The PEPA Workbench: A Tool to Support a Process Algebra-based Approach to Performance Modelling

Stephen Gilmore and Jane Hillston

Department of Computer Science, The University of Edinburgh

Abstract. In this paper we present a new technique for performance modelling and a tool supporting this approach. Performance Evaluation Process Algebra (PEPA) [1] is an algebraic language which can be used to build models of computer systems which capture information about the performance of the system. The PEPA language serves two purposes as a formal description language for computer system models. The performance-related information in the model may be used to predict the performance of the system whereas the behavioural information in the model may be exploited when reasoning about the functional behaviour of the system (e.g. when finding deadlocks or when exhibiting equivalences between sub-components). In this paper we concentrate on the performance aspects of the language.

A method of reasoning about PEPA models proceeds by considering the derivation graph obtained from the model using the underlying operational semantics of the PEPA language. The derivation graph is systematically reduced to a form where it can be treated as the state transition diagram of the underlying stochastic (in fact, Markovian) process. From this can be obtained the infinitesimal generator matrix of the Markov process. A steady state probability distribution for the system can then be obtained, if it exists.

We have implemented a prototype tool which supports this methodology from the initial checking of the well-formedness of the PEPA model through the creation of the state transition diagrams to the calculation of performance measures based on the infinitesimal generator matrix. The tool is implemented in Standard ML [2] and provides an interface to the Maple Symbolic Algebra package [3] for the solution of matrix equations.

1 Introduction

Formal descriptions of computer systems are amenable to analysis by a range of formal techniques. At the simplest level, they may be checked for conformance with the syntax, grammar and type-correctness rules of the formal language used. More advanced analysis may involve deriving properties of a system from its description: either by deduction or by calculation. For concurrent systems modelled by an algebraic description, the properties which may be checked include freedom from deadlock and algebraic equivalence under observation with a simpler description which serves as a specification of the system. When the algebraic description is enhanced with information about the system's expected performance—as in PEPA—still further properties can be calculated. These include steady-state probabilities and rewards which may be used to derive performance measures.

The goal of the work described here is to provide a workbench for a designer of a computer system who is working from an initial PEPA model. As with other languages based on algebra and processes, e.g. CCS [4], PEPA is a parsimonious language which provides the essential, simple tools for system description. The formality and succinctness of the language have enabled the authors to design and build a workbench which assists with checking and reasoning about PEPA descriptions. Use of PEPA and the workbench is illustrated by an example taken from the area of communication networks.

2 The PEPA Language

The motivation for process algebra-based techniques for the quantitative analysis of computer systems have been presented in detail elsewhere [5, 6]. Some of the advantages of such an approach are:

- The system is represented as a collection of active agents who cooperate to achieve the behaviour of the system. This *cooperator paradigm* is particularly apt for modelling many modern computer systems.
- Compositional reasoning is an integral part of the modelling language.
- The formal definition clarifies the task of providing tools for model manipulation, simplification and analysis.
- Process algebra has growing importance as a design methodology [7, 8] and so this approach offers the possibility of integrating performance analysis into the system design process.

From a performance point of view, process algebras, such as CCS, lack essential, *quantifiable* information about time and uncertainty. Timed extensions of some process algebras have been proposed [9, 10, 11, 12] but these make a distinction between time progressing and computation progressing. PEPA, and TIPP, developed at Erlangen, take an alternative approach—time is incorporated into the algebra by associating a random variable, representing *duration*, with each *activity*[1]. We assume a *race condition* between simultaneously enabled activities. Thus, as in probabilistic process algebras, we replace the nondeterministic branching by probabilistic branching, and the timing behaviour of the system is captured. This is analogous to the association of a duration with the firing of a timed transition in a generalised stochastic Petri net [13].

It was important when designing the PEPA language to retain the key features of a process algebra which had motivated the approach: compositionality, parsimony, and the existence of a formal definition. However, it was also necessary to incorporate features to make the language suitable for capturing the performance-related information about the system. This additional information can be added as an annotation to an existing model or design.

[1] '*Activity*' is used instead of the usual process algebra '*action*' to distinguish between timed and instantaneous behaviour respectively.

2.1 PEPA Terminology

In PEPA a system is described as an interaction of *components* and these components engage, either individually or cooperatively, in *activities*. The components will correspond to identifiable substructures in the system, or rôles in the behaviour of the system. They represent the active units within a system; the activities capture the actions of those units. For example, a queue may be considered to consist of an arrival component and a service component which interact to form the behaviour of the queue.

A component may be atomic or may itself be composed of components. Thus the queue in the above example may be considered to be a component. Each component has a behaviour which is defined by the activities in which it can engage. Actions of the queue might be *accept*, when a customer enters the queue, *service*, or *loss*, when a customer is turned away because of a full buffer.

Each activity has an *action type*. We assume that each discrete action within a system has a unique type and there is a countable set, $\mathcal{A}$, of all possible such types. The action types of a PEPA term correspond to the actions of the system being modelled. There are situations when a system is carrying out some action (or sequence of actions) the identity of which is unknown or unimportant. To capture these situations there is a distinguished action type, τ, which can be regarded as the *unknown* type. Activities of this type are private to the component in which they occur.

Every activity in PEPA has an associated duration which is a random variable with an exponential distribution. Since an exponential distribution is uniquely determined by its parameter, the duration of an activity may be represented by a single real number parameter. This parameter is referred to as the *activity rate* (or simply *rate*) of the activity; it may be any positive real number, or the distinguished symbol $\top$, which should be read as "*unspecified*".

