Stochastic Process Algebras Meet Eden*

Natalia López, Manuel Núñez, and Fernando Rubio

Dept. Sistemas Informáticos y Programación
Universidad Complutense de Madrid, E-28040 Madrid, Spain
{natalia,mn,fernando}@sip.ucm.es

Abstract. Process algebras represent an appropriate mechanism to formally specify concurrent systems. In order to get a thorough knowledge of these systems, some *external* formalism must be used. In this paper we propose an integrated framework where a (non-trivial) process algebra is combined with a (concurrent) functional language. Specifically, we consider a stochastic process algebra featuring value passing where distributions are not restricted to be exponential. In order to study properties of these specifications, we *translate* them into functional programs written in Eden. This functional language 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. We present an example showing how specifications can be translated into Eden and how quantitative properties can be studied.

Keywords: Process algebras, functional programming.

1 Introduction

Process algebras (see [2] for a good overview) are a very suitable formalism to specify concurrent systems because they allow to model them in a compositional manner. Process algebras have been extended to allow the specification of systems where not only functional requirements but also performance ones are included. Therefore, there have been several timed and/or probabilistic extensions of process algebras (e.g. [30,17,13,12,31,8,28,15,10,23,1]). An attempt to integrate time and probabilistic information has been given by introducing *stochastic process algebras* (e.g. [14,20,4]) 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* [9] 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. [18]). Unfortunately, this is not the case if probability distributions are *general* (a preliminary step has been given in [22] where semi-Markov

* Research supported in part by the CICYT projects TIC2000-0701-C02-01 and TIC2000-0738, and the Spanish-British Acción Integrada HB 1999-0102.

chains are considered). So, in order to analyze this kind of systems, a different approach should be used. Another possibility, that we will use in this paper, consists in the *simulation* of the specified 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.

In this paper we provide an integrated framework that allows us to study systems containing stochastic information by simulating their behaviors in an appropriate language. In order to provide an expressive specification language we will consider a process algebra, that we call `VPSPA`, featuring value passing as well as the possibility to specify stochastic behaviors, where probability distributions are not restricted to be exponential, together with a notion of parallel composition. Let us note that value passing is not usually included in stochastic process algebras (to the best of our knowledge, [3] represents the only proposal for such a stochastic language). Anyway, the inclusion of value passing does not produce too many additional difficulties. However, in order to eliminate the restriction to exponential distributions we need a complicate semantic model[1].

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 [34] 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 for 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 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. [33]) but usually by means of very *low-level* concurrency primitives. On the contrary, the programming language Eden [7,6] 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 taken place since then (e.g. [32,24]). 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

[1] The combination of a parallel operator and general distributions produce a lot of semantical difficulties. That is why stochastic models are usually based on (semi)-Markov chains.

a set of profiling utilities (called Paradise [19]) that facilitates the analysis of programs, so that it can also be used to study the real behaviour at runtime.

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 *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 necessary to have only 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 [29] among many others). In the stochastic process algebras area, [11] 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. In that implementation, at each point of the execution, it is computed the set of all the possible next states of the whole system. So, it suffers from the inconvenience of state explosion. Moreover, our approach is a real concurrent implementation, where processes really evolve independently. In fact, we run the processes in parallel. Finally, our generated code is a usual program, so that a user with some command in functional languages may optimize it, if necessary, by hand.

The rest of the paper is organized as follows. In Section 2 we present our value passing stochastic process algebra. 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 present our methodology for translating from specifications to Eden programs. Finally, in Section 5 we introduce an example. We present its specification, the translation into Eden, and we study some quantitative properties.

2 A Stochastic Process Algebra with Value Passing

In this section we present our language and its semantics. Our model is based on [26] 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 in-

terested in. Moreover, despite of the complexity of the operational definition, the behavior of this operator is the *expected* in usual process algebras. Every process will be able to perform either actions for transmitting a message (output actions) or for receiving them (input actions), or stochastic actions (delays). These delays will be represented by random variables. 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 $t < 0$. The reason for this restriction is that random variables are always associated with time distributions.

Definition 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. □

Regarding *communication* actions, they can be divided into *output* and *input actions*. Next, we define our alphabet of actions.

Definition 2. We consider a set of *communication* actions $\mathtt{Act} = \mathtt{Input} \cup \mathtt{Output}$, such that $\mathtt{Input} \cap \mathtt{Output} = \emptyset$. We suppose that there exists a bijection $f : \mathtt{Input} \longrightarrow \mathtt{Output}$. For any *input* action $a? \in \mathtt{Input} \subseteq \mathtt{Act}$, $f(a?)$ is denoted by the *output* action $a! \in \mathtt{Output} \subseteq \mathtt{Act}$. 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 $\mathcal{C}_{\mathtt{Act}}$). 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 $\mathcal{X}$) [2]. 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, $IO = \{c(x) \mid c \in \mathtt{Input}, x \in \mathcal{X}\} \cup \{c(v) \mid c \in \mathtt{Output}, v \in \mathit{Val}\}$. We define the channel of a communication action $c(m) \in IO$, denoted by $ch(c(m))$, as a if $c \in \{a?, a!\}$. □

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* $\mathrm{Loc} = \{loc_i \mid loc \in \{l, r\} \ \wedge \ i \in \mathbb{N}^+\}$ where l stands for left and r for right.

