Performance modelling in PEPA with higher-order functions

Stephen Gilmore Jane Hillston*

March 23, 2001

Abstract

Performance Evaluation Process Algebra (PEPA) is a high-level modelling language for distributed systems. In this paper we describe a novel approach to composing PEPA models where the language is supplemented with the structuring features and language constructs of a strongly typed higher-order functional programming language, Standard ML. Together, this language combination provides greatly improved support than was previously available for experimenting with PEPA models to check for errors in their formulation.

1 Introduction

Modelling the performance of yet-to-be-constructed software systems is becoming an increasingly common activity among practising software developers. Recent developments in practical modelling languages for software systems have included introducing language extensions which explicitly support performance modelling of software systems [1]. Our motivation here is to demonstrate that more abstract languages for performance modelling can be made more familiar to software engineers by supplementing them with programming language constructs which are familiar from functional programming languages. As our example we show how the stochastic process algebra PEPA [2] can be extended with the constructs from the Standard ML programming language [3]. The PEPA language is supported by a range of modelling tools which allow models to be solved for steady-state analysis. One of the primary modelling tools for PEPA is the PEPA Workbench [4] but other tools have also been applied to a range of example models [5].

Process algebras are parsimonious languages. PEPA offers only indispensable connectors for defining the behaviour of structured performance models. This makes the definition of the language readily amenable to formal analysis which guarantees the validity or reasoning methods such as proof by construction of bisimulation equivalences between PEPA components. The language definition also ensures that these equivalences are meaningful ones in terms of the model's effective performance as expressed through random variables drawn from exponential distributions.

* Laboratory for Foundations of Computer Science, The University of Edinburgh, Edinburgh, EH9 3JZ, Scotland. Email: `stg@dcs.ed.ac.uk`, `jeh@dcs.ed.ac.uk`

For models of modest size and detail, the use of a compact modelling language is highly desirable. Mistakes of mis-interpretation are less likely because of the simplicity of the language and the goal of effective and precise communication is served well. As models grow in size and detail together with our ability to solve them it becomes more important that the modelling language used be able to formally express scope and structure within models, in addition to providing other means of expressing flow of control and logical dependencies within models. One advantage of this added facility for defining models is that the re-use of existing tested models is better supported, because of better language support for encapsulation and component definition.

2 Embedding PEPA in Standard ML

The program structuring facilities of Standard ML are typical of those of a higher-order functional programming language. Functions are first-class and can be passed as parameters, returned as results, or used as values which can be stored in a data structure such as a list or a binary tree. The ability to manipulate functions in this way is used in our embedding of PEPA components in Standard ML. PEPA components describe the performance of activities by using conditional recursive definitions. We map PEPA components onto Standard ML functions and provide datatype constructors which can be used to compose these functions as structured PEPA components can be built using the combinators of the language.

PEPA has four combinators, prefix (.), choice (+), co-operation ($\bowtie$) and hiding (/). These are mapped on to the constructors of a Standard ML datatype which can be used to compose component definitions. The constructors of this datatype are `*`, `+`, `<>` and `/`. These identifiers are chosen to resemble the corresponding symbols in the PEPA syntax, making their encoding relatively straightforward. The appendix gives a full introduction to the PEPA language.

2.1 A simple example

As a simple example consider the following PEPA model which defines the very simple model of a transmitter which transmits at rate t or rate $t/2$ to a receiver which receives data at rate r. This transmission is conducted through the medium of a network which passively cooperates with the transmitter and the receiver. This is the meaning of the $\top$ symbol used in the occurrences of the transmit and receive activities specified in the description of the network, that it is passive with respect to these activities. Transmissions are visible outside the system but receipt of transmissions is hidden. This small example uses all four of the PEPA combinators.

$$
\begin{aligned}
\mathit{Transmitter} &\stackrel{\mathit{def}}{=} (\mathit{trans}, t).\mathit{Transmitter} + (\mathit{trans}, t/2).\mathit{Transmitter} \\
\mathit{Receiver} &\stackrel{\mathit{def}}{=} (\mathit{recv}, r).\mathit{Receiver} \\
\\
\mathit{Network} &\stackrel{\mathit{def}}{=} (\mathit{trans}, \top).\mathit{Network}' \\
\mathit{Network}' &\stackrel{\mathit{def}}{=} (\mathit{recv}, \top).\mathit{Network} \\
\\
\mathit{System} &\stackrel{\mathit{def}}{=} (\mathit{Transmitter} \underset{\{\mathit{trans}\}}{\bowtie} \mathit{Network} \underset{\{\mathit{recv}\}}{\bowtie} \mathit{Receiver})/\{\ \mathit{recv}\ \}
\end{aligned}
$$