An $M/M/1/N/N$ queue in which the arrival process is suspended when the buffer is full, is represented as follows:

$$\begin{aligned}
Arrival_0 &\overset{\mathit{def}}{=} (accept, \lambda).Arrival_1 \\
\vdots\ &\ \ \vdots \\
Arrival_i &\overset{\mathit{def}}{=} (accept, \lambda).Arrival_{i+1} + (serve, \top).Arrival_{i-1} \qquad 1 \leq i \leq N-1 \\
\vdots\ &\ \ \vdots \\
Arrival_N &\overset{\mathit{def}}{=} (serve, \top).Arrival_{N-1} \\
Server &\overset{\mathit{def}}{=} (serve, \mu).Server \\
\\
Queue_0 &\overset{\mathit{def}}{=} Arrival_0 \underset{\{serve\}}{\bowtie} Server
\end{aligned}$$

Each activity, a, is defined as a pair (α, r) where $\alpha \in \mathcal{A}$ is the action type and r is the activity rate. It follows that there is a set of activities, $\mathcal{A}ct \subseteq \mathcal{A} \times \mathbb{R}^+$, where $\mathbb{R}^+$ is the set of positive real numbers together with the symbol $\top$.

When enabled, an activity $a = (\alpha, r)$, will delay for a period determined by its associated distribution, denoted $F_a(t)$ $(= 1 - e^{-rt})$. We can think of this as the activity setting a timer whenever it becomes enabled. The time allocated to the timer is determined by the rate of the activity. If several activities are enabled at the same time each will have its own associated timer. When the first timer finishes that activity takes place—the activity is said to *complete* or *succeed*. This means that the activity is considered to "happen": an external observer will witness the event of an activity of type α. An activity may be *preempted*, or *aborted*, if another one completes first.

2.2 The Syntax and Semantics of PEPA

Components and activities are the primitives of the language PEPA; the language also provides a small set of combinators. As explained in the previous section the behaviour of a component is characterised by its activities. However, this behaviour may be influenced by the environment in which the component is placed. The combinators of the language allow expressions, or terms, to be constructed defining the activities which components may undertake and the interactions between them.

The syntax for terms in PEPA is defined as follows:

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

Prefix: $(\alpha, r).P$ Prefix is the basic mechanism by which the behaviours of components are constructed. The component $(\alpha, r).P$ carries out activity (α, r), which has action type α and a duration which is exponentially distributed with parameter r (mean $1/r$). The time taken for the activity to complete will be some Δt, drawn from the distribution. The component subsequently behaves as component P. When $a = (\alpha, r)$ the component $(\alpha, r).P$ may be written as $a.P$.

It is assumed that there is always an implicit resource, some underlying resource facilitating the activities of the component which is not modelled explicitly. Thus the time elapsed before activity completion represents use of the resource by the component enabling the activity. For example, this resource might be processor time or CPU cycles within a processor, depending on the system and the level at which the modelling takes place.

Choice: $P + Q$ The component $P + Q$ represents a system which may behave either as P or as Q. $P + Q$ enables all the current activities of P and all the current activities of Q. Whichever enabled activity completes it must clearly belong to either P or Q. In this way the first activity to complete distinguishes one of the components. The other component of the choice is discarded. The continuous nature of the probability distributions ensures that the probability of P and Q both completing an activity at the same time is zero. The system will subsequently behave as P' or Q' respectively, where P' is the component which results from P completing the activity, and similarly Q'.

There is an underlying assumption that P and Q are competing for the same implicit resource. Thus the choice combinator represents competition between components.

Cooperation: $P \underset{L}{\bowtie} Q$ The cooperation combinator is in fact an indexed family of combinators, one for each possible set L of action types. The set L, the *cooperation set*, defines the action types on which the components P and Q must synchronise or *cooperate*, i.e. it determines the interaction between the components.

All activities of P and Q which have types which do not occur in L will proceed unaffected. These are termed *individual* activities. In contrast *shared* activities, activities whose type does occur in L, will only be enabled in $P \underset{L}{\bowtie} Q$ when they are enabled in both P and Q. Thus one component may become blocked, waiting for the other component to be ready to participate. These activities represent situations in the system when the components need to work together to achieve an action. In general both components will need to complete some work, corresponding to their own representation of the action. Thus a new *shared* activity is formed by the cooperation $P \underset{L}{\bowtie} Q$, replacing the individual activities of P and Q. This activity will have the same action type as the two contributing activities and a rate reflecting the rate of the slower participant.

If an activity has an unspecified rate in a component, the component is *passive* with respect to that action type. This means that although the cooperation of the component may be required to achieve an activity of that type the component does not contribute to the work involved. An example might be the rôle of a channel in a message passing system: the *cooperation* of the channel is essential if a transfer is to take place but the transfer involves no work on the part of the channel. This may be regarded as one component *coopting* another.

In contrast to choice, it is assumed that P and Q each have their own implicit resource. Activities with action types in the set L are assumed to require the simultaneous involvement of both components, both resources. The unknown action type, τ, may not appear in any cooperation set.

Hiding: P/L The component behaves as P except that any activities of types within the set L are *hidden*, meaning that their type is not witnessed upon completion. Instead they appear as the unknown type τ and can be regarded as an internal delay by the component.

Hiding does not have any effect upon the activities a component may engage in individually, but a hidden activity is witnessed only as a delay of the unknown type, τ. The duration of an activity is unaffected if it is hidden. However, a hidden activity cannot be carried out in cooperation with any other component. In effect the action type of a hidden activity is no longer externally accessible, to an observer or to another component.

