First Passage Time Analysis of Stochastic Process Algebra Using Partial Orders

Theo C. Ruys[1], Rom Langerak[1], Joost-Pieter Katoen[1], Diego Latella[2], and Mieke Massink[2]

[1] University of Twente, Department of Computer Science.
PO Box 217, 7500 AE Enschede, The Netherlands.
{ruys,langerak,katoen}@cs.utwente.nl
[2] Consiglio Nazionale delle Ricerche, Istituto CNUCE.
Via Alfieri 1, S. Cataldo, Pisa, Italy.
{d.latella,m.massink}@cnuce.cnr.it

Abstract This paper proposes a partial-order semantics for a stochastic process algebra that supports general (non-memoryless) distributions and combines this with an approach to numerically analyse the first passage time of an event. Based on an adaptation of McMillan's complete finite prefix approach tailored to event structures and process algebra, finite representations are obtained for recursive processes. The behaviour between two events is now captured by a partial order that is mapped on a stochastic task graph, a structure amenable to numerical analysis. Our approach is supported by the (new) tool FOREST for generating the complete prefix and the (existing) tool PEPP for analysing the generated task graph. As a case study, the delay of the first resolution in the root contention phase of the IEEE 1394 serial bus protocol is analysed.

1 Introduction

In the classical view of system design, two main activities are distinguished: performance evaluation and validation of correctness. Performance evaluation studies the performance of the system in terms like access time, waiting time and throughput, whereas validation concentrates on the functional behaviour of the system in terms of e.g. safety and liveness properties. With the advent of embedded and multi-media communication systems, however, insight in both the functional and the real-time and performance aspects of applications involved becomes of critical importance. The seperation of these issues does not make sense anymore.

As a result, performance aspects have been integrated in various specification formalisms. A prominent example is stochastic process algebra in which features like compositionality and abstraction are exploited to facilitate the modular specification of performance models. Most of these formalisms, however, restrict delays to be governed by negative exponential distributions. The interleaving

T. Margaria and W. Yi (Eds.): TACAS 2001, LNCS 2031, pp. 220–235, 2001.

semantics typically results in a mapping onto continuous-time Markov chains (CTMC) [1,13,19], a model for which various efficient evaluation algorithms exist to determine (transient and stationary) state-based measures. Although this approach has brought various interesting results, tools, and case studies, the state space explosion problem – in interleaving semantics parallelism leads to the product of the component state spaces – is a serious drawback. Besides that, the restriction to exponential distributions is often not realistic for adequately modelling phenomena such as traffic sources or sizes of data files stored on web-servers that exhibit bursty heavy-tail distributions.

This paper proposes a partial-order semantics for a stochastic process algebra with general (continuous) distributions and combines this with techniques to compute the mean delay between a pair of events. The semantics is based on event structures [28], a well-studied partial-order model for process algebras. These models are less affected by the state space explosion problem as parallelism leads to the sum of the components state spaces rather than to their product. Moreover, these models are amenable to extensions with stochastic information [4]. A typical problem with event structures though is that recursion leads to infinite structures, whereas for performance analysis finite representations are usually of vital importance[1]. To overcome this problem we use McMillan's complete finite prefix approach [26]. This technique, originally developed for 1-safe Petri nets and recently adapted to process algebra [24], constructs an initial part of the infinite semantic object that contains all information on reachable states and transitions.

In our stochastic process algebra the advance of (probabilistic) time and the occurrence of actions is separated. This separation of discrete and continuous phases is similar to that in many timed process algebras and has been recently proposed in the stochastic setting [16,17]. Most recent proposals for incorporating general distributions into process algebra follow this approach [3,6]. As a result of this separation, interaction gets an intuitive meaning – "wait for the slowest process" – with a clear stochastic interpretation. Moreover, abstraction of actions becomes possible. We will show that due to this separation the complete finite prefix approach for process algebra [24] can be easily exploited. We use the prototype tool FOREST to automatically generate a complete finite prefix from a (stochastic) process algebraic specification.

From the finite prefixes we generate so-called stochastic task graphs, acyclic directed graphs where nodes represent tasks of which the delay is represented by a random variable and arcs denote causal dependencies between tasks. Efficient numerical analysis techniques exist for task graphs, and have been implemented. For series-parallel graphs numerical results are exact and algorithms exist to compute the distribution of the delay between a start and finish task. For arbitrary graphs various approximate techniques exist to compute (rather exact) bounds on the mean delay [21]. We use the PEPP tool suite [8,15] to analyse the task graphs generated from the complete prefixes.

[1] Apart from discrete-event simulation techniques and analysis techniques for regular structures (such as birth-death processes), that we do not consider here.

Most attempts to incorporate general distributions in process algebra aim at discrete-event simulation techniques [3,6,14]. To the best of our knowledge, this paper presents the first approach to analyse stochastic process algebraic specifications that may contain general distributions in a numerical manner.

The applicability of our approach is illustrated by analysing the root contention phase within the IEEE 1394 serial bus protocol [20]. In particular, we analyse the distribution of the delay between the detection of a root contention and its first resolution.

The paper is organised as follows. In Sect. 2 we present a stochastic process algebra with general distributions. In Sect. 3 we show how to obtain annotated partial orders using the FOREST tool for finite prefixes. In Sect. 4 we discuss how these partial orders can be seen as task graphs that can be analysed with the tool PEPP. In Sect. 5 we show how to combine FOREST and PEPP in order to perform a mean delay analysis of events after a specific state. Sect. 6 contains an application to the IEEE 1394 protocol, and Sect. 7 is devoted to conclusions and further work. An extended version of this paper can be found in [31].

