An Efficient Algorithm for Aggregating PEPA Models

Stephen Gilmore, Jane Hillston and Marina Ribaudo

Abstract— **Performance Evaluation Process Algebra (PEPA) is a formal language for performance modelling based on process algebra. It has previously been shown that using the process algebra apparatus compact performance models can be derived which retain the essential behavioural characteristics of the modelled system. However no efficient algorithm for this derivation was given. In this paper we present an efficient algorithm which recognises and takes advantage of symmetries within the model and avoids unnecessary computation. The algorithm is illustrated by a multiprocessor example.**

Keywords— **Performance modelling, model aggregation, performance evaluation tools, stochastic process algebras.**

I. Introduction

In recent years several Markovian process algebras (MPAs) have been presented in the literature. These include PEPA [1], MTIPP [2], and EMPA [3]. As with classical process algebras, these formalisms allow models of systems to be constructed which are amenable to functional or behavioural analysis by a variety of techniques. Additionally they allow timing information to be captured in those models and so facilitate performance analysis via the solution of a Continuous Time Markov Chain (CTMC).

Process algebras have several attractive features: a facility for high-level definition, compositional structure and the existence of formally defined equivalence relations which can be used to compare models. In the Markovian context theoretical results have shown that it is possible to exploit these equivalence relations at the level of the model description to generate an aggregated CTMC in a compositional way [4]. This is of great practical importance because, like all state-based modelling techniques, MPA models suffer from the *state space explosion* problem. Although prototype tools have been developed for model exploration [5, 6, 7], little work has been done to exploit to the full the potential to use equivalence relations to achieve effective aggregation and thus to put the theoretic results to practical use. In this paper we describe an algorithm to carry out efficient aggregation and its implementation in the PEPA Workbench.

Aggregation is a widely used and well-understood technique for reducing the size of CTMC used in performance analysis. The state space of the CTMC is partitioned into a number of classes, each of which is treated as a single state in a new derived stochastic process. If the partition can be shown to have a condition known as *lumpability* [8], this new stochastic process will again be a CTMC and amenable to numerical solution of a steady state probability distribution via linear algebra. In the MPA context the partitioning is carried out using a formally defined equivalence relation which establishes behavioural or observational equivalence between states within a model. The equivalence relation which is generally discussed in relation to aggregation is called *strong equivalence* (for PEPA), *Markovian bisimulation* (for MTIPP) or *extended Markovian bisimulation equivalence* (for EMPA). However there are some problems with applying this equivalence relation/aggregation at the syntax level in a compositional way. These are discussed in more detail in Section V. In this paper we use a finer equivalence relation, called *isomorphism*, which although it may result in coarser aggregations has the advantage of being readily amenable to automatic generation of equivalence classes at the syntax level. Thus the construction of the complete state space can be avoided and the aggregated CTMC is constructed directly.

The rest of the paper is structured as follows. In Section II we introduce the PEPA language, its operational semantics, and aggregation via isomorphism. The algorithm for the computation of a reduced state space is discussed in Section III, an example is presented in Section IV. Some cases in which the algorithm cannot achieve the optimal theoretical partitioning are discussed in Section V. Section VI presents some related approaches and, finally, Section VII concludes the paper presenting some possible future investigation.

II. PEPA

Performance Evaluation Process Algebra (PEPA) is an algebraic description technique based on a classical process algebra and enhanced with stochastic timing information. This extension results in models which may be used to calculate performance measures as well as deduce functional properties of the system. In this section we briefly introduce PEPA; more detailed information can be found in [1].

Process algebras are mathematical theories which model concurrent systems by their algebra and provide apparatus for reasoning about the structure and behaviour of the model. In classical process algebras, e.g. Calculus of Communicating Systems (CCS [9]), time is abstracted away—actions are assumed to be instantaneous and only relative ordering is represented—and choices are generally nondeterministic. If an exponentially distributed random variable is used to specify the duration of each action the process algebra may be used to represent a Markov process.

S. Gilmore and J. Hillston are with the Laboratory for Foundations of Computer Science, The University of Edinburgh, Scotland. Email: {stg, jeh}@dcs.ed.ac.uk

M. Ribaudo is with the Dipartimento di Informatica, Università di Torino, Italy. Email: marina@di.unito.it

This approach is taken in PEPA and several of the other Markovian process algebras [2, 3].

The basic elements of PEPA are *components* and *activities*, corresponding to *states* and *transitions* in the underlying CTMC. Each activity is represented by two pieces of information: the label, or *action type*, which identifies it, and the *activity rate* which is the parameter of the negative exponential distribution determining its duration. Thus each action is represented as a pair (α, r). We assume that the set of possible action types, $\mathcal{A}$, includes a distinguished type, τ. This type denotes internal, or "unknown" activities and provides an important abstraction mechanism.

The process algebra notation for representing systems is wholly based on the use of a formal 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 following grammar:

$$\begin{array}{lcl} S & ::= & (\alpha, r).S \ \mid \ S + S \ \mid \ C_S \\ P & ::= & P \underset{L}{\bowtie} P \ \mid \ P/L \ \mid \ C \end{array}$$

where 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_S stands for constants which denote sequential components. The component combinators, together with their names and interpretations, are presented informally below.

Prefix, $(\alpha, r).S$

The basic mechanism for describing the behaviour of a system is to give a component a designated first action using the prefix combinator, ".". 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. The set of all action types is denoted by $\mathcal{A}$. Sequences of actions can be combined to build up a life cycle for a component. For example:

$$\mathit{Comp} \stackrel{\text{def}}{=} (\mathit{error}, \epsilon).(\mathit{repair}, \rho).\mathit{Comp}$$

Choice, $S + S$

The life cycle of a sequential component may be more complex than any behaviour which can be expressed using the prefix combinator alone. The choice combinator captures the possibility of competition or selection between different possible activities. The component $S_1 + S_2$ represents a system which may behave either as S_1 or as S_2. The activities of both S_1 and S_2 are enabled. The first activity to complete distinguishes one of them: the other is discarded. The system will then behave as the derivative resulting from the evolution of the chosen component. For example, the faulty component considered above may also be capable of completing a task satisfactorily:

$$\mathit{Comp} \stackrel{\text{def}}{=} (\mathit{error}, \epsilon).(\mathit{repair}, \rho).\mathit{Comp} \ + \ (\mathit{task}, \mu).\mathit{Comp}$$

Constant, C

As we have already seen, it is convenient to be able to assign names to patterns of behaviour associated with components. Constants provide a mechanism for doing this. They are components whose meaning is given by a defining equation: e.g. $C \stackrel{\text{def}}{=} P$, which gives the constant C the behaviour of the component P.

Cooperation, $P \underset{L}{\bowtie} P$

Most systems are comprised of several components which interact. In PEPA direct interaction, or *cooperation*, between components is represented by the combinator "$\underset{L}{\bowtie}$". The set L, of visible action types ($L \subseteq \mathcal{A} \setminus \{\tau\}$), is significant because it determines those activities on which the components are forced to synchronise. Thus the cooperation combinator is in fact an indexed family of combinators, one for each possible *cooperation set* L. When cooperation is not imposed, namely for action types not in L, the components proceed independently and concurrently with their enabled activities. However if a component enables an activity whose action type is in the cooperation set it will not be able to proceed with that activity until the other component also enables an activity of that type. The two components then proceed together to complete the *shared activity.* The rate of the shared activity may be altered to reflect the work carried out by both components to complete the activity.

For example, the faulty component considered above may need to cooperate with a resource in order to complete its task. This cooperation is represented as follows:

$$\mathit{System} \stackrel{\text{def}}{=} \mathit{Comp} \underset{\{\mathit{task}\}}{\bowtie} \mathit{Res}$$

If the component also needs to cooperate with a repairman in order to be repaired this could be written as:

$$\mathit{System} \underset{\{\mathit{repair}\}}{\bowtie} \mathit{Repman}$$

or, equivalently

$$(\mathit{Comp} \underset{\{\mathit{task}\}}{\bowtie} \mathit{Res}) \underset{\{\mathit{repair}\}}{\bowtie} \mathit{Repman}$$

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, denoted $(\alpha, \top)$. This means that the rate of the activity is left unspecified and is determined upon cooperation, by the rate of the activity in the other component. All passive actions must be synchronised in the final model.

If the cooperation set is empty, the two components proceed independently, with no shared activities. We use the compact notation, $P \parallel Q$, to represent this case. Thus, if two components compete for access to the resource and the repairman we would represent the system as

$$((\mathit{Comp} \parallel \mathit{Comp}) \underset{\{\mathit{task}\}}{\bowtie} \mathit{Res}) \underset{\{\mathit{repair}\}}{\bowtie} \mathit{Repman}$$

Hiding, P/L

The possibility to abstract away some aspects of a component's behaviour is provided by the hiding operator "/". 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. For example, in the system introduced above we may wish to ensure that these components have exclusive access to the resource in order to complete their task. Thus we hide the action type *task*, ensuring that even when the system is embedded in an environment no other component can access the task activity of the resource:

$$\mathit{System} \stackrel{\text{def}}{=} ((\mathit{Comp} \parallel \mathit{Comp}) \underset{\{\mathit{task}\}}{\bowtie} \mathit{Res})/\{\mathit{task}\}$$

Once an activity is hidden it only appears as the unknown type τ; the rate of the activity, however, remains unaffected.

The precedence of the combinators provides a default interpretation of any expression. Hiding has highest precedence with prefix next, followed by cooperation. Choice has the lowest precedence. Brackets may be used to force an alternative parsing or simply to clarify meaning.

A. Operational semantics and the underlying CTMC

The model components capture the structure of the system in terms of its *static* components. The dynamic behaviour of the system is represented by the evolution of these components, either individually or in cooperation. The form of this evolution is governed by a set of formal rules which give an operational semantics of PEPA terms. The semantic rules, in the structured operational style of Plotkin, are shown in Fig. 1; the interested reader is referred to [1] for full 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. For one example, the two rules for choice show that the choice operator is symmetric and preserves the potential behaviours of its two operands. For another, the cooperation operator has a special case where the two cooperands do not synchronise on any activities. The notation for this case is $E \parallel F$. In this case the three rules would simplify to the two which are shown below.

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

These rules capture the intuitive understanding that two components which do not synchronise on any activities cannot influence each other's computational state. In the case of components which do synchronise the rate of the resulting activity will reflect the capacity of each component to carry out activities of that type. For a component E and action type α, this is termed the *apparent rate* of α in E, denoted $r_\alpha(E)$. It is the sum of the rates of the α type activities enabled in E. The exact mechanism used to determine the rate of the shared activity will be explained shortly.

