Formalising the Specification of Rewards with PEPA

Graham Clark[1]

Department of Computer Science, The University of Edinburgh, Edinburgh, EH9 3JZ, UK.
E-mail: gcla@dcs.ed.ac.uk

Abstract

Stochastic process algebras provide a novel way to model the performance of systems. As well as facilitating a compositional approach, process algebra models view a system in terms of its behaviour rather than its state space. With traditional performance modelling techniques, the method of defining reward structures in order to derive performance measures is widely used. However, due to the behavioural approach, there are some philosophical and technical problems in using this method of reward structure specification directly in the process algebra setting. This paper describes work towards a formal and behaviour-based approach to specifying performance measures. This is developed in the context of Hillston's PEPA, a popular SPA. A simple language is described which is capable of expressing many of the most common performance measures. The main motivation for this work is practical—work on the implementation of this technique in the PEPA Workbench is close to completion.

1 Introduction

Including performance characteristics in the design of a model from the outset affords obvious benefits to the modeller. When integrated with a process algebra, at each stage in the design it is possible to ensure that the model performs as well as is required; this is in addition to ensuring that the model has the intended functional behaviour. This led to the development of stochastic process algebras (SPA, [1, 2, 3]). Verifying that a model possesses a particular logical property is called *model checking* [4]. The development of efficient algorithms to test the qualitative behaviour of a model is still an active area of research. With traditional performance modelling techniques, the method of defining reward structures in order to derive performance measures is widely used and understood. However, there are some philosophical and technical problems in using this method of reward structure specification directly in the process algebra setting. SPA terms model the actions of a system, i.e. its behaviour; it is unwieldy to attempt to use the usual state-based methods of specifying performance measures. This problem was identified in [5]. His preliminary work followed the idea that a *temporal logic* formula could be used to partition a process algebra model, each partition exhibiting a particular behaviour. Thus, whereas a temporal logic formula is used to verify the functional behaviour of a process algebra model, Hermanns considered using a formula to discriminate between stages

[1] Presented at the 4th Workshop on Process Algebra and Performance Modelling, Torino, Italy, 4th and 5th July 1996. This work is supported by EPSRC grant 95306547

of a process' life in which it could, and could not exhibit a particular behaviour. This observation lies behind the work outlined in this paper. By making use of a simple *modal logic*, the formalisation of performance measures is described with respect to a particular SPA, PEPA [3]. A simple language is described which is capable of expressing many of the most common performance measures, for instance throughput and utilisation. This work has pragmatic motivations. The PEPA Workbench [6] is a useful tool which assists in the analysis of PEPA models. Currently, the model solution procedure is completely automated. It should prove useful when the specification of performance measures is formalised, and thus lead to the automatic derivation of performance measures too.

The rest of the paper is organised as follows. In Section 2, PEPA, and its use for performance modelling, are briefly revised. The current approach to deriving performance measures from PEPA models, and its problems, are described. Section 3 discusses some requirements of a formal method for specifying performance measures, and the main contribution of this paper is the use of the logic in the language presented in Section 4. After giving a formal meaning to the language, its use is demonstrated in examples presented in Section 5. In Section 6 some details of the implementation work are given, and the paper concludes with an outline of future work in Section 7.

2 Background

The work in this paper is in the context of a particular stochastic process algebra, Hillston's PEPA. Its features are described next.

2.1 PEPA

Classical process algebras such as CCS [7] and CSP [8] disregard the notion of time and model functional behaviour only. PEPA extends these algebras by associating a random variable, which represents a duration, with every action type. Therefore activities take time, and the performance of components can be studied. The random variables are chosen to be exponentially distributed to give a clear relationship between the SPA model and a continuous time Markov process. It is from the underlying Markov process that performance measures can be calculated.

PEPA models are constructed from components, which are able to interact with each other. Each of these components is capable of performing activities. Formally, an activity $a \in \mathcal{A}ct$ is described as a pair (α, r), where $\alpha \in \mathcal{A}$ is the type of the activity, known as the *action type*, and $r \in \mathbb{R}^+$ is the activity rate. $\mathbb{R}^+$ is defined as the set of positive real numbers together with the symbol $\top$. This symbol denotes undefined activity rates. These activities may only take place in synchrony with another activity of the same type, whose rate is not undefined. When r is defined, its value is the parameter of an exponential distribution and thus governs the duration of the activity. The set of all syntactic action types is denoted by $\mathsf{action\text{-}type}$.

In typical process algebra style, a small set of combinators is used to construct larger process algebra terms from smaller ones—this is the compositional style. These combinators include sequential composition $((\alpha, r).P)$, synchronisation $(P \underset{L}{\bowtie} Q)$, encapsulation

(P/L), and equational definition ($P \stackrel{\text{def}}{=} Q$). The set of all syntactically well-formed PEPA terms is denoted by process.

Some definitions from [3] will be used in this paper, and they are introduced here. A PEPA process P may perform a (α, r) activity and evolve into Q. In this case $P \xrightarrow{(\alpha,r)} Q$, and Q is a (one-step) *derivative* of P. The relation denoted by $\xrightarrow{(\alpha,r)}$ is called the *transition relation.* In similar style to CCS, PEPA defines an encapsulation operator. If $P \xrightarrow{(\alpha,r)} P'$ and $\alpha \in L$, then $P/L \xrightarrow{(\tau,r)} P'/L$; the α action type has been *hidden.* Given a PEPA process $P \stackrel{\text{def}}{=} Q \underset{L}{\bowtie} R$, components Q and R are said to be *top-level components.* Further, for any PEPA process P, the *derivative set* of P, $ds(P)$, is the least set of derivatives closed under the $\xrightarrow{(\alpha,r)}$ relation, and it thus captures all the reachable states of the system i.e.

- ▹ if $Q \stackrel{\text{def}}{=} P$, then $Q \in ds(P)$
- ▹ if $Q \in ds(P)$, and $Q \xrightarrow{(\alpha,r)} Q'$ for some (α, r), then $Q' \in ds(P)$

PEPA has a formal operational semantics and this associates a labelled *multi*-transition system with each PEPA process expression. A labelled transition system is a triple $(S, T, \longrightarrow)$ where S is a set of states, T a set of transition labels, and $\longrightarrow \subseteq S \times T \times S$ is the transition relation. With PEPA, the states are syntactic process expressions, the transition labels are the (α, r) activities, and the transition relation is given by the operational rules. However a multi-transition relation must be used because the timing behaviour of a process will depend on the number of instances of an activity that are enabled.

Finally, the rate at which one PEPA component is able to evolve into another is defined. Given two PEPA components, P and P', such that P' is a one-step derivative of P, the *transition rate* between the two is denoted $q(P, P')$. Thus the rate at which a component P may evolve into component P' is defined as the sum of the rates of the activities that relate the two i.e.

$$q(P, P') = \sum \{\!| \, r \mid P \xrightarrow{(\alpha,r)} P' \, |\!\}$$

2.2 Reward Structures

When the mathematical structure underlying a performance model is Markovian, its quantitative behaviour can be calculated using *reward structures.* These are introduced by Howard [9], and provide a very general framework for specifying and deriving performance measures. Howard actually defines reward structures with respect to continuous time semi-Markov processes; in this full generality, a reward structure is:

- ▹ a *yield* function, $y_{ij}(\sigma)$; while the process occupies state i of the semi-Markov process, having chosen a successor state j, it earns reward at a rate $y_{ij}(\sigma)$ at a time σ after entering the state.

