Stochastic Process Algebras

Allan Clark, Stephen Gilmore, Jane Hillston, and Mirco Tribastone

LFCS, School of Informatics, University of Edinburgh

Abstract. In this tutorial we give an introduction to stochastic process algebras and their use in performance modelling, with a focus on the PEPA formalism. A brief introduction is given to the motivations for extending classical process algebra with stochastic times and probabilistic choice. We then present an introduction to the modelling capabilities of the formalism and the tools available to support Markovian based analysis. The chapter is illustrated throughout by small examples, demonstrating the use of the formalism and the tools.

1 Introduction

Process algebras emerged as a modelling technique for the functional analysis of concurrent systems approximately twenty years ago. Over the last 17 years there have been several attempts to take advantage of the attractive features of this modelling paradigm within the field of performance evaluation.

Stochastic process algebras (SPA) were first proposed as a tool for performance and dependability modelling in 1990 [1]. At that time there was already a plethora of techniques for constructing performance models so the introduction of another one could have been deemed unnecessary if it were not for the fact that SPA offered something new—formally defined compositionality. Queueing networks, which have been widely used for performance modelling for more than thirty years, have an inherent compositionality but this is implicit and informal. Stochastic extensions of Petri nets have a semantic model but, in general, no clear compositional structure. In the process algebra the compositionality is explicit—provided by the combinators of the language—and formal—supported by the semantics and equivalence relations of the language.

It was immediately clear that having this explicit structure within models offers benefits for model construction:

- when a system consists of interacting components, the components, and the interaction, can each be modelled separately;
- models have a clear structure and are easy to understand;
- models can be constructed systematically, by either elaboration or refinement;
- the possibility of maintaining a library of model components, supporting model reusability, is introduced.

Many case studies demonstrating these and other benefits have appeared in the literature [2,3,4,5,6,7,8,9,10,11,12,13,14,15,16].

M. Bernardo and J. Hillston (Eds.): SFM 2007, LNCS 4486, pp. 132–179, 2007.
© Springer-Verlag Berlin Heidelberg 2007

A limitation of the initial SPA languages was their lack of expressiveness with respect to timing distributions. Essentially, they restricted consideration to models in which all durations were represented by negative exponentially distributed random variables. Some later work has aimed to change this situation by considering languages in which generally distributed random variables may be associated with the actions of a model. However such models are not so amenable to quantitative analysis and therefore their practical uptake has been limited.

The remainder of this tutorial is organised as follows. In the following section we present a short introduction to classical process algebras as they are used for system verification from a functional or qualitative point of view. Stochastic process algebras generally, and the language PEPA specifically, are presented in Section 3. Section 4 describes model analysis. The tools available to support the approaches we have described are discussed in Section 5, and we present some case studies in the following section. In Section 7 we continue to advanced topics such as continuous state-space approximation.

2 Classical Process Algebras

Process algebras are abstract languages used for the specification and design of concurrent systems. The most widely known process algebras are Milner's Calculus of Communicating Systems (CCS) [17] and Hoare's Communicating Sequential Processes (CSP) [18]. The stochastic process algebras take inspiration from both these formalisms. Models in CCS and CSP have been used extensively to establish the correct behaviour of complex systems by deriving *qualitative* properties such as *freedom from deadlock* or *livelock*.

In the process algebra approach systems are modelled as collections of entities, called *agents*, which execute atomic *actions*. These actions are the building blocks of the language and they are used to describe sequential behaviours which may run concurrently, and synchronisations or communications between them.

In CCS two agents communicate when one performs an action, a say, while the other performs the complementary action $\bar{a}$. The resulting communication action has the distinguished label τ, which represents an *internal* action that is invisible to the environment. Agents may proceed with their internal actions simultaneously but it is important to note that the semantics given to the language imposes an interleaving on such concurrent behaviour. The basic calculus contains the following primitives for defining agents:

prefix	$a.B$	after action a the agent becomes B
parallel composition	$A \mid B$	agents A and B proceed in parallel
choice	$A + B$	the agent behaves as A or B depending on which acts first

restriction	$A\backslash M$	the set of labels M is hidden from outside agents
relabelling	$A[a_1/a_0, ..]$	in this agent label a_1 is renamed a_0
the null agent	$\mathbf{0}$	this agent cannot act (deadlock)

The communication mechanism in CSP is different as there is no notion of complementary actions: this is a major distinction between CCS and CSP. In CSP two agents communicate by simultaneously executing actions with the same label. Since during the communication the joint action remains visible to the environment, it can be reused by other concurrent processes so that more than two processes can be involved in the communication (*multiway synchronisation*). This is the communication mechanism adopted by most of the SPA languages.

Like many other process algebras, CCS is given a structured operational semantics (SOS), using a labelled transition system. From this a *derivative tree* or *graph* may be constructed in which language terms form the nodes and transitions are the arcs. This structure is a useful tool for reasoning about agents and the systems they represent. It is also the basis of the *bisimulation* style of equivalence. In this style of equivalence, the actions of an agent characterise it, so two agents are considered to be equivalent if they are observed to perform exactly the same actions. Strong and weak forms of equivalence are defined depending on whether the internal actions of an agent are deemed to be observable.

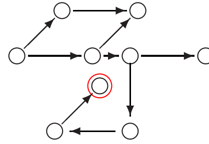
In CCS and CSP, since the objective is qualitative analysis rather than quantitative, time is abstracted away. Various suggestions for incorporating time into these formalisms have been investigated (see [19] for an overview). For example, Temporal CCS [20] extends CCS with *fixed delays* and *wait-for synchronisation* (asynchronous waiting):

fixed time delay	(t)	the agent must wait t time units before performing its next action
wait-for synchronisation	δ	the agent may idle indefinitely until its next action is possible
non-temporal deadlock	$\underline{0}$	the agent idles indefinitely and never engages in further actions.

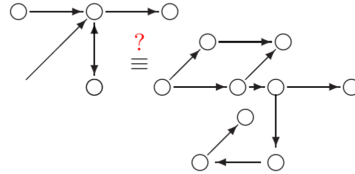
Note that most of the timed extensions, including TCCS, retain the assumption that actions are instantaneous and regard time progression as orthogonal to the activity of the system. In contrast, the early SPAs generally associated a random variable, representing duration, with each action. The alternative approach, of separating action and time, is adopted in most of the work incorporating non-exponentially distributed durations.

Similarly process algebras are often used to model systems in which there is uncertainty about the behaviour of a component, but this uncertainty is

Will the system arrive in a particular state?



Does system behaviour match its specification?



Does a given property ϕ hold within the system?

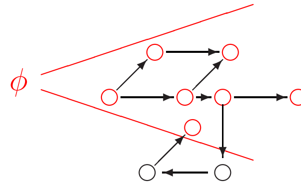


Fig. 1. Functional analysis of process algebra

abstracted away so that all choices become nondeterministic. Probabilistic extensions of process algebras, such as PCCS [21], allow this uncertainty to be quantified using a probabilistic choice combinator. In this case a probability is associated with each possible outcome of a choice. In some SPA an alternative approach is taken—we assume that a *race condition* resolves choices when more than one (timed) action can occur.

The Concurrency Workbench (CWB) [22] is a tool that automates the checking of assertions about CCS models in order to establish properties of the systems they describe. As well as the basic calculus, it supports a synchronous variant and the temporal extension, TCCS. The CWB allows simple properties, such as presence of deadlock, to be checked directly, but needs more specific properties to be expressed in a suitable logic. In the context of process algebra modelling, a process logic is a natural way to frame properties and queries. Such logics, known as *modal logics*, express assertions about changing state. There is a simple modal logic, Hennessy-Milner logic [23], for immediate possibilities in a model, and an extended logic, the modal $\mu-$calculus[24], with fixed point operators for recursive definitions.

3 Stochastic Process Algebra: PEPA

Process algebras offer several attractive features which are not necessarily available in existing performance modelling paradigms. The most important of these are *compositionality*, the ability to model a system as the interaction of its subsystems,

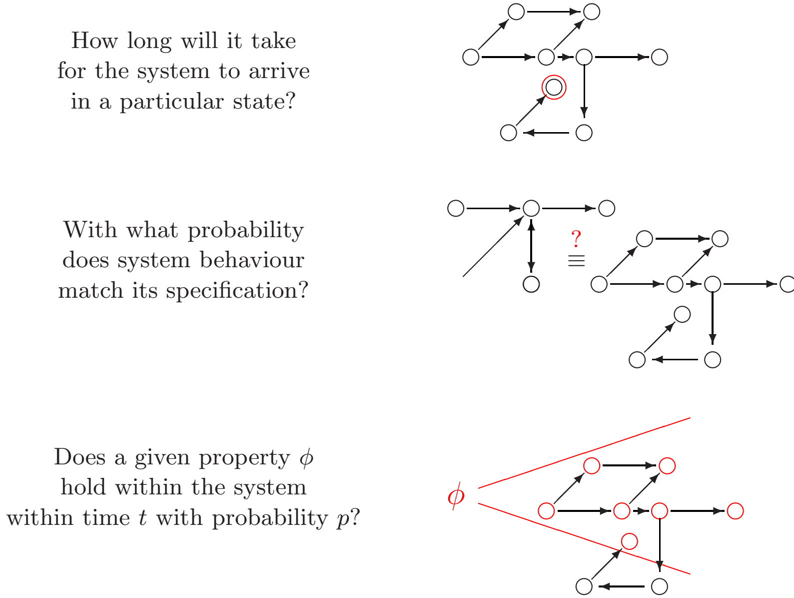


Fig. 2. Quantitative analysis of stochastic process algebra

formality, giving a precise meaning to all terms in the language, and *abstraction*, the ability to build up complex models from detailed components but disregarding internal behaviour when it is appropriate to do so. Queueing networks offer compositionality but not formality; stochastic extensions of Petri nets offer formality but not compositionality; neither offer abstraction mechanisms.

In the early 1990s several stochastic extensions of process algebra appeared in the literature, motivated by a desire to add quantification to process algebra models and make them suitable for performance modelling. These included TIPP [25] from the University of Erlangen, EMPA[1] [26,27] from the University of Bologna, PEPA [28,29] from the University of Edinburgh and SPADE[2] [30] from Imperial College. PEPA was the first language to be developed with the intention of generating Markov processes which could be solved numerically for performance evaluation, but versions of TIPP and EMPA from around the same time are similarly Markovian based. The other Markovian-based SPA, emerged a little later the stochastic π-calculus [31] and IMC [32] and differ in terms of their synchronisation and treatment of delay, respectively. For the remainder of this section, and the following one, we concentrate on PEPA; however, towards the end of this section we will discuss how TIPP and EMPA differ from PEPA. SPADE was developed with a different motivation, relating to generalised semi-Markov processes and simulation. Several other calculi have also

[1] Originally called simply MPA.
[2] Originally called CCS+.

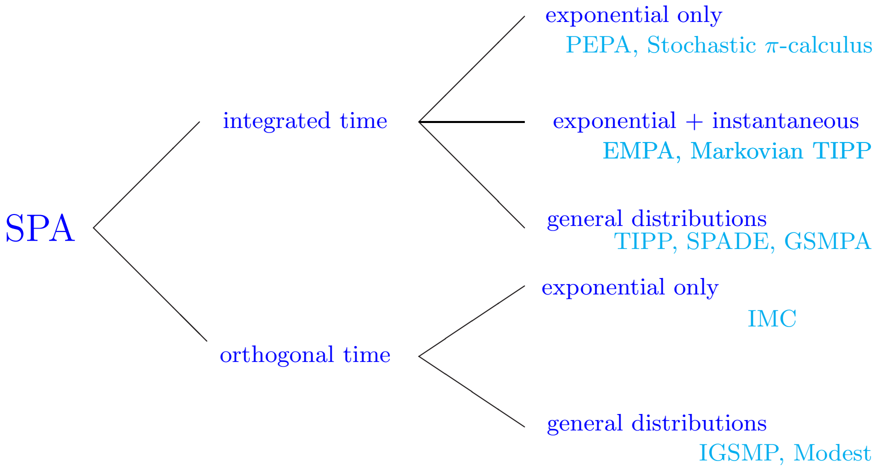


Fig. 3. Classification of the stochastic process algebras

incorporated generally distributed activities or delays, e.g. Modest [33], IGSMP [34] and GSMPA [35].

PEPA (Performance Evaluation Process Algebra) extends classical process algebra by associating a random variable, representing duration, with every action. These random variables are assumed to be exponentially distributed and this leads to a clear relationship between the process algebra model and a continuous time Markov process. Via this underlying Markov process performance measures can be extracted from the model.

PEPA models are described as interactions of *components*. Each component can perform a set of actions: an action $a \in \mathcal{A}ct$ is described as a pair (α, r), where $\alpha \in \mathcal{A}$ is the *type* of the action and $r \in \mathbb{R}^+$ is the parameter of the negative exponential distribution governing its duration. Whenever a process P can perform an action, an instance of a given probability distribution is sampled: the resulting number specifies how long it will take to *complete* the action. A small but powerful set of combinators is used to build up complex behaviour from simpler behaviour. The combinators are familiar from classical process algebra: prefix, choice, parallel composition and abstraction. We explain each of the combinators informally below. A formal operational semantics for PEPA is available in [29].

Prefix: A component may have purely sequential behaviour, repeatedly undertaking one activity after another and eventually returning to the beginning of its behaviour. A simple example is a web service within a distributed system, which can serve one request at a time. Each application requiring the web service will need to gain access to the service which will then only be made available for another application when a response has been successfully transferred.

$$WS \stackrel{\text{def}}{=} (request, \top).(serve, \mu).(respond, \top).WS$$

In some cases, as here, the rate of an action is outside the control of this component. Such actions are carried out jointly with another component, with this component playing a passive role. For example, the web service is passive with respect to the *request* action, as it cannot influence the rate at which requests arrive, and this is recorded by the distinguished symbol, $\top$ (called "top").

Choice: A choice between two possible behaviours is represented as the sum of the possibilities. For example, if we consider an application in a distributed system, a computation may have two possible outcomes: access to a locally available method is required (with probability p_1) or access to a remote web service is necessary (with probability $p_2 = 1 - p_1$). In this example the *think* action denotes processing within the application. These alternatives are represented as shown below:

$$\begin{aligned} Appl \stackrel{\text{def}}{=} & (think, p_1\lambda).(local, m).Appl \\ & + \ (think, p_2\lambda).(request, rq).(respond, rp).Appl \end{aligned}$$

A race condition governs the behaviour of simultaneously enabled actions so the choice combinator represents pre-emptive selection with re-sampling. The continuous nature of the probability distributions ensures that the actions cannot occur simultaneously. Thus a sum will behave as either one summand or the other. When an action has more than one possible outcome, e.g. the *think* action in the application, it is represented by a choice of separate actions, one for each possible outcome. The rates of these actions are chosen to reflect their relative probabilities.

Parallel composition: As mentioned earlier, PEPA and most of the other SPA adopt the parallel composition from CSP, rather than that from CCS. Correspondingly, there is no notion of complementary actions and multiway synchronisations are possible.

In the web service example, we have already anticipated that the application and the web service will be working together within the same system. This will require them to *cooperate* when the application needs the service offered by the web service, which is not available locally. In contrast, the local activities of the application can be carried out independently of the web service. Cooperation over given actions is reflected in the parallel composition by the *cooperation set*, $L = \{request, serve, respond\}$ in this case. Actions in this set require the simultaneous involvement of both components. The resulting action, a *shared* action, will have the same type as the two contributing actions and a rate reflecting the rate of the action in the slowest participating component. Note that this means that the rate of a passive action will become the rate of the action it cooperates with.