2 A Stochastic Process Algebra

Let Act be a set of actions, $a \in Act$, $A \subset Act$, and F, G be *general* continuous probability distributions. The distinction between observable and invisible actions plays no role in this paper. The stochastic process algebra used here is a simple process algebra that contains two types of prefix processes: process $a; B$ (*action prefix*) that is able to immediately offer action a while evolving into B, and $\langle F \rangle; B$ (*timed prefix*) that evolves into process B after a delay governed by the continuous distribution F. That is, the probability that $\langle F \rangle; B$ evolves into B before t time units is $F(t)$. In the sequel such actions are called *delay* actions. The syntax of our language is given by the following grammar:

$$B \ ::= \ \mathsf{stop} \ \big| \ a; B \ \big| \ \langle F \rangle; B \ \big| \ B + B \ \big| \ B \parallel_A B \ \big| \ \mathsf{P}$$

The *inaction* process stop cannot do anything. The *choice* between B_1 and B_2 is denoted by $B_1 + B_2$. *Parallel composition* is denoted by $B_1 \parallel_A B_2$ where A is the set of synchronizing actions; $B_1 \parallel_\varnothing B_2$ is abbreviated to $B_1 \parallel\!\parallel\!\parallel B_2$. Processes cannot synchronise on delay actions. The semantics of the parallel operator $\parallel_A$ follows the semantics of the parallel operator of LOTOS [2] and thus allows for multi-way synchronisation. Finally, P denotes *process instantiation* where a behaviour expression is assumed to be in the context of a set of process definitions of the form $\mathsf{P} := B$ with B possibly containing process instantiations of P. In this paper, we assume that a process algebra expression has a finite number of reachable states.

A few words on $B_1 + B_2$ are in order. $B_1 + B_2$ behaves either as B_1 or B_2, but not as both. At execution the fastest process, i.e., the process that is enabled first, is selected. This is known as the race condition. If this fastest process is not uniquely determined, a non-deterministic selection among the fastest processes is made.

Example 1. In the rest of this paper we use the following stochastic process algebra expression as a running example:

$$B_{ex} := (a;\ \langle G\rangle;\ d;\ \mathsf{stop}\ \|_{a,d}\ a;\ \langle F_1\rangle;\ c;\ \langle F_2\rangle;\ d;\ \mathsf{stop})\ \|_c\ b;\ c;\ \mathsf{stop}$$

3 Partial Orders, Finite Prefixes and FOREST

In [25,26], McMillan presents an algorithm that, for a given 1-safe Petri net, constructs an initial part of its occurrence net called *unfolding* or maximal branching process [9,28]. The so-called *complete finite prefix* of the occurrence net contains all information on reachable states and transitions. An important optimisation of the algorithm has been defined in [11]. This complete finite prefix can be used as the basis for model checking [10,34].

In [24], Langerak and Brinksma adopt the complete finite prefix approach for process algebra for a model similar to occurrence nets called *condition event structures*. In doing so, they have given an event structure semantics to process algebra. In this section, we briefly recall some definitions of [11] and [24] that are needed for the remainder of this paper. We show how to obtain partial orders from local configurations. Finally, we introduce FOREST, a prototype tool which is based on the results of [24].

Conditions and Events. A process algebra expression can be decomposed into so-called conditions, which are action prefix expressions together with information about the synchronisation context [29]. A *condition* C is defined by

$$C\ ::=\ \mathsf{stop} \mid a;B \mid \langle F\rangle;B \mid C\|_A \mid \|_A C$$

where B is a process algebra expression. Intuitively, a condition of the form $C\|_A$ means that C is the left operand of a parallel operator with action set A. Similarly, a condition of the form $\|_A C$ means that C is the right operand of a parallel operator with action set C. For the construction of the complete finite prefix, the distinction between action prefix conditions and time prefix conditions plays no role; in the sequel both prefix conditions will be represented by the expression $a;B$.

A *condition event structure* is a 4-tuple $(\mathbb{C},\mathbb{E},\bowtie,\prec)$ with $\mathbb{C}$ a set of conditions, $\mathbb{E}=\mathbb{E}_{act}\cup\mathbb{E}_{delay}$ a set of events, $\bowtie\ \subset \mathbb{C}\times\mathbb{C}$, the choice relation (symmetric and irreflexive), and $\prec\ \subseteq (\mathbb{C}\times\mathbb{E})\cup(\mathbb{E}\times\mathbb{C})$ the *flow* relation. The set $\mathbb{E}_{act}$ is the set of action events and $\mathbb{E}_{delay}$ is the set of delay events. Let E be a set of events, then the function $delay(E)$ returns the delay events of E, i.e. $delay(E)=\{e\in E \mid e\in\mathbb{E}_{delay}\}$. Condition event structures are closely related to Petri nets; the conditions correspond to places whereas the events correspond to transitions. In [24], actions and process instantiations are labelled with unique indices. These indices are used to create unique event identifiers. Furthermore, these indices are used to efficiently compute the finite prefix. For this paper, these indices and identifiers are not important, and therefore omitted.

States. A *state* is a tuple $(S,\ R)$ with $S \subseteq \mathbb{C}$ a set of conditions, and $R \subseteq S \times S$, an irreflexive and symmetric relation between conditions called the choice relation: $R \subseteq \bowtie$. A state $(S,\ R)$ corresponds to a 'global state' of the system; for each process in the system it stores the possible next condition(s). In fact, a state can always be represented by a process algebra expression. Conditions and their choice relations can be obtained by decomposing a process algebra expression. The decomposition function *dec*, which maps a process algebra expression B onto a state, is recursively defined by $dec(B) = (S(B),\ R(B))$ with

$$\begin{array}{lcl}
dec(\mathsf{stop}) & = & (\{\mathsf{stop}\},\ \varnothing) \\
dec(a; B) & = & (\{a; B\},\ \varnothing) \\
dec(B_1 \parallel_A B_2) & = & (S(B_1) \parallel_A \cup \parallel_A S(B_2),\ R(B_1) \parallel_A \cup \parallel_A R(B_2)) \\
dec(B_1 + B_2) & = & (S(B_1) \cup S(B_2),\ R(B_1) \cup R(B_2) \cup (S(B_1) \times S(B_2))) \\
dec(\mathsf{P}) & = & dec(B) \text{ if } \mathsf{P} := B
\end{array}$$

In [24] it is shown how this decomposition function can be used to construct a derivation system for condition event transitions (i.e. the $\prec$ relation).