▷ a *bonus* function, $b_{ij}(v)$; when the transition from state i to state j is made at some time v, the process earns the fixed reward $b_{ij}(v)$.

However, a PEPA model yields a continuous time Markov process, and the full generality of the reward structure is not used here. In fact, for the simple formalisation described here, it will suffice to fix the yield rate to be constant during the occupancy of a state; of course, the yield rate also cannot depend on the successor state. This gives

$$y_{ij}(\sigma) = y_i$$

Bonus functions are discarded; therefore, a reward structure will consist of a constant yield function only. This corresponds with the information obtained by solving a PEPA model, namely the steady state probability distribution. Useful performance measures can be calculated by combining the distribution data with the yield function, since both relate to the state space of the model. However it is worth noting that in the case of the CTMC, the distinction between bonus rates and yield rates can be overcome, and bonuses can be represented by yields; see [10].

Different modelling paradigms use reward structures too. For instance, stochastic Petri nets will often define the reward structure in terms of the reachability graph, which can be generated given a particular initial net marking. In this way interesting performance measures can be calculated from a SPN model. For example, consider the simple stochastic Petri net model given in Figure 1.

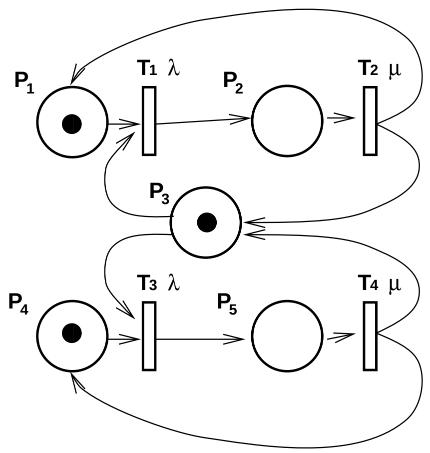

Figure 1 A simple stochastic Petri net example

Place P_3 acts as a semaphore, and ensures that places P_2 and P_5 cannot both contain tokens at the same time. In this way, the Petri net can be seen to model two processors, where each wishes to gain exclusive access to "common memory". The reachability graph is simple in this case, and is shown in Figure 2.

An example of an interesting performance measure may be the throughput of accesses to common memory. The reward structure is simple in this case—a reward is assigned to each state that enables transition T_2 or T_4; the value assigned is the rate at which the state in the reachability graph is left.

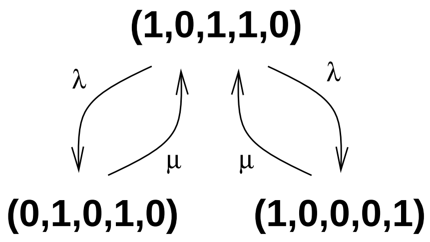

Figure 2 The reachability graph

Since typically, stochastic process algebras also generate a Markovian model, rewards can be adapted for use with this method of modelling too. In the literature, several systems have been described using PEPA, and reward structures have been constructed 'manually' (by examining regular patterns in the state space of process expressions) and then used to calculate pertinent measures, such as throughput and utilisation, e.g. [11]. Work has also been done on creating a unified approach to the specification of performance measures using models called *stochastic activity networks*, an extension of Petri nets. In [12], reward structures quantify performance measures at the level of the model; there is also a useful description of the types of rewards they are able to capture in general.

2.3 The PEPA Workbench

In a practical field such as performance modelling, models of real-life systems are likely to become very large—this is perhaps inevitable due to the state space explosion caused by modelling components concurrently. As a result, tool support is vital in generating and manipulating the Markov model. To this end, the PEPA Workbench was developed. It allows models to be defined in the syntax of PEPA. These are then processed by the Workbench which generates a representation of the *infinitesimal generator matrix*. This is in a form suitable for analysis by several computer algebra and especially linear algebra packages. Thus the Markov process can be solved, and a steady state probability distribution calculated. However, at the moment, the Workbench does not support the automatic calculation of performance measures—this is done manually by the user.

Calculating Rewards

The following procedure may be used to calculate performance measures:

- ▹ Create a model and submit it to the PEPA Workbench for analysis. This will produce a representation of the state space, and the infinitesimal generator matrix. Submit the matrix data to a linear algebra package to generate the steady state distribution. This is all completely automated.
- ▹ Form the reward structure over the state space which will generate the performance measure of interest. This involves deciding which states of the (perhaps huge) state space require consideration, and choosing values to be associated with each of these states. For instance, when calculating utilisation of a resource, the value 1 should be associated with each state in which use of the resource is "enabled", and the value

0 associated with all other states. However, often the value will not be as simple as a constant, and may depend on other factors. This is not done automatically. Once a reward value is associated with every state a reward vector can be formed.

- ▹ Use the steady state probability distribution to calculate the total reward. If a vector of the reward values for each state has been produced, this can be done automatically in whatever linear algebra package is being used.

A Simple Example

Here, an example of the calculation of a performance measure is given, which illustrates the problems with a lack of automation.

$$\begin{aligned} Worker &\stackrel{\text{def}}{=} (\texttt{holding}, \texttt{whr}).(\texttt{idling}, \texttt{ir}).Worker \\ Resource &\stackrel{\text{def}}{=} (\texttt{holding}, \texttt{rhr}).(\texttt{update}, \texttt{ur}).Resource \\ System &\stackrel{\text{def}}{=} Worker \underset{\{holding\}}{\bowtie} Resource \end{aligned}$$

Consider agent *System* above. Suppose the throughput of the constituent process *Worker* was required. A tool such as the PEPA Workbench would generate a representation of the *state space* that may look something like

```
1       Worker|Resource
2       (idling, ir).Worker|(update, ur).Resource
3       Worker|(update, ur).Resource
4       (idling, ir).Worker|Resource
```

Now, the user would realise that the throughput of *Worker* is determined by either of its two activities, since they are sequential. Thus, with each state that enables, say the `holding` activity, the user would associate the rate at which the state may change to another via an activity of type `holding`. How does the user determine which states enable the `holding` activity? At the moment, this is done by inspection.

2.4 The Problem

The following practically motivated problems are evident :

- ▹ Calculating measures manually is very error-prone.
- ▹ What if the model is prohibitively large to analyse by hand? General purpose tools may help (for instance the UNIX command `grep`), but may give the user a false sense of confidence in the correctness of the analysis. It is certainly easy to accidentally "miss out" enabling states from calculations leading to incorrect performance measures.
- ▹ What if the performance measure relies on more complicated process behaviour? It may not be possible to enlist the help of general purpose tools in this situation, and the only option would be a long and arduous manual analysis. The value associated with these states may be a complicated expression, meaning an even higher risk of error.