If, for simplicity, we assume that the distributed system consists of just two independent applications, the system is represented as the cooperation of the applications and the web service as follows:

$$Sys_1 \stackrel{\text{def}}{=} \big(Appl \parallel Appl\big) \underset{L}{\bowtie} WS \qquad L = \{request, serve, respond\}$$

The combinator $\parallel$ is a degenerate form of the cooperation combinator, formed when two components behave completely independently, without any cooperation between them, as in the case of the two independent applications. This pure parallel combinator can be thought of as cooperation over the empty set: $(Appl \underset{\emptyset}{\bowtie} Appl)$.

Abstraction: Again, the abstraction mechanism used in SPA follows CSP rather than CCS. It is often convenient to hide some actions, making them private to the component or components involved. The duration of the actions is unaffected, but their type becomes hidden, appearing instead as the unknown type τ. Components cannot synchronise on τ. For example, as we further develop the model of the distributed system we may wish to hide the access of a application to its local method. This might lead to a new representation of the application:

$$Appl' \stackrel{\text{def}}{=} Appl/\{local\}$$

and a corresponding new representation of the system:

$$Sys_2 \stackrel{\text{def}}{=} \big(Appl' \parallel Appl'\big) \underset{L}{\bowtie} WS \qquad L = \{request, serve, respond\}$$

Note that this is quite different from the CCS restriction operator which prevents actions of the given label from occurring.

Use of the hiding combinator has two implications. Firstly, it ensures that no components added to the model at a later stage can invoke this method of the application. Secondly, private actions are deemed to have no contribution to the performance measures being calculated and this might subsequently suggest simplifications to the model.

Throughout the simple example above we have used constants such as WS to associate names with behaviours. Using recursive definitions we have been able to describe components with infinite behaviours without the use of an explicit recursion operator.

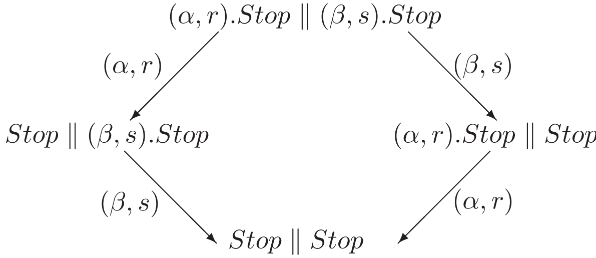
Representing the components of the system as separate components means that we can easily extend our model. Now we may want to consider a distributed system consisting of more than two applications which act independently of each other but compete for the use of web service. To enhance fault tolerance the web service may be replicated. This extension may be achieved compositionally by combining more instances of the components already described. For example, in the case of three applications and two instances of the web service we have:

$$Sys_3 \stackrel{\text{def}}{=} \big(Appl \parallel Appl \parallel Appl\big) \underset{L}{\bowtie} (WS \parallel WS) \qquad L = \{request, serve, respond\}$$

3.1 Designing the Language

Action durations. The selection of a negative exponential distribution as the governing distribution for the action durations in PEPA and other SPA has profound consequences. In terms of the underlying stochastic process, it is the only choice which gives rise to a Markov process. In terms of the process algebra it

is the only choice which preserves the well-known *expansion law* which underlies the interleaving semantics. In both cases this is due to the *memoryless* property of the exponential distribution: the time until the next event is independent of the time since the last event—the exponential distribution "forgets" how long it has already waited. Thus if we consider a process $(\alpha, r).Stop \parallel (\beta, s).Stop$, from the semantics we derive:



In a generally timed (or even deterministically timed) scenario it would be important to record the elapsed time in the intermediate states in order to know the residual time of the remaining activity. For example, the time needed to complete β in $Stop \parallel (\beta, s).Stop$ should reflect the time already taken to complete activity α. However the memoryless property of the exponential distribution tells us that the distribution of the residual time in β is the same as it was initially in state $(\alpha, r).Stop \parallel (\beta, s).Stop$ before any time had elapsed. Thus we retain the expansion law of classical process algebra:

$$(\alpha, r).Stop \parallel (\beta, s).Stop = \\ (\alpha, r).(\beta, s).(Stop \parallel Stop) + (\beta, s).(\alpha, r).(Stop \parallel Stop)$$

Later formalisms which incorporated general distributions either avoided the issue of residual durations by separating actions and delays (e.g. Modest [33] and IGSMP [34]), or used a finer-grained semantics such as ST-semantics to distinguish the start and stop of each action (e.g. GSMPA [35]).

Another major difference between the SPA formalisms concerns immediate or instantaneous actions. EMPA has immediate actions, modelled after the immediate transitions of GSPN. Each immediate action has an associated priority level and an associated weight. Immediate actions always have higher priority than exponentially timed actions, so a choice between such actions is resolved by priorities. If two immediate actions of the same priority level are concurrently enabled, the choice is resolved on the basis of their associated weights. The inclusion of immediate actions in TIPP, in addition to those with an associated exponentially distributed delay, has also been investigated. These actions were used to model logical [36] or control activities [37]. In these cases it was assumed that the environment of the component will resolve choices, but this opens the possibility that a model may contain non-determinism. Such a model is considered to be under-specified.

Cooperation. Communication or parallel composition is the essence of compositionality in process algebras. It gives structure to models, indicating which actions may be undertaken concurrently, and which cannot.

For most SPA the choice was made to adopt the multiway synchronisation using shared names (as in CSP) rather than complementary actions (as in CCS). This means that components or agents jointly perform actions of the same type, when the parallel composition dictates it. The motivation was to represent something more general than communication. In performance models interaction often captures resource usage and the objective of the model is to study the constraints imposed on components by competition over resources. In this context the multiway synchronisation offered more generality. However this choice was independent of the quantification of action durations, as witnessed by the adoption of CCS-style synchronisation in the stochastic π-calculus which generates a Markov process in the same way as PEPA [31].

Nevertheless the quantification of action duration did pose a challenge for the definition of cooperation. Actions which are to be performed jointly may each have been assigned rates (durations) in their respective components. The best way to resolve what should be the rate of the shared action has been a topic of some debate. The differing solutions adopted have become the main distinguishing feature of the various SPA formalisms.

The first observation is that if we view the joint action as a "synchronisation" as in the sense of *barrier synchronisation* in parallel programming then the correct duration would be the maximum of the durations, i.e. the maximum of the random variables. The unfortunate problem is that the maximum of two or more exponentially distributed random variables is not exponentially distributed.

In PEPA it is assumed that each component has *bounded capacity* to carry out activities of any particular type, determined by the *apparent rate*. For a component P and action type α, the apparent rate of α in P, denoted $r_\alpha(P)$, is the sum of the rates of each α action enabled in P. This corresponds to the rate at which P appears to an external observer to carry out an α action, due to the superposition principle of the negative exponential distribution. The definition of cooperation in PEPA is based on the assumption that a component cannot be made to exceed its bounded capacity, meaning that the apparent rate of the shared action will be the minimum of the apparent rates of the components involved.

In TIPP the "rate" is assumed to represent work capacity in one partner of the synchronisation and work demand in the other. The rate of the shared action is then taken to be the product of the two component rates. In contrast, in EMPA it is assumed that in any synchronisation exactly one participant has an explicit representation for the rate of the activity, all other participants being *passive* with respect to this activity, prepared to proceed at the rate of the active participant. This scheme does satisfy the principle of bounded capacity but the restriction has implications for the compositionality of the language. The formalisms which separate action and time evolution avoid this issue by only allowing synchronisation on untimed actions. The issue of timed synchronisation is discussed in [38,39] and in detail in Bradley's thesis [40].

4 Model Analysis

The formality of the process algebra approach allows us to assign a precise meaning to every language expression. This implies that once we have a language description of a given system its behaviour can be deduced automatically. The meaning, or semantics, of a PEPA expression is provided by SOS rules as for CCS, which associates a labelled multi-transition system with every expression in the language [29].

A labelled transition system $(S, T, \{\xrightarrow{t} \mid t \in T\})$ consists of a set of states S, a set of transition labels T and a transition relation $\xrightarrow{t} \subseteq S \times S$. For PEPA the states are the syntactic terms in the language, the transition labels are the actions ($(type, rate)$ pairs), and the transition relation is given by the semantic rules. A multi-transition relation is used because the number of instances of a transition (action) is significant since it can affect the timing behaviour of a component.

Based on the transition relation, a transition diagram, called the *derivation graph* (DG), can be associated with each language expression. This graph describes all the possible evolutions of any component and provides a useful way to reason about the behaviour of a model. A certain amount of care is needed in defining the derivation graph. Consider a simple component, P, which will repeatedly carry out the action $a = (\alpha, r)$, i.e. $P \stackrel{\text{def}}{=} (\alpha, r).P$. For a classical process algebra we need only consider which actions it is possible for an agent to perform. Thus, the agent $P + P$ has the same behaviour as the agent P—both are capable of an α named action and subsequently behave as P—so these agents are considered to be equivalent. In a SPA multiple instances of an action become apparent because the duration of an action of that type will be the minimum of the corresponding random variables, i.e. the apparent rate of the action will be the sum of the rates. Thus $P + P$ appears to carry out the first α named action at twice the rate of the agent P. Consequently the two cannot be regarded as equivalent.

Alternative solutions have been offered for this problem. In TIPP and EMPA supplementary labels are used to distinguish instances of multiply enabled actions, and the underlying structure is still a labelled transition system. In PEPA the semantics of the language is given in terms of a labelled multi-transition system with the transition relation represented as a multi-relation in which the multiplicities of arcs are recorded.

An example derivation graph is shown in Figure 4 where the DG of the PEPA model Sys_4, consisting of a single application accessing the web service, is shown. For didactic purposes, in the left hand part of the figure we have expanded the derivatives of the components *Appl* and WS.

Inspection of the DG allows one to derive qualitative properties of the model. In this case, for instance, we can see that the PEPA model is free from deadlock and live. Moreover, the Markov process underlying any finite PEPA component can be obtained directly from the DG: a state of the Markov process is associated with each node of the graph and the transitions between states are defined by

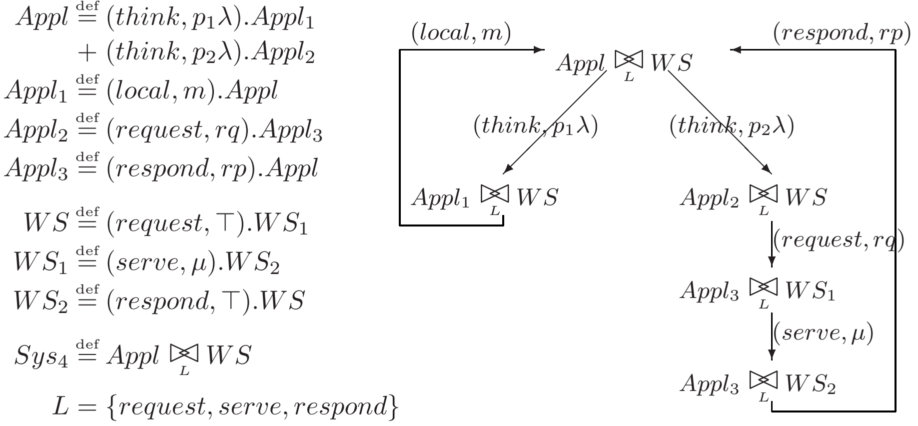


Fig. 4. Derivation graph underlying Sys_4

considering the rates labelling the arcs. Since all activity durations are exponentially distributed, the total transition rate between two states will be the sum of the activity rates labelling arcs connecting the corresponding nodes in the DG. Starting from the DG of Figure 4, the derivation of the corresponding Markov process is straightforward and results in the generator matrix shown below.

$$\boldsymbol{Q} = \begin{pmatrix} -\lambda & p_1\lambda & p_2\lambda & 0 & 0 \\ m & -m & 0 & 0 & 0 \\ 0 & 0 & -rq & rq & 0 \\ 0 & 0 & 0 & -\mu & \mu \\ rp & 0 & 0 & 0 & -rp \end{pmatrix}$$

Once obtained, the infinitesimal generator matrix can be used for a variety of different analysis techniques. Most commonly the model is subjected to steady state analysis. This assumes that the Markov process will eventually reach a regular pattern of behaviour and the probability distribution over the states of the model will cease to change, i.e. that the Markov process is *ergodic*. For such models the steady state probability distribution can be derived and reveals much information about the steady state, or equilibrium, behaviour of the model. In addition, any Markov process (both ergodic and not) can be subjected to transient analysis. In its simplest form a transient analysis will derive the state probability distribution for a given starting state and after a given time. However, it is also the basis of more sophisticated analyses such as calculating first passage and response time distributions.

In order to ensure that the Markov process underlying a PEPA model is ergodic, the DG of a PEPA model must be strongly connected. Necessary conditions for ergodicity, at the syntactic level of a PEPA model, have been defined [29]. For example, if cooperation occurs it must be the highest level combinator.

The class of PEPA terms which satisfy these syntactic conditions are termed *cyclic components* and they can be described by the following grammar:

$$P ::= S \mid P \underset{L}{\bowtie} P \mid P/L$$
$$S ::= (\alpha, r).S \mid S + S \mid A$$

All the models we have discussed so far satisfy the syntactic conditions required to be cyclic models.

It is well known that if the Markov process is ergodic, it is possible to compute the steady state probability distribution over all the possible states by solving the matrix equation $\pi \boldsymbol{Q} = 0$ where $\boldsymbol{Q}$ is the generator matrix of the Markov process and π is the state probability vector, such that $\sum_i \pi_i = 1$.

The probability distribution of the states of the model is often not the ultimate goal of performance analysis. Performance measures such as throughput and utilisation are often derived via a *reward structure* which is defined over the Markov process. This can either be done explicitly by the modeller, or as we will see, automatically by the tool for commonly required measures. A reward structure associates a value or *reward* with each state of the model. For steady state measures, the expected value of the reward (i.e. the sum over the entire state space of (probability of a state $\times$ reward in that state)) is calculated. In a process algebra it can be easier to associate rewards with actions. In this case the reward associated with a state will be the total reward attached to the actions that the state enables. Note that in PEPA no reward can be attached to internal, τ, actions.

4.1 Case Studies

As originally intended, PEPA has been applied to study the performance characteristics of a number of computer and communication systems. Initial examples focussed on well-known standard performance evaluation abstractions such as *multi-server multi-queue* systems [41] and various queueing systems [4]. However over time more realistic case studies emerged, both from the PEPA group and from others. For example, in [42] the performance impact of fault-tolerant protocols within a distributed system framework is evaluated. In [5] Bowman *et al.* develop a model of multimedia traffic characteristics and use it to derive quality of service measures such as jitter, throughput and latency. In an investigation of ways in which to ease the development of parallel database systems, the STEADY group at the Heriot-Watt University proposed the use of *performance estimators*. PEPA was used to verify the output of the performance estimators for a number of particular configurations and therefore improve confidence in the approach [43].

In recent work a group at the PRiSM Laboratory of the University of Versailles are working on a novel active rule-based approach to active networks (networks in which intermediate nodes supplement routing of data with some computation) [44]. A PEPA model was used to study the impact of the "*active*" traffic on the non-active cross-traffic in terms of loss rate and latency within an active switch

[45]. Furthermore the models were validated against simulation models of the same system and showed very good agreement [46].

In addition, the formalism has been applied to a number of other problems which are beyond the usual arena of computer performance evaluation.

Inland shipping. Luk Knapen of Hasselt applied PEPA to study traffic flow within the inland shipping network of Belgium focussing in particular on the locks and movable bridges.

Robotic workcells. Robert Holton of the University of Bradford used PEPA models to analyse the performance and functional correctness of a robotic workcell designed for a automated manufacturing system [3,2].

Cellular telephone networks. A team from the PRiSM Laboratory at the University of Versailles considered a problem of dimensioning in a cellular telephone network. They used a PEPA model to study the impact on call blocking and dropping of allocating bandwidth resources between micro and macro-cell level [8]. They took advantage of automatic aggregation [47].

