\ \mathtt{else}\ Q \xrightarrow{\omega} Q'}$$

$$\frac{P \xrightarrow{a!(v)} P'}{(P,\mu) \xrightarrow{a!(v)} (P',\mu')} \qquad \frac{P \xrightarrow{a?(x)} P',\ \mu_a \neq [\ ],\ v{=}head(\mu_a)}{(P,\mu) \xrightarrow{a?(v)} (P'[x/v],\mu'')} \qquad \frac{P \xrightarrow{\tau} P'}{(P,\mu) \xrightarrow{\tau} (P',\mu)}$$

$$\begin{array}{ll} \mu' = put(\mu,\mu_a,v) & \mu'' = get(\mu,\mu_a) \\ put(\mu,\mu_a,v) = \mu - \{\mu_a\} \cup \{enqueue(v,\mu_a)\} & get(\mu,\mu_a) = \mu - \{\mu_a\} \cup \{dequeue(\mu_a)\} \end{array}$$

Fig. 1. Operational semantics of `VPASPA` and `VPASPA`* (1/2).

modified (by using the function *get*). The third rule deals with internal actions. If a process P may evolve internally then (P,μ) will also perform an internal action and the queue will not change.

Example 2.1. Let us consider the processes $P_1 = a?(x)\,;Q_1$ and $P_2 = a!(v)\,;b?(x)\,;Q_2$. Let $P = P_1 \|_\emptyset P_2$ be the parallel composition of both processes and (P,I) its image in `VPASPA`*. We have that (P,I) may initially perform only the transition $(P,I) \xrightarrow{a!(v)} (P_1 \|_\emptyset b?(x)\,;Q_2,\mu)$, where μ is the set of queues such that for any $a' \in \mathcal{C}_{\mathtt{Act}-\{a\}}$ we have $\mu_{a'} = [\ ]$ and $\mu_a = [v]$. Let us consider now $(P',\mu) = (P_1 \|_\emptyset b?(x)\,;Q_2,\mu)$. The right hand component of the parallel composition in (P',μ) is blocked until a value is received through the channel b. So, the only available transition is $(P',\mu) \xrightarrow{a?(v)} (Q_1[x/v] \|_\emptyset b?(x)\,;Q_2,I)$. Let us note that the value previously stored in μ_a has been consumed. Thus, the set of queues becomes again the set of empty queues. □

Next we present the operational rules dealing with stochastic actions (see Figure 2). The definition of this part of the operational semantics is inspired by [BG02]. Before explaining these rules, we will define some auxiliary predicates and functions. We will consider that if a process is able to perform an *urgent action*, that is, either an internal action or an output action sending a value, or an input action such that the queue associated with the corresponding channel is not empty (that is, a value can be dequeued to be received), then no stochastic transition will be allowed. Intuitively, the predicate *urgent*(P,μ) holds iff (P,μ) may immediately perform an action transition.

Definition 2.5. Let $(P,\mu) \in$ `VPASPA`*. We define the predicate *urgent*(P,μ) as follows:

$$urgent(P,\mu) \text{ iff } \exists (P',\mu') \in \mathtt{VPASPA}^* : (P,\mu) \xrightarrow{\omega} (P',\mu') \text{ and } \omega \in \{a?(v), a!(v) \mid a \in \mathcal{C}_{\mathtt{Act}}, v \in Val\} \cup \{\tau\}$$

□

Let us remark that the use of the predicate *urgent* in the forthcoming operational rules in Figure 2 does not generate *recursive calls* since this predicate has been defined based only on the rules appearing in Figure 1. We will use an additional auxiliary predicate. Given the fact that start of delays has higher priority than

$$\frac{}{\xi;P \xrightarrow{\xi_1^+} \xi_1^-;P} \qquad \frac{}{\xi_1^-;P \xrightarrow{\xi_1^-} P}$$

$$\frac{P \xrightarrow{\xi_i^+} P'}{P+Q \xrightarrow{\xi_i^+} P'+Q} \qquad \frac{P \xrightarrow{\xi_i^-} P'}{P+Q \xrightarrow{\xi_i^-} P'}$$

$$\frac{Q \xrightarrow{\xi_i^+} Q'}{P+Q \xrightarrow{\xi_i^+} P+Q'} \qquad \frac{Q \xrightarrow{\xi_i^-} Q'}{P+Q \xrightarrow{\xi_i^-} Q'}$$

$$\frac{P \xrightarrow{\xi_i^+} P'}{P\|_M Q \xrightarrow{\xi_{n(M_\xi)}^+} P'\|_{M\cup\{(\xi,n(M_\xi),l_i)\}}Q} \qquad \frac{P \xrightarrow{\xi_i^-} P',\ (\xi,j,l_i)\in M}{P\|_M Q \xrightarrow{\xi_j^-} P'\|_{M-\{(\xi,j,l_i)\}}Q}$$

$$\frac{Q \xrightarrow{\xi_i^+} Q'}{P\|_M Q \xrightarrow{\xi_{n(M_\xi)}^+} P\|_{M\cup\{(\xi,n(M_\xi),r_i)\}}Q'} \qquad \frac{Q \xrightarrow{\xi_i^-} Q',\ (\xi,j,r_i)\in M}{P\|_M Q \xrightarrow{\xi_j^-} P\|_{M-\{(\xi,j,r_i)\}}Q'}$$

$$\frac{P \xrightarrow{\xi_i^+} P'}{P/A \xrightarrow{\xi_i^+} P'/A} \qquad \frac{P \xrightarrow{\xi_i^-} P'}{P/A \xrightarrow{\xi_i^-} P'/A}$$

$$\frac{P \xrightarrow{\xi_i^+} P',\ \neg urgent(P,\mu)}{(P,\mu) \xrightarrow{\xi_i^+} (P',\mu)} \qquad \frac{P \xrightarrow{\xi_i^-} P',\ \neg urgent(P,\mu),\ \mathtt{Init}(P,\mu)=\emptyset}{(P,\mu) \xrightarrow{\xi_i^-} (P',\mu)}$$

Fig. 2. Operational semantics of VPASPA and VPASPA* (2/2).

any termination action, we need to take into account the set of starting stochastic actions that a process can perform.

Definition 2.6. Let P be a process in VPASPA. We define the *set of initial stochastic actions of a process* P, denoted by $\mathtt{InStoc}(P)$, as the set of stochastic actions that the process can start immediately, that is,

$$\begin{array}{lllll}
\mathtt{InStoc}(\text{stop}) & = & \mathtt{InStoc}(a?(x)\,;P) & = & \mathtt{InStoc}(a!(v)\,;P) = \emptyset \\
\mathtt{InStoc}(\xi\,;P) & = & \{\xi\} & & \\
\mathtt{InStoc}(P+Q) & = & \mathtt{InStoc}(P\,\|_M\,Q) & = & \mathtt{InStoc}(P)\cup\mathtt{InStoc}(Q) \\
\mathtt{InStoc}(P/A) & = & \mathtt{InStoc}(P) & & \\
\mathtt{InStoc}(X := P) & = & \mathtt{InStoc}(P[X/X := P]) & &
\end{array}$$

$$\mathtt{InStoc}(\mathtt{if}\ e\ \mathtt{then}\ P\ \mathtt{else}\ Q) = \begin{cases} \mathtt{InStoc}(P) & \text{if } \mathtt{Eval}(e) \\ \mathtt{InStoc}(Q) & \text{otherwise} \end{cases}$$

Let $(P,\mu)\in$ VPASPA*. We define the *set of initial stochastic actions of* (P,μ), denoted by $\mathtt{Init}(P,\mu)$, as

$$\mathtt{Init}(P,\mu) = \begin{cases} \emptyset & \text{if } urgent(P,\mu) \\ \mathtt{InStoc}(P) & \text{otherwise} \end{cases}$$

□

In the first part of Figure 2, all stochastic transitions will be allowed without taking into account the priority of internal actions over stochastic actions, and of starting actions over termination actions. These restrictions will be given in the definition of the upper level of the operational semantics (the last two rules of Figure 2). Let us note that the rules on the left hand side consider the performance of starting actions, while the right hand side rules consider termination actions. An operational transition as $\xi \,;\, P \xrightarrow{\xi_1^+} \xi_1^- \,;\, P$ indicates the start of a delay given by ξ. Similarly, $\xi_1^- \,;\, P \xrightarrow{\xi_1^-} P$ denotes the termination of the delay. As we explained before, we index stochastic actions to distinguish different occurrences of the same variable. In the case of prefix, we just add the label 1 (a more complex mechanism to assign labels appears in the context of the parallel operator).

The following four rules define the behavior of the choice between processes when the actions to perform are stochastic. The left hand side rules indicate that one of the components may start a delay. Let us note that starting actions do not resolve the choice because they only denote the possibility of delaying a process. The delay finishes when the corresponding termination action is performed. The rules in the right hand side consider the case of these actions. Besides, termination actions resolve choices. This interpretation of the choice operator follows the so-called *race policy*. An example of the race policy is given in the forthcoming example.

The behavior of the parallel operator is defined by the following four rules appearing in Figure 2. If any of the processes of the composition can perform a start action, left hand side rules, then the action is performed. However, a new index is given to the stochastic action (see Definition 2.7 below for the function determining this new index). In order to index random variables we use dynamic names. The chosen index will be the minimum natural number that it is not being currently used to label other occurrences of the same random variable in the process. Once a stochastic transition is completed, its corresponding index is liberated and the set of current indexes associated with the parallel operator has to be modified. Actually, M records both the indexes that are in use for each random variable, and the *localities* of the corresponding random variable, that is, if it has been performed either from the left hand side (marked with a l) or from the right hand side of the parallel composition (marked with a r). It also records the index associated with the random variable in the environment of the component that performs the action.