Variable: X If E is a component expression which contains a variable X, then $E\{P/X\}$ denotes the component formed when every occurrence of X in E is replaced by the component P. More generally, an indexed set of variables, $\tilde{X}$, may be replaced by an indexed set of components $\tilde{P}$, as in $E\{\tilde{P}/\tilde{X}\}$.

Constant: $A \overset{\text{def}}{=} P$ We assume that there is a countable set of *constants*. Constants are components whose meaning is given by a defining equation such as $A \overset{\text{def}}{=} P$ which gives the constant A the behaviour of the component P. This is how we assign names to components (behaviours).

The semantics of the language, presented in structured operational semantics style, are shown in Figure 1. The transitional semantics over PEPA is then given by the least multi-relation $\longrightarrow \subseteq PEPA \times \mathcal{A}ct \times PEPA$ satisfying the rules.

Prefix

$$\frac{}{(\alpha, r).E \xrightarrow{(\alpha,r)} E}$$

Cooperation

$$\frac{E \xrightarrow{(\alpha,r)} E'}{E \underset{L}{\bowtie} F \xrightarrow{(\alpha,r)} E' \underset{L}{\bowtie} F} \ (\alpha \notin L) \qquad \frac{F \xrightarrow{(\alpha,r)} F'}{E \underset{L}{\bowtie} F \xrightarrow{(\alpha,r)} E \underset{L}{\bowtie} F'} \ (\alpha \notin L)$$

$$\frac{E \xrightarrow{(\alpha,r_1)} E' \quad F \xrightarrow{(\alpha,r_2)} F'}{E \underset{L}{\bowtie} F \xrightarrow{(\alpha,R)} E' \underset{L}{\bowtie} F'} \ (\alpha \in L)$$

where $R = \dfrac{r_1}{r_\alpha(E)} \dfrac{r_2}{r_\alpha(F)} \min(r_\alpha(E), r_\alpha(F))$
and $r_\alpha(E)$ is the apparent rate of α in E

Choice

$$\frac{E \xrightarrow{(\alpha,r)} E'}{E + F \xrightarrow{(\alpha,r)} E'} \qquad \frac{F \xrightarrow{(\alpha,r)} F'}{E + F \xrightarrow{(\alpha,r)} F'}$$

Hiding

$$\frac{E \xrightarrow{(\alpha,r)} E'}{E/L \xrightarrow{(\alpha,r)} E'/L} \ (\alpha \notin L) \qquad \frac{E \xrightarrow{(\alpha,r)} E'}{E/L \xrightarrow{(\tau,r)} E'/L} \ (\alpha \in L)$$

Constant

$$\frac{E \xrightarrow{(\alpha,r)} E'}{A \xrightarrow{(\alpha,r)} E'} \ (A \overset{def}{=} E)$$

Fig. 1. Operational Semantics of PEPA

When the set L is empty, $\underset{L}{\bowtie}$ has the effect of parallel composition, allowing components to proceed concurrently without any interaction between them. We use the more concise notation $P \parallel Q$ (the *parallel* combinator) to represent $P \underset{\emptyset}{\bowtie} Q$.

Execution Strategies and the Exponential Distribution The *race condition* governs the dynamic behaviour of a model whenever more than one activity is enabled. This means that we may think of all the activities attempting to proceed but only the ‘fastest’ succeeding. Of course, which activity is ‘fastest’ on successive occasions will vary due to the nature of the random variables determining the durations of activities. The probability that a particular activity completes will be given by the ratio of the activity rate of that activity to the sum of the activity rates of all the enabled activities.

We assume that the introduction of cooperation between two components implies that in general they are independent and running on separate resources. Thus we can think of their individual activities as interleaving. On the other hand, when there is a choice between components we assume that they are competing for the same underlying resource and that in fact only one of them gains the use of that resource. Thus we have two different preemption scenarios: *preemptive-resume* for cooperation and *preemptive-restart with resampling* for choice. However, we take advantage of the memoryless property of the exponential distribution which makes the two equivalent and always assume a preemptive-restart policy (with resampling). This allows us to formulate Expansion Laws of the form shown below: this would not be possible if another distribution were associated with activity durations as in some versions of TIPP [14].

Expansion Law Let $P \equiv P_1 \underset{L}{\bowtie} P_2$. Then

$$\begin{aligned} P = & \sum\{(\alpha, r).(P_1' \underset{L}{\bowtie} P_2) : P_1 \xrightarrow{(\alpha,r)} P_1';\ \alpha \notin L\} \\ & + \sum\{(\alpha, r).(P_1 \underset{L}{\bowtie} P_2') : P_2 \xrightarrow{(\alpha,r)} P_2';\ \alpha \notin L\} \\ & + \sum\{(\alpha, r).(P_1' \underset{L}{\bowtie} P_2') : P_1 \xrightarrow{(\alpha,r_1)} P_1';\ P_2 \xrightarrow{(\alpha,r_2)} P_2';\ \alpha \in L\} \end{aligned}$$

Recent work on TIPP [5] has concentrated on the subset of the language in which all activity durations are exponentially distributed. The major differences between PEPA and this subset are in the definition of the cooperation or parallel composition, and more importantly, the choice of a multi-relation, rather than a relation, to capture the operational semantics of the language.

