The PEPA Feature Construct

Stephen Gilmore and Jane Hillston

LFCS, The University of Edinburgh, Edinburgh EH9 3JZ, Scotland.
Email: {stg, jeh}@dcs.ed.ac.uk

Abstract. We show how the PEPA performance modelling language could be extended with a *feature construct* which can be used to describe modifications to PEPA models. We provide this construct with an operational description which conservatively extends the operational semantics of the PEPA language. We then show how the feature construct can be applied in a small case study.

1 Introduction

PEPA (Performance Evaluation Process Algebra) [1] is a performance modelling notation. It is also a process algebra, a concise mathematical language which is amenable to formal reasoning. PEPA is defined by an unambiguous semantics which makes clear the meaning of all models which are expressed in the language. It has been used to investigate the behaviour and performance of a diversity of distributed and concurrent systems [2–6].

Constructing performance models of distributed systems is a worthwhile activity. Distributed systems often have designs which are both complex and novel. An ill-considered design decision can lead to an implementation which fails to achieve planned levels of service or has unnecessarily high running costs.

As is the case for other performance modelling notations, PEPA can be applied either prospectively to assess the viability of a candidate design for a yet-to-be-constructed system or retrospectively to provide insight into a fully functional operation. Our novel contribution in this paper is to show how the PEPA language could be extended with syntactic support for the description of enhancements to models which reflect enhancements to the system under study. We use the term *features* to describe these enhancements to both systems and models, in keeping with the use of this term in the telecommunications industry and in software development. With this extension, the PEPA language can more easily be used throughout the entire lifetime of a complex distributed system, tracking adaptive and corrective maintenance in a formal setting.

Improving the suitability of the PEPA language for modelling complex systems is a useful extension. The difficulty in computer system development stems from the desire to create sophisticated and comprehensive products. These are constructed piece by piece and are subject to many unpredictable revisions over time. This is our motivation for considering a formal means

of expressing the addition of new features to an exisiting system. The fact that a system has interesting performance qualities which are worth investigating via the construction and analysis of system models does not make it impervious to modifications and the addition of new features.

One way to address the problem of modelling the addition of features is to modify the extant system description in such a way that the new feature is incorporated or interwoven into the model description. This might at first seem to be an attractive option since it does not require any modification to the existing modelling language which was used to describe the original system. The modeller can simply think that they are only constructing a model of a more complex system, namely the one which includes the added feature. However, this approach has the disadvantage that it reduces the intellectual leverage which the identification of features gives both to system designers and system builders. Firstly, the valuable documentation function which features provide has not been exploited with the consequence that no formal record of the change history of the system is being created. Secondly, the dependency of the added feature upon components of the existing system will be unclear. Effectively, *all* of the existing system description has been considered to be essential to the description of the new feature. It is not usual that this is the case in practice.

We adopt the principle that feature descriptions require a different form of expression from system descriptions. Particularly, their descriptions should make clear the dependencies of components of the existing system. This places a demand upon our modelling language to provide some distinctive syntactic support for the formal expression of features. Other authors have also argued that the addition of a feature construct to an existing modelling language is the right method by which to make progress in this problem [7,8].

2 Design of the feature construct

PEPA is a small language with essential, simple combinators. A description appears in Appendix A. Readers who are already familiar with the PEPA language can omit this summary, which is standard. Readers unfamiliar with PEPA who are keen to understand the technical details of this paper should study the language summary in Appendix A before proceeding. Briefly, PEPA *components* perform *timed activities*. Each PEPA model defines a labelled multi-transition system which can be read as a Continuous Time Markov Chain (CTMC) by ignoring the activity names which label the arcs from one state to another.

A feature construct for PEPA must add value without incurring unnecessary loss of simplicity. (The feature construct can itself be seen as a *feature* which is being added to the existing PEPA language. Many of the good practices which are applied when adding functional features to software systems

have their analogues here where we are adding a model structuring feature to an existing modelling language.)

Following [7], we consider that a feature construct should describe features formally as self-contained units of functionality. It should be possible to consider features in isolation, without complete knowledge of the system to which they are being added. A feature construct should be general-purpose, allowing a number of different types of features to be added. However, it should not allow undisciplined modifications which would be hard to reason about or understand. This last requirement would rule out of consideration as candidates a number of powerful, but low-level, macro-like operators.