Prefix

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

Choice

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

Cooperation ($\alpha \notin L$)

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

Cooperation ($\alpha \in L$)

$$\frac{E \xrightarrow{(\alpha,r_1)} E' \quad F \xrightarrow{(\alpha,r_2)} F'}{E \underset{L}{\bowtie} F \xrightarrow{(\alpha,R)} E' \underset{L}{\bowtie} F'} \qquad \begin{aligned} R &= \frac{r_1}{r_\alpha(E)} \frac{r_2}{r_\alpha(F)} r_m \\ r_m &= \min(r_\alpha(E), r_\alpha(F)) \end{aligned}$$

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

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

Fig. 1. Operational semantics of PEPA

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

$$((\mathit{Comp} \parallel \mathit{Comp}) \underset{\{\mathit{task}\}}{\bowtie} \mathit{Res}) \underset{\{\mathit{repair}\}}{\bowtie} \mathit{Repman}$$

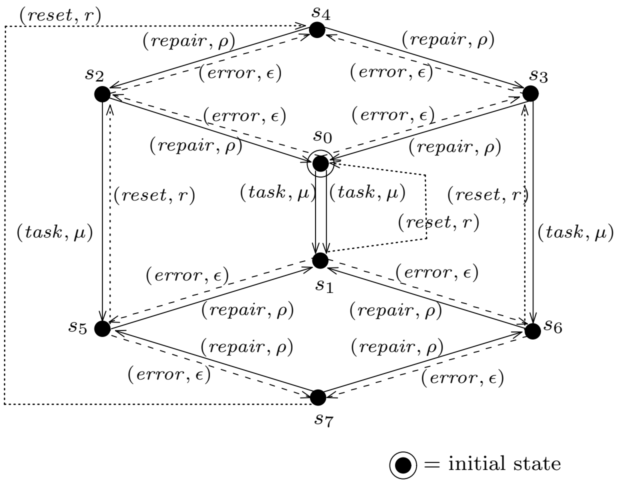


Fig. 2. DG for the Multi-Component model (without hiding)

State	Corresponding derivative
s_0	$((Comp \parallel Comp) \underset{\{task\}}{\bowtie} Res) \underset{\{repair\}}{\bowtie} Repman$
s_1	$((Comp \parallel Comp) \underset{\{task\}}{\bowtie} (reset, r).Res) \underset{\{repair\}}{\bowtie} Repman$
s_2	$(((repair, \rho).Comp \parallel Comp) \underset{\{task\}}{\bowtie} Res) \underset{\{repair\}}{\bowtie} Repman$
s_3	$((Comp \parallel (repair, \rho).Comp) \underset{\{task\}}{\bowtie} Res) \underset{\{repair\}}{\bowtie} Repman$
s_4	$(((repair, \rho).Comp \parallel (repair, \rho).Comp) \underset{\{task\}}{\bowtie} Res) \underset{\{repair\}}{\bowtie} Repman$
s_5	$(((repair, \rho).Comp \parallel Comp) \underset{\{task\}}{\bowtie} (reset, r).Res) \underset{\{repair\}}{\bowtie} Repman$
s_6	$((Comp \parallel (repair, \rho).Comp) \underset{\{task\}}{\bowtie} (reset, r).Res) \underset{\{repair\}}{\bowtie} Repman$
s_7	$(((repair, \rho).Comp \parallel (repair, \rho).Comp) \underset{\{task\}}{\bowtie} (reset, r).Res) \underset{\{repair\}}{\bowtie} Repman$

TABLE I
STATES OF THE DERIVATION GRAPH OF FIG. 2

is shown in Fig. 2, assuming the following definitions:

$$\begin{aligned} Comp &\stackrel{\text{def}}{=} (error, \epsilon).(repair, \rho).Comp \\ &+ \;(task, \mu).Comp \\ Res &\stackrel{\text{def}}{=} (task, \top).(reset, r).Res \\ Repman &\stackrel{\text{def}}{=} (repair, \top).Repman \end{aligned}$$

For simplicity, in the figure we have chosen to name the derivatives with short names $s_i, i = 0 \ldots 7$; the corresponding complete names are listed in Table 1. Note that there is a pair of arcs in the derivation graph between the initial state s_0 and its one-step derivative s_1. These capture the fact that there are two distinct derivations of the activity $(task, \mu)$ according to whether the first or second component completes the task in cooperation with the resource, even though the resulting derivative is the same in either case.

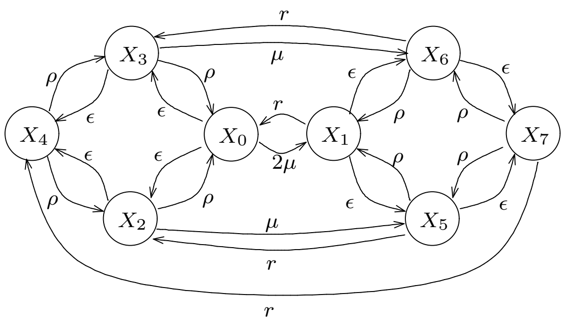


Fig. 3. CTMC underlying the Multi-Component model

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

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 fixed 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. This capacity is the apparent rate. The apparent rate of α in a cooperation $P \underset{\{\alpha\}}{\bowtie} Q$ will be the minimum of $r_\alpha(P)$ and $r_\alpha(Q)$. The rate of any particular shared activity will be the apparent rate of the shared activity weighted by the conditional probability of the contributing activities in the cooperating components. The interested reader is referred to [1] for more details.

The DG is the basis of the underlying 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 P' in the DG is the rate at which the system changes from behaving as component P to behaving as P'. It is denoted by $q(P, P')$ and is the sum of the activity rates labelling arcs connecting node P to node P'. For example, the state transition diagram for the CTMC underlying the simple component model is shown in Fig. 3. Note the arc labelled with rate 2μ between states X_0 and X_1, representing the derivatives $((Comp \parallel Comp) \underset{\{task\}}{\bowtie} Res) \underset{\{repair\}}{\bowtie} Repman$ and $((Comp \parallel Comp) \underset{\{task\}}{\bowtie} (reset, r).Res) \underset{\{repair\}}{\bowtie} Repman$ respectively.

In order for the CTMC to be *ergodic* its DG must be strongly connected. Some necessary conditions for ergodicity, at the syntactic level of a PEPA model, have been defined [1]. These syntactic conditions are imposed by the grammar introduced earlier.

B. Aggregation in PEPA via isomorphism

Equivalence relations, and notions of equivalence generally, play an important role in process algebras, and defining useful equivalence relations is an essential part of language development. For PEPA various equivalence relations have been defined. These include *isomorphism*, which captures the intuitive notion of equivalence between language terms based on isomorphic derivation graphs, and *strong equivalence*, a more sophisticated notion of equivalence based on *bisimulation*.

Any equivalence relation defined over the state space of a model will induce a partition on the state space. Aggregation is achieved by constructing such a partition and forming the corresponding *aggregated* process. In the aggregated process each partition of states in the original process forms one state. If the original state space is $\{X_0, X_1, \ldots, X_n\}$ then the aggregated state space is some $\{X_{[0]}, X_{[1]}, \ldots, X_{[N]}\}$, where $N \leq n$, ideally $N \ll n$. In general, when a CTMC is aggregated the resulting stochastic process will not have the Markov property. However if the partition can be shown to satisfy the so-called *lumpability* condition, the property is preserved and the aggregation is said to be *exact*.

When the model considered is derived from a process algebra such as PEPA it is possible to establish useful algebraic properties of the equivalence relation used. The most important of these is *congruence*. An equivalence relation is a congruence with respect to the operators of the language if substituting an equivalent component within a model expression gives rise to an equivalent model; e.g. if P is equivalent to P', then $P \underset{L}{\bowtie} Q$ is equivalent to $P' \underset{L}{\bowtie} Q$. When a congruence is used as the basis for aggregation in a compositional model, the aggregation may be carried out component by component, avoiding the construction of the complete state space because the aggregated component will be equivalent to the original. Nevertheless this approach is applied at the semantic level of the model and necessitates the expansion and subsequent partitioning of relevant state spaces. Moreover, the reduced model produced in this way may not be as compact as would be achieved by aggregating the complete model directly, making a further application of aggregation necessary, in this case to the model consisting of the aggregated components.

Both isomorphism and strong equivalence are congruence relations that can be used as the basis for exact aggregation of PEPA models, based on lumpability [4]. In either case the relation is used to partition the state space (possibly compositionally), and so the underlying CTMC, and each such equivalence class forms one state in the aggregated state space. In the algorithm which we are presenting here we use the isomorphism relation: the use of strong equivalence for the same purpose is discussed in Section V.

The use of the isomorphism relation may seem surprising since the more powerful bisimulation-style equivalence relations are one of the attractive features of process algebras and are often cited as one of the benefits of these formal languages. In contrast, isomorphism has received little attention in the literature. In part, this is because in classical process algebra the objective is to use an equivalence relation to determine when two agents or system descriptions exhibit the same behaviour. In stochastic process algebra greater emphasis is placed on using equivalence relations to partition the derivation graph of the model in order to produce an aggregation resulting in a smaller underlying Markov process. It has been shown that PEPA's strong equivalence relation is a powerful tool for aggregation in this style, always resulting in a lumpably equivalent Markov process [1]. However, we believe that in many instances isomorphism can also be useful for this purpose. Since it is a more discriminating notion of equivalence it may give a finer partition and thus less aggregation than strong equivalence. On the other hand, as we will show, it may be detected at the syntactic level of the system description without the recourse to the semantic level which is necessary to detect strong equivalence in general. Thus a reduced derivation graph is generated without the need to construct the original derivation graph.

In the following section we present the algorithm which exploits isomorphism, while in Section VI we discuss its relation to other work on automated aggregation.

III. Algorithm

The algorithm for computing the reduced derivation graph of a PEPA model begins by pre-processing a model which has been supplied by the modeller. The purpose of this pre-processing is to re-express the model in a more convenient form for the production of the aggregated derivation graph. The aggregated derivation graph has at its nodes, equivalence classes of PEPA terms, rather than single syntactic expressions. During the pre-processing step the PEPA syntax is systematically replaced and the model expression is converted into a *vector form*, which is then minimised and converted into its *canonical form*. Every distinct PEPA expression maps to a distinct vector form, but equivalent (isomorphic) expressions will have the same canonical representation.

Once this pre-processing is complete, the generation of the reduced derivation graph can begin. This process alternates between generating all of the one-step derivatives of the present state and compacting these in order to group together derivatives which have the same canonical representation.

