# A Function-Equivalent Components Based Simplification Technique for PEPA Models*

Jane Hillston and Leïla Kloul

[1] LFCS, The University of Edinburgh, Edinburgh EH9 3JZ, Scotland
jeh@inf.ed.ac.uk
[2] PRiSM, Université de Versailles, 45, Av. des Etats-Unis, 78000 Versailles
kle@prism.uvsq.fr

**Abstract.** PEPA has recently been extended with functional rates [1][2]. These functions allow the specification of indirect interaction between components in such a way that the rate of an activity may be made dependent on the local state currently exhibited by one or more components. In this paper we demonstrate that these rates allow a systematic simplification of models in which there is appropriate indirect interaction between components. We investigate the interplay between this style of simplification and aggregation based on bisimulation, and establish a heuristic for applying both techniques in a complementary fashion.

## 1 Introduction

State space explosion remains the prevailing problem of most state-based modelling techniques. In general we focus on the impact of this problem on model solution when the size of the matrix representing the model becomes so large that the solution is intractable. However the largeness of models has other implications for the modelling process. During model construction the complexity of the system being represented may make it difficult for the modeller to keep track of all necessary aspects in the model. Compositional modelling techniques, such as stochastic process algebras, go some way towards alleviating this problem by supporting a *divide-and-conquer* approach to system representation. Nevertheless the number of components involved may still become large, representing a cognitive burden on the modeller.

In this paper we present a technique which aims to identify cases where components may be eliminated from a model description. In the simplest such cases the component may be eliminated without significant change to the remaining components. However, in general, these components may be playing a vital role within the model which must still be captured. In this case we achieve the same behaviour by the use of functional rates within the remaining components. Eliminating components in this way addresses the two issues of complexity

* This work was partially supported by the DEGAS IST-2001-32072, SENSORIA IST-3-016004-IP-09 funded by the FET Proactive Initiative on Global Computing and by EPSRC Advanced Research Fellowship EP/c543696/01.

discussed above. The model construction complexity is reduced, as the expression of the model is now made in terms of a fewer number of components. Moreover, the internal representation of the model is also, in general, more compact, as the functional rates allow a reduction in the size of the corresponding matrix. For a formalism like PEPA, for which a Kronecker representation has been developed [1], the reduction of the number of components in a model implies also the reduction of the number of matrices required for the tensorial representation of the corresponding Markov chain.

Since we aim to preserve the same behaviour the underlying Markov process remains unchanged and so the reachable state space is not reduced. Nevertheless we feel that this approach can be regarded as a model simplification technique since it has the effect of extending the class of tractable models.

The rest of this paper is structured as follows. In the following section, we give a brief overview of PEPA. We then go on to explain the version of PEPA extended with functional rates that we will use in this paper. In Section 3 we give a definition of the notion of *function-equivalent components* and show how functional rates may be used to eliminate these components from a model. In Section 5 we investigate the interplay between the model-level simplification approach based on function-equivalent components and the state-level aggregation technique based on the bisimulation relation, *strong equivalence*. This is illustrated by a small example. We present the algorithm for the automatic removal of function-equivalent components and demonstrate its application to a larger example in Section 6. We discuss related work in Section 7. Finally, we conclude in Section 8 with a summary of the results and a discussion of future work.

## 2 PEPA

In PEPA a system is described as an interaction of *components* which engage, either singly or multiply, in *activities*. These basic elements of PEPA, components and activities, correspond to *states* and *transitions* in the underlying Markov process. Each activity has an *action type*. Activities which are private to the component in which they occur are represented by the distinguished action type, $\tau$. The duration of each activity is represented by the parameter of the associated exponential distribution: the *activity rate*. This parameter may be any positive real number, or the distinguished symbol $\top$ (read as *unspecified*). Thus each activity, $a$, is a pair $(\alpha, r)$ consisting of the action type and the activity rate respectively. We assume a countable set of components, denoted $\mathcal{C}$, and a countable set, $\mathcal{A}$, of all possible action types. We denote by $\mathcal{A}ct \subseteq \mathcal{A} \times \mathbb{R}^+$, the set of activities, where $\mathbb{R}^+$ is the set of positive real numbers together with the symbol $\top$.

PEPA provides a small set of combinators which allow expressions to be constructed defining the behaviour of components, via the activities they undertake and the interactions between them.

**Prefix** $(\alpha, r).P$**:** This is the basic mechanism for constructing component behaviours. The component carries out activity $(\alpha, r)$ and subsequently behaves as component $P$.

**Choice $P+Q$:** This component may behave either as $P$ or as $Q$: all the current activities of both components are enabled. The first activity to complete, determined by a *race condition*, distinguishes one component, the other is discarded.

**Cooperation $P \underset{L}{\bowtie} Q$:** Components proceed independently with any activities whose types do not occur in the *cooperation set* $L$ (*individual activities*). However, activities with action types in the set $L$ require the simultaneous involvement of both components (*shared activities*). When the set $L$ is empty, we use the more concise notation $P \parallel Q$ to represent $P \underset{\emptyset}{\bowtie} Q$.

The published stochastic process algebras differ on how the rate of shared activities are defined [4]. In PEPA the shared activity occurs at the rate of the slowest participant. If an activity has an unspecified rate, denoted $\top$, the component is *passive* with respect to that action type. This means that the component does not influence the rate at which any shared activity occurs.