In the particular setting of the PEPA modelling language we want a feature construct which is applicable, general, clear and easy to explain. It must have a formal definition. Two candidates present themselves as being possibly suitable; *re-binding* and *parametric definition*.

2.1 Re-binding component definitions

PEPA is a *compositional* description language so in principle new features could be installed by re-binding the definitions of key components. Such an extension to the PEPA language would meet most of our criteria for a feature construct. Re-binding definitions is a general-purpose concept and it can certainly be described both formally and with clarity. Unfortunately, it fails to meet our key criteria of applicability because it cannot describe the most general case of making unforeseen extensions to an existing system. The use of re-binding as a feature construct is only applicable in the cases where the designers of the system have previously loaded the system with re-programmable hooks. It is usual to describe such systems as *feature-ready* because they have been designed in the anticipation of the addition of new features in particular ways. Extensible software systems such as Web browsers with a "plug-in" capability are an example of feature-ready systems.

2.2 Parametric components

Our winning candidate for a feature construct for PEPA is the use of parameterised components. The parameters capture the dependency of the feature on the existing system. By defining our new feature in terms of the existing system, monitored for essential behaviour, we allow for system reconfiguration and the introduction of new components.

Definition: A *feature for a PEPA model* consists of:

- one or more parameterised components which describe the behaviour of the newly added feature possibly re-using existing components;
- optionally, some non-parameterised components which are used to structure the new feature (components may be re-used here also); and
- a new *system equation* which describes how the new system is built from the existing one and the components of the two kinds described above.

3 Semantics of the feature construct

A feature will utilize a parameterized component and it will also typically make use of simple components. Incorporating a feature into a system is done by instantiating the parameter of the feature by the existing system, optionally combined with new simple components.

In operation, a feature monitors its base system. Whenever the base system makes a transition it is checked against the triggers for the feature. If it is not a trigger the base system proceeds as before. Otherwise, the transition is replaced as indicated and the feature determines how to proceed.

3.1 Impose and treat

A useful separation of concerns in describing feature integration is the distinction between *imposing* new behaviour on the system and *treating* the existing behaviour in a new way. In a state-based modelling approach such as ours this divides into redirecting the system evolution into new states in the former case and substituting new activities for existing ones in the latter case. In either case, some activity is used as a trigger for the new feature. If the activity happens then the new feature comes into effect. If not then the system behaves as before. Assume that the parameter P has an (α, r) transition to P', then there are four possible outcomes.

Treat (1):	$S(P) \xrightarrow{(\alpha,r)} S(P')$	α is not a trigger
Treat (2):	$S(P) \xrightarrow{(\alpha,r)} S'(P', Q)$	α is a trigger for Q
Impose (1):	$S(P) \xrightarrow{(\alpha,r)} S(P')$	α is not a trigger
Impose (2):	$S(P) \xrightarrow{(\beta,s)} S'(P', Q)$	α is a trigger for Q

Formally, in both cases there are two possibilities. Either the transition which the existing system would perform has a matching transition in the transitions of the parameterised component or it does not. In the cases where there is a matching transition the new state of the system is dictated by a parameterised component whose behaviour can depend on either P' or Q. In the cases where only one of the possible outcomes is of importance the parameterised component need only receive a single component as its parameter. In this case it might be that the derivative S' of S is itself. The Treat rule can introduce new behaviour. Additionally, the Impose rule is to replace an activity α performed at rate r by an activity β performed at rate s.

It is easy to see that the expressiveness of the Treat rule subsumes that of the Impose rule. When the metavariables α and β denote the same PEPA activity and the metavariables r and s denote the same PEPA rate then the Treat rule expresses the same adaptation of the existing system behaviour

Impose

$$\frac{P \xrightarrow{(\alpha,r)} P' \quad S(P) \overset{(\alpha,r)}{\not\Longrightarrow}}{S(P) \xrightarrow{(\alpha,r)} S(P')} \qquad \frac{P \xrightarrow{(\alpha,r)} P' \quad S(P) \overset{(\alpha,r)}{\Longrightarrow} S'(P',Q)}{S(P) \xrightarrow{(\alpha,r)} S'(P',Q)}$$

