Systematic transformations to find quasi-reversible structures in PEPA models

Stephen Gilmore[1], Zully Grant-Duff[1,2], Peter Harrison[2], and Jane Hillston[1]

[1] Laboratory for Foundations of Computer Science, The University of Edinburgh, Edinburgh EH9 3JZ, Scotland.

[2] Department of Computing, Imperial College of Science, Technology and Medicine, University of London, London SW7 2BZ, England.

Abstract. Efficient product form solution is one of the major attractions of queueing networks for performance modelling purposes. These models rely on a form of interaction between nodes in a network which allows them to be solved in isolation, since they behave as if independent up to normalisation. Several papers have explored the extent to which product form structures can be identified in models expressed in a Markovian process algebra (MPA). In [1], Harrison and Hillston presented a syntactic characterisation of MPA models which give rise to quasi-reversible structure, leading to product form solution. This characterisation means that models may be systematically checked for the quasi-reversible property, without expanding the state space. It is one of the features of process algebra models, however, that different syntactic presentations may give rise to the same underlying structure. In this paper we describe a set of transformations which may be used to manipulate the syntactic presentation of a PEPA model, thus allowing a previously obscured quasi-reversible structure to be detected.

1 Introduction

Compositionality is often cited as one of the major advantages of stochastic process algebras (SPA). The ability to develop the model of a complex system in terms of subsystems, which are smaller and more easily modelled, has clear benefits for model construction. Using a SPA, unlike a stochastic Petri net (SPN), this structure remains self-evident within the constructed model. It has already been demonstrated that for the Markovian process algebras (MPA), the structure can also be exploited for model simplification [2, 3]. A disadvantage of the compositional approach is the state space explosion problem—since the global state of the model is expressed in terms of the local states of all the submodels, the size of the state space, in general, grows exponentially.

Queueing networks offer a compositional approach to the construction of models which is often exploited for the decomposed solution of the underlying Markov process. The system is decomposed into service centres which represent separate aspects of processing, and a customer's progress through the system is represented as a journey between these nodes. Many such networks have been shown to exhibit a product form equilibrium distribution for their steady state probabilities, leading to efficient solution techniques in which the service centres are solved in isolation. Thus the advantages of compositionality for model construction are gained without the disadvantages of the global state space.

Several papers have explored the possibility of finding similar *product form* solutions for MPA models [1, 4–7]. Each of these results relies on forming a syntactic characterisation, in terms of the MPA, of models which will exhibit a product form solution when considered at the underlying Markov process level.

For example in [1], considering the quasi-reversibility property, given a model

$$P \underset{L}{\bowtie} Q$$

the authors derive syntactic conditions on P, Q and L which allow us to recognise when the equilibrium distribution of the model, denoted $\Pi(\cdot)$, has a product form, i.e.

$$\Pi(P \underset{L}{\bowtie} Q) \equiv \Pi(P) \cdot \Pi(Q)$$

This characterisation allows models to be systematically checked for the quasi-reversibility property, without expanding the global state space.

However, it is one of the features of process algebra models that different syntactic presentations may give rise to the same underlying stochastic structure. Thus it is possible that a model which does not match the quasi-reversibility characterisation, nevertheless has a Markov process which is quasi-reversible. Our objective in this paper is to widen the class of models which may be automatically recognised as amenable to product form solution due to the quasi-reversible property, by defining systematic transformations to the syntactic presentations of models. Thus we take the previously published characterisation as a canonical form and attempt to transform any given model into that form. Success establishes that the model is indeed quasi-reversible, and so has product form solution.

The rest of the paper is structured as follows. In Section 2 we define the class of quasi-reversible PEPA models. In Section 3 we introduce rewriting rules for such models. In Section 4 we present an example of the application of these rules. In Section 5 we discuss our implementation of the rules in a term-rewriting kit for PEPA. Conclusions are presented in Section 6. Several MPAs have been proposed in the literature; in this paper, as in all the cited work on product form MPA, Hillston's PEPA (Performance Evaluation Process Algebra) [8] is used. We assume that the reader is familiar with PEPA and with Markov processes.

2 Quasi-reversible structures in PEPA models

Quasi-reversibility, as defined in terms of queues and customers, has been used to show that queues behave as if independent (up to normalisation) in steady-state queueing networks possessing certain properties. Therefore, given information about the influence of the rest of the network on each queue, these queues may be solved in isolation. The required information is the arrival rate of customers when the system is in equilibrium and it is usually derived from traffic equations.

Quasi-reversibility imposes a strong condition on the arrival rates of customers but a weaker condition on probability flux than that imposed by *reversibility.* A reversible process must necessarily satisfy the detailed balance equations

$$\pi(x)\, q(x,x') = \pi(x')\, q(x',x) \qquad \text{for all states } x, x'$$

where q is the process's instantaneous state transition matrix. A quasi-reversible process satisfies instead *partial balance equations* of the form

$$\pi(x) \sum_{x' \in S'} q(x,x') = \sum_{x' \in S'} \pi(x') q(x',x)$$

for all states x and a corresponding subset of states S'. Thus for any state x the probability flux into and out of the subset of states S' is balanced. In queueing terms the set of states, S', comprises those states which differ from x by the arrival of exactly one customer at some server. Additionally, the arrival rates of customers must be independent of the current state of the queue. It has been shown that for both open and closed networks of queues, which are quasi-reversible when considered in isolation, the queues behave independently (up to normalisation in the closed case), meaning that the equilibrium distribution has a product form. Moreover the network itself forms a quasi-reversible system.

Consider a stochastic process $X(t)$ the state of which can change either via an *input process* that causes one of a countable set of specified types of change in the state, say $x \to x'$, or via a dual *output process* that causes a change of state in the opposite direction, for example $x' \to x$. In queueing network theory the input process corresponds to arrivals to a queue (or subnetwork of queues) and the output process corresponds to departures.