**Hiding $P/L$:** This behaves as $P$ except that any activities of types within the set $L$ are *hidden*, i.e. they exhibit the unknown type $\tau$ and can be regarded as an internal delay by the component. These activities cannot be carried out in cooperation with another component.

**Constant $A \overset{def}{=} P$:** Constants are components whose meaning is given by a defining equation. $A \overset{def}{=} P$ gives the constant $A$ the behaviour of the component $P$. This is how we assign names to components (behaviours).

The evolution of a model is governed by the structured operational semantics rules of the language. These define the admissible transitions or state changes associated with each combinator. A race condition governs the dynamic behaviour of a model whenever more than one activity is enabled.

The action types which the component $P$ may next engage in are the *current action types* of $P$, a set denoted $\mathcal{A}(P)$. This set is defined inductively over the syntactic constructs of the language [3]. For example, $\mathcal{A}(P+Q) = \mathcal{A}(P) \cup \mathcal{A}(Q)$.

The activities which the component $P$ may next engage in are the *current activities* of $P$, a multiset denoted $\mathcal{A}ct(P)$. When the system is behaving as component $P$ these are the activities which are enabled. Note that the dynamic behaviour of a component depends on the number of instances of each enabled activity and therefore we consider *multisets* of activities as opposed to *sets* of action types. $\mathcal{A}ct(P)$ is defined inductively over the structure of $P$.

The "states" of a PEPA model as it evolves are the syntactic terms, or *derivatives*, which the model will go through. The *derivative set* of a PEPA component $C$ is denoted $ds(C)$ and is the set of components which capture all the reachable states of the system. It is also necessary to refer to the complete set of action types which are used within the behaviour of a component $C$, i.e. all the possible action types which may be witnessed as a component evolves. This set will be denoted $\boldsymbol{\mathcal{A}}(C)$. The *complete action type set* of a component $C$ is:

$$\boldsymbol{\mathcal{A}}(C) = \bigcup_{C' \in ds(C)} \mathcal{A}(C').$$

Since we aim to undertake numerical solution it is important that we ensure that the components within the model, as well as the model itself, are finite and ergodic. Necessary (but not sufficient) conditions for the ergodicity of the Markov process in terms of the structure of the PEPA model have been identified and can be readily checked [3]. These conditions imply that the model must be a *cyclic* PEPA component.

A cyclic component is one in which behaviour may always be repeated, however the model evolves from this component it will always eventually return to this component and this set of behaviours. This leads us to define the syntax of PEPA expressions in terms of *sequential components* $S$ and *model components* $P$:

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

where $A$ denotes a constant which is either a sequential or a model component and $A_S$ denotes a constant which is a sequential component.

When a model component is defined it consists of one or more cooperating components, and these cooperating components remain apparent in every derivative of the model. Thus the sequential components involved in a model, and the cooperation sets in operation between them, will remain static throughout its evolution. Only the particular derivatives exhibited by each of the sequential components may change.

## 2.1 PEPA with Functional Rates

A functional dependency may involve one or several components. In a functional dependency involving a single component, the rate value of an activity of the component depends on the current state of the component itself. This is equivalent to the presence of several apparent rates for the activity in the component. Since each activity is represented explicitly in each local state it has always been possible to capture this form of dependency in PEPA. When this is expressed as a functional dependency, the rate value expressed as a function of the current component state is still a positive real number and can never be zero. This adds nothing new to the expressiveness of the language.

In contrast the ability to have an activity rate which is dependent on the local state of another component has not been possible previously (except in the special circumstance of shared activities). The introduction of this form of functional dependency into PEPA allows the dependent rate to include the value zero, indicating that an activity is blocked by the local state of another component. When the dependency is between two or more components it implies that either the activity to be performed by the first component and/or its rate value will be determined by the current state of the other component(s). The rate value may then be any non-negative real number $\mathbb{R}^+$ including zero, particularly when the choice of the activity to be performed is done according to the state of another component.

The introduction of functional dependencies in PEPA therefore requires us to relax the constraint on the definition domain of an activity rate. Thus, the set of activities $\mathcal{A}ct$ is now defined as $\mathcal{A}ct \subseteq \mathcal{A} \times \mathbb{R}^*$ where $\mathbb{R}^*$ is the set of non-negative real numbers together with symbol $\top$. The syntax of sequential components is modified to allow an activity to be defined in terms of an action type and an *expression* $e$, which can be either a rate, or a function, or a product of a rate and a function [1, 2].

$$S ::= (\alpha, e).S \mid S + S \mid A \qquad\qquad e ::= r \mid f \mid r \times f \mid \top$$

where $f : 2^{\mathcal{C}} \longrightarrow \mathbb{R}^*$ is a function from one or more components to the non-negative reals.

## 3 Function-Equivalent Components in PEPA

In this section we give a formal definition of the notion of *function-equivalent components* and show that PEPA models can be reduced by eliminating this type of component. This definition is the basis for automatically detecting suitable components within a model.

A sequential component $S_k$ is a *function-equivalent* component in a model component $C$ if $S_k$ is a sequential component of $C$ and for all derivatives $C_i \in ds(C)$ given the current derivatives of the other sequential components $S_{j_i}, j \neq k$ of the model, the current derivative of $S_k, S_{k_i}$, can be inferred. This definition implies that the function-equivalent component interacts with other components of the model in such a way that its states can always be inferred from the states of these components. For this to be the case it follows that the component never acts independently and all its activities are carried out in cooperation with the other components. As we will show later in the paper, if a sequential component is shown to be a function-equivalent component, then this component can be eliminated and replaced by appropriate functions in the other components of the model.