Treat

$$\frac{P \xrightarrow{(\alpha,r)} P' \quad S(P) \overset{(\alpha,\_)}{\not\Longrightarrow}}{S(P) \xrightarrow{(\alpha,r)} S(P')} \qquad \frac{P \xrightarrow{(\alpha,r)} P' \quad S(P) \overset{(\alpha,\_)}{\Longrightarrow} (\beta,s).S'(P',Q)}{S(P) \xrightarrow{(\beta,s)} S'(P',Q)}$$

Choose

$$\frac{}{S(P,Q) \longrightarrow S'(P)} \qquad \frac{}{S(P,Q) \longrightarrow S'(Q)}$$

Fig. 1. Semantics of the feature construct

as the Impose rule. It is also easy to see that the Impose rule by itself is not sufficient because it is not possible to prevent the exisiting system model from performing its *first* activity. This is a limitation because we wish to be able to impose new behaviour on *any* state of the exisiting system, even the initial state. The Treat rule does not suffer from this limitation.

Since the Treat rule subsumes the Impose rule, and is applicable in more situations, why then do we keep the Impose rule at all? The reason is that in practice it gives the most convenient form of expression to new features. The more general rule would almost always be used to simulate Impose.

It could be that we wish to make only a slight amendment to the existing system and it should not be the case that the feature construct *forces* reconfiguration. A simple example would be renaming activities as in "whenever the base system does activity α, do β instead and then continue as the base system". In telephony the Call Forwarding feature does this. This is achieved by furnishing the residual of a parameterised component with the two possible outcomes of the system as its parameters. Branching the system to track the evolution of its two possible futures on every feature application is of course impractical so we require the parameterised residual at the next step to select one of the derivatives with the Choose rules. Where appropriate we abbreviate pairing and choosing to follow only the selected component.

We make the preceeding informal description fully formal in the operational definition of the feature construct presented in Figure 1. This is built structurally from the transition relation on PEPA components (written $\longrightarrow$) and the transitions of parameterised components (written $\Longrightarrow$, we will use the notation $\not\Longrightarrow$ to indicate the absence of such a transition). The definition of the transition relation for PEPA components is in Figure 9 in Appendix A. The definition of the transition relation for parameterised components is in Figure 2. It depends on the definition given in Figure 3 of the judgement relation for contexts (written $\vdash$).

Overriding

$$\frac{}{C \vdash (\alpha, r).S(E) \stackrel{(\alpha,r)}{\Longrightarrow} S(E)}$$

$$\frac{}{C \vdash (\alpha, \_).(\beta, s).S(E) \stackrel{(\alpha,\_)}{\Longrightarrow} (\beta, s).S(E)}$$

Option

$$\frac{C \vdash E_1 \stackrel{(\alpha,r)}{\Longrightarrow} S(E)}{C \vdash E_1 + E_2 \stackrel{(\alpha,r)}{\Longrightarrow} S(E)} \qquad \frac{C \vdash E_2 \stackrel{(\alpha,r)}{\Longrightarrow} S(E)}{C \vdash E_1 + E_2 \stackrel{(\alpha,r)}{\Longrightarrow} S(E)}$$

$$\frac{C \vdash E_1 \stackrel{(\alpha,\_)}{\Longrightarrow} (\beta, s).S(E)}{C \vdash E_1 + E_2 \stackrel{(\alpha,\_)}{\Longrightarrow} (\beta, s).S(E)} \qquad \frac{C \vdash E_2 \stackrel{(\alpha,\_)}{\Longrightarrow} (\beta, s).S(E)}{C \vdash E_1 + E_2 \stackrel{(\alpha,\_)}{\Longrightarrow} (\beta, s).S(E)}$$

Application

$$\frac{C \vdash E' \stackrel{(\alpha,r)}{\Longrightarrow} S'(F)}{S(P) \stackrel{(\alpha,r)}{\Longrightarrow} S'(Q)} \quad (S(E) \stackrel{\text{def}}{=} E' \text{ and } C \vdash E = P, F = Q)$$

$$\frac{C \vdash E' \stackrel{(\alpha,\_)}{\Longrightarrow} (\beta, s).S'(F)}{S(P) \stackrel{(\alpha,\_)}{\Longrightarrow} (\beta, s).S'(Q)} \quad (S(E) \stackrel{\text{def}}{=} E' \text{ and } C \vdash E = P, F = Q)$$