The algorithm proceeds on the assumption that the model supplied is in *reduced named norm form*. In the *named form* representation each derivative of each sequential component is explicitly named. In the *norm form* the model is expressed as a single model equation which consists of cooperations of sequential components governed by hiding sets. In the *reduced form* all cooperation and hiding sets have been reduced by removing any redundant elements. If the supplied model is not in this form the necessary restructuring is carried out before the algorithm is applied. The functions to achieve this carry out routine checks on the validity of the model supplied and the modifications that they make are completely transparent to the modeller.

We now proceed to describe these steps in more detail.

A. Restructuring the model

During the application of the algorithm it is convenient to have intermediate derivatives in the model bound to identifiers. We generate these identifiers as we decompose the defining equations for each sequential component. For example, if the defining equation is

$$\mathit{Comp} \stackrel{\mathrm{def}}{=} (\mathit{error}, \epsilon).(\mathit{repair}, \rho).\mathit{Comp} \; + \; (\mathit{task}, \mu).\mathit{Comp}$$

we introduce a name for the intermediate derivative by replacing this single equation by the following pair of equations.

$$\begin{aligned} \mathit{Comp} &\stackrel{\mathrm{def}}{=} (\mathit{error}, \epsilon).\mathit{Comp}' \; + \; (\mathit{task}, \mu).\mathit{Comp} \\ \mathit{Comp}' &\stackrel{\mathrm{def}}{=} (\mathit{repair}, \rho).\mathit{Comp} \end{aligned}$$

Once this has been done for each sequential component the model is said to be in *named form*.

As described in Section II, a PEPA model consists of a collection of defining equations for sequential components and model components. One of the model components is distinguished by being named as the initial state of the model. The definition of this component may refer to other model components, defined by other equations. We wish to eliminate uses of model components from that definition, in order to reduce it to a normed form in which the only identifiers used are those of sequential components. We proceed by back-substituting the model component definitions into the defining equation of the distinguished component. For example, the pair of equations

$$\begin{aligned} \mathit{Config} &\stackrel{\mathrm{def}}{=} \mathit{System} \underset{\{\mathit{repair}\}}{\bowtie} \mathit{Repman} \\ \mathit{System} &\stackrel{\mathrm{def}}{=} \mathit{Comp} \underset{\{\mathit{task}\}}{\bowtie} \mathit{Res} \end{aligned}$$

will become

$$\mathit{Config} \stackrel{\mathrm{def}}{=} \mathit{Comp} \underset{\{\mathit{task}\}}{\bowtie} \mathit{Res} \underset{\{\mathit{repair}\}}{\bowtie} \mathit{Repman}$$

We continue this process until it converges to a definition of a *normed model equation* which consists only of cooperations of sequential components governed by hiding sets.

If the cooperation or hiding sets in a model definition contain unnecessary or redundant elements the equivalence classes formed by the algorithm may not be optimal. Thus we can, in some circumstances, improve the subsequent performance of the algorithm by removing redundant elements from these sets before the algorithm is applied. Furthermore, the presence of redundant elements in cooperation or hiding sets can be regarded as a potential error on the part of the modeller; consequently the modeller is warned of any reduction.

We have previously presented efficient algorithms for computing the sets of activities ($\mathcal{A}ct$) which are performed by PEPA model components [10] and we use these to reduce to the minimum the size of cooperation and hiding sets in the following way.

$$\begin{aligned} P \underset{L}{\bowtie} Q &\rightsquigarrow P \underset{L_0}{\bowtie} Q \quad \text{where } L_0 = L \cap (\mathcal{A}ct(P) \cup \mathcal{A}ct(Q)) \\ P/L &\rightsquigarrow P/L_0 \qquad \text{where } L_0 = L \cap \mathcal{A}ct(P) \end{aligned}$$

This reduction is applied systematically throughout the normed model equation. This operation is bounded in complexity by the size of the static representation of the input PEPA model and thus there is no hidden cost here of a traversal of the state space which is generated by the dynamic exploration of the model.

B. Pre-processing: vector form, minimisation, canonicalisation

The vector form of a model expression represents the model in the most suitable form for our aggregation algorithm because it is amenable to efficient calculation of its canonical form. Here we present the vector form as a vector of sequential components with decorated brackets denoting the scope of these sets. We use subscripted brackets to delimit a cooperation set and superscripted angle brackets to delimit hiding sets.

In the implementation these vectors are represented by linked lists which provide for efficient manipulation when forming canonical representatives. Re-ordering and rearrangement of the representations of components in the vector forms can then be achieved by safe, statically-checked pointer manipulation, thereby avoiding the overhead of the repeated copying of data values which would be incurred by the use of an array-based representation.

Definition 1 (Vector Form) *For a model expression, we define the vector form inductively over the structure of the expression: let M, N be expressions and C be a constant denoting a sequential component.*
1. $\mathsf{vf}\,(M \underset{L}{\bowtie} N) = (\mathsf{vf}\,M, \mathsf{vf}\,N)_L$
2. $\mathsf{vf}\,(M/L) = {}^{L}\langle \mathsf{vf}\,M \rangle$
3. $\mathsf{vf}\,(C) = C$

In the following we write $\mathbf{P}$ to denote a vector $(P_1, \ldots, P_n)$.

As with the normed model equation, the vector form representation contains within a single expression all of the information about the static structure of the model. It records the name of the current derivative of each of the sequential components in addition to the scope of the cooperation and hiding sets which are in force. The vector form alone is not sufficient to allow us to compute the derivation graph of the model: the defining equations for the sequential components are also needed.

Because it is generated directly from the full model equation the vector form may include some redundancies. Hence, we include a preprocessing step which is carried out to reduce the vector form generated by a straightforward translation of the model equation to the vector form which will be used for the remainder of the state space exploration. This step consists of generating the *minimal representation* of the vector form, which is minimal with respect to the number of brackets needed to record the scope of the cooperation and hiding sets. As we will see, reducing the number of brackets in the vector form may have significant impact on the aggregation which can be achieved. Thus we can perform the following simplifications:

Elimination of redundant cooperation brackets: this arises when we have a component such as $Q \underset{L}{\bowtie} (P \underset{L}{\bowtie} R)$. The vector form of this component would be $(Q, (P, R)_L)_L$. When contiguous brackets have the same decoration in this way the inner one can be eliminated. In this example this results in $(Q, P, R)_L$.

Elimination of redundant hiding brackets: this would arise whenever hiding brackets are contiguous regardless of their decoration. For example, if we had a component $(P/L)/K$ its vector form would be ${}^K\langle {}^L\langle P\rangle\rangle$. This would be reduced to ${}^{K\cup L}\langle P\rangle$.

From the minimal vector form we reduce the model representation to its *canonical form.* We can choose an arbitrary ordering on component terms—one suitable ordering is lexicographic ordering. We denote this ordering by $P \leq Q$ or $Q > P$. We denote the canonicalisation function by $\mathcal{C}$. We insert a component P into a vector $\mathbf{P}$ using $\mathcal{I}_P\,\mathbf{P}$. The definitions of these functions are shown in Definition 2. The definitions are not complex but we include them here for completeness and in order to prevent there appearing to be any hidden complexity in their definitions.

Definition 2 (Canonicalisation and insertion) *These are the definitions of the canonicalisation function $\mathcal{C}$ and the insertion function $\mathcal{I}$. There are three cases in the definition of each of these functions.*

1. $\mathcal{C}C = C$
2. $\mathcal{C}\,{}^L\langle P\rangle = {}^L\langle \mathcal{C}P\rangle$
3. $\mathcal{C}(P_1, \ldots, P_n)_L = \mathcal{I}_{\mathcal{C}P_1} \cdots \mathcal{I}_{\mathcal{C}P_n}\,()_L$

1. $\mathcal{I}_P\,()_L = (P)_L$
2. $\mathcal{I}_P\,(P_1, \ldots, P_n)_L = (P, P_1, \ldots, P_n)_L$ *if* $P \leq P_1$
3. $\mathcal{I}_P\,(P_1, \ldots, P_n)_L = \mathcal{I}_{P_1}\,\mathcal{I}_P\,(P_2, \ldots, P_n)_L$ *if* $P > P_1$

C. Generating the aggregated derivation graph

The previous pre-processing steps have been applied to the input PEPA model to facilitate the subsequent application of the aggregation algorithm. Before pre-processing the model was represented by a PEPA expression, which represented an individual (initial) state and contained all the information necessary for its dynamic evolution. After the pre-processing steps have been performed, the expression is reduced to canonical, minimal vector form, which retains only information about the state structure of the model and represents an equivalence class of states. Thus this canonical vector form is a reduced representation in two senses. Firstly, the information about the dynamic behaviour, cooperation sets and hiding sets, which is common to all states of the model, is factored out and stored separately. Secondly, each canonical vector form may in fact represent a number of equivalent model states which would have distinct vector forms.

Generating the reduced derivation graph now proceeds via the following two steps which are carried out alternately until the state space has been fully explored.

Derivation: Given the vector form the objective is to find all enabled activities and record them in a list, paired with the vector form of the corresponding derivative. This is done by recursing over the static structure of the current derivative. At the lowest level the sequential components are represented simply as a derivative name. At this point the defining equations are used to find the activity, or set of activities, which are enabled by the derivative. We can identify three cases:

- Individual activities which are not within the scope of a hiding operator are recorded directly with the resulting derivative.
- Individual activities which are within the scope of a hiding operator are recorded as τ actions with the appropriate rate together with the resulting derivative.
- Activities which are within the scope of a cooperation set are compared with the enabled activities of the other components within the cooperation. If there is no matching activity the individual activity is discarded; otherwise, as above, the activity is recorded together with the resulting vector form.

Reduction: Carrying out the derivation may have given rise to vector forms which are not canonical. Moreover several of the (activity, vector form) pairs may turn out to be identical once the vector form is put into canonical form. In this case the multiplicity is recorded and only one copy is kept.

These two steps have to be repeated until there are no elements left in the set of unexplored derivative classes.

In the remainder of this section we present these steps more formally, but first we introduce some notation for describing the formulation and manipulation of vectors and vector forms.

- Given a vector $\mathbf{P}$, we write $(P_i \in \mathbf{P} : \phi)$ to denote the sub-vector of those elements of $\mathbf{P}$ which satisfy the predicate ϕ. When the vector $\mathbf{P}$ is obvious from the context we shall omit it, writing $(P_i : \phi)$ as an abbreviation.
- We write $\mathbf{P}[P_i := P_i']$ to denote the vector obtained from $\mathbf{P}$ by substituting P_i' for P_i.
- When $\mathbf{S}$ is a sub-vector $(S_1, \ldots, S_n)$, and $\mathbf{S}'$ similarly, we write $\mathbf{P}[\mathbf{S} := \mathbf{S}']$ as an abbreviation for $\mathbf{P}[S_1 := S_1'] \cdots [S_n := S_n']$. Note that we only use vector substitution between vectors with the same number of elements.