Definition 1 *Such a stochastic process is* quasi-reversible *if, and only if,* $X(t)$ *is a stationary Markov process such that for all times* t_0 *the state* $X(t_0)$ *is independent of:*

1. *the input process after* t_0*;*
2. *the output process before* t_0*.*

By Burke's theorem, the M/M/1 queue with constant rates is quasi-reversible. It is also obviously reversible, by inspection of its state transitions which form a birth-death process, but in general neither of the properties of reversibility and quasi-reversibility implies the other.

We call the thinned input process obtained by selecting only those state changes of a particular type r the *input subprocess* r. The following proposition then holds:

Proposition 1 *For a quasi-reversible process:*

1. *the input subprocesses are Poisson with constant rate;*
2. *the output subprocesses are Poisson with constant rate.*

The key property we require is that each input subprocess has *constant rate*, $\alpha(r)$ say for subprocess r. Using a combination of reversibility and quasi-reversibility arguments, it can be shown that a single multi-class queue and networks thereof satisfy *local* balance equations and have product-form solutions.

We are interested in cases when the Markov process underlying a PEPA model has a product form equilibrium distribution which corresponds to the co-operating components within the model. In these cases the probability of a given model derivative will be the product of the probabilities of the corresponding derivatives in the cooperating components, possibly subject to renormalisation. In other words, if the derivative P_i is expressed in state vector representation as $(C_{1_i}, C_{2_i}, \ldots, C_{n_i})$, then Π, the steady state probability distribution, over the derivatives of P, can be expressed as

$$\Pi(P_i) = \frac{1}{G_i} \prod_{j=1}^{n_i} \pi_j(C_{j_i})$$

where π_j is the steady state probability distribution over the derivatives of $C_j, 1 \leq j \leq n_i$, and G_i is a normalisation constant. The form of the cooperating components is restricted in order to ensure that each element corresponds to a quasi-reversible Markov process. Moreover, the nature of the interaction between them ensures the quasi-reversibility of the model component considered as a whole.

Definition 2 *A PEPA component M is a* QR component *if and only if its underlying Markov process is a quasi-reversible process.*

As a first step we identify activities that are analogous to customer arrivals and departures in a queue. The crucial characteristic of these pairs of events is the fact that they *reverse* each other: one event followed immediately by the other returns us to the original state.

Definition 3 *A PEPA component P is said to enable a* reverse pair $(\alpha, -\alpha)$*, if $(\alpha, r) \in \mathcal{A}ct(P)$ and for every (α, r)-derivative P' there exists $(-\alpha, s) \in \mathcal{A}ct(P')$ such that P is an $(-\alpha, s)$-derivative of P', where $r, s \in \mathbb{R}^+ \cup \{\top\}$.*

Note that if $(\alpha, -\alpha)$ is a reverse pair between components P and P', then $(-\alpha, \alpha)$ is a reverse pair between P' and P, i.e. $-(-\alpha) = \alpha$. For example, in the queue modelled below, $(\mathit{arrive}, \mathit{serve})$ is a reverse pair in every derivative, but $(\mathit{serve}, \mathit{arrive})$ is another reverse pair in all Q_i, $i \geq 1$.

$$\begin{aligned} Q_0 &\stackrel{\text{def}}{=} (\mathit{arrive}, \lambda).Q_1 \\ Q_i &\stackrel{\text{def}}{=} (\mathit{arrive}, \lambda).Q_{i+1} + (\mathit{serve}, \mu).Q_{i-1} \qquad i \geq 1 \end{aligned}$$

If all the activities that a sequential PEPA component enables constitute half of a reverse pair then it is clear how to form its underlying reversed Markov process. For any reverse pair $(\alpha, -\alpha)$, if α occurs in the original process at time t then in the reverse process $-\alpha$ occurs at time $-t$.

The simplest form of QR components will be those which exhibit a strict input/output behaviour with respect to their reverse pairs. As a preliminary we generalise the notion of an (α, r)-derivative.

Definition 4 *If* $P \underbrace{\xrightarrow{(\alpha,r)} \cdots \xrightarrow{(\alpha,r)}}_{n \text{ times } (\alpha,r)} P'$ *then* P' *is an* $(\alpha, r)^n$-derivative *of* P.

In order to form an *input process*, corresponding to arrivals at a queue, we require that the same *input* action (the (α, r) of a reverse pair $(\alpha, -\alpha)$) is enabled in all the derivatives of the component and that the component must change state monotonically whenever such an action occurs.

Definition 5 *A PEPA component* P *with initial component* P_0 *is an* input-output *component if* P_0 *enables only passive activities* $(\alpha, \top)$ *such that:*

1. $(\alpha, -\alpha)$ *is a reverse pair;*
2. $(\alpha, \top)$ *and* $(-\alpha, s)$*, for some* s*, are the only activities enabled by any derivative of* P_0*; and*
3. *the* $(\alpha, \top)^n$*-derivatives of* P_0 *are distinct for all* n*.*

The set of action types $\mathcal{I}(P) = \mathcal{A}(P_0)$ is the set of *input actions* of P. These correspond directly to the input subprocesses in the underlying Markov process: one action type for each input subprocess. The *output actions* are those which reverse the input actions:

$$\mathcal{O}(P) = \{\beta \mid \beta = -\alpha \wedge \alpha \in \mathcal{I}(P)\}$$

The completion of activities of these types corresponds to the output subprocess of the appropriate class. We assume that the input and output actions of any input-output process are disjoint.

A PEPA component containing passive activities which are not shared with an active component is incomplete. We say that a component $S_\alpha^> \overset{\text{def}}{=} (\alpha, r).S_\alpha^>$ such that $r \neq \top$, is an *external source* for an input-output process P if $\alpha \in \mathcal{I}(P)$.