Definition 2.7. Let $\xi \in \mathcal{V}$, $M \subseteq \mathcal{V} \times \mathbb{N}^+ \times \mathrm{Loc}$. We define the *minimum value not used in M to index ξ*, denoted by $n(M_\xi)$, as

$$n(M_\xi) = \min\big(\{j \mid \nexists (\xi, j, loc_i) \in M, loc \in \{l, r\}, i \in \mathbb{N}^+ \ \wedge\ 1 \le j \le mx\} \cup \{mx + 1\}\big)$$

where $mx = \max\{j \mid (\xi, j, loc_i) \in M, loc \in \{l, r\}, i \in \mathbb{N}^+\}$. □

For example, if the tuple $(\xi, n(M_\xi), l_i)$ is stored in M, this means that ξ^+ has been performed from the left hand side of the parallel operator due to P has performed ξ_i^+.

If the termination action ξ_i^- can be performed by P, that is, we have $(\xi, j, l_i) \in M$, then the parallel composition is able to perform the same termination action but indexed by j. In addition, the corresponding index is erased from M. A similar situation appears for Q and (ξ, j, r_i). The following two rules deal with hiding. If a process P may perform a stochastic action then P/A is able to perform it too. Let us remark that urgency in the context of the hiding operator due to *new* hidden actions will be dealt with, as well as other *urgency* situations, in the second-level operational rules.

Finally, the last two rules consider the second-level operational semantics for stochastic actions. In both rules, we have to consider that no stochastic action can be performed if one of the components can evolve urgently. If a process P can perform a starting stochastic action, and P is not able to perform any urgent action then (P, μ) can perform the starting stochastic action and the set of queues will not vary. Besides, as we said before, starting actions have priority over termination actions. The predicate $\texttt{Init}(P, \mu) = \emptyset$ indicates that the process (P, μ) cannot perform any starting action. Thus, if P may perform a termination action and neither urgent nor starting actions are available, then (P, μ) is able to finish its delay without modifying μ. In particular, a stochastic action cannot be *completed* if another stochastic action can be started.

Example 2.2. Let us consider the process $P = \xi \,;\, P_1 + \psi \,;\, P_2$ and a queue μ. We have that (P, μ) may perform the transitions $(P, \mu) \xrightarrow{\xi_1^+} (Q_1, \mu)$ and $(P, \mu) \xrightarrow{\psi_1^+} (Q_2, \mu)$, where $Q_1 = \xi_1^- \,;\, P_1 + \psi \,;\, P_2$ and

$$\begin{tikzcd}
 & & \xi \,;\, P_1 + \psi \,;\, P_2 \arrow[dll, "\xi_1^+"'] \arrow[drr, "\psi_1^+"] & & \\
\xi^- \,;\, P_1 + \psi \,;\, P_2 \arrow[drr, "\psi_1^+"'] & & & & \xi \,;\, P_1 + \psi^- \,;\, P_2 \arrow[dll, "\xi_1^+"] \\
 & & \xi^- \,;\, P_1 + \psi^- \,;\, P_2 \arrow[dll, "\xi_1^-"'] \arrow[drr, "\psi_1^-"] & & \\
P_1 & & & & P_2
\end{tikzcd}$$

Fig. 3. Example of race policy.

$Q_2 = \xi \,;\, P_1 + \psi_1^- \,;\, P_2$. Then, (Q_1, μ) may perform only the transition $(Q_1, \mu) \xrightarrow{\psi_1^+} (R, \mu)$ while (Q_2, μ) may perform the transition $(Q_2, \mu) \xrightarrow{\xi_1^+} (R, \mu)$, for $R = \xi_1^- \,;\, P_1 + \psi_1^- \,;\, P_2$. That is, in both cases we have that after two transitions P evolves into the process R. Finally, R may perform one of the two available termination actions. In both cases the choice is resolved. That is, we have the transitions $(R, \mu) \xrightarrow{\xi_1^-} (P_1, \mu)$ and $(R, \mu) \xrightarrow{\psi_1^-} (P_2, \mu)$. In each case, we consider that P has been delayed according to the random variable ξ (resp. ψ). The intuitive interpretation of these transitions is that the process P will be delayed an amount of time depending on which random variable took the smallest value (according to their corresponding probability distribution functions). In Figure 3 we show a graphical representation of this example. $\Box$

Next we define the set of *input actions* and *output actions* that a process can perform. These notations will be used in the forthcoming Section 4.

Definition 2.8. We inductively define the set of *input actions* that a process P can perform, denoted by $\texttt{Inputs}(P)$, as:

$$\begin{array}{ll}
\texttt{Inputs}(\text{stop}) = \emptyset & \texttt{Inputs}(P + Q) = \texttt{Inputs}(P) \cup \texttt{Inputs}(Q) \\
\texttt{Inputs}(\xi \,;\, P) = \texttt{Inputs}(P) & \texttt{Inputs}(P \parallel_M Q) = \texttt{Inputs}(P) \cup \texttt{Inputs}(Q) \\
\texttt{Inputs}(a!(v) \,;\, P) = \texttt{Inputs}(P) & \texttt{Inputs}(P/A) = \texttt{Inputs}(P) - A \\
\texttt{Inputs}(a?(x) \,;\, P) = \texttt{Inputs}(P) \cup \{a\} & \texttt{Inputs}(X := P) = \texttt{Inputs}(P) \\
\multicolumn{2}{l}{\texttt{Inputs}(\texttt{if } e \texttt{ then } P \texttt{ else } Q) = \texttt{Inputs}(P) \cup \texttt{Inputs}(Q)}
\end{array}$$

We denote by $\texttt{Outputs}(P)$ the set of *output actions* that P can perform. Formally:

$$\begin{array}{ll}
\texttt{Outputs}(\text{stop}) = \emptyset & \texttt{Outputs}(P + Q) = \texttt{Outputs}(P) \cup \texttt{Outputs}(Q) \\
\texttt{Outputs}(\xi \,;\, P) = \texttt{Outputs}(P) & \texttt{Outputs}(P \parallel_M Q) = \texttt{Outputs}(P) \cup \texttt{Outputs}(Q) \\
\texttt{Outputs}(a!(v) \,;\, P) = \texttt{Outputs}(P) \cup \{a\} & \texttt{Outputs}(P/A) = \texttt{Outputs}(P) - A \\
\texttt{Outputs}(a?(x) \,;\, P) = \texttt{Outputs}(P) & \texttt{Outputs}(X := P) = \texttt{Outputs}(P) \\
\multicolumn{2}{l}{\texttt{Outputs}(\texttt{if } e \texttt{ then } P \texttt{ else } Q) = \texttt{Outputs}(P) \cup \texttt{Outputs}(Q)}
\end{array}$$

$\Box$

In order to conclude the presentation of our language, we would like to remark that in the trade-off between complexity of the semantic model and capabilities of the language, we have preferred to define a very expressive language. Actually, a designer using our language may take advantage of the complex features, just by having an intuitive idea of the operators, without going through all the semantic details.

2.1. Stochastic Strong Bisimulation for VPASPA

In this section we define a notion of (strong) bisimulation for our language. First, we introduce the function $\oplus$. This operator is used to compute a random variable distributed as the minimum of a finite set of random variables.

Definition 2.9. Let ξ_1, ξ_2 be independent random variables with probability distribution functions F_{ξ_1} and F_{ξ_2}, respectively. We define the *combined addition* of ξ_1 and ξ_2, denoted by $\xi_1 \oplus \xi_2$, as the random variable $F_{\xi_1 \oplus \xi_2}(x) = F_{\xi_1}(x) + F_{\xi_2}(x) - F_{\xi_1}(x) \cdot F_{\xi_2}(x)$.

This operator can be generalized to an arbitrary (finite) number of random variables. Let $\Psi = \{\xi_i\}_{i \in I}$ be a non-empty finite set of independent random variables. We define the *combined addition* of the variables in Ψ, denoted by $\oplus\Psi$, as the random variable such that $F_{\oplus\Psi}(x) = \sum_{\emptyset \subset \Phi \subseteq \Psi} (-1)^{(|\Phi|+1)} F_{\otimes\Phi}(x)$ where the function $F_{\otimes\Psi}$ is defined as $F_{\otimes\Psi}(x) = \prod_{i \in I} F_{\xi_i}(x)$. We consider, for convenience, that we have $F_{\oplus\emptyset}(x) = 0$ for any $x \in \mathbb{R}$. □

Let us note that for singleton sets, that is $\Psi = \{\xi\}$, we have $\oplus\Psi = \xi$. Also note that this operator does not correspond with the usual definition of addition of random variables. Next, we present our notion of bisimulation. We start by giving an auxiliary definition.

Definition 2.10. Let $(P, \mu) \in$ VPASPA*, $C \subseteq$ VPASPA*, and $t_0 \in \mathbb{R}^+$. The *cumulative rate to reach the set C from (P, μ) before time* t_0, denoted by $\xi_s((P, \mu), C, t_0)$, is defined as

$$\oplus\{\xi \mid (P, \mu) \xrightarrow{\xi_i^+} (P', \mu) \text{ and } (P', \mu) \in C \text{ and } F_\xi(t_0) > 0\}$$

□

The cumulative rate of a process computes the combined addition of all the stochastic transitions that the process can perform to reach any process belonging to C. For example, the process $P = (\xi\,; P') + (\psi\,; P')$ has a probability to reach P' before a certain amount of time given by the minimum of both random variables, that is, the combined addition of ξ and ψ. Let us note that in the case that ξ and ψ were exponential distributions with parameters λ_1 and λ_2 respectively, the probability to reach P' would be given by an exponential distribution with parameter $\lambda_1 + \lambda_2$.