Automotive diagnostic expert systems. Console *et al.* of the University of Turin constructed a PEPA model of an automatic diagnostic system to be deployed in a car. A large number of sensors were placed around the car and some number could trigger an alarm. The role of the PEPA model was to provide probabilistic reasoning to resolve the likely cause of the alarm based on previous observations of the timing and frequency of individual faults [48].

5 Tool Support

Case studies of the size and complexity described above are only possible if the modelling process has adequate support. In this section we describe some of the tool support which is available for performance modelling using stochastic process algebras. We focus primarily on the tools which support PEPA and the analysis techniques that they offer. There is a brief discussion of other SPA tools at the end of the section.

5.1 PEPA Tools

The PEPA Plugin Project. The PEPA Plugin Project is a software tool for reasoning about the various stages of the Markovian analysis of PEPA models. The tool is implemented as a collection of plug-ins for Eclipse [49], an extensible integrated development environment for a large variety of programming and modelling languages such as Java, C++, Python and UML. This framework was chosen for three main reasons. First, Eclipse is a freely available product. Second, it is widely supported by a growing community of users and businesses. Third, it can run on a variety of platforms, as it is implemented in Java and the graphical library used for the user interface is available on many operating systems.

The functionalities of the tool are accessible both programmatically and through a more user-friendly graphical interface. In the remainder of this section we focus on the latter method. Resources of an Eclipse workspace can be

manipulated using two main classes of tools, *editors* and *views*. The former follow the traditional open-save-close cycle pattern. The latter are typically used to navigate resources, modify properties of a resource and provide additional information on the resource being edited.

The PEPA Plugin contributes an editor for the language and views which assist the user during the entire cycle of model development. Static analysis is used for checking the well-formedness of a model and detecting potential errors prior to inferring the derivation graph of the system. A well-formed model can be derived, i.e. the underlying Markov process is extracted and the corresponding state space can thus be navigated and filtered via the *State Space* view. Finally, the CTMC allows numerical steady-state analyses such as activity throughput and component utilisation.

Editor. A PEPA editor is opened for files in the Eclipse workspace which have the `pepa` extension. The editor provides a convenient way to run a parser which translates the model description in the PEPA language into an in-memory representation suitable for further processing. This form is represented graphically in the *AST* view by means of a hierarchical structure for the model. The in-memory model also acts as an intermediate form for converting PEPA models into external formats. In particular, the PEPA plugin project provides an exporter to EMF [50], the de-facto standard for data exchange within the Eclipse framework.

Static Analysis. Static analysis deals with checking the well-formedness of a PEPA model. Because of its low computational cost, static analysis is performed every time the text of the model description is saved. The output of this tool is a contribution of a list of messages to the already existing Eclipse *Problem* view. The information provided can be grouped into two categories: *warnings* are messages about low priority problems which do not prevent further processing; *errors* are instead critical problems which must be fixed in order to continue the model development process. For instance, basic warning messages are about rate or process definitions which are defined in the model description but never used; error messages can be about rate or process names which are used but never defined.[3]

More advanced static analysis is carried out to detect potential local deadlocks, redundant declaration of actions of the cooperation operator and unguarded component uses giving rise to non-well-founded definitions of processes, i.e. self-containing processes. In order to fulfill these tasks, the model's in-memory representation is iteratively walked to create two support data structures: *complete action types set* and *used definition set*.

The complete action type set $\boldsymbol{\mathcal{A}}$ of a component is the set of all the action types which can performed by the component during its evolution. This set can be calculated according to Tab. 2. For example, if we consider the model in Fig. 5 (for the sake of clarity we omit the actual rate values) then the complete action type sets of its constants are as follows:

[3] It is worthwhile noting that rate names must be declared before using them in a prefix definition. However, there is no such a rule with regards to process definitions.

$$\begin{aligned}\boldsymbol{\mathcal{A}}(P1) &= \{\alpha,\beta,\gamma,\delta\}\\ \boldsymbol{\mathcal{A}}(P2) &= \{\alpha,\beta,\gamma,\delta\}\\ \boldsymbol{\mathcal{A}}(P3) &= \{\alpha,\beta,\gamma,\delta\}\\ \boldsymbol{\mathcal{A}}(Q1) &= \{\alpha,\beta,\epsilon,\eta\}\\ \boldsymbol{\mathcal{A}}(Q2) &= \{\alpha,\beta,\epsilon,\eta\}\\ \boldsymbol{\mathcal{A}}(Q3) &= \{\alpha,\beta,\epsilon,\eta\}\end{aligned} \tag{5.1}$$

The used definition set $\boldsymbol{\mathcal{U}}$ of a component is the set of all the constants which the

Table 2. Rules for deriving the complete action type set

Constant $A \stackrel{\text{def}}{=} P$	$\boldsymbol{\mathcal{A}}(A) = \boldsymbol{\mathcal{A}}(P)$
Prefix $(\alpha, r).P$	$\boldsymbol{\mathcal{A}}\big((\alpha, r).P\big) = \{\alpha\} \cup \boldsymbol{\mathcal{A}}(P)$
Choice $P + Q$	$\boldsymbol{\mathcal{A}}(P + Q) = \boldsymbol{\mathcal{A}}(P) \cup \boldsymbol{\mathcal{A}}(Q)$
Cooperation $P \underset{L}{\bowtie} Q$	$\boldsymbol{\mathcal{A}}(P \underset{L}{\bowtie} Q) = \boldsymbol{\mathcal{A}}(P) \cup \boldsymbol{\mathcal{A}}(Q)$
Hiding $P\backslash\{L\}$	$\boldsymbol{\mathcal{A}}(P\backslash\{L\}) = \boldsymbol{\mathcal{A}}(P) - L$

component behaves as during its evolution. This set can be calculated according to the rules in Tab. 3. The used definition sets of the constants of the model in Fig. 5 are as follows:

$$\begin{aligned}\boldsymbol{\mathcal{U}}(P1) &= \{P1, P2, P3\}\\ \boldsymbol{\mathcal{U}}(P2) &= \{P1, P2, P3\}\\ \boldsymbol{\mathcal{U}}(P3) &= \{P1, P2, P3\}\\ \boldsymbol{\mathcal{U}}(Q1) &= \{Q1, Q2, Q3\}\\ \boldsymbol{\mathcal{U}}(Q2) &= \{Q1, Q2, Q3\}\\ \boldsymbol{\mathcal{U}}(Q3) &= \{Q1, Q2, Q3\}\end{aligned} \tag{5.2}$$

$$\begin{aligned}P1 &\stackrel{\text{def}}{=} (\alpha, r).P2 + (\beta, s).P3\\ P2 &\stackrel{\text{def}}{=} (\gamma, t).P1\\ P3 &\stackrel{\text{def}}{=} (\delta, u).P1\\ Q1 &\stackrel{\text{def}}{=} (\alpha, \top).Q2 + (\beta, \top).Q3\\ Q2 &\stackrel{\text{def}}{=} (\epsilon, v).Q1\\ Q3 &\stackrel{\text{def}}{=} (\eta, w).Q1\\ P1 &\underset{\{\alpha,\beta\}}{\bowtie} Q1\end{aligned}$$

Fig. 5. An example of PEPA model

Let us consider two processes which cooperate over a non-empty set of action types. A local deadlock is a condition that may occur when one process cannot proceed because it is in a state where it is synchronised on an activity which can never be performed by its partner. The model in Fig. 6 exhibits a local deadlock

Table 3. Rules for deriving the used definition set

Constant $A \stackrel{\text{def}}{=} P$	$\{P\} \cup \boldsymbol{\mathcal{U}}(P)$
Prefix $(\alpha, r).P$	$\boldsymbol{\mathcal{U}}(P)$
Choice $P + Q$	$\boldsymbol{\mathcal{U}}(P) \cup \boldsymbol{\mathcal{U}}(Q)$
Cooperation $P \bowtie_{L} Q$	$\boldsymbol{\mathcal{U}}(Q) \cup \boldsymbol{\mathcal{U}}(R)$
Hiding $P \backslash \{L\}$	$\boldsymbol{\mathcal{U}}(P)$

in the initial state, because the action type α cannot be performed by either *Q1* or *Q2*. Local deadlock conditions are critical errors which can be statically detected by examining the used definition set of each cooperation of a model. In particular, a cooperation $P \bowtie_{L} Q$ gives rise to a deadlock on the action α if the following condition holds:

$$\exists \alpha \in L : \alpha \notin \boldsymbol{\mathcal{A}}(P) \cap \boldsymbol{\mathcal{A}}(Q), \alpha \in \boldsymbol{\mathcal{A}}(P) \cup \boldsymbol{\mathcal{A}}(Q) \tag{5.3}$$

The tool emits warning messages if it discovers the existence of redundant definition of action types in cooperation sets. A cooperation specifies a redundant action type α if the following condition holds:

$$\exists \alpha \in L : \alpha \notin \boldsymbol{\mathcal{A}}(P) \cup \boldsymbol{\mathcal{A}}(Q) \tag{5.4}$$

The used definition set allows for the detection of non-guarded recursive definitions of components. In Fig. 7 is shown a model exhibiting such a condition. A subset of the infinite labeled transition system of component *P1* is:

$$\begin{aligned} P1 &\stackrel{(\gamma,t)}{\longrightarrow} P2 \parallel P3 \parallel P3 \\ &\stackrel{(\gamma,t)}{\longrightarrow} P2 \parallel P3 \parallel P3 \parallel P3 \\ &\stackrel{(\gamma,t)}{\longrightarrow} P2 \parallel P3 \parallel P3 \parallel P3 \parallel P3 \\ &\stackrel{(\gamma,t)}{\longrightarrow} \dots \end{aligned}$$

This gives rise to an infinite-state Markov chain. We wish to work with finite-state Markov chains so we reject definitions such as these as being ill-formed.

$$\begin{aligned} P1 &\stackrel{\text{def}}{=} (\alpha, r).P2 \\ P2 &\stackrel{\text{def}}{=} (\gamma, t).P1 \\ Q1 &\stackrel{\text{def}}{=} (\beta, s).Q2 \\ Q2 &\stackrel{\text{def}}{=} (\epsilon, v).Q1 \\ P1 &\bowtie_{\{\alpha\}} Q1 \end{aligned}$$

Fig. 6. Example of a PEPA model with local deadlock

$$\begin{aligned} &\dots \\ &\mathit{P1} \stackrel{\text{def}}{=} \mathit{P2} \parallel \mathit{P3} \\ &\mathit{P2} \stackrel{\text{def}}{=} (\gamma, t).\mathit{P1} \\ &\dots \end{aligned}$$

Fig. 7. Example of a PEPA model with non-guarded recursive definitions of components

The used definition set is calculated for each process constant $A \stackrel{\text{def}}{=} P$ which defines a cooperation (in the example, $\boldsymbol{\mathcal{U}}(\mathit{P1})$ would be calculated). Such a constant is not well-formed if the following condition holds:

$$A \in \boldsymbol{\mathcal{U}}(A) \tag{5.5}$$

The PEPA Plugin project provides a tool for state space derivation, i.e. the process of extracting a Markov process from the labeled transition system of the PEPA model. The output of the tool is the state space and the corresponding infinitesimal generator of the CTMC. The state space can be navigated and filtered via the *State Space* view. The state space is represented in a tabular form: the first column is the state number; then follow as many columns as the number of top-level components of the system. The tabular representation of the state space of the model in Fig. 5 would be as in Tab. 4.

Table 4. Tabular representation of the state space of the example model

State Number	First Component	Second Component
1	*P1*	*Q1*
2	*P3*	*Q3*
3	*P3*	*Q1*
4	*P1*	*Q3*
5	*P2*	*Q2*
6	*P2*	*Q1*
7	*P1*	*Q2*

A variety of filter options is available in order to narrow down the number of states shown in the view. The user can exclude/include states which have a sequential component in a particular local state or states which contain unnamed processes (i.e., prefixes). More precise filtering can be obtained by means of a pattern language which allows the user to match local states which contain given top-level component local states at specified positions. The components are separated by vertical bars and a wild-card is used to disregard positions which are of no interest. According to the example in Fig. 5, the pattern `P1 | *` would match the states whose first top-level component is in state P1,

thus displaying states 1,4,7; the pattern `* | Q2` would match states 5,7. For a more concise description of the filter, the generic pattern `P` is considered as an abbreviation of `P | * | ... | *`.

Additionally, the plug-in contributes the *Single Step Navigator*, a tool for walking the state space. This is particularly useful for debugging purposes. It consists of two tables containing the list of incoming and outgoing states. The sequential components which cause the transition to be performed are highlighted and an option allows the user to make filtered states not walkable.

A model whose state space is derived successfully is amenable to performance analysis which can be carried out by calculating the steady-state probability distribution of the CTMC over the state space. The user interface provides a dialogue wizard which guides the user through this process. The wizard is a graphical interface to the MTJ toolkit [51], the library used for the numerical solution of the Markov chain, allowing the user to choose and tune the parameters of an extensive selection of solvers and preconditioners.

After the model is successfully solved, the State Space view is updated with information on the obtained steady-state probability distribution which is shown on an additional column. Additional analysis can be carried out via the *Performance Evaluation* view, which permits throughput and utilisation analysis. Throughput is an action-related metric showing the rate at which an action is performed at steady-state; utilisation is related to a sequential component showing the steady-state distribution probability over its local states.

In order to better illustrate these metrics, let us consider the model in Fig. 8 consisting of one single sequential component evolving through three local states. With rates $r = 2$, $s = 1$, $t = 1$ the utilisation figures are $P1 = 0.2$, $P2 = 0.4$, $P3 = 0.4$ whereas throughput is 0.4 for each action.

$$
\begin{aligned}
\mathit{P1} &\stackrel{\text{def}}{=} (\alpha, r).\mathit{P2} \\
\mathit{P2} &\stackrel{\text{def}}{=} (\beta, s).\mathit{P3} \\
\mathit{P3} &\stackrel{\text{def}}{=} (\gamma, t).\mathit{P1}
\end{aligned}
$$

Fig. 8. A tiny PEPA model with one sequential component

Experimentation is a tool for sensitivity analysis. The user can supply ranges for rate values against which the performance metrics described above are calculated. Results are then shown in the form of graphs for which a number of exporting options are available.

The Imperial PEPA Compiler. The Imperial PEPA Compiler (IPC) [52] provides an alternative implementation of the PEPA language, providing a bridge to performance analysis tools developed at Imperial College by Knottenbelt and his group [53,54].

The ipc tool translates an input PEPA model into the Petri net notation provided by Dnamaca [53]. Its support for the PEPA language is comprehensive.

Apparent rates are supported, as are anonymous components. The great advantage of accessing the functionality of the Dnamaca analyser is that other forms of analysis (beyond steady-state) become available.

The steady state probability distribution represents the behaviour of the system at equilibrium, where the influence of the initial state of the system is no longer measurable. Some performance measures of interest cannot be derived from the results of steady state analysis. Examples of performance measures in the class of non-equilibrium measurements include *mean time to failure* analysis, as computed in the evaluation of dependable systems. Other examples include the probabilistic quality-of-service guarantees which underpin most commercial service level agreements (SLAs): e.g. the probability that a 10-node cluster should be able to process 3000 database transactions in less than 6 seconds should be greater than 0.915; or a train service should not run more than 10 minutes late more than 20% of the time.

More generally, such measures necessitate the computation of *passage-time quantiles* which detail the probability of passing through the system evolution from a start state to an end state (or a set of starting states to a set of end states). The computation of such measures depends on the aggregate time behaviour across a whole system of complex interactions. The computation of passage-time quantiles depends on *transient analysis* of the CTMC, which is more expensive than steady-state analysis in both run-time and memory consumption.