Proposition 2 *If a PEPA model* M *consists of an input-output component* P *in cooperation with an external source* $S_\alpha^>$ *for each* $\alpha \in \mathcal{I}(P)$*,*

$$M \overset{\text{def}}{=} P \underset{\mathcal{I}(P)}{\bowtie} (S_{\alpha_1}^> \parallel \cdots \parallel S_{\alpha_n}^>)$$

where $\mathcal{I}(P) = \{\alpha_1, \ldots \alpha_n\}$*, then* M *is a QR component, i.e. the Markov process underlying* M *is quasi-reversible.*

This follows from the definition of quasi-reversibility.

If $\mathcal{I}(P) = \{\alpha\}$ then by the definition of an input-output component, the Markov process underlying M is isomorphic to a reversible queue with one class of customers, whose arrival process is determined by the external source. These basic components, whether with one input activity type or more, correspond to the queues within a network. The class of QR components is larger than this; we can enlarge it by considering the interactions of such components that correspond to networks of queues.

Now we consider models formed by the interaction of input-output components, and their appropriate external sources. Suppose $C \stackrel{\mathrm{def}}{=} P \underset{L}{\bowtie} Q$ where both P and Q are input-output components. We say that $\alpha \in L$ is a *channel from P to Q* if $\alpha \in \mathcal{O}(P) \cap \mathcal{I}(Q)$; similarly $\beta \in L$ is a *channel from Q to P* if $\beta \in \mathcal{I}(P) \cap \mathcal{O}(Q)$. Note that since the input and output sets of all components are distinct all channels are uni-directional.

If every action in the set $L \cap (\mathcal{I}(P) \cup \mathcal{I}(Q))$ forms a channel in the component $P \underset{L}{\bowtie} Q$, the interaction is termed a *flow cooperation.* (Any action in L but not in $\mathcal{I}(P) \cup \mathcal{I}(Q)$ is unable to participate in the cooperation, of course, but would be syntactically valid.) In the more general case, a cooperation of n input-output components $P_1, \ldots, P_n$, with state vector representation $(P_1, \ldots, P_n)$, is a flow cooperation if, when we consider the pairwise interactions between the components, they all form channels. More formally, for any $P_i, P_j,\ 1 \leq i, j \leq n$ such that $i \neq j$, if P_i and P_j cooperate over α then either α is a channel from P_i to P_j or α is a channel from P_j to P_i. We denote the set of interactions between P_i and P_j by L_{ij}.

For any QR component, the input actions are those corresponding to an input subprocess in the underlying quasi-reversible process, and the output actions are those corresponding to an output subprocess. A flow cooperation between such components is then defined as above.

A flow cooperation of input-output components is said to be *closed* if it has no actions enabled which do not form a channel. In this case all the input action needs of the cooperating components are supplied by output actions of other components in the cooperation, and similarly all output actions produced by components are shared, as input actions, with other components. Conversely, a flow cooperation which enables actions which are not part of a channel is termed *open.* Such input actions must be provided with an external source of appropriate type to complete the model. The resulting actions will form an input subprocess to the model. Similarly, if a model enables an output action which does not form a channel the model will exhibit an output subprocess.

In [1] it is established that an open flow cooperation of QR components is a QR component; moreover, the underlying Markov process has a product form equilibrium distribution over the components. In the case of a closed flow cooperation the model is no longer a QR component, since there are no input or output subprocesses. Nevertheless, components in such a cooperation behave as if they were independent, up to normalisation, resulting in a product form equilibrium distribution for the model. For the remainder of this paper we will only consider closed flow cooperations.

3 An Algebra of State Vectors using PEPA Combinators

3.1 Vector Form

When a PEPA model is defined to consist of one or more cooperating components, these will be apparent in every derivative of the model. The *vector form* of a model expression represents the state of a model as a tuple of local states (one for each component) within decorated brackets. As in [9], we use subscripted brackets to denote the scope of a cooperation set. We do not consider the hiding operator.

Definition 6 (Vector Form) *For a model expression, we define the vector form inductively over the structure of the expression: let P and Q be expressions and C be a constant denoting a sequential component.*

1. $\mathsf{vf}(P \underset{L}{\bowtie} Q) = (\mathsf{vf}(P), \mathsf{vf}(Q))_L$
2. $\mathsf{vf}(C) = C$

This representation contains all the information about the static structure of the model: it records the current derivative of each sequential component as well as the scope of the cooperation sets which are in force. However this is not sufficient to allow us to compute the derivation graph: the defining equations for the sequential components are also needed.

In the following subsection we introduce our transformation system as a set of rules which permit re-writing of state vector expressions. This requires an overloading of the PEPA combinators—prefix, choice and cooperation—over the space of such vectors. The meaning of these overloadings are given in the semantics shown in Figure 1. The presentation uses the notion of *apparent rate*, r_α, defined in [8]. The equality used in the rules denotes an *isomorphism relation*: two state vector expressions are considered equal in this sense if and only if, by the rules given in Figure 1, they give rise to isomorphic derivation graphs. As a consequence, the rules can be applied *right-to-left* or *left-to-right.*

3.2 Transformation System

In the style of Burstall and Darlington's *fold-unfold* methodology [10], PEPA constant definitions are used as equational re-write rules. In this sense, the substitution of a component by its definition is an *unfold* step; the substitution of a vector form is a *definition/abstraction* step and the *primitive laws* are those based on the properties of the PEPA combinators. In addition, we have the *rules* we present below for manipulating state vectors.

In all the rules we assume a pair of components, P and Q, between which there is a flow cooperation, governed by the cooperation set L. The first rule dictates when a state vector may be prefixed by an activity. The first sub-rule corresponds to the case when the action type α forms a channel between P and Q; the second and third, the cases when it does not.