The rules which govern the derivation step of the algorithm are shown in Fig. 4. The rule for constant formally states that at the lowest level defining equations are used to find the activity or activities which can be inferred from a derivative name. The two rules for hiding correspond to the first two cases identified above. The most complex rules are those for cooperation, the third case above. We examine these in more detail.

The first rule states the condition under which a number of identical activities, (α, r), give rise to derivatives which have identical canonical forms. For this to be the case the activity (α, r) must be enabled by one or more component P_i of $\mathbf{P}$. Moreover, for each such possible activity, the vector form of the resulting derivative is always the same when canonicalised. Formally,

$$P_i \xrightarrow{(\alpha,r)} P_i' \quad \wedge \quad \mathcal{C}P_i' = \mathcal{C}\sigma$$

where σ is an arbitrary element of the vector $\mathbf{S'}$, say its first element S_1'. Note that the equation $\mathcal{C}P_i' = \mathcal{C}S_1'$ does not imply that P_i' and S_1' are equal, only that they are in the same equivalence class because they have equal canonical forms. The vector $\mathbf{S'}$ is defined as the sub-vector consisting of those derivatives which may potentially change via an (α, r) activity.

$$\mathbf{S'} = (P_i' : P_i \xrightarrow{(\alpha,r)} P_i')$$

Having now formed a vector $\mathbf{S}$ satisfying these conditions for the activity (α, r) we can compute the rate at which the component performs this activity and evolves to the canonical representative of the derivatives as $|\mathbf{S}| \cdot r$, since the total rate into the equivalence class will be the sum of the rates of the individual activities which may make the move.

In the case where only one of the elements of the vector performs an activity α the complication due to the consideration of multiplicities does not arise and the rule simplifies to be equivalent to the following.

$$\frac{P_i \xrightarrow{(\alpha,r)} P_i'}{\mathbf{P}_L \xrightarrow{(\alpha,r)} \mathcal{C}(\mathbf{P}_L[P_i := P_i'])} \ (\alpha \notin L)$$

The complexity in the second rule for cooperation is due to the need to calculate the rate at which the sub-vector of components in cooperation performs the activity. Here also there is a simpler case, where the vector is of size two. This special case of the rule affords easier comparison with the operational semantics of PEPA, as presented in Fig. 1.

$$\frac{P_1 \xrightarrow{(\alpha,r_1)} P_1' \quad P_2 \xrightarrow{(\alpha,r_2)} P_2'}{(P_1, P_2)_L \xrightarrow{(\alpha,R)} \mathcal{C}(P_1', P_2')_L} \ (\alpha \in L)$$

The rate R of the activity which is performed in cooperation is computed from the individual rates r_1 and r_2 as in the corresponding cooperation rule in Fig. 1.

Constant

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

Hiding

$$\frac{\mathbf{P} \xrightarrow{(\alpha,r)} \mathbf{P'}}{{}^{L}\langle \mathbf{P} \rangle \xrightarrow{(\alpha,r)} {}^{L}\langle \mathbf{P'} \rangle} \ (\alpha \notin L) \qquad \frac{\mathbf{P} \xrightarrow{(\alpha,r)} \mathbf{P'}}{{}^{L}\langle \mathbf{P} \rangle \xrightarrow{(\tau,r)} {}^{L}\langle \mathbf{P'} \rangle} \ (\alpha \in L)$$

Cooperation $(\alpha \notin L)$

$$\frac{\begin{array}{l}\mathbf{S} \ = (P_i : P_i \xrightarrow{(\alpha,r)} P_i' \wedge \mathcal{C}P_i' = \mathcal{C}S_1') \\ \mathbf{S'} = (P_i' : P_i \xrightarrow{(\alpha,r)} P_i')\end{array}}{\mathbf{P}_L \xrightarrow{(\alpha,|\mathbf{S}|\cdot r)} \mathcal{C}(\mathbf{P}_L[\mathbf{S} := \mathbf{S'}])}$$

Cooperation $(\alpha \in L)$

$$\frac{\begin{array}{l}\mathbf{S} \ = (P_i : P_i \xrightarrow{(\alpha,r_i)} P_i') \\ \mathbf{S'} = (P_i' : P_i \xrightarrow{(\alpha,r_i)} P_i')\end{array}}{\mathbf{P}_L \xrightarrow{(\alpha,R)} \mathcal{C}(\mathbf{P}_L[\mathbf{S} := \mathbf{S'}])} \qquad \begin{array}{l} R \ = \ \min\{r_\alpha(P_i)\} r_p \\ r_p \ = \displaystyle\prod_{i=1}^{|\mathbf{S}|} \frac{r_i}{r_\alpha(P_i)} \end{array}$$

Fig. 4. Operational semantics of vector form

D. Implementation

The state space reduction algorithm has been added to the PEPA Workbench [5], the modelling package which implements the PEPA language and provides a variety of solution and analysis facilities for PEPA models.

The algorithm is presented in pseudo-code form in Fig. 5 and Fig. 6. The driving force of the algorithm is provided by the procedure *vfderive* which, given a derivative of the model, finds its enabled transitions using the function *cderiv*, and calls itself on the resulting derivative. The function *cderiv* carries out the canonicalisation of the one-step derivatives which it has produced using the function *derivatives*. This function has different cases depending on the structure of the vector form being handled, each reflecting the appropriate rule(s) in the semantics. For example, in the case of a choice the list of possible derivatives consists of the list of derivatives of the second component of the choice appended to the list of derivatives of the first. The derivatives of a vector of cooperating components are computed by using the function *cooperations* to derive transitions and the function *disallow* to enforce that activities of types in a cooperation set are not carried out without a partner. We make use of a function *lookup* to retrieve the definitions of component identifiers from the environment.

```
derivatives(P) =
  switch P is
    case (P_1)_L                          (* unary cooperation *)
      d ← derivatives(P_1);
      return update(d, proc (a, P') is (a, (P')_L));
    end case;

    case (P_1, ..., P_n)_L                (* n-ary cooperation *)
      d_1 ← derivatives((P_1)_L);
      d_2 ← derivatives((P_2, ..., P_n)_L);
      f_1 ← disallow(d_1, L);
      f_2 ← disallow(d_2, L);
      h_1 ← update(f_1, proc (a, (P')_L) is
                        (a, (P', P_2, ..., P_n)_L));
      h_2 ← update(f_2, proc (a, (P'_2, ..., P'_n)_L) is
                        (a, (P_1, P'_2, ..., P'_n)_L));
      h_3 ← cooperations(d_1, d_2, L);
      return append(h_1, h_2, h_3);
    end case;

    case ^L⟨P⟩                            (* hiding *)
      d ← derivatives(P);
      return filter(d, L);
    end case;

    case (α, r).P                         (* prefix *)
      return singleton((α, r), P);
    end case;

    case P + Q                            (* choice *)
      d_1 ← derivatives(P);
      d_2 ← derivatives(Q);
      return append(d_1, d_2);
    end case;

    case const C                          (* constant *)
      return derivatives(lookup(C));
    end case;
  end switch;
```

```
cooperations(d_1, d_2, L) =
  if d_1 or d_2 is empty
  then return ∅
  else
    begin
      remove ((a_1, r_1), P1_{L_1}) from d_1;
      remove ((a_2, r_2), P2_{L_2}) from d_2;
      if a_1 = a_2 and a_1 in L
      then c_1 ← singleton((a_1, min(r_1, r_2)),
                           (append(P1, P2))_L)
      else c_1 ← ∅;
      c_2 ← cooperations(((a_1, r_1), P1_{L_1}), d_2, L);
      c_3 ← cooperations(d_1, ((a_2, r_2), P2_{L_2}), L);
      c_4 ← cooperations(d_1, d_2, L);
      return append(c_1, c_2, c_3, c_4);
    end;
```

Fig. 5. Pseudo-code for the algorithm (i)

```
vfderive(P) =
  if not marked (P)
  then
    begin
      mark P;
      for each ((α, r), n, P') in cderiv(P)
      do
        output transition (P --(α, n·r)--> P');
        vfderive(P');
      end for;
    end;
```

```
cderiv(P) =
  d ← derivatives(P);
  d' ← update(d, proc (a, P) is (a, 𝒞P));
  r ← ∅;
  while d' is not empty do
    choose (a, P) from d' ;
    if (a, n, P) in r for some n
    then replace (a, n, P) by (a, n + 1, P) in r
    else add (a, 1, P) to r;
    remove (a, P) from d';
  end while;
  return r;
```

```
update(d, proc P) =
  r ← ∅;
  while d is not empty do
    remove x from d;
    add P(x) to r;
  end while;
  return r;
```

```
filter(d, L) =
  r ← ∅;
  while d is not empty do
    remove ((α, r), P) from d;
    if α in L
    then add ((τ, r), P) to r
    else add ((α, r), P) to r
  end while;
  return r;
```

```
disallow(d, L) =
  r ← ∅;
  while d is not empty do
    remove ((α, r), P) from d;
    if α not in L
    then add ((α, r), P) to r
  end while;
  return r;
```

Fig. 6. Pseudo-code for the algorithm (ii)

Finally, the function *update* takes a set of elements and a procedure and returns a set in which each element has been modified by the procedure.

The modification to the PEPA Workbench required the alteration of the data structure which is used to represent PEPA models as an abstract syntax tree within the Workbench. The representation of cooperations between pairs of components was generalised to extend to lists of components. If the PEPA model which is submitted for processing does not contain any structure which can be exploited by the state space reduction algorithm then this change is invisible to any user of the Workbench. However, if the PEPA model does contain either repeated components or other structure which can be exploited then the benefits become apparent to the user of the Workbench in terms of reduced time to generate the CTMC representation of the model and in terms of the matrix of smaller dimension required for its storage, once the model gets above a certain size (see Table III).

IV. Example

In this section we show how the algorithm works on an example. We consider a multiprocessor system with a shared memory, we derive the corresponding PEPA model, and then the underlying derivation graphs, both ordinary and aggregated. Some alternatives to our approach are discussed in Section V, introduced by means of small variants of the same example.

A. Multiprocessor system

Consider a multiprocessor system with a shared memory. Processes running on this system have to compete for access to the common memory: to gain access and to use the common memory they need also to acquire the system bus which is released when access to the common memory is terminated; for simplicity the bus will not be explicitly represented in the following. Processes are mapped onto processors. The processors are not explicitly represented but they determine the rate of activities in the associated processes, i.e. all processes have the same functional behaviour, but actions progress at different speeds depending on the processor on which they are running, and the number of processes present on the processor. It is the modeller's responsibility to select rates appropriately.