In order to formalise the definition of a function-equivalent component we consider first when a component can be identified as having no independent activities. Within a model component, each sequential component may be within the scope of several cooperation sets. For example, in the component

$$X \overset{\mathit{def}}{=} (P \underset{L}{\bowtie} R) \underset{K}{\bowtie} (S \underset{N}{\bowtie} T)$$

the subcomponent $R$ can act independently on any action types in the set $N$ which do not occur in $K$ or $L$, but must have the cooperation of other subcomponents to achieve actions in the set $L \cup K$, whereas the subcomponent S can act independently on any action types in the set $L \setminus (K \cup N)$, but must have the cooperation of other subcomponents to achieve actions in the set $K \cup N$. Thus we can identify the *interface* of a component i.e. those activities on which it must interact. In the following we formalise this idea. First, we define a partial order, $\prec$, over components, which captures the notion of *being a subcomponent*:

**Definition 1 (Subcomponents)**

*1.* $R \prec P$ *if* $R \in ds(P)$
*2.* $R \prec P + Q$ *if* $(R \prec P) \vee (R \prec Q)$

*3.* $R \prec P \underset{L}{\bowtie} Q$ *if* $(R \prec P) \vee (R \prec Q)$
*4.* $R \prec P/L$ *if* $R \prec P$
*5.* $R \prec A$ *if* $(A \overset{\text{def}}{=} P) \wedge (R \prec P)$

The interface of a sequential component within a component model is defined to be the union of all the cooperation sets whose scope includes the component $R$.

**Definition 2 (Interface).** *For any sequential component $R$ within a model component $C$ (i.e. $R \prec C$) the* interface *of $R$ within $C$, denoted $\mathcal{I}(C :: R)$, is the set of action types on which $R$ is required to cooperate. It is defined in terms of the subsidiary function $\mathcal{J}$; $\mathcal{I}(P :: R) = \mathcal{J}(P :: R, \emptyset)$, where $\mathcal{J}$ is defined as follows*

*1.* $\mathcal{J}(R :: R, K) = K$
*2.* $\mathcal{J}(P \underset{L}{\bowtie} Q :: R, K) = K' \cup K''$ *if* $\mathcal{J}(P :: R, K \cup L) = K'$ *and* $\mathcal{J}(Q :: R, K \cup L) = K''$
*3.* $\mathcal{J}(P/L :: R, K) = \mathcal{J}(P :: R, K \setminus L)$
*4.* $\mathcal{J}(A :: R, K) = \mathcal{J}(P :: R, K)$ *if* $A \overset{\text{def}}{=} P$
*5.* $\mathcal{J}(R' :: R, K) = \emptyset$ *if* $R \not\prec R'$.

When all the possible actions of a sequential component are constrained by its interface the component is never free to act independently; it must cooperate with other components to complete *any* action. Such a component can be viewed as being subservient to the rest of the model, and is called a *resource component.*

**Definition 3 (Resource Components).** *A sequential component $R$ in a model $C$ is a* resource component *if there is only one instance of $R$ within $C$ and the complete action type set of $R$ is a subset of its interface within $C$, i.e.*

$$\boldsymbol{\mathcal{A}}(R) \subseteq \mathcal{I}(C :: R)$$

*or if each instance of $R$ is a resource component within a submodel $C'$ of $C$ and the submodels are independent of each other (i.e. composed using $\|$).*

**Example 1: A distributed memory system.** Consider a system in which two processors act independently using two different memory elements. Each processor accesses to the data in its memory element, does computations and finally stores the results in its memory. As the access to the memory elements to get or save the data is defined by the processors, both memory elements can be modelled using the same component *Memory.*

$$\begin{array}{llll} Proc_1 & \overset{\text{def}}{=} (get, p_1).(compute, p_2).Proc_1' & Proc_2 & \overset{\text{def}}{=} (get, q_1).(compute, q_2).Proc_2' \\ Proc_1' & \overset{\text{def}}{=} (save, p_3).Proc_1 & Proc_2' & \overset{\text{def}}{=} (save, q_3).Proc_2 \end{array}$$

$$Memory \overset{\text{def}}{=} (get, \top).(save, \top).Memory$$

$$\mathbf{System_1} \overset{\text{def}}{=} (Memory \underset{\{get, save\}}{\bowtie} Proc_1) \parallel (Memory \underset{\{get, save\}}{\bowtie} Proc_2)$$

In this model defined by $\mathbf{System_1}$, *Memory* is a resource component because each of its instances in the model must synchronise on both activities *get* and *save.*

**Example 2:** Consider the following model components

$$R \overset{\text{def}}{=} (\alpha, r_\alpha).(\beta, r_\beta).R \qquad P \overset{\text{def}}{=} (\alpha, r_1).P \qquad Q \overset{\text{def}}{=} (\beta, r_2).Q$$

$$\mathbf{System_2} \overset{\text{def}}{=} (R \underset{\{\alpha\}}{\bowtie} P) \parallel (R \underset{\{\beta\}}{\bowtie} Q)$$

In contrast, in this model, $R$ is not a resource component since it can behave independently: one instance can act independently on $\alpha$ and the other can act independently on $\beta$.

In the following, we will focus on a particular class of resource components termed *simple resource components.* A sequential component $R$ is a *simple component* if

$$R \equiv S_1 + S_2 + \cdots + S_n$$

for some distinct cyclic components $S_1, S_2, \ldots S_n$ constructed using only prefix, with no repeated activities within a cycle and such that the last action of each cycle returns to $R$.