Via ipc, the unique solution capabilities of Dnamaca become available and because of this it is possible to efficiently perform passage time analysis over PEPA models [52,54]. Start and end points are specified using the concept of *stochastic probes* developed by Argent-Katwala, Bradley and Dingle [55]. Stochastic probes are themselves PEPA components which have been generated from regular expression-based inputs.

The PRISM model checker. PRISM is a probabilistic model checker developed by Kwiatkowska's group at the University of Birmingham. It supports discrete time Markov chains and Markov decision processes as well as CTMCs. The standard input to PRISM is a model described in a simple reactive modules language. PEPA was integrated into the tool via a compiler which translates PEPA models into this language. The developers at the University of Birmingham extended PRISM's modelling capabilities to implement at the binary decision diagram level PEPA's combinators (cooperation and hiding).

Integration into PRISM enables model checking of the CTMC underlying a PEPA model against properties expressed in Continuous Stochastic Logic (CSL) [56]. It also provides access to the efficient numerical solutions of PRISM based on MTBDDs [57] and sparse matrix representation. PRISM has been applied successfully to a number of PEPA (and PEPA net) case studies [58,59].

The Möbius modelling platform. The Möbius modelling framework [60] was developed at the University of Illinois Urbana-Champaign. It is both a multi-formalism and multi-paradigm modelling tool, i.e. it aims to offer the user a choice of model description techniques and solution methods. Moreover it is

designed to allow a model to be composed of submodels which may be expressed in different formalisms. It has a broad spectrum of users in North America. Integrating PEPA into Möbius offered opportunities to present stochastic process algebra to users who were previously unfamiliar with the formalism, and to expore the possibilities of interaction between modelling formalisms [61].

5.2 Related Work

Over the years, several software tools have been made available for supporting computer-aided analysis with process algebra. TwoTowers [62], for example, provides a similar range of tools for the stochastic process algebra EMPAgr.

TIPP-Tool. The TIPP-Tool is a prototype modelling tool for creating and evaluation TIPP models of parallel and distributed systems. It supports a LOTOS-oriented input language and as well as facilities to apply functional analysis based on reachability analysis, it provides a set of numerical solution modules for the stationary and transient analysis of the Markov process underlying a TIPP specification [63,64].

In order to evaluate the performance of the specification the derived transition system serves as a base for further reduction into a Markov process. For the steady state analysis of the underlying Markov process a variety of numerical algorithms are available: LU-factorization, power method, and Gauss-Seidel iteration scheme. TIPP-Tool supports also transient analysis by providing methods to compute the mean time to absorption of an absorbing Markov chain or the transient state probabilities. For the latter a refined randomisation scheme is provided.

The result of numerical analysis is usually a vector with state probabilities. In order to obtain more sophisticated and expressive results the user can specify measures. This is done via rewards that are assigned to states which match a regular expression which the user must specify. Series of experiments are also supported by allowing rates to be symbolic variables. The specification of the model, as well as the measures and experiments, is supported through a graphical user interface.

TwoTowers. TwoTowers [62], which supports modelling with the SPA language EMPA, builds on two existing tools, CWB-NC [65] for functional analysis and MARCA [66] for performance analysis. The specification language for TwoTowers is EMPA_r, an extension of EMPA to include the specification of rewards—in the subsequent analysis these rewards are used to derive performance measures. The other SPA tools include the facility to associate a reward structure with a model; in EMPA_r the reward structure is assumed to be an integral part of the model.

TwoTowers has a graphical user interface written in Tcl/Tk. This interface allows the user to edit and compile specifications and provides access to the various analysis routines. CWB-NC provides a suite of functional analysis techniques: model checking, equivalence checking, preorder checking and reachability analysis. MARCA provides for both steady state and transient performance analysis of the underlying Markov process. In addition there is a simulation engine which allows models to be simulated.

More information about TwoTowers is available at:
(`http://www.sti.uniurb.it/bernardo/twotowers`).

MoDeST. The MoDeST modelling language (Modelling and Description language for Stochastic Timed systems) [33] enriches a process algebra with atomic statements to control the granularity of transitions, non-deterministic and probabilistic branching and timing. The MoDeST language provides conventional programming constructs such as iteration, alternatives, atomic statements, and exception handling in the style of user-friendly specification languages such as Promela. The MoDeST semantics maps each MoDeST specification onto a *stochastic timed automata*, a modelling formalism which subsumes timed, stochastic and probabilistic automata.

The MoDeST language has been integrated into the Möbius multi-paradigm modelling framework [67] as an atomic model. MoDeST models which do not use non-determinism can be assessed quantitatively using the discrete-event simulator of Möbius or its Markovian analysers. MoDeST models are mapped onto C++ code which links against the Möbius Abstract Functional Interface (AFI).

Verification of properties of MoDeST models can be performed using the CADP Tools [68].

6 Case Studies

6.1 Roland the Gunslinger

In this subsection we consider a sequence of small examples based around a character known as "Roland the Gunslinger". These simple models are intended to be intuitive to understand but yet show some of the main features of the language and demonstrate a variety of solution techniques. A more substantial example is presented in the following subsection.

Roland alone. Roland Deschain is a gunslinger and his primary activity is firing his gun which is a six-shooter, i.e. there is room in the barrel for six bullets at a time. When his gun is empty Roland will reload the gun and then continue shooting.

$$
\begin{aligned}
Roland_6 &\stackrel{\text{def}}{=} (fire, r_{fire}).Roland_5 \\
Roland_5 &\stackrel{\text{def}}{=} (fire, r_{fire}).Roland_4 \\
Roland_4 &\stackrel{\text{def}}{=} (fire, r_{fire}).Roland_3 \\
Roland_3 &\stackrel{\text{def}}{=} (fire, r_{fire}).Roland_2 \\
Roland_2 &\stackrel{\text{def}}{=} (fire, r_{fire}).Roland_1 \\
Roland_1 &\stackrel{\text{def}}{=} (fire, r_{fire}).Roland_{empty} \\
Roland_{empty} &\stackrel{\text{def}}{=} (reload, r_{reload}).Roland_6
\end{aligned}
$$

If we suppose that Roland has two guns then he should be allowed to fire either gun independently. A simplistic way to model this is to have two instances of *Roland* in parallel:

$$Roland_6 \parallel Roland_6$$

However, this model does not capture the fact that Roland needs both hands in order to reload either gun. The simplest way to fix this is to assume that Roland only reloads both guns when both are empty.

$$Roland_6 \underset{\{reload\}}{\bowtie} Roland_6$$

In the remaining models, we consider only the case of Roland using his shotgun, which has only two bullets before it needs reloading, and requires both hands for firing.

Choice. In the first straightforward model of Roland, he was simply firing his guns. We now consider a model which captures the possibility that Roland will miss or hit his target.

Upon his travels Roland will encounter some enemies with whom he will have no choice but to enter combat. In this model it is assumed that his enemies do not possess the skill required to seriously harm Roland. Therefore he never dies but simply encounters villians and fires at them until he successfully hits them. Each hit is assumed to be fatal and it is assumed that a sense of honour prevents an enemy from attacking Roland if he is already involved in a gun fight.

The rates involved in this model are given in Table 5; each is measured in seconds, so a rate of 1.0 would indicate that the action is expected to occur once every second. There is one special parameter, $p_{hit\text{-}success}$ which is a probability measure, used to calculate the values for the rates r_{hit} and r_{miss}.

$$\begin{aligned}
Roland_{idle} &\stackrel{\text{def}}{=} (attack, r_{attack}).Roland_2 \\
Roland_2 &\stackrel{\text{def}}{=} (hit, r_{hit}).(reload, r_{reload}).Roland_{idle} + (miss, r_{miss}).Roland_1 \\
Roland_1 &\stackrel{\text{def}}{=} (hit, r_{hit}).(reload, r_{reload}).Roland_{idle} + (miss, r_{miss}).Roland_{empty} \\
Roland_{empty} &\stackrel{\text{def}}{=} (reload, r_{reload}).Roland_2
\end{aligned}$$

Table 5. Parameter settings for the $Roland_2$ model

parameter	*value*	*explanation*
r_{fire}	1.0	Roland can fire the gun once per-second
$p_{hit\text{-}success}$	0.8	Roland has an 80% success rate
r_{hit}	0.8	$r_{fire} \times p_{hit\text{-}success}$
r_{miss}	0.2	$r_{fire} \times (1 - p_{hit\text{-}success})$
r_{reload}	0.3	It takes Roland about 3 seconds to reload
r_{attack}	0.01	Roland is attacked once every 100 seconds

Steady-State Analysis. This model can be used to calculate the probability that at any time Roland is involved in a battle. Using steady state analysis this amounts to calculating the probability that *Roland* is in any of the states in which a battle is on-going, i.e. $Roland_2$, $Roland_1$ and $Roland_{empty}$. Alternatively one can calculate the probability that *Roland* is in the single peaceful state $Roland_{idle}$ and subtract it from 1. This was done for the above model, for the parameter values shown in Table 5, giving the result:

```
State Measure 'roland peaceful' % 100 seconds

 mean          9.5490716180e-01
```

This shows that there is more than a 95 percent chance that Roland is not currently involved in a gun battle. This is intuitively what we would expect since he is attacked once every 100 seconds and will usually take around one second to fire each bullet. Two bullets then cost him a further three seconds to reload, however since his success rate is at 80 percent, he will often not need to reload.

Transient Analysis. Transient analysis could be used here to determine the probability that Roland will have killed some enemy within a given time, say two minutes, of starting off on his travels.

Passage-Time Analysis. An example of a passage-time analysis for this model would calculate the probability that at a given time after he is attacked, Roland has killed his attacker. This would involve calculating the probability that the model performs a *hit* action within the given time after performing an *attack* action.

The graph on the left hand side of Figure 9 shows an example of this kind of analysis. It shows the probability that Roland will successfully perform a *hit* action a given time after an *attack* action. This also confirms our instincts concerning the steady-state analysis. Since there is a 95 percent chance that Roland is not involved in a gun battle, and one occurs about once every 100 seconds, then we should expect gun battles to last for around five seconds. Looking at the graph on the left hand side of Figure 9 we see that the probability that Roland has performed a *hit* action five seconds after an *attack* action is quite high at just over 90 percent.

We can also measure, for example, the probability that Roland will miss after having been attacked. This probability is somewhat low. One of the reasons is that, if Roland hits the target with his first shot then in order to observe a miss action in the model we will have to wait until Roland is attacked again. The graph on the right hand side of Figure 9 shows the probability curve for the same time period as the first graph. Because the attack rate is low, in this period of time it is unlikely that Roland will be attacked for a second time. For this reason the graph looks similar to the first graph, but translated down the probability axis. The initial rise in probability corresponds, as in the first graph, to the probability that Roland will fire his gun within that time.

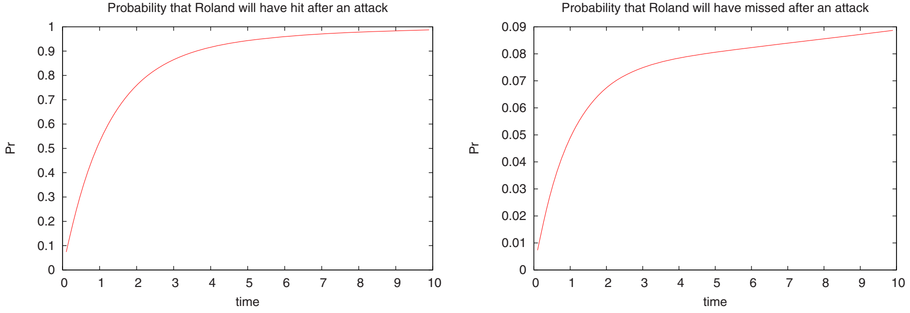


Fig. 9. Probability of events occurring after an attack event

Cooperation. We now consider the model augmented to allow the enemies of Roland to fight back. However, they are currently somewhat ineffective and always miss Roland when they fire. (The next model will fix this.) This model can be used to calculate properties such as the likelihood that an enemy will manage to fire one shot before they are killed by Roland.

Table 6. Parameters for the enemies

parameter	*value*	*explanation*
r_{attack}	0.01	Roland is attacked once every 100 seconds
$r_{e\text{-}miss}$	0.3	Enemies can fire only once every 3 seconds

$$
\begin{aligned}
Roland_{idle} &\stackrel{\text{def}}{=} (attack, \top).Roland_2 \\
Roland_2 &\stackrel{\text{def}}{=} (hit, r_{hit}).(reload, r_{reload}).Roland_{idle} + (miss, r_{miss}).Roland_1 \\
Roland_1 &\stackrel{\text{def}}{=} (hit, r_{hit}).(reload, r_{reload}).Roland_{idle} + (miss, r_{miss}).Roland_{empty} \\
Roland_{empty} &\stackrel{\text{def}}{=} (reload, r_{reload}).Roland_2
\end{aligned}
$$

$$
\begin{aligned}
Enemies_{idle} &\stackrel{\text{def}}{=} (attack, r_{attack}).Enemies_{attack} \\
Enemies_{attack} &\stackrel{\text{def}}{=} (fire, r_{e\text{-}miss}).Enemies_{attack} + (hit, \top).Enemies_{idle}
\end{aligned}
$$

$$
Roland_2 \underset{\{hit\}}{\bowtie} Enemies_{idle}
$$

Notice that in this model the behaviour of the enemy has been simplified. There is no running out of bullets or reloading. This model can be thought of as an approximation to a more complicated component similar to the one which models Roland. The rate at which the enemy fires encompasses all of the actions, including the reloading of an empty gun. The analyses associated with this model are very similar to those for the previous model. Steady-state analysis can be used to determine the likelihood that Roland is currently in a peaceful state.

It is also sometimes useful to carry out a validation of the model by calculating a metric which we believe we already know the value of. For example in this model we could make such a sanity check by calculating the probability that the model is in a state in which Roland is idle but the enemies are not, or vice versa. This should never occur and hence the probability should be zero.

Transient analysis could again be used to calculate the expected time before Roland is attacked or the expected time before Roland has made a kill.

Sensitivity Analysis. Due to the roles which activities play in creating the dynamics of our stochastic process algebra model it may be that increasing the rate of one activity increases the score obtained by the model on our chosen performance measure of interest. Conversely, increasing the rate of another activity may decrease the score which we get. Changing one rate a little may vary the score a lot. Changing another rate a lot might only vary the score a little. The study of how changes in performance depend on changes in parameter values in this way is known as *sensitivity analysis*.

Sensitivity analysis is performed by solving the model many times while varying the rates slightly. For this model we chose to vary three of the rates involved and measured for each combination of rates the passage-time probability that the model performs a *hit* action after performing an *attack* action.

The results are shown in Figure 10. The first three graphs measure the sensitivity of each of the three rates. The top left graph shows the effect that varying the $p_{hit\text{-}success}$ parameter has. The top right graph depicts the effect of varying the r_{fire} rate and finally the middle left graph shows the r_{reload} rate.

From these graphs one can deduce that the strongest influence is from the $p_{hit\text{-}success}$ parameter. To see this, notice the greater curvature of the graph of probability against time as the value of the $p_{hit\text{-}success}$ parameter is increased. In contrast, the top right and middle left graphs show little of the warping that is seen in the first graph.

In the final three graphs we measure the effect that varying one rate has, on the effect of another rate. In most models the effect which one rate has depends on the values of the other rates. For example, in our model, clearly if both the r_{reload} and r_{fire} rates are small then the effect of the $p_{hit\text{-}success}$ is large since Roland pays a large penalty whenever he misses. If, however, these two rates are large then Roland pays less of a penalty for missing and hence the effect of increasing (or decreasing) $p_{hit\text{-}success}$ is diluted.

