UNE IMP

EHE QMIPS CWI

INR UER

ZAR UTO

Quantitative Modeling in Parallel Systems

Detecting reversibility in Markovian Process Algebras

R W1.T1.1.v1

Madhu D K Bhabuta Peter G. Harrison Kamyar Kanani

Department of Computing

Imperial College

180 Queen's Gate, London SW7 2BZ

Distribution Level: All partners

Approved by: Peter G. Harrison

Abstract A methodology is developed to determine when a process defined in terms of a Markovian Process Algebra is a reversible Markov process and so has a product form solution. In particular, we show how to detect birth-death processes, tree-like extensions of these and examine an approach to detecting more general reversible processes through the Kolmogorov criteria applied to a minimal state transition graph. Finally, we indicate how to exploit the compositionality of process algebras to detect more complex reversible, and other product-form, processes formed from interactions of simpler processes already known to be reversible.

The QMIPS Project The QMIPS project is a collaborative research project supported by the CEC as ESPRIT-BRA project no 7269. It is being carried out by the following organisations: CWI (Amsterdam), EHEI (University of Paris V), Imperial College (London), INRIA (Sophia-Antipolis), University of Erlangen, University of Newcastle, University of Torino and University of Zaragoza.

1 Introduction

Recent advances in extending process algebras to include time and probablistic behaviour e.g. in [11], [5] have made it possible to use them as a performance modelling tool. Markov models have for long been widely used for this purpose but their direct specification is cumbersome and error-prone. However, when a Markov model has a *product form solution*, closed form results can be used efficiently. In particular, product-form solutions exist for Markov models which are reversible. In this paper we present a methodology to determine when the stationary Markov process associated with a given process algebra program is reversible and so has a product form solution.

We first introduce in the next section the algebra that we use although we emphasise that the methodology we present is applicable to any Markovian Process Algebra (MPA). Section 3 gives a brief introduction to reversibility in general and sketches how we may recognise systematically whether or not the Markov Process associated with a MPA program is reversible. Such detection can be straightforward when an MPA program is isomorphic to the definition of a reversible Markov process. However, when there are also "internal" instantaneous transitions—such as τ-actions in CCS but not only these—the structure of the underlying Markov chain becomes obscured and detection can be difficult. Such problems are addressed in section 4 and a detection algorithm is prescribed. The paper concludes by considering how the methodology can contribute to recognising more general product-form processes, for example those that are quasi-reversible.

2 Markovian Process Algebra

The algebra we use is based on CCS [10], with stochastic, timed extensions very like those in PEPA [5]. The methodology presented to recognise reversibility in Markovian Process Algebras is, however, generic and applicable to most MPAs.

The syntax and structural operational semantics we detail below, under the assumption that the reader is familiar with the concepts of process algebra. A detailed introduction is given in [10].

2.1 BNF notation for MPA

$$
\begin{array}{rcl}
<\mathrm{Vis}> & = & \mathrm{nil} \mid \mathrm{a} \mid \mathrm{b} \mid \mathrm{c} \mid \cdots \\
<\mathrm{Acts}> & = & \tau \cup <\mathrm{Vis}> \\
<\mathrm{Rate}> & = & \top \mid \alpha \mid \beta \mid \gamma \mid \cdots \\
<\mathrm{MActs}> & = & (\mathrm{Acts}, \mathrm{Rate}) \\
<\mathrm{Process}> & = & 0 \mid A \mid B \mid C \mid \cdots \\
 & \mid & <\mathrm{MActs}> . <\mathrm{Process}> \\
 & \mid & <\mathrm{Process}> + <\mathrm{Process}> \\
 & \mid & <\mathrm{Process}> \mid <\mathrm{Process}> \\
 & \mid & <\mathrm{Process}> [\mathrm{Vis}/\mathrm{Vis}] \\
 & \mid & <\mathrm{Process}> \backslash \{\mathrm{Vis}\}
\end{array}
$$

$\top$ is an undefined rate and 0 is the stopped process.

2.2 Operational Semantics for MPA

ActionPrefixing

$$\frac{}{(\mathrm{a}, \lambda).\mathrm{P} \stackrel{(\mathrm{a},\lambda)}{\rightarrow} \mathrm{P}}$$

Choice

$$\frac{\mathrm{P} \stackrel{(\mathrm{a},\lambda)}{\rightarrow} \mathrm{P}'}{\mathrm{P} + \mathrm{Q} \stackrel{(\mathrm{a},\lambda)}{\rightarrow} \mathrm{P}'} \qquad \frac{\mathrm{Q} \stackrel{(\mathrm{b},\mu)}{\rightarrow} \mathrm{Q}'}{\mathrm{P} + \mathrm{Q} \stackrel{(\mathrm{b},\mu)}{\rightarrow} \mathrm{Q}'}$$