If a resource component is simple it implies that it offers alternative behaviours through its interface but once one of those behaviours is chosen (on the first action) the pattern of behaviour is set until the chosen cycle is completed and the choice is offered again. This type of repeated cyclic behaviour is, for example, exhibited by web services.

**Definition 4 (Arbiter).** *A simple resource component* $R \equiv S_1 + S_2 + \cdots + S_n$*, in a model* $C \equiv (P_1 \parallel P_2 \parallel \cdots \parallel P_k) \underset{L}{\bowtie} R$ *is an* arbiter *between* $P_1, P_2, \ldots, P_k$ *if for all* $i \in 1, \ldots, k$ *and* $j \in 1, \ldots, n$*, if* $\boldsymbol{\mathcal{A}}(P_i) \cap \boldsymbol{\mathcal{A}}(S_j) \neq \emptyset$ *then* $\boldsymbol{\mathcal{A}}(S_j) \subseteq \boldsymbol{\mathcal{A}}(P_i)$*.*

**Example 3: A simple web service.** Consider a system in which two clients interact with a web service *WS*. *Client*$_1$ repeatedly generates tasks of type 1 which it submits to the web service and waits for a response before displaying the results. *Client*$_2$ generates tasks of type 2 or type 3. Type 2 tasks require interaction with the web service, analogously to type 1 tasks, whereas type 3 tasks are processed locally before display. The PEPA model is the following.

$$\begin{aligned}
\mathit{Client}_1 &\overset{\text{def}}{=} (\mathit{task}_1, t_1).(\mathit{request}_1, \rho_1).\mathit{Client}_1' \\
\mathit{Client}_1' &\overset{\text{def}}{=} (\mathit{response}_1, \top).(\mathit{display}, d).\mathit{Client}_1
\end{aligned}$$

$$\begin{aligned}
\mathit{Client}_2 &\overset{\text{def}}{=} (\mathit{task}_2, t_2).(\mathit{request}_2, \rho_2).\mathit{Client}_2' \\
&\quad + \ (\mathit{task}_3, t_3).(\mathit{process}, p).(\mathit{display}, d).\mathit{Client}_2 \\
\mathit{Client}_2' &\overset{\text{def}}{=} (\mathit{response}_2, \top).(\mathit{display}, d).\mathit{Client}_2
\end{aligned}$$

$$\mathit{WS} \overset{\text{def}}{=} (\mathit{request}_1, \top).(\mathit{response}_1, r_1).\mathit{WS} + (\mathit{request}_2, \top).(\mathit{response}_2, r_2).\mathit{WS}$$

$$\mathbf{Web\_Service} \overset{\text{def}}{=} \big(\mathit{Client}_1 \parallel \mathit{Client}_2\big) \underset{L}{\bowtie} \mathit{WS}$$

where $L = \{\mathit{request}_1, \mathit{response}_1, \mathit{request}_2, \mathit{response}_2\}$. In this model, component *WS* is an arbiter between components *Client*$_1$ and *Client*$_2$.

**Proposition 1.** *If a simple resource component $R$ is an arbiter within a PEPA model $C$, then $R$ is a function-equivalent component and thus can be removed from the model, the Markov process generated by the resulting model $C'$ being isomorphic to the Markov process underlying $C$.*

**Proof:** As all activities of an arbiter component are shared activities, each step of the evolution of this component coincide with one step of evolution of one of the other components of the model. The structure of the arbiter, and the form of its interaction with the other components of the model ensure that even if it changes state during the evolution of the model, all its states can be inferred from the other components of the model. $\square$

Such a component allows two or more other components of the model to indirectly influence one another. This kind of interaction between the components can be managed using functions in the rates of their shared activities. The definition of these functions takes into account the states of the components involved in the cooperation. This allows the model to keep the form of scheduling initially imposed by $R$.

The introduction of appropriate functional rates instead of an arbiter component will not have any impact on the correctness of the model and thus the Markov process underlying the reduced model is guaranteed to be isomorphic to the one generated by the initial model.

**Corollary 1.** *If a simple resource component $R$ is a single state arbiter within a PEPA model $C$, then $R$ is an identity function-equivalent component and thus can be removed from the model, the Markov process generated by the resulting model $C'$ being isomorphic to the Markov process underlying $C$ (subject to some transfer of rates when $C$ is passive with respect to an activity).*

This corresponds to the case of an arbiter in which the cycle of each component $S_i$ in $R$ has only one activity:

$$R \overset{def}{=} (\alpha_1, r_1).R + (\alpha_2, r_2).R + \ldots + (\alpha_n, r_n).R$$

As $R$ always exhibits the same state and all its activities are shared activities, $R$ can be removed from the model. This type of component does not enforce any scheduling between the other components of the model as the cycle defining the use of $R$ by any component $P_i$ is limited to one activity. For this reason its removal is not conditioned by the use of functional rates. Therefore, when removed, $R$ can be replaced by an identity function in the other components which share $R$'s activities.

However we should consider carefully the activity rates of $R$. If all activities have unspecified rates ($\top$), then the rate of these activities are already defined by the other components of the model. So $R$ can be eliminated without any impact on the other components of the model. If at least one of its activities $\alpha$ has a specified rate, then we need to compute the rate of each instance of $\alpha$ in each component $P_i$ which shares this activity with $R$. As explained in Section 6 this computation takes place during the generation of the derivation graph.