A protocol which is not completely fair, but simply prevents one processor from monopolising the memory, might impose that after each access of a processor to the memory, some other processor must gain access before the first can access again. A process running on the ith processor is represented as P_i:

$$P_i \stackrel{\text{def}}{=} (think, \lambda_i).(get_i, g).(use, \mu_i).(rel, r).P_i$$

In this case, in order to impose the protocol, the memory is modelled as remembering which processor had access last. Access for this processor is disabled.

$$Mem_i \stackrel{\text{def}}{=} \sum_{\substack{j=1 \\ j\neq i}}^{N} (get_j, \top).(use, \top).(rel, \top).Mem_j$$

If there are n_i processes running on the ith processor the system is modelled by the following expression

$$Sys \stackrel{\text{def}}{=} (\underbrace{P_1 \parallel \cdots \parallel P_1}_{n_1} \parallel \cdots \parallel \underbrace{P_N \parallel \cdots \parallel P_N}_{n_N}) \underset{\{get_i, use, rel\}}{\bowtie} Mem_k$$

Note that in the cooperation set of this model expression, and throughout the remainder of the paper, we write get_i as a shorthand for $get_i \mid 1 \leq i \leq N$. We assume that the starting state of the system excludes access of an arbitrary processor, number k. The vector form of the model *Sys*, derived applying the equations of Definition 1, has the following form:

$$((P_1, \ldots, P_1, \ldots, P_N, \ldots, P_N)_{\emptyset}, Mem_k)_{\{get_i, use, rel\}}$$

We now show an example derivation of the state space of *Sys*, both ordinary and aggregated. For simplicity we consider a smaller system Sys' in which we have only two processors and only two replicas of the same process running on each processor. The simplified system is thus specified as

$$Sys' \stackrel{\text{def}}{=} (P_1 \parallel P_1 \parallel P_2 \parallel P_2) \underset{\{get_1, get_2, use, rel\}}{\bowtie} Mem_1$$

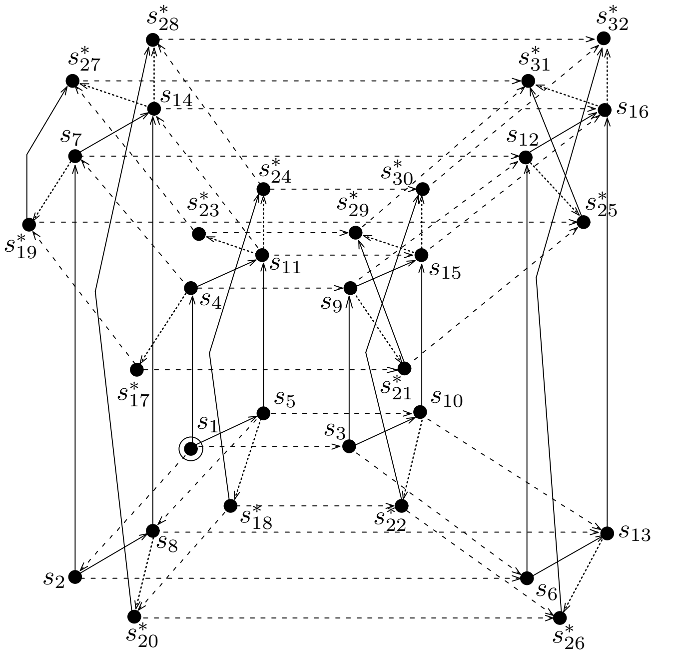
We can expand the derivatives of the processes P_i, for $i = 1, 2$, and of the memory Mem_1 as follows:

$$\begin{aligned}
P_i &\stackrel{\text{def}}{=} (think, \lambda_i).P_i' \\
P_i' &\stackrel{\text{def}}{=} (get_i, g).P_i'' \\
P_i'' &\stackrel{\text{def}}{=} (use, \mu_i).P_i''' \\
P_i''' &\stackrel{\text{def}}{=} (rel, r).P_i
\end{aligned}$$

$$\begin{aligned}
Mem_1 &\stackrel{\text{def}}{=} (get_2, \top).Mem_1' \\
Mem_1' &\stackrel{\text{def}}{=} (use, \top).Mem_1'' \\
Mem_1'' &\stackrel{\text{def}}{=} (rel, \top).Mem_2 \\
Mem_2 &\stackrel{\text{def}}{=} (get_1, \top).Mem_2' \\
Mem_2' &\stackrel{\text{def}}{=} (use, \top).Mem_2'' \\
Mem_2'' &\stackrel{\text{def}}{=} (rel, \top).Mem_1
\end{aligned}$$

Complete derivation graph

The complete derivation graph of Sys', computed using the PEPA Workbench [5] with the aggregation algorithm switched off, has 96 states and 256 transitions. A portion of this graph is shown in Fig. 7. To make the drawing easier to understand we have chosen to name the derivatives with short names s_i or s_i^*, depending on whether the state has been completely expanded (s_i) or not (s_i^*) (i.e. whether all its one-step derivatives are also represented).



Initial state $s_1 : ((P_1, P_1, P_2, P_2)_\emptyset, Mem_1)_L$

$(think, \lambda_1)$ $(think, \lambda_2)$ (get_2, g)

Fig. 7. Ordinary derivation graph of Sys'

The vector forms corresponding to the derivatives are listed in Table II: each row contains the name of the state and the corresponding vector form. Moreover, it contains information on whether the vector form is canonical or not, and the name of the state which represents the corresponding canonical vector form.

Aggregated derivation graph

The aggregated derivation graph, computed using the PEPA Workbench with the aggregation algorithm switched on, has 42 states and 88 transitions. A portion of this graph is shown in Fig. 8 and can be compared with the one of Fig. 7.

The PEPA model Sys' has been constructed according to the algorithm: the sequential components defining the processes and the memory are composed by means of the cooperation operator to obtain the model equation. All the derivatives have been explicitly named and we can use the model equation to generate the vector form of the model which does not have redundant brackets, and therefore no elimination is required.

At this point the aggregated state space can be obtained by considering canonical vector forms only, as shown in the graph of Fig. 8 in which only a subset of the states of Table II, those corresponding to the canonical vector forms, is explicitly outlined. The names of the nodes are again s_i or s_i^* and the integer numbers in round brackets close to them specify the number of equivalent states they represent. These numbers can be computed by considering the number of replicas of the same process in the model equation and the numbers of equal derivatives in each vector form.

State/rep	Vector Form	Canonical?
s_1/s_1	$((P_1, P_1, P_2, P_2)_\emptyset, Mem_1)_L$	Yes
s_2/s_3	$((P_1', P_1, P_2, P_2)_\emptyset, Mem_1)_L$	No
s_3/s_3	$((P_1, P_1', P_2, P_2)_\emptyset, Mem_1)_L$	Yes
s_4/s_5	$((P_1, P_1, P_2', P_2)_\emptyset, Mem_1)_L$	No
s_5/s_5	$((P_1, P_1, P_2, P_2')_\emptyset, Mem_1)_L$	Yes
s_6/s_6	$((P_1', P_1', P_2, P_2)_\emptyset, Mem_1)_L$	Yes
s_7/s_{10}	$((P_1', P_1, P_2', P_2)_\emptyset, Mem_1)_L$	No
s_8/s_{10}	$((P_1', P_1, P_2, P_2')_\emptyset, Mem_1)_L$	No
s_9/s_{10}	$((P_1, P_1', P_2', P_2)_\emptyset, Mem_1)_L$	No
s_{10}/s_{10}	$((P_1, P_1', P_2, P_2')_\emptyset, Mem_1)_L$	Yes
s_{11}/s_{11}	$((P_1, P_1, P_2', P_2')_\emptyset, Mem_1)_L$	Yes
s_{12}/s_{13}	$((P_1', P_1', P_2', P_2)_\emptyset, Mem_1)_L$	No
s_{13}/s_{13}	$((P_1', P_1', P_2, P_2')_\emptyset, Mem_1)_L$	Yes
s_{14}/s_{15}	$((P_1', P_1, P_2', P_2')_\emptyset, Mem_1)_L$	No
s_{15}/s_{15}	$((P_1, P_1', P_2', P_2')_\emptyset, Mem_1)_L$	Yes
s_{16}/s_{16}	$((P_1', P_1', P_2', P_2')_\emptyset, Mem_1)_L$	Yes
s_{17}^*/s_{18}^*	$((P_1, P_1, P_2'', P_2)_\emptyset, Mem_1')_L$	No
s_{18}^*/s_{18}^*	$((P_1, P_1, P_2, P_2'')_\emptyset, Mem_1')_L$	Yes
s_{19}^*/s_{22}^*	$((P_1', P_1, P_2'', P_2)_\emptyset, Mem_1')_L$	No
s_{20}^*/s_{22}^*	$((P_1', P_1, P_2, P_2'')_\emptyset, Mem_1')_L$	No
s_{21}^*/s_{22}^*	$((P_1, P_1', P_2'', P_2)_\emptyset, Mem_1')_L$	No
s_{22}^*/s_{22}^*	$((P_1, P_1', P_2, P_2'')_\emptyset, Mem_1')_L$	Yes
s_{23}^*/s_{24}^*	$((P_1, P_1, P_2'', P_2')_\emptyset, Mem_1')_L$	No
s_{24}^*/s_{24}^*	$((P_1, P_1, P_2', P_2'')_\emptyset, Mem_1')_L$	Yes
s_{25}^*/s_{26}^*	$((P_1', P_1', P_2'', P_2)_\emptyset, Mem_1')_L$	No
s_{26}^*/s_{26}^*	$((P_1', P_1', P_2, P_2'')_\emptyset, Mem_1')_L$	Yes
s_{27}^*/s_{30}^*	$((P_1', P_1, P_2'', P_2')_\emptyset, Mem_1')_L$	No
s_{28}^*/s_{30}^*	$((P_1', P_1, P_2', P_2'')_\emptyset, Mem_1')_L$	No
s_{29}^*/s_{30}^*	$((P_1, P_1', P_2'', P_2')_\emptyset, Mem_1')_L$	No
s_{30}^*/s_{30}^*	$((P_1, P_1', P_2', P_2'')_\emptyset, Mem_1')_L$	Yes
s_{31}^*/s_{32}^*	$((P_1', P_1', P_2'', P_2')_\emptyset, Mem_1')_L$	No
s_{32}^*/s_{32}^*	$((P_1', P_1', P_2', P_2'')_\emptyset, Mem_1')_L$	Yes

TABLE II
States, representatives and vector forms for $L = \{get_1, get_2, use, rel\}$

As an example, let us consider the state s_{10} which corresponds to the vector form $((P_1, P_1', P_2, P_2')_\emptyset, Mem_1)_L$. This state represents four equivalent derivatives. This number can be computed by dividing the product of the factorial of the numbers of the repeated instances of components by the product of the factorial of the numbers of identical derivatives in the vector form.

$$n = \frac{2! \cdot 2!}{1! \cdot 1! \cdot 1! \cdot 1! \cdot 1!} = 4$$

More generally the formula could be expressed as follows

$$n = \frac{n_1! \cdots n_N!}{(n_{1,1}! \cdots n_{1,k_1}!) \cdots (n_{N,1}! \cdots n_{N,k_N}!)}$$

where n_i, for $i = 1, 2, \ldots, N$, is the number of processes running on the same processor and $n_{i,j}$ are the numbers of equal derivatives of P_i, such that $\sum_{j=1}^{k_j} n_{i,j} = n_i$.