2.3 Generating and Solving the Underlying Markov Process

For any PEPA model we can define a multigraph—the *derivation graph*—based on the operational semantics. This is a graph in which language terms form the nodes and the arcs represent the possible transitions (activities) between them; it is a multigraph since we distinguish between different instances of the same activity. This derivation graph provides a useful way to reason about the behaviour of a model. Moreover it is used to generate the stochastic process underlying any PEPA model. Each node of the derivation graph is taken to be a state in the stochastic process and the transition rate between states is the sum of the rates shown on arcs connecting the nodes in the multigraph. This is analogous to the use of the reachability graph in stochastic extensions of Petri nets such as GSPN [13]. For the $M/M/1/N/N$ queue considered earlier the derivation graph is shown in Figure 2.

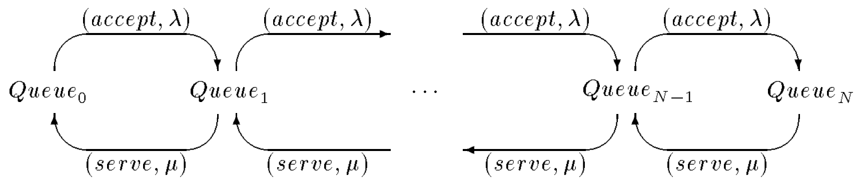

$$\begin{tikzcd} Queue_0 \arrow[r, bend left, "{(accept, \lambda)}"] & Queue_1 \arrow[l, bend left, "{(serve, \mu)}"] \arrow[r, bend left, "{(accept, \lambda)}"] & \cdots \arrow[l, bend left, "{(serve, \mu)}"] \arrow[r, bend left, "{(accept, \lambda)}"] & Queue_{N-1} \arrow[l, bend left, "{(serve, \mu)}"] \arrow[r, bend left, "{(accept, \lambda)}"] & Queue_N \arrow[l, bend left, "{(serve, \mu)}"] \end{tikzcd}$$

Fig. 2. The derivation graph for an $M/M/1/N/N$ queue

Theorem 2.1 *In a PEPA model if we define the stochastic process $X(t)$, such that $X(t) = C_i$ indicates that the system behaves as component C_i at time t, then $X(t)$ is a Markov process.*

We can construct transition rates $q(C_i, C_j)$ between components of the system as follows:

$$q(C_i, C_j) = \sum_{a \in \mathcal{A}ct(C_i|C_j)} r_a \qquad \text{where } \mathcal{A}ct(C_i \mid C_j) = \{\!| \, a \in \mathcal{A}ct(C_i) \mid C_i \xrightarrow{a} C_j \, |\!\}$$

Typically this multiset will only contain one element. The $q(C_i, C_j)$, or q_{ij}, are the off-diagonal elements of the infinitesimal generator matrix of the Markov process, $\boldsymbol{Q}$. Diagonal elements are formed as the negative sum of the non-diagonal elements of each row, i.e. $q_{ii} = -\sum_{j \neq i} q_{ij}$.

The conditions which must be satisfied in order to guarantee the existence of an equilibrium distribution for a Markov process, and for this to be the same as the limiting distribution, are well-known—*a* stationary *or* equilibrium *probability distribution,* $\mathbf{\Pi}$, *exists for every time homogeneous irreducible Markov process whose states are all positive-recurrent.*

All PEPA models are time-homogeneous since all activities are time-homogeneous: the rate and type of activities enabled by a component are independent of time. The other conditions, irreducibility and positive-recurrent states, are easily expressed in terms of the derivation graph of the PEPA model. We only consider PEPA models with a finite number of states so if the model is irreducible then all states must be positive-recurrent i.e. the derivation graph is strongly connected. In terms of the PEPA model this means that all behaviours of the system must be recurrent; in particular, for every choice, whichever path is chosen it must eventually return to the point where the choice can be made again, possibly with a different outcome.

It is interesting to note that *deadlock* and *livelock* in the process algebra model will correspond to an absorbing state, or set of states respectively, in the underlying Markov process. However, in this paper, we restrict ourselves to models without such features; for more details see [1].

Solving the Markov Process For finite state PEPA models whose derivation graph is strongly connected, (ergodic Markov process) the equilibrium distribution of the model, $\mathbf{\Pi}$, is found by solving the matrix equation

$$\mathbf{\Pi}\boldsymbol{Q} = \mathbf{0} \tag{2.1}$$

subject to the normalisation condition

$$\sum \mathbf{\Pi}(C_i) = 1 \tag{2.2}$$

The computer algebra package Maple[2] [3] is used to find $\mathbf{\Pi}$. The equations 2.1 and 2.2 are combined by replacing a column of $\boldsymbol{Q}$ by a column of 1s and placing a 1 in the corresponding row of $\mathbf{0}$. Moreover, since Maple deals with row vectors instead of column vectors, this modified $\boldsymbol{Q}$ is transposed.

[2] Maple is a registered trademark of Waterloo Maple Software.

Reward Structures and Derivation of Performance Measures Performance measures are derived by defining a *reward structure* over a model in a similar way to the use of reward structures in [15]. Reward structures have generally been explicitly treated only in the context of performability modelling, where reliability and performance aspects of a system are considered together. However, such structures may also be used to define performance measures.