Configurations. Let $(\mathbb{C}, \mathbb{E}, \bowtie, \prec)$ be a condition event structure. We adopt some Petri net terminology: a *marking* is a set of conditions. A *node* is either a condition or an event. The *preset* of a node n, denoted by $\bullet n$, is defined by $\bullet n = \{m \in \mathbb{C} \cup \mathbb{E} \mid m \prec n\}$, and the *postset* $n\bullet$ by $n\bullet = \{m \in \mathbb{C} \cup \mathbb{E} \mid n \prec m\}$. The *initial marking* M_0 is defined by $M_0 = \{c \in \mathbb{C} \mid \bullet c = \varnothing\}$. An event e is *enabled* in a marking M if $\bullet e \subseteq M$. Let M be a marking, then we define the function *enabled*(M) as follows: *enabled*$(M) = \{e \in \mathbb{E} \mid \bullet e \subseteq M\}$.

The transitive and reflexive closure of the flow relation $\prec$ is denoted by $\leq$. The *conflict* relation on nodes, denoted by $\#$, is defined as follows: let n_1 and n_2 be two different nodes, then $n_1 \ \# \ n_2$ iff there are two distinct nodes m_1 and m_2, such that $m_1 \leq n_1$ and $m_2 \leq n_2$, with either (i) m_1 and m_2 are two conditions in the choice relation, i.e. $m_1 \bowtie m_2$, or (ii) m_1 and m_2 are two events with $\bullet m_1 \cap \bullet m_2 \neq \varnothing$. Two nodes n_1 and n_2 are said to be *independent*, notation $n_1 \asymp n_2$, iff $\neg(n_1 \leq n_2) \wedge \neg(n_2 \leq n_1) \wedge \neg(n_1 \bowtie n_2)$.

Let c be a condition, then we define $\bowtie(c)$ to be the set of conditions in choice with c, i.e. $\bowtie(c) = \{c' \in \mathbb{C} \mid c \bowtie c'\}$. Similarly for a set of conditions C: $\bowtie(C) = \{c' \in \mathbb{C} \mid \exists c \in C : c \bowtie c'\}$. For an event $e \in \mathbb{E}$, and M and M' markings, there is an *event transition* $M \xrightarrow{e} M'$ iff $\bullet e \subseteq M$ and $M' = (M \cup e\bullet) \setminus (\bullet e \cup \bowtie(\bullet e))$. An *event sequence* is a sequence of events $e_1 \ldots e_n$ such that there are markings $M_1,\ \ldots,\ M_n$ with $M_0 \xrightarrow{e_1} M_1 \longrightarrow \ldots \xrightarrow{e_n} M_n$.

We call $E_{conf} = \{e_1,\ \ldots,\ e_n\}$ a *configuration* of the condition event structure. A configuration E_{conf} must be conflict-free and backward closed with respect to the $\leq$ relation. For an event $e \in \mathbb{E}$, the *local configuration* $[e]$ is defined by $[e] = \{e' \in \mathbb{E} \mid e' \leq e\}$. The causal ordering $\leq$ restricted to $\mathbb{E} \times \mathbb{E}$ induces a *partial order* over a local configuration $[e]$ (see [22,28,30]).

A *cut* is a marking M which is maximal w.r.t. set inclusion and such that for each pair of different conditions c and c' in M the following holds: $c \asymp c'$ or $c \bowtie c'$. It can be shown [24] that each configuration corresponds to a cut which

can be uniquely associated to a state. The state corresponding to the cut of a configuration E_{conf} is denoted by $State(E_{conf})$.

Unfolding. In [24], Langerak and Brinksma present an algorithm to *unfold* a process algebra expression B into a condition event structure $Unf(B)$. The representation $Unf(B)$ may be infinite for recursive processes. In order to overcome this problem they adopted McMillan's approach to compute the so-called *complete finite prefix* of a Petri net unfolding to the setting of condition event structures. The finite prefix algorithm is based on a partial order relation $\sqsubset$, called an *adequate order*. This relation is defined on finite configurations of $Unf(B)$. This adequate order $\sqsubset$ is used to identify so-called *cut-off* events which do not introduce new global states. An event e is a *cut-off event* if $Unf(B)$ contains a local configuration $[e_0]$ such that (i) $State([e]) = State([e_0])$ and (ii) $[e_0] \sqsubset [e]$. So, a cut-off event is an event of which the marking corresponds to a global state which has already been identified 'earlier' in the unfolding. Conceptually a finite prefix is obtained by taking an unfolding $Unf(B)$ and cutting away all successor nodes of cut-off events. It is clear that the finite prefix depends on the adequate order $\sqsubset$ used to compare configurations. Furthermore, the complete finite prefix approach only works for finite state processes, i.e. processes with a finite number of reachable states. In this paper we adopt the adequate order of [24]. The complete finite prefix corresponding with this adequate order is denoted by $FP(B)$.

Example 2. Fig. 1 shows the condition event structure of the unfolding $Unf(B_{ex})$ of the process algebra expression B_{ex}. As the process algebra expression B_{ex} does not contain process recursion, the unfolding $Unf(B_{ex})$ is already finite by itself. Conditions are represented by circles and events are depicted by squares. The initial marking M_0 is represented by the three conditions at the top of Fig. 1. The local configuration of event c is $[c] = \{a, \langle F_1 \rangle, b, c\}$. The state of configuration $[c]$ is formally represented by $State([c]) = (\{\langle G \rangle; d; \mathsf{stop} \parallel_{a,d} \parallel_c, \parallel_{a,d} \langle F_2 \rangle; d; \mathsf{stop} \|_c, \|_c \mathsf{stop}\}, \varnothing)$. The process algebra expression corresponding with $State([c])$ is $(\langle G \rangle; d; \mathsf{stop} \parallel_{a,d} \langle F_2 \rangle; d; \mathsf{stop}) \parallel_c \mathsf{stop}$. The local configuration of event d is $[d] = \{a, b, \langle G \rangle, \langle F_1 \rangle, c, \langle F_2 \rangle, d\}$. The state of configuration $[d]$ is represented by the three leaf conditions. The partial order of the events within $[d]$ is induced by the flow relation $\prec$ which is depicted by the arrows between the conditions and events.