Fig. 2. Transition relation for parameterised definitions

3.2 Parameterised definitions

Parameterised components are used to express the intervention of the new feature on the existing system, as defined in Figure 2. We reuse the prefix notation to express the activity of overriding the state and activities of the existing system and the subsequent evolution to a new parameterised system. In this reuse the two types of features are distinguished syntactically by the number of activities which prefix their parameterised residual. If there is just one activity then this is an *impose* feature and the activity performed will be mirrored by the replacement activity. If there are two activities then the first is the trigger for the feature (which will be absorbed) and the second is the replacement activity. The rate of the triggering activity is not significant and is denoted by an underscore. We use the plus notation in parameterised components to express the option of monitoring a range of activities, summing over all of the possibilities. By considering the syntactic form of such a sum it is simple to derive statically the set of significant activities which can trigger the activation of a newly defined feature. Just as significant is the use of this set of activity names to determine which activities *cannot* cause the invocation of a feature.

In the definition of the transition relation in Figure 2 the metavariable E used in parametric definitions is sometimes used to stand for a single component and sometimes for a pair of components, allowing a single rule to encompass both cases. A similar device could be used to allow a single trigger for a feature to be replaced by a sequence of activities.

3.3 Contexts

We introduce definition contexts to explain the binding of actual component parameters to formal parameter identifiers. We allow components to be parameterised by other components but the parameterisation is first-order, that is we do not allow parameterised components to be passed as parameters.

More formally, parameters range over PEPA expressions extended with formal parameter identifiers in addition to the identifiers used for PEPA constants. To preclude syntactic ambiguity, we use the convention that if a constant appears in a formal parameter expression then it denotes the component bound to that constant name and it is not a re-use of that identifier with another meaning in the body of the parameterised component definition. Thus it is not possible to make a hole in the scope of a component definition by re-using the identifier of that component as the identifier of a formal parameter.

Given the above syntactic restriction on formal parameter identifiers we note that contexts cannot contain re-definitions of identifiers of constants. The notation $C, I = P$ therefore denotes a context C extended by the binding of an identifier I to a component P. Such an expression can be used to judge that the identifiers I and P both denote the same component. The notation $E = P, L$ is used to denote a list of equalities beginning with one between E and P and continuing with those in L.

Note that, as is usual with definitional equality, the equality symbol used in contexts is not a commutative operator. In constrast the relations which are used to judge semantic equivalence between PEPA components such as PEPA's strong equivalence (also known as Markovian bisimulation) both preserve the familiar logical properties of equivalence relations and are congruences over all of the PEPA operators. This allows the component-wise substitution of equals for equals which is the foundation of a compositional modelling approach such as that embodied by PEPA.

$$\frac{}{C \vdash P = P} \qquad \frac{}{C, I = P \vdash I = P} \qquad \frac{C \vdash E = P \qquad C \vdash L}{C \vdash E = P, L}$$

$$\frac{C \vdash E_1 = P_1 \qquad C \vdash E_2 = P_2}{C \vdash E_1 \underset{L}{\bowtie} E_2 = P_1 \underset{L}{\bowtie} P_2} \qquad \frac{C \vdash E = P}{C \vdash E/L = P/L} \qquad \frac{C \vdash E = P}{C \vdash E = A}(A \stackrel{\text{def}}{=} P)$$

Fig. 3. The judgement relation for definition contexts

4 Example

We present here an example which serves only to help explain the use of the feature construct, not to act as a compelling defense of the use of formal notations in performance modelling. We consider first the very simple model of a transmitter which transmits at rate t 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.

The feature which we add to this simple system is a new component which monitors the network bandwidth which the transmitter has been able to obtain and signals whenever this drops below a critical threshold. In this circumstance the transmitter halves its transmission rate (for example, in a multimedia application by sampling an analogue input signal at half of the previous rate or in another application by applying data compression). The model is presented in Figure 4.