Rule 1 (State vector prefix). *Assume P and Q are a pair of* flow cooperating components*, in the cooperation $P \underset{L}{\bowtie} Q$, with derivative state vector $(P', Q')_L$.*

Prefix

$$\frac{}{(\alpha, r).(P, Q)_L \xrightarrow{(\alpha,r)} (P, Q)_L}$$

Choice

$$\frac{(P, Q)_L \xrightarrow{(\alpha,r)} (P', Q')_L}{(P, Q)_L + (R, S)_K \xrightarrow{(\alpha,r)} (P', Q')_L} \qquad \frac{(R, S)_K \xrightarrow{(\alpha,r)} (R', S')_K}{(P, Q)_L + (R, S)_K \xrightarrow{(\alpha,r)} (R', S')_K}$$

Cooperation

$$\frac{(P, Q)_L \xrightarrow{(\alpha,r)} (P', Q')_L}{((P, Q)_L, (R, S)_K)_M \xrightarrow{(\alpha,r)} ((P', Q')_L, (R, S)_K)_M} \ (\alpha \notin M)$$

$$\frac{(R, S)_K \xrightarrow{(\alpha,r)} (R', S')_K}{((P, Q)_L, (R, S)_K)_M \xrightarrow{(\alpha,r)} ((P, Q)_L, (R', S')_K)_M} \ (\alpha \notin M)$$

$$\frac{(P, Q)_L \xrightarrow{(\alpha,r_1)} (P', Q')_L \qquad (R, S)_K \xrightarrow{(\alpha,r_2)} (R', S')_K}{((P, Q)_L, (R, S)_K)_M \xrightarrow{(\alpha,s)} ((P', Q')_L, (R', S')_K)_M} \ (\alpha \in M),$$

$$\text{where } s = \frac{r_1}{r_\alpha(P \underset{L}{\bowtie} Q)} \frac{r_2}{r_\alpha(R \underset{K}{\bowtie} S)} \min(r_\alpha(P \underset{L}{\bowtie} Q), r_\alpha(R \underset{K}{\bowtie} S))$$

Fig. 1. PEPA operational semantics extended to state vectors

Rule 1.1 If α is a channel, *such that $\alpha \in \boldsymbol{\mathcal{A}}(P) \cup \boldsymbol{\mathcal{A}}(Q)$ and $\alpha \in L$, then*

$$((\alpha, r).P', (\alpha, r).Q')_L = (\alpha, r).(P', Q')_L$$

Rule 1.2 If $\alpha \notin L$, then

$$((\alpha, r).P', Q)_L = (\alpha, r).(P', Q)_L$$

Rule 1.3 If $\alpha \notin L$, then

$$(P, (\alpha, r).Q')_L = (\alpha, r).(P, Q')_L$$

The second rule shows how choice between state vectors arises, when there are two channels between the components.

Rule 2 (State vector choice). *Assume P and Q are a pair of* flow co-operating components*, in the cooperation $P \underset{L}{\bowtie} Q$, with derivative state vectors $(P', Q')_L$ and $(P'', Q'')_L$. If α and β are* channels *such that $\{\alpha, \beta\} \subset \boldsymbol{\mathcal{A}}(P) \cup \boldsymbol{\mathcal{A}}(Q)$ and $\{\alpha, \beta\} \subset L$, then*

$$((\alpha, r).P' + (\beta, s).P'', (\beta, s).Q'' + (\alpha, r).Q')_L = (\alpha, r).(P', Q')_L + (\beta, s).(P'', Q'')_L$$

The third rule defines how progress may be blocked by the cooperation set. In the rule (β, s) affects only the state of the Q component (left hand side of the rule). Behaviourally this is not possible as (β, s) is a channel and must therefore affect the state of both cooperating components. Since (β, s) does not appear in both tuple elements the transformation rule discards its effect on the combined state as *invalid*.

Rule 3 (Activity blocking). *Assume P and Q are a pair of* flow cooperating components*, in the cooperation $P \underset{L}{\bowtie} Q$, with derivative state vector $(P', Q)_L$, $Q \equiv (\beta, s).Q'$. If β is a* channel *such that $\beta \in \boldsymbol{\mathcal{A}}(Q)$ and $\beta \in L$, then*

$$((\alpha, r).P', (\beta, s).Q')_L = (\alpha, r).(P', Q)_L$$

where $\alpha \neq \beta$ and $\alpha \notin L$

In the case of the fourth rule, (β, s) appears in both tuple elements hence the transformation rule accounts for its effect on the *combined* state. Note that the term $(\beta, s).(P'', Q')$ describes a *flow transition* whose input-output nature is reflected in the fact that both the state of the source P and that of the destination Q must change.

Rule 4 (Choice and cooperation). *Assume P and Q are a pair of* flow cooperating components*, in the cooperation $P \underset{L}{\bowtie} Q$, with derivative state vectors $(P', Q)_L$,and $(P'', Q'')_L$, $Q \equiv (\beta, s).Q''$. If β is a* channel *such that $\beta \in \boldsymbol{\mathcal{A}}(P) \cup \boldsymbol{\mathcal{A}}(Q)$ and $\beta \in L$, then*

$$((\alpha, r).P' + (\beta, s).P'', (\beta, s).Q'')_L = (\alpha, r).(P', Q)_L + (\beta, s).(P'', Q'')_L$$

where $\alpha \neq \beta$ and $\alpha \notin L$.

The fifth rule can be regarded as a generalisation of Rule 3 and Rule 4.