Furthermore, focussing on formalising reward specification with respect to PEPA models should prove interesting for a more aesthetic reason. When working with models, especially since at a low-level they are viewed as labelled transition systems, the concept of the state space is frequently useful. A reward structure may be only a (yield) function that assigns a value to each state of the system (and thus would neglect bonus rewards). This is the view used when working with, for instance, stochastic Petri nets. However, specifying a performance measure in this way is at odds with the process algebra style of modelling. A good exposition of the differences between these two formalisms is given in [13]. Central to the concept of a process algebra model is its behaviour; therefore, at a high-level, the focus is on what the system can do at each stage of its life, rather than in which stage of its life it is currently. Therefore, it would seem that the traditional notion of reward structure would have to be forced to sit with the process algebra view. Much better would be a formalism that assigned rewards by partitioning the process with an appropriate behavioural specification.

Another problem with specifying reward structures at the state space level is that they would make no use of the extra information provided by the process algebra modelling approach, specifically compositionality. Typically, a process algebra model will make explicit the disparate components of a model, since each component will be modelled in a reasonably independent fashion. The complete model will then consist of the "composition" of these components, which will allow them to communicate and cooperate. However, this information is lost at the state space level. It is clear that taking advantage of compositionality would prove useful—rewards could be defined with respect to a single component only (e.g. to determine the throughput of a particular processor in a model containing several), and with an appropriate formal foundation, could then be derived as part of the larger model.

Therefore, it is useful to investigate a behaviour-based approach to defining performance measures.

3 Requirements for Reward Specification

To formalise and automate the derivation of performance measures, the following is required

- ▹ A method of unambiguously and formally specifying which states of a process are those of interest.
- ▹ A method of associating a value with those states.

The total reward (thus the performance measure) can then be calculated using a simple "sum of products" expression. Ideally, the specification should be based on the behaviour of the system under consideration. This method would be more at ease with a process algebra definition of a model. This could be achieved by developing a higher-level syntax which allows the user to focus on the model behaviour, and less on any thought of state. The calculation of the measure will be a simple job given the information from a steady state analysis and the reward specification procedure. However, the first two points above require more consideration. Firstly, in general the specification of performance measures may become more complicated. Certainly, it seems that most useful measures (for instance throughput, utilisation) can be specified very simply—they typically only rely on the one-step behaviour from each process algebra derivative. Therefore, many simple measures can be defined by considering activities that are enabled in particular model derivatives, and not looking beyond this. But the required rewards may not be this simple, and any formalism should be capable of expressing measures relying on more than one-step behaviour. Secondly, how complicated are the values that can be assigned to states? Perhaps this is much less of an issue, since it is not problematic to allow any reasonable (computable) expression. There is a question over what model data is available for use in any such expression; for instance, it will certainly be necessary to be able to consider rates of activities enabled by the PEPA model and its derivatives.

In order to help develop an appropriate formalism, some useful performance measures were identified—these can be classified as follows :

- *rate* type.
- *counting* type.
- *delay* type.
- *percentage* type.

An example of a rate type performance measure would be the throughput of a server or the mean number of messages sent along a network per unit time. An example of the counting type would be the mean number of items in a queue, or the mean number of failed computers in a network. An example of the delay type would be the mean length of time a resource is used, or the mean time taken for message transmission. Finally, an example of a percentage type would be the percentage of time a resource spends in use. Certainly any method of specification should at least be capable of expressing useful measures of these types. This classification may not be complete, and could accurately be described as heuristic. However, by basing the specification technique on reward structures, it will be seen that a large class of these types of measures will be subsumed by the one method. This is due to the flexibility afforded by the use of models as general as reward structures.

It should be remembered that the primary motivation for this work is pragmatic in nature—a major aim was that it was suitable for implementation in the PEPA Workbench, and would thus provide an automated method for specifying and then calculating performance measures. This goal highlights an important concern in the design of the procedure. When solving Markovian models, software tools generate the entire state-space of the model. For instance, in the PEPA Workbench, this is necessary in order to create the infinitesimal generator matrix. It would be unfortunate if the process of assigning rewards should mean a second traversal of the entire state space after it is constructed in its entirety—this would be inefficient. Therefore a method that could assign rewards "on the fly" as the state space is generated would be preferable since it would avoid the need to traverse the state-space again, after generation. Added emphasis is placed on this observation by the fact that many recent Markovian models generate huge state-spaces, certainly in the order of hundreds of thousands of states. Thus a major concern in the design of the procedure is that it allows an efficient implementation.

4 The Reward Language

Using the simple language described here, the process of specifying performance measures is split into two stages:

- ▹ Defining a *reward specification*, which associates a value with a particular process derivative, if it is capable of behaving as required.
- ▹ Defining an *attachment* which determines with which process derivatives a particular reward specification is associated.

The meaning of the reward specification will depend on how it is "attached" to a PEPA model; this is because the associated value may depend on information local to the derivative under consideration. This will be explained when the semantics of the reward language is described in Section 4.3.

Formally, each reward specification can be considered as a pair consisting of a logical formula and a reward expression. If the particular derivative under consideration is capable of the behaviour described in the logical formula, then a reward is assigned to that derivative. The value of the reward corresponds to the evaluation of a simple arithmetic expression.

Before defining the syntax of the reward language, it will be necessary to describe the particular logic used in reward specifications.

4.1 Logic Syntax

The formalism chosen for defining properties of processes is based on a simple modal logic called Hennessy-Milner logic (HML, [14, 15]). Formulas are built from Boolean connectives and modal operators; the syntax is given below.

$$\Phi \ (\mathsf{formula}) ::= \mathtt{tt} \mid \mathtt{ff} \mid \neg\Phi \mid \Phi_1 \wedge \Phi_2 \mid \Phi_1 \vee \Phi_2 \mid [K]\Phi \mid \langle K \rangle \Phi$$

K ranges over subsets of action *types*, that is $K \subseteq \mathcal{A}$. HML only allows a single action type to appear in a modality; thus this logic generalises HML. However this logic is capable only of describing qualitative behaviour; there is no provision for specifying properties involving timing. Since, as described in Section 2.1, a PEPA process can be regarded as a labelled multi-transition system, formulas are interpreted with respect to this slightly more elaborate model.