Existing system

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

Addition of monitoring feature

$$
\begin{aligned}
\mathit{Transmitter}' &\stackrel{\text{def}}{=} (\mathit{trans}, t/2).\mathit{Transmitter}' \\
\mathit{Monitor} &\stackrel{\text{def}}{=} (\mathit{low}, l).\mathit{Monitor}' \\
\mathit{Monitor}' &\stackrel{\text{def}}{=} (\mathit{high}, h).\mathit{Monitor} \\
\\
S(M \parallel (T \underset{\{\mathit{trans}\}}{\bowtie} N \underset{\{\mathit{recv}\}}{\bowtie} R)) &\stackrel{\text{def}}{=} \\
& \quad (\mathit{low}, l).S(\mathit{Monitor}' \parallel (\mathit{Transmitter}' \underset{\{\mathit{trans}\}}{\bowtie} N \underset{\{\mathit{recv}\}}{\bowtie} R)) \\
& \quad + \\
& \quad (\mathit{high}, h).S(\mathit{Monitor} \parallel (\mathit{Transmitter} \underset{\{\mathit{trans}\}}{\bowtie} N \underset{\{\mathit{recv}\}}{\bowtie} R)) \\
\\
\mathit{System}' &\stackrel{\text{def}}{=} S(\mathit{Monitor} \parallel \mathit{System})
\end{aligned}
$$

Fig. 4. Use of the feature construct

It is possible that the congestion on the network will subsequently reduce. In this case the monitor will signal that it is suitable for the transmitter to resume transmitting at the higher rate. The details of how the monitor measures the consumption of the available bandwidth are abstracted away in the model and the signals to switch between transmitting at the low rate and the high rate are simply modelled by stochastic events with parameters l and h respectively. From a performance modelling point of view, the effect of adding the new feature is to turn the *Transmitter* component from a simple Poisson arrival process into a Markov modulated process. The new feature ensures that the correct *Transmitter* component is used whenever the *Monitor* component changes state. Upon witnessing a *low* signal from the monitor, the values held by the formal parameters M and T will be lost and the primed versions of the *Monitor* and *Transmitter* components will be used. Upon witnessing a *high* signal from the monitor, the values held by the formal parameters M and T will similarly be lost and the unprimed versions of the *Monitor* and *Transmitter* components will be reinstated.

The fact that there was no planning for the subsequent addition of a monitoring feature in the original model is reflected in the fact that the activities which are of interest to the monitoring feature are the *low* and *high* activities which are themselves newly added to the system via the *Monitor* component. We can see statically from the definition of the monitoring feature alone that the activities *trans* and *recv* are not used anywhere in the definition. This provides a simple and efficient check of freedom of interference between features and components.

5 Implementation

We have provided an embedding of the extended PEPA language in the higher-order functional programming language Standard ML [9] a language which we have previously used in the implementation of other software tools for the PEPA language, notably the PEPA Workbench [10].

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 crucially used in our embedding of PEPA components in Standard ML. In a language which did not support functions as first-class values the encoding of PEPA components would involve much more indirection. 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. Featured PEPA components are modelled as higher-order functions.

PEPA has four combinators, prefix (.), choice (+), co-operation ($\bowtie$, or $\|$ if the co-operation set is empty) 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 $*$, $+$, $\langle$, $\rangle$, $\|$ and /. These identifiers are chosen to resemble the corresponding symbols in the PEPA syntax, making their encoding relatively straightforward.

$$
\begin{array}{rcl}
\textbf{datatype}\ \mathit{Component} = * & \textbf{of} & (\mathit{Activity} * \mathit{Rate}) * (\mathit{unit} \rightarrow \mathit{Component}) \\
\mid + & \textbf{of} & \mathit{Component} * \mathit{Component} \\
\mid \| & \textbf{of} & \mathit{Component} * \mathit{Component} \\
\mid \langle & \textbf{of} & \mathit{Component} * \mathit{Activity\ list} \\
\mid \rangle & \textbf{of} & \mathit{Component} * \mathit{Component} \\
\mid / & \textbf{of} & \mathit{Component} * \mathit{Activity\ list}
\end{array}
$$

Fig. 5. Datatype definition for the PEPA combinators

To explain the use of these constructors, when the plus constructor is give a pair of components ($\mathit{Component} * \mathit{Component}$), it labels these as a choice, which is itself a kind of component.