Let us note that the operational semantics allows to generate all the stochastic actions. However some of them could not be performed. Let us consider the following process $P = \xi + \psi$, where ξ is distributed like a uniform distribution in the interval $[1, 2]$, that is, it is able to be performed with probability 1 in time 2 and ψ is distributed like a uniform distribution in the interval $[4, 5]$, that is, it is able to be performed with probability greater than 0 only after 4 units of time have passed. In this case ψ will never be finished (not even started). Thus, these possibilities have to be taken into account. So, we have included the condition $F_\xi(t) > 0$ for a particular time t to consider only those feasible random variables while computing the cumulative rate to reach a certain set from a particular process. In the definition of the bisimulation semantics, this time will be the maximum waiting time of the processes considered. This maximum waiting time of a process is defined as the maximum time that can pass before the process must perform either an urgent or a stochastic action. Formally,

Definition 2.11. Let (P, μ) be a process in VPASPA*. We define the *maximum waiting time of* (P, μ), denoted by $maxW(P, \mu)$, as

$$maxW(P, \mu) = \begin{cases} 0 & \text{if } urgent(P, \mu) \\ \\ \min\{t \mid \exists \xi \in \mathcal{V} : \, (P, \mu) \xrightarrow{\xi_i^+} \wedge F_\xi(t) = 1\} & \text{otherwise} \end{cases}$$

□

We define bisimulation in two steps. First, we introduce the equivalence for communication processes. Then, we say that two processes P and Q are bisimilar if the corresponding communication processes (P, I) and (Q, I) are bisimilar with respect to the previous notion.

Definition 2.12. An equivalence relation $\mathcal{R}^*$ on $\texttt{VPASPA}^*$ is a *strong stochastic bisimulation* iff for any pair of processes $(P,\mu),(Q,\nu) \in \texttt{VPASPA}^*$ such that $((P,\mu),(Q,\nu)) \in \mathcal{R}^*$ we have that for any $\alpha \in IO \cup \{\tau\}$ and for any $C \in \texttt{VPASPA}^*/\mathcal{R}$:

- $(P,\mu) \xrightarrow{\alpha} (P',\mu')$ implies $\exists (Q',\nu') \in \texttt{VPASPA}^*$ such that $(Q,\nu) \xrightarrow{\alpha} (Q',\nu')$ and $((P',\mu'),(Q',\nu')) \in \mathcal{R}^*$.
- $maxW(P,\mu) = t$ implies $t = maxW(Q,\nu)$ and $\xi_s((P,\mu),C,t) \asymp_t \xi_s((Q,\nu),C,t)$.

We say that two processes $P,Q \in \texttt{VPASPA}$ are *strongly stochastic bisimilar*, denoted by $P \sim_s Q$, if there exists a strong stochastic bisimulation that contains the pair $((P,I),(Q,I))$. □

Let us remark that we do not include the symmetrical cases (where P and Q exchange *roles*) because we already ask $\mathcal{R}^*$ to be an equivalence relation (in particular $\mathcal{R}^*$ is symmetric).

Lemma 2.1. Strong stochastic bisimilarity on $\texttt{VPASPA}$ is an equivalence relation on $\texttt{VPASPA}$. Moreover, $\sim_s$ is the largest strong stochastic bisimulation on $\texttt{VPASPA}$. □

3. The Concurrent Functional Language Eden

In this section we sketch the main features of Eden. Some knowledge of the functional paradigm is desirable, but is not essential. In particular, we will omit the description of the typing mechanism. Regarding the pure functional paradigm we will briefly explain a powerful mechanism to improve the quality of functional programs: Pattern Matching. Most of the programs written by using pattern matching can be rewritten by using `if then else` constructions, but the resulting programs are not so elegant. Then, we will concentrate on the concurrent features of Eden. A complete presentation of Eden can be found in [BLOP98].

A functional program consists in a list of function definitions. Each function can be defined in terms of several rules. Pattern matching is used to specify when a rule can be applied. A rule can be applied only if the arguments *match* the corresponding *pattern*. For instance, in the following example, the first rule is applied if its second input *matches* the *pattern* associated to the empty list, while the second rule is applied otherwise:

```
map f []     = []
map f (x:xs) = f x : map f xs
```

where `[]` denotes an empty list, and `(x:xs)` denotes a list whose head is `x` and whose tail is `xs`. As shown below, the `map` function can also be defined without using pattern matching. However, in general, pattern matching versions use to be shorter and clearer.

```
map f xs = if empty xs then []
                       else f head(xs) : map f tail(xs)
```

Eden extends Haskell by means of syntactical constructions to define and create (concurrent) processes. Eden distinguishes between *process abstractions* and *process instantiations*. The former are used to define the behavior of processes, but without actually creating them. The latter are used to create instances of processes. This distinction allows the creation of as many instances of the same process as needed. Eden includes a new expression `process x -> e` having `x` as input(s) and `e` as output(s). Process abstractions can be compared to functions. The main difference is that the former, when instantiated, are executed in a separate process. Besides, they perform communications by using the interface of that process. For example, the following process has two inputs and two outputs. It receives an integer and a string as inputs, and it produces a string and an integer as outputs.

```
p = process (n,s) -> (out1,out2)
 where (out1,out2) = f n s
       f 0 s = ("hello",length s)
       f n s = ("bye",length s)
```

The first thing this process needs to know is whether the first parameter is zero or not. So, it must wait until it receives a value through the first input channel. Afterwards, depending on that value, it will output either the string ‘hello’ and the length of the second input (if the input value is 0), or the string ‘bye’ and also the length of the second input (otherwise).

A *process instantiation* is achieved by using the predefined infix operator `(#)`. Each time a binding `outputs = p # inputs` is found, a new process is created to evaluate the abstraction `p`. This new process will be able to receive values through the input channels associated to `inputs`, and send values through `outputs`. The actual readers of `outputs` and the actual producers of `inputs` will be detected by means of the data dependencies of processes. For instance, in the next example, `p1` will be able to receive data both from the second output of `p2` and from the first output of `p3`. The actual readers of both `c` and `g` will depend on the context in which process `q` is instantiated:

```
 q = process () -> (c,g)
  where (a,b,c) = p1 # (e,f)
        (d,e)   = p2 # (h,b)
        (f,g,h) = p3 # (a,d)
```

In addition to process abstractions and instantiations, Eden provides a predefined function called `merge` that can be used to combine a list of lists into only one. This is useful to select from a list of alternatives. For instance, in case we want to select from two alternatives, the following `merge2` function can be used:

```
data Either a b = Left a | Right b
merge2 xs ys = merge [map Left xs, map Right ys]
```

After combining two lists `xs` and `ys`, we can use pattern matching over the merged list to know which inputs come from the *left* lists and which ones from the *right* list. This will be useful to choose between two different possible alternatives:

```
   ... f (merge2 xs ys) ...
   f (Left x : zs ) = ...
   f (Right y : zs) = ...
```

Now we will present an Eden example. The following process defines the behavior of a simple timer. A timer receives as input a list of signals. When it receives a `Start`, the `timer` process waits `t` seconds, then it communicates a `TimeOut`, and after that it recursively starts again waiting for a new `Start`. The function `seq` is used to sequentially compose two actions, and `sleep` is a Haskell primitive for delaying a process. Let us note that Eden does not include *send* or *receive* constructors: Communication is automatic and only happens because of the data dependencies of the processes. Actually, this mechanism is close to the one for communication in process algebras.

```
timer t = process starts -> f starts
  where f (x:xs) = seq (sleep t) (TimeOut : f xs)
```

Let us note that both the input and the output of the `timer` process are lists of messages. In fact, Eden processes use lists as both input and output channels. This enables to use the same Eden channel to communicate several times. Moreover, considering channels as lists will allow us to simulate a recursive process (of the process algebra) by using a unique Eden process. This Eden process will be defined by means of a recursive function. In this case, reading several times through the same channel corresponds to reading several elements of the corresponding input list; sending several times through an output channel corresponds to writing several elements to the output list. In the rest of the paper, we will assume that any Eden process communicates by using lists.

Finally, let us comment on the features that Eden includes to analyze quantitative properties. If we are interested in measuring different aspects of our programs, it is not enough (in general) to run them (several times). In order to know the *real* behavior of our programs, we need some extra feedback. We can obtain that feedback by using Paradise [HPR00], an Eden profiler based on GranSim [Loi96]. When using Paradise, the Eden program runs as usually but it also records, in a log file, all the relevant events happening during the execution: Creation of processes, communications of values, processes getting blocked, etc. After finishing the execution, several visualization tools can be used to analyze the log results. For instance, the evolution in time of the state, either running or blocked waiting for communication, of a process or set of processes can be viewed. It is also possible to combine log files corresponding to different executions in order to obtain statistics of properties about the behavior of the processes. This allows us to check whether the actual results really fit the theoretical predictions, as well as to obtain accurate statistics without performing complex theoretical studies.

4. The Translation from VPASPA to Eden

In this section we present how process algebraic specifications are translated into Eden programs. We will consider that an Eden program implements a VPASPA specification if any (input, output, or internal) transition of the specification can be *simulated* by a channel of the Eden program, and if any delay of the specification is reflected in that program. The rest of this section is devoted to present our methodology for the translation of VPASPA specifications into Eden programs. Let us remark that even though the translation is done automatically, in order to optimize the generated code it is needed some command in functional programming, although not necessarily in Eden.

We suppose a specification written in VPASPA where we have a set of equations as:

$$X_1 := P_1 \qquad X_2 := P_2 \qquad \cdots \qquad X_n := P_n$$

We will impose the following restriction: Any occurrence of parallel operators in the processes P_j will be of the form: $X_{i_1} \parallel_\emptyset (X_{i_2} \parallel_\emptyset \cdots (X_{i_{j-1}} \parallel_\emptyset X_{i_j}) \cdots)$ where $\{X_{i_1}, \cdots, X_{i_j}\} \subseteq \{X_1, \cdots, X_n\}$. Intuitively, we consider that any occurrence of the parallel operator represents the composition of *real* processes (so, they have a *name*). Note that this is not a real restriction because $P \parallel_\emptyset Q$ can always be defined as $X := P$, $Y := Q$ and then $X \parallel_\emptyset Y$. This restriction simplifies the translation mechanism while we do not lose expressibility. Let us remark that our syntax allows to index parallel operators with any set of indexes. However, as we have already commented, non-empty sets of indexes are used only in the definition of the operational semantics. That is why we have assumed that all of them are (initially) empty. Let us also note that these indexes do not appear in the translation into Eden. This is so because every Eden process will manage its own delays. So, there cannot be any confusion, as it happens in the process algebra, regarding which process is delayed. For a similar reason, the distinction between start and termination actions appearing in the definition of the operational semantics is not needed.