Definition 3. The set of processes, denoted by $\mathtt{VPSPA}$, is given by the following set of expressions:

$$P ::= \mathrm{stop} | \xi; P | a?(x); P | a!(v); P | P + P | \mathtt{if}\ e\ \mathtt{then}\ P\ \mathtt{else}\ P | P \|_M P | P/A | X := P$$

where $X \in Id$ (the set of *process variables*), $\xi \in \mathcal{V}$, $a?, a! \in \mathtt{Act}$, $x \in \mathcal{X}$, $v \in \mathit{Val}$, $A \subseteq \mathcal{C}_{\mathtt{Act}}$, e is an expression such that $\mathtt{Eval}(e) \in \mathit{Bool}$ ($\mathtt{Eval}(e)$ represents the evaluation of e), and $M \subseteq (\mathcal{V} \times \mathbb{N}^+ \times \mathrm{Loc})$. □

[2] Sometimes we are not interested in transmitting a message because that communication is only a synchronization point. In this case, we *send* the value $-$.

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, this means that both delays are given by two independent random variables identically distributed. The term stop denotes a process that cannot execute any action. A process $\xi\,;P$ waits a random amount of time (determined by the distribution function associated with ξ) and then it behaves as P. 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 [5], this is one of the mechanisms to deal with general distributions in the context of parallel. The process $a?(x)\,;P$ waits until it receives a value v on the input action $a?$, and after that P behaves as $P[v/x]$, where $P[v/x]$ denotes the substitution of all the free occurrences of x in P by v. Let us note that x is bound in $a?(x)\,;P$. The process $a!(v)\,;P$ transmits the value v on the output action $a!$ and after that it behaves like P. The process $P+Q$ behaves either like P or like Q. The process $\texttt{if}\ e\ \texttt{then}\ P\ \texttt{else}\ Q$ 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 asynchronously. Besides, if one of the processes is prepared to perform an input action and the other one an output action on the same channel, both actions can be performed synchronously, and some values can be exchanged from one process to the other. In order to avoid *undesirable* effects with some recursive processes, we assign indexes to delays. Specifically, problems may appear if we have several *starts* of the same delay in the context of a parallel operator. This is the case, for instance, in the process $\text{rec}\ X.(\xi\parallel_M a;X)$. Thus, for each start stochastic action ξ_i^+ there will be a unique termination action ξ_i^- associated with it. That information will be stored in M. Initially, we suppose $M=\emptyset$. Let us remark that, given the fact that we consider random variables (instead of probability distribution functions), indexes would not be needed if we restrict occurrences of the parallel computation in the scope of the recursive definitions. So, the parameter M could be removed. The process P/A expresses that P is restricted to perform only the communication actions that are not in A. Finally, $X:=P$ denotes the definition of a (possible recursive) process.

The operational semantics for the language (presented in Figure 1) is inspired in [5]. We will use the following conventions: $P\xrightarrow{\omega}P'$ expresses that there exists a transition from P to P' labeled by the action ω; $P\xrightarrow{\omega}$ stands for there exists $P'\in\texttt{VPSPA}$ such that $P\xrightarrow{\omega}P'$; and we write $P\not\xrightarrow{\omega}$ if there does not exist $P'\in\texttt{VPSPA}$ such that $P\xrightarrow{\omega}P'$. We consider that a special action τ denotes internal activity, that is, an internal communication between processes. Let us note that if a process is able to perform an internal action, then no stochastic transition will be allowed. This property is called *urgency* and it is a standard mechanism in stochastic process algebras. For that purpose the following predicate is defined to express the stability of a process. Intuitively, a process is *stable* if it cannot perform any internal communication.