4.2 Logic Semantics

The particular semantics chosen are simple and reflect the intended use of the logic in specifying simple behavioural properties of PEPA processes. A formula will be interpreted in conjunction with a particular state of the labelled multi-transition system. The evaluation of such a pair will result in a Boolean value. The notation for the evaluation of a formula Φ at a particular state s is $\| \Phi \|_s$, and is given inductively on the structure of Φ. The semantics are given in Figure 3.

$$
\begin{aligned}
\| \mathtt{tt} \|_s &= T(\in \{T, F\}) \\
\| \mathtt{ff} \|_s &= F \\
\| \neg\Phi \|_s &= \neg \| \Phi \|_s \\
\| \Phi_1 \vee \Phi_2 \|_s &= \| \Phi_1 \|_s \vee \| \Phi_2 \|_s \\
\| \Phi_1 \wedge \Phi_2 \|_s &= \| \Phi_1 \|_s \wedge \| \Phi_2 \|_s \\
\| [K]\Phi \|_s &= \bigwedge\{\| \Phi \|_{s'} \colon s \xrightarrow{(\alpha,r)} s', \alpha \in K\} \\
\| \langle K\rangle\Phi \|_s &= \bigvee\{\| \Phi \|_{s'} \colon s \xrightarrow{(\alpha,r)} s', \alpha \in K\}
\end{aligned}
$$

Figure 3 Semantics of the logic

No distinction has been made between Boolean operators and the operators of the modal logic—the context is sufficient for distinguishing which meaning is in use. The intuitive meaning of the modal operators is clear. $\| [K]\Phi \|_s$ is true if *for all* states s' such that there is a transition from s to s' via an activity whose type is in K, $\| \Phi \|_{s'}$ is true. Similarly, $\| \langle K\rangle\Phi \|_s$ is true if *there exists* a state s', such that there is a transition from s to s' via an activity whose type is in K, and $\| \Phi \|_{s'}$ is true. These operators do not make use of the fact that the semantics of PEPA generate a labelled *multi*-transition system. Since the logic only captures qualitative behaviour, the timing differences that result when states are joined by multiple identical activities are immaterial.

Next, the syntax and semantics of the reward expressions are defined. In this paper, the syntactic class of alphanumeric sequences is denoted by string, and the syntactic class of expressions representing real numbers is denoted by real; these require no formal definition here.

4.3 Syntax and Semantics of Reward Expressions

The syntax of reward expressions is very simple, indeed it captures little more than a trivial syntax for arithmetic. The only additions to this are three bound variables; the syntax is given below.

$$
\begin{aligned}
e\ (\mathsf{expr}) &::= (e) \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 * e_2 \mid e_1/e_2 \mid \mathsf{atom} \\
\mathsf{atom} &::= \mathsf{real} \mid m_c \mid m_p \mid rate(\mathsf{action\text{-}type})
\end{aligned}
$$

The bound variables, named m_c, m_p and $rate()$, will be used to denote real numbers. Their meaning will be dependent on the rewards already calculated, and the particular labelled multi-transition system which results from the PEPA model under consideration. They exist for pragmatic reasons—they are useful in specifying performance measures. These simple expressions are given a semantics below. Because the meanings of the bound variables depend on assigned rewards and activity rates, the semantic function is decorated with information from the transition system.

First, a reward function is defined. If P is the PEPA model under consideration, then let $\rho : ds(P) \longrightarrow \mathbb{R}$, where $ds(P)$ is the derivative set of P. Therefore, given a derivative (in fact a state of the transition system) ρ gives the reward assigned to that derivative.

Given this reward assignment function, the semantic function relies on a pair of PEPA processes (derivatives) to define the meaning of reward expressions; the semantics are given in Figure 4.

$$
\begin{array}{rcl}
\text{let } h & \equiv & (P_p, P) \text{ ; then} \\
\| (e) \|_h & = & \| e \|_h \\
\| e_1 \; op \; e_2 \|_h & = & \| e_1 \|_h \; op \; \| e_2 \|_h \\
\| m_c \|_h & = & \rho(P) \\
\| m_p \|_h & = & \rho(P_p) \\
\| rate(\alpha) \|_h & = & \sum \{ q(P, P') \mid P \xrightarrow{(\alpha, r)} P' \text{ for some } \mathtt{r} \}
\end{array}
$$

Figure 4 Semantics of reward expressions

P_p is named such that it suggests a "previous" derivative. Thus, in order to assign a reward to a particular derivative P, it may be necessary to know a derivative P_p such that $P_p \xrightarrow{(\alpha,r)} P$. Thus, m_c and m_p are intended to give the reward expression access to any previously assigned, and current rewards, allowing reward expressions to make use of a reward assignment made to a prior, or to the current derivative. The variable $rate()$, allows activity rates to be used in expressions—specifically, reward values can be assigned to a derivative P which make use of the transition rate from P to successor derivatives via an activity of type α. This is the way in which timing information may be incorporated into reward specifications. The binary operator *op* is intended to capture the obvious binary operators defined in the syntax above.

Given the semantics for the logic, and for reward expressions, it is now simple to state the syntax of the PEPA reward language. Given this, the definitions given in the preceding sections will be tied together with the reward language semantics, which will illustrate how the language is to be used to specify performance measures.

4.4 Reward Language Syntax

The syntax of reward specifications is given below.

$$
\begin{array}{rcl}
\textsf{reward} & ::= & \text{“reward”} \; \textsf{string} \; \text{“=”} \; \textsf{reward\_spec} \\
\textsf{reward\_spec} & ::= & \textsf{formula} \; \text{“}\Rightarrow\text{”} \; \textsf{expr}
\end{array}
$$

However, these rewards do not mean anything in isolation; a method of associating them with processes is required, as stated in Section 4. This requires an explanation of *attachments*; the syntax is given below.

$$
\begin{array}{rcl}
\textsf{attachment} & ::= & \text{“attach”} \; \textsf{string} \; \text{“to”} \; \textsf{select-process modifiers} \\
\textsf{select-process} & ::= & \textsf{process}\text{“(”} \; \textsf{nat} \; \text{“)”} \\
\textsf{modifiers} & ::= & [\text{“depth first”} \mid \text{“breadth first”}]
\end{array}
$$

Thus, an attachment is a method of examining the derivative set of a PEPA process (or at a lower level, traversing the state space of the multi-transition system). The reward specification should be evaluated at a start point i.e. an initial PEPA process, and then the attachment used to determine for which derivatives the procedure should be reapplied. Thus the reward assignment procedure is iterative. Of course, this vague description is less than satisfactory. Since the meaning of the individual components used in the procedure have been more formally defined, a semantics is given below to illustrate the effect of the reward assignment procedure.

4.5 Reward Language Semantics

The semantics given to the reward language depends on two further definitions. These help capture the idea that rewards can be attached to components of PEPA processes, and thus allow the structure of the process algebra model to be exploited when specifying rewards. For instance, given a large PEPA model, it may be interesting to only examine the performance of a single component queue. It should be possible to disregard the behaviour of the rest of the model, at least up to its interaction with the queue under examination. First a notion is required for the *context* of a component.

