Experiences with the PEPA Performance Modelling Tools

Graham Clark Stephen Gilmore Jane Hillston Nigel Thomas*

Abstract

The PEPA language [1] is supported by a suite of modelling tools which assist in the solution and analysis of PEPA models. The design and development of these tools have been influenced by a variety of factors, including the wishes of other users of the tools to use the language for purposes which were not anticipated by the tool designers. In consequence, the suite of PEPA tools has adapted to attempt to serve the needs of these users while continuing to support the language designers themselves. In this paper we report on our use of the PEPA tools and give some advice gained from our experiences.

1 Introduction

PEPA (Performance Evaluation Process Algebra) extends classical process algebra with the capacity to assign rates to the activities which are described in an abstract model of a system. Taken together, the information about the rates of performance of activities and the definition of the outcome of performing an activity specify a stochastic process and thus PEPA is said to be a *stochastic process algebra*. The PEPA language has been applied as a modelling language for distributed computer and telecommunications systems and for components of flexible manufacturing systems such as robotic workcells. Performance modelling with process algebras is the topic of workshops such as PAPM [2, 3].

The PEPA language provides a small set of combinators. These allow language terms to be constructed defining the behaviour of components, via the activities they undertake and the interactions between them. The syntax may be formally introduced by means of the grammar shown in Figure 1. In the grammar S denotes a *sequential component* and P denotes a *model component* which executes in parallel. C stands for a constant which denotes either a sequential or a model component, as defined by a defining equation. C when subscripted with an S stands for constants which denote sequential components. The component combinators, together with their names and interpretations, are presented informally below: further information is in the appendix.

Prefix: The basic mechanism for describing the behaviour of a system is to give a component a designated first action using the prefix combinator, denoted by a full stop. For example, the component $(\alpha, r).S$ carries out activity (α, r), which has action type α and an exponentially distributed duration with parameter r, and it subsequently behaves as S. Sequences of actions can be combined to build up a life cycle for a component.

Choice: The life cycle of a sequential component may be more complex than any behaviour which can be expressed using the prefix combinator alone. The choice combinator captures the possibility of competition or selection between different possible activities. The component $P+Q$ represents a system which may behave either as P or as Q. The activities of both P and Q are enabled. The first

*Laboratory for Foundations of Computer Science, The University of Edinburgh, James Clerk Maxwell Building, Mayfield Road, Edinburgh EH9 3JZ. Email: {gcla, stg, jeh, nat}@dcs.ed.ac.uk.

$$
\begin{array}{llll}
S & ::= & & \text{(sequential components)} \\
 & & (\alpha, r).S & \text{(prefix)} \\
 & | & S + S & \text{(choice)} \\
 & | & C_S & \text{(constant)} \\
 & & & \\
P & ::= & & \text{(model components)} \\
 & & P \bowtie_{L} P & \text{(cooperation)} \\
 & | & P/L & \text{(hiding)} \\
 & | & C & \text{(constant)}
\end{array}
$$

Figure 1: The syntax of PEPA

activity to complete distinguishes one of them: the other is discarded. The system will then behave as the derivative resulting from the evolution of the chosen component.

Constant: It is convenient to be able to assign names to patterns of behaviour associated with components. Constants provide a mechanism for doing this. They are components whose meaning is given by a defining equation.

Hiding: The possibility to abstract away some aspects of a component's behaviour is provided by the hiding operator, denoted by the division sign in P/L. Here, the set L of visible action types identifies those activities which are to be considered internal or private to the component. These activities are not visible to an external observer, nor are they accessible to other components for cooperation. Once an activity is hidden it only appears as the unknown type τ; the rate of the activity, however, remains unaffected.

Cooperation: Most systems are comprised of several components which interact. In PEPA direct interaction, or *cooperation*, between components is represented by the butterfly combinator. The set which is used as the subscript to the cooperation symbol determines those activities on which the *cooperands* are forced to synchronise. Thus the cooperation combinator is in fact an indexed family of combinators, one for each possible *cooperation set* L. When cooperation is not imposed, namely for action types not in L, the components proceed independently and concurrently with their enabled activities. However if a component enables an activity whose action type is in the cooperation set it will not be able to proceed with that activity until the other component also enables an activity of that type. The two components then proceed together to complete the *shared activity*. The rate of the shared activity may be altered to reflect the work carried out by both components to complete the activity.

In some cases, when an activity is known to be carried out in cooperation with another component, a component may be *passive* with respect to that activity. This means that the rate of the activity is left unspecified and is determined upon cooperation, by the rate of the activity in the other component. All passive actions must be synchronised in the final model.

If the cooperation set is empty, the two components proceed independently, with no shared activities. We use a compact notation—with the two cooperands separated by parallel lines—to represent this case.