FOREST. FOREST[2][31] is a prototype tool that is based on the unfolding and finite prefix algorithms of [24]. Given a process algebra expression B (with a finite number of reachable states), FOREST computes the corresponding complete finite prefix $FP(B)$ as a condition event structure. The tool allows to use McMillan's original adequate ordering or the adequate ordering defined in [24]. FOREST is used as a prototype tool to experiment with several aspects of the unfolding algorithm, like alternative adequate orderings, independence algorithms, cut-off

[2] FOREST stands for "a tool for event structures".

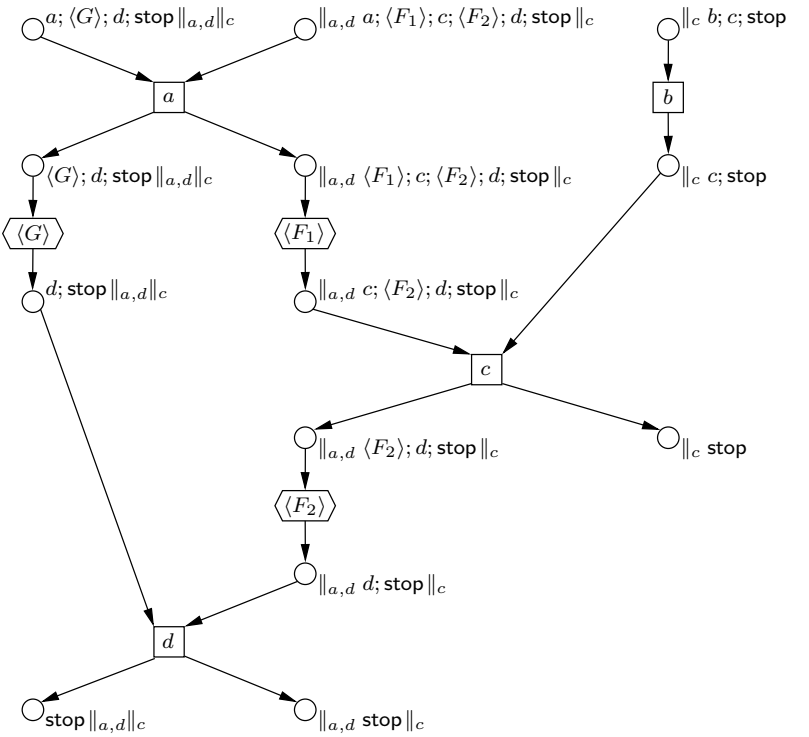

Fig. 1. Condition event structure of the unfolding $Unf(B_{ex})$.

criteria, etc. For these experimental purposes, FOREST can either export the finite prefix $FP(B)$ to a textual representation or to a format suitable as input for graph drawing tools like `vcg` [32] or `dot` [12]. Future additions to FOREST will include an interactive visual simulator and a model checking module. FOREST has been implemented in C++ (4000 lines of code) and the development took roughly eight man months.

This section has only briefly addressed the construction of the complete finite prefix $FP(B)$ of a process algebra expression B. For the remainder of this paper, the most important aspect of the unfolding algorithm is that the construction of the condition event structure induces a partial order on the events of a (local) configuration. FOREST can be used to compute such partial orders.

4 Task Graph Analysis and PEPP

The tool PEPP[3] has been developed at the University of Erlangen [8,15] in the early nineties of the previous century. The tool has a broad functionality, amongst which program instrumentation, monitoring (using the hardware

[3] PEPP stands for "Performance Evaluation of Parallel Programs".

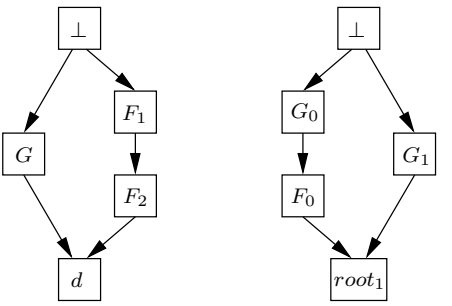

Fig. 2. Task graphs of (a) the local configuration of event d of the running example B_{ex} and (b) the local configuration of event $root_1$ of the root contention protocol (discussed in Sect. 6).

monitor ZM4) and trace analysis. In this paper, we only use the following functionality of PEPP: (i) the creation of task graphs and (ii) the automatic analysis of task graphs.

Task graphs consist of nodes connected by directed edges. Nodes, which represent tasks to be executed, can be of several types (e.g. hierarchical, cyclic and parallel nodes) but here we will only use so-called *elementary* nodes that model activities taking a certain amount of time, i.e. delay actions. The time that an activity or task takes is governed by a continuous distribution function. The dependency between tasks is modelled by the directed edges between the nodes.

PEPP supports several built-in distribution functions, like deterministic, exponential, approximate, and mixed Erlang distributions, the parameters of which can be chosen by the user. It is also possible to use general distributions in a numerical form, that can either be created by the user or by the additional tool CAPP[4] [27]. A numerical representation of a distribution is given by a text file containing the offset of the density function, the step size and the density values for each step. CAPP also allows the graphical representation of distribution and density functions. Nodes can be created interactively by the user via a graphical interface. Nodes can be connected by edges representing causal dependencies. These dependencies are required to be acyclic and the resulting graph is called a task graph. In fact a task graph can be seen as a partial ordering of nodes.

Example 3. Suppose we are interested in the running time of event d of B_{ex}, starting from the initial state. If we only consider the delay actions and the causal dependencies of $Unf(B_{ex})$ of Fig. 1, we obtain the task graph of Fig. 2 (a).

Analysis of task graphs. After a task graph has been input to PEPP, the run time distribution of the model can be analysed in several ways. The most attractive mode of analysis is via SPASS[5]. In order to analyse a task graph

[4] CAPP stands for "Calculation and Presentation Package".
[5] SPASS stands for "Series Parallel Structure Solver".

using SPASS it has to be in *series-parallel* reducible form. This means it can be reduced to a single node by successively applying two reduction steps:

- *Series reduction:* in this reduction step a sequence of nodes is reduced to a single node.
- *Parallel reduction:* in this reduction step several parallel nodes with the same predecessors and successors are reduced to a single node.