A PEPA *context* is a place-holder (*hole*) [.] in a PEPA process. The syntax of contexts is given below.

$$c\ (\mathsf{context}) ::= [.] \mid (\alpha, r).c \mid P + c \mid c + P \mid c/L \mid P \underset{L}{\bowtie} c \mid c \underset{L}{\bowtie} P$$

This commonly used syntax for contexts is unfortunate due to its clash with the syntax of the modal logic. A place-holder in a context, [.], should not be confused with the modal box operator $[\alpha]$. A process P in context c, $c[P]$, is expressible as one of

1. P
2. $(\alpha, r).c'[P]$
3. $c'[P] + Q$ (or $Q + c'[P]$)
4. $c'[P]/L$
5. $c'[P] \underset{L}{\bowtie} Q$ (or $Q \underset{L}{\bowtie} c'[P]$)

for some smaller context c'. For example, the context of P in $(Q + P) \underset{L}{\bowtie} R$ is $(Q + [.]) \underset{L}{\bowtie} R$. A context is called *dynamic* if it is an instance of item 3, and is termed *static* if it is an instance of item 4 or 5. It is an *empty* context if it is an instance of item 1. Further, a context c is called *immediate* if $c[P]$ is such that c' is empty. Therefore, if P is a top-level component in a PEPA process, a PEPA context with a place-holder for P would be an immediate context. Thus putting a process P in a context means constructing a larger model by filling in the contextual "hole" with P. Furthermore, a *contextual transition relation* is defined which captures the notion of a subcomponent proceeding in context. This is illustrated in Figure 5.

$$
\begin{aligned}
&P \xrightarrow{(\alpha,r)}_c P' \text{ if } c \text{ is immediate and}\\
&\qquad\qquad \text{if } c \text{ is dynamic and } c[P] \xrightarrow{(\alpha',r')} P'\\
&\qquad\qquad \text{or if } c \text{ is static and } c[P] \xrightarrow{(\alpha',r')} c'[P']\\
&\qquad \text{or if } c \text{ is not immediate and}\\
&\qquad\qquad \text{there exists } c_1, c_2 \text{ such that } c_1 \text{ is not empty, } [P] \equiv c_1[c_2[P]] \text{ and}\\
&\qquad\qquad c \text{ is dynamic and } c_2[P] \xrightarrow{(\alpha',r')}_{c_1} P'\\
&\qquad\qquad \text{or } c \text{ is static and } c_2[P] \xrightarrow{(\alpha',r')}_{c_1} c_2[P']\\
&\text{where } \alpha' = \tau \neq \alpha \Rightarrow c[P] \equiv c_1[c_2[P]/L] \text{ for some } c_1, c_2 \text{ and } L, \text{ with } \alpha \in L\\
&\text{and } r' \neq r \Rightarrow c[P] \equiv c_1[c_2[P] \underset{L}{\bowtie} Q] \text{ for some } c_1, c_2, Q \text{ and } L, \text{ with } \alpha \in L
\end{aligned}
$$

Figure 5 The Definition of the Contextual Transition Relation

The intuition behind this is simple—a subcomponent can proceed in context only if its context allows it. Because of the operational semantics of PEPA, a process can perform an activity if and only if its components interact in such a way that it can. For instance, let $P \stackrel{\text{def}}{=} Q \underset{L}{\bowtie} R$. Clearly P can only perform an activity if either of its subcomponents Q or R can independently perform an activity, or if Q and R can cooperate to perform an activity. Thus, given the context $c \equiv [.] \underset{L}{\bowtie} R$,

$$Q \xrightarrow{(\alpha,r)}_c Q' \text{ if and only if } Q \xrightarrow{(\alpha,r)} Q' \text{ can be used to derive } Q \underset{L}{\bowtie} R \xrightarrow{(\alpha',r')} Q' \underset{L}{\bowtie} R$$

for some appropriate activity (α', r'). Note that in the above definition, it is neither required that $\alpha = \alpha'$, nor $r = r'$. Intuitively this is because when a subcomponent contributes to the evolution of a system, the system's environment may not see the type of the activity if it is *hidden*, or may witness the activity occur at a different rate if the subcomponent *cooperates* with another. Therefore, the contextual transition relation captures the ability of a process to evolve within the context of a larger model. This is necessary for calculating rewards that are attached to subcomponents of a larger process.

Finally, before the semantics of the reward language is given, a syntactic method for selecting a subcomponent of a model is described. At the moment, this is done in a naïve manner by using a natural number to "pick out" the component of interest from the larger model. The function count below identifies the number of syntactic instances of a subcomponent in a larger model. The function takes two arguments—the first is the PEPA model, and the second is the subcomponent.

$$
\begin{aligned}
count(Q, Q) &= 1\\
count((\alpha, r).P, Q) &= count(P, Q)\\
count(P + R, Q) &= count(P, Q) + count(R, Q)\\
count(P/L, Q) &= count(P, Q)
\end{aligned}
$$

$$count(P \underset{L}{\bowtie} R, Q) \;=\; count(P, Q) + count(R, Q)$$

Now, using *count*, a function *nth* is defined which given arguments n and processes P and Q picks out the nth syntactic subcomponent of Q in P. The function returns a context c as defined above, such that $c[Q] \equiv P$.

$$\begin{array}{rcl}
nth(1, Q, Q) & = & [.] \\
nth(i, (\alpha, r).P, Q) & = & (\alpha, r).(nth(i, P, Q)) \\
nth(i, P + R, Q) & = & nth(i, P, Q) + R \text{ if } count(P, Q) \geq i \\
& \text{else} & P + count(i - count(P, Q), R, Q) \\
nth(i, P/L, Q) & = & nth(i, P, Q)/L \\
nth(i, P \underset{L}{\bowtie} R, Q) & = & nth(i, P, Q) \underset{L}{\bowtie} R \text{ if } count(P, Q) \geq i \\
& \text{else} & P \underset{L}{\bowtie} count(i - count(P, Q), R, Q)
\end{array}$$

These functions are used to give a meaning to expressions in the syntactic class select-process.

Now the semantics of the reward language can be defined. The semantic evaluation function assumes the existence of a reward specification, $r \equiv$ reward $\phi \Rightarrow e$, and a *sequence* of attachments $\langle a_1, a_2, \ldots, a_n \rangle$. A sequence is used because the evaluation of an attachment will typically generate several new attachments, each to be evaluated in turn. This corresponds to traversing the derivative space of the model. The order in which each attachment is evaluated may be important since rewards can depend on rewards assigned to "previous" derivatives. The complete PEPA model of interest is denoted by P, and any subcomponents are denoted by Q. The semantic function takes as an argument the reward assignment function ρ and evaluates to a new function, say ρ'. This possibly modified assignment function will reflect any new rewards that have been assigned to the PEPA model. Thus the semantics defines a reward assignment function that associates a reward with each derivative of the PEPA model. Evaluating the sequence of attachments is a simple job—each is evaluated individually, in order. This is shown below.

$$\begin{array}{rcl}
\| \langle \rangle \|_\rho & = & \rho \\
\| \langle a_i, a_{i+1}, \ldots, a_m \rangle \|_\rho & = & \| \langle a_{i+1}, a_{i+2}, \ldots, a_m \rangle \|_{\rho'} \;\text{ where } \rho' = \| a_i \|_\rho
\end{array}$$

Describing the evaluation of an attachment, call it a_i, is more complicated. The evaluation depends on