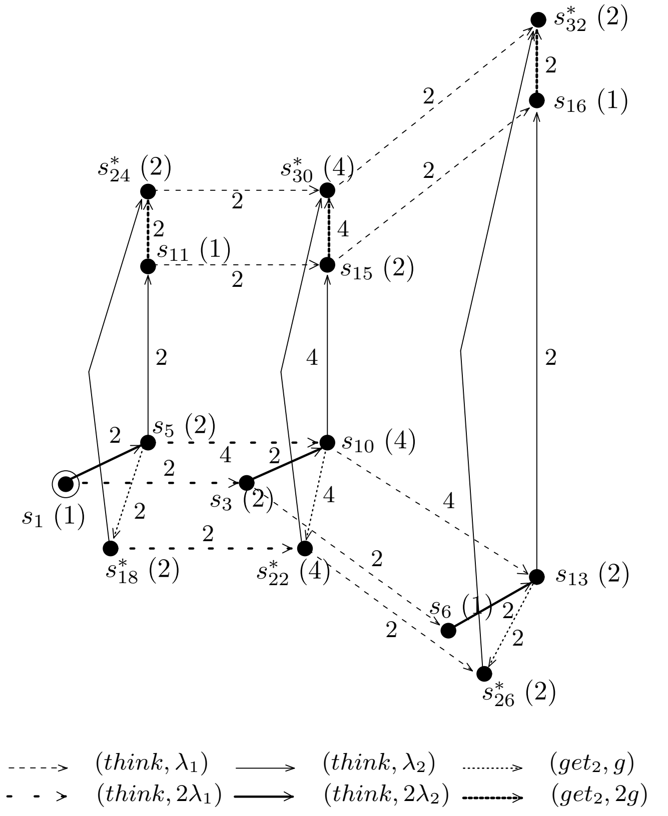


Fig. 8. Aggregated derivation graph of Sys'

The multiplicities of the arcs are also represented and indicate the number of arcs which have been folded together. The fact that a single arc represents one or more activities of the same type is reflected in the rate of the action that labels the arc itself. For instance, the model evolves from the state s_1 to the states s_3 by executing an action *think* with a rate $2\lambda_1$ because whenever the model is in state s_1, i.e. $(P_1 \parallel P_1 \parallel P_2 \parallel P_2) \underset{\{get_1, get_2, rel\}}{\bowtie} Mem_1$, two activities $(think, \lambda_1)$ are concurrently enabled.

Notice that the aggregation we obtain corresponds to finding permutations of the same components within brackets. This form of aggregation is pictorially represented in Fig. 8 by flattening equivalent nodes of the derivation graph of Fig. 7 onto the same plane.

B. Timings

We ran different configurations of the multiprocessor system on a Pentium III machine with clock frequency of 500 MHz and 128 MBytes of memory. The times recorded in Table III take into account both CPU time and the time necessary for file I/O.

If there is a single component P_i running on each processor, no aggregation is possible and the execution times of the basic and the modified Workbench are almost the same. As soon as we add replicas of the same process, the state space aggregation becomes apparent as well as the reduction in the execution times, particularly when the size of the model grows.

2 Processors	**Derivation graph**		
Processes	States	Trans.	Time (sec.)
1 P_1, 1 P_2	16	24	0.011
2 P_1, 1 P_2	40	84	0.020
2 P_1, 2 P_2	96	256	0.047
3 P_1, 2 P_2	224	720	0.132
3 P_1, 3 P_2	512	1920	0.371
4 P_1, 3 P_2	1152	4928	1.048
4 P_1, 4 P_2	2560	12288	2.887
2 Processors	**Aggregated derivation graph**		
Processes	States	Trans.	Time (sec.)
1 P_1, 1 P_2	16	24	0.010
2 P_1, 1 P_2	26	47	0.015
2 P_1, 2 P_2	42	88	0.028
3 P_1, 2 P_2	58	129	0.045
3 P_1, 3 P_2	80	188	0.075
4 P_1, 3 P_2	102	247	0.110
4 P_1, 4 P_2	130	324	0.173

3 Processors	**Derivation graph**		
Processes	States	Trans.	Time (sec.)
1 P_1, 1 P_2, 1 P_3	72	156	0.037
2 P_1, 1 P_2, 1 P_3	176	480	0.093
2 P_1, 2 P_2, 1 P_3	416	1360	0.262
2 P_1, 2 P_2, 2 P_3	960	3648	0.754
3 P_1, 2 P_2, 2 P_3	2176	9408	2.144
3 P_1, 3 P_2, 2 P_3	4864	23552	5.888
3 P_1, 3 P_2, 3 P_3	10752	57600	15.932
4 P_1, 3 P_2, 3 P_3	23552	138240	42.580
4 P_1, 4 P_2, 3 P_3	51200	326656	111.529
4 P_1, 4 P_2, 4 P_3	110592	761856	285.204
3 Processors	**Aggregated derivation graph**		
Processes	States	Trans.	Time (sec.)
1 P_1, 1 P_2, 1 P_3	72	156	0.041
2 P_1, 1 P_2, 1 P_3	116	284	0.079
2 P_1, 2 P_2, 1 P_3	186	505	0.159
2 P_1, 2 P_2, 2 P_3	297	882	0.320
3 P_1, 2 P_2, 2 P_3	408	1259	0.525
3 P_1, 3 P_2, 2 P_3	560	1792	0.839
3 P_1, 3 P_2, 3 P_3	768	2544	1.391
4 P_1, 3 P_2, 3 P_3	976	3296	2.036
4 P_1, 4 P_2, 3 P_3	1240	4267	3.098
4 P_1, 4 P_2, 4 P_3	1575	5520	4.430

TABLE III

EXECUTION TIMES OF THE BASIC AND MODIFIED WORKBENCH

V. ALTERNATIVE AGGREGATIONS

In this section we illustrate some cases in which our algorithm, or indeed any syntactic approach, cannot achieve the optimal theoretical partitioning. In particular we show how greater aggregation could be achieved in some circumstances if strong equivalence was used to generate partitions instead of isomorphism. Note, however, that these cases rely on quite strong conditions on apparently unrelated activity rates. It is not clear that such conditions occur with sufficient frequency in real models to justify the additional complexity needed to implement an approach based on strong equivalence.

The strong equivalence relation is a more sophisticated notion of equivalence, in the bisimulation style, based on observed behaviour. In general, in a process algebra, two terms are considered bisimilar if their externally observed behaviour appears to be the same. Strong equivalence assumes that both the action type and the apparent rate of each activity is observable. Informally, two PEPA components are strongly equivalent if their *total conditional transition rates* to strongly equivalent terms are the same for all action types.

The *conditional transition rate* from P to P' via an action type α is denoted by $q(P, P', \alpha)$. This is the sum of the activity rates labelling arcs connecting the corresponding nodes in the DG which are also labelled by the action type α. The conditional transition rate is thus the rate at which a system behaving as component P evolves to behaving as component P' as the result of completing an activity of type α. If we consider a *set* of possible derivatives $\mathcal{S}$, the *total conditional transition rate* from P to $\mathcal{S}$, denoted $q[P, \mathcal{S}, \alpha]$, is equal to

$$q[P, \mathcal{S}, \alpha] = \sum_{P' \in \mathcal{S}} q(P, P', \alpha)$$

The definition can thus be formally stated as follows.

Definition 3 *Let $\mathcal{T}$ denote the set of all language terms, or derivatives. An equivalence relation over derivatives, $\mathcal{R} \subseteq \mathcal{T} \times \mathcal{T}$, is a* strong equivalence *if whenever $(P, Q) \in \mathcal{R}$ then for all $\alpha \in \mathcal{A}$ and for all $\mathcal{S} \in \mathcal{T}/\mathcal{R}$,*

$$q[P, \mathcal{S}, \alpha] \ = \ q[Q, \mathcal{S}, \alpha]$$

We say that P and Q are strongly equivalent*, denoted by $P \cong Q$ if $(P, Q) \in \mathcal{R}$ for some strong equivalence $\mathcal{R}$, i.e.*

$$\cong \ = \bigcup \{\mathcal{R} \mid \mathcal{R} \text{ is a strong equivalence } \}$$

Two of the following examples demonstrate the use of strong equivalence for aggregation. However, in the first example we show how the abstraction operator may be used at a higher syntactic level in the model and introduce symmetries between components which appear quite distinct in their defining equations. These symmetries rely on the context in which the components are placed, something not currently captured by our algorithm.

In [11], Ribaudo distinguishes two form of aggregation which can be found using strong equivalence. *Horizontal aggregation* arises from the interleaving of the activities of similarly behaved components. This aggregation takes advantage of repeated instances of the same pattern of behaviour within the overall model structure. The aggregation found using our algorithm may be termed a horizontal aggregation. In contrast, *vertical aggregation* arises when there are repeated patterns of behaviour within a single component. In the second example presented below, a variant of the multiprocessor model is considered in which a horizontal aggregation can be found using strong equivalence although isomorphism would regard the components as distinct. Finally we give an example where a vertical aggregation is possible with strong equivalence but not with isomorphism, and consequently not with our syntactic approach.

A. Aggregation via abstraction

The facility to hide or abstract action types within a PEPA model is designed to give the modeller the freedom to construct components in detail to ensure that their behaviour is accurately represented but to subsequently restrict the visible action types to only those relevant to the current modelling study. For example, in the model of the multiprocessor presented in the previous section, the modeller may choose to hide all the get_i actions. In terms of capturing the correct behaviour of the protocol it was important that these action types were distinguished; but in terms of the complete model they may all be regarded as internal τ actions.

Hiding all of these activities introduces strong symmetries into the model in terms of its functional behaviour. If, moreover, we find that the processes which are running on different processors share the same timing characteristics, i.e. $\lambda_i = \lambda_j$ and $\mu_i = \mu_j$ for all $i \neq j$, then the symmetries are apparent in all aspects of the model's behaviour. Only one process can access the memory at any time—and for the subsequent memory access its host processor is excluded—but the processes on all other processors behave equivalently. This means that we need only consider two classes of processes, those excluded and those eligible for access, regardless of their placement on processors. Once the get_i activities are all hidden it is no longer possible to identify from these processes which type of process is operating.