The model is encoded using our PEPA embedding in Standard ML thus.

```
  datatype Activity  =  trans | recv
                          | tau of { hidden : Activity };
      datatype Rate  =  t | t_half | r | top;

 fun Transmitter ()  =  (trans, t) * Transmitter
                     +  (trans, t_half) * Transmitter;
    fun Receiver ()  =  (recv, r) * Receiver;

     fun Network ()  =  (trans, top) * Network'
    and Network' ()  =  (recv, top) * Network;

      fun System ()  =  (Transmitter() <[trans]> Network()
                                <[recv]> Receiver()) / [recv];
```

The datatype definitions at the beginning encode the names of activities and rates which are used in the model. The benefit which comes from this are the guarantees provided by the strict static type-checking of the Standard ML language. For example, i) no activities or rates can be used within the model without an accompanying declaration and ii) an activity name cannot appear where a rate is expected. The polymorphic type inference [6] mechanism of Standard ML means that this benefit is provided without the imposition of additional syntactic clutter such as typing assignments in function definitions.

The PEPA language defines a distinguished activity name, τ, which indicates that the activity is a private one. Components cannot co-operate on τ activities. The `tau` constructor of the activity datatype provides the value `tau { hidden = recv }` which additionally captures in the Standard ML encoding the information that the activity which was hidden was `recv`.

The PEPA language also defines a distinguished symbol $\top$ which is used to indicate passive co-operation in an activity. This is encoded as the constructor `top` in the datatype of activity rates.

Each of the PEPA component definitions translates into a function definition, introduced by the keyword *`fun`*. Where a group of component definitions are mutually recursive—as *Network* and *Network*$'$ are—they are introduced by a simultaneous binding where the function definitions are separated by the keyword *`and`*. All functions in Standard ML have a single argument and so the definition of the function name is followed by Standard ML's *void* type, denoted by `()`. Each of the sets used in the PEPA model—for synchronisation or hiding—contains only a single activity name but if more are required they can be included in lists such as `[trans, recv]`.

2.2 Using Standard ML's structuring facilities

In the previous example it might be the case that we view the definition of the component *Network*$'$ as being subordinate to the *Network* component and would like to hide the identifier *Network*$'$ so that only the identifier *Network* is visible. PEPA's definitional equality $\stackrel{def}{=}$ provides us with a method of associating a name which a behaviour but does not provide a method for hiding the names which are bound in this way. With our embedding in Standard ML we can make a component definition local to an expression by using the *let .. in .. end* construct as shown below.

```
fun Network () =
  let
    fun Network' () = (recv, top) * Network
  in
    (trans, top) * Network'
  end
```

Management of the identifier namespace is an important issue for larger development projects and as our ability to solve models increases, so to does the usefulness of namespace management constructs such as *let .. in .. end*.

2.3 Encoding parameterised components

Some model components are inherently parametric, with behaviour which counts a certain type of events. One such typical component is a bounded queue, which can be encoded as a Standard ML function with different cases to deal with boundary conditions. In the example below, we fix the maximum queue length to be six. The function `succ` returns the successor of an integer value and the function `pred` returns the predecessor of an integer value.

```
fun Queue 0 ()  =  (accept, lambda) * Queue 1     (* empty *)
  | Queue 6 ()  =  (serve, top) * Queue 5          (* full *)
  | Queue n ()  =  (accept, lambda) * Queue (succ n)
                +  (serve, top) * Queue (pred n)
```

Because the Standard ML language supports unbounded data structures we can equally easily encode unbounded queues, as can be expressed in extensions of PEPA such as PEPA$^{\infty}_{\mathrm{ph}}$ [7].

```
fun Queue 0 ()  =  (accept, lambda) * Queue 1     (* empty *)
  | Queue n ()  =  (accept, lambda) * Queue (succ n)
                +  (serve, top) * Queue (pred n)
```