The simple model from the previous section is encoded using our PEPA embedding in Standard ML in Figure 6. The datatype definitions at the

$$
\begin{array}{l}
\textbf{datatype}\ \mathit{Activity} = \mathit{trans} \mid \mathit{recv} \mid \mathit{tau}\ \textbf{of}\ \{\mathit{hidden} :\ \mathit{Activity}\}; \\
\quad \textbf{datatype}\ \mathit{Rate} = t \mid r \mid \mathit{top}; \\
\\
\textbf{fun}\ \mathit{Transmitter}\,() = (\mathit{trans}, t) * \mathit{Transmitter}; \\
\quad \textbf{fun}\ \mathit{Receiver}\,() = (\mathit{recv}, r) * \mathit{Receiver}; \\
\\
\quad \textbf{fun}\ \mathit{Network}\,() = (\mathit{trans}, \mathit{top}) * \mathit{Network}' \\
\textbf{and}\ \mathit{Network}'\,() = (\mathit{recv}, \mathit{top}) * \mathit{Network}; \\
\\
\quad \textbf{fun}\ \mathit{System}\,() = \mathit{Transmitter}\,()\ \langle[\mathit{trans}]\rangle\ \mathit{Network}\,()\ \langle[\mathit{recv}]\rangle\ \mathit{Receiver}\,();
\end{array}
$$

Fig. 6. Implementation of the original system

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,

1. no activities or rates can be used within the model unless there is an accompanying declaration, and
2. an activity name cannot appear where a rate is expected.

The polymorphic type inference [11] 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 ability to describe activity names such as $tau\{hidden = recv\}$ which is a representation of a τ activity 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 *unit* pattern, analogous to the void type in Java. Each of the synchronisation sets used in the PEPA model contains only a single activity name but if more are required they can be included in lists such as $[trans, recv]$.

For this simple model to be extended with the addition of the monitoring feature we need to extend the *Activity* datatype with the activity names *low* and *high* and to extend the *Rate* datatype with the rate *t_half*. We then add the new component definitions and the new feature, building the formal description of the new system from the description of the existing one by the application of the monitoring feature. The definitions of the transmitter and monitor components are straightforward and are omitted here. The accompanying definitions are presented in Figure 7. The monitoring feature is parameterised by a component. Using Standard ML's layered pattern matching, identifiers are provided for this parameter as a whole (*Param*) and for its subcomponents (M, T, N, R). The *step* function unrolls the component definition by a single transition returning the (*act*, *rate*) pair and the one-step derivative, *Param′*. The activity type determines the subsequent behaviour.

```
fun S (Param as (M || ((T ⟨[trans]⟩ N) ⟨[recv]⟩ R))) () =
  let val ((act, rate), Param′) = step (Param) in
  case (act, rate) of
    (low, l) =>
      (low, l) ∗ S(Monitor′() || ((Transmitter′() ⟨[trans]⟩ N) ⟨[recv]⟩ R))
  | (high, h) =>
      (high, h) ∗ S(Monitor() || ((Transmitter() ⟨[trans]⟩ N) ⟨[recv]⟩ R))
  | default => (act, rate) ∗ S(Param′())
  end;

val System′ = S(Monitor() || System());
```

Fig. 7. Implementation of the monitoring feature

6 Related and future work

It is a good practice in any model development where a series of unpredictable revisions can occur to formally document system changes. A feature construct provides a formalism for such documentation. However, in the case of models which are used for performance analysis purposes there is an added urgency to formally document changes. The performance modelling process proceeds by specifying and evaluating measures of interest such as throughput, utilisation and bandwidth. These auxiliary definitions which accompany a system model are called *reward specifications*. Correct reward specifications are themselves difficult to obtain but the difficulty of relating reward specifications is acknowledged as being considerably more difficult again [12]. With a formal record of the changes to a model we at least have the promise of being able to automatically update the accompanying reward specifications. This remains as future work.

We are developing a model checker to allow formulae of the probabilistic modal logic PML_μ [13] to be checked over PEPA models which use the feature construct described here. Our present working implementation of the model checker has proven useful in detecting errors in PEPA models. The model checker has the useful facility to return a counterexample to show how the formula fails to be satisfied. This provides valuable guidance in the process of disgnosis of the critical flaw in the model and the subsequent re-working of the model to eliminate the error.