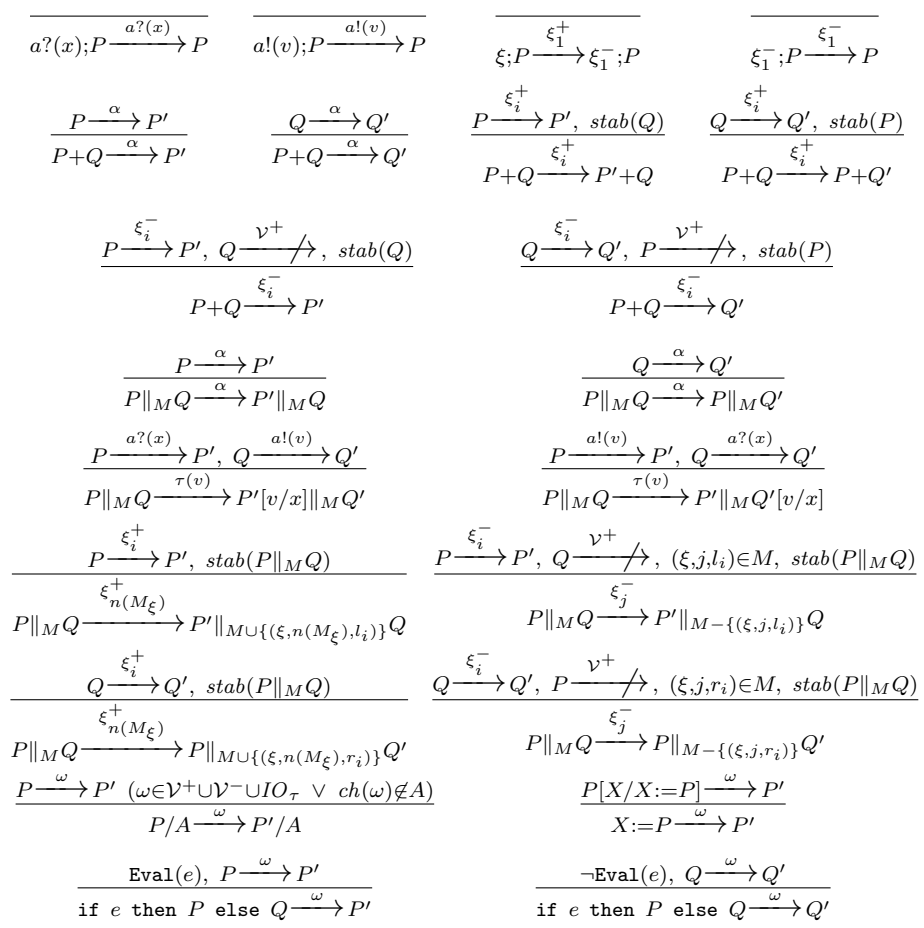

Fig. 1. Operational Semantics of VPSPA

Definition 4. Let P be a process. We define the *stability* of P, denoted by $stab(P)$, by structural induction as:

$$
\begin{aligned}
&stab(\text{stop}) = stab(\xi\,;P) = stab(a?(x)\,;P) = stab(a!(v)\,;P) = \texttt{true}\\
&stab(P+Q) = stab(\texttt{if } e \texttt{ then } P \texttt{ else } Q) = stab(P) \ \wedge \ stab(Q)\\
&stab(P \parallel_M Q) = stab(P) \wedge stab(Q)\\
&\qquad\qquad \wedge \nexists\, a \in \texttt{Act}\colon (P \xrightarrow{a?(x)} \wedge\, Q \xrightarrow{a!(v)}) \vee (P \xrightarrow{a!(x)} \wedge\, Q \xrightarrow{a?(v)})\\
&stab(P/A) = stab(P)\\
&stab(X := P) = stab(P[X/X := P])
\end{aligned}
$$

□

In the definition of the operational semantics we have extended the set of communications with a set of internal communications $IO_\tau = \{\tau(v) \mid v \in Val\}$, as well as sets of starting actions $\mathcal{V}^+ = \{\xi_i^+ \mid \xi \in \mathcal{V} \ \wedge \ i \in \mathbb{N}^+\}$ and termination

actions $\mathcal{V}^- = \{\xi_i^- \mid \xi \in \mathcal{V} \ \wedge \ i \in \mathbb{N}^+\}$. In that definition, we consider $a \in \mathcal{C}_{\mathtt{Act}}$, $\xi \in \mathcal{V}$, $\alpha \in IO \cup IO_\tau$, $\omega \in IO \cup IO_\tau \cup \mathcal{V}^+ \cup \mathcal{V}^-$, and $A \subseteq \mathcal{C}_{\mathtt{Act}}$. The first two rules of the operational semantics are the usual ones for the prefix operator for standard actions. The following two rules deal with stochastic transitions. 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. We consider that the start of a delay has higher priority than any termination action. So, a stochastic action cannot be *completed* if any stochastic action can be started. In order to index random variables we use dynamic names, that is, we assign to each random variable an index. This index will be the minimum natural number that it is not being currently used to label other occurrences of the same random variable. Once a stochastic transition is finished, its corresponding index is liberated. The following six rules represent the behaviour of the choice between processes. If one of the components of the choice can perform an action, then the choice can also perform it. If the action is stochastic, it has to be taken into account that the other process cannot evolve internally. Besides, if the stochastic action is a termination action, it is necessary to assure that the other process can perform neither a start action nor an internal action. The behaviour of the parallel operator is defined by the following eight rules. The first four rules represent the performance of communication actions. We briefly sketch how stochastic transitions are treated in the context of a parallel composition (the interested reader is pointed to [5] for additional details). If any of the processes of the composition can perform a start action and they cannot perform a communication, the composition can also perform it. However, we need to associate to that action a different index. It will be the minimum natural number not already used for this random variable in the scope of the corresponding parallel composition. This index is given by the following function:

Definition 5. 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

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

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

So, 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^- with an index i can be performed by P and neither a communication action nor a start action can be performed, and there exists $(\xi, j, l_i) \in M$, then the parallel composition is able to perform the same termination action but indexed by j. The following two rules are the usual ones for the restriction operator and for recursive processes. The last two rules express the behaviour of the conditional operator. Moreover, even though our operational semantics has rules with negative premises, this does not cause any problem because a stratification in terms of [16] can be easily given, so the uniqueness of (stochastic) labeled transition systems is guaranteed.