PEPA serves as a high-level notation for Markov modelling because it is possible to generate directly from a PEPA model a continuous-time Markov process which faithfully encodes the behavioural and temporal aspects of the PEPA model. Through the analysis and solution of this Markov process the modeller can undertake an experimental investigation of the system which the model represents.

However, the PEPA notation is more than simply a concrete syntax for describing Markov processes. Central to the design of the language is the identification and representation of compositional structure within a model. This structure proves to be valuable both in gaining confidence that a given model correctly represents the intended system under investigation and also when seeking a solution for the corresponding Markov process.

One reason to fix on a formal notation for a task such as performance modelling is to avoid misunderstanding and misinterpretation of a model. Of course, even when a notation is carefully defined, as PEPA is, there may still be errors of misrepresentation of parts of the system within the model but all of the users of the model can at least agree on the correct interpretation of a given model through recourse to the formal definition of the language. Another reason to fix on a formal notation for performance modelling is to be able to automate some parts of the manipulation and checking of models, and that is our topic here.

We have implemented a suite of integrated tools which perform the well-formedness checking of PEPA models and allow a modeller to investigate their model through generation and solution of the corresponding Markov process or through simulation. We have made these tools freely available to others and have modified and extended them in response to their suggestions and observations. We hope that our description of this process here will be useful to others who are undertaking similar work in constructing performance modelling tools, whether based on process algebra or not.

2 Structure of this paper

In the following section we describe the PEPA modelling tools in sufficient detail to allow the reader to understand their capabilities. In Section 4 we describe in outline the aims and interests of those who have been working on the development of the PEPA language itself. These are the primary users of the PEPA modelling tools. In Section 5 we describe the community of users outside this group who have adopted the tools and applied them to their problems of interest. After introducing these two groups of PEPA users we explain in Section 6 how their contrasting interests make demands on the nature of the PEPA tools.

3 The PEPA modelling tools

There are six tools in the PEPA suite:

1. the workbench;
2. the state finder;
3. the reward assessor;
4. the analyser;
5. the discrete event simulator; and
6. the PEPA-to-Ada translator.

3.1 The PEPA Workbench

The PEPA Workbench is used to check the well-formedness of PEPA models and to generate from them their Markov process representation. It detects faults such as deadlocks and cooperations which do not involve active participants. It is described in full in an earlier paper [4].

In essence, the translation process which occurs within the PEPA Workbench accepts a PEPA model as input and produces a matrix containing the Markov process encoding of the model given. This relationship is depicted in Figure 2. In the example shown there, the PEPA model has two components which execute

$$
\boxed{\begin{array}{lcl}
P_1 & \stackrel{\mathit{def}}{=} & (\mathit{start}, r_1).P_2 \\
P_2 & \stackrel{\mathit{def}}{=} & (\mathit{run}, r_2).P_3 \\
P_3 & \stackrel{\mathit{def}}{=} & (\mathit{stop}, r_3).P_1 \\
\\
 & P_1 \parallel P_1 &
\end{array}}
\quad \Rightarrow \quad
\begin{pmatrix}
-2r_1 & r_1 & r_1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & -r_1-r_2 & 0 & r_2 & r_1 & 0 & 0 & 0 & 0 \\
0 & 0 & -r_1-r_2 & 0 & r_1 & r_2 & 0 & 0 & 0 \\
r_3 & 0 & 0 & -r_1-r_3 & 0 & 0 & 0 & r_1 & 0 \\
0 & 0 & 0 & 0 & -2r_2 & 0 & r_2 & r_2 & 0 \\
r_3 & 0 & 0 & 0 & 0 & -r_1-r_3 & r_1 & 0 & 0 \\
0 & r_3 & 0 & 0 & 0 & 0 & -r_2-r_3 & 0 & r_2 \\
0 & 0 & r_3 & 0 & 0 & 0 & 0 & -r_2-r_3 & r_2 \\
0 & 0 & 0 & r_3 & 0 & r_3 & 0 & 0 & -2r_3
\end{pmatrix}
$$

Figure 2: The PEPA Workbench schema

in parallel, both initiated in state P_1. The components are independent (they do not *cooperate*) and each has three states and so the generator matrix for the corresponding Markov process has dimension nine. Even with an example as small as this one, it is apparent that constructing a Markov process matrix by hand directly is an error-prone activity. The matrix is an unstructured representation of the system: subcomponents, states and activities are not named. Only rates of activities are entered into the matrix. The useful descriptive function provided by the PEPA notation is to allow matrices such as these to be described by more compact and expressive process algebra models.

In practice, both the mathematical syntax used within a PEPA description and the mathematical syntax used to express a matrix must be replaced by rather more prosaic concrete syntax in order that the input can be conveniently composed with a simple text editor and in order that the resulting matrix representation can be conveniently processed by other tools. In addition, the PEPA Workbench must also produce a *state table* which explains which states of the PEPA model have been assigned which numerical indices for the matrix.