In the following section, we show how an arbiter component can be eliminated from the model using functional rates.

## 4 A Function Based Simplification Approach

In PEPA components are able to influence one another in two ways, both related to activities. The first one is a direct interaction between the components and is modelled using shared activities (cooperation). The other form of interaction is less direct as the activity rate within a component can be influenced by the local states of one or more other components in the model. This implies that the activity may or may not be performed by the component according to a rate value determined by the current state of the other component(s). Indeed, this rate may have any non-negative value, including zero which aborts the activity.

In general, the use of functional rates can lead to a reduction in the model expression because they avoid explicitly modelling all parts of a system's behaviour. This is the case for PEPA models with arbiter resource components.

We have seen so far an arbiter component may be necessary to ensure the correct behaviour of a model; they enforce the necessary scheduling between the model's components. Thus, an arbiter component may be seen as another indirect means for the components of a model to influence one another. But, as stated previously, this is exactly what the functional rates allow us to do in PEPA. Therefore, we propose to replace arbiter components using functional rates in the other components of the model with which they share their activities. For example, consider again the web service model (**Example 3**). In this model, component *WS* is an arbiter component and therefore can be removed from the model and replaced with appropriate functions as follows:

$$\begin{aligned}
Client_1 &\overset{\text{def}}{=} (task_1, t_1).(request_1, f \times \rho_1).Client_1' \\
Client_1' &\overset{\text{def}}{=} (response_1, r_1).(display, d).Client_1
\end{aligned}$$

$$\begin{aligned}
Client_2 &\overset{\text{def}}{=} (task_2, t_2).(request_2, g \times \rho_2).Client_2' \\
&\quad + \ (task_3, t_3).(process, p).(display, d).Client_2 \\
Client_2' &\overset{\text{def}}{=} (response_2, r_2).(display, d).Client_2
\end{aligned}$$

$$\textbf{Web\_Service2} \overset{\text{def}}{=} Client_1 \parallel Client_2$$

where $f$ and $g$ are defined as follows:

$$f(x) = \begin{cases} 0 \ \textit{if} \ x \equiv Client_2' \\ 1 \ \textit{otherwise} \end{cases} \qquad g(y) = \begin{cases} 0 \ \textit{if} \ y \equiv Client_1' \\ 1 \ \textit{otherwise} \end{cases}$$

$y$ and $x$ being the state of component $Client_1$ and $Client_2$ respectively.

**Example 4: A multiprocessor shared memory.** Consider a system in which two processors compete for access to a shared memory via a bus. The processors are independent and both follow the same pattern of behaviour: each does computations, acquires the bus, sends the message and then releases the bus.

$$
\begin{aligned}
&\mathit{Proc} \overset{\text{def}}{=} (\mathit{compute}, r_1).(\mathit{acquire}, r_2).\mathit{Proc}' && \mathit{Bus} \overset{\text{def}}{=} (\mathit{acquire}, \top).\mathit{Bus}' \\
&\mathit{Proc}' \overset{\text{def}}{=} (\mathit{transmit}, r_3).(\mathit{release}, r_4).\mathit{Proc} && \mathit{Bus}' \overset{\text{def}}{=} (\mathit{release}, \top).\mathit{Bus}
\end{aligned}
$$

$$\mathit{Memory} \overset{\text{def}}{=} (\mathit{transmit}, \top).\mathit{Memory}$$

$$\mathbf{Machine} \overset{\text{def}}{=} \left((\mathit{Proc} \parallel \mathit{Proc}) \underset{L_1}{\bowtie} \mathit{Bus}\right) \underset{L_2}{\bowtie} \mathit{Memory}$$

where $L_1 = \{\mathit{acquire}, \mathit{release}\}$ and $L_2 = \{\mathit{transmit}\}$. In this model, components *Memory* and *Bus* are arbiter components and therefore can be removed from the model. The elimination of *Memory* is straightforward and does not require the modification of the other components because the rate of the shared activity *transmit* is unspecified in *Memory*. In contrast, in order to remove component *Bus*, we need to introduce appropriate functional rates in the other components where the cooperation activities *acquire* and *release* appear. By doing so, we obtain the following model:

$$
\begin{aligned}
&\mathit{Proc}_{10} \overset{\text{def}}{=} (\mathit{compute}, r_1).\mathit{Proc}_{11} && \mathit{Proc}_{20} \overset{\text{def}}{=} (\mathit{compute}, r_1).\mathit{Proc}_{21} \\
&\mathit{Proc}_{11} \overset{\text{def}}{=} (\mathit{acquire}, f_1 \times r_2).\mathit{Proc}_{12} && \mathit{Proc}_{21} \overset{\text{def}}{=} (\mathit{acquire}, f_2 \times r_2).\mathit{Proc}_{22} \\
&\mathit{Proc}_{12} \overset{\text{def}}{=} (\mathit{transmit}, r_3).\mathit{Proc}_{13} && \mathit{Proc}_{22} \overset{\text{def}}{=} (\mathit{transmit}, r_3).\mathit{Proc}_{23} \\
&\mathit{Proc}_{13} \overset{\text{def}}{=} (\mathit{release}, g_1 \times r_4).\mathit{Proc}_{10} && \mathit{Proc}_{23} \overset{\text{def}}{=} (\mathit{release}, g_2 \times r_4).\mathit{Proc}_{20}
\end{aligned}
$$

$$\mathbf{Machine}' \overset{\text{def}}{=} \mathit{Proc}_{10} \parallel \mathit{Proc}_{20}$$

where $f_j$ and $g_j$, $j = 1, 2$, are functions defined, when $k = 1, 2$ $k \neq j$, as

$$
f_j(y) = \begin{cases} 1 \; \mathit{if} \;\; y \equiv \mathit{Proc}_{k0} \\ 0 \; \mathit{otherwise} \end{cases} \qquad g_j(x) = \begin{cases} 1 \; \mathit{if} \;\; x \equiv \mathit{Proc}_{k0} \\ 0 \; \mathit{otherwise} \end{cases}
$$

$x$ and $y$ being the state of $\mathit{Proc}_k$ appropriately. Note that the functions $g_j$ are not essential for the correct behaviour of the model and may be omitted. The functions $f_j$ are sufficient to guarantee the correct behaviour of the model.