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

Definition 6. We inductively define the set of *input actions* that a process P can perform, denoted by $\texttt{Inputs}(P)$, as:

$$\texttt{Inputs}(\text{stop}) = \emptyset \qquad \texttt{Inputs}(P + Q) = \texttt{Inputs}(P) \cup \texttt{Inputs}(Q)$$
$$\texttt{Inputs}(\xi\,;P) = \texttt{Inputs}(P) \qquad \texttt{Inputs}(P \parallel_M Q) = \texttt{Inputs}(P) \cup \texttt{Inputs}(Q)$$
$$\texttt{Inputs}(\tau\,;P) = \texttt{Inputs}(P) \qquad \texttt{Inputs}(P/A) = \texttt{Inputs}(P) - A$$
$$\texttt{Inputs}(a?(x)\,;P) = \texttt{Inputs}(P) \cup \{a\} \qquad \texttt{Inputs}(X := P) = \texttt{Inputs}(P)$$
$$\texttt{Inputs}(a!(v)\,;P) = \texttt{Inputs}(P) \qquad \texttt{Inputs}(\texttt{if } e \texttt{ then } P \texttt{ else } Q) = \texttt{Inputs}(P) \cup \texttt{Inputs}(Q)$$

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

$$\texttt{Outputs}(\text{stop}) = \emptyset \qquad \texttt{Outputs}(P + Q) = \texttt{Outputs}(P) \cup \texttt{Outputs}(Q)$$
$$\texttt{Outputs}(\xi\,;P) = \texttt{Outputs}(P) \qquad \texttt{Outputs}(P \parallel_M Q) = \texttt{Outputs}(P) \cup \texttt{Outputs}(Q)$$
$$\texttt{Outputs}(\tau\,;P) = \texttt{Outputs}(P) \qquad \texttt{Outputs}(P/A) = \texttt{Outputs}(P) - A$$
$$\texttt{Outputs}(a?(x)\,;P) = \texttt{Outputs}(P) \qquad \texttt{Outputs}(X := P) = \texttt{Outputs}(P)$$
$$\texttt{Outputs}(a!(v)\,;P) = \texttt{Outputs}(P) \cup \{a\} \qquad \texttt{Outputs}(\texttt{if } e \texttt{ then } P \texttt{ else } Q) = \texttt{Outputs}(P) \cup \texttt{Outputs}(Q)$$

The set of *immediate output actions* P can perform, denoted by $\texttt{Imm\_Outputs}(P)$, is given by $\texttt{Imm\_Outputs}(P) = \{a \in \mathcal{C}_{\texttt{Act}} \mid \exists P' \in \texttt{VPSPA}, v \in Val : P \xrightarrow{a!(v)} P'\}$. □

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 [7].

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 x : map f 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 synchronize on the first input channel until it receives a value. 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 synchronizations:

```
... 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. Notice that Eden does not include *send* or *receive* constructors: Communication and synchronization 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 the input of the `timer` process is a list of *start* messages, and the output is also a list. 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 [19], an Eden profiler based on GranSim [25]. 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, and to obtain accurate statistics without performing complex theoretical studies.

4 From VPSPA to Eden

In this section we present how process algebraic specifications are translated into Eden programs. First, we need to define what a correct implementation is. We will consider that an Eden program implements a VPSPA specification if any (input, output, or internal) transition of the specification can be *simulated* by a channel of the Eden program, and any delay of the specification is reflected in that program. The formal definition is a little bit involved and can be found in [27]. The rest of this section is devoted to present our methodology for the translation of VPSPA specifications into Eden programs. Even though the translation can be 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 VPSPA 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 VPSPA 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 o11 ... o1r1
```

```
F2 = process (i21,...,i2s2) -> (o21,...,o2r2)
 where (o21,...,o2r2) = f2 i21 ... i2s2 o21 ... o2r1
 .....
Fn = process (in1,...,insn) -> (on1,...,onrn)
 where (on1,...,onrn) = fn in1 ... insn on1 ... onrn
```

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 $P_j = \text{stop}$ 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 ... is = translation(P)
```

where `ij` is the input channel corresponding to $a?$. In this case, we define 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 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 i2...is = translation(P1)
       g n 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 predefined 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)` and `wait(poisson lambda)` for Dirac and Poisson 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 different numbers of parameters.

Analogously to the `if then else`, the choice operator can also be translated by using pattern matching, the difference being 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)
f (Left alt1 : zs)  = ...
f (Right alt2 : zs) = ...
```

If P_j is the parallel composition of m process variables, that is, a 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 $1 \leq j \leq m$ we have that `Gj` is the Eden process abstraction corresponding to Y_j, $\{\texttt{inj1}, ..., \texttt{injsj}\}$ is the set of input channels of `Gj`, and $\{\texttt{outj1}, ..., \texttt{outjrj}\}$ is its set of output channels. 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 $\{\texttt{o1}, ..., \texttt{or}\}$).