Rule 5 (Choice, cooperation and blocking). *Assume P and Q are a pair of* flow cooperating components*, in the cooperation $P \underset{L}{\bowtie} Q$, with derivative state vectors $(P', Q)_L$,and $(P'', Q'')_L$, $Q \equiv (\gamma, t).Q'$. If β and γ are* channels *such that $\{\beta, \gamma\} \subset L$, $\beta \in \boldsymbol{\mathcal{A}}(P) \cup \boldsymbol{\mathcal{A}}(Q)$ and $\gamma \in \boldsymbol{\mathcal{A}}(Q)$ then*

$$((\alpha, r).P' + (\beta, s).P'', (\beta, s).Q'' + (\gamma, t).Q')_L = (\alpha, r).(P', Q)_L + (\beta, s).(P'', Q'')_L$$

where $\alpha \neq \beta \neq \gamma$ and $\alpha \notin L$.

4 An example of quasi-reversibility

In this section we show how to formally transform one representation of a system into another, using the transformation rules introduced in the previous section. We consider a simple closed queueing network, consisting of two FIFO servers and a population of size three, initially at the first server. This system suggests two distinct PEPA representations. In the first, one PEPA component is defined for each server/queue. These components have a local state or derivative representing each possible state of the corresponding queue. This representation is shown in Figure 2.

$$
\begin{array}{ll}
S_{10} \stackrel{\text{def}}{=} (go_{21}, \mu_2).S_{11} & S_{20} \stackrel{\text{def}}{=} (go_{12}, \mu_1).S_{21} \\
S_{11} \stackrel{\text{def}}{=} (go_{21}, \mu_2).S_{12} + (go_{12}, \mu_1).S_{10} & S_{21} \stackrel{\text{def}}{=} (go_{12}, \mu_1).S_{22} + (go_{21}, \mu_2).S_{20} \\
S_{12} \stackrel{\text{def}}{=} (go_{21}, \mu_2).S_{13} + (go_{12}, \mu_1).S_{11} & S_{22} \stackrel{\text{def}}{=} (go_{12}, \mu_1).S_{23} + (go_{21}, \mu_2).S_{21} \\
S_{13} \stackrel{\text{def}}{=} (go_{12}, \mu_1).S_{12} & S_{23} \stackrel{\text{def}}{=} (go_{21}, \mu_2).S_{22}
\end{array}
$$

$$
\mathit{System} \stackrel{\text{def}}{=} S_{13} \underset{\{go_{12}, go_{21}\}}{\bowtie} S_{20}
$$

Fig. 2. Description 1.1: PEPA model of the two node system using two components

In the second representation a single PEPA component is used to capture the behaviour of the complete system, which in effect is sequentialised. This representation is shown in Figure 3. The first representation clearly satisfies the criteria for giving rise to a quasi-reversible Markov process:

- the activities go_{12} and go_{21} form a reverse pair in each component;
- each of the components is an input-output component; and
- the cooperation is a flow cooperation.

However in the second representation for component S_{1320} this is not the case.

4.1 Transformation from Description 1.1 to Description 1.2

When dealing with a server-based description such as this we simply start by representing the top level cooperation by a pair, our hypothesis being that there is a flow cooperation between S_{20} and S_{13} and that go_{12}, go_{21} are channels. Once the rewrite rules are applied to this *initial equation* new pairs are introduced and other definitions are (subsequently) tupled accordingly.

$$S_{1320} \stackrel{\text{def}}{=} (go_{12}, \mu_1).S_{1221} \tag{1}$$
$$S_{1221} \stackrel{\text{def}}{=} (go_{12}, \mu_1).S_{1122} + (go_{21}, \mu_2).S_{1320} \tag{2}$$
$$S_{1122} \stackrel{\text{def}}{=} (go_{12}, \mu_1).S_{1023} + (go_{21}, \mu_2).S_{1221} \tag{3}$$
$$S_{1023} \stackrel{\text{def}}{=} (go_{21}, \mu_2).S_{1122} \tag{4}$$
$$System \stackrel{\text{def}}{=} S_{1320}$$

Fig. 3. Description 1.2: PEPA model of the two node system using a single component

The starting point is a tuple of PEPA terms derived from Description 1.1. Let G be the set $\{go_{12}, go_{21}\}$.

$$\begin{aligned}(S_{13}, S_{20})_G &\stackrel{\text{def}}{=} ((go_{12}, \mu_1).S_{12}, (go_{12}, \mu_1).S_{21})_G \\ &\quad \{\text{by Rule 1}\} \\ &= (go_{12}, \mu_1).(S_{12}, S_{21})_G \end{aligned} \tag{5}$$

The behaviours defined by equations (5) and (1) are identical, in the sense that they give rise to isomorphic derivation graphs (only the names of derivatives may differ). This is clear since we have syntactically identical definitions up to renaming. Here there is a direct correspondence between the vectors $(S_{13}, S_{20})_G$ and $(S_{12}, S_{21})_G$ and the terms S_{1320} and S_{1221} respectively. (In this sense, no further transformation steps are required. For completeness one could rewrite equation (5) to $S_{1320} \stackrel{\text{def}}{=} (go_{12}, \mu_1).S_{1221}$, by substitution of constant names.)

$$\begin{aligned}(S_{12}, S_{21})_G &\stackrel{\text{def}}{=} ((go_{21}, \mu_2).S_{13} + (go_{12}, \mu_1).S_{11}, (go_{12}, \mu_1).S_{22} + (go_{21}, \mu_2).S_{20})_G \\ &\quad \{\text{by Rule 2}\} \\ &= (go_{21}, \mu_2).(S_{13}, S_{20})_G + (go_{12}, \mu_1).(S_{11}, S_{22})_G && (6) \\ &\quad \{\text{by commutativity of choice}\} \\ &= (go_{12}, \mu_1).(S_{11}, S_{22})_G + (go_{21}, \mu_2).(S_{13}, S_{20})_G && (7)\end{aligned}$$

Again, the behaviours defined by equations (7) and (2) are identical. The arguments to justify this are the same as above. Also, the choice combinator is assumed commutative which is easily shown by appealing to its definition. This will give us an exact match to the term S_{1221}.