As the emphasis in a PEPA model is on the behaviour of the system in terms of activities, rather than states, we associate rewards with certain activities within the system. The reward associated with a derivative (and underlying state), is then the sum of the rewards attached to activities enabled by the derivative. The performance measure is then defined as the total reward based on the steady state probability distribution, i.e. if ρ_i is the reward associated with derivative C_i, and $\mathbf{\Pi}(\cdot)$ is the steady state probability distribution of the underlying Markov process, then the total reward R is

$$R = \sum_i \rho_i \; \mathbf{\Pi}(C_i)$$

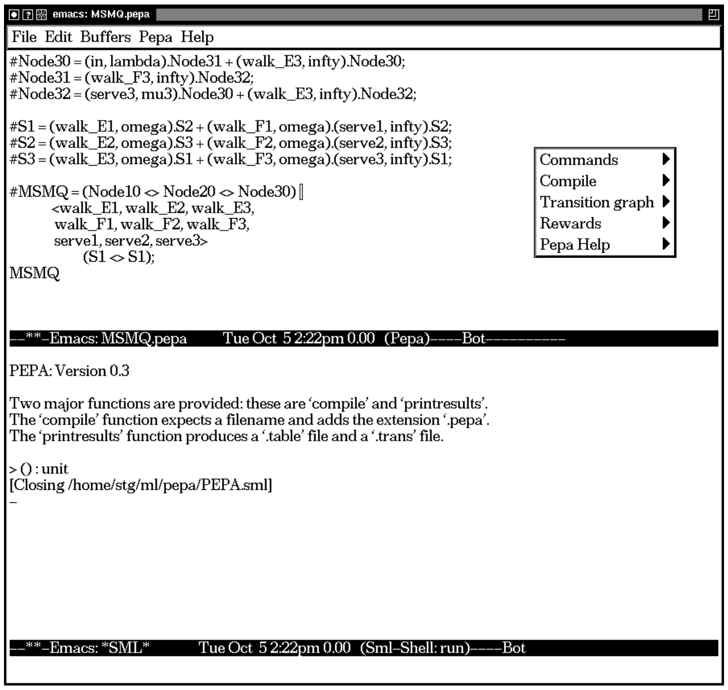
In this way, as in Stochastic Reward Networks [15], the rewards can be defined at the level of the PEPA model, rather than at the level of the underlying Markov process.

3 The PEPA Workbench

The design philosophy behind the PEPA workbench was to provide a set of simple tools to allow a skilled user of the PEPA language to delegate to machine assistance some of the routine tasks in checking PEPA descriptions and performing calculations of transition graphs and rewards. The Standard ML language was chosen as the implementation language for the workbench because it had previously been successfully used for the implementation of the Concurrency Workbench (for CCS and TCCS) [16] and choosing the same language may allow us to re-use some of the Concurrency Workbench code. Standard ML has also been used for theorem provers and other software tools locally since it provides high-level functionality via higher-order polymorphic functions. However, these functional language features are smoothly integrated with imperative assignment which allows the convenient construction of efficient programs. Standard ML is a strongly-typed, secure programming language and its use gives us confidence in the correctness of the workbench.

3.1 The Workbench Implementation

The workbench takes the form of a Standard ML image with the functionality implemented as Standard ML functions which have been pre-compiled. This provides a convenient and secure mechanism for exporting the PEPA workbench while also conveniently providing a powerful command line interface in the Standard ML language itself. A screen dump showing the workbench being accessed via the Lemacs editor is given in Figure 3. Some simple Emacs Lisp routines provide pull-down menus with submenus for issuing workbench commands. The benefits of the design of Standard ML are inherited by this process. For example, PEPA descriptions can easily be stored as



```
#Node30 = (in, lambda).Node31 + (walk_E3, infty).Node30;
#Node31 = (walk_F3, infty).Node32;
#Node32 = (serve3, mu3).Node30 + (walk_E3, infty).Node32;

#S1 = (walk_E1, omega).S2 + (walk_F1, omega).(serve1, infty).S2;
#S2 = (walk_E2, omega).S3 + (walk_F2, omega).(serve2, infty).S3;
#S3 = (walk_E3, omega).S1 + (walk_F3, omega).(serve3, infty).S1;

#MSMQ = (Node10 <> Node20 <> Node30) ||
        <walk_E1, walk_E2, walk_E3,
         walk_F1, walk_F2, walk_F3,
         serve1, serve2, serve3>
              (S1 <> S1);
MSMQ

--**-Emacs: MSMQ.pepa        Tue Oct  5 2:22pm 0.00   (Pepa)----Bot----------

PEPA: Version 0.3

Two major functions are provided: these are 'compile' and 'printresults'.
The 'compile' function expects a filename and adds the extension '.pepa'.
The 'printresults' function produces a '.table' file and a '.trans' file.

> () : unit
[Closing /home/stg/ml/pepa/PEPA.sml]
_

--**-Emacs: *SML*          Tue Oct  5 2:22pm 0.00  (Sml-Shell: run)----Bot
```

Fig. 3. The PEPA workbench

Standard ML values in the Standard ML environment and moribund values will then be taken away by the built-in garbage collector of the system, freeing the user of the workbench from the problem of managing and conserving space while generating large graphs. As a further example, it is easy to interrupt a PEPA workbench session at any time and still be able to return to it later simply by exporting the Standard ML image. No re-compilation of the PEPA description will be necessary upon returning to the session.

3.2 The PEPA Parser

PEPA is a mathematical notation and in designing a parser for the notation it was necessary to decide whether to use an extended character set for input or to decide to devise a replacement concrete syntax for the mathematical symbols. The second option was chosen. Distinct precedences were assigned to the connectives: the hiding operator was given highest precedence with prefix next, followed by co-operation. The choice operator was given lowest precedence. Parentheses were provided to allow the user to enforce the alternative parsing. The language does not have a local block construct so the processing of names is simplified. Separate name spaces are maintained for activities and components. Rates may either be entered as symbolic values or as numeric literals.