It is rather easy to implement the restriction 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 `Gis` are hidden. Thus, if the set of actions `A` is to be restricted, 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 2. Consider the process P given in Example 3, where $P_1 = c?(y)\,;b!(1)$ and $P_2 = d!(0)$, and let $Q := a!(1)\,;\mathrm{stop}$. Finally, 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 # ()
```

□

5 An Example: The Token Ring

In this section we present a medium-size example. This relatively complex specification shows 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 the example will be the following. First, we introduce the problem. Then we indicate the processes that we will use. Afterwards, for each specification we give the corresponding translation. Finally, we study some quantitative properties.

A token ring [21] is a network in which each station is only connected with two other stations. A station receives messages from the previous one and sends messages to the following. The last one sends messages to the first one, producing a ring structure. Each station can send its own generated messages but it can also forward the ones it receives.

A token circulates around the ring whenever all stations are idle. When a station needs to transmit a new message, it is required to seize the token and remove it from the ring before transmitting. As there is only one token, only one station can transmit at a given instant, solving the channel access problem.

Each station has two operating modes: *listen* and *transmit*. If the station is in listening mode, the input messages are simply forwarded to the output (after a short delay), unless the station is the actual receiver of the message. In that case, it just reads it without forwarding it. In the transmitting mode, which is only entered after the token is owned, the station is able to send its own generated messages. In this case, a queue is used to keep track of all the messages it needs to send. In order to guarantee fairness, a timer is used to control the maximum time

a station can own the token. So, the station owning the token enters listening mode either if it has no more messages to transmit or if it receives a timeout from the timer. More specifically, after receiving a timeout the station is allowed to send a last message.

When traffic is light, the token circulates around the ring until a station seizes it, transmits a message, and releases the token again. When the traffic is heavy, as soon as a station releases the token, the following station seizes it and transmits new messages. Therefore, the network efficiency can approach 100%.

The token ring has been widely studied in the context of process algebras. Regarding stochastic process algebras, a study, similar to ours but restricted to exponential distributions, is given in [3].

The main processes of this system are: The $MsgGen$, the $Timer$, the $Queue$, the $MsgTrans$, the $Station$, and the $TokenRing$. In this example, we will consider that the addition operator fulfills the following condition: Given n, we have $i +_n 1 = i + 1$ for any $1 \leq i \leq n - 1$, and $n +_n 1 = 1$. For the sake of clarity, we use $+$ to denote this addition module n instead of $+_n$.

The $MsgGen$. This process generates messages. The generation of messages follows a Poisson distribution with parameter λ. After generating the message, it sends it to a queue, and then it is able to generate another message.

$$MsgGen_i := \xi_{po(\lambda)} \,;\, generatedMsg_i!(msg) \,;\, enqueue_i!(msg) \,;\, MsgGen_i$$

To properly send messages, the generator needs to receive its identity and the size of the ring. Hence the two integer parameters appearing in the translation:

```
msgGen i n = process () -> enqueues    where
  enqueues    = f i n lambda
  f i n lambda= seq (wait(poisson lambda))(generateMsg i n :f i n lambda)
```

The $Timer$. It controls the time that a station owns the token. It will start its behavior when it receives a starting message. After an amount of time, given by a Dirac distribution, a timeout is generated, unless an interrupting message from the $MsgTrans$ ($intTimer$) is received.

$$Timer_i := startTimer_i?(y) \,;(intTimer_i?(y') \,;\, Timer_i + \xi_{D(c)} ; timeOut_i!(-) \,;\, Timer_i)$$

In the corresponding Eden process, we just need to merge the timeout messages and the stops. After that, if a timeout is produced, it is communicated to the $MsgTrans$. In case the timer is interrupted, we wait again until a new start message arrives.