The next two steps complete the transformation. Remarks are as above.

$$\begin{aligned}(S_{11}, S_{22})_G &\stackrel{\text{def}}{=} ((go_{21}, \mu_2).S_{12} + (go_{12}, \mu_1).S_{10}, (go_{12}, \mu_1).S_{23} + (go_{21}, \mu_2).S_{21})_G) \\ &\quad \{\text{by Rule 2}\} \\ &= (go_{21}, \mu_2).(S_{12}, S_{21})_G + (go_{12}, \mu_1).(S_{10}, S_{23})_G \\ &\quad \{\text{by commutativity of choice}\}\end{aligned}$$

$$= (go_{12}, \mu_1).(S_{10}, S_{23})_G + (go_{21}, \mu_2).(S_{12}, S_{21})_G$$

$$\begin{aligned}(S_{10}, S_{23})_G &\overset{\text{def}}{=} ((go_{21}, \mu_2).S_{11}, (go_{21}, \mu_2).S_{22})_G \\ &\quad \{\text{by Rule 1}\} \\ &= (go_{21}, \mu_2).(S_{11}, S_{22})_G\end{aligned}$$

4.2 Transformation from Description 1.2 into Description 1.1

We are interested in the structure which is hidden in this monolithic description. For QR models, as this one, there will be a successful transformation into a server centric description as shown below. This means that both descriptions are equivalent, i.e. isomorphic, and give rise to identical Markov processes. Our ultimate aim is to derive a flow cooperation representation, where all interactions are channels. Accordingly, once tuples are introduced, the next step is for the transformation to construct the cooperation set by identifying channels. We will find it useful to appeal to the following proposition.

Proposition 3 *Let $P \underset{L}{\bowtie} Q$ be a flow cooperation with derivative state vector $(P', Q')_L$ and $\alpha \in L$ a channel from P to Q. If there exists activity (β, s) such that (α, β) form a reverse pair between $(P, Q)_L$ and $(P', Q')_L$ then β is a channel from Q to P and thus $\beta \in L$.*

The proof is immediate by appeal to the definitions of *channel* and *reverse pair.*

Transformation Steps The first step in the transformation consists of substituting every derivative component in the definition of S_{1320} by a pair of components. For the case of (the initial state) S_{1320} we name them A_1 and A_2 (note that the choice of name is irrelevant). Our hypothesis is that S_{1320} is a flow cooperation over some set L such that go_{12} is a channel in L—which is inferred from the definition of S_{1320} itself. The objective of the transformation is to derive a definition for A_1 and A_2 and construct the set L. This will be done incrementally by adding to the initial set $L_0 = \{go_{12}\}$.

$$(A_1, A_2)_L \overset{\text{def}}{=} (go_{12}, \mu_1).(B_1, B_2)_L \tag{8}$$

$$(B_1, B_2)_L \overset{\text{def}}{=} (go_{12}, \mu_1).(C_1, C_2)_L + (go_{21}, \mu_2).(A_1, A_2)_L \tag{9}$$

$$(C_1, C_2)_L \overset{\text{def}}{=} (go_{12}, \mu_1).(D_1, D_2)_L + (go_{21}, \mu_2).(B_1, B_2)_L \tag{10}$$

$$(D_1, D_2)_L \overset{\text{def}}{=} (go_{21}, \mu_2).(C_1, C_2)_L \tag{11}$$

From equations (8) and (9) we infer that (go_{12}, go_{21}) is a reverse pair between $(A_1, A_2)_L$ and $(B_1, B_2)_L$ and (go_{21}, go_{12}) is a reverse pair between $(B_1, B_2)_L$ and $(A_1, A_2)_L$. Since go_{12} is a channel between A_1 and A_2 its reverse go_{21} is also a channel. From equations (14) and (11) we infer that (go_{12}, go_{21}) is a reverse pair between $(C_1, C_2)_L$ and $(D_1, D_2)_L$ and (go_{21}, go_{12}) is a reverse pair between $(D_1, D_2)_L$ and $(C_1, C_2)_L$. In other words, the components A_1 and A_2 enable the reverse pair (go_{12}, go_{21}). Hence, $L_1 = L_0 \cup \{go_{21}\}$.

Applying Rule 1 we rewrite all equations above.

$$\begin{aligned}
(A_1, A_2)_L &\overset{\text{def}}{=} ((go_{12}, \mu_1).B_1, (go_{12}, \mu_1).B_2)_L && (12)\\
(B_1, B_2)_L &\overset{\text{def}}{=} ((go_{12}, \mu_1).C_1, (go_{12}, \mu_1).C_2)_L \\
&\quad + ((go_{21}, \mu_2).A_1, (go_{21}, \mu_2).A_2)_L && (13)\\
(C_1, C_2)_L &\overset{\text{def}}{=} ((go_{12}, \mu_1).D_1, (go_{12}, \mu_1).D_2)_L \\
&\quad + ((go_{21}, \mu_2).B_1, (go_{21}, \mu_2).B_2)_L && (14)\\
(D_1, D_2)_L &\overset{\text{def}}{=} ((go_{21}, \mu_2).C_1, (go_{21}, \mu_2).C_2)_L && (15)
\end{aligned}$$

Further, applying Rule 2 to equations 13 and 14 we have:

$$\begin{aligned}
(A_1, A_2)_L &\overset{\text{def}}{=} ((go_{12}, \mu_1).B_1, (go_{12}, \mu_1).B_2)_L \\
(B_1, B_2)_L &\overset{\text{def}}{=} ((go_{12}, \mu_1).C_1 + (go_{21}, \mu_2).A_1, (go_{12}, \mu_1).C_2 + (go_{21}, \mu_2).A_2)_L \\
(C_1, C_2)_L &\overset{\text{def}}{=} ((go_{12}, \mu_1).D_1 + (go_{21}, \mu_2).B_1, (go_{12}, \mu_1).D_2 + (go_{21}, \mu_2).B_2)_L \\
(D_1, D_2)_L &\overset{\text{def}}{=} ((go_{21}, \mu_2).C_1, (go_{21}, \mu_2).C_2)_L
\end{aligned}$$