We have presented a method of describing additional features of a system separately from the description of the system itself. As features are progressively added, the complexity of a system inevitably grows. One avenue of future work is the investigation of the estimation of the additional complexity which is brought to the system by the addition of a single feature. Concretely, we could ask for a formula which computes how the size of the state space of the extended system will increase as a function of the state spaces of the systems which are used as actual parameters of parameterised components.

Conclusions

If performance modelling notations and tools are to realise the valuable contribution which they promise for the development of reliable and efficient complex software systems they must provide support not only for the initial design of systems but also for their correction, revision, and subsequent extension. We have shown how a new construct could be added to the PEPA performance modelling language: *parameterised components*. The new construct satisfies many of the desired goals of a feature construct and in addition promotes *structured whole-lifecycle performance modelling* of complex software systems. We have given the construct an operational semantics which builds upon the existing semantics of PEPA. We have applied the construct in a small case study.

Acknowledgements

The authors are grateful to Mark Ryan for suggestions which helped to improve the structure of the paper and the presentation of the feature construct in particular. This paper has benefited significantly from the helpful comments from the anonymous referees.

Stephen Gilmore is supported by Esprit Working group FIREworks and by the 'Distributed Commit Protocols' grant from the EPSRC. Jane Hillston is supported by the EPSRC 'COMPA' grant.

References

1. J. Hillston. *A Compositional Approach to Performance Modelling*. Cambridge University Press, 1996.
2. D.R.W. Holton. A PEPA specification of an industrial production cell. In S. Gilmore and J. Hillston, editors, *Proceedings of the Third International Workshop on Process Algebras and Performance Modelling*, pages 542–551. Special Issue of *The Computer Journal*, 38(7), December 1995.
3. 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.
4. A. El-Rayes, M. Kwiatkowska, and S. Minton. Analysing performance of lift systems in PEPA. In R. Pooley and J. Hillston, editors, *Proceedings of the Twelfth UK Performance Engineering Workshop*, pages 83–100, Department of Computer Science, The University of Edinburgh, September 1996.
5. H. Bowman, J. Bryans, and J. Derrick. Analysis of a multimedia stream using stochastic process algebra. In C. Priami, editor, *Sixth International Workshop on Process Algebras and Performance Modelling*, pages 51–69, Nice, September 1998.
6. L. Kloul, J.M. Fourneau, and F. Valois. Performance modelling of hierarchical cellular networks using PEPA. In J. Hillston, editor, *Proceedings of the Seventh International Workshop on Process Algebras and Performance Modelling*, Zaragosa, Spain, September 1999.
7. M. C. Plath and M. D. Ryan. Plug and play features. In W. Bouma, editor, *Feature Interactions in Telecommunications Systems V*. IOS Press, 1998.
8. M. D. Ryan. Feature-oriented programming: A case study using the SMV language. Technical report, School of Computer Science, The University of Birmingham, UK, September 1997.
9. R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML*. The MIT Press, 1996.
10. 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.
11. R. Milner. A theory of type polymorphism in programming languages. *Journal of Computer and System Science*, 17(3):348–375, 1978.

12. J. Bradley and N. Thomas. Constructing a partial order for performance measures. In *Proceedings of the Sixteenth Annual UK Performance Engineering Workshop*, pages 177–186, Durham, United Kingdom, July 2000. UK Performance Engineering Workshop Press.
13. 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 8.

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

Fig. 8. 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 9 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. The notation $r_\alpha(E)$ which is used in the third cooperation rule denotes the apparent rate of α in E.

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

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

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

Fig. 9. The operational semantics of PEPA

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

$$r_\alpha((\beta, r).P) = \begin{cases} r, & \alpha = \beta \\ 0, & \alpha \neq \beta \end{cases} \qquad r_\alpha(P + Q) = r_\alpha(P) + r_\alpha(Q)$$

$$r_\alpha(P/L) = \begin{cases} r_\alpha(P), & \alpha \notin L \\ 0, & \alpha \in L \end{cases} \quad r_\alpha(P \underset{L}{\bowtie} Q) = \begin{cases} r_\alpha(P) + r_\alpha(Q), & \alpha \notin L \\ \min(r_\alpha(P), r_\alpha(Q)), & \alpha \in L \end{cases}$$

Fig. 10. The apparent rate of α in PEPA components

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.

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 (see Figure 10). 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 in Figure 8.

A.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`.