To keep such graphs comprehensible to humans we fix the time at which the probability is measured. The first of these graphs on the middle right of Figure 10 measures the effect of varying r_{reload} against $p_{hit\text{-}success}$. Similarly the bottom left graph depicts varying r_{reload} against r_{fire}; and finally, the bottom left varies r_{fire} and $p_{hit\text{-}success}$. This final graph is interesting. On the far left it can be seen that when the r_{fire} is low, as we increase $p_{hit\text{-}success}$ there is close to a linear increase in the probability. However, when the r_{fire} is high the graph of probability against $p_{hit\text{-}success}$ rises sharply and then becomes less steep. This is most likely because when the r_{fire} rate is high, the penalty for Roland reloading is also relatively high in comparison and therefore the benefit of avoiding this is greater.

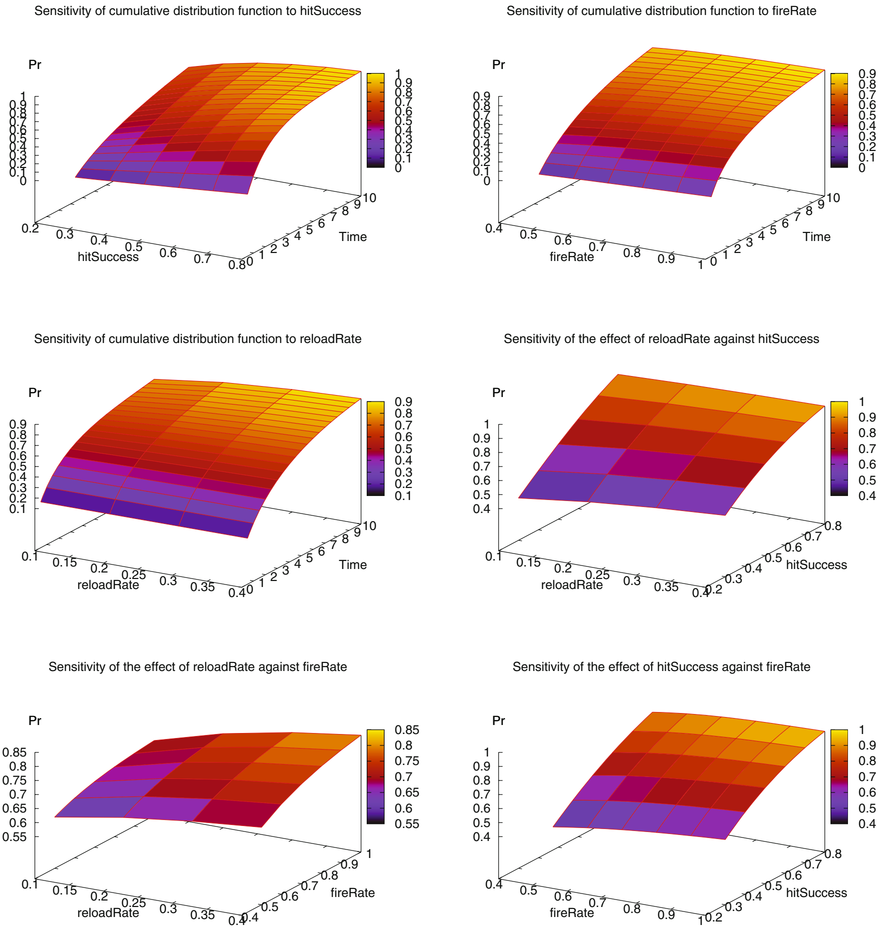


Fig. 10. Graphs of cumulative distribution function sensitivity to changes in rates for the passage from attack to Roland killing the enemy

Accurate Enemies. We now allow the enemies of Roland to actually hit him. This means that Roland may die. It is important to note that this has the consequence that the model will always deadlock. The underlying Markov process is no longer ergodic.

To maintain the simplicity of the model we assume that the enemies can only hit Roland once every 50 seconds. Note that this rate approximates the rate of a more detailed model in which we would assign a process to the enemies which is much like that of the process which describes Roland. That is, it can fire and miss, run out of bullets and reload etc. before finally hitting Roland. The only new parameter is $r_{e\text{-}hit}$ which is assigned a value 0.02 to reflect this assumption.

$$
\begin{aligned}
Roland_{idle} &\stackrel{\text{def}}{=} (attack, \top).Roland_2 \\
Roland_2 &\stackrel{\text{def}}{=} (hit, r_{hit}).(reload, r_{reload}).Roland_{idle} + (miss, r_{miss}).Roland_1 \\
&\quad + (e\text{-}hit, \top).Roland_{dead} \\
Roland_1 &\stackrel{\text{def}}{=} (hit, r_{hit}).(reload, r_{reload}).Roland_{idle} + (miss, r_{miss}).Roland_{empty} \\
&\quad + (e\text{-}hit, \top).Roland_{dead} \\
Roland_{empty} &\stackrel{\text{def}}{=} (reload, r_{reload}).(reload, r_{reload}).Roland_2 + (e\text{-}hit, \top).Roland_{dead} \\
Roland_{dead} &\stackrel{\text{def}}{=} Stop
\end{aligned}
$$

$$
\begin{aligned}
Enemies_{idle} &\stackrel{\text{def}}{=} (attack, r_{attack}).Enemies_{attack} \\
Enemies_{attack} &\stackrel{\text{def}}{=} (e\text{-}hit, r_{e\text{-}hit}).Enemies_{idle} + (hit, \top).Enemies_{idle}
\end{aligned}
$$

$$
Roland_2 \underset{\{hit, attack, e\text{-}hit\}}{\bowtie} Enemies_{idle}
$$

Steady-State Analysis. This model has the interesting property that the model will always deadlock: because there is an infinite supply of enemies eventually Roland will always die. This means that steady-state analysis would not be used on such a model, although a possible use would be as a validation of the model, as was done for the previous model.

Transient Analysis. Transient analysis on this model can be used to calculate the expected time at which Roland will die, or rather the probability that Roland is dead after a given amount of time. As the time increases this should tend towards probability 1.

Passage-Time Analysis. As before, the passage-time analysis on this model would be used to calculate the probability of a given event happening at a given time after another given event. Here we might again choose the starting event to be an attack on Roland, and the ending event could be either Roland dying or Roland winning the gun fight.

More Cooperation. The cooperation so far has involved the synchronisation between two processes on events that they have either caused directly or are directly affected by. In this section cooperation is used to synchronise between components of the model such that they observe events which they neither directly cause nor are directly affected by. In this particular example an accomplice is befriended by Roland from time to time and whenever an enemy attacks, Roland and the accomplice fight together. Whenever either one of them kills the enemy the other must observe this action, so as to stop firing at a dead opponent.

The component representing Roland is now modified to include actions, which Roland does not participate in, such as his accomplice killing the enemy, but which nevertheless alter Roland's state and therefore must be witnessed.

Table 7. Parameter values for the accomplice

parameter	*value*	*explanation*
$r_{befriend}$	0.001	Roland befriends a stranger once every 1000 seconds
$r_{a\text{-}fire}$	1.0	the accomplice can also fire once per second
$p_{a\text{-}hit\text{-}success}$	0.6	the accomplice has a 60 percent accuracy
$r_{a\text{-}hit}$	0.6	$r_{fire} \times p_{hit\text{-}success}$
$r_{a\text{-}miss}$	0.4	$r_{fire} \times (1.0 - p_{hit\text{-}success})$
$r_{a\text{-}reload}$	0.25	it takes the accomplice 4 seconds to reload

$$
\begin{aligned}
Roland_{idle} &\stackrel{\text{def}}{=} (attack, \top).Roland_2 + (befriend, r_{befriend}).Roland_{idle} \\
Roland_2 &\stackrel{\text{def}}{=} (hit, r_{hit}).Roland_{hit} + (a\text{-}hit, \top).Roland_{idle} + (miss, r_{miss}).Roland_1 \\
Roland_1 &\stackrel{\text{def}}{=} (hit, r_{hit}).Roland_{hit} + (a\text{-}hit, \top).(reload, r_{reload}).Roland_{idle} \\
&\quad + (miss, r_{miss}).Roland_{empty} \\
Roland_{hit} &\stackrel{\text{def}}{=} (enemy\text{-}die, \top).(reload, r_{reload}).Roland_{idle} \\
Roland_{empty} &\stackrel{\text{def}}{=} (reload, r_{reload}).Roland_2 + (a\text{-}hit, \top).(reload, r_{reload}).Roland_{idle}
\end{aligned}
$$

The attack is assumed to be a concerted effort between Roland and his accomplice but we do not wish to leave Roland vulnerable when he has no accomplice. For this reason the representation of the accomplice includes a state when the accomplice is absent. In this state the accomplice component will passively participate in any attack which Roland makes. The alternative would be that Roland was blocked from attacking when he had no accomplice. Also note that, just as Roland witnesses if the accomplice kills the enemy, the accomplice also witnesses if Roland kills the enemy.

$$
\begin{aligned}
Accomplice_{abs} &\stackrel{\text{def}}{=} (befriend, r_{befriend}).Accomplice_{idle} + (hit, \top).Accomplice_{abs} \\
&\quad + (attack, \top).Accomplice_{abs}
\end{aligned}
$$

$$
\begin{aligned}
Accomplice_{idle} &\stackrel{\text{def}}{=} (attack, \top).Accomplice_2 \\
Accomplice_2 &\stackrel{\text{def}}{=} (a\text{-}hit, r_{a\text{-}hit}).Accomplice_{hit} + (hit, \top).Accomplice_{idle} \\
&\quad + (miss, r_{miss}).Accomplice_1 + (enemy\text{-}hit, \top).Accomplice_{abs} \\
Accomplice_1 &\stackrel{\text{def}}{=} (a\text{-}hit, r_{a\text{-}hit}).Accomplice_{hit} \\
&\quad + (hit, \top).(reload, r_{a\text{-}reload}).Accomplice_{idle} \\
&\quad + (miss, r_{miss}).Accomplice_{empty} + (enemy\text{-}hit, \top).Accomplice_{abs} \\
Accomplice_{hit} &\stackrel{\text{def}}{=} (enemy\text{-}die, \top).(reload, r_{a\text{-}reload}).Accomplice_{idle} \\
Accomplice_{empty} &\stackrel{\text{def}}{=} (reload, r_{a\text{-}reload}).Accomplice_2 + (enemy\text{-}hit, \top).Accomplice_{abs} \\
&\quad + (hit, \top).(reload, r_{a\text{-}reload}).Accomplice_{idle}
\end{aligned}
$$

The component representing the enemy is similar to before.

$$Enemies_{idle} \stackrel{\text{def}}{=} (attack, r_{attack}).Enemies_{attack}$$
$$Enemies_{attack} \stackrel{\text{def}}{=} (enemy\text{-}hit, r_{e\text{-}hit}).Enemies_{attack} + (enemy\text{-}die, \top).Enemies_{idle}$$

The system equation is as follows:

$$(Roland_2 \underset{\{hit,\, a\text{-}hit,\, befriend\}}{\bowtie} Accomplice_{abs}) \underset{\{attack,\, enemy\text{-}die,\, enemy\text{-}hit\}}{\bowtie} Enemies_{idle}$$

Steady-State Analysis. As before steady-state analysis can be used to determine the probability that at any given time Roland is involved in a gun battle. Additionally this can now be used to determine the likelihood that Roland is on his own or has an accomplice. It is interesting to note the relations between the rates involved in the model and the subsequent probabilities. Additionally the relations between each of the steady-state proabilities. Since Roland cannot perform a befriending action while currently involved in a confrontation with an enemy, the probabilty that Roland is in such a battle clearly affects the probability that he is alone in his quest. So for example if Roland's success rate is reduced then gun battles will take longer to resolve, hence Roland will be involved in a gun battle more often, and therefore he will befriend fewer accomplices.

Transient Analysis. An additional transient analysis would be to determine the expected time after Roland has set off before he meets his first accomplice.

Passage-Time Analysis. As with all the previous models the passage-time analysis will measure the probability starting from an *attack* action. Possible actions to stop the analysis at would be the event of the enemy's death or that of the accomplice. Since all gun battles now end in the enemy being killed stopping the analysis there would give us the expected duration of any one gun battle. Stopping the analysis with the death of the accomplice would also incorporate the chance that the enemy is killed but a further enemy attacks and hits the accomplice. However, the extra probability of this is rather small because of the low-rate at which Roland is attacked. Finally the passage-time analysis could be stopped on either of the two hit events, this would give us the probability at a given time after an attack event that either the accomplice or the enemy has been shot.

This model also presents a further possible starting action besides that of the *attack* action, that is the *befriend* action. An interesting passage-time query would be the probability that a given length of time after a *befriend* action has occurred that a *enemy-hit* action occurs. This would give the modeller an estimate of the duration of Roland's friendships.

Hiding. There is a possible deficiency in the above model. What if the enemy starts to perform a *befriend* action (or equally a *hit* or *a-hit* action)? This would invalidate our model as it would model strange things happening, for example if Roland would not be able to meet any new accomplices. Of course the problem is not that the enemy might use this as an underhand tactic since it is the modeller that is describing the enemy component. The problem is that the modeller

is fallible and may make a mistake, especially if the enemy component becomes more complex. One way to avoid this is to 'hide' those actions only Roland and the accomplice should cooperate on.

To do this for our model we can simply change the system equation to read:

$$((Roland_2 \underset{L_1}{\bowtie} Accomplice)/L_1) \underset{L_2}{\bowtie} Enemies_{idle}$$

where $L_1 = \{hit, a\text{-}hit, befriend\}$ and $L_2 = \{attack, enemy\text{-}die, enemy\text{-}hit\}$.

6.2 Web Service Composition

As example of a realistic case study, we consider an example of a business application which is composed from a number of offered web services. Furthermore there is an access control issue, as it must be ensured that the web service consumer has the requisite authority to execute the web services it requests. A schematic representation of the system is depicted in Fig. 11.

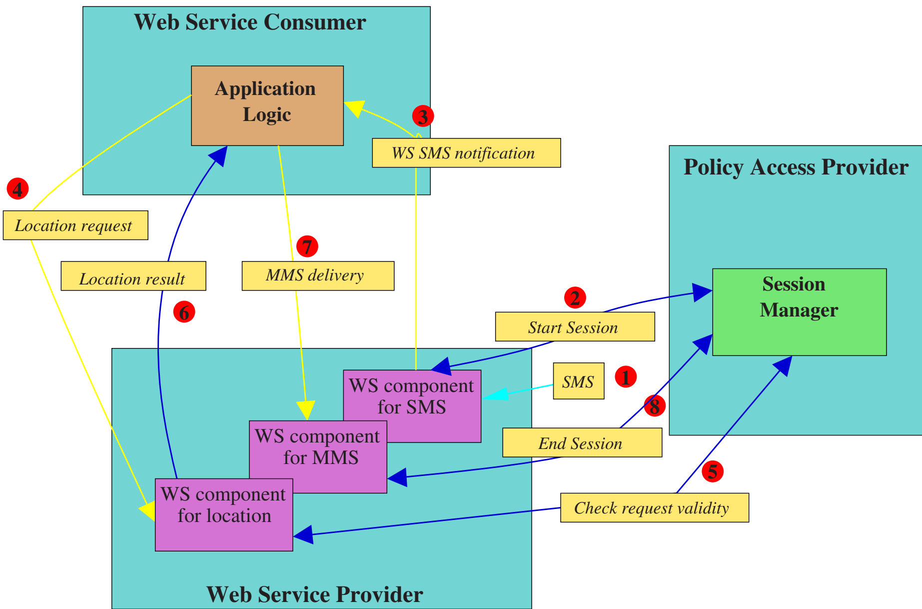


Fig. 11. Schematic representation of the web service composition

The scenario is as follows. Several web services are combined to define the business logic of an application. For example, consider an application to find the nearest restaurant for a user and show it on a map. This could involve web services for SMS and MMS handling in addition to the User Location web service. Moreover, a user should not be able to gain access to location information of an arbitary user.