First, we compute the sets of input and output channels of the VPASPA processes $P_1, \ldots, P_n$. For any $1 \leq j \leq n$ let $I_j = \{i_{j1}, \ldots, i_{js_j}\} = \texttt{Inputs}(P_j)$ and $O_j = \{o_{j1}, \ldots, o_{jr_j}\} = \texttt{Outputs}(P_j)$. We will generate n process abstractions in Eden. These process abstractions will have the same (number of) input and output channels as the corresponding process P_i:

```
F1 = process (i11,...,i1s1) -> (o11,...,o1r1)
 where (o11,...,o1r1) = f1 i11 ... i1s1
F2 = process (i21,...,i2s2) -> (o21,...,o2r2)
 where (o21,...,o2r2) = f2 i21 ... i2s2
 .....
Fn = process (in1,...,insn) -> (on1,...,onrn)
 where (on1,...,onrn) = fn in1 ... insn
```

In the special case where the output channels are independent we can use a different function for defining each of these channels. These functions will only depend on the inputs of the process:

```
F = process (i1,...,is) -> (o1,...,or)
 where o1 = f1 i1 ... is
        ....
       or = fr i1 ... is
```

Depending on the processes P_j, the functions `f1,...,fr` will be defined in a different way. If $P_j = \mathrm{stop}$ then we simply consider that there are no outputs. More exactly, empty lists are output through all of the output channels (let us recall that in Eden, input and output channels are considered to be lists). The three cases corresponding to prefixes are easy. If $P_j = a!(v)\,;P$ then we will have a process:

```
Fj = process (i1,...,is) -> (o1,...,v:oj,...,or)
 where (o1,...,oj,...,or) = translation(P)
```

where `oj` is the output channel corresponding to $a!$. In this case, we firstly output the value `v` through `oj` and then we follow with the translation of P. If $P_j := a?(x)\,;P$ we have:

```
Fj = process (i1,...,ij,...,is) -> (o1,...,or)
 where (o1,...,or)                        = g i1 ... ij ... is
       g i1 ... (Patternsj : vs) ... is = translation(P)
```

where `ij` is the input channel corresponding to $a?$. In this case, we use an auxiliary function `g`. This function is defined by pattern matching on the input `ij`: It will take different values according to the received value through the input channel `ij`. The different patterns will have to do with the occurrences of constructors `if then else` in P. If there are no such occurrences, we will have a unique pattern (that is, there is no distinction in P depending on the values taken by x); otherwise, the boolean conditions appearing in the definition of the `if then else` constructors will be taken into account to define the different `Patterns`.

Example 4.1. Let $P := a?(x)\,;$ `if` $(x = 1)$ `then` P_1 `else` P_2. The translation of P is given by:

```
FP = process (i1,...,is) -> (o1,...,or)
 where (o1,...,or) = g i1...is
       g (1 : vs) i2...is = translation(P1)
       g (n : vs) i2...is = translation(P2)
```

where the inputs of the process are $\texttt{Inputs}(P_1) \cup \texttt{Inputs}(P_2) \cup \{a\} = \{\texttt{i1}, ..., \texttt{is}\}$, the outputs are $\texttt{Outputs}(P_1) \cup \texttt{Outputs}(P_2) = \{\texttt{o1}, ..., \texttt{or}\}$, and we suppose that the input channel corresponding to $a?$ is `i1`. □

If $P_j = \xi\,; P$ then we will have a process:

```
Fj = process (i1,...,is) -> (o1,...,or)
 where (o1,...,or) = seq (wait xi) (translation(P))
```

Let us remind that `seq` represents the sequential composition operator. Besides, `(wait xi)` interrupts the execution of the process by a random amount of time depending on the distribution given by `xi`. We have added `wait` to Eden, as no such operator was defined in the original compiler. The implementation of this operator is based on the primitive constructor `sleep` and on the generation of random delays. We have also included in Eden the most common distributions. For instance, exponential distributions are translated as `wait(expo lambda)` and uniform distributions over the interval (a, b) are translated as `wait(unif a b)`. In this paper we will also use `wait(dirac a)`, `wait(poisson lambda)`, and `wait(normal a b)` for Dirac, Poisson, and normal distributions, respectively. Obviously, more distributions can be added to Eden as needed. In this case, we would like to remark that functional languages (in particular Eden) are very suitable to define different distributions and to deal with them. Moreover, this is the case even if they depend on a different number of parameters.

Analogously to the `if then else`, the choice operator is also translated by using pattern matching. The difference in this case is that `merge2` is used to decide which branch should be chosen. In order to choose between two alternatives, we merge the corresponding channels. The appropriate branch is selected by pattern matching on the left or right.

```
  out = f (merge2 alt1 alt2)
    where f (Left  a1 : vs) = ...
          f (Right a2 : vs) = ...
```

If P_j is the parallel composition of m process variables, that is, if P_j is the following process $P_j = Y_1 \parallel_\emptyset (Y_2 \parallel_\emptyset \cdots (Y_{m-1} \parallel_\emptyset Y_m) \cdots)$, where $\{Y_1, \ldots, Y_m\} \subseteq \{X_1, \ldots, X_n\}$, we will generate a process instantiation for each of the variables:

```
Fj = process (i1,...,is) -> (o1,...,or)
 where (out11,...,out1r1) = G1 # (in11,...,in1s1)
       (out21,...,out2r2) = G2 # (in21,...,in2s2)
                       ........
       (outm1,...,outmrm) = Gm # (inm1,...,inmsm)
```

where for any $1 \leq l \leq m$ we have that `Gl` is the Eden process abstraction corresponding to Y_l and $\{\texttt{inl1}, ..., \texttt{inlsl}\}$, $\{\texttt{outl1}, ..., \texttt{outlrl}\}$ are the sets of input and output, respectively, channels of `Gl`. Let us note that, by the definition of the functions `Inputs` and `Outputs`, we have $\{\texttt{i1}, ..., \texttt{is}\} = \{\texttt{input} \mid \exists\, 1 \leq k \leq m,\ 1 \leq l \leq s_k : \texttt{input} = \texttt{inkl}\}$ (similarly for $\{\texttt{o1}, ..., \texttt{or}\}$). Besides, let us remark that we do not need to explicitly deal with the queues of the parallel operator. The reason is that Eden provides an internal queue for each channel of each process instantiation.

It is rather easy to implement the hiding operator. The only externally visible actions of `Fj` will be those declared in its input an output lists (that is, `i1...is`,`o1...or`), while the remaining actions performed by the `Gi`'s are hidden. Thus, if the set of actions `A` is to be hidden, it is enough to define $\texttt{Inputs(Fj)} = (\bigcup \texttt{Inputs(P}_\texttt{i}\texttt{)}) - \texttt{A}$, and similarly for the outputs. Notice that each output (e.g. `outij`) of a process may correspond to an input (e.g. `inkl`) of another process. In that case, to enable the communication, it is enough to use the same name in the translation (e.g. `a`) for both the input and the output.

Example 4.2. Let us consider the process P given in Example 4.1, where $P_1 = c?(y)\,;b!(1)$ and $P_2 = d!(0)$. Besides, let $Q := a!(1)\,;\text{stop}$ and let us take $(P \parallel_\emptyset Q)/\{a\}$. The translation of this last process is given by `F` below (the translation of Q is given by `FQ`).

```
FQ = process () -> a
   where a = [1]
F = process (c) -> (b,d)
  where (b,d) = FP # (a,c)
        a     = FQ # ()
```
□

Let us finally comment on some optimizations that can be done after the automatic translation from `VPASPA` into Eden is performed. In many situations, a specification involves several replicated processes P_i, all of them communicating with a unique process Q. In this case, a naïve translation will create a different channel in Q for each of the P_i processes. Thus, if we have n processes P_i then Q will have n different channels. Fortunately, a functional language like Eden can deal nicely with these situations, as we can trivially use lists of channels instead of explicitly handling n individual channels. We will present an illustrative example of this situation when defining a *Line* in Section 5.2. Moreover, in some situations the process Q behaves in the same way with all the processes P_i. So, each time that Q sends a value v to any process P_i, it also sends it to the rest of processes. If this is the case, we can take advantage of the multicasting facilities of Eden that allow us to use a single channel to communicate with several processes. For instance, in the following example we have that the process `q` can broadcast `start` messages to n different processes `p`:

```
  outps  = [(p i) # starts | i <- [0..n-1]
  starts = q # ()
```

As a final optimization, sometimes we may need to translate a multiple choice as $\sum_{i=1}^{n} signal_i?(x)$. In this case, we could implement it by means of applying $n-1$ times the translation of the binary choice. Nevertheless, we can do it better by using the `merge` function, that can deal with arbitrary long lists of alternatives. This can be specially useful when a process Q can receive messages from several processes P_i, as we will see in Section 5.2.

Definition 4.1. For any $P \in \texttt{VPASPA}$ we write $\texttt{translation}(P)$ to denote the Eden program produced as a translation of P. □

The translations produced by our algorithm are *correct* in the sense that they are conservative with respect to our notion of bisimulation and the semantic framework defined in [BLOP98] for Eden programs. The presentation of this operational semantics is out of the scope of this paper. Intuitively, operational rules show the evolution of the (parts of an) Eden program by taking into account the messages transmitted among the components. Equivalence between Eden programs with respect to this semantics is denoted by $\equiv$. The following result (the proof is not complex but cumbersome) states our notion of correctness.