SPASS analysis is an exact type of analysis; the SPASS reductions preserve the performance analysis aspects. A series-parallel task graph is reduced by SPASS to a single node with a distribution that represents the first passage time of the complete task graph. This distribution function is calculated in a numerical way and can be visualised (together with its corresponding density function) using CAPP.

If a task graph is not series-parallel reducible it can be analysed using several well-known approximate (bounding) methods [15,21]. The basic idea behind these approximations is that nodes are added or deleted until the task graph becomes series-parallel reducible. This leads to upper and lower bounds of the first passage time of the task graph. PEPP offers several of these bounding techniques.

It is also possible to approximate the analysis by transforming the task graph into an interleaving transition system, and approximating the distributions by deterministic and exponential distributions. This approximation suffers heavily from state space explosion problems and does not exploit the advantages of the partial order properties; for these reasons this analysis method has not been used in this paper.

PEPP is a powerful analysis tool for stochastic task graphs. For simple examples these task graphs can easily be created in a manual way. For realistic system designs, though, developing task graphs in a direct way becomes more and more cumbersome and error prone. An effective solution to this problem, as first recognised by Herzog in [18], is to automatically generate task graphs in a compositional manner from a stochastic process algebra specification. While [18] reveals some problems in using task graphs as a semantic model for (non-recursive) stochastic process algebras, our approach – that is aimed at recursive processes – is to generate task graphs from a finite event structure semantics.

5 First Passage Time Analysis

In Sect. 3, we discussed how a partial order of events of a local configuration $[e]$ can be obtained from a finite prefix $FP(B)$ of an unfolding. In this section we discuss how the partial orders generated by FOREST can be used for first passage time analysis with PEPP.

Algorithm 1 constructs a task graph of the local configuration of an event e starting from the initial state of a process algebra expression. With PEPP we can compute the first passage time of event e to occur (starting from the initial state). If the partial order of $[e]$ happens to be series-parallel reducible, PEPP will even compute the distribution function of the runtime.

Algorithm 1. Construct the task graph for $[e]$ starting from the initial state of B.

1. Specify the target event e within the process algebra expression B.
2. Use FOREST to compute the finite prefix $FP(B)$ until event e has occurred or the complete finite prefix has been generated. If the prefix does not contain e (which means that e is not reachable), then stop; apparently the problem was not well-defined.
3. Consider the local configuration $[e]$; together with the causal ordering $\leq$ this induces a partial order P.
4. Project P onto the delay events. This yields a task graph that can be used as input to PEPP.

But the analysis is not restricted to starting from the initial state. We can also supply a set of independent events $\{e_1, \ldots, e_n\}$ as a starting point, and ask for the passage time for an event e to occur after these events. There is however a constraint involved here: if after the events $\{e_1, \ldots, e_n\}$ a delay action is enabled, then this delay action has to be causally dependent on at least one event in $\{e_1, \ldots, e_n\}$. In other words, the following should hold: $\forall e_{en} \in \mathit{enabled}(\mathit{State}([e_1] \cup \ldots \cup [e_n])) : e_{en} \in \mathbb{E}_{\mathit{delay}} \Rightarrow \exists e_i \in \{e_1, \ldots, e_n\} : e_i \leq e_{en}$. Otherwise there is no way to determine when such a delay event e_{en} may have started. Algorithm 2 shows how to apply PEPP when the start state is determined by a set of independent events within the finite prefix $FP(B)$. Of course, the target event e has to be causally dependent on the events in $\{e_1, \ldots, e_n\}$. If this is not the case, Algorithm 1 – started in step 5 of Algorithm 2 – will unsuccessfully terminate in step 2.

Note that the partial orders obtained by both algorithms are only useful for PEPP if the configuration between the initial event(s) and the target event e contains at least one delay event.

Example 4. Consider Fig. 1 which corresponds to the condition event structure of $\mathit{Unf}(B_{ex})$. Suppose we are interested in the runtime of event d. The sets $\{a\}$ and $\{a, b\}$ can both be used as input for Algorithm 2; in fact, for this example they will all yield the task graph of Fig. 2 (a). The singleton set $\{b\}$ can also be used as a starting point for Algorithm 2, as the delay events $\langle G \rangle$ and $\langle F_1 \rangle$ are not enabled in $\mathit{State}([b])$; only the event a is enabled in $\mathit{State}([b])$. Again, the task graph of Fig. 2 (a) will be computed. The singleton set $\{c\}$, however, cannot be used as a valid input for Algorithm 2 as the delay event $\langle G \rangle$, which is enabled in $\mathit{State}([c])$, does not depend on event c. The set $\{a, c\}$ cannot be used as input for Algorithm 2 either, because the events a and c are not independent: $a \leq c$.

6 The Root Contention Phase in IEEE 1394

This section discusses a small case study where we applied our approach to compute the mean passage time of the first resolution of the root contention phase

Algorithm 2. Construct the task graph for $[e]$ starting from $\{e_1, \ldots, e_n\}$.

1. Specify the target event e within the process algebra expression B.
2. Use FOREST to compute the finite prefix $FP(B)$ until the events $\{e_1, \ldots, e_n\}$ have all occurred or the complete finite prefix has been generated. If $FP(B)$ does not contain all events of $\{e_1, \ldots, e_n\}$, then stop; apparently the problem was not well-defined.
3. If there are conflicts among the events in $\{e_1, \ldots, e_n\}$ then stop, as apparently the problem is not well-defined; otherwise continue.
4. Calculate $S = \mathit{State}([e_1] \cup \ldots \cup [e_n])$.
5. Check if all enabled delay actions which causally depend on S are dependent on at least one event from $\{e_1, \ldots, e_n\}$. If not, stop; apparently the problem is not well-defined. Otherwise, apply Algorithm 1 with S as initial state and compute the partial order of the local configuration of target event e.

of the IEEE 1394 protocol [20]. Due to space limitations only the FOREST and PEPP models of the root contention phase are discussed here. A more thorough discussion can be found in [31].