```
timer = process (startsTimer,intsTimer) -> timeOuts
  where timeOuts = f startsTimer intsTimer
        f (x:xs) ys = g (merge2 ys [wait (dirac c)]) xs ys
        g (Left _ : _) xs (y:ys) = f xs ys
        g (Right _ : _)  xs ys   = TimeOut : f xs ys
```

The $Queue$. We use a parameter to store the messages created by the station.

$$
\begin{array}{lll}
Queue_i & := Queue'_i(<>) & \\
Queue'_i(c) & := \texttt{if}\ \ (c = x\#c')\ \texttt{then} & enqueue_i?(y)\,;\, Queue'_i(c\#y) \\
 & & \quad + fetch_i?(y)\,;\, dequeue_i!(x)\,;\, Queue'_i(c') \\
 & \texttt{else} & enqueue_i?(y)\,;\, Queue'_i(c\#y) \\
 & & \quad + fetch_i?(y)\,;\, dequeue_i!(-)\,;\, Queue_i
\end{array}
$$

In Eden, we can reduce the number of cases because queuing a new element can be done regardless of the queue being empty or not. Let us note that we merge both input channels in order to be able to synchronize on them in any order.

```
queue = process (enqueues,fetchs) -> dequeues
  where dequeues = f Empty (merge2 enqueues fetchs)
        f q (Left msg : xs)     = f (addElem msg q) xs
        f Empty (Right fe : xs) = Nothing : f Empty xs
        f q (Right fe : xs)     = Just (first q) : f (extractFirst q) xs
```

The $MsgTrans$. It receives messages from the previous station. Then it checks whether it is the final receiver of the message. If so, it informs about its reception. Otherwise the message can be either the token or a message sent to another station. In the latter case, that message is automatically forwarded to the next station. In the former case, a timer is started, and messages are taken from the queue and sent to the next station. This is iterated until no more messages are available or until a timeout is received from the timer. In the latter case, the transmitter is allowed to send a last message. Notice that there are three possible states: *with* the token, *without* it, and *lastwith*. The last state denotes that it owns the token but that only one more message can be sent before releasing the token.

$$
\begin{array}{lll}
MsgTrans_{i,without} & := transMsg_i?(msg); & \\
 & \quad \texttt{if}\ \ msg = (m,i)\ \texttt{then}\ \ read_i!(msg)\,;\, MsgTrans_{i,without} & \\
 & \quad \texttt{else if}\ \ msg = token\ \texttt{then}\ \ MsgTrans_{i,with} & \\
 & \qquad \texttt{else}\ \ \xi_{exp(\gamma)}\,;\, transMsg_{i+1}!(msg)\,;\, MsgTrans_{i,without} & \\
MsgTrans_{i,with} & := startTimer_i!(-)\,;\, fetch_i!(-)\,;\, MsgTrans_{i,with2} & \\
MsgTrans_{i,with2} & := dequeue_i?(msg)\,;\, Sending_i(msg) & \\
 & \quad + timeOut_i?(x)\,;\, MsgTrans_{i,lastwith} & \\
Sending_i(msg) & := \texttt{if}\ \ msg = -\ \texttt{then}\ \ intTimer_i!(-)\,;\, \xi_{U(a,b)}; & \\
 & \qquad\qquad transMsg_{i+1}!(token)\,;\, MsgTrans_{i,without} & \\
 & \qquad \texttt{else}\ \ \xi_{exp(\gamma)}\,;\, transMsg_{i+1}!(msg); & \\
 & \qquad\qquad fetch_i!(-)\,;\, MsgTrans_{i,with2} & \\
MsgTrans_{i,lastwith} & := dequeue_i?(msg); & \\
 & \quad \texttt{if}\ \ msg = - & \\
 & \quad \texttt{then}\ \ \xi_{U(a,b)}\,;\, transMsg_{i+1}!(token)\,;\, MsgTrans_{i,without} & \\
 & \quad \texttt{else}\ \ \xi_{exp(\gamma)}\,;\, transMsg_{i+1}!(msg)\,;\, \xi_{U(a,b)}; & \\
 & \qquad transMsg_{i+1}!(token)\,;\, MsgTrans_{i,without} &
\end{array}
$$

In the previous specification $exp(\gamma)$ denotes a random variable distributed as an exponential distribution with parameter γ. Besides $U(a,b)$ denotes a random variable distributed as a uniform distribution over the interval (a,b). The corresponding Eden process is shown below. The type `Maybe` is used to either receive

a message `Just message` from the queue, or `Nothing` when the queue is empty. Let us remark that `inms` represents the messages received through $TransMsg_i$, while `outms` are the ones to be sent through $TransMsg_{i+1}$.