Lemma 4.1. Let P, Q be strongly stochastic bisimilar processes. We have $P \sim_s Q$ iff $\texttt{translation}(P) \equiv \texttt{translation}(Q)$. □

5. Examples

In this section we present two relatively complex specifications to show that our translation mechanism produces Eden programs *close enough* to the original specifications. Let us remark that in our specifications, for the sake of clarity, we use sometimes additional auxiliary processes. So, in some cases, auxiliary processes in the specification will be embedded into a unique Eden process. The pattern for the presentation of both examples will be the same. First, we introduce the problem. Then we indicate the processes that we will use. Afterwards, for each specification we give the corresponding translation. Finally, in Section 5.3 we study some quantitative properties of the examples. Finally, in Section 5.3 we study some quantitative properties of the examples. It is worth to point out that the validity of our case studies strongly rely on two of the design decisions that we took when defining our process algebra: Messages are sent in the same order that they are received and both output and input transitions are performed as soon as possible. Let us remind that in the case of input transitions, they can be performed as soon as the corresponding queue is not empty.

We will use some classical distributions such as exponential distributions, denoted in the examples by $\xi_{exp(\lambda)}$, Poisson distributions, denoted by $\xi_{Po(\lambda)}$, normal distributions, denoted by $\xi_{N(a,b)}$, random variables uniformly distributed over an interval (a,b), denoted by $\xi_{U(a,b)}$, and Dirac distributions, denoted by $\xi_{D(c)}$.

5.1. The Hubble Space Telescope

This example is borrowed from [Her00]. However, we have introduced some changes to adapt it to our framework, where general distributions are allowed. We have chosen this example because, even though it is relatively simple, it presents in a very clear way most of the features appearing in a stochastic process algebra. The Hubble space telescope (HST) is an orbiting astronomical observatory launched in 1990 and scheduled to operate during more than twenty years. This telescope has six gyroscopes as a part of the HST pointing system. They provide the information about where the telescope is pointing, by taking into account the movements of the spacecraft. The system can work accurately with only three gyroscopes. If a fourth gyro fails then the HST turns itself into a sleep mode. Once the sleep mode has started, the Base prepares a mission to repair it. During the time of this preparation and reparation, there exists the possibility to crash if the remaining gyroscopes fail. If the telescope does not crash during this phase then the system is restarted.

In the specification of the system we will consider the following main processes: The *Gyro*, the *Controller*, the *Stabilizer*, the *Base*, and finally the *HST*.

The *Gyro*. A gyroscope has two main states: *working* and *broken*, being initially *working*. A working gyro can get broken after some time (given by an exponential distribution) or it can receive a *restart* signal. In the latter case the state will not vary; in the former, the state becomes *waiting*. The gyro will remain in this auxiliary *waiting* state until the controller acknowledges him that it has received its failure message. In that moment, it becomes *officially* broken. A broken gyro waits for a *restart* signal. Once this signal is received, the state of the gyro is again *working*.

$$
\begin{array}{lll}
Gyro_i & := & Gyro_{i,working} \\
Gyro_{i,working} & := & \xi_{exp(\lambda)}\,;fail!(i)\,;Gyro_{i,waiting} + restart_i?(-)\,;Gyro_{i,working} \\
Gyro_{i,waiting} & := & restart_i?(-)\,;Gyro_{i,waiting} + ack_i?(-)\,;Gyro_{i,broken} \\
Gyro_{i,broken} & := & restart_i?(-)\,;Gyro_{i,working}
\end{array}
$$

This specification is translated into Eden as shown below. As the specification of the gyro has three states, the Eden process will also use a state indicating whether the gyro is `Broken`, `Waiting`, or `Working`. Let us note that `merge2` is used in the `Working` state as a mechanism to choose between the two possible alternatives: failing or restarting:

```
gyro i = process (restarts,acks) -> fails
  where fails = f Working restarts acks
        f Working xs ys = fWork (merge2 [wait (expo lambda)] xs) xs ys
          where fWork (Left v : vs) xs ys     = Fail i : f Waiting xs ys
                fWork (Right v: vs) (x:xs) ys = f Working xs ys
        f Waiting xs ys = fWait (merge2 xs ys) xs ys
          where fWait (Left v : vs) (x:xs) ys = f Waiting xs ys
                fWait (Right v: vs) xs (y:ys) = f Broken xs ys
        f Broken (x:xs) ys = f Working xs ys
```

The *Controller*. The controller receives either *failure* messages from the gyros or *restartc* messages from the Base. Each time it receives a *failure* message, after some fix amount of time has passed (this is indicated by a Dirac distribution) it must acknowledge its reception to the corresponding gyro. If it receives four *failure* messages then a *sleep* message is generated. After that, the controller could receive two more *failure* messages; in this case, a *crash* message would be sent. When a *restartc* message is received from the Base, the controller is responsible for sending a message to the gyros so that they are aware of the restart phase. Besides, it also sets to zero the current number of broken gyros. The controller begins with zero failures received. We use six auxiliary processes to describe the behavior of the controller.

$$\begin{array}{lll}
Controller & := & Controller_0 \\
Controller_3 & := & restart_c?(-)\,;\xi_{D(c)}\,;restart_1!(-)\,;restart_2!(-)\,;restart_3!(-)\,;restart_4!(-); \\
 & & restart_5!(-)\,;restart_6!(-)\,;Controller_0 \\
 & & +\,fail?(x)\,;\xi_{D(c')}\,;ack_x!(-)\,;sleep!(-)\,;Controller_4 \\
Controller_5 & := & restart_c?(-)\,;\xi_{D(c)}\,;restart_1!(-)\,;restart_2!(-)\,;restart_3!(-)\,;restart_4!(-); \\
 & & restart_5!(-)\,;restart_6!(-)\,;Controller_0 \\
 & & +\,fail?(x)\,;\xi_{D(c')}\,;ack_x!(-)\,;crash!(-)\,;\mathrm{stop} \\
Controller_k & := & restart_c?(-)\,;\xi_{D(c)}\,;restart_1!(-)\,;restart_2!(-)\,;restart_3!(-)\,;restart_4!(-); \\
{\scriptstyle 0\leq k<5,\,k\neq 3} & & restart_5!(-)\,;restart_6!(-)\,;Controller_0 \\
 & & +\,fail?(x)\,;\xi_{D(c')}\,;ack_x!(-)\,;Controller_{k+1}
\end{array}$$

For the sake of clarity, in the previous specification we have used syntactic sugar when using the action $ack_i!(-)$. For instance, we could avoid using it by defining $Controller_3$ as follows:

$$\begin{array}{lll}
Controller_3 & := & restart_c?(-)\,;\xi_{D(c)}\,;restart_1!(-)\,;restart_2!(-)\,;restart_3!(-)\,;restart_4!(-); \\
 & & restart_5!(-)\,;restart_6!(-)\,;Controller_0 \\
 & & +\,fail?(x)\,;\xi_{D(c')}\,;\texttt{if}\ x=1\ \texttt{then}\ ack_1!(-)\,;sleep!(-)\,;Controller_4 \\
 & & \qquad\qquad\qquad\texttt{else if}\ x=2\ \texttt{then}\ ack_2!(-)\,;sleep!(-)\,;Controller_4\ \texttt{else}\ \ldots
\end{array}$$

In the translation, we also need to use auxiliary functions for each of the auxiliary processes. Additionally, as an optimization, the Eden program only needs one restart message that will be broadcast to all the gyros:

```
controller = process (fails,restartcs) -> (sleeps,crashs,ackss,restarts)
  where (sleeps,crashs,ackss,restarts) = f 0 restartcs fails
        f 3 xs ys = f3 (merge2 xs ys) xs ys
          where f3 (Left v : vs) (x:xs) ys = seq (wait (dirac c))(ys1,ys2,ys3,Restart:ys4)
                  where (ys1,ys2,ys3,ys4) = f 0 xs ys
                f3 (Right v: vs) xs (y:ys) = seq (wait (dirac c’))
                                                 (Sleep:ys1,ys2,insert v ys3,ys4)
                  where (ys1,ys2,ys3,ys4) = f 4 xs ys
        f 5 xs ys = f5 (merge2 xs ys) xs ys
          where f5 (Left v : vs) (x:xs) ys = seq (wait (dirac c))(ys1,ys2,ys3,Restart:ys4)
                  where (ys1,ys2,ys3,ys4) = f 0 xs ys
                f5 (Right v: vs) xs (y:ys) = seq (wait (dirac c’))
                                                 (ys1,Crash:ys2,insert v ys3,ys4)
                  where (ys1,ys2,ys3,ys4) = ([],[],[],[])
        f k xs ys = fk (merge2 xs ys) xs ys
          where fk (Left v : vs) (x:xs) ys = seq (wait (dirac c))(ys1,ys2,ys3,Restart:ys4)
                  where (ys1,ys2,ys3,ys4) = f 0 xs ys
                fk (Right v: vs) xs (y:ys) = seq (wait (dirac c’))
                                                 (ys1,ys2,insert v ys3,ys4)
                  where (ys1,ys2,ys3,ys4) = f (k+1) xs ys
```

The *Stabilizer*. The stabilizer is a process that communicates *failure* messages from the gyros to the controller. It also allows the controller to send *ack* and *restart* messages to the gyros. Let us remind that the parameter I appearing in the parallel operator denotes that all the queues associated with channels are initially empty.

$$Stabilizer \ := \ (Controller \parallel_{\emptyset} Gyro_1 \parallel_{\emptyset} Gyro_2 \parallel_{\emptyset} Gyro_3 \parallel_{\emptyset} Gyro_4 \parallel_{\emptyset} Gyro_5 \parallel_{\emptyset} Gyro_6)/A$$

where $A = \{fail\} \cup \{restart_l, ack_l \mid 1 \leq l \leq 6\}$. The Eden implementation just connects the inputs and outputs of the processes in the adequate way. All the *fail* messages outcoming from the gyros are merged into a unique list and they are communicated to the controller. Besides, the *restarts* are forwarded to all the gyros. This is actually an optimization of the resulting code because different names for communicating with all the gyros are not needed:

```
stabilizer = process restartcs -> (sleeps,crashs)
  where fails = merge [gyro # (restarts,ackss!!i) | i <-[0..5]]
        (sleeps,crashs,restarts,ackss) = controller # (fails,restartcs)
```