This is where the access control aspect becomes important. Therefore, in addition to the requested web services, the web service provider may need to interact with some authorisation component to check that the current user has the correct authority to access the requested information. In adddition the service provider may stipulate some further conditions, such as that only one location request may be made per session:

1. The user activates a service by sending an SMS to a service centre number. This is handled by an appropriate web service.
2. This initiates a *start-session* message to be sent to the Policy Access Provider.
3. A notification is sent to the application that an SMS has arrived.
4. The application requests the user's location from a location web service.
5. The web service contacts the session manager within the policy access provider to check the validity of the request.
6. If the validity check is OK the location web service will return the location to the application which uses it to construct the appropriate map for the user.
7. This is then passed as an MMS to the MMS web service which delivers it to the user.
8. The MMS web service terminates the session with the Session Manager.

We model such a system with the following PEPA model. It has three types of model component, corresponding to the three large rectangles in Figure 11. Note that although the Web Service Provider consists of three distinct elements, we are interested in the session associated with each Web Service Consumer. Each session is associated with an instance of the Web Service Provider. Thus, concurrency is introduced into the model by allowing multiple sessions rather than by representing the constituent web services separately.

Component *Customer*. The customer's behaviour is simply modelled with two local states. In the first state the customer sends a request to the system via the *getSMS* action. She then waits for a response which triggers the *getMap* transition if it is successful. Therefore we associate the user-perceived system performance with the throughput of this action, which can be calculated directly from the steady-state probability distribution of the underlying Markov chain.

$$\begin{aligned} Customer &\stackrel{\text{def}}{=} (getSMS, r_1).Customer_1 \\ Customer_1 &\stackrel{\text{def}}{=} (getMap, \top).Customer + (get404, \top).Customer \end{aligned}$$

In this model sending either an error message *get404* or the requested map occur at the same rate *r8* and MMS passing between web services is ten times as fast as the communication with the user.

Component *WSConsumer*. The web service consumer, *WSConsumer*, follows a simple pattern of behaviour. Once it is notified that a session has been started by the user (via SMS message), it initiates a request for the user's current location and waits for a response. If the request was valid, the location is returned and used to compute the appropriate map for the user, which is then sent via an MMS message, using the web service for this.

$$
\begin{aligned}
\mathit{WSConsumer} &\stackrel{\text{def}}{=} (\mathit{notify}, \top).\mathit{WSConsumer}_2 \\
\mathit{WSConsumer}_2 &\stackrel{\text{def}}{=} (\mathit{locReq}, r_4).\mathit{WSConsumer}_3 \\
\mathit{WSConsumer}_3 &\stackrel{\text{def}}{=} (\mathit{locRes}, \top).\mathit{WSConsumer}_4 \\
&\quad + (\mathit{locErr}, \top).\mathit{WSConsumer} \\
\mathit{WSConsumer}_4 &\stackrel{\text{def}}{=} (\mathit{compute}, r_7).\mathit{WSConsumer}_5 \\
\mathit{WSConsumer}_5 &\stackrel{\text{def}}{=} (\mathit{sendMMS}, r_9).\mathit{WSConsumer}
\end{aligned}
$$

Component *WSProvider*. As explained above, although the Web Service Provider can be viewed as consisting of three independent web services, the use of sessions restricts a user's access to these services to be sequential. We assume that there is a distinct instance of the component *WSProvider* for each distinct session. As each would be in a distinct thread it is reasonable for there to be concurrency at this level. The activities of the component are as outlined in the scenario above. Note that the *checkValid* action is represented twice, to capture the two possible distinct outcomes of the action. If the check is successful the location must be returned to the Web Service Consumer in the form of a map (*getMap*). However, if the check revealed an invalid request (*locErr*) then an error must be returned to the Web Service Consumer (*get404*) and the session terminated (*stopSession*).

$$
\begin{aligned}
\mathit{WSProvider} &\stackrel{\text{def}}{=} (\mathit{getSMS}, \top).\mathit{WSProvider}_2 \\
\mathit{WSProvider}_2 &\stackrel{\text{def}}{=} (\mathit{startSession}, r_2).\mathit{WSProvider}_3 \\
\mathit{WSProvider}_3 &\stackrel{\text{def}}{=} (\mathit{notify}, r_3).\mathit{WSProvider}_4 \\
\mathit{WSProvider}_4 &\stackrel{\text{def}}{=} (\mathit{locReq}, \top).\mathit{WSProvider}_5 \\
\mathit{WSProvider}_5 &\stackrel{\text{def}}{=} (\mathit{checkValid}, 99 \cdot \top).\mathit{WSProvider}_6 \\
&\quad + (\mathit{checkValid}, \top).\mathit{WSProvider}_{10} \\
\mathit{WSProvider}_6 &\stackrel{\text{def}}{=} (\mathit{locRes}, r_6).\mathit{WSProvider}_7 \\
\mathit{WSProvider}_7 &\stackrel{\text{def}}{=} (\mathit{sendMMS}, \top).\mathit{WSProvider}_8 \\
\mathit{WSProvider}_8 &\stackrel{\text{def}}{=} (\mathit{getMap}, r_8).\mathit{WSProvider}_9 \\
\mathit{WSProvider}_9 &\stackrel{\text{def}}{=} (\mathit{stopSession}, r_2).\mathit{WSProvider} \\
\mathit{WSProvider}_{10} &\stackrel{\text{def}}{=} (\mathit{locErr}, r_6).\mathit{WSProvider}_{11} \\
\mathit{WSProvider}_{11} &\stackrel{\text{def}}{=} (\mathit{get404}, r_8).\mathit{WSProvider}_9
\end{aligned}
$$

Component *PAProvider*. In our model the Policy Access Provider has a very simple behaviour. It simply maintains a thread for each session and carries out the validity check on behalf of the Web Service Provider. This representation of the *PAProvider* is stateful.

$$PAProvider \stackrel{\text{def}}{=} (startSession, \top).PAProvider_2$$
$$PAProvider_2 \stackrel{\text{def}}{=} (checkValid, r_5).PAProvider_3$$
$$PAProvider_3 \stackrel{\text{def}}{=} (stopSession, \top).PAProvider$$

An alternative design is to have a stateless implementation, as below.

$$\begin{aligned} PAProvider \stackrel{\text{def}}{=}\ & (startSession, \top).PAProvider \\ & + (checkValid, r_5).PAProvider \\ & + (stopSession, \top).PAProvider \end{aligned}$$

We will contrast these two versions in our model analysis.

Model Component *WSComp*. The complete system is represented by some number of instances of the components interacting on their shared activities:

$$\begin{aligned} WSComp \stackrel{\text{def}}{=} \big(& (Customer[N_C] \underset{L_1}{\bowtie} WSProvider[N_{WSP}]) \\ & \underset{L_2}{\bowtie} WSConsumer[N_{WSC}]\big) \\ & \quad \underset{L_3}{\bowtie} PAProvider[N_{PAP}] \end{aligned}$$

where the cooperation sets are

$$L_1 = \{getSMS, getMap, get404\}$$
$$L_2 = \{notify, locReq, locRes, locErr, sendMMS\}$$
$$L_3 = \{startSession, checkValid, stopSession\}$$

and N_C, N_{WSC}, N_{WSP} and N_{PAP} are the number of instances of *Customer*, *WSConsumer*, *WSProvider* and *PAProvider* respectively.

6.3 Performance Analysis of the Web Service Composition Case Study

In this section we carry out steady-state analysis on the Web Service Composition case study in order to tune the parameters of the system. To accomplish this task we use a modified version of the model in which the customer is explicitly modelled as a component of the system. The values for each rate are shown in Table 8.

Suppose that we want to design the system in such a way that it can handle 30 independent customers. The modeller may have constraints on some parameters such as the network delays because those are limited by the available technology. However, there are a number of degrees of freedom which let her vary, for example, the number of threads of control of the components of the system. The purpose is to deliver a satisfactory service in a cost-effective way. The simplest example of a cost function may be a linearly dependency on the number of copies of a component or the rate at which an activity is performed.

The graph in Fig. 12 shows the throughput of the *getMap* action as the number of customers varies between 1 and 30. Each line represents a given number of

Table 8. Parameters used in the performance analysis of the Web Service composition

parameter	*value*	*explanation*
r_1	0.0010	rate at which customers request maps
r_2	0.5	rate at which a session can be started
r_3	0.1	notification exchange between consumer and provider
r_4	0.1	rate at which requests for customer's location can be satisfied
r_5	0.05	rate at which the provider can check the validity of the incoming request
r_6	0.1	rate at which location information can be returned to the consumer
r_7	0.05	rate at which maps can be generated
r_8	0.02	rate at which MMS messages can be sent from provider to customer
r_9	$10.0 * r_8$	rate at which MMS messages can be sent via the Web Service

copies of the *WSProvider* component in the system. When the total number of customers is 30, two providers lead to a throughput which is twice as much as in the base system configuration with one provider only. However, as the number of provider increases the incremental benefit becomes less significant. In particular, the system with four copies is just 8.7% faster than the system with three. In the following we set to three the copies of *WSProvider*.

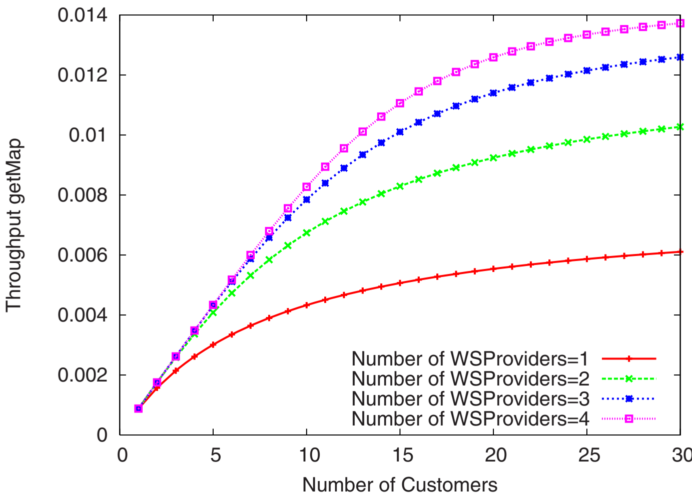


Fig. 12. Throughput of *getMap* for changes in the number of *WSProvider* and customers

In Fig. 13 is shown the effect that the rate at which the users initiate the request (r_1) has on the *getMap* throughput for different values of the copies of the *WSConsumer*. Every line starts to plateau at approximately $r_1 = 0.010$ following

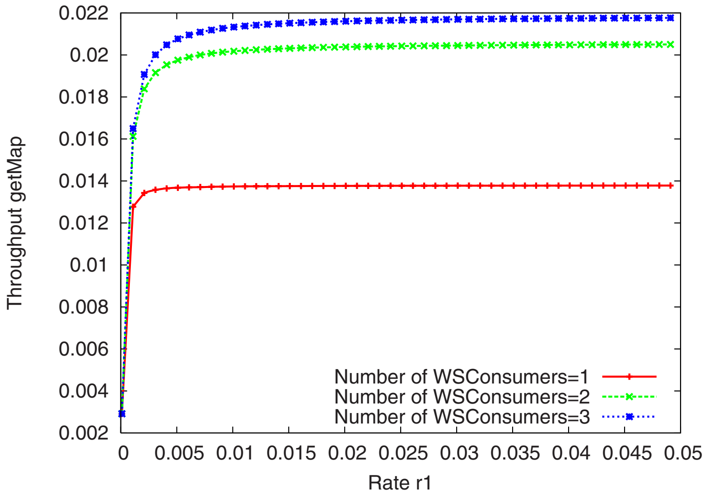


Fig. 13. Throughput of *getMap* for changes in the number of *WSConsumer* and r_1

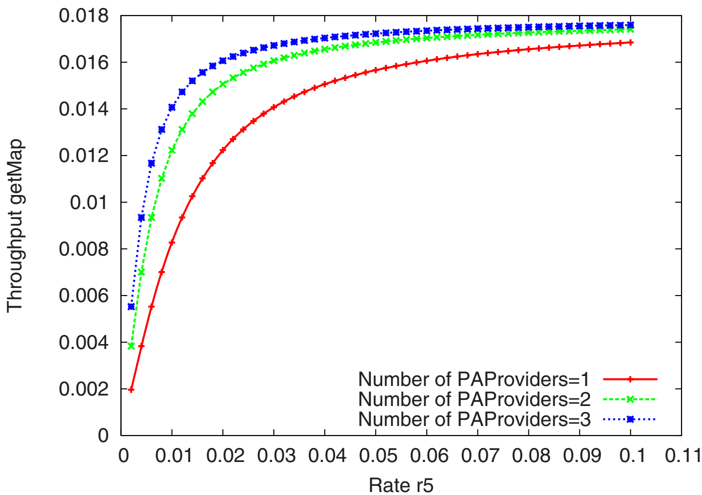


Fig. 14. Throughput of *getMap* for changes in the number of *PAProvider* and r_5

an initial sharp increase. This suggests that the system can guarantee satisfactory behaviour under the constraint that the users' request rate is below that threshold. In addition, the graph gives the modeller insights into the optimal number of operating threads of control of *WSConsumer*, which we believe is two as the additional third copy is not well matched by performance boost. Hence, in order to tune *PAProvider*—the remaining system component—we set *WSConsumer* to that value.

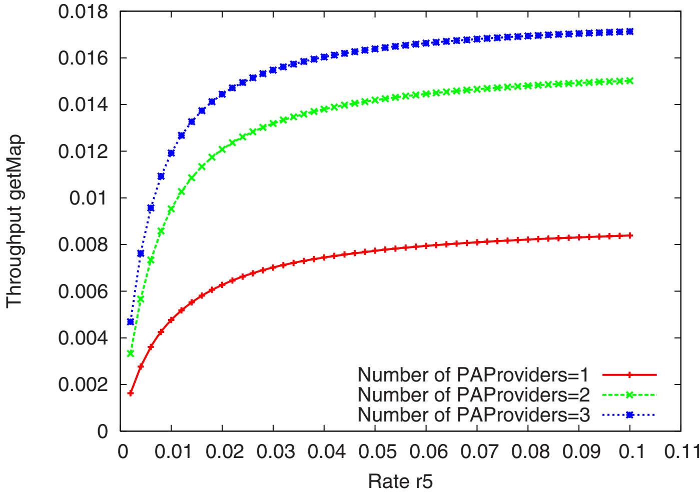


Fig. 15. Throughput of *getMap* for changes in the number of stateful *PAProvider* and r_5

The same approach can be applied to the optimisation of the number of copies of *PAProvider*. Here we are particularly interested in the overall impact of the rate at which the validity check is performed. Slower rates may mean more computationally expensive validation, whereas faster rates may involve less accuracy and lower security of the system. Such effects are measured in Fig. 14 where the *getMap* throughput is plotted against r_5 for different *PAProvider* pool sizes. A sharp increase followed by a constant levelling off suggests that optimal rate values lie on the left of the plateau, as faster rates do not improve the system considerably. As for the optimal number of copies of *PAProvider*, deploying two copies rather than one dramatically increases the quality of service of the overall system. With a similar approach as previously discussed, the modeller may want to consider the trade-off between the cost of adding a third copy and the throughput increase.

Evaluation of an alternative design of *PAProvider*. We conclude this section by showing how this model can be used to evaluate alternative designs of parts of the system. Here, we focus on *PAProvider* which has been originally modelled as a stateless component. Any of its services can be called at any point, the correctness of the system being guaranteed by implementation-specific constraints such as session identifiers being uniquely assigned to the clients and passed as parameters of the method calls.