```
msgTrans i state =process(inms,dequeues,timeOuts) ->
                          (outms,reads,fetchs,startsTimer,intsTimer) where
  (outms,reads,fetchs,startsTimer,intsTimer)
    = f state inms (merge2 dequeues timeOuts)
  f Without (Token : inms') mts = f With inms' mts
  f Without (M m : inms') mts
   = if (isToMe m i) then (ys1,m:ys2,ys3,ys4,ys5)
                     else seq (wait(expo gamma))(M m:ys1,ys2,ys3,ys4,ys5)
    where (ys1,ys2,ys3,ys4,ys5) = f Without inms' mts
  f With xs ys = (y1,ys2,Fetch:ys3,StartTimer:ys4,ys5)
    where (ys1,ys2,ys3,ys4,ys5) = f With2 xs ys
  f With2 xs (Left Nothing : mts)
    = seq (wait (unif a b))(Token:ys1,ys2,ys3,ys4,IntTimer:ys5)
    where  (ys1,ys2,ys3,ys4,ys5) = f Without xs mts
  f With2 xs (Left (Just m): mts) = seq (wait (expo gamma))
                                        (M m:ys1,ys2,Fetch:ys3,ys4,ys5)
    where (ys1,ys2,ys3,ys4,ys5) = f With2 xs mts
  f With2 xs (Right _ : mts) = (ys1,ys2,ys3,ys4,ys5)
    where  (ys1,ys2,ys3,ys4,ys5) = f LastWith xs mts
  f LastWith xs (Left Nothing : mts)
    = seq (wait (unif a b))(Token:ys1,ys2,ys3,ys4,ys5)
    where  (ys1,ys2,ys3,ys4,ys5) = f Without xs mts
  f LastWith xs (Left (Just m) : mts)
    = seq (wait (expo gamma))
          (M m:(seq (wait (unif a b))(Token:ys1)),ys2,ys3,ys4,ys5)
    where  (ys1,ys2,ys3,ys4,ys5) = f Without xs mts
```

The *Station*. This process is a parallelization of the message generator, the queue to store the messages, the message transmitter, and the timer. The stations have an initial state representing the operating mode of each station at the beginning of the execution. It is either *listen* or *transmit*, depending on the possession of the token. Firstly, $Station_1$ has the token.

$$Station_{i,initstate} := (MsgGen_i \parallel_\emptyset Queue_i \parallel_\emptyset MsgTrans_{i,initstate} \parallel_\emptyset Timer_i)/S_i$$

where $S_i = \{enqueue_i, dequeue_i, fetch_i, startTimer_i, intTimer_i, timeOut_i\}$, and *initstate* is *with* for $Station_1$, and *without* for $Station_2 \ldots Station_n$.

The Eden implementation only needs to instantiate the corresponding processes, establishing the appropriate data dependencies:

```
station i n initState = process inms -> (outms,reads)
  where enqueues = (msgGen i n) # ()
        dequeues = queue # (enqueues,fetchs)
        timeOuts = timer # (startsTimer,intsTimer)
        (outms,reads,fetchs,startsTimer,intsTimer)
           = (msgTrans i initState) # (inms,dequeues,timeOuts)
```

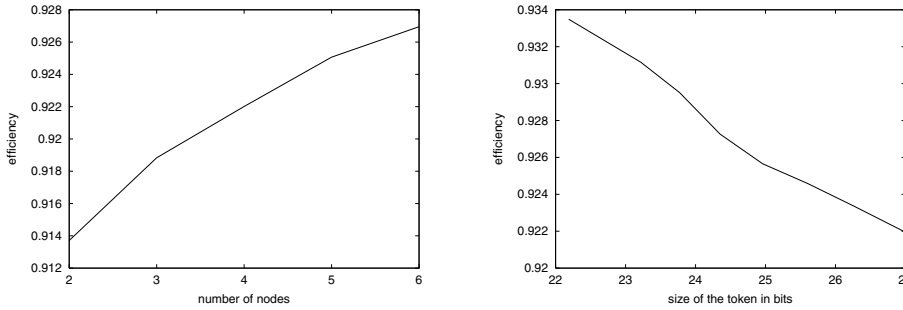

Fig. 2. Network efficiency depending on the number of stations (left) and on the token size (right)

The *TokenRing*. This process represents the whole system. The ring is a parallelization of n stations. Each station is communicated with the neighbor stations by sending messages from one to another through the action $transMsg$.

$$TokenRing := (Station_{1,with} \parallel_\emptyset Station_{2,without} \parallel_\emptyset \cdots \parallel_\emptyset Station_{n,without})/T$$

where $T = \{transMsg_i \mid 1 \leq i \leq n\}$.

The corresponding Eden program only needs to communicate all the stations in a ring fashion, and to initially assign the token to the first station. For doing that, we just need to create the appropriate dependencies. Our transformation rules can generate Eden code for a fix number of stations. For instance, in case we only have three stations, the following code will be used:

```
tokenRing 3 = merge [external1,external2,external3]    where
  (internal1,external1) = (station 1 3 With) # (internal3)
  (internal2,external2) = (station 2 3 Without) # (internal1)
  (internal3,external3) = (station 3 3 Without) # (internal2)
```

But the code can be optimized by hand to deal with a variable number of stations. This can be done by using a list comprehension and the `unzip` function which converts a list of pairs into a pair of lists:

```
tokenRing n = merge (external1:externals)    where
  (internal1,external1) = (station 1 n With) # (internals'!!n)
  (internals,externals)
    = unzip [(station i n Without)) # (internals'!!(i-1)) | i<-[2..n]]
  internals' = internal1 : internals
```

Finally, we use the profiling utilities of Eden to *measure* quantitative properties of our programs. In the Token Ring, we are interested in knowing the *efficiency* in the use of the network. We define efficiency as the time used for transmitting real messages divided by the total time transmitting messages (both messages and tokens). It was enough to profile the tokens and messages transmitted, and the graphics in Figure 2 were generated. The left one shows that the

efficiency increases as the number of stations increases, because more messages are generated and transmitted. The right one shows that the efficiency is better when the token size is smaller. Thus, the token should be as small as possible, but there is a minimum size defined in terms of the size of the ring.

Acknowledgments

The authors thank Ismael Rodríguez and the anonymous referees for valuable comments on a previous version of this paper.

References

1. J.C.M. Baeten and C.A. Middelburg. Process algebra with timing: Real time and discrete time. In J.A. Bergstra, A. Ponse, and S.A. Smolka, editors, *Handbook of process algebra*, chapter 10. North Holland, 2001.
2. J.A. Bergstra, A. Ponse, and S.A. Smolka, editors. *Handbook of Process Algebra*. North Holland, 2001.
3. M. Bernardo. *Theory and application of extended markovian process algebra*. PhD thesis, Università di Bologna, 1999.
4. 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.
5. M. Bravetti and R. Gorrieri. The theory of interactive generalized semi-markov processes. To appear in Theoretical Computer Science., 2001.
6. S. Breitinger, U. Klusik, R. Loogen, Y. Ortega, and R. Peña. DREAM: the distributed Eden abstract machine. In *Implementation of Functional Languages. LNCS 1467.*, pages 250–269. Springer, 1998.
7. 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.
8. J. Bryans, J. Davies, and S. Schneider. Towards a denotational semantics for ET-LOTOS. In *CONCUR'95, LNCS 962*, pages 269–283, 1995.
9. E.M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.
10. R. Cleaveland, Z. Dayar, S.A. Smolka, and S. Yuen. Testing preorders for probabilistic processes. *Information and Computation*, 154(2):93–148, 1999.
11. P.R. D'Argenio. *Algebras and Automata for Timed and Stochastic Systems*. PhD thesis, Department of Computer Science. University of Twente, 1999.
12. J. Davies and S. Schneider. A brief history of timed CSP. *Theoretical Computer Science*, 138:243–271, 1995.
13. 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.
14. 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.

15. C. Gregorio and M. Núñez. Specifying and verifying the Alternating Bit Protocol with Probabilistic-Timed LOTOS. In *COST 247 International Workshop on Applied Formal Methods in System Design*, pages 38–50, 1996.
16. Jan Friso Groote. Transition system specifications with negative premises. *Theoretical Computer Science*, 118:263–299, 1993.
17. H. Hansson. *Time and Probability in Formal Design of Distributed Systems*. PhD thesis, Department of Computer Systems. Uppsala University, 1991.
18. 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.
19. F. Hernández, R. Peña, and F. Rubio. From GranSim to Paradise. In *Trends in Functional Programming*, pages 11–19. Intellect, 2000.
20. J. Hillston. *A Compositional Approach to Performance Modelling*. Cambridge University Press, 1996.
21. IEEE. 802.5: Token ring access method. IEEE, 1985.
22. 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.
23. 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.
24. U. Klusik, R. Loogen, S. Priebe, and F. Rubio. Implementation skeletons in Eden: Low-effort parallel programming. In *Implementation of Functional Languages, IFL'00, LNCS 2011*. Springer, 2001.
25. H.W. Loidl. *GranSim User's Guide*. Department of Computing Science, Glasgow University, 1996.
26. N. López and M. Núñez. A testing theory for generally distributed stochastic processes. In *CONCUR 2001, LNCS 2154*, pages 321–335. Springer, 2001.
27. N. López, M. Núñez, and F. Rubio. Implementation relation between `VPSPA` and Eden, 2002. `http://dalila.sip.ucm.es/~natalia/ifm02/appendix.ps`.
28. G. Lowe. Probabilistic and prioritized models of timed CSP. *Theoretical Computer Science*, 138:315–352, 1995.
29. J.A. Mañas and T. de Miguel. From LOTOS to C. In *International Conference on Formal Description Techniques for Distributed Systems and Communications Protocols*, pages 79–84. Elsevier, 1988.
30. X. Nicollin and J. Sifakis. An overview and synthesis on timed process algebras. In *Computer Aided Verification'91, LNCS 575*, pages 376–398, 1991.
31. M. Núñez, D. de Frutos, and L. Llana. Acceptance trees for probabilistic processes. In *CONCUR'95, LNCS 962*, pages 249–263. Springer, 1995.
32. 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.
33. 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.
34. S.L. Peyton Jones and J. Hughes, editors. *Report on the Programming Language Haskell 98*. Available at http://www.haskell.org, 1999.