The *Base*. The base waits either a crash message or a sleep message. In the first case the process stops. If a sleep message is received then the base sends messages to prepare a mission, to launch it, and to repair the gyros. The time elapsed before the reparation is finished is given by an exponential distribution μ. If the gyros are repaired then a restart message is generated to be sent to the whole system.

$$\begin{array}{lll} Base & := & crash?(-)\,;finish!(-)\,;\mathrm{stop} \\ & & +sleep?(-)\,;\xi_{exp(\mu)}\,;restart_c!(-)\,;Base \end{array}$$

The Eden program only needs to follow the previous definition:

```
base = process (sleeps,crashs) -> (restartcs,finishs)
  where (restartcs,finishs) = f (merge2 crashs sleeps)
        f (Left x : xs)  = (ys1,Finish:ys2)
          where (ys1,ys2) = ([],[])
        f (Right x : xs) = seq (wait (expo mu)) (Restartc: ys1,ys2)
          where (ys1,ys2) = f xs
```

The *HST*. Finally, the HST corresponds to the whole system. In this process we have the communication between the base and the stabilizer.

$$HST \quad := \quad (Base \parallel_{\emptyset} Stabilizer)/\{restart_c, crash, sleep\}$$

The translation into Eden only needs to establish the appropriate connections:

```
hst = finishs
  where (sleeps,crashs)    = stabilizer # restartcs
        (restartcs,finishs) = base # (sleeps,crashs)
```

5.2. CSMA/CD

Our second example is more complex than the previous one. It describes a real protocol for local area networks with bus topologies. Another formalization of the same protocol appears in [Ber99]. However, our presentation is completely different. For instance, we are not restricted to exponential distributions, we include value passing, and we allow asynchronous communications.

The protocol we describe is the CSMA/CD (see e.g. [Tan96]), the IEEE 802.3 standard. We consider n stations connected by a bus. If a station wants to transmit a value then it has to listen to the channel in order to determine whether it is idle or it is being used by another station. In the former case, the station starts sending its message; in the latter case it has to back off a random amount of time before checking the channel again. It must be taken into account that a collision with a message of another station can occur while a station is transmitting a message. In that case, both stations must back off a random amount of time before trying to send their messages again. Fortunately, collisions cannot take place at any moment, but only during a short period at the beginning of the transmission of the message. This period depends on the latency of the network: After a while, all the stations will detect that the channel is busy, and they will not try to send a new message until the channel is idle again.

In the specification of the system we will consider the following main processes: The *Station*, the *Sensor*, the *Channel*, the *Line*, and finally the *CSMA/CD*.

The *Station*. A station can be modeled by a parallel composition of two processes: *GenMsg* is a generator of messages and *StationTrans* transmits the messages created by *GenMsg*. Let us remark that all the generated messages will be stored in the queue of the parallel operator. Thus, we do not need a process to store the messages. In fact, the *Station* will generate messages independently of the possibility to transmit them.

We consider that the time elapsed between the creation of two different messages follows a Poisson distribution with parameter λ:

$$GenMsg_i := \xi_{Po(\lambda)} \,;\, genMsg_i!(msg) \,;\, GenMsg_i$$

The translation to Eden is as follows:

```
genMsg i = process () -> msgs
  where msgs = f lambda
        f lambda = seq (wait (poisson lambda)) (generateMsg i : f lambda)
```

We split the definition of the process *StationTrans* into four stages. Once a message is received from the generator, the process *StationTrans* asks the sensor whether the bus is idle or not. Then, it waits in *Sense* state until it receives an answer. If the channel was busy then it has to back off. Otherwise, and after a fix processing delay (given by a Dirac distribution), it notifies that it is starting a new transmission, entering in stage *Prop*. The transmission take place in this stage, but it is not sure that there will not be collisions until a *tranMsg* is received. This message notifies that the *dangerous* period of time has expired. In that case, we notify that the message will be successfully sent. After an amount of time given by a normal distribution the transmission of the message will finish.

Let us remark that a collision signal can be received at any stage of the process. Thus, we need to allow them to receive a collision signal. When a collision happens, the station waits an amount of time given by an exponential distribution. Then, it asks again the sensor whether the bus is idle or not. The only exception is the first stage: As it has not yet tried to start a transmission, the collision signal is just ignored.

$$\begin{array}{lll}
StationTrans_i & := & genMsg_i?(x) \,;\, sensorReady_i!(-) \,;\, StationSense_i(x) \\
 & & +\ signalColl_i?(-) \,;\, StationTrans_i \\
StationSense_i(msg) & := & sense_i?(x) \,;\, \texttt{if}\ x = idle \\
 & & \qquad\qquad \texttt{then}\ \xi_{D(a)} \,;\, startTransMsg_i!(msg) \,;\, StationProp_i(msg) \\
 & & \qquad\qquad \texttt{else}\ StationBack_i(msg) \\
 & & +\ signalColl_i?(-) \,;\, StationBack_i(msg) \\
StationProp_i(msg) & := & tranMsg_i?(-) \,;\, send!(msg) \,;\, \xi_{N(b,c)} \,;\, endTransMsg_i!(-) \,;\, StationTrans_i \\
 & & +\ signalColl_i?(-) \,;\, StationBack_i(msg) \\
StationBack_i(msg) & := & \xi_{exp(\mu)} \,;\, sensorReady_i!(-) \,;\, StationSense_i(msg) \\
 & & +\ signalColl_i?(-) \,;\, StationBack_i(msg)
\end{array}$$

The translation into Eden also takes into account the four stages of the definition. Following the specification, the translation of each of the stages starts with a choice between the two alternatives, using an auxiliary function to implement what have to be done in each of the cases:

```
stationTrans i = process (msgs,senses,colls,trans) -> (sends,readys,starts,ends)
  where (sends,readys,starts,ends) = f Trans msgs senses colls trans
        f Trans xs ys zs ws = ftrans (merge2 xs zs) xs ys zs ws
          where ftrans (Left msg : vs) (x:xs) ys zs ws = (ys1,Ready:ys2,ys3,ys4)
                  where (ys1,ys2,ys3,ys4) = f (Sense msg) xs ys zs ws
                ftrans (Right v : vs) xs ys (z:zs) ws  = f Trans xs ys zs ws
        f (Sense msg) xs ys zs ws = fSense (merge2 ys zs) xs ys zs ws
          where fSense (Left Idle : vs) xs (y:ys) zs ws
                     = seq (wait (dirac a)) (ys1,ys2,Start i:ys3,ys4)
                  where (ys1,ys2,ys3,ys4) = f (Prop msg) xs ys zs ws
                fSense (Left Busy : vs) xs (y:ys) zs ws = f (Back msg) xs ys zs ws
                fSense (Right v : vs)   xs ys (z:zs) ws = f (Back msg) xs ys zs ws
        f (Prop msg) xs ys zs ws = fProp (merge2 ws zs) xs ys zs ws
          where fProp (Left v : vs) xs ys zs (w:ws) = (msg:ys1,ys2,ys3,ys4)
                  where (ys1,ys2,ys3,ys4) = seq (wait (normal b c))
```

```
                                        (zs1,zs2,zs3,End:zs4)
                       (zs1,zs2,zs3,zs4) = f Trans xs ys zs ws
               fProp (Right v:vs) xs ys (z:zs) ws = f (Back msg) xs ys zs ws
   f (Back msg) xs ys zs ws = fBack (merge2 [wait (expo mu)] zs) xs ys zs ws
     where fBack (Left v : vs) xs ys zs ws = (ys1,Ready:ys2,ys3,ys4)
             where (ys1,ys2,ys3,ys4) = f (Sense msg) xs ys zs ws
           fBack (Right v: vs) xs ys (z:zs) ws = f (Back msg) xs ys zs ws
```

A *Station* is just a composition of a message generator, that is the process *GenMsg*, and a process to transmit the messages, that is *StationTrans*.

$$Station_i := \quad (GenMsg_i \parallel_{\emptyset} StationTrans_i)/\{genMsg_i\}$$

The translation into Eden connects the output of `genMsg` to the appropriate input of `stationTrans`:

```
station i = process (senses,colls,trans) -> (sends,readys,starts,ends)
  where msgs = (genMsg i) # ()
        (sends,readys,starts,ends) = (stationTrans i) # (msgs,senses,colls,trans)
```

The *Sensor*. Each station is provided with a *Sensor* to detect whether the bus is being used or not. Thus, the process will be parametrized by the state of the bus, being initially idle. A sensor can receive either a *sensorReady* request, or a *set* one. In the first case, the current state of the bus (v) will be output after a fix delay given by a Dirac distribution. In the second case, the state of the sensor will be changed after another fix delay.

$$\begin{array}{lll} Sensor_i & := & SensorS_i(Idle) \\ SensorS_i(v) & := & sensorReady_i?(-)\,;\xi_{D(d)}\,;sense_i!(v)\,;SensorS_i(v) + \; set_i?(x)\,;\xi_{D(e)}\,;SensorS_i(x) \end{array}$$

The Eden translation is as follows:

```
sensor = process (readys,sets) -> senses
  where senses = f Idle (merge2 readys sets) readys sets
        f state (Left x :xs) (y:ys) zs = seq (wait (dirac d)) (state : f state xs ys zs)
        f state (Right x:xs) ys (newState:zs) = seq (wait (dirac e)) (f newState xs ys zs)
```