- ▷ the reward assignment function, ρ
- ▷ the PEPA model being studied, P
- ▷ a sequence of attachments yet to be evaluated, $\langle a_n, \ldots, a_p \rangle$
- ▷ the derivative to which the last reward was attached, denoted Q_p

The initial value of the reward assignment function is the mapping $\rho : ds(P) \longrightarrow \mathbb{R}$ such that $\rho(P) = 0$, for all P. This is chosen arbitrarily—however it is a reasonable and useful choice in practice. The initial sequence of attachments is the empty sequence, $\langle\rangle$. Initially, the derivative to which the last reward was assigned is undefined. Now, the attachment to be evaluated next is given below.

$$a_i = \text{ attach } r \text{ to } Q(j)\ modifier$$

The meaning of a_i given the data described above i.e. $\rho, P, \langle a_n, \ldots, a_p\rangle$ and Q_p can be given as follows:

$$\begin{aligned} \text{let } c &= nth(j, P, Q) \text{ and } h \equiv (Q_p, Q) \\ \text{if } \ \| \phi \|_Q &= T \\ \text{then } \rho' &= \rho[c'[Q] \mapsto \| e \|_h] \ \forall c' \text{ such that } c[Q] \xrightarrow{(\alpha, r)} c'[Q] \text{ for some } (\alpha, r) \end{aligned}$$

where ρ' is created by ordinary function perturbation; however, it is important to note that because, in general, the behaviour of a subcomponent will be analysed, when calculating the truth of the logical formula the ordinary transition relation arrow $\longrightarrow$ is replaced in the logic semantics by the *contextual transition relation arrow*, $\longrightarrow_c$. In this way, the logic semantics will capture the ability of a subcomponent to behave in context. If a reward value is assigned, the total effect of the above will then be that the current reward value will be propagated from the current derivative to those derivatives that represent further activity, but not by the subcomponent of interest. This makes intuitive sense—if a particular component is in a particular state, and earns a reward, the behaviour of the context of the component, if it doesn't involve it, cannot change the component's state. Therefore, the reward is propagated.

Now let $\langle a_1, \ldots, a_m\rangle$ be a sequence of attachments, in arbitrary order, such that

$$a_k \equiv \text{ attach } r \text{ to } Q_k(j)\ modifier \text{ , where } Q \xrightarrow{(\alpha, r)}_c Q_k \text{ for some } (\alpha, r)$$

This sequence of attachments represents the calculations that must be made for a new set of derivatives—those derivatives that result from the current subcomponent evolving in context. Rather than propagating a reward, it is necessary to test these new derivatives against the logical formula in the reward specification once more. So

$$\begin{aligned} \| a_i \|_\rho = \ & \| \langle a_1, \ldots, a_m, a_n, \ldots, a_p\rangle \|_{\rho'} \text{ where } Q_p = Q, \text{ if } modifier = \text{ depth-first} \\ & \| \langle a_n, \ldots, a_p, a_1, \ldots, a_m\rangle \|_{\rho'} \text{ where } Q_p = Q, \text{ if } modifier = \text{ breadth-first} \end{aligned}$$

Intuitively, the evaluation of the attachment traverses the PEPA model, while focussing on a particular subcomponent. The derivatives that may be assigned rewards are those derivatives that can be reached by following activities performed by the initial PEPA subcomponent *in the context of c*. Therefore, if the initial subcomponent may perform an activity such that in context the model evolves too, then the resulting new derivatives will subsequently be considered for a reward. Note that the PEPA model must be able to evolve as a result of the subcomponent evolving. For example, if the subcomponent Q may evolve into Q' by an activity of type α, if within the model Q is within the context of a cooperation not allowing an activity of type α, $c[Q']$ will not be tested against the reward specification.

5 Examples

In this section, the specification language is used to express performance measures of the types described in Section 3. Since 1996 saw the European Championships held in the United Kingdom, it seemed fitting to use a model of a football match as a vehicle for expressing performance measures. The first example is explained in some detail, in order to illustrate the semantics given in Section 4.5. Following this, some further examples are given to highlight the kinds of performance measures that can be calculated.

5.1 The Model

In order to model a football team, it would seem reasonable to first model the players. Since this will be a small example, a team is restricted to two players, a "defender" and an "attacker". The definitions are given in Figure 6.

$$
\begin{aligned}
Defender_i &\stackrel{\text{def}}{=} (\mathtt{pass}, \top).DribblingDefender_i + (\mathtt{trap}, \lambda_t).DribblingDefender_i \\
&\quad + (\mathtt{tackle}, \lambda_c).Defender_i \\
DribblingDefender_i &\stackrel{\text{def}}{=} (\mathtt{pass}, \lambda_p).Defender_i + (\mathtt{tackle}, \top).Defender_i \\
\\
Attacker_i &\stackrel{\text{def}}{=} Defender_i[Attacker/Defender] \\
DribblingAttacker_i &\stackrel{\text{def}}{=} DribblingDefender_i[Attacker/Defender] \\
&\quad + (\mathtt{goal_i}, \lambda_g).Attacker_i \\
\\
Team_i &\stackrel{\text{def}}{=} Attacker_i \underset{\{pass,tackle\}}{\bowtie} Defender_i
\end{aligned}
$$

Figure 6 Definition of a football team

The "[a/b]" notation denotes simple text substitution of "a" for "b", and simplifies the reading of the definitions. A player may pass the ball to his[1] teammate, may "trap" (and thus gain possession of) a loose ball, or may tackle an opponent. A forward has an additional capability, that of scoring a goal. The synchronisation set of $Team$ ensures the ball is only passed to teammates, and that a tackle cannot be made without the team losing possession of the ball. Note of course that the player with the ball controls the rate at which the ball is passed.

[1] of course we imply "his/her" throughout this paper

This model can be developed into a match with the additional definitions given in Figure 7.

$$
\begin{aligned}
Ball &\stackrel{\text{def}}{=} (\texttt{trap}, \top).Possession \\
Possession &\stackrel{\text{def}}{=} (\texttt{tackle}, \top).Ball + (\texttt{goal}, \top).(\texttt{kickoff}, \top).Ball \\
Goals_i &\stackrel{\text{def}}{=} (\texttt{goal}, \top).(\texttt{roar}_{\texttt{3-i}}, \lambda_r).(\texttt{kickoff}, \lambda_k).Goals_i \\
Euro96 &\stackrel{\text{def}}{=} (Team_1 \underset{L}{\bowtie} Team_2) \underset{M}{\bowtie} Ball \underset{N}{\bowtie} (Goals_1 \underset{\emptyset}{\bowtie} Goals_2) \\
&\quad \text{where } L = \{\texttt{tackle}\}, M = \{\texttt{trap}, \texttt{tackle}, goal_i\} \text{ and } N = \{\texttt{goal}_\texttt{i}, \texttt{kickoff}\}
\end{aligned}
$$

Figure 7 Definition of a football match

A match consists of two teams, two goals and a ball. One team may tackle another in which case the ball is free and either team may gain possession of it. Each team may score a goal; if one does it is imagined the crowd will roar its delight.