The investigation into properties of interest to the performance modeller working with PEPA proceeds by determining the steady-state probability distribution for the system. The purpose of the analysis performed by the Workbench has been to reject PEPA models which are not well-formed. A model which was not well-formed would not correspond to an irreducible Markov process over a finite state space. These are necessary conditions to guarantee that all of the states of the Markov process are positive-recurrent, and thus reaches an equilibrium state.

The steady-state distribution may be found by applying any one of a number of linear algebra solution methods to the generator matrix, although as we note in Section 6, some of these methods are of limited use for PEPA models of even modest size. For this reason we have recently extended the PEPA Workbench with the capability to reduce models to a canonical form internally, thereby automatically aggregating the model [5]. The effect of aggregation on a model is shown in Figure 3, generated by the PEPA Workbench itself. Where a new expression for a rate is formed in the aggregated model it appears as a black dot in the picture.

3.2 The PEPA State Finder

The PEPA State Finder allows a PEPA modeller to investigate the state space which has been generated by the PEPA Workbench. The state finder provides a simple pattern language which the modeller can use to describe states of interest. Continuing the example from the previous section, a modeller interested in testing how often either component in the tiny example was the first derivative of P would prepare the input

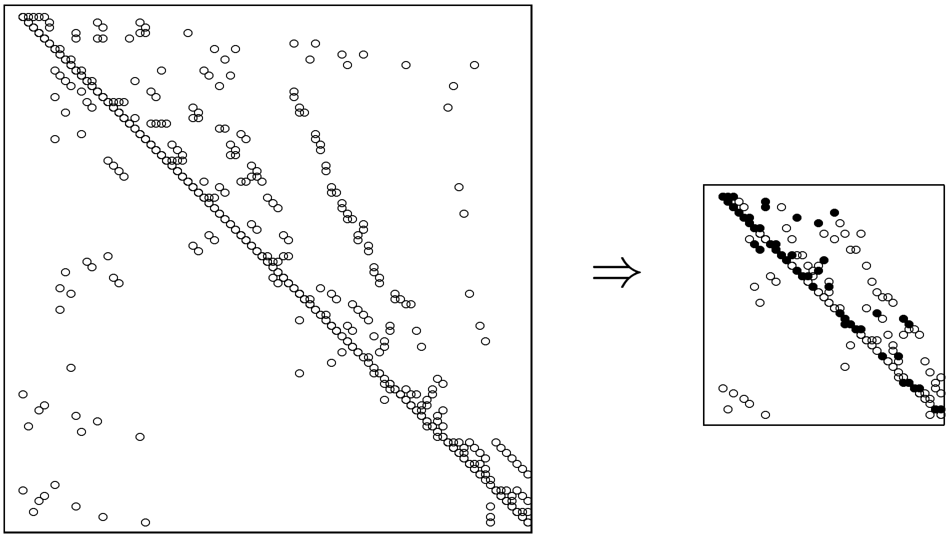

Figure 3: Aggregation of models by the PEPA Workbench

shown in the left-hand box in Figure 4. The PEPA State Finder would generate the function which is shown in the right-hand box.

% We are checking
% for component P_1
% in either case
test: $P_1 \mid *$
test: $* \mid P_1$

$\Rightarrow$

test := **proc** (π) (		
	$\pi[1]$	# *from state* $P_1 \parallel P_1$
	$+\,\pi[3]$	# *from state* $P_1 \parallel P_2$
	$+\,\pi[6]$	# *from state* $P_1 \parallel P_3$
	$+\,\pi[2]$	# *from state* $P_2 \parallel P_1$
	$+\,\pi[4]$	# *from state* $P_3 \parallel P_1$
)		
end:		

Figure 4: A function generated by the PEPA State Finder

Notice that here the function correctly avoids counting twice the state with two copies of the first derivative of P. This would be a mistake which could easily be made if building up the function from a number of smaller functions, defined separately. Typically, the PEPA State Finder is used when investigating the results obtained after solving the Markov process produced by the Workbench although this is not the only possible use. Another use is to pose questions about the reachability of certain states within the model. The analysis which can be performed in this way is a weak form of liveness analysis, allowing the modeller to determine in some cases that certain parts of the model are redundant. This is typically caused because the conditions are never met to allow some cooperations between model components to take place.

A significant disadvantage of the pattern language is that it is too syntactic, distinguishing between states which are (logically) equivalent. The PEPA Reward Assessor provides a more expressive logical notation for characterising states.