In either case it is unlikely that we would wish to initialise the queue to contain any data initially. In this case we can partially apply the definition to get a definition of a queue where the parameterised definition is hidden and errors such as an inadvertent call to `Queue 7` are prevented. (Such a mistake in initialising the queue would allow the queue to grow and shrink without restriction until the queue length fell to six, at which point the definition would cause the queue to regain the behaviour of a queue with a maximum capacity of six.) Standard ML's *local .. in .. end* construct is used to make a group of definitions local to other definitions, as shown below.

```
local
  fun queue 0 ()  =  (accept, lambda) * queue 1
    | queue 6 ()  =  (serve, top) * queue 5
    | queue n ()  =  (accept, lambda) * queue (succ n)
                  +  (serve, top) * queue (pred n)
in
  val Queue = queue 0
end
```

An alternative method of achieving the same effect would be to use Standard ML's *exception* mechanism to reject any queue lengths which were out of range.

2.4 Parameterising on activities and rates

By further parameterising the model on the names of activities and their rates and the capacity of the queue we produce a very general template for queue which can be instantiated to deliver the queue which we want.

```
fun Queue (a, ar) (s, sr) capacity =
  let
    fun queue n () =
      if (n = 0) then (a, ar) * queue 1
      else if (n = capacity) then (s, sr) * queue (pred n)
           else (a, ar) * queue (succ n) +
                    (s, sr) * queue (pred n)
  in
    queue 0
  end
```

One possible use of such a queue template would be to produce a single cell with `put` and `get` activities.

```
val Cell = Queue (put, top) (get, top) 1
```

3 Combinators

Parameterisation on activities and rates gives us the ability to make general-purpose definitions of component behaviours which will then have a number of different realisations. Combinators which allow us to modify the behaviour of existing components are also very useful. The activities performed by a component are found by using the `step` function, which gives the activity name, the rate and the resulting component. Any of these may then be modified before the next one-step derivative is found. Two examples are presented below.

3.1 Renaming

One way in which renaming can be useful is when we have not had the foresight to parameterise a component definition on an activity name. In this case we compute a one-step derivative, test for the activity which has been renamed and replace it where needed. Renaming is a static activity so the same renaming will need to be applied to the one-step derivative of the component.

```
fun (P renaming (alpha, beta)) () =
  let
    val ((act, rate), P') = step P   (* unroll P one step *)
    val P'' = P'() renaming (alpha, beta)
  in
    if act = alpha then (beta, rate) * P'' else (act, rate) * P''
  end
```

3.2 Coroutines

We define a *Cycle* combinator which is a version of Milner's *Before* combinator [8] for the classical process algebra CCS modified to combine two PEPA components. Unlike CCS, PEPA has no notion of component termination and so the sequential composition of components which is defined for CCS in the *Before* combinator has no meaning in PEPA. However, two non-terminating processes can be defined so that each of them will yield to allow the other to progress. In this way, their combination will cycle between the behaviour of the first component and the behaviour of the second component.

```
fun Cycle (P, Q) () =
  let
    val ((act, rate), P') = step P   (* unroll P one step *)
  in
    case act of
      yield => (yield, rate) * Cycle (Q, P'())
    | default => (act, rate) * Cycle (P'(), Q)
  end
```

4 Verifying higher-order models

The primary reason to be concerned about the expressiveness of the modelling language which is used for performance modelling is that a well-designed language can be helpful in early detection of errors in models. Because the performance modelling process is a time-consuming one, in addition to being resource-intensive, significant effort can be saved if errors can be found in models before they are solved. We have implemented an interpreter for the PEPA modelling language, allowing initial lightweight experimentation with models. If this experimentation proves to be successful then the next stage in the modelling process is model checking properties of the system over the transition graph. The companion language which we use for specifying properties of PEPA models is the probabilistic modal logic PML_μ [9, 10].

5 Conclusions

The embedding of PEPA in Standard ML can be used simply to provide a simple experimental implementation of a PEPA model in a strongly-typed language. This implementation can be executed with our interpreter. This process alone may reveal errors in a model but it can be taken further because logical properties can be checked against the model. This allows a graded migration from limited testing at one extreme to exhaustive symbolic probabilistic model checking at the other. The Standard ML embedding of the PEPA language also provides a useful testbed for the design of simple combinators for PEPA components and can serve as a designer's workbench for experimenting with simple extensions to the language. New combinators are encoded as component-manipulating functions and can be used to modify the behaviour of components by replacing some transitions with others or by combining two components to generate a third. We have found these combinators to be useful in practice.