Now a simple measure is defined for the model *Euro*96 and explained in some detail. Then the specification of some example performance measures is given using the reward language.

5.2 A Simple Performance Measure—Percentage Possession

A statistic commonly seen on television is the percentage possession of each team. It would be interesting to start with the percentage of time a particular team, say $Team_1$, has possession of the ball. The strategy is to define behaviourally when $Team_1$ has possession; it can be seen that this is when they are able to score a goal (the attacker has the ball), or when they are able to make a pass such that they are then able to score a goal (the defender has the ball). Therefore a valid reward specification is:

$$
\begin{aligned}
\texttt{reward perc\_possession} &= \langle \texttt{goal}_\texttt{i} \rangle \texttt{tt} \\
&\vee \ \langle \texttt{pass} \rangle \langle \texttt{goal}_\texttt{i} \rangle \texttt{tt} \Rightarrow 1
\end{aligned}
$$

Of course this captures those times when *either* team has the ball. Therefore, to focus on $Team_1$ only, the following attachment is required:

$$\texttt{attach perc\_possession to } Team_1$$

It is worth describing in some detail how the semantics described earlier can be used to calculate a value for this performance measure. The procedure begins by considering the "initial state", that is the process *Euro*96. It is assumed that the reward assignment function $\rho : ds(Euro96) \longrightarrow \mathbb{R}$ is such that $\rho(P) = 0$, for all $P \in ds(Euro96)$. Now the reward specification is evaluated. This corresponds to evaluating

$$\langle \mathtt{goal_i} \rangle \mathtt{tt} \vee \langle \mathtt{pass} \rangle \langle \mathtt{goal_i} \rangle \mathtt{tt}$$

for $Team_1$ in the context

$$([.] \underset{L}{\bowtie} Team_2) \underset{M}{\bowtie} Ball \underset{N}{\bowtie} (Goals_1 \underset{\emptyset}{\bowtie} Goals_2)$$

Intuitively, $Team_1$ is incapable of the described behaviour in the given context—in the initial state, neither team has the ball, and thus neither team may pass the ball or score a goal. It is necessary to first witness an activity of type `trap` ensuring a team gains possession of the ball. Therefore, a reward is not assigned to this derivative.

Following the semantics, it can be seen that $Team_2$ can perform an activity of type `trap` while $Team_1$ remains inactive. This means that $Team_2$ has won the ball. From this position $Team_2$ may pass the ball (the `pass` activity), and perhaps eventually score a goal (the `goal` activity). The derivatives in which these activities are possible all contain $Team_1$ as a subcomponent, therefore, it has not evolved. Thus these derivatives all correspond to those that will not be assigned a reward value. This is correct since it is intended that a reward is assigned only if $Team_1$ has possession.

Now, it is also possible that $Team_1$ may gain possession of the ball from the start, by performing an activity of type `trap` itself. The derivative that results must be tested for a reward since $Team_1$ has evolved in context. But now by a simple inspection, it can be seen that in the context of the football game, after performing an activity of type `trap`, $Team_1$ satisfies the logical formula $\langle \mathtt{pass} \rangle \langle \mathtt{goal_i} \rangle \mathtt{tt}$; it is certainly possible to pass due to internal cooperation between *Attacker* and *Defender*, and following this, it is possible to score by "cooperation" with $Goal_2$ ($Team_2$'s goal). Therefore, this derivative will be assigned a reward, and this is correct because at this point, $Team_1$ clearly has possession.

In this vein, rewards are propagated to derivatives that correspond to the game evolving and the subcomponent remaining the same, and the reward specification is tested against those derivatives that result from the subcomponent evolving in the context of the game. The effect is that derivatives modelling $Team_1$ in possession will be assigned a reward of 1, and all others will receive a value of 0. By computing the described sum of products expression given the steady state vector, this can be interpreted as the percentage of time $Team_1$ has possession of the ball.

5.3 A Throughput Measure

Another obvious question to be asked of this model is "what is the final score?". This can be calculated by discovering how often a goal is scored by each side. Of course the model is solved for steady state, but it is possible to calculate "a number" of goals scored since it is possible to calculate the number of goals scored per unit time. Alternatively, the ratio of the team's goals could be calculated to ascertain which team is superior.

Specifying the reward for this measure is particularly simple. The reward to specify the number of goals $Team_1$ scores is given below:

$$\texttt{reward score } = \langle \texttt{goal}_2 \rangle \texttt{tt} \Rightarrow rate(\texttt{goal}_2)$$

If the derivative can perform an activity of type $\texttt{goal}_2$, then it will be assigned the value of $rate(\texttt{goal}_2)$. This will evaluate to the sum of the rates at which the derivative can evolve via an activity of type $\texttt{goal}_2$. Therefore the value will give a measure of the throughput of the $\texttt{goal}_2$ activity. A naïve attachment will do:

$$\texttt{attach score to } Euro96$$

This means that the reward specification will be evaluated against each derivative of the $Euro96$ model, and those derivatives that correspond to positions where $Team_1$ may score will be assigned a reward. The sum of products calculation will give the throughput of the $\texttt{goal}_2$ activity, indicating "how often" a goal is scored. By a similar performance measure in which $\texttt{goal}_1$ replaces $\texttt{goal}_2$, the rate at which $Team_2$ scores can be calculated, and the teams compared.

5.4 A Counting Measure

The PEPA Reward Language can also be used to calculate counting type measures. To illustrate this, consider the following definitions:

$$\begin{aligned}
Critic &\stackrel{\text{def}}{=} Critic_0 \\
Critic_0 &\stackrel{\text{def}}{=} (\texttt{tackle}, \top).Critic_0 + (\texttt{pass}, \top).Critic_1 \\
Critic_i &\stackrel{\text{def}}{=} (\texttt{tackle}, \top).Critic_{i-1} + (\texttt{pass}, \top).Critic_{i+1}, 1 < i < 5 \\
Critic_5 &\stackrel{\text{def}}{=} (\texttt{tackle}, \top).Critic_4 + (\texttt{pass}, \top).Critic_5
\end{aligned}$$

$$LiveMatch \stackrel{\text{def}}{=} Euro96 \underset{\{tackle, pass\}}{\bowtie} Critic$$

A football critic analyses the "quality" of the football. If passes are frequent, he is pleased and his rating of the game rises. If he witnesses frequent tackles, he is displeased and his rating falls. It may be interesting to calculate the rating from 1 to 5 which $Critic$ gives to the $Euro96$ football match. This can be done by basing the reward assigned to a derivative on the reward assigned to a previous derivative. For example, a particular derivative may correspond to the critic giving the game a rating of 3. By an activity of type $\texttt{pass}$, this should evolve into a derivative which the critic should rate as 4. Clearly, the reward assigned to the second derivative could be the value of the reward assigned to the first derivative, plus 1. This can be done using the following reward specification:

$$\texttt{reward match\_rating} = \langle \texttt{pass} \rangle \texttt{tt} \Rightarrow m_p + 1$$

Again a naïve attachment suffices—therefore each derivative is tested against the logical formula. Since the procedure begins with the initial derivative, by following the `pass` activities it is possible to assign higher ratings to those areas of the derivative space that are reached by a higher number of passes. The attachment is given below:

`attach match_rating to` $LiveMatch$

Therefore this performance measure implements a simple "counter".