3.3 The PEPA Reward Assessor

Classical process algebras are complemented by modal logics, which formally express properties of a model in terms of its behaviour. Verifying that a system possesses a particular logical property is called *model checking*. A technique is described in [6] which utilises a simple formalism, based on Hennessy-Milner logic. It was developed with the explicit intention of supporting behavioural reasoning about the quantitative behaviour of PEPA models. At the level of the underlying Markov process, rewards are still used to calculate performance measures; however whether or not a reward is assigned to a component of the model is decided in terms of the *behaviour* of the component, in keeping with the process algebra approach.

Formulae are built from Boolean connectives and modal operators thus:

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

where K ranges over action types. This logic has a simple semantics; for example a PEPA model satisfies the formula $\langle a \rangle\texttt{tt}$ if it may perform an activity of type a such that its derivative satisfies $\texttt{tt}$ (which every model does, trivially). Therefore this formula specifies that a process may perform an a activity. Similarly, a process satisfies $\Phi_1 \wedge \Phi_2$ if it satisfies both Φ_1 and Φ_2.

The PEPA Reward Assessor is an experimental extension to the PEPA Workbench which allows rewards to be associated with model states which satisfy a given logical formula. It implements a significant subset of the ideas described in [6]. This technique provides an expressive way in which to discriminate between uninteresting states, and states which are relevant to a particular performance measure. Using the PEPA Reward Assessor, it is possible to isolate states which regular expressions alone could not capture. Furthermore, use of this technique confers the added advantage that reward specifications need not be modified to correctly compute performance measures for models which have been reduced, for example by using the aggregation method described in [5]. However this extra expressibility requires care and the effective use of this tool may require a little more experience than that needed to use the PEPA State Finder.

Figure 5 shows a throughput analysis of a simple model using the PEPA Workbench. *Dev* attempts to transmit a packet consisting of a connection header and some data. However it may fail to connect, in which case it immediately retries. We would only wish to consider the throughput of legitimate data, and not failed connection attempts; therefore the logical expression selects only those states in which *Dev* has made a connection, and is about to transmit data. These states are assigned the data transmission rate as a reward. The six-state model has a reward assigned to states 4 and 6 only.

3.4 The PEPA Analyser

One appealing aspect of the use of a formal notation for the high-level description of Markov processes is the potential to automate the recognition of certain classes of processes which have desirable solution properties. The definition of the method of recognising these classes solely from their description in the high-level notation is termed *finding a syntactic characterisation*. The strength of these characterisations is that they do not require the generation and examination of the Markov process encoding of the model and thus they are applicable to models whose state space is too large to allow them to be examined by numerical solution. The PEPA Analyser detects properties such as *quasi-separability* in models [7]. Models which are quasi-separable can be simplified significantly through a decomposition which can still allow the exact computation of many performance measures of interest. An extension is planned to address *quasi-reversible* and *reversible* models also.

% We only measure throughput of successful transmissions
throughput: $\langle trans \rangle \mathtt{tt} \wedge [trans][trans]\mathtt{ff} = rate(trans)$

$$
\begin{aligned}
Dev &\stackrel{def}{=} (check, r_1).Try \\
Try &\stackrel{def}{=} (trans, fail).Try + (trans, succeed).Con \\
Con &\stackrel{def}{=} (trans, data).Dev \\
Bus &\stackrel{def}{=} (check, \top).(reset, r_2).Bus
\end{aligned}
$$

$$Dev \underset{\{check\}}{\bowtie} Bus$$

$\Downarrow$

R[1]	:=	R[1]+0.0:	%	*Dev*	|	*Bus*
R[2]	:=	R[2]+0.0:	%	*Try*	|	$(reset, r_2).Bus$
R[3]	:=	R[3]+0.0:	%	*Dev*	|	$(reset, r_2).Bus$
R[4]	:=	R[4]+data:	%	*Con*	|	$(reset, r_2).Bus$
R[5]	:=	R[5]+0.0:	%	*Try*	|	*Bus*
R[6]	:=	R[6]+data:	%	*Con*	|	*Bus*

Figure 5: The PEPA Reward Assessor schema

3.5 The PEPAroni discrete-event simulator

A theme of recent stochastic process algebra research has been the introduction of *generally distributed* activities. The hope is that such an addition will lead to a modelling language which retains the structural benefits of a process algebra, while allowing greater expressibility when modelling systems. The addition of such general probability distributions renders false the convenient Markovian assumption, and obtaining solutions for such models is difficult in general. There are two approaches to this problem. The first is to resort to stochastic simulation, deriving performance measures from particular simulation runs of the model. If done with care, this can yield a useful degree of confidence in the model's behaviour. The second approach is to make use of the theory of stochastic insensitivity, the application of which guarantees that in some circumstances, exponential distributions may be used to replace general distributions, without affecting the model's steady state solution. Conventional solution methods for Markovian models may then be used.