Both previous examples show that the functions are only necessary in the first activity that a component $P$ shares with the resource component. For $P$, the function associated with the rate of the first activity can be regarded as the *access ticket* to the resource component and this ticket must be validated to make the access possible. Once $P$ is using the resource component, the other components of the model cannot use it.

## 5 Interplay Between the Function-Based Simplification Approach and the Aggregation Technique

There is an established aggregation technique for PEPA models based on the notion of *strong equivalence* between states. The aggregation may result in a single component being substituted for a number of components. Thus, like the function-based simplification technique, it can result in a reduction in the model expression. However, in the case of aggregation, this usually had the consequence that the underlying state space is also reduced. This suggests that the best results may be obtained if the two techniques are applied in conjunction. In this section we make some observations about how this can be achieved and the interplay between the two techniques.

Recall that if $\mathcal{C}$ is the set of all PEPA components, and $q[P, S, \alpha]$ is the total conditional transition rate from component $P$ to the set of components $S$, then strong equivalence is defined as follows:

**Definition 5 (Strong equivalence).** *An equivalence relation* $\mathcal{R} \subseteq \mathcal{C} \times \mathcal{C}$ *is a* strong equivalence *if whenever* $(P, Q) \in \mathcal{R}$ *then for all* $\alpha \in \mathcal{A}$ *and for all* $S \in \mathcal{C}/R$

$$q[P, S, \alpha] = q[Q, S, \alpha]$$

In many cases strong equivalence exists between the derivatives of a model because there is strong equivalence between components of the model and their interleaving can be eliminated.

Let us consider the original model of the multiprocessor system as given in **Example 4** and apply the aggregation technique on component $\mathit{Proc} \parallel \mathit{Proc}$ of the system equation. The model resulting from such an operation is the following, where a single derivative represents each equivalence class.

$$\begin{array}{llll}
\mathit{Procs}_{00} & \overset{\text{def}}{=} & (\mathit{compute}, 2r_1).\mathit{Procs}_{01} & [({\scriptstyle Proc}10, {\scriptstyle Proc}20)]_{00} \\
\mathit{Procs}_{01} & \overset{\text{def}}{=} & (\mathit{compute}, r_1).\mathit{Procs}_{11} & \\
 & + & (\mathit{acquire}, r_2).\mathit{Procs}_{02} & [({\scriptstyle Proc}10, {\scriptstyle Proc}21), ({\scriptstyle Proc}11, {\scriptstyle Proc}20)]_{01} \\
\mathit{Procs}_{02} & \overset{\text{def}}{=} & (\mathit{transmit}, r_3).\mathit{Procs}_{03} & \\
 & + & (\mathit{compute}, r_1).\mathit{Procs}_{12} & [({\scriptstyle Proc}10, {\scriptstyle Proc}22), ({\scriptstyle Proc}12, {\scriptstyle Proc}20)]_{02} \\
\mathit{Procs}_{03} & \overset{\text{def}}{=} & (\mathit{compute}, r_1).\mathit{Procs}_{13} & \\
 & + & (\mathit{release}, r_4).\mathit{Procs}_{00} & [({\scriptstyle Proc}10, {\scriptstyle Proc}23), ({\scriptstyle Proc}13, {\scriptstyle Proc}20)]_{03} \\
\mathit{Procs}_{11} & \overset{\text{def}}{=} & (\mathit{acquire}, 2r_2).\mathit{Procs}_{12} & [({\scriptstyle Proc}11, {\scriptstyle Proc}21)]_{11} \\
\mathit{Procs}_{12} & \overset{\text{def}}{=} & (\mathit{transmit}, r_3).\mathit{Procs}_{13} & [({\scriptstyle Proc}11, {\scriptstyle Proc}22), ({\scriptstyle Proc}12, {\scriptstyle Proc}21)]_{12} \\
\mathit{Procs}_{13} & \overset{\text{def}}{=} & (\mathit{release}, r_4).\mathit{Procs}_{01} & [({\scriptstyle Proc}11, {\scriptstyle Proc}23), ({\scriptstyle Proc}13, {\scriptstyle Proc}21)]_{13}
\end{array}$$

$$\mathit{Bus} \overset{\text{def}}{=} (\mathit{acquire}, \top).(\mathit{release}, \top).\mathit{Bus} \qquad \mathit{Memory} \overset{\text{def}}{=} (\mathit{transmit}, \top).\mathit{Memory}$$

$$\mathbf{Machine_a} \overset{\text{def}}{=} \left(\mathit{Procs}_{00} \underset{L_1}{\bowtie} \mathit{Bus}\right) \underset{L_2}{\bowtie} \mathit{Memory}$$

where $L_1 = \{\mathit{acquire}, \mathit{release}\}$ and $L_2 = \{\mathit{transmit}\}$. In the new model, *Memory* is a single state arbiter component as before, and its removal is straightforward. Similarly, *Bus* is also an arbiter component as before. However, we would like to highlight that in the aggregated model, its removal is no longer conditioned by the use of functional rates in the other components of the model. Indeed, the aggregation has reduced the number of components that have to cooperate with *Bus* to one component. Moreover, the scheduling imposed previously by *Bus* between the two model components *Proc* is already taken into account in the resulting component $Procs_{00}$. Therefore the removal of *Bus* becomes straightforward and the new system equation is $\mathbf{Machine'_a} \stackrel{\text{def}}{=} Procs_{00}$.