FOREST. Fig. 3 presents the specification of the root contention protocol in our process algebra. The model itself is based on [33]. The two Node_i processes are connected to each other by two Wire_i processes, that represent the communication lines between the components. Each Node_i process has a Buf_i process which can hold a single message from the other $\mathsf{Node}_{(1-i)}$. New messages from $\mathsf{Node}_{(1-i)}$ will simply overwrite older messages. Both nodes start (via Proc_i) to wait $g_i(t)$ units of time. If after waiting, the buffer is still empty (i.e. $\mathit{check\_emp}_i$), the node will sent a $\mathit{send\_req}_i$ to its partner and will subsequently wait for an acknowledgement. If this acknowledgement (i.e. $\mathit{check\_ack}_i$) arrives, Node_i will declare itself a child using action child_i. On the other hand, if after waiting $g_i(t)$ units of time, Node_i receives a $\mathit{check\_req}_i$ action, it declares itself to be the leader using action root_i. The delay of the communication line is modelled by the delay action $\langle F_i \rangle$.

The basic idea behind the protocol is that if the waiting times $g_i(t)$ of the two nodes are different, the 'slowest' node will become root. Since with probability one the outcomes of the waiting times $g_i(t)$ will eventually be different, the root contention protocol will terminate with probability one [33].

Apart from the performance analysis of the protocol that we report on in this paper, the specification of Fig. 3 may readily be used for a functional analysis of the protocol. The condition event structure generated by FOREST for this process algebraic expression contains 57 events (of which 8 are cut off-events) and 210 conditions.

To illustrate both algorithms of Sect. 5 we have identified a start state in the process algebra expression of Fig. 3 (i.e. corresponding with the events $\{e_1, \ldots, e_n\}$) from which we want to compute the first passage time to another state (i.e. target event e). The start state is defined by the first occurrence of the

The root contention protocol is modelled by the following stochastic process algebra expression:

$$(\mathsf{Node}_0 \;|||\; \mathsf{Node}_1) \;\|_{Glob}\; (\mathsf{Wire}_0 \;|||\; \mathsf{Wire}_1)$$

with the following (process) definitions ($i \in \{0,1\}$) :

$$\begin{array}{lcl}
\mathsf{Node}_i & := & (\mathsf{Proc}_i \;\|_{Loc}\; \mathsf{Buf}_i) \\
\mathsf{Proc}_i & := & \langle G_i \rangle;\; (\mathit{check_emp}_i;\; \mathsf{SndReq}_i + \mathit{check_req}_i;\; \mathsf{SndAck}_i) \\
\mathsf{SndAck}_i & := & \mathit{send_ack}_i;\; \mathit{root}_i;\; \mathsf{stop} \\
\mathsf{SndReq}_i & := & \mathit{send_req}_i;\; (\mathit{check_req}_i;\; \mathsf{Proc}_i + \mathit{check_ack}_i;\; \mathit{child}_i;\; \mathsf{stop}) \\
\mathsf{Buf}_i & := & \mathit{check_emp}_i;\; \mathsf{Buf}_i + \mathit{recv_req}_i;\; \mathsf{BufReq}_i + \mathit{recv_ack}_i;\; \mathsf{BufAck}_i \\
\mathsf{BufReq}_i & := & \mathit{check_req}_i;\; \mathsf{Buf}_i + \mathit{recv_req}_i;\; \mathsf{BufReq}_i + \mathit{recv_ack}_i;\; \mathsf{BufAck}_i \\
\mathsf{BufAck}_i & := & \mathit{check_ack}_i;\; \mathsf{Buf}_i + \mathit{recv_req}_i;\; \mathsf{BufReq}_i + \mathit{recv_ack}_i;\; \mathsf{BufAck}_i \\
\mathsf{Wire}_i & := & \mathit{send_req}_i;\; \mathsf{WireReq}_i + \mathit{send_ack}_i;\; \mathsf{WireAck}_i \\
\mathsf{WireReq}_i & := & \langle F_i \rangle;\; \mathit{recv_req}_{(1-i)};\; \mathsf{Wire}_i + \mathsf{Wire}_i \\
\mathsf{WireAck}_i & := & \langle F_i \rangle;\; \mathit{recv_ack}_{(1-i)};\; \mathsf{Wire}_i + \mathsf{Wire}_i \\
\mathit{Glob} & == & \{\mathit{send_req}_0,\; \mathit{send_req}_1,\; \mathit{send_ack}_0,\; \mathit{send_ack}_1, \\
 & & \mathit{recv_req}_0,\; \mathit{recv_req}_1,\; \mathit{recv_ack}_0,\; \mathit{recv_ack}_1\} \\
\mathit{Loc} & == & \{\mathit{check_emp}_i,\; \mathit{check_req}_i,\; \mathit{check_ack}_i\}
\end{array}$$

Fig. 3. Process algebra expression of the root contention protocol.

following actions: $\mathit{send\_req}_0$, $\mathit{recv\_req}_0$, $\mathit{send\_req}_1$ and $\mathit{recv\_req}_1$. That is, just before *both* the $\mathit{check\_req}_0$ and $\mathit{check\_req}_1$ actions are about to happen. In the root contention protocol, this corresponds to the situation in which both processes are about to receive the parent request of their contender, which will initiate a new contention resolution phase. In the graph representation of the corresponding condition event structure, this set of starting events can easily be identified, due to the flow relation $\prec$ and the induced causal order $\leq$. The complete $FP(B)$ is omitted due to its size, though.

From these four events, we are interested in the delay until the first occurrence of the event corresponding with action root_1, that is, the first resolution after the contention, which declares Node_1 to be the root. Fig. 4 shows the partial order of the events leading to this root_1 event. It is generated using the graph drawing tool `dot` [12]. Note that the events $\mathit{check\_req}_0$ and $\mathit{check\_req}_1$ are indeed the first events that can occur. Within FOREST, distribution events all have a *del_* prefix.

PEPP. For the runtime analysis with PEPP[6], only the delay events of the partial order are of interest. Fig. 2 (b) shows the task graph as used by PEPP containing only the delay events and the elementary start and end events; it is the projection of Fig. 4 on the delay events. For the events $\langle G_0 \rangle$ and $\langle G_1 \rangle$ we have used the same uniform distribution function $G(t)$, that is used in [5] for the

[6] For our experiments we used version 3.3 of PEPP (released in July 1993) [7].