Acknowledgements

Stephen Gilmore is supported by the 'Distributed Commit Protocols: Interactive Simulation, Verification and Behavioural Analysis' grant from the EPSRC and by Esprit Working group FIREworks. Jane Hillston is supported by the EPSRC 'Computer System Modelling via Markovian Process Algebra' (COMPA) grant.

References

[1] R. Pooley and P. King. The Unified Modeling Language and performance engineering. *IEE Proceedings—Software*, 146(1):2–10, February 1999.

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

[3] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML.* The MIT Press, 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] G. Clark, S. Gilmore, J. Hillston, and N. Thomas. Experiences with the PEPA performance modelling tools. *IEE Proceedings—Software*, 146(1):11–19, February 1999. Special issue of papers from the Fourteenth UK Performance Engineering Workshop.

[6] R. Milner. A theory of type polymorphism in programming languages. *Journal of Computer and System Science*, 17(3):348–375, 1978.

[7] M. Kwiatkowska, A. El-Rayes, and G. Norman. Solving infinite stochastic process algebra models through matrix-geometric methods. In J. Hillston, editor, *Proceedings of the Seventh Annual Workshop on Process Algebra and Performance Modelling*, Zaragosa, Spain, September 1999. University of Zaragosa Press.

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

[9] G. Clark, S. Gilmore, and J. Hillston. Specifying performance measures for PEPA. In J.-P. Katoen, editor, *Proceedings of the Fifth International AMAST Workshop on Real-Time and Probabilistic Systems*, number 1601 in LNCS, pages 211–227, Bamberg, Germany, May 1999. Springer-Verlag.

[10] G. Clark, S. Gilmore, J. Hillston, and M. Ribaudo. Exploiting modal logic to express performance measures. In B.R. Haverkort, H.C. Bohnenkamp, and C.U. Smith, editors, *Computer Performance Evaluation: Modelling Techniques and Tools, Proceedings of the 11th International Conference*, number 1786 in LNCS, pages 211–227, Schaumburg, Illinois, USA, March 2000. Springer-Verlag.

A Summary of the PEPA language

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.

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

Figure 1: The syntax of PEPA

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.

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 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 activity to complete distinguishes one of them: the other is discarded. The system will 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 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 (we write $P \parallel Q$ as an abbreviation for $P \underset{L}{\bowtie} Q$ when L is empty). 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.

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, are presented in Figure 3 without further comment; the interested reader is referred to [2] 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.

A.1 Apparent rates

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. The notation $r_\alpha(P)$ is used to denote the apparent rate of α in P. It

$$
\begin{aligned}
r_\alpha((\beta, r).P) &= \begin{cases} r, & \text{if } \alpha = \beta \\ 0, & \text{if } \alpha \neq \beta \end{cases} \\
r_\alpha(P + Q) &= r_\alpha(P) + r_\alpha(Q) \\
r_\alpha(P/L) &= \begin{cases} r_\alpha(P), & \text{if } \alpha \notin L \\ 0, & \text{if } \alpha \in L \end{cases} \\
r_\alpha(P \underset{L}{\bowtie} Q) &= \begin{cases} r_\alpha(P) + r_\alpha(Q), & \text{if } \alpha \notin L \\ \min(r_\alpha(P), r_\alpha(Q)), & \text{if } \alpha \in L \end{cases}
\end{aligned}
$$

Figure 2: The apparent rate of α in PEPA components

is the sum of the rates of the α type activities enabled in P. For a cooperation, if α is in the cooperation set, the slowest participant determines the rate of the cooperation.

A.2 The derivation graph

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

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

Prefix

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

Cooperation

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

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

$$\text{where } R = \frac{r_1}{r_\alpha(E)} \frac{r_2}{r_\alpha(F)} \min(r_\alpha(E), r_\alpha(F))$$

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

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

Figure 3: The operational semantics of PEPA

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 [2]. These syntactic conditions are imposed by the grammar introduced earlier.

A.3 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`.