The development of PEPAroni was motivated by both of the approaches described above. However, rather than focusing on developing the PEPA modelling framework to support general distributions, the intention was to investigate situations in which general distributions may safely replace exponential distributions in PEPA models.

Rather than developing a simulation application from scratch, it made sense to investigate simulation frameworks upon which a process algebra interface could be built. One such framework is *simjava* [8]. Implemented in Java, it provides a framework for discrete event simulation, and, if the developer feels particularly creative, supports the deployment of animations as applets.

Arguably, the fact that all simulations would be based upon Java might have disadvantages, but in practice, this has not turned out to be the case. Speed is an issue when long simulation runs are desirable for statistical accuracy. In the past, Java applications have tended to run considerably more slowly than native compiled code. However this problem is alleviated by the recent appearance of just-in-time compilers for Java. Secondly, our language choice marked a departure from the traditional use of the Standard ML programming language, and its attendant aids to the programmer. Fortunately this has not been a serious issue either, since the PEPA interface is being developed in Pizza [9], a superset of Java. (The name for our application was simply too good an opportunity to miss.) Pizza extends Java with several features, of which the most useful was *class cases*, allowing objects to be built and processed in a very similar fashion to those created with datatypes in Standard ML. Process algebra models are built compositionally, and the operational semantics of PEPA define process behaviour compositionally. This elegance gave us a degree of confidence in the correctness of our implementation. The Pizza compiler translates programs directly to Java bytecode, and so no portability is lost.

PEPAroni is currently under active development. An alternative to providing concrete simulation models is to translate a PEPA process directly into the syntax of another programming language, thus allowing it to be used as a template for software development. This is the approach taken by the PEPA-to-Ada translator, described next.

3.6 The PEPA-to-Ada translator

One area where stochastic process algebras can be most fruitfully applied is in the investigation of the potential performance of a yet-to-be-constructed software system. One example of such a software system which would benefit from an initial performance-based exploration of its design and construction might be the control system for a hybrid manufacturing process. In these application areas embedded systems languages such as Ada [10] have proven themselves to be reliable and useful. We considered the problem of developing an Ada software system from a PEPA performance model previously [12, 13] and we have implemented a translator from PEPA to Ada which faithfully represents both the behavioural and performance aspects of the PEPA model in the generated Ada program. The Ada program which is produced can then be used directly as a simulator for the PEPA model or it can be used as the basis of further software development work which will eventually form the system implementation.

The idea that a performance model of a software system could be expressed in the implementation language of the system itself might give the idea that a special-purpose modelling language is not necessary. To show that it is still a *convenient* part of the process consider the PEPA model shown in Figure 6.

$$
\begin{array}{lcl}
\mathit{Buffer} & \stackrel{\mathit{def}}{=} & (\mathit{Put}, r).(\mathit{Get}, r).\mathit{Buffer} \\
\mathit{Producer} & \stackrel{\mathit{def}}{=} & (\mathit{Put}, \top).\mathit{Producer} \\
\mathit{Consumer} & \stackrel{\mathit{def}}{=} & (\mathit{Get}, \top).\mathit{Consumer}
\end{array}
$$

$$
\mathit{Producer} \underset{\{\mathit{Put}\}}{\bowtie} \left((\mathit{Buffer} \underset{\{\mathit{Get}\}}{\bowtie} \mathit{Consumer}) \parallel (\mathit{Buffer} \underset{\{\mathit{Get}\}}{\bowtie} \mathit{Consumer}) \right)
$$

Figure 6: An example input for the PEPA-to-Ada translator

This model describes a producer which puts goods into one of two buffers. These goods are extracted from each of the buffers by the consumer which is associated with that buffer. Thus the producer and the buffers cooperate on the *Put* action and the consumers cooperate with their buffer on the *Get* action. The *implicit choice* of buffer made by the producer and the coupling of buffers and consumers is achieved through the specification of co-operation sets for these model components. In contrast these must be represented explicitly in the programming language, making it a more cumbersome and less expressive notation for analysis and reasoning: the generated Ada code for this example is 80 lines long.

4 The language developers

The PEPA modelling tools have been developed in order to deepen our understanding of the language and to allow us to thoroughly check examples of modest size which we wish to use as illuminating examples in papers and articles. Sometimes these examples are sufficiently simple that it would be conceivable to examine them by hand but our motivation for using the tools is the reduced potential for clerical errors which they afford.

Of course, the classification of examples as being either little illustrations or realistic case studies is not a precise one. To a theoretician a little illustration might have only tens of states and a realistic case study a hundred. For a practising performance engineer, such models are more likely to have thousands of states and millions of states respectively. The existence of a suite of modelling tools such as ours helps us to move from the theoretician's world closer to the world of the practising performance engineer.