In contrast, if we first apply the function-based simplification technique we obtain the model $\mathbf{Machine'} \stackrel{\text{def}}{=} Proc_{10} \parallel Proc_{20}$ defined earlier in which $Proc_1$ and $Proc_2$ involve functional rates. In order to apply aggregation to this model we must first define strong equivalence for components which contain functional rates. The original definition of strong equivalence suggests that this will be achieved by extending the definition of conditional transition rate to include the possibility that the transition rate concerned may be a function.

**Definition 6 (Conditional transition rate).** *The* conditional transition rate *between two derivatives $C_i$ and $C_j$, via a given action type $\alpha$, denoted $q(C_i, C_j, \alpha)$, is defined to be the sum of the constant and the functional activity rates associated with transitions between $C_i$ and $C_j$ in the derivation graph which are labelled by $\alpha$.*

Note that the evaluation of a function is unequivocal because we are considering the transition rates from a particular derivative. Each derivative corresponds to a particular set of local states for each component, thus determining the appropriate value of the function.

The total conditional transition rate to a set is defined, as previously, as the sum of conditional transition rates from the component to elements of the set. Thus it follows that two derivatives with functional rates for an action type $\alpha$ will be strongly equivalent if the functions in each derivative will give the same transition rate to each strong equivalence class. For the example above this is readily shown to be the case. In the cases where the equivalence class has more than one element it is clear that $Proc_{10} \parallel Proc_{2k} \cong Proc_{1k} \parallel Proc_{20}$ for all $k = 0, 1, 2, 3$ and $f_1$ and $f_2$ ($g_1$ and $g_2$) will have the same evaluation.

If the techniques are considered as alternative means of model simplification there is a clear preference for strong equivalence based aggregation since this can reduce the size of the underlying state space. But when both techniques are applicable, it seems that it is possible to apply the techniques in either order. However, in general, establishing strong equivalence without functional rates will be less involved and therefore to be preferred computationally. Moreover, as we have seen, first carrying out the aggregation to remove interleavings may simplify the function-based reduction because the need for functions to control scheduling may have been eliminated.

In the following we describe the algorithm that allows the simplification of a PEPA model by removing the arbiter components.

## 6 An Algorithm for Eliminating Arbiter Components

Assume that $R$ is an arbiter in model $C$, i.e. $R$ is a simple resource component $R \equiv S_1 + S_2 + \cdots + S_n$, in a model $C \equiv (P_1 \parallel P_2 \parallel \cdots \parallel P_k) \underset{L}{\bowtie} R$. Then for each component $P_i$ we can partition the derivative set of $P_i$ into two disjoint subsets

$ds_R(P_i)$, corresponding to states where $P_i$ is "using" $R$ and $ds_{free}(P_i)$ where it is not. A component is "using" the resource when it has cooperated on the first activity of one of the cycles $S_i$ in $R$, but not yet cooperated on the last activity of the cycle.

$$ds(P_i) = ds_R(P_i) \cup ds_{free}(P_i) \qquad\qquad ds_R(P_i) \cap ds_{free}(P_i) = \emptyset$$

Moreover, we denote by $C\_state(P_i)$ the current state of component $P_i$ and define the two sets $\mathcal{B}(\alpha)$ and $\mathcal{I}nput(R)$. The former contains the components which have $\alpha$ in their action type set and the latter the action types that component $R$ may engage in, in its initial state.

**Single state arbiter components.** This type of component has only one derivative but may have the choice between several activities to engage in.

$$R \overset{def}{=} (\alpha_1, r_1).R + (\alpha_2, r_2).R + \ldots + (\alpha_n, r_n).R$$

As stated before, when such a component is removed, logically it can be replaced by an identity function because it does not enforce any scheduling between the other components of the model. In practical terms the introduction of such a function can be omitted as it has no impact on the behaviour but overloads the notation and introduces an unnecessary extra computation time.

However, when removing such a component from a model, we should pay attention to the rates of the activities in this component. These rates may be real values or unspecified rates and according to this the elimination of this component may or may not bring changes to the other components of the model.

1. If all the activities enabled by the arbiter component have unspecified rates, then the component can be removed automatically from the model without any changes in the other components of the model.
2. If the rate of an action type $\alpha_i \in \boldsymbol{\mathcal{A}}(R)$ is specified, then for each component $P_j$ in the model and each instance $\alpha_i^*$ of $\alpha_i$ in $\boldsymbol{\mathcal{A}ct}(P_j)$, a new rate should be computed. For that we need to know the current state of each component in $\mathcal{B}(\alpha)$. To define this within a function would necessitate a complete derivation of the state space of the remainder of the model. Thus we leave this to be done during the usual generation of the derivation graph of the model. For each arc of the graph to generate with label $\alpha$, we consider the components in competition for this action type and compute the associated rate for $\alpha$ considering the rates of these components at this stage of the graph. The functions instead are used to control whether an activity of this type should be allowed or not, without complete specification of the rate. Thus our functions always have the value 1 or 0 which can be regarded as switching an activity *on* or *off* respectively.