Notationally, even with the above additions, PEPA is certainly not a large language. For this reason, we decided not to use the Standard ML versions of the well-known

Lex and Yacc tools to generate a lexical analyser and a parser. This decision has a favourable consequence since using these tools would mean that they would be added to the exported image of the workbench, making it larger than really necessary. Instead, a Burge-style parser [17] has been produced for the PEPA language. This is a compact, elegant functional program which uses infix function symbols to encode the operators which combine productions in a formal description notation such as BNF. This provides a simple correspondence with the grammar for the language which makes the parser both easy to construct and easy to modify. Coding a Burge-style parser elegantly requires the language to provide polymorphic functions as first-class objects, which Standard ML does. In general, these parsers are not as efficient as Yacc-generated parsers but the efficiency of the PEPA parser is perfectly acceptable.

3.3 Computing the Transition Graph

The possible transitions of a PEPA component are obtained by following the transitions given in the operational semantic rules in Figure 1. The built-in exhaustiveness checking of the pattern-matching process deployed in this function checks that all program forms are handled by the function. Initially, the semantic rules were encoded in a naïve functional prototype implementation. This had the virtue of being obviously faithful to the language definition as given by the operational semantics but, as expected, this implementation was intolerably inefficient. Even when a sophisticated optimizing compiler was used to compile the workbench, small PEPA descriptions executed on a SparcStation 10/52 with 160Mb of memory had a running time of several hours. For some mid-sized PEPA descriptions, this prototype would exhaust the machine's memory and fail without delivering the transition graph.

After some study and analysis, a minor modification was made to produce the next version of the workbench. This used the imperative features of Standard ML to avoid some redundant re-computation which was being performed by the functional prototype. This modification was modest enough that we may be sure it did not alter the program's output from the results which would have been obtained from the prototype, thus maintaining our confidence in its correctness. However, now the workbench will calculate the transition graphs of mid-sized PEPA descriptions in a few seconds when running on a more modest SparcStation ELC with only 16Mb of memory!

This decrease in run time makes possible the interactive form of experimentation which we hoped that the workbench would provide, making it a considerably more useful tool. In addition, a decrease in memory utilization was achieved, facilitating the analysis of models greater than the largest which could have been handled by the functional prototype of the workbench.

3.4 Interfacing with Maple

The matrix manipulation routines which are required to solve the generator matrix either symbolically or numerically are provided by the Maple computer algebra package. It was judged to be simpler to use the existing Maple routines rather than re-implement these in Standard ML. Thus we have implemented the functionality to allow a workbench user to call Maple from the workbench. This enables a workbench user to pass PEPA values between Standard ML and Maple, manipulating them using whichever system is more useful for the processing task at hand.

Using the derivation graph the workbench specifies entries for the generator matrix in Maple syntax. Thus, results from the workbench can be written as Maple files and loaded into Maple. These files contain the results of the workbench analysis of the PEPA model and it is important that the PEPA user should be able to read these files in order to be able to check that the PEPA model has the behaviour which was expected. For this reason, the Maple input file is annotated with PEPA transition notation explaining the significance of the transition in the user's terms. These are written using Maple's comment notation and are therefore ignored by Maple.

4 Investigating a Simple MSMQ System

We illustrate the use of PEPA as a modelling paradigm, and the workbench, in an example taken from the study of communication systems. Polling systems have been used extensively over the last twenty years to investigate many computer and communication systems [18]. In these systems a single server circulates amongst a number of queues providing service according to a predetermined discipline. Extracting performance measures for these systems is non-trivial since the congestion at any one queue is dependent on the congestion at the other queues in the system. Recently these systems have been extended by the introduction of one or more additional servers to form *multi-server multi-queue* (MSMQ) systems [19]. MSMQ systems have been used to model applications in which multiple resources are shared among several users, possibly with differing requirements. In particular these models have been applied to local area network architectures, with ring topologies and scheduled access, in which more than one node may transmit simultaneously. For example, slotted rings and rings with multiple tokens are modelled as MSMQ systems by Yang et al. in [20].

Exact solutions for MSMQ systems have only recently been provided by Ajmone Marsan et al., [19]. In this paper we extend the class of asymmetric models considered by those authors. In [19] they consider a system of N nodes in which one node has capacity K and arrival rate $K\lambda$ while all other nodes have capacity 1 and arrival rate λ. This represents a network in which one node has high traffic and the other nodes have light traffic, such as a LAN connecting several diskless workstations and one file server. It was shown that the presence of the heavily loaded node did not greatly affect the mean waiting time of customers at lightly loaded nodes. Here we consider a system of N nodes each with capacity 1 and arrival rate λ but with customers at one node placing a larger service requirement on the server. We investigate the effect of this on the average waiting time of customers at the other nodes.

4.1 Model

We consider an MSMQ system in which there are four nodes, and two servers. Service is *limited*, meaning that each server serves at most one customer at each visit to each node. This corresponds to the *release-by-source* access mechanism for slotted rings. Moreover, only one server may service a node at any given time. Buffering is *restricted*:

a customer occupies a place in the buffer until its service is complete, and the arrival process is suspended whenever the buffer is full. We assume that the arrival process at each node is Poisson with parameter λ, and that normal service, heavy service and walk times in the system are exponentially distributed with rates μ, $m\mu$ and ω respectively.