Last, by tuple equality:

$$\begin{aligned}
A_1 &\overset{\text{def}}{=} (go_{12}, \mu_1).B_1 \\
A_2 &\overset{\text{def}}{=} (go_{12}, \mu_1).B_2 \\
B_1 &\overset{\text{def}}{=} (go_{12}, \mu_1).C_1 + (go_{21}, \mu_2).A_1 \\
B_2 &\overset{\text{def}}{=} (go_{12}, \mu_1).C_2 + (go_{21}, \mu_2).A_2 \\
C_1 &\overset{\text{def}}{=} (go_{12}, \mu_1).D_1 + (go_{21}, \mu_2).B_1 \\
C_2 &\overset{\text{def}}{=} (go_{12}, \mu_1).D_2 + (go_{21}, \mu_2).B_2 \\
D_1 &\overset{\text{def}}{=} (go_{21}, \mu_2).C_1 \\
D_2 &\overset{\text{def}}{=} (go_{21}, \mu_2).C_2
\end{aligned}$$

The transformation terminates successfully since all named derivatives define a reachable state (i.e. all constants defined are specified in at least one equation on the right hand of the $\overset{\text{def}}{=}$ symbol). This is confirmation of original hypothesis—that the monolithic representation concealed a flow cooperation. Had our hypothesis been wrong the transformation would not have terminated.
The constructed specification is then

$$\mathit{System} \overset{\text{def}}{=} A_1 \underset{L}{\bowtie} A_2$$

where $L = L_1 = \{go_{12}, go_{21}\}$.

5 Implementation

Having now developed a set of formally defined transformations on PEPA models we have implemented these in a term rewriting system which checks the correct application of each of the rules from the previous section. Our system is called the PEPA Term Kit and we have used it to check the derivations which were presented in the previous section. The PEPA Term Kit is implemented in the strongly-typed functional language Standard ML [11]. Standard ML provides a metalanguage for defining other formal languages, whether they are programming languages or modelling languages as is the case for PEPA here. We have previously developed an embedding of PEPA in Standard ML [12] and our implementation reuses this existing language embedding.

The PEPA Term Kit presents users with an interactive top-level loop which allows them to enter model definitions, fold and unfold definitions and apply rewriting rules. After experimenting with the interactive application of the rules to strengthen understanding, the user can define tactics as in the style of Edinburgh LCF [13]. These tactics can be used to apply combinations of rules in a single step. The necessary uses of rules and tactics to prove a desired goal can be stored in a proof script which can be replayed as needed. The use of such a proof script is shown in Figure 4.

PEPA model definitions are packed into modules called structures. These model definitions introduce the ground terms of the rewriting system where the static type system of the language separates out the identifiers for activity types from the identifiers for rates or for model components. We apply one of Standard ML's parametric modules to a pair of model definitions in two such structures. This produces a set of custom unification procedures for these definitions. One of these unification procedure computes the most general unifier for pairs of terms which contain a left subterm from the first model definition and a right subterm from the second model definition. Another unification procedure checks that substitutions are applied consistently to lists of such pairs. Lists such as these are used to posit equivalences between definitions in the first model and definitions in the second. The application of the unification procedure for lists establishes that the posited equivalences genuinely do hold.

The PEPA Term Kit is available for download from the Internet address `http://www.dcs.ed.ac.uk/pepa/termkit`.