The *Line*. The *Line* process can communicate with any station of the system. When a line is not being used (*Empty* stage), it can receive a *startTransMsg* from any station. Then, after a delay given by a Dirac distribution, it will notify all the sensors that the bus is currently busy. After that, the transmission stage starts, where two alternative possibilities appear depending on whether a new *startTransMsg* arrives or not: If it does not arrive before a delay given by a Dirac distribution then the transmission will be successful; otherwise, the line will enter collision mode after a fix delay. In case the line enters the *Success* stage, it notifies it to the appropriate station, it *Waits* until the transmission ends, and finally it sets the sensors to idle mode after a fix delay. In case the line enters the *Collision* stage, the collision signal is sent, and the sensors are set to idle mode.

$$\begin{array}{lll} Line & := & LineEmpty \\ LineEmpty & := & \sum_{i=1}^{n} startTransMsg_i?(-)\,;\xi_{D(g)}\,;set_1!(busy)\,;\cdots;set_n!(busy)\,;LineTrans_i \\ LineTrans_i & := & \xi_{D(h)}\,;LineSuccess_i + \sum_{j=1}^{n} startTransMsg_j?(-)\,;\xi_{D(g)}\,;LineCollision_i \\ LineSuccess_i & := & transMsg_i!(-)\,;LineWait_i \\ LineWait_i & := & endTransMsg_i?(-)\,;\xi_{D(g)}\,;set_1!(idle)\,;\cdots;set_n!(idle)\,;Line \\ LineCollision_i & := & signalColl_1!(-)\,;\cdots;signalColl_n!(-)\,;set_1!(idle)\,;\cdots;set_n!(idle)\,;Line \end{array}$$

Let us remark that a line can communicate with n different $Sensor_i$ processes and also with n different $Station_i$ processes. In the case of the sensors, it always behaves in the same way with them. Thus, we can take advantage of Eden multicasting facilities: It will only be necessary to define a channel to communicate with the sensors and the messages will be automatically forwarded to all all them. In the case of the stations, the same optimization can be used for the collision signals but not for the other messages. Thus, the translation uses the lists of channels `startss`, `endss`, and `transs`. As `startss` and `endss` are lists of input channels, the translation begins by converting them into single channels by using `merge`, so that they can be handled during the rest of the transformation:

```
line = process (startss,endss) -> (sets,colls,transs)
  where (sets,colls,transs) = f Empty (merge startss) (merge endss)
        f Empty (Start i:is) es = seq (wait (dirac g))
                                      (Busy:ys1,ys2,yss3)
            where (ys1,ys2,yss3) = f (Trans i) is es
        f (Trans i) is es = fTrans (merge2 [wait (dirac h)] is) is es
            where fTrans (Left _ : _) is es = f (Success i) is es
                  fTrans (Right _: _) (i2:is) es = seq (wait (dirac g))
                                                       f (Collision i) is es
        f (Success i) is es = (ys1,ys2, insert i Trans yss3)
            where (ys1,ys2,yss3) = f (Wait i) is es
        f (Wait i) is (e:es) = seq (wait (dirac g))
                                    (Idle:ys1,ys2,yss3)
            where (ys1,ys2,yss3) = f Empty is es
        f (Collision i) is es = (Idle:ys1,Coll:ys2,yss3)
            where (ys1,ys2,yss3) = f Empty is es
```

The *Channel*. A *Channel* is a parallel composition of a line and n sensors.

$$Channel \;:=\; ((Sensor_1 \parallel_\emptyset \ldots \parallel_\emptyset Sensor_n) \parallel_\emptyset Line)/\{set_i \mid 1 \leq i \leq n\}$$

The translation to Eden only needs to properly connect the channels corresponding to the sensors and the line. Let us note that all the `sets` are automatically forwarded to all the sensors, while each of the sensors needs to receive its `readys`, and not those corresponding to the rest of sensors:

```
channel = process (readyss,startss,endss) -> (colls,transs,sensess)
  where sensess = [(sensor i) # (readyss!!i,sets) | i <- [0..n-1]]
        (sets,colls,transs) = line # (startss,endss)
```

The *CSMA/CD*. The whole system is a parallel composition of n stations and a channel.

$$CSMA/CD \;:=\; ((Station_1 \parallel_\emptyset \ldots \parallel_\emptyset Station_n) \parallel_\emptyset Channel)/A$$

where $A = (\{signalColl\} \cup \{startTransMsg_i, transMsg_i, endTransMsg_i, sensorReady_i, sense_i | 1 \leq i \leq n\})$ Finally, the translation of the main process creates n stations and one channel. In addition, it adequately connects their channels. The function `unzip4` is used to transform a list of tuples into a tuple of lists. It is necessary to use it when a list of processes is created and each of them produces more than one output:

```
csma_cd = process () -> sendss
  where (sendss,readyss,startss,endss)
            = unzip4 [(station i) # (sensess!!i,colls,transs!!i) | i <- [0..n-1]]
        (colls,transs,sensess) = channel # (readyss,startss,endss)
```

5.3. Study of Performance Properties

In this section we show, by means of the previous examples, some of the facilities provided by Paradise. Firstly, it can help us to understand the behavior of a specification with graphs like the ones presented in Figure 4. In this setting, the y axis represents the number of processes inside the system, while the x axis represents the evolution in time. Two different colors are used for each type of process (in our case, there are two types: the gyros and the base). The first color is used to represent how many processes are currently working, while the second one is used to represent how many of them are currently blocked (i.e. waiting for synchronization). In the first graph, it can be seen that initially there are six gyros working. We also have the process representing the base, which is blocked until a repair mission needs to be done. After some time, one of the gyros breaks down. This can be easily seen because it turns into black in the graph. Afterwards, more gyros get broken, until the moment in which only two are working. In that moment, the base starts the

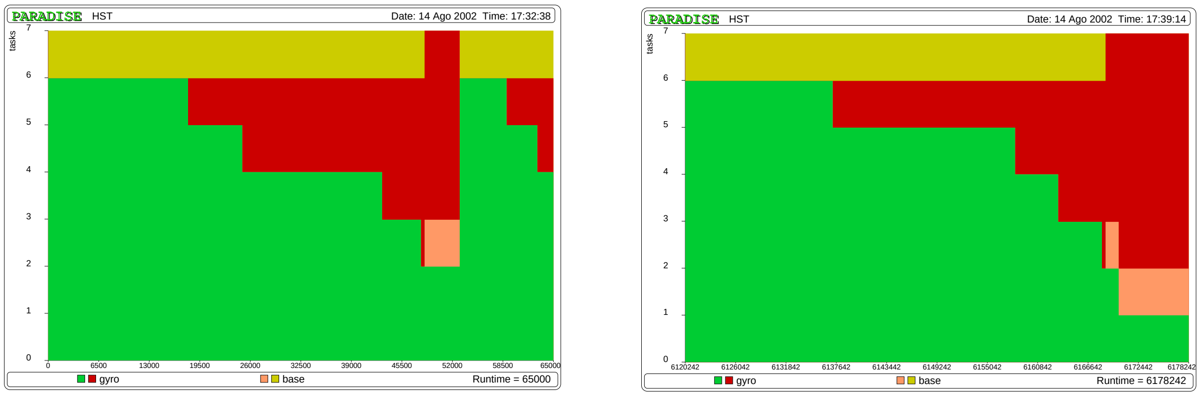

Fig. 4. Behavior of HST: A first mission (left) and the final crash of the system (right)

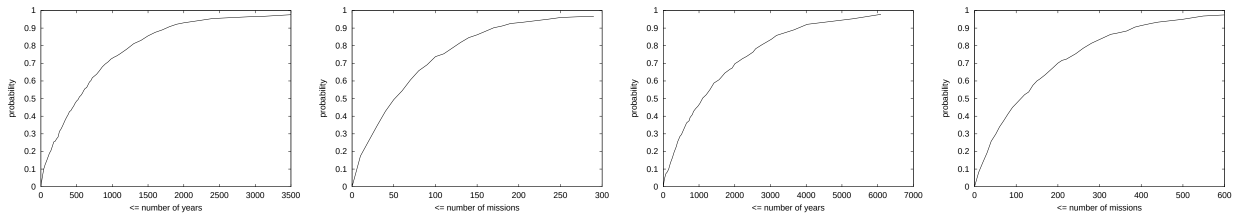

Fig. 5. Number of years and number of missions before crashing

preparation of a repair mission. When this mission is finished, the six gyros are again working and the base gets blocked waiting for a new mission. The second graph shows the crash of the system: After crashing four gyros the base starts a new mission, but the other two gyros fail before the mission is completed. Thus, the whole system crashes. Let us remark that it is possible to zoom into the desired part of the graph, and also to show only the desired processes. For instance, in the shown graphs we have not included the controller and the stabilizer.

Second and more important, we can use Paradise to *measure* quantitative properties of our programs. In the Hubble example, we are interested in knowing the expected operating time of the system, as well as the number of reparations that are going to be needed during the life of the telescope. For the first information, it is only necessary to record the time the program needs to finish. In the second case, we have to explicitly indicate to Paradise that it must count the number of `Restart` messages. Figure 5 shows the probability of having a crash before n years and the probability of needing less than n missions before crashing. The first two graphs assume that the reparation of the system takes one year (on average), while the other two assume that it takes nine months. In both cases we consider that the average time before a gyro fails is ten years and that the controller spends three days to enter sleep mode.

Finally, we have also used Paradise to measure properties of the CSMA/CD specification. In this case, the most interesting result consists in computing the probability of collisions. We have measured it both by varying the number of stations and by varying the frequency at which messages are generated (for a fix number of six stations). The results are shown in Figure 6. As expected, the probability of collision increases as the number of stations increases and also as the frequency of messages increases. Let us remark that our tools allow us not only to compute the tendencies of the graphs, but also to compute the exact values obtained for each of the situations analyzed.

It is worth to point out that, despite the differences between the corresponding specifications, our results are similar (at least the observed tendencies) to the ones obtained in [BCSS98, Her00].