For example, consider the multiprocessor with three processors, and two processes running on the first, one on the second and two on the third. Then if we regard the system immediately after the process P_2 has completed an access to the memory and when one other process is waiting for access, the behaviour of the system is isomorphic regardless of whether the waiting process is on processor 1 or processor 3, i.e. all the following states are isomorphic:

$$\begin{aligned} & ((P_1' \parallel P_1 \parallel P_2 \parallel P_3 \parallel P_3) \underset{L}{\bowtie} Mem_2)/\{get_i\} \\ = \ & ((P_1 \parallel P_1' \parallel P_2 \parallel P_3 \parallel P_3) \underset{L}{\bowtie} Mem_2)/\{get_i\} \\ = \ & ((P_1 \parallel P_1 \parallel P_2 \parallel P_3' \parallel P_3) \underset{L}{\bowtie} Mem_2)/\{get_i\} \\ = \ & ((P_1 \parallel P_1 \parallel P_2 \parallel P_3 \parallel P_3') \underset{L}{\bowtie} Mem_2)/\{get_i\} \end{aligned}$$

Although these states are equivalent by isomorphism our algorithm would not place them within a single partition but into two: one consisting of the first two states and one consisting of the second pair. This is because the processes operating on different processors have distinct names and distinct actions get_i—this is necessary to ensure the correct functioning of the protocol—and the syntactic form of minimisation that we use cannot recognise that in some contexts P_1 and P_3 will behave equivalently.

This could be regarded as a penalty for the richness of the language. For example, the analogous situation does not arise in Petri net-based models because there is no notion of abstraction or hiding.

B. Horizontal aggregation via strong equivalence

Isomorphism is a strict structural equivalence: there must be a one-to-one relationship between both derivatives and activities. The observation-based strong equivalence is not so strict. Although corresponding derivatives must be capable of the same action types at the same apparent rates, how these are implemented as activities in the derivatives may differ as the following example demonstrates.

Suppose that on processor 1 two different types of process may be running. The first is identical to the process P_1 discussed in Section IV-A. The second has a similar pattern of behaviour but has two alternative local computations between accesses to the common memory. The process $\bar{P}_1$ is represented below.

$$\begin{array}{lcl} P_1 & \stackrel{\mathrm{def}}{=} & (\mathit{think}, \lambda_1).P_1' \\ P_1' & \stackrel{\mathrm{def}}{=} & (\mathit{get}_1, g).P_1'' \\ P_1'' & \stackrel{\mathrm{def}}{=} & (\mathit{use}, \mu_1).P_1''' \\ P_1''' & \stackrel{\mathrm{def}}{=} & (\mathit{rel}, r).P_1 \end{array} \qquad \begin{array}{lcl} \bar{P}_1 & \stackrel{\mathrm{def}}{=} & (\mathit{think}, \lambda_{11}).\bar{P}_1' \\ & + & (\mathit{think}, \lambda_{12}).\bar{P}_1' \\ \bar{P}_1' & \stackrel{\mathrm{def}}{=} & (\mathit{get}_1, g).\bar{P}_1'' \\ \bar{P}_1'' & \stackrel{\mathrm{def}}{=} & (\mathit{use}, \mu_1).\bar{P}_1''' \\ \bar{P}_1''' & \stackrel{\mathrm{def}}{=} & (\mathit{rel}, r).\bar{P}_1 \end{array}$$

If the rates of the *think* activities are such that $\lambda_1 = \lambda_{11} + \lambda_{12}$, then $\bar{P}_1$ is strongly equivalent to P_1 although the two are clearly not isomorphic. Thus if we consider the system

$$(P_1 \parallel \bar{P}_1 \parallel P_2 \parallel P_2) \underset{L}{\bowtie} \mathit{Mem}_2$$

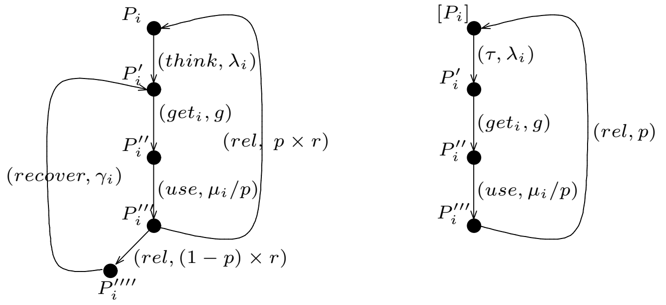
our algorithm will distinguish the derivatives $(P_1'' \parallel \bar{P}_1' \parallel P_2' \parallel P_2') \underset{L}{\bowtie} \mathit{Mem}_2'$ and $(P_1' \parallel \bar{P}_1'' \parallel P_2' \parallel P_2') \underset{L}{\bowtie} \mathit{Mem}_2'$ whereas a partitioning based on strong equivalence would consider them to be equivalent. In this case the state space aggregated by our algorithm will have 64 states whereas aggregation based on strong equivalence would result in 42 states.

C. Vertical aggregation via strong equivalence

We can identify a second source of aggregation which can be achieved by strong equivalence but which is not captured by our algorithm: so-called *vertical aggregation*. Here we illustrate the vertical aggregation case by means of another variant of the multiprocessor example. We consider a process which, after the use of the memory, can detect an error. If this is the case it does not return directly to the initial state; instead, it must complete a recovery action and repeat the access to the memory. For this new process the expansion of derivatives could be as follows:

$$\begin{array}{lcl} P_i & \stackrel{\mathrm{def}}{=} & (\mathit{think}, \lambda_i).P_i' \\ P_i' & \stackrel{\mathrm{def}}{=} & (\mathit{get}_i, g).P_i'' \\ P_i'' & \stackrel{\mathrm{def}}{=} & (\mathit{use}, \mu_i).P_i''' \\ P_i''' & \stackrel{\mathrm{def}}{=} & (\mathit{rel}, (1-p) \times r).P_i + (\mathit{rel}, p \times r).P_i'''' \\ P_i'''' & \stackrel{\mathrm{def}}{=} & (\mathit{recover}, \gamma_i).P_i' \end{array}$$

where p is the probability that an error occurs. The derivation graph of such a process P_i is shown in Fig. 9(a). Now we suppose that the action types *think* and *recover* are hidden and become internal to the component. Moreover,



(a) (b)

Fig. 9. Derivation graphs of P_i

we assume that $\lambda_i = \gamma_i$. If this is the case the derivatives P_i and P_i'''' are strongly equivalent, and we can aggregate them to form the macro-state $[P_i]$. Similarly, we combine the arcs labelled $(\mathit{rel}, (1-p) \times r)$ and $(\mathit{rel}, p \times r)$ into a single arc labelled (rel, r) connecting P_i''' and $[P_i]$ (see Fig. 9(b)).

Clearly this form of aggregation relies on the information about the operational behaviour of the component represented in the derivation graph. It cannot be detected by the purely syntactic means used in our algorithm. Approaches based on bisimulation style equivalences, such as strong equivalence, work at the semantic rather than the syntactic level. Thus they are not, in general, comparable to our approach.

VI. Related work

The exploitation of symmetries to achieve aggregation of performance models is a well-explored topic. Several automated approaches have been described in the literature. In this section we give a brief account of some of the work that has appeared in the context of stochastic Petri nets and stochastic process algebras, and explain how that work relates to our own. In each case the objective is to generate a partitioning of the original CTMC which satisfies the condition of lumpability.

The closest approach to our own is the work on a class of stochastic coloured Petri nets called *Stochastic Well-formed Nets* (SWN) [12]. Stochastic Petri nets (SPN) [13] have been extensively used for the functional analysis and performance evaluation of distributed systems. Their modelling primitives consist of *places* and (timed) *transitions*, representing system states and system events respectively. Just as in PEPA, in order to analytically solve an SPN model, the associated stochastic process must be derived by computing the set of reachable states (*markings*). Moreover, just as in PEPA, for realistic systems the computation of the state space can often lead to models whose size makes them intractable.

In order to tackle this problem SWN allow the construction of a parametric representation of a system. This is achieved by *folding* similar subnets and by adding a colour structure to distinguish tokens that, after the folding, belong to the same place. The nets are restricted in terms of the possible colour domains for places and transitions and in terms of the possible colour functions. These restrictions allow symmetric structures within the model to be exploited for solution purposes. In particular, these structures are automatically detected and the reduced state space is constructed without recourse to the complete state space. The reduction is obtained through the concept of *symbolic marking* [12].

Informally, a symbolic marking corresponds to an equivalence class of ordinary markings sharing the same characteristics. In fact, the ordinary markings in the same equivalence class enable the same set of transitions, whose firings lead to new ordinary states which are still equivalent, i.e. belong to the same symbolic marking.

Starting from a symbolic representation of the initial marking a symbolic reachability graph is constructed via a symbolic firing rule. Each symbolic marking is represented in a minimal, canonical form. Note that unlike our algorithm in which minimisation is carried out only in the preprocessing, in the SWN case minimisation has to be repeated after each symbolic derivation step. The symbolic reachability graph is used to generate a reduced CTMC and it has been proved [14] that it is lumpably equivalent to the original CTMC. Thus the same performance estimates can be computed with a lower computational cost.

Another Petri net-based approach has been developed in the context of Stochastic Activity Networks (SAN) [15]. This formalism incorporates features of both SPNs and queueing models and makes use of compositional operators, similar to those found in process algebras. The primitives of the formalism are *places*, *activities* (equivalent to Petri nets transitions), which may be guarded by *input gates*, representing enabling rules, and *output gates* representing completion rules. Once submodels have been constructed representing the components of the system they may be combined using the *replication* and *join* operations. The replication operator captures the case of a system containing two or more identical subsystems. The join operator combines SAN submodels of different types. Use of these operators makes symmetries within the model explicit and so facilitates a compact representation of the state space.

The structure of a composed SAN is represented by a directed tree with different types of nodes. Leaf nodes capture the distinct SAN submodels, i.e. the basic elements to which the construction operators apply. Internal nodes with one child are replication nodes, their child being the submodel to be replicated. Internal nodes with two or more children are join nodes, the children representing the submodels to be joined together.

From this tree a state representation is automatically extracted that is minimal in the sense that states which differ only by a permutation of repeated components are grouped together into a single combined state. Each such state is represented by recording, for each replication node, the number of replicated SANs in each possible submodel marking, and for each join node, a vector of the markings of each joined submodel. In addition each state maintains information about the desired performance variable [15] but this is outside the scope of this paper. There are clear parallels between this state representation and our vector form discussed in Section III-B.