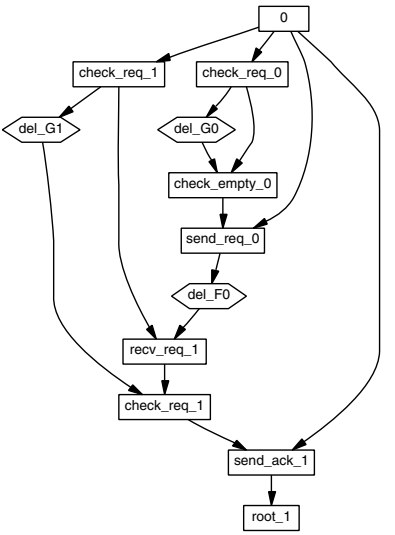

Fig. 4. Partial order of the events leading to the $root_1$ event.

transient analysis (via simulation) of the root contention protocol. A graphical representation of the distribution function $G(t)$ is given in Fig. 5. The unit of the Figures 5 and 6 is $\frac{1}{100}\,\mu sec$ as PEPP requires fixed step-size. The parameters D, N and I are parameters of the density function $g(t)$ which is defined as tuple $g = (D, N, g_0, \ldots, g_I)$, where D is the displacement between 0 and g_0 and N is the order of the distribution [15]. For the delay event $\langle F_0 \rangle$ we used an uniform distribution function $F(t)$ between $\frac{2}{198}$ and $\frac{3}{198}$, assuming that the transmission speed of the lines is $198m/\mu sec$.

We used PEPP to analyse the task graph corresponding with the local configuration of the $root_1$ event. Fig. 6 shows a graphical representation of the density function of the first passage time together with the average time ($0.51\,\mu sec$) of the partial order leading to $root_1$. Note that the results obtained only relate to the time of the root contention when contention is resolved on the first attempt of the protocol. It does not provide information on the transient behaviour of the protocol.

7 Conclusions

In this paper we discussed a partial-order semantics for a stochastic process algebra that supports general (non-memoryless) distributions and combined this with an approach to numerically analyse the mean delay between two events. Based on an adaption of McMillan's complete finite prefix approach tailored to event structures and process algebra, we used FOREST to obtain finite representations for recursive processes. The behaviour between two events is now captured by a partial order of events that can be mapped on a stochastic task graph. We used PEPP for numerical analysis of such task graphs.

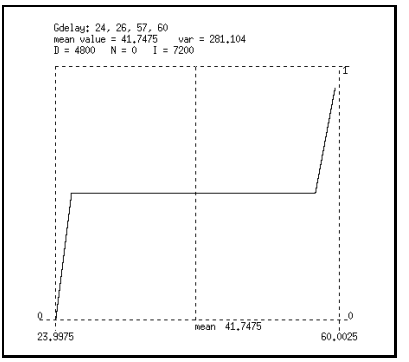

Fig. 5. Distribution function $G(t)$.

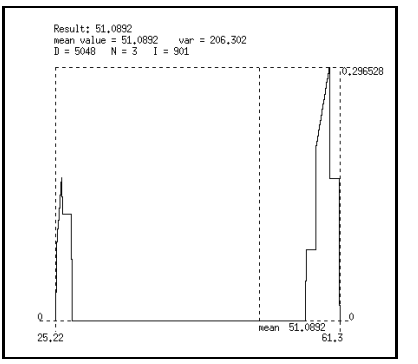

Fig. 6. Density function of the runtime of the configuration leading to the $root_1$ event.

The paper presents a novel application of McMillan's finite prefix algorithm. Furthermore, the work can be seen as a successor of [4] in the sense that it shows the practical feasibility of the use of event structure semantics for stochastic analysis.

A clear advantage of our approach is that we are able to reason about both the functional and non-functional aspects of systems using the same model and notation. Furthermore, as our approach uses *general* distributions, hence, it can still be used when approximations through exponential distributions are no longer realistic.

We foresee three different uses of our approach for performance modelling. First, it is possible that the first passage time between two events is simply what one is interested in, and then our approach yields the answer. Secondly, our approach might play an auxiliary role in establishing the right parameters for performance models of other types that can then be further analysed. Thirdly, our approach might be the first step in a more evolved numerical calculation exploiting more features of PEPP.

In the current setting we are able to compute the first passage time of a single event. Our next step will be to try to adopt our approach to the combined runtimes of conflicting events and repetitive events. In [23], Langerak has shown how to derive a graph rewriting system from the complete finite prefix of a condition event structure. We are currently studying the use of a graph rewriting system as the basis for transient analysis with PEPP, using its more advanced node types like cyclic nodes, hierarchical nodes and probabilistic choice. This would make it possible to compare our work with discrete-event simulation approaches like ⏢ [5,6]. Furthermore, we are currently working on an (user) interface to integrate FOREST and PEPP.

Acknowledgements We want to thank Markus Siegle from the University of Erlangen for kindly giving us permission to use the PEPP tool. Thanks to Holger Hermanns for support, discussions and valuable help in unravelling CAPP representations.

Boudewijn Haverkort is thanked for his helpful suggestions to improve both the contents and readability of the paper.