Note that from the point of view of the state space, the removal of a single state arbiter component does not bring any benefit.

**Multi-states arbiter components.** Unlike single state arbiter components, the removal of these components from the model allows us to reduce the size of the model, its representation and its state space when the Kronecker form is the internal representation used.

The general algorithm consists mainly in defining, for each component $P_i$ and action type $\alpha$ shared between this component and an arbiter component, a function $g_i(\alpha)$. This function allows $P_i$ to *know* when the arbiter component is free and thus usable. Application of the algorithm takes place at the model (syntactic) level and results in the removal of all arbiter components.

```
//Algorithm

 for each component $P_i$
     // define a function $f_i$ over $ds(P_i)$
     for each $P'_i \in ds(P_i)$

         $f_i = \begin{cases} 1 \text{ if } P'_i \in ds_{free}(P_i) \\ 0 \text{ if } P'_i \in ds_R(P_i) \end{cases}$

     end for
 end for

for each $\alpha \in \mathcal{I}nput(R)$
     for each $P_j \in \mathcal{B}(\alpha)$

         $g_j(\alpha) = \prod_{\substack{i=1, i \neq j \\ P'_i = C\_State(P_i)}}^{n} f_i(P'_i)$

         replace $(\alpha, r)$ in $P_j$ by $(\alpha, g_j(\alpha) \times r)$
     end for
end for
```

## 7 Related Work

Current research addresses the definition of efficient techniques for constructing and analysing large models. These techniques fall into two categories: "largeness avoidance" and "largeness tolerance" [9]. While the former refers to approaches that aim to keep the size of the model representation as small as possible at every stage of the modelling and analysis process, the latter category focusses on sparse storage techniques and memory-efficient numerical methods.

Like decomposition techniques, the Kronecker approach, and techniques which exploit model symmetries, the function-equivalent components based simplification technique belongs to the first category. However, unlike these techniques which have been widely reported in the literature, the possibility of using functional rates to eliminate components has not really been investigated. This technique has been identified for SAN for some time [5]. However, to the best of our knowledge, there has not been any work on a systematic way to identify suitable components and automatically eliminate them. Instead the previous approach relies on the expertise/skill of the modeller. Here we can envisage the elimination being carried out transparently to the user since the identification of suitable components is based on readily checked syntactic conditions, and the algorithm of the previous section provides an automatic means of carrying out the reduction.

Process algebras which encompass functions have previously appeared in the literature (see for example $\mu$CRL [6][7]). However the role of the functions here is somewhat different. Moreover this is the first stochastic process algebra to incorporate them.

## 8 Conclusions

In this paper, we have identified a class of *function-equivalent* PEPA components and we have shown, using functional rates, that any component of this class can be eliminated from a model. An algorithm allowing the automatic removal of these components has been developed.

This new simplification technique allows the reduction of the number of components in a model and thus the number of matrices required for a Kronecker representation of the underlying Markov process, when this internal representation is used. Furthermore the model expression is simplified although the state space remains the same and subsequent solution is exact.

Moreover, we have investigated the interplay between this simplification approach and the aggregation technique characterised by strong equivalence in PEPA. Combining these techniques, when possible, may allow the modeller to push the current limits of PEPA in terms of tractable systems. Our goal in the future is to extend the class of function-equivalent components and to integrate this new simplification approach into the PEPA Workbench [8].

## References

1. J. Hillston and L. Kloul, An Efficient Kronecker Representation for PEPA Models, LNCS, N.2165, pp.120-135, Proc. of the Joint International Workshop, PAPM-PROBMIV 2001, Aachen
2. J. Hillston and L. Kloul, Formal Techniques for Performance Analysis: blending SAN and PEPA, Submitted
3. J. Hillston, A compositional approach to performance modelling, PhD. The University of Edinburgh, 1994
4. J. Hillston The Nature of Synchronisation Proc. of 2nd Process Algebra and Performance Modelling Workshop, U. Herzog and M. Rettelbach (Editors), 1994
5. P. Fernandes, B. Plateau and W.J. Stewart Optimizing Tensor Product Computations in Stochastic Automata Networks, RAIRO - Operations Research, vol. 32, no. 3, pp:325-351, July, 1998
6. J.F. Groote and A. Ponse, Proof Theory for $\mu$CRL: A Languages for Processes with Data, Semantics of Specification Languages 1993, pp.232-251
7. J.F. Groote and A. Ponse, The Syntax and Semantics of $\mu$CRL, In A.Ponse,C. Verhoef & S.F.M. van Vlijmen, eds, Algebra of Communicating Processes, Workshops in Computing, pp.26-62,1994
8. S. Gilmore and J. Hillston, The PEPA Workbench: A Tool to Support a Process Algebra-based Approach to Performance Modelling, Proc. of the 7th Int. Conf. on Modelling Techniques and Tools for Computer Performance Evaluation, LNCS, Vol.794, pp:353-368, Vienna 1994
9. K.S. Trivedi and M. Malhotra, Reliability and Performability Techniques and Tools: A Survey, In B. Walke and O. Spaniol, editors, Messung, Modellierung und Bewertung von Rechen- und Kommunikationssystemen, pages 27–48, Springer, Aachen, September 1993