6 Conclusions

We have defined and implemented a term-rewriting system for stochastic process algebra models expressed in the PEPA modelling language. The term-rewriting system ensures the legitimate application of transformation rules to the syntactic representation of a model. By directing the application of the rules a user of the system can rewrite a model into a syntactically characterised quasi-reversible form without expanding the global state space. An efficient product form solution can then be obtained for a model in this quasi-reversible form.

```
PEPA Term Kit, Version 0.01 "Largs"                                                       (* start of session *)
[introducing top level component definitions]
[building unifier for PEPA terms]
> structure Unify :                                                                (* defining a module which *)
  {val (''e, ''f) appendWithConsistencyCheck :                             (* provides unification procedures *)
    (''e * ''f) list -> (''e * ''f) list -> (''e * ''f) list,
  val 'e applySubstitution :
    (Activity * Activity/1) list * (Rate * Rate/1) list *
    (Identifier * Component/1) list * 'e -> Component -> Component/1,
  val (''e, ''f) insert : ''e * ''f -> (''e * ''f) list -> (''e * ''f) list,
  val (''e, 'f) lookup : (''e * 'f) list -> ''e -> 'f,
  val unifiable : Component * Component/1 -> bool,                                 (* test if two terms unify *)
  val unifiableList : (Component * Component/1) list -> bool,             (* test if two lists of terms unify *)
  val unifier : Component * Component/1 ->
    (Activity * Activity/1) list * (Rate * Rate/1) list *                     (* compute most general unifier *)
    (Identifier * Component/1) list * (Identifier/1 * Component) list,
  val unifierList :
    (Component * Component/1) list ->
    (Activity * Activity/1) list * (Rate * Rate/1) list *
    (Identifier * Component/1) list * (Identifier/1 * Component) list}                 (* end of Unify module *)
[installed unifier for PEPA terms]
[defining transformations]
> structure Transformations :                                                 (* defining the transformations *)
  {structure Queue1 :                                                           (* the first Queue definition *)
    {datatype Activity = (Activity, {con go12, go21 : Activity}),                     (* the type of activities *)
      datatype Component = (Component,                                     (* the component combinator syntax *)
        {con $ : Identifier -> Component,                                                         (* variable *)
          con * : (Activity * Rate) * Component -> Component,                                       (* prefix *)
          con + : Component * Component -> Component,                                               (* choice *)
          con - : Component list * Activity list -> Component}),                               (* cooperation *)
      datatype Identifier = (Identifier,                                             (* the model identifiers *)
        {con S10, S11, S12, S13, S20, S21, S22, S23, System : Identifier}),
      datatype Rate = (Rate,{con mu1 : Rate, con mu2 : Rate}),                             (* the type of rates *)
      val Components : Identifier list,
      val commute : Component -> Component,
      val defn : Identifier -> Component,
      val fold : Component -> Identifier,
      val recUnfold : Component -> Component,
      val rule : int -> Component -> Component,                                (* apply a transformation rule *)
      val show : Component -> string,
      val showActivity : Activity -> string,
      val showComponent :
        (Activity -> string) -> (Rate -> string) -> (Identifier -> string) ->
        Component -> string,
      val showIdentifier : Identifier -> string,
      val showRate : Rate -> string,
      val unfold : Identifier -> Component}                                   (* end of first Queue definition *)
  structure Queue2 :                                                           (* the second Queue definition *)
    {datatype Activity = (Activity/1, {con go12, go21 : Activity/1}),                 (* the type of activities *)
      datatype Component =                                                 (* the component combinator syntax *)
        (Component/1,
        {con $ : Identifier/1 -> Component/1,                                                     (* variable *)
          con * : (Activity/1 * Rate/1) * Component/1 -> Component/1,                               (* prefix *)
          con + : Component/1 * Component/1 -> Component/1,                                         (* choice *)
          con - : Component/1 list * Activity/1 list -> Component/1}),                         (* cooperation *)
      datatype Identifier = (Identifier/1,                                           (* the model identifiers *)
        {con S1023, S1122, S1221, S1320, System : Identifier/1}),
      datatype Rate = (Rate/1,{con mu1 : Rate/1, con mu2 : Rate/1}),                       (* the type of rates *)
      val Components : Identifier/1 list,
      val commute : Component/1 -> Component/1,
      val defn : Identifier/1 -> Component/1,
      val fold : Component/1 -> Identifier/1,
      val recUnfold : Component/1 -> Component/1,
      val rule : int -> Component/1 -> Component/1,                            (* apply a transformation rule *)
      val show : Component/1 -> string,
      val showActivity : Activity/1 -> string,
      val showComponent :
        (Activity/1 -> string) -> (Rate/1 -> string) ->
        (Identifier/1 -> string) -> Component/1 -> string,
      val showIdentifier : Identifier/1 -> string,
      val showRate : Rate/1 -> string,
      val unfold : Identifier/1 -> Component/1}                              (* end of second Queue definition *)
  val assert1, assert2, assert3, assert4 : bool,                                               (* four lemmas *)
  val theorem1 : bool}                                                                (* equivalence theorem *)
[assertion checked]
[assertion checked]
[assertion checked]
[assertion checked]
[theorem checked]
Exiting PEPA Term Kit                                                                       (* end of session *)
```

Fig. 4. Loading a proof script into the PEPA Term Kit

Acknowledgements: This work was supported by the Engineering and Physical Sciences Research Council (COMPA project, G/L10215) and by the DEGAS (Design Environments for Global ApplicationS) project funded by the FET Proactive Initiative on Global Computing.

References

1. P. Harrison and J. Hillston. Exploiting Quasi-reversible Structures in Markovian Process Algebra Models. *The Computer Journal*, 38(6), 1995. Special Issue: Proc. of 3rd Process Algebra and Performance Modelling Workshop.
2. J. Hillston. Compositional Markovian Modelling Using a Process Algebra. In W.J. Stewart, editor, *Numerical Solution of Markov Chains*. Kluwer, 1995.
3. J.M. Fourneau, L. Kloul, and F. Valois. Performance modelling of hierarchical cellular networks using PEPA. In *Proc. of 7th Process Algebra and Performance Modelling Workshop*, Zaragoza, 1999.
4. M. Sereno. Towards a Product Form Solution of Stochastic Process Algebras. *The Computer Journal*, 38(6), 1995. Special Issue: Proc. of 3rd Process Algebra and Performance Modelling Workshop.
5. J. Hillston and N. Thomas. A syntactical analysis of reversible PEPA models. In C. Priami, editor, *Proceedings of the Sixth Annual Workshop on Process Algebra and Performance Modelling*, pages 37–49, Nice, France, September 1998.
6. J. Hillston and N. Thomas. Product form solution for a class of PEPA models. *Performance Evaluation*, 35(3–4):171–192, 1999.
7. G. Clark and J. Hillston. Product form solution for an insensitive stochastic process algebra structure. *Performance Evaluation*, to appear in 2002.
8. J. Hillston. *A Compositional Approach to Performance Modelling*. Cambridge University Press, 1996.
9. S. Gilmore, J. Hillston, and M. Ribaudo. An efficient algorithm for aggregating PEPA models. *IEEE Trans. on Software Engineering*, 27(5):449–464, May 2001.
10. R. M. Burstall and J. Darlington. A transformational system for developing recursive programs. *Journal of the ACM*, 21(1):44–67, January 1977.
11. Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML: Revised 1997*. The MIT Press, 1997.
12. S. Gilmore and J. Hillston. Performance modelling in PEPA with higher-order functions. In N. Thomas and J. Bradley, editors, *Proceedings of the Sixteenth UK Performance Engineering Workshop*, pages 35–46, Durham, July 2000.
13. R. Milner, M. Gordon, and C. Wadsworth. *Edinburgh LCF*. Springer-Verlag, Berlin, 1979.