The existence of the tools has also helped us to ensure the accuracy of translations and operations on PEPA models. Syntactic manipulations are delicate constructions and without the existence of a tool to apply the manipulation it is all too easy to convince oneself of the validity of a seriously flawed construction. Perhaps not unexpectedly, this was particularly the case with the translation into the Ada programming language where we found that several of the subtle complexities of the Ada programming language had escaped us before we implemented and tested our translation tool.

5 The user community

The user perspective on our tools is quite different from ours. It has been our experience that the community of users who have adopted our tools have come from a background where they have been accustomed to modelling systems with formal notations such as CCS or CSP [14, 15]. In these settings only the behaviour of the system is modelled and thus there is no notion of *model solution* to discover a measure such as throughput or component utilisation. Given this background, the model solution part of the performance modelling process came as an unexpected complication.

In another way the user perspective on the PEPA tools differs from ours. We have undertaken case studies [16] because we have felt that it would be an unwise practice to develop a theoretical framework for efficient investigation of distributed systems without simultaneously assessing its practical utility. However, when others have used our modelling tools we have found that it is frequently because they have a particular application which they wish to investigate and they have adopted the PEPA formalism for the description and analysis of the application of interest.

In this setting it is then typical for the users of our modelling tools to observe that they would wish to have an extended modelling language which included, for example, operators such as the LOTOS *disabling* operator or sequential composition at the level of model components [17] or parametric definition of components [18]. These facilities would make the modelling language more comfortable to use but until complementary solution techniques have been devised they would in fact only serve to allow the description of models with larger state spaces, thereby giving additional problems for the model solution utilities.

Nonetheless, users' opinions of our modelling tools have been favourable. In [18] the authors describe their experience with the PEPA Workbench as "positive, in the sense that we did not encounter any flaws and the results obtained were consistent with—while at the same time more accurate than—those obtained by the traffic analysis methods". We have often hoped that stochastic process algebras would complement existing longer-established methods of performance modelling so it comes as a welcome discovery to find that they can also *improve* on the results obtained by existing methods!

6 Solution methods and tools

The benefits to be gained from using a stochastic process algebra to model a computer system come from the insights into the dynamic behaviour of the system which can be obtained by investigation of the alteration of the steady-state probability distribution of the system under changes in the rates of performance of activities. From a controlled series of experiments such as these the modeller can determine a variety of performance measures such as the identification of bottlenecks, the throughput of items in the system, component utilisation or mean waiting times.

Before we gained experience in the use of our performance modelling tools we had pessimistically assumed that limits on the effectiveness of our tools would be set by their own inability to generate Markov process representations of PEPA models. In contrast to our expectations, we have discovered that the limits on the applicability of our tools come from the inability of other packages to solve the Markov process representations which we generate. We have also discovered that different solution tools suit different groups of PEPA users. We discuss these findings here.

6.1 Maple

For small examples such as the ones which we include in expository papers and articles we have found that the Maple computer algebra package provides an excellent working environment for analysis of model solutions. Maple has a profound advantage over many of its competitors: it can provide *symbolic* solutions in terms of the activity rates from the model. However, solutions which can be computed are typically symbolic in only one of the rates. The advantage of offering symbolic solutions is further qualified by the fact that it limits the size of models whose solutions may be computed. Maple uses Gaussian elimination with partial pivoting, which we found to be suitable for models of only a few thousand states.

6.2 Matlab, Mathematica and others

Larger models, such as those investigated by some of our users, outstrip the capabilities of the Maple computer algebra system but do fall within the scope of numerical computing platforms such as Matlab and Mathematica. Both [17] and [18] used Matlab to solve their models and analyse the results. Here the size of the models which could be solved increased ten-fold from those which could be solved by Maple.

6.3 Bespoke solvers

Mathematical modelling environments such as Maple, Matlab and others provide a comfortable working environment for less experienced modellers because they provide convenient and effective representations of the matrices and vectors which are needed to store the Markov process encoding of the user's model and the steady-state probability distribution. In addition, they provide flexible Algol-like programming languages for the definition of functions over these matrices and vectors, as illustrated in Figure 4. However, if one is willing to forgo some of these comforts it is possible to analyse larger models or to process them more efficiently.

One of the most effective solution methods which we have used is the preconditioned biconjugate gradient method. We obtained a finely tuned implementation of this from the Numerical Recipes collection [19].

For a model of nearly 20,000 states we reduced the solution time by a factor of fifteen when compared with the Matlab solution time for the steady-state distribution of the same model.

However, in view of the complications which we have already noted that users can encounter we have decided to distribute the PEPA modelling tools only in those forms which work with high-level computing environments such as Maple, Matlab and Mathematica. In this way, we hope to reduce the potential for error when working with unstructured numerical values as one does when running a C routine outside such an environment.