References

1. 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.
2. T. Bolognesi and E. Brinksma. Introduction to the ISO Specification Language LOTOS. *Computer Networks and ISDN Systems*, 14:25–59, 1987.
3. M. Bravetti and R. Gorrieri. Interactive Generalized Semi-Markov Processes. In J. Hillston and M. Silva, editors, *Proc. of PAPM'99*, pages 83–98, Zaragoza, Spain, September 1999.
4. E. Brinksma, J.-P. Katoen, R. Langerak, and D. Latella. A Stochastic Causality-Based Process Algebra. *The Computer Journal*, 38(7):552–565, 1995.
5. P. R. D'Argenio. *Algebras and Automata for Timed and Stochastic Systems*. PhD thesis, University of Twente, Enschede, The Netherlands, November 1999.
6. P. R. D'Argenio, J.-P. Katoen, and E. Brinksma. An Algebraic Approach to the Specification of Stochastic Systems (extended abstract). In D. Gries and W.-P. de Roever, editors, *Proc. of PROCOMET'98*, pages 126–148, Shelter Island, New York, USA, 1998. Chapman & Hall.
7. P. Dauphin, F. Hartleb, M. Kienow, V. Mertsiotakis, and A. Quick. PEPP: Performance Evaluation of Parallel Programs – User's Guide — Version 3.3. Technical Report 17/93, IMMD VII, University of Erlangen–Nürnberg, Germany, 1993.
8. P. Dauphin, R. Hofmann, R. Klar, B. Mohr, A. Quick, M. Siegle, and F. Sötz. ZM4/SIMPLE: a General Approach to Performance-Measurement and Evaluation of Distributed Systems. In T. L. Casavant and M. Singhal, editors, *Advances in Distributed Computing: Concepts and Design*. IEEE Computer Society Press, 1992.
9. J. Engelfriet. Branching Processes of Petri Nets. *Acta Informatica*, 28(6):575–591, 1991.
10. J. Esparza. Model Checking Using Net Unfoldings. *Science of Computer Programming*, 23(2):151–195, 1994.
11. J. Esparza, S. Römer, and W. Vogler. An Improvement of McMillan's Unfolding Algorithm. In T. Margaria and B. Steffen, editors, *Proc. of TACAS'96*, LNCS 1055, pages 87–106, Passau, Germany, March 1996. Springer-Verlag.
12. E. R. Gansner, E. Koutsofios, and S. C. N. an Kiem-Phong Vo. A Technique for Drawing Directed Graphs. *IEEE Transactions on Software Engineering*, 19(3):214–230, 1993.
13. 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 L. Donatiello and R. Nelson, editors, *Proc. of PERFORMANCE'93*, LNCS 729, pages 121–146, Rome, Italy, September 1993. Springer-Verlag.
14. P. G. Harrison and B. Strulo. Stochastic Process Algebra for Discrete Event Simulation. In F. Bacelli, A. Jean-Marie, and I. Mitrani, editors, *Quantitative Methods in Parallel Systems*, Esprit Basic Research Series, pages 18–37. Springer-Verlag, 1995. Chapter 2.
15. F. Hartleb. Stochastic Graph Models for Performance Evaluation of Parallel Programs and the Evaluation Tool PEPP. In N. Götz, U. Herzog, and M. Rettelbach,

editors, *Proc. of the QMIPS Workshop on Formalisms, Principles and State-of-the-art*, pages 207–224, Erlangen/Pommersfelden, Germany, March 1993. Arbeitsbericht Band 26, Number 14.
16. H. Hermanns. *Interactive Markov Chains*. PhD thesis, University of Erlangen, Nürnberg, Germany, 1998.
17. H. Hermanns and J.-P. Katoen. Automated Compositional Markov Chain Generation for a Plain-Old Telephone System. *Science of Computer Programming*, 36(1):97–127, 2000.
18. U. Herzog. A Concept for Graph-Based Stochastic Process Algebras, Generally Distributed Activity Times, and Hierarchical Modelling. In M. Ribaudo, editor, *Proc. of PAPM'96*, pages 1–20. C.L.U.T. Press, 1996.
19. J. Hillston. *A Compositional Approach to Performance Modelling*. Distinguished Dissertations Series. Cambridge University Press, 1996.
20. IEEE Computer Society. *IEEE Standard for a High Performance Serial Bus*, Std 1394-1995 edition, 1996.
21. R. Klar, P. Dauphin, F. Hartleb, R. Hofmann, B. Mohr, A. Quick, and M. Siegle. *Messung and Modellierung Paralleler und Verteilter Rechensysteme (in german)*. Teubner-Verlag, Stuttgart, 1995.
22. R. Langerak. *Transformations and Semantics for LOTOS*. PhD thesis, University of Twente, Enschede, The Netherlands, November 1992.
23. R. Langerak. Deriving a Graph Grammar from a Complete Finite Prefix of an Unfolding. In I. Castellani and B. Victor, editors, *Proc. of EXPRESS'99*, ENTCS 27, Eindhoven, The Netherlands, August 1999. Elsevier Science Publishers.
24. R. Langerak and E. Brinksma. A Complete Finite Prefix for Process Algebra. In N. Halbwachs and D. Peled, editors, *Proc. of CAV'99*, LNCS 1633, pages 184–195, Trento, Italy, July 1999. Springer-Verlag.
25. K. L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. Kluwer Academic Publishers, 1993.
26. K. L. McMillan. A Technique of State Space Search Based on Unfolding. *Formal Methods in System Design*, 6(1):45–65, 1995.
27. V. Mertsiotakis. Extension of the Graph Analysis Tool SPASS and Integration into the X-Window Environment of PEPP (in german). Technical report, Department of Computer Science VII, IMMD VII, University of Erlangen–Nürnberg, Germany, 1991. Internal study.
28. M. Nielsen, G. D. Plotkin, and G. Winskel. Petri Nets, Event Structures and Domains, Part 1. *Theoretical Computer Science*, 13(1):85–108, 1981.
29. E.-R. Olderog. *Nets, Terms and Formulas*, volume 23 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1991.
30. A. Rensink. *Models and Methods for Action Refinement*. PhD thesis, University of Twente, Enschede, The Netherlands, August 1993.
31. T. C. Ruys. *Towards Effective Model Checking*. PhD thesis, University of Twente, Enschede, The Netherlands, March 2001. To be published.
32. G. Sander. Graph Layout through the VCG Tool. In R. Tamassia and I. G. Tollis, editors, *Proc. of the Int. Workshop on Graph Drawing (GD'94)*, LNCS 894, pages 194–205, Princeton, New Jersey, USA, 1994. Springer-Verlag.
33. M. I. Stoelinga and F. W. Vaandrager. Root Contention in IEEE 1994. In J.-P. Katoen, editor, *Proc. of ARTS'99*, LNCS 1601, pages 53–74, Bamberg, Germany, May 1999. Springer-Verlag.
34. F. Wallner. Model-Checking LTL using Net Unfoldings. In A. J. Hu and M. Y. Vardi, editors, *Proc. of CAV'98*, LNCS 1427, pages 207–218, Vancouver, Canada, July 1998. Springer-Verlag.