Another design of a component which offers the same functionalities is that of a stateful provider. In PEPA such a service can be modelled as a sequential component with three local states (see above). This implementation has the consequence that there can never be at any point in time more than N_{WSP} *WSProvider* which have started a session with a *PAProvider*. This is because the provider has to release a previous session in order to start another one.

The graph in Fig. 15 measures the same metrics as in Fig. 14 when the stateful provider is employed. It shows that the incremental gain in adding more copies has become more noteworthy. However, the modeller may want to prefer the original version, as three copies of the stateful provider deliver about as much as the throughput of only one copy of the stateless implementation.

7 Advanced Topics

Like all state based modelling techniques, stochastic process algebra models are subject to the problem of state space explosion — the generated models may be intractable because of their size. A variety of techniques have been proposed for tackling this problem in the context of stochastic process algebra. Below we briefly discuss two of them:

- model reduction and model simplification via equivalence relations;
- fluid approximation of the state space.

7.1 Equivalence Relations and Model Manipulation

The state space explosion problem arises because although the compositionality of SPA can greatly aid model construction, in general the compositionality does not assist in the model solution and the resulting models may be too large to solve. This has led to research into how model simplification and aggregation techniques can be applied in the process algebra setting. Many such techniques are known in the context of Markov processes but are based on conditions phrased in terms of the process or its generator matrix. Moreover application of these techniques often relies on the expertise of the modeller. The challenge for SPA has been to define such model manipulation techniques in the context of the process algebra, in such a way that it can subsequently be applied automatically. Some significant results have been achieved in this area through the use of equivalence relations which provide the basis for comparing and manipulating models within a formal framework. Furthermore the compositionality of the process algebra allows these techniques to be applied to part of the model whilst maintaining the integrity of the model as a whole.

There have been two principal approaches to model manipulation in SPA:

model simplification: Here an equivalence relation is used in order to establish behavioural or observational equivalence *between models*. The aim is to replace one model by an equivalent one which is more desirable from a solution point of view. Once the desirable model has replaced the original, the underlying Markov process is generated as usual, associating one state with each node in the labelled transition system generated by the semantics. Equivalence relations which have been used in this way are *weak isomorphism* in PEPA [29,69], *Markovian bisimulation* and *weak bisimulation* in TIPP [70].

model aggregation: Here an equivalence relation is used in order to establish behavioural or observational equivalence *between states within a model*. The

aim is to use an alternative mapping from the labelled transition system, given by the semantics of the model, to the underlying Markov process. The equivalence relation is used to partition the nodes of the labelled transition system into equivalence classes. Then, instead of the usual one-to-one correspondence between nodes and states, one state in the underlying Markov process is associated with each equivalence class of nodes. The hope is that this will generate a Markov process with a smaller number of states. The equivalence relation which has been used in this way is variously called *strong equivalence* (PEPA) [29], *Markovian bisimulation* (TIPP) [71], and *extended Markovian bisimulation equivalence* (EMPA) [27].

Equivalence relations and model manipulations will be discussed in more detail in another chapter within this volume [72].

The basis of aggregation is the observation that it can be sufficient to consider the behaviour of one element within an equivalence class of elements who all behave in the same way. The simplest way in which such equivalence classes arise is if we have repeated instances of identical components within the model. For this case, for PEPA models we have developed an automatic method which generates the CTMC corresponding the equivalence classes, rather than the individual states, on-the-fly [47]. This relies on a *canonical representation* of states within the model which makes it clear syntactically when they are equivalent, while also keeping track of how many instances there are in each such equivalence class.

Since the static cooperation combinators remain unchanged in all states of a model, it is often convenient to represent the states in *vector form*. The state vector records one entry for each sequential component of the PEPA model. These components will be present in each derivative of the model, although they will change their local state or derivative. Thus the global state can be represented as a vector or sequence of local derivatives.

If a model contains equivalent components there may be multiple states within the model which exhibit the same behaviour and so we may aggregate the model. The derivation graph is then constructed in terms of equivalence classes of syntactic terms and this is used as the basis of the CTMC construction [47]. Canonicalisation involves reordering entries within the vector in a way that strong equivalence, the Markovian bisimulation of PEPA models, is respected, but which places elements within subvectors of equivalent components in lexicographical order. Further details can be found in [47].

7.2 Continuous State Space Approximation

Even with the use of aggregation some model still remain too large to be readily analysed using Markovian techniques. Recent work has considered a radically different approach to tackling the state space explosion problem when modelling with a process algebra such as PEPA [73]. The approach is based on two shifts from the usual perspective:

- Firstly, we do not aim to calculate the probability distribution over the entire state space of the model. We choose a more abstract state representation in

terms of state variables, quantifying the types of behaviour evident in the model.

– Secondly, we assume that these state variables are subject to *continuous* rather than *discrete* change.

Once these adjustments are made the system is amenable to efficient solution as a set of ordinary differential equations (ODEs), leading to the evaluation of transient, and in the limit, steady state measures.

State representation. As we have seen the usual state representation is in terms of the syntactic forms of the model expression, or when aggregation is applied, in terms of a canonical representation of an equivalence class of states.

The work on continuous approximation proposes an alternative vector form for capturing the state information of models with repeated components. In the state vector form, even when the canonical representation is used there is one entry in the vector for each sequential component in the model. When the number of repeated components becomes large this can be prohibitively expensive in terms of storage. In the alternative vector form there is one entry for each local derivative of each type of component in the model. Two components have the same type if their derivation graphs are isomorphic. The entries in the vector are no longer syntactic terms representing the local derivative of the sequential component, but the *number* of components currently exhibiting this local derivative.

To clarify the distinction between the two vector forms consider the small example defined below, consisting of interacting processors and resources:

$$\begin{aligned}
Processor_0 &\stackrel{\text{def}}{=} (task1, r_1).Processor_1 \\
Processor_1 &\stackrel{\text{def}}{=} (task2, r_2).Processor_0 \\
Resource_0 &\stackrel{\text{def}}{=} (task1, r_1).Resource_1 \\
Resource_1 &\stackrel{\text{def}}{=} (reset, s).Resource_0
\end{aligned}$$

$$(Resource_0 \parallel Resource_0) \underset{\{task1\}}{\bowtie} (Processor_0 \parallel Processor_0)$$

The canonical state vector form corresponding to this example with the given configuration is shown in Figure 16a). Here the initial state is represented explicitly as $((Resource_0, Resource_0), (Processor_0, Processor_0))_{\{task1\}}$. In contrast, in the numerical vector form, shown in Figure 16b), the initial state is $(2, 0, 2, 0)$ where the entries in the vector are counting the number of $Resource_0$, $Resource_1$, $Processor_0$, $Processor_1$ local derivatives respectively, exhibited in the current state. In the canonical state vector representation we record the number of elements in each equivalence class (shown in square brackets in Figure 16a). The total rate of the transitions between the canonical states is derived from this number of instances, the number of enabled activities and their relative probabilities. In the numerical state vector representation each vector is a single state and the rates of the transitions between states are derived directly from the vector and the activity rate, as explained below.

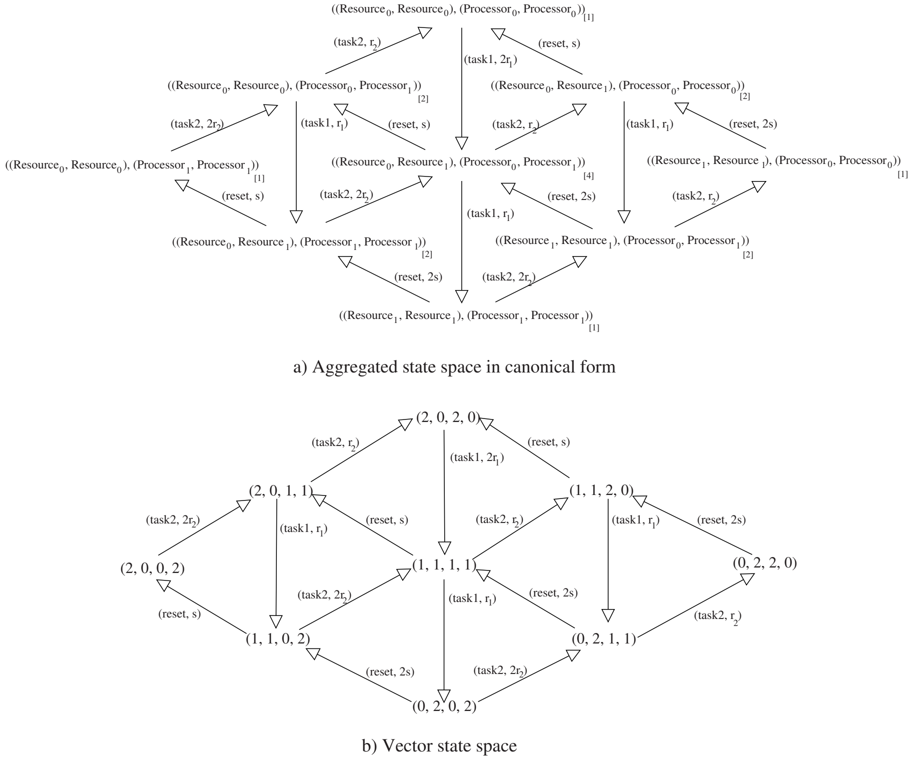


Fig. 16. Illustrative example of contrasting state representations

In the current configuration of the model, with two instances of each component type, it is clear that the state vector form and the numerical vector form each have four elements, but if we consider a configuration with ten instances of each component type it becomes clear that the numerical form is much more compact.

The numerical vector form for an arbitrary PEPA model is defined as follows.

Definition 1 (Numerical Vector Form). *For an arbitrary PEPA model $\mathcal{M}$ with n component types $\mathcal{C}_i, i = 1, \ldots, n$, each with N_i distinct derivatives, the* numerical vector form *of $\mathcal{M}$, $\mathcal{V}(\mathcal{M})$, is a vector with $N = \sum_{i=1}^{n} N_i$ entries. The entry v_{i_j} records how many instances of the jth local derivative of component type $\mathcal{C}_i$ are exhibited in the current state.*

If there is a large number of instances of each component type the domain of values of each entry in $\mathcal{V}(\mathcal{M})$ is large. If K_i is the number of components of type $\mathcal{C}_i$ in the initial configuration of the model then each entry in the ith subvector will have domain $0, \ldots, K_i$.

The system is inherently discrete with the entries within the numerical vector form always being non-negative integers and always being incremented or decremented in steps of one. When the numbers of components are large these steps

are relatively small and we can approximate the behaviour by considering the movement between states to be continuous, rather than occurring in discontinuous jumps. In this case we can replace the discrete event system represented by the derivation graph of a PEPA process by a continuous model, represented by a set of coupled ordinary differential equations. The numerical vector form of state representation is an intermediate step to achieving that. Considering these states of the process and the activities which are enabled, and the states they lead to, we are able to construct an *activity matrix* which records the impact of each activity type on the number of each component type. From this the appropriate system of ODEs is derived (see [73] for details).

Small example revisited. Let us consider again the small example considered earlier, assuming now that there are large numbers of processors and resources:

$$\begin{aligned}
\mathit{Processor}_0 &\stackrel{\text{def}}{=} (\mathit{task}1, r_1).\mathit{Processor}_1 \\
\mathit{Processor}_1 &\stackrel{\text{def}}{=} (\mathit{task}2, r_2).\mathit{Processor}_0 \\
\mathit{Resource}_0 &\stackrel{\text{def}}{=} (\mathit{task}1, r_1).\mathit{Resource}_1 \\
\mathit{Resource}_1 &\stackrel{\text{def}}{=} (\mathit{reset}, s).\mathit{Resource}_0
\end{aligned}$$

$$(\mathit{Resource}_0 \parallel \cdots \parallel \mathit{Resource}_0) \underset{\{\mathit{task}1\}}{\bowtie} (\mathit{Processor}_0 \parallel \cdots \parallel \mathit{Processor}_0)$$

Let n_1 denote the number of $\mathit{Processor}_0$ entities, n_2 the number of $\mathit{Processor}_1$ entities, n_3 the number of Res_0 entities and n_4 the number of $\mathit{Resource}_1$ entities. The activity matrix corresponding the component definitions is shown in Fig. 17.

	$task_1$	$task_2$	$reset$	
$Processor_0$	-1	$+1$	0	n_1
$Processor_1$	$+1$	-1	0	n_2
$Resource_0$	-1	0	$+1$	n_3
$Resource_1$	$+1$	0	-1	n_4

Fig. 17. Activity matrix for the simple Processor-Resource model

From the matrix, we derive each differential equation in turn. For state variable n_i, consider row i. Each non-zero entry in the row will results in one term within the equation.

$$\begin{aligned}
\frac{dn_1(t)}{dt} &= -r_1 \min(n_1(t), n_3(t)) + r_2 n_2(t) \\
\frac{dn_2(t)}{dt} &= r_1 \min(n_1(t), n_3(t)) - r_2 n_2(t) \\
\frac{dn_3(t)}{dt} &= -r_1 \min(n_1(t), n_3(t)) + s n_4(t) \\
\frac{dn_4(t)}{dt} &= r_1 \min(n_1(t), n_3(t)) - s n_4(t)
\end{aligned}$$

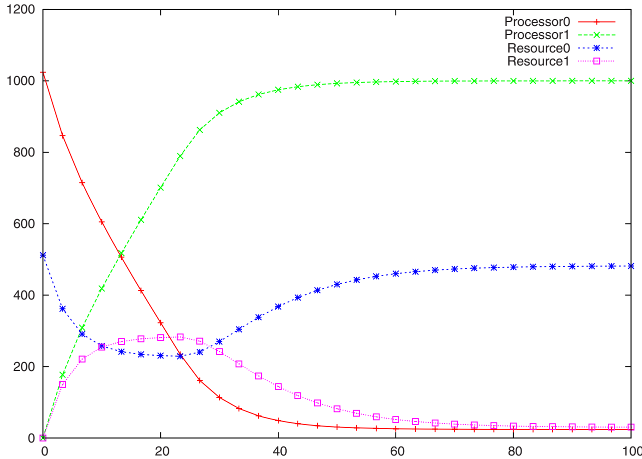


Fig. 18. Graph showing the changing numbers of copies of $Processor_0$, $Processor_1$, $Resource_0$ and $Resource_1$ as a function of time, obtained by numerically integrating the differential equations for this system. The values of the rates were $r_1 = 0.125$, $r_2 = 0.003$ and $s = 0.1$.

Note that the form of the system of equations is independent of the number of components included in the initial configuration of the model. The only impact of changing the number of instances of each component type is to alter the initial conditions. Thus, if there are initially 1024 processors, all starting in state $Processor_0$ and 512 resources, all of which start in state $Resource_0$, the initial conditions will be:

$$n_1(0) = 1024 \qquad n_2(0) = 0 \qquad n_3(0) = 512 \qquad n_4(0) = 0$$

Numerically integrating the differential equations for this system to generate a time series plot for the first 100 seconds of the system evolution starting from the above initial value problem produces the graph shown in Fig. 18.

8 Conclusions and Summary

In this tutorial we have described an algebraic description technique, based on a classical process algebra, and enhanced with timing information. This extension results in models which may be used to calculate performance measures as well as deduce functional properties of the system. Several interesting analysis techniques of SPA models including steady state, transient and response time analysis of the underlying CTMC have been discussed, together with an introduction to the tools which support these analysis techniques. We have demonstrated the approach on a number of small models as well as a more realistic example of a service-oriented architecture. Finally we outlined some more advanced topics related to SPA and highlighted some on-going work.

Acknowledgements

This work has been supported by the project EU FET-IST Global Computing 2 project SENSORIA (”Software Engineering for Service-Oriented Overlay Computers” (IST-3-016004-IP-09)). Jane Hillston is also supported by EPSRC Advanced Research Fellowship EP/c543696/01.