7 Implementation notes

With only a few exceptions, the PEPA modelling tools are implemented in the functional programming language Standard ML [20]. This language has served us well in allowing us to produce reliable tools quickly and to have them be flexible enough to permit revision and modification to adapt to the different needs of different user groups. Standard ML also supports the dissemination and delivery of our tools because it is available for a wide number of platforms in a variety of implementations, some of which have very modest system requirements. We have found that the language's strong type system is a significant aid in eliminating a number of errors which would have manifested themselves at run-time if working in an untyped language.

Standard ML has provided a clean implementation platform for us, which we have rarely needed to leave. Two exceptions to this have been the C implementation of the biconjugate gradient method solver and the use of a UNIX tool (`grep`) called from the PEPA State Finder. This latter excursion from Standard ML proved to be a foolish one since we discovered that it gave rise to problems for users whose version of `grep` differed from ours. We re-wrote the PEPA State Finder to eliminate the need to use regular expressions to generate the model functions, and thereby repaired the portability problem which we had introduced. Furthermore, on the final occasion we did not develop in Standard ML, the ML-like features provided by the Pizza programming language were familiar, and provided some confidence in the correctness of our implementation.

7.1 Availability of the modelling tools

The PEPA modelling tools, together with user documentation and papers and example PEPA models are available from the PEPA Web page at the address `http://www.dcs.ed.ac.uk/pepa`.

8 Further work

Development of the existing PEPA modelling tools is ongoing and other directions will also be explored, including *debugging*. The performance measures which are calculated by the solution of the generated Markov process, or by simulation, are only as good as the model from which they were derived. If there is a logical flaw in the model of the system then the results could be entirely worthless, irrespective of whether they were derived by solution or simulation. Thus the suite of PEPA modelling tools seeks to give the modeller a number of methods of uncovering errors in their models. These include state-space construction, reachability analysis, satisfaction of a logical description, examination of simulation runs and will also include interactive debugging. The PEPA debugger is not yet complete but will eventually allow modellers to set breakpoints on both states and activities, run their models until a breakpoint is encountered and then fire selected transitions and continue. The degree of coverage of the state space of the model will be estimated.

9 Conclusions

We believe that others who are developing tools for a performance modelling notation which extends a behavioural modelling notation—as our stochastic process algebras extend classical process algebras—might find that the users who adopt their tools have similar backgrounds and expectations to ours. We were surprised by the diversity and range of the tools which were required to support a small modelling language. Many of these have been created and others are planned. Our aim has been to provide modellers who are working with the PEPA modelling language with high-level tools to support all aspects of the performance modelling process and give them a variety of ways of investigating, exploring and analysing the models which they create.

Acknowledgements

Robert Holton wrote the Ada statistics gathering packages which is used by the Ada code which is generated by the PEPA-to-Ada translator. Graham Clark is supported by EPSRC studentship 95306547. Stephen Gilmore is supported by the 'Distributed Commit Protocols' grant from the EPSRC and by Esprit Working group FIREworks. Jane Hillston and Nigel Thomas are supported by the ESPRC 'COMPA' grant.

References

[1] J. Hillston. *A Compositional Approach to Performance Modelling*. Cambridge University Press, 1996.

[2] S. Gilmore and J. Hillston, editors. *Proceedings of the Third International Workshop on Process Algebras and Performance Modelling*. Special Issue of *The Computer Journal*, 38(7), December 1995.

[3] M. Ribaudo, editor. *Proceedings of the Fourth Annual Workshop on Process Algebra and Performance Modelling*. Dipartimento di Informatica, Universitá di Torino, CLUT, July 1996.

[4] S. Gilmore and J. Hillston. The PEPA Workbench: A Tool to Support a Process Algebra-based Approach to Performance Modelling. In *Proceedings of the Seventh International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, number 794 in Lecture Notes in Computer Science, pages 353–368, Vienna, May 1994. Springer-Verlag.

[5] S. Gilmore, J. Hillston, and M. Ribaudo. An efficient algorithm for aggregating PEPA models. Submitted for publication, 1997.

[6] G. Clark. Formalising the specification of rewards with PEPA. In Ribaudo [3], pages 139–160.

[7] N. Thomas and S. Gilmore. Applying quasi-separability to Markovian process algebra. In C. Priami, editor, *Proceedings of the Sixth International Workshop on Process Algebra for Performance Modelling*, Nice, France, September 1998.

[8] F. Howell and R. McNab. simjava: a discrete event simulation package for Java with applications in computer systems modelling. *First International Conference on Web-based Modelling and Simulation, San Diego CA*, 1998.

[9] M. Odersky and P. Wadler. Pizza into Java: Translating theory into practice. *Proc. 24th ACM Symposium on Principles of Programming Languages*, January 1997.