6 Implementation

An empirical test of the utility of the PEPA reward language would be to use it in practice when modelling with PEPA. Therefore work is underway on an implementation of this formalism within the PEPA Workbench. As mentioned, at the moment there is no integrated method of specifying and calculating performance measures. The reward language attempts to bridge that gap; it remains to be seen how successfully it does so.

The PEPA Workbench is implemented in Standard ML (SML), a strongly-typed functional language with imperative features. SML has already been used successfully in the implementation of tools such as the Concurrency Workbench [16], and therefore it may be possible to re-use existing code. Of course, SML's features also reinforce confidence in the correctness of the implementation. Therefore, naturally SML was also used to extend the PEPA Workbench with the reward language. The application was restructured to take advantage of SML's module system, allowing modular and incremental development. This facilitated implementation work on the reward language, while keeping alterations to the Workbench itself to a minimum.

By using the MLLex and MLYacc tools along with the Moscow ML compiler, it was possible to parse, construct and evaluate reward specifications simply. An example reward spec is given in Figure 8; it is (a slight expansion of) the example given in Section 5.2, specifying that $Team_1$, a subcomponent of $Euro96$, has possession of the football.

```
reward perc_possession = ( <goal1>tt | <goal2>tt )
                         | <pass>( <goal1>tt | <goal2>tt ) => 1;

attach perc_possession to Team1;
```

Figure 8 A performance measure specified in the extended PEPA Workbench

At the moment there are restrictions on what measures it is possible to analyse, simply because the methods have not yet been implemented. This is the subject of work in progress. However, because the logic is reasonably simple (certainly it cannot express infinite, or temporal properties), it seems possible to model-check "on the fly", as the underlying model is being constructed. This is because each formula only requires "local" knowledge about the underlying transition system, since each formula has finite length. Therefore it seems that as the underlying states are generated, the formula in the reward specification can be evaluated too—if the truth of a formula is required for a particular derivative, only finitely (very few) transitions need be computed from that state before there is enough information to do this. This will avoid constructing the CTMC in one pass, and then computing rewards in another.

7 Future Work

There are many possible areas of future work:

- ▹ It would be useful if a way could be found to show that specifying performance measures in this way is, in some sense, "adequate". No claims are made in this paper as to how useful this method is—however, analysing simple PEPA systems shows that it can cope with specifying simple measures.
- ▹ This method does not cope with delay type measures—indeed, it seems that modelling using process algebras means that it is hard to derive measures such as "the average time between 2 activities".
- ▹ Although this method attempts to allow the modeller to specify measures using behavioural reasoning, it could be improved. Indeed, it would be useful if the modeller did not have to consider activities at all, rather asking questions such as "what is the throughput of this processor?"
- ▹ It is hoped that formalising the specification of rewards may lead to model simplification techniques. It may be the case that only considering a model with respect to a particular performance measure means that any states irrelevant to the value of the measure may be disregarded.
- ▹ It would be interesting to study transient analysis—many interesting measures can be calculated using transient analysis techniques.
- ▹ The implementation of this technique in the PEPA Workbench should be completed. This would give PEPA modellers the capability to automatically extract performance measures from their models.

Acknowledgements

I would like to thank my supervisor Jane Hillston for her guidance, and for her considerable assistance with this paper, especially towards its conclusion. I would also like to thank my friends, and peers, Steven Webster, Marcus Alphey, Jörj Taylor and Andrew Pearson for their corrections and entertaining modelling suggestions. My favourite was Andrew's kitchen scenario where deadlock occurred because his flatmate Dave "ate all the pies" (pies being a resource).

References

[1] U. Herzog. Formal Description, Time and Performance Analysis: A Framework. Technical Report 15/90, IMMD VII, Friedrich-Alexander-Universität, Erlangen-Nürnberg, Germany, September 1990.

[2] N. Götz, U. Herzog, and M. Rettelbach. Multiprocessor and Distributed System Design: The Integration of Functional Specification and Performance Analysis using Stochastic Process Algebras. In *Performance'93*, 1993.

[3] J. Hillston. *A Compositional Approach to Performance Modelling.* PhD thesis, The University of Edinburgh, 1994. CST-94-108.

[4] C. Stirling and D. Walker. Local model checking in the modal mu-calculus. *Theoretical Computer Science*, 89:333–354, 1989.

[5] H. Hermanns. Performance Prediction of Behavioural Descriptions with Temporal Logics. Extended Abstract.

[6] S. Gilmore and J. Hillston. The PEPA Workbench: A Tool to Support a Process Algebra-based Approach to Performance Modelling. In G. Haring and G. Kotsis, editors, *Proceedings of 7th Conf. on Mod. Techniques and Tools for Computer Perf. Eval.*, volume 794 of *LNCS*, pages 353–368, 1994.

[7] R. Milner. *Communication and Concurrency.* International Series in Computer Science. Prentice Hall, 2nd edition, 1989.

[8] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, Englewood Cliffs, NJ, 1985.

[9] R. A. Howard. *Dynamic Probabilistic Systems*, volume II: Semi-Markov and Decision Processes, chapter 13, pages 851–915. John Wiley & Sons, New York, 1971.

[10] B. R. Haverkort and K. S. Trivedi. Specification techniques for Markov reward models. *Discrete Event Dynamic Systems: Theory and Applications*, 3(2/3):219–247, July 1993.

[11] J. Hillston and M. Ribaudo. *Stochastic Process Algebras: A New Approach to Performance Modeling*, chapter 10.

[12] W. H. Sanders and J. F. Meyer. A unified approach for specifying measures of performance, dependability, and performability. *Dependable Computing and Fault Tolerant Systems*, 4, 1991.

[13] S. Donatelli, J. Hillston, and M. Ribaudo. A Comparison of Performance Evaluation Process Algebra and Generalized Stochastic Petri Nets. In *Proc. of the 6th Petri Nets and Performance Models Workshop*, pages 158–168, October 1995.

[14] M. Hennessy and R. Milner. On observing nondeterminism and concurrency. In J. W. de Bakker and J. van Leeuwen, editors, *Proceedings 7^{th} ICALP,* Noordwijkerhout, volume 85 of *Lecture Notes in Computer Science*, pages 299–309. Springer-Verlag, July 1980.

[15] M. Hennessy and R. Milner. Algebraic laws for nondeterminism and concurrency. *Journal of the ACM*, 32(1):137–161, January 1985.

[16] Rance Cleaveland, Joachim Parrow, and Bernhard Steffen. The concurrency workbench: A semantics based tool for the verification of concurrent systems. Technical Report ECS-LFCS-89-83, Laboratory for Foundations of Computer Science, Dept. of Computer Science, University of Edinburgh, 1989.