References

1. Herzog, U.: Formal description, time and performance analysis: A framework. Technical Report 15/90, IMMD VII, Friedrich-Alexander-Universität, Erlangen-Nürnberg, Germany (September 1990)
2. Holton, D.: A PEPA specification of an industrial production cell. In Gilmore, S., Hillston, J., eds.: Proceedings of the Third International Workshop on Process Algebras and Performance Modelling, Special Issue of *The Computer Journal*, 38(7) (December 1995) 542–551
3. Gilmore, S., Hillston, J., Holton, D., Rettelbach, M.: Specifications in Stochastic Process Algebra for a Robot Control Problem. International Journal of Production Research **34**(4) (1996) 1065–1080
4. Thomas, N., Hillston, J.: Using Markovian process algebra to specify interactions in queueing systems. Technical Report ECS-LFCS-97-373, Laboratory for Foundations of Computer Science, Department of Computer Science, The University of Edinburgh (1997)
5. Bowman, H., Bryans, J., Derrick, J.: Analysis of a multimedia stream using stochastic process algebra. In Priami, C., ed.: Sixth International Workshop on Process Algebras and Performance Modelling, Nice (September 1998) 51–69
6. Console, L., Picardi, C., Ribaudo, M.: Diagnosis and Diagnosability Analysis using PEPA. In: Proc. of 14th European Conference on Artificial Intelligence, Berlin (August 2000) A longer version appeared in the Proc. of 11th Int. Workshop on Principles of Diagnosis (DX00), Morelia, Mexico, June 2000.
7. Hillston, J., Kloul, L.: Performance investigation of an on-line auction system. Concurrency and Computation: Practice and Experience **13** (2001) 23–41
8. Forneau, J., Kloul, L., Valois, F.: Performance modelling of hierarchical cellular networks using PEPA. Performance Evaluation **50**(2–3) (November 2002) 83–99
9. Brodo, L., Degano, P., Gilmore, S., Hillston, J., Priami, C.: Performance evaluation for global computation. In Priami, C., ed.: Global Computing: Programming environments, languages, security, and analysis of systems. Proceedings of the IST/FET International Workshop (GC 2003). Volume 2874 of LNCS., Rovereto, Italy, Springer-Verlag (February 2003) 229–253
10. Buchholtz, M., Gilmore, S., Hillston, J., Nielson, F.: Securing statically-verified communications protocols against timing attacks. Electr. Notes Theor. Comput. Sci. **128**(4) (2005) 123–143
11. Hillston, J., la Kloul, L., Mokhtari, A.: Towards a feasible active networking scenario. Telecommunication Systems **27**(2–4) (October 2004) 413–438
12. Fourneau, J.M., Kloul, L.: A precedence PEPA model for performance and reliability analysis. In Horváth, A., Telek, M., eds.: Formal Methods and Stochastic Models for Performance Evaluation: Third European Performance Engineering Workshop (EPEW 2006). Number 4054 in LNCS, Springer-Verlag (June 2006) 1–15

13. Duguid, A.: Coping with the parallelism of BitTorrent: Conversion of PEPA to ODEs in dealing with state space explosion. In Asarin, E., Bouyer, P., eds.: Formal Modeling and Analysis of Timed Systems, 4th International Conference, FORMATS 2006, Paris, France, September 25-27, 2006, Proceedings. Volume 4202 of Lecture Notes in Computer Science., Springer (2006) 156–170
14. Gilmore, S., Tribastone, M.: Evaluating the scalability of a web service-based distributed e-learning and course management system. In Mario Bravetti, M.T.N.n., Zavattaro, G., eds.: Third International Workshop on Web Services and Formal Methods (WS-FM'06). Volume 4184 of Lecture Notes in Computer Science., Vienna, Austria, Springer (2006) 156–170
15. Razafindralambo, T., Valois, F.: Performance evaluation of backoff algorithms in 802.11 ad-hoc networks. In: PE-WASUN '06: Proceedings of the 3rd ACM international workshop on Performance evaluation of wireless ad hoc, sensor and ubiquitous networks, New York, NY, USA, ACM Press (2006) 82–89
16. Razafindralambo, T., Valois, F.: Stochastic behavior study of backoff algorithms in case of hidden terminals. In: Proceedings of the 17th Annual IEEE International Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC'06), IEEE Press (2006) 1–6
17. Milner, R.: Communication and Concurrency. Prentice-Hall (1989)
18. Hoare, C.: Communicating Sequential Processes. Prentice-Hall (1985)
19. Nicollin, X., Sifakis, J.: An Overview and Synthesis on Timed Process Algebras. In: Real-Time: Theory in Practice. Springer LNCS 600 (1991) 526–548
20. Moller, F., Tofts, C.: A Temporal Calculus for Communicating Systems. In Baeten, J., Klop, J., eds.: CONCUR'90. Volume 458 of LNCS., Springer-Verlag (August 1989) 401–415
21. Jou, C.C., Smolka, S.: Equivalences, Congruences and Complete Axiomatizations of Probabilistic Processes. In Baeten, J., Klop, J., eds.: CONCUR'90. Volume 458 of LNCS. Springer-Verlag (August 1990) 367–383
22. Edinburgh Concurrency Workbench. `http://homepages.inf.ed.ac.uk/perdita/cwb/`
23. Hennessy, M., Milner, R.: On observing nondeterminism and concurrency. In de Bakker, J.W., van Leeuwen, J., eds.: Proceedings 7th ICALP, *Noordwijkerhout*. Volume 85 of Lecture Notes in Computer Science., Springer-Verlag (July 1980) 299–309
24. Kozen, D.: Results on the propositional μ-calculus. Theoretical Computer Science **27**(3) (1983) 333–354
25. Götz, N., Herzog, U., Rettelbach, M.: TIPP—a language for timed processes and performance evaluation. Technical Report 4/92, IMMD7, University of Erlangen-Nürnberg, Germany (November 1992)
26. Bernardo, M., Gorrieri, R., Donatiello, L.: MPA: A Stochastic Process Algebra. Technical Report UBLCS-94-10, Laboratory of Computer Science, University of Bologna (May 1994)
27. Bernardo, M., Gorrieri, R.: A Tutorial on EMPA: A Theory of Concurrent Processes with Nondeterminism, Priorities, Probabilities and Time. Theoretical Computer Science **to appear** (1998)
28. Hillston, J.: PEPA - Performance Enhanced Process Algebra. Technical report, Dept. of Computer Science, University of Edinburgh (March 1993)
29. Hillston, J.: A Compositional Approach to Performance Modelling. PhD thesis, Department of Computer Science, University of Edinburgh (April 1994) CST-107-94.

30. Strulo, B.: Process Algebra for Discrete Event Simulation. PhD thesis, Imperial College (1993)
31. Priami, C.: Stochastic π-Calculus. The Computer Journal **38**(6) (1995)
32. Hermanns, H.: Interactive Markov Chains: The Quest for Quantified Quality. Volume 2428 of LNCS. Springer (2002)
33. D'Argenio, P., Hermanns, H., Katoen, J.P., Klaren, R.: MoDeST: A modelling language for stochastic timed systems. In: Process Algebra and Probabilistic Methods, Springer-Verlang LNCS 2165 (2001) 87–104
34. Bravetti, M., Gorrieri, R.: The theory of Interactive Generalized Semi-Markov Processes. Theoretical Computer Science **282**(1) (June 2002) 5–32
35. Bravetti, M., Bernardo, M., Gorrieri, R.: From EMPA to GSMPA: Allowing for general distributions. In Brinksma, E., Nymeyer, A., eds.: Proc. of the 5th Int. Workshop on Process Algebras and Performance Modeling (PAPM '97). (1997) 17–33
36. Rettelbach, M.: Probabilistic Branching in Markovian Process Algebras. The Computer Journal **38**(6) (1995) Special Issue: Proc. of 3rd Process Algebra and Performance Modelling Workshop.
37. Hermanns, H., Rettelbach, M., Weiß, T.: Formal Characterisation of Immediate Actions in SPA with Nondeterministic Branching. The Computer Journal **38**(6) (1995) Special Issue: Proc. of 3rd Workshop on Process Algebras and Performance Modelling.
38. Hillston, J.: The nature of synchronisation. In Herzog, U., Rettelbach, M., eds.: Proceedings of the Second International Workshop on Process Algebras and Performance Modelling, Erlangen (November 1994) 51–70
39. Ribaudo, M.: Understanding Stochastic Process Algebras via their Stochastic Petri Net Semantics. In Herzog, U., Rettelbach, M., eds.: Proc. of 2nd Process Algebra and Performance Modelling Workshop. (1994)
40. Bradley, J.: Towards Reliable Modelling with Stochastic Process Algebras. PhD thesis, Department of Computer Science, University of Bristol (1999)
41. Hillston, J.: A Compositional Approach to Performance Modelling. Cambridge University Press (1996)
42. Clark, G., Gilmore, S., Hillston, J., Ribaudo, M.: Exploiting modal logic to express performance measures. In Haverkort, B., Bohnenkamp, H., Smith, C., eds.: Computer Performance Evaluation: Modelling Techniques and Tools, Proceedings of the 11th International Conference. Number 1786 in LNCS, Schaumburg, Illinois, USA, Springer-Verlag (March 2000) 211–227
43. Dempster, E.W., Tomov, N.T., Lü, J., Pua, C.S., Williams, M.H., Burger, A., Taylor, H., Broughton, P.: Verifying a performance estimator for parallel DBMSs. In: Proceedings of EuroPar (EuroPar'98). (September 1998)
44. Bouzeghoub, M., Kloul, L., Mokhtari, A.: A new active network framework based on active rules. Technical Report 2002/21, PRiSM, Université de Versailles (2002)
45. Hillston, J., Kloul, L., Mokhtari, A.: Active nodes performance analysis using PEPA. [74] 244–256
46. Hillston, J., Kloul, L., Mokhtari, A.: Towards a feasible active networking scenario. Telecommunication Systems **27**(2–4) (2004) 413–438
47. Gilmore, S., Hillston, J., Ribaudo, M.: An efficient algorithm for aggregating PEPA models. IEEE Transactions on Software Engineering **27**(5) (May 2001) 449–464
48. Console, L., Picardi, C., Ribaudo, M.: Diagnosis and Diagnosability Analysis using Process Algebras. In: Proc. of 11th Int. Workshop on Principles of Diagnosis (DX00), Morelia, Mexico (June 2000)

49. Eclipse.org home. `http://www.eclipse.org`
50. Eclipse Modeling Framework. `http://www.eclipse.org/home`
51. Matrix Toolkit for Java. `http://rs.cipr.uib.no/mtj/`
52. Bradley, J., Dingle, N., Gilmore, S., Knottenbelt, W.: Derivation of passage-time densities in PEPA models using IPC: The Imperial PEPA Compiler. In Kotsis, G., ed.: Proceedings of the 11th IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunications Systems, University of Central Florida, IEEE Computer Society Press (October 2003) 344–351
53. Knottenbelt, W.: Generalised Markovian analysis of timed transition systems. Master's thesis, University of Cape Town (1996)
54. Bradley, J., Dingle, N., Gilmore, S., Knottenbelt, W.: Extracting passage times from PEPA models with the HYDRA tool: A case study. [74] 79–90
55. Argent-Katwala, A., Bradley, J., Dingle, N.: Expressing performance requirements using regular expressions to specify stochastic probes over process algebra models. In: Proceedings of the Fourth International Workshop on Software and Performance, Redwood Shores, California, USA, ACM Press (January 2004) 49–58
56. PRISM. `http://www.cs.bham.ac.uk/~dxp/prism/index.php`
57. Hermanns, H., Meyer-Kayser, J., Siegle, M.: Multi-terminal binary decision diagrams to represent and analys e continuous time markov chains. In: Proc. of 3rd Intl. Workshop on the Numerical Solution of Mark ov Chains. (1999) 188–207
58. Gilmore, S., Kloul, L.: A unified tool for performance modelling and prediction. In S. Anderson, B.L., Felici, M., eds.: Proceedings of the 22nd International Conference on Computer Safety, Reliability and Security (SAFECOMP 2003). Volume 2788 of LNCS., Springer-Verlag (2003) 179–192
59. Gilmore, S., Hillston, J., Kloul, L., Ribaudo, M.: Software performance modelling using PEPA nets. In: Proceedings of the Fourth International Workshop on Software and Performance, Redwood Shores, California, USA, ACM Press (January 2004) 13–24
60. Clark, G., Courtney, T., Daly, D., Deavours, D., Derisavi, S., Doyle, J.M., Sanders, W.H., Webster, P.: The Möbius modeling tool. In: Proc. of 9th Int. Workshop on Petri Nets and Performance Models, Aachen, Germany (September 2001) 241–250
61. Clark, G., Sanders, W.: Implementing a stochastic process algebra within the Möbius modeling framework. In de Alfaro, L., Gilmore, S., eds.: Proceedings of the first joint PAPM-PROBMIV Workshop. Volume 2165 of Lecture Notes in Computer Science., Aachen, Germany, Springer-Verlag (September 2001) 200–215
62. : TwoTowers 5.1. `http://www.sti.uniurb.it/bernardo/twotowers/`
63. Hermanns, H., Mertsiotakis, V.: A Stochastic Process Algebra Based Modelling Tool. In: Proc. of the 11th UK Performance Engineering Workshop for Computer and Telecommunication Systems, Springer (1995)
64. Hermanns, H., Herzog, U., Klehmet, U., Mertsiotakis, V., Siegle, M.: Compositional performance modelling with the TIPPtool. In: Proc. of 10th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation. Volume 1469 of LNCS., Palma de Mallorca, Springer-Verlag (1998)
65. Cleaveland, W., Sims, S.: The NCSU Concurrency Workbench. In: Proc. of Int. Conf. on Computer Aided Verification (CAV'96). Volume 1102 of LNCS., Springer-Verlag (1996) 394–397
66. Stewart, W.: Introduction to the Numerical Solution of Markov Chains. Princeton University Press (1994)

67. Bohnenkamp, H., Courtney, T., Daly, D., Derisavi, S., Hermanns, H., Katoen, J.P., Klaren, R., Lamb, V., Sanders, W.: On integrating the Möbius and Modest modeling tools. In: Proceedings of the 2003 International Conference on Dependable Systems and Networks (DSN'03), IEEE Computer Society Press (2003)
68. Garavel, H., Lang, F., Mateescu, R.: An overview of CADP 2001. Technical Report RT-254, INRIA (2001)
69. Clark, G.: An Extended Weak Isomorphism for Model Simplification. In Brinksma, E., Nymeyer, A., eds.: Proc. of 5th Process Algebra and Performance Modelling Workshop. (1997)
70. Mertsiotakis, V.: Approximate Analysis Methods for Stochastic Process Algebras. PhD thesis, Universität Erlangen–Nürnberg, Martensstraße 3, 91058 Erlangen (September 1998)
71. Hermanns, H., Rettelbach, M.: Syntax, Semantics, Equivalences and Axioms for MTIPP. In Herzog, U., Rettelbach, M., eds.: Proc. of 2nd Process Algebra and Performance Modelling Workshop. (1994)
72. Bernardo, M.: Behavioural equivalences and model manipulations. In Bernardo, M., Hillston, J., eds.: Formal Methods for Performance Evaluation. LNCS. Springer (2007)
73. Hillston, J.: Fluid flow approximation of PEPA models. In: Proceedings of the Second International Conference on the Quantitative Evaluation of Systems, Torino, Italy, IEEE Computer Society Press (September 2005) 33–43
74. Jarvis, S., ed.: Proceedings of the Ninteenth UK Performance Engineering Workshop, University of Warwick (July 2003)