[10] J. Barnes. *Programming in Ada 95*. Addison-Wesley, 1996.

[11] R. Pooley and J. Hillston, editors. *Proceedings of the Twelfth UK Performance Engineering Workshop*, Department of Computer Science, The University of Edinburgh, September 1996.

[12] S. Gilmore, J. Hillston, and D.R.W. Holton. From SPA models to programs. In Ribaudo [3], pages 179–198.

[13] S. Gilmore and J. Hillston. Refining internal choice in PEPA models. In Pooley and Hillston [11], pages 49–64.

[14] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[15] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[16] S. Gilmore, J. Hillston, D.R.W. Holton, and M. Rettelbach. Specifications in Stochastic Process Algebra for a Robot Control Problem. *International Journal of Production Research*, 34(4):1065–1080, 1996.

[17] D.R.W. Holton. A PEPA specification of an industrial production cell. In Gilmore and Hillston [2], pages 542–551.

[18] A. El-Rayes, M. Kwiatkowska, and S. Minton. Analysing performance of lift systems in PEPA. In Pooley and Hillston [11], pages 83–100.

[19] W.H. Press, S.A. Teukolsky, W.T. Vetterling, and B.F. Flannery. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, second edition, 1992.

[20] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML: Revised 1997*. The MIT Press, 1997.

A Operational Semantics and the Underlying CTMC

Model components capture the structure of the system in terms of its *static* components. The dynamic behaviour of the system is represented by the evolution of these components, either individually or in cooperation. The form of this evolution is governed by a set of formal rules which give an operational semantics of PEPA terms. The semantic rules, in the structured operational style of Plotkin, are presented in Figure 7 without further comment; the interested reader is referred to [1] for more details. The rules are read as follows: if the transition(s) above the inference line can be inferred, then we can infer the transition below the line.

Thus, as in classical process algebra, the semantics of each term in PEPA is given via a labelled *multi-transition* system—the multiplicities of arcs are significant. In the transition system a state corresponds to each syntactic term of the language, or *derivative*, and an arc represents the activity which causes one derivative to evolve into another. The complete set of reachable states is termed the *derivative set* of a model and these form the nodes of the *derivation graph* formed by applying the semantic rules exhaustively.

The timing aspects of components' behaviour are not represented in the states of the derivation graph, but on each arc as the parameter of the negative exponential distribution governing the duration of the corresponding activity. The interpretation is as follows: when enabled an activity $a = (\alpha, r)$ will delay for a period sampled from the negative exponential distribution with parameter r. If several activities are enabled concurrently, either in competition or independently, we assume that a *race condition* exists between them. Thus the activity whose delay before completion is the least will be the one to succeed. The evolution of the model will determine whether the other activities have been *aborted* or simply *interrupted* by the state

Prefix

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

Cooperation

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

$$\frac{E \xrightarrow{(\alpha,r_1)} E' \;\; F \xrightarrow{(\alpha,r_2)} F'}{E \underset{L}{\bowtie} F \xrightarrow{(\alpha,R)} E' \underset{L}{\bowtie} F'} \; (\alpha \in L) \qquad \text{where } R = \frac{r_1}{r_\alpha(E)} \frac{r_2}{r_\alpha(F)} \min(r_\alpha(E), r_\alpha(F))$$

$r_\alpha(E)$ 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

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

Figure 7: The operational semantics of PEPA

change. In either case the memoryless property of the negative exponential distribution eliminates the need to record the previous execution time.

When two components carry out an activity in cooperation the rate of the shared activity will reflect the working capacity of the slower component. We assume that each component has a capacity for performing an activity type α, which cannot be enhanced by working in cooperation (it still must carry out its own work), unless the component is passive with respect to that activity type. For a component P and an action type α, this capacity is termed the *apparent rate* of α in P. It is the sum of the rates of the α type activities enabled in P. The apparent rate of α in a cooperation between P and Q over α will be the minimum of the apparent rate of α in P and the apparent rate of α in Q.

The derivation graph is the basis of the underlying Continuous Time Markov Chain (CTMC) which is used to derive performance measures from a PEPA model. The graph is systematically reduced to a form where it can be treated as the state transition diagram of the underlying CTMC. Each derivative is then a state in the CTMC. The *transition rate* between two derivatives P and Q in the derivation graph is the rate at which the system changes from behaving as component P to behaving as Q. It is denoted by $q(P,Q)$ and is the sum of the activity rates labelling arcs connecting node P to node Q. In order for the CTMC to be *ergodic* its derivation graph must be strongly connected. Some necessary conditions for ergodicity, at the syntactic level of a PEPA model, have been defined [1]. These syntactic conditions are imposed by the grammar introduced earlier.