The PEPA model of this system is shown in Figure 4. The components of the model of the system are the servers, and the nodes. Since the structure of the system is simple we model each node as a single entity.

$$
\begin{aligned}
Node_{j0} &\overset{\mathit{def}}{=} (in, \lambda).Node_{j1} + (walk\_E_j, \top).Node_{j0} && \text{for } 1 \leq j \leq N \\
Node_{j1} &\overset{\mathit{def}}{=} (walk\_F_j, r_N).Node_{j2} \\
Node_{j2} &\overset{\mathit{def}}{=} (serve_j, \mu_j).Node_{j0} + (walk\_E_j, \top).Node_{j2} \\
& && \text{where } \mu_j = \begin{cases} \mu & \text{if } j = 1 \\ m\mu & \text{if } 1 < j \leq N \end{cases} \\
S_j &\overset{\mathit{def}}{=} (walk\_F_j, \omega).(serve_j, \top).S_{j\oplus 1} + (walk\_E_j, \omega).S_{j\oplus 1} \\
& && \text{where } \; j \oplus 1 = 1 \text{ when } j = N \\
MSMQ &\overset{\mathit{def}}{=} (Node_{10} \parallel Node_{20} \parallel Node_{30} \parallel Node_{40}) \underset{\substack{\{walk\_F_j, \\ walk\_E_j, serve_j\}}}{\bowtie} (S_1 \parallel S_1) && \text{for } 1 \leq j \leq 4
\end{aligned}
$$

Fig. 4. PEPA model of an asymmetric MSMQ system with restricted buffering

S_j denotes a server ready to approach the jth node in the system. There are two possibilities: either it walks to the node and finds it empty or occupied, or it walks to the node and finds a customer requiring service and no other server currently present. These two possibilities are represented by the two activities $walk\_E_j$ and $walk\_F_j$ respectively. After the former activity the server is ready to approach the next node, but after the latter it must remain at $Node_j$ until the service is complete. The rate at which service occurs is determined by the node. All the nodes appear alike to the server but we must distinguish between them in order to maintain the cyclic scheduling. Similarly, each of the servers appear alike to the nodes. The two servers do not directly interact with each other so they may be represented as $S_j || S_j$ or $S_j || S_k$.

Each node, $Node_j$ has three distinguishable states depending on whether the buffer is empty or full, and whether a full buffer is occupied by a server. These are represented by the three derivatives of the node component, $Node_{j0}$, $Node_{j1}$ and $Node_{j2}$. An arrival may occur only when the node is empty and this is represented by an in activity with rate λ. The node will enable a walk to the node without engaging the server, $walk\_E$, when it is empty ($Node_{j0}$) or when it is already occupied by a server ($Node_{j2}$). It will enable a walk and engage the server, $walk\_F$, whenever the buffer is full but there is no server currently present ($Node_{j1}$). In each case the rate of the walk activity is determined by the server. Although the nodes are not passive with respect to the $walk\_F$ action type, we assume that the corresponding activity rate r_N is greater than ω. When the buffer of the node is full and a server is present a $serve$ activity will be enabled with a rate determined by the node. This activity must be completed before arrivals are resumed at the node.

The system has four nodes, so that when the server leaves $Node_4$ it walks on to $Node_1$. The nodes are independent, but must cooperate with a server to complete a $walk\_E$, $walk\_F$ or $serve$ activity.

4.2 Solution

The values which were assigned to the parameters are shown in Table 1. The effect of varying the service rate of customers at $Node_1$ was investigated with respect to the mean customer waiting time at the other nodes. The model has 560 states and 2064 transitions.

in	*serve*$_j$ $(j = 2, 3, 4)$	*serve*$_j$ $(j = 1)$	*walk_E*	*walk_F*
λ	μ	$m\mu$	ω	ω
0.1	1	$1 \leq 1/m \leq 5$	10	10

Table 1. Parameter values assigned to the PEPA MSMQ model

For each node we calculate the mean customer waiting time, W_j, by applying Little's Law to the node. The mean number of customers present at the node, N_j, is found by noting that there is exactly one customer present whenever the activity in is *not* enabled. Thus if we associate a reward of 1 with the activity in we can calculate the reward R_{in_j}. This has the effect of associating a reward of 1 with all states in which $Node_j$ is unoccupied. Then

$$N_j = 1 - R_{in_j}.$$

The throughput at the node, X_j, is found as the throughput of the activity $serve_j$, calculated by associating a reward of μ_j with the activity. Little's Law calculates the mean time spent in the node by a customer so the mean customer waiting time, W_j is:

$$W_j = \frac{N_j}{X_j} - \frac{1}{\mu_j} \tag{4.3}$$

The mean customer waiting time at each of the nodes, as the service demand at $Node_1$ increases, is shown in the graph shown in Figure 5. The expected waiting time for customers at $Node_1$ increases only slightly as the service demand at that node is increases. However at the other nodes the expected customer waiting time grows as the service demand at the $Node_1$ increases. It is interesting to note that this rate of growth is slightly slower at the node immediately downstream from the distinguished node ($Node_2$) as it is able to take advantage of the second server overtaking the server occupied at $Node_1$.