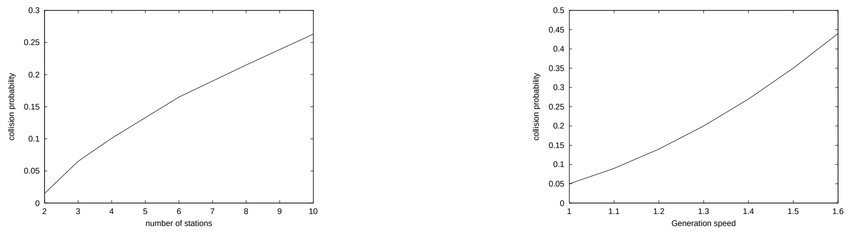

Fig. 6. Probability of collision depending on the number of stations (left) or on the speed generating messages (right).

6. Conclusions

In this paper we have presented an integrated framework to study quantitative properties of stochastic process algebraic specifications. We implement specifications as Eden programs and study properties of the resulting functional programs. We have given a translation algorithm. This algorithm takes as parameter the stochastic process and produces the final Eden program, which is fully runnable. Nevertheless, in some cases optimizations of the code are desirable and they can be done *by hand*. Finally, by means of two examples, we have shown both how are the programs generated by our translator and how quantitative properties can be studied.

The main advantage of our framework is that a thorough knowledge of specifications can be gained by simulating them with their translations. This is specially relevant in the case of performance properties, given the fact that current model checking techniques cannot be used if we consider general distributions.

Acknowledgments

We would like to thank the anonymous referees for their careful reading and valuable comments on the previous version of this paper.

References

[ABL99] R.M. Amadio, G. Boudol, and C. Lhoussaine. The receptive distributed pi-calculus (extended abstract). In *FSTTCS'99, LNCS 1738*, pages 304–315. Springer, 1999.

[AM94] H.R. Andersen and M. Mendler. An asynchronous process algebra with multiple clocks. In *ESOP'94, LNCS 778*, pages 58–73. Springer, 1994.

[BBG98] M. Bravetti, M. Bernardo, and R. Gorrieri. Towards performance evaluation with general distributions in process algebras. In *CONCUR'98, LNCS 1466*, pages 405–422. Springer, 1998.

[BCSS98] M. Bernardo, W.R. Cleaveland, S.I. Sims, and W.J. Stewart. TwoTowers: A tool integrating functional and performance analysis of concurrent systems. In *Formal Description Techniques for Distributed Systems and Communication Protocols (XI), and Protocol Specification, Testing, and Verification (XVIII)*, pages 457–467. Kluwer Academic Publishers, 1998.

[Ber99] M. Bernardo. *Theory and Application of Extended Markovian Process Algebra*. PhD thesis, Università di Bologna, 1999.

[BG98] 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.

[BG02] M. Bravetti and R. Gorrieri. The theory of interactive generalized semi-Markov processes. *Theoretical Computer Science*, 282(1):5–32, 2002.

[BGZ98] N. Busi, R. Gorrieri, and G. Zavattaro. A process algebraic view of linda coordination primitives. *Theoretical Computer Science*, 192(2):167–199, 1998.

[BLOP98] S. Breitinger, R. Loogen, Y. Ortega, and R. Peña. Eden: Language definition and operational semantics. Technical Report, Bericht 96-10, Philipps-Universität Marburg, Germany, 1998. Available at `http://dalila.sip.ucm.es/funcional/publicaciones/REPORT-98.ps`.

[BM02] J.C.M. Baeten and C.A. Middelburg. *Process algebra with timing*. EATCS Monograph. Springer, 2002.

[BPS01] J.A. Bergstra, A. Ponse, and S.A. Smolka, editors. *Handbook of Process Algebra*. North Holland, 2001.

[CDSY99] R. Cleaveland, Z. Dayar, S.A. Smolka, and S. Yuen. Testing preorders for probabilistic processes. *Information and Computation*, 154(2):93–148, 1999.

[CGP99] E.M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[CH90] R. Cleaveland and M. Hennessy. Priorities in process algebras. *Information and Computation*, 87:58–77, 1990.

[CLN01] R. Cleaveland, G. Lüttgen, and V. Natarajan. Priority in process algebra. In J.A. Bergstra, A. Ponse, and S.A. Smolka, editors, *Handbook of process algebra*, chapter 12. North Holland, 2001.

[D'A99] P.R. D'Argenio. *Algebras and Automata for Timed and Stochastic Systems.* PhD thesis, Department of Computer Science. University of Twente, 1999.

[dBKP92] F.S. de Boer, J.W. Klop, and C. Palamidessi. Asynchronous communication in process algebra. In *LICS'92*, pages 137–147. IEEE Computer Society Press, 1992.

[DKB98] P.R. D'Argenio, J.-P. Katoen, and E. Brinksma. General purpose discrete-event simulation using ♠. In *6th International Workshop on Process Algebra and Performance Modelling*, pages 85–102, 1998.

[DS95] J. Davies and S. Schneider. A brief history of timed CSP. *Theoretical Computer Science*, 138:243–271, 1995.

[GH02] H. Garavel and H. Hermanns. On combining functional verification and performance evaluation using CADP. In *FME 2002, LNCS 2391*, pages 410–429. Springer, 2002.

[GHR93] N. Götz, U. Herzog, and M. Rettelbach. Multiprocessor and distributed system design: The integration of functional specification and performance analysis using stochastic process algebras. In *16th Int. Symp. on Computer Performance Modelling, Measurement and Evaluation (PERFORMANCE'93), LNCS 729*, pages 121–146. Springer, 1993.

[GSS95] R. van Glabbeek, S.A. Smolka, and B. Steffen. Reactive, generative and stratified models of probabilistic processes. *Information and Computation*, 121(1):59–80, 1995.

[Han91] H. Hansson. *Time and Probability in Formal Design of Distributed Systems.* PhD thesis, Department of Computer Systems. Uppsala University, 1991.

[Her98] H. Hermanns. *Interactive Markov Chains.* PhD thesis, Universität Erlangen-Nürnberg, 1998.

[Her00] H. Hermanns. Performance and reliability model checking and model construction. In *5th Int. Workshop on Formal Methods for Industrial Critical Systems*, 2000.

[Hil96] J. Hillston. *A Compositional Approach to Performance Modelling.* Cambridge University Press, 1996.

[HKMS00] H. Hermanns, J.-P. Katoen, J. Meyer-Kayser, and M. Siegle. Towards model checking stochastic process algebra. In *IFM 2000, LNCS 1945*, pages 420–440. Springer, 2000.

[HPR00] F. Hernández, R. Peña, and F. Rubio. From GranSim to Paradise. In *Trends in Functional Programming*, pages 11–19. Intellect, 2000.

[HS00] P.G. Harrison and B. Strulo. SPADES – a process algebra for discrete event simulation. *Journal of Logic Computation*, 10(1):3–42, 2000.

[IHK01] G.G. Infante López, H. Hermanns, and J.-P. Katoen. Beyond memoryless distributions: Model checking semi-Markov chains. In *PAPM-PROBMIV 2001, LNCS 2165*, pages 57–70. Springer, 2001.

[JYL01] B. Jonsson, W. Yi, and K.G. Larsen. Probabilistic extensions of process algebras. In J.A. Bergstra, A. Ponse, and S.A. Smolka, editors, *Handbook of process algebra*, chapter 11. North Holland, 2001.

[KLPR01] U. Klusik, R. Loogen, S. Priebe, and F. Rubio. Implementation skeletons in Eden: Low-effort parallel programming. In *12th Implementation of Functional Languages (IFL'00), LNCS 2011*, pages 71–88. Springer, 2001.

[LN01] N. López and M. Núñez. A testing theory for generally distributed stochastic processes. In *CONCUR 2001, LNCS 2154*, pages 321–335. Springer, 2001.

[LNR02] N. López, M. Núñez, and F. Rubio. Stochastic process algebras meet Eden. In *Integrated Formal Methods 2002, LNCS 2335*, pages 29–48. Springer, 2002.

[Loi96] H.W. Loidl. *GranSim User's Guide.* Department of Computing Science, Glasgow University, 1996.

[MdM88] J.A. Mañas and T. de Miguel. From LOTOS to C. In *FORTE'88*, pages 79–84. North-Holland, 1988.

[NdF95] M. Núñez and D. de Frutos. Testing semantics for probabilistic LOTOS. In *Formal Description Techniques VIII*, pages 365–380. Chapman & Hall, 1995.

[NdFL95] M. Núñez, D. de Frutos, and L. Llana. Acceptance trees for probabilistic processes. In *CONCUR'95, LNCS 962*, pages 249–263. Springer, 1995.

[NS91] X. Nicollin and J. Sifakis. An overview and synthesis on timed process algebras. In *Computer Aided Verification'91, LNCS 575*, pages 376–398, 1991.

[Núñ03] M. Núñez. Algebraic theory of probabilistic processes. *Journal of Logic and Algebraic Programming*, 56(1–2):117–177, 2003.

[PGF96] S.L. Peyton Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *ACM Symp. on Principles of Prog. Lang. POPL'96*, pages 295–308. ACM Press, 1996.

[PH99] S.L. Peyton Jones and J. Hughes, editors. *Report on the Programming Language Haskell 98.* Available at http://www.haskell.org, 1999.

[PR01] R. Peña and F. Rubio. Parallel functional programming at two levels of abstraction. In *Principles and Practice of Declarative Programming (PPDP01)*, pages 187–198. ACM Press, 2001.

[Rog97] D. Rogerson. *Inside COM: Microsoft's Component Object Model.* Microsoft Press, 1997.

[Rub03] F. Rubio. A framework for nesting arbitrary skeletons. In *LCR02.* Springer, 2003. To appear.

[Sti01] Colin Stirling. *Modal and Temporal Properties of Processes.* Springer, 2001.

[Tan96] A.S. Tanenbaum. *Computer Networks.* Prentice Hall, 1996.