The other work on aggregation of stochastic process algebra models is developed almost entirely at the semantic level. In this approach well known graph partitioning algorithms are used to reduce the labelled transition system underlying the process algebra model [16, 4]. In [17] a more syntactic approach is taken but this is on an ad hoc basis without a corresponding tool implementation. Equational laws derived from Markovian bisimulation, which is equivalent to strong equivalence, are used to obtain state space reduction of a MTIPP model. This is achieved by term rewriting based on judicious application of the laws. However, although good results can be obtained on particular models, no set of term rewriting rules which can be used for aggregation purposes have been found.

In some approaches good results have been obtained by modifying and restricting the combinators of the language to make symmetries more explicit and disallowing difficult cases. For example, in [18] a symmetric parallel composition operator, denoted $\{n!P\}S$ is used to capture the case of n-ary parallel composition of identical replicas, all synchronising on actions in S. This operator provides a means of expressing a number of replicated copies of a process but it cannot express synchronisation of repeated copies over different synchronisation sets. The operational semantics of the new operator is consistent with the usual parallel composition but a reduced state space is produced. This can be regarded as the SPA equivalent of the SAN approach outlined above. States which differ only by a permutation of replicated submodels are treated as equivalent.

Earlier work on MTIPP took a similar approach in terms of altering the combinators of the language. In [19] a replication operator, here denoted $!^n_S\ P$ has the same informal semantics as $\{n!P\}S$ above. Hiding and the usual general parallel composition operator are removed from the language. The distinction of this approach is that a denotational matrix semantics is given rather than the more usual operational semantics. Using this approach the infinitesimal generator matrix of the CTMC is constructed directly. Moreover Rettelbach and Siegle show that the transition matrix resulting from the semantics are minimal with respect to Markov chain lumpability (i.e. the matrices do not have subsets of equivalent states).

The disadvantages of both these approaches are that they require the modeller to adhere to a new set of combinators and this form of cooperation does not allow different synchronisation sets amongst replicas of the same component. The techniques do not appear to have been automated. In contrast our algorithm works transparently with the PEPA language taking advantage of whatever symmetries are present in the model submitted to the PEPA Workbench by the user.

VII. Conclusions and further work

We have shown how the existence of isomorphisms between terms in the derivation graph of a stochastic process algebra model can be exploited to aggregate the state space of the model. Our algorithm for this collapses the derivation graph at each model state and does not require a costly computation of bisimulation equivalence between components of the model. We have found it to be applicable in situations where the full derivation graph is too large even to be generated [20]. Further, we believe that many of the models which occur in practice would contain symmetries of the types which can be exploited by isomorphism. However, against these advantages our algorithm cannot be guaranteed to achieve the maximum possible aggregation for all models.

Generating an aggregated derivation graph will allow speedier computation of the steady state probability distribution of the CTMC which corresponds to a PEPA model. We have not discussed in this paper the influence of aggregation on the interpretation of this probability distribution in terms of the given PEPA model. When examining the steady state distribution in order to determine performance factors such as throughput and utilisation the PEPA modeller must now select sets of model states of interest via the description of canonical representatives in the state space. This is an added reason for choosing to aggregate with isomorphism instead of with bisimulation because the formation of a canonical representative of an isomorphism class is simpler. However, the full investigation of this issue remains as further work.

Our work has been influenced by earlier work on SWN [21]. However, we stress that significant adjustments to the approach have been necessary for the development of the algorithm for SPA: it is not a straightforward translation of results. Nevertheless we feel that there is considerable benefit to be gained from studying the relationship between formalisms with the objective of importing ideas, and when appropriate, techniques from one to the other.

Acknowledgements

This collaboration took place due to the British Council/MURST project "Rom/889/94/9: An enhanced toolset for performance engineers". Stephen Gilmore is supported by the 'Distributed Commit Protocols' grant from the EPSRC and by Esprit Working group FIREworks. Jane Hillston is supported by the ESPRC 'COMPA' grant.

The authors would like to thank the anonymous referees for helpful comments on an earlier version of this paper and to thank Graham Clark for implementation work on the PEPA Workbench.

References

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

[2] N. Götz, H. Hermanns, U. Herzog, V. Mertsiotakis, and M. Rettelbach, "Stochastic process algebras," in *Quantitative Methods in Parallel Systems*, F. Baccelli, A. Jean-Marie, and I. Mitrani, Eds., Basic Research Series, pp. 3–17. Springer, 1995.

[3] M. Bernardo and R. Gorrieri, "A Tutorial on EMPA: A Theory of Concurrent Processes with Nondeterminism, Priorities, Probabilities and Time," *Theoretical Computer Science*, vol. 202, no. 1–2, pp. 1–54, 1998.

[4] J. Hillston, "Compositional Markovian modelling using a process algebra," in *Proc. 2nd International Workshop on the Numerical Solution of Markov Chains*, Raleigh, North Carolina, January 1995.

[5] S. Gilmore and J. Hillston, "The PEPA Workbench: A tool to support a process algebra based approach to performance modelling," in *Proc. Seventh International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, Vienna, May 1994, number 794 in Lecture Notes in Computer Science, pp. 353–368, Springer-Verlag.

[6] M. Bernardo, W.R. Cleaveland, S.T. Sims, and W.J. Stewart, "TwoTowers: A tool integrating function and performance analysis of concurrent systems," in *Proc. of IFIP Joint Int. Conf. on Formal Description Techniques and Protocol Specification, Testing and Verification*. 1998, North-Holland (IFIP).

[7] H. Hermanns, U. Herzog, U. Klehmet, V. Mertsiotakis, and M. Siegle, "Compositional performance modelling with TIPPTool," in *Proc. of Int. Conf. on Modelling Techniques and Tools for Computer Performance Evaluation*, R. Puigjaner, Ed., Palma de Mallorca, Spain, September 1998, number 1469 in LNCS, Springer-Verlag.

[8] J.G. Kemeny and J.L. Snell, *Finite Markov Chains*, Van Nostrand, Princeton, NJ, 1960.

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

[10] S. Gilmore, J. Hillston, and D.R.W. Holton, "From SPA models to programs," in *Proceedings of the Fourth Annual Workshop on Process Algebra and Performance Modelling*, M. Ribaudo, Ed. Università di Torino, July 1996, pp. 179–198.

[11] M. Ribaudo, "On the aggregation techniques in stochastic Petri nets and stochastic process algebras," in *Proceedings of the Third International Workshop on Process Algebras and Performance Modelling*, S. Gilmore and J. Hillston, Eds. Dec. 1995, pp. 600–611, Special Issue of *The Computer Journal*, 38(7).

[12] G. Chiola, C. Dutheillet, G. Franceschinis, and S. Haddad, "Stochastic Well-Formed coloured nets for symmetric modelling applications," *IEEE Transactions on Computers*, vol. 42, no. 11, Nov. 1993.

[13] M.K. Molloy, "Performance analysis using stochastic Petri nets," *IEEE Transactions on Computers*, vol. 31, no. 9, pp. 913–917, Sept. 1982.

[14] G. Chiola, C. Dutheillet, G. Franceschinis, and S. Haddad, "Stochastic Well-Formed coloured nets and multiprocessor modelling applications," in *High-Level Petri Nets. Theory and Application*, K. Jensen and G. Rozenberg, Eds. Springer Verlag, 1991.

[15] J.F. Meyer and W.H. Sanders, "Reduced Base Model Construction Methods for Stochastic Activity Networks," *IEEE Journal on Selected Areas in Communications*, vol. 9, no. 1, pp. 25–36, Jan. 1991, Special issue on Computer-Aided Modeling, Analysis and Design of Communication networks.

[16] H. Hermanns and U. Herzog, "Compositional nets and compositional aggregation," in *Performance Models for Discrete Event Systems with Synchronisations: Formalisms and Analysis Techniques*, G. Balbo and M. Silva, Eds., vol. 2, pp. 553–582. Prensas Universarias de Zaragoza, 1998.

[17] H. Hermanns, U. Herzog, and V. Mertsiotakis, "Stochastic Process Algebras as a Tool for Performance and Dependability Modelling," in *IEEE International Computer Performance and Dependability Symposium*, Erlangen, 1995, pp. 102–111.

[18] H. Hermanns and M. Ribaudo, "Exploiting Symmetries in Stochastic Process Algebras," in *Proc. of 12th European Simulation Multiconference (Manchester, UK)*. 1998, SCS Europe.

[19] M. Rettelbach and M. Siegle, "Compositional Minimal Semantics for the Stochastic Process Algebra TIPP," in *Proc. of the 2^{nd} Process Algebra and Performance Modelling Workshop*, M. Rettelbach U. Herzog, Ed., Erlangen, 1994.

[20] J. Hillston and L. Kloul, "Investigating an On-Line Auction System using PEPA," *Concurrency: Practice and Experience*, 2000, To appear.

[21] G. Chiola, C. Dutheillet, G. Franceschinis, and S. Haddad, "On Well-Formed coloured nets and their symbolic reachability graph," in *Proc. 11^{th} Intern. Conference on Application and Theory of Petri Nets*, Paris, France, June 1990, Reprinted in *High-Level Petri Nets. Theory and Application*, K. Jensen and G. Rozenberg (editors), Springer Verlag, 1991.

Stephen Gilmore received both his B.Sc. and his Ph.D. from the Queen's University of Belfast in Northern Ireland. He currently holds a lectureship in Computer Science at The University of Edinburgh where he is a member of the Laboratory for Foundations of Computer Science and an associate member of the Institute for Computing Systems Architecture. His interests include the development of tools to support the performance modelling process. His personal web page with information on research projects and copies of his published papers can be found at `http://www.dcs.ed.ac.uk/~stg`.

Jane Hillston received a B.A. and an MSc. in Mathematics from the University of York and Lehigh University, respectively. After a brief period working in industry she joined the Department of Computer Science at the University of Edinburgh as a research assistant in 1989. She received a Ph.D. in Computer Science from that university in 1994. Since 1995 she has been a lecturer in Computer Science and a member of the Laboratory for Foundations of Computer Science. Her principal research interests are in the use of process algebras to model computer systems, and the investigation of issues of compositionality with respect to Markov processes.

Marina Ribaudo graduated in Computer Science at the University of Torino (Italy) in 1990 and she obtained a Ph.D. in Computer Science from the same University in 1995. Since July 1995 she has been a researcher at the Computer Science Department of the University of Torino. Her current research interests are in the area of performance evaluation of computer systems, mainly in the fields of stochastic process algebras and stochastic Petri nets. Her Ph.D. thesis examines the relationships existing between these two formalisms.