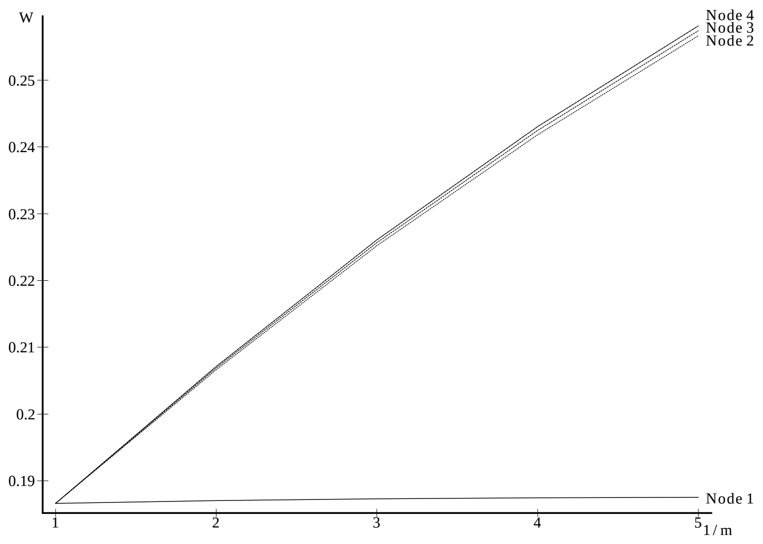


Fig. 5. A plot of mean customer waiting times

Note that using this approach asymmetric systems are handled as easily as symmetric ones. As with GSPNs the major problem of the approach is state space explosion. However, unlike GSPNs, the formal nature of the language makes it easy to detect symmetries within the system and to take advantage of these to simplify the model. A full description of these simplification techniques is beyond the scope of this paper but details can be found in [6].

5 Future Extensions

Although it would be possible to extend the PEPA language to include more combinators, we are not tempted to do this. The economy of PEPA makes reasoning about PEPA descriptions easier and made it straightforward for us to implement the workbench.

Some features could be added to the input language of the workbench to make the concrete syntax version of PEPA models shorter. These would include providing an array mechanism to allow the convenient description of families of related components. This feature is already present in the PEPA mathematical notation in the use of subscripting to denote component families.

More interesting planned extensions to the workbench include the addition of increased support for the experimentation process, allowing the workbench to take advantage of Maple's ability to solve global balance equations symbolically. In part this will rely on implementing an equivalence checker for PEPA components. This algebraic equivalence (known as *bisimulation*) will enable the user of the workbench to solve more complex models by replacing complex components with simpler ones which are algebraically equivalent. Finally, we intend to investigate the use of alternative algorithms for the balance equations to replace the Gaussian elimination with partial pivoting currently used.

References

1. J. Hillston. PEPA - Performance Enhanced Process Algebra. Technical report, Dept. of Computer Science, University of Edinburgh, March 1993.
2. R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. The MIT Press, 1990.
3. B.W. Char, K.O. Geddes, G.H. Gonnet, M.B. Monagan, and S.M.Watt. *Maple Reference Manual*. 1988.
4. R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.
5. 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.
6. J. Hillston. *A Compositional Approach to Performance Modelling*. PhD thesis, Department of Computer Science, University of Edinburgh, 1994. to appear.
7. I.S.O. LOTOS : A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. IS 8807, TC97/SC21, 1989.
8. C.J. Koomen. *The Design of Communicating Systems: A System Engineering Approach*. Kluwer, 1991.
9. F. Moller and C. Tofts. A Temporal Calculus for Communicating Systems. In J.C.M. Baeten and J.W. Klop, editors, *CONCUR'90*, volume 458 of *LNCS*, pages 401–415. August 1989.
10. X. Nicollin and J. Stifakis. An Overview and Synthesis on Timed Process Algebras. In *Real-Time: Theory in Practice*, pages 526–548. Springer LNCS 600, 1991.
11. J. Davies and S. Schneider. A Brief History of Timed CSP. Technical report, Programming Research Group, Oxford University, Oxford OX1 3QD, September 1992.
12. B. Strulo. *Process Algebra for Discrete Event Simulation*. PhD thesis, Imperial College, 1993. to appear.
13. M. Ajmone Marsan, G. Conte, and G. Balbo. A Class of Generalised Stochastic Petri Nets for the Performance Evaluation of Multiprocessor Systems. *ACM Transactions on Computer Systems*, 2(2):93–122, May 1984.
14. N. Götz, U. Herzog, and M. Rettelbach. TIPP—a language for timed processes and performance evaluation. Technical Report 4/92, IMMD7, University of Erlangen-Nürnberg, Germany, November 1992.
15. J.K. Muppala and K.S. Trivedi. Composite Performance and Availability Analysis Using a Hierarchy of Stochastic Reward Nets. In G. Balbo and G. Serazzi, editors, *Computer Performance Evaluation: Modelling Techniques and Tools*, pages 335– 349. Elsevier, February 1991.
16. R. Cleaveland, J. Parrow, and B. Steffen. The concurrency workbench: A semantics-based tool for the verification of concurrent systems. *ACM Transcations on Programming Languages and Systems*, 15(1):36–72, January 1993.
17. W.H. Burge. *Recursive Programming Techniques*. Addison-Wesley, 1975.
18. H. Takagi. Queueing Analysis of Polling Models: An Update. In H. Takagi, editor, *Stochastic Analysis of Computer and Communication Systems*, pages 267 – 318. IFIP, 1990.
19. M. Ajmone Marsan, S. Donatelli, and F. Neri. GSPN Models of Markovian Multiserver Multiqueue Systems. *Performance Evaluation*, 11:227–240, 1990.
20. Q. Yang, D. Ghosal, and L. Bhuyan. Performance Analysis of Multiple Token Ring and Multiple Slotted Ring Networks. In *Proceedings of Computer Network Symposium*, pages 79–86, Washington DC, 1986. IEEE.

Table of Contents

This article was processed using the LaTeX macro package with LLNCS style