Restriction

$$\frac{\mathrm{P} \stackrel{(\mathrm{a},\lambda)}{\rightarrow} \mathrm{P}'}{\mathrm{P}\backslash\mathrm{L} \stackrel{(\mathrm{a},\lambda)}{\rightarrow} \mathrm{P}'} \quad (a, \lambda) \neq \mathrm{L}$$

Relabelling

$$\frac{\mathrm{P} \stackrel{(\mathrm{a},\lambda)}{\rightarrow} \mathrm{P}'}{\mathrm{P}[\mathrm{f}] \stackrel{\mathrm{f}(\mathrm{a},\lambda)}{\rightarrow} \mathrm{P}'[\mathrm{f}]}$$

Parallel

$$\frac{\mathrm{P} \stackrel{(\mathrm{a},\lambda)}{\rightarrow} \mathrm{P}' \quad \mathrm{Q} \stackrel{(\overline{\mathrm{a}},\top)}{\rightarrow} \mathrm{Q}'}{\mathrm{P} \mid \mathrm{Q} \stackrel{(\tau,\lambda)}{\rightarrow} \mathrm{P}' \mid \mathrm{Q}'} \qquad \frac{\mathrm{P} \stackrel{(\mathrm{a},\top)}{\rightarrow} \mathrm{P}' \quad \mathrm{Q} \stackrel{(\overline{\mathrm{a}},\mu)}{\rightarrow} \mathrm{Q}'}{\mathrm{P} \mid \mathrm{Q} \stackrel{(\tau,\mu)}{\rightarrow} \mathrm{P}' \mid \mathrm{Q}'}$$

$$\frac{\mathrm{P} \stackrel{(\mathrm{a},\lambda)}{\rightarrow} \mathrm{P}' \quad \mathrm{Q} \stackrel{(\overline{\mathrm{a}},\mu)}{\rightarrow} \mathrm{Q}'}{\mathrm{P} \mid \mathrm{Q} \stackrel{(\tau,\min(\lambda,\mu))}{\rightarrow} \mathrm{P}' \mid \mathrm{Q}'}$$

$$\frac{\mathrm{P} \stackrel{(\mathrm{a},\lambda)}{\rightarrow} \mathrm{P}'}{\mathrm{P} \mid \mathrm{Q} \stackrel{(\mathrm{a},\lambda)}{\rightarrow} \mathrm{P}' \mid \mathrm{Q}} \qquad \frac{\mathrm{Q} \stackrel{(\mathrm{b},\mu)}{\rightarrow} \mathrm{Q}'}{\mathrm{P} \mid \mathrm{Q} \stackrel{(\mathrm{b},\mu)}{\rightarrow} \mathrm{P} \mid \mathrm{Q}'}$$

The $\mid$ operator is associative and commmutative. When 2 processes are placed in parallel, it is unclear what is meant by the rate at which the communication takes place. We have chosen a semantics where the rate of synchronisation between two actions is that of the slower one [6].

3 Reversible Markov Processes

The Markov process is the basis of many modelling methodologies and central to it is the concept of state and state transitions. The state of a physical system at any given time may be described by a set of values of measurable properties of that system. State transitions describe the evolution of the system over time, e.g. a queue can often be expressed in terms of its length and transitions between the different lengths are determined by events—arrivals of customers and service by the server. The state and state transitions thus fully describe the system.

A Markov Process is reversible if and only if the probability flux between any pair of states is the same in both directions. More formally, there must exist a collection of positive numbers, $\pi(j)$ summing to unity that satisfy the *detailed balance condition*:

$$\pi(j)\, q_{jk} = \pi(k)\, q_{kj}$$

where q_{jk} is the instantaneous transition rate from state j to state k. Kolmogorov's criterion states that a Markov process is reversible iff the product of transition rates, q_{ij} between states i, j satisfy

$$q_{12}q_{23}\cdots q_{k-1,k}q_{k,1} = q_{1k}q_{k,k-1}\cdots q_{32}q_{21}$$

for all state cycles $1, \ldots, k, 1$. A more complete account of reversibility can be found in [7].

3.1 Detection strategy

We propose to recognise reversible MPA programs, i.e. those with associated Markov chains that are reversible, by first defining a mapping for processes to states and then determining the state transition matrix via the given MPA programs. The methodology we will use is :-

1. Specify a MPA program along with a function that maps a process to its associated Markov state, if there is one, else returns 'undefined'.
2. Construct the derivation tree of the processes in the given MPA program, that is, unfold it.
3. Apply the process $\rightarrow$ state function to the (process) nodes in the derivation tree to determine the state transition graph of the Markov process.
4. Analyse the state transition graph and test it for reversibility as follows:-
   (a) Check that all the transitons are bidirectional, i.e. if there is an arc i $\rightarrow$ j there is also an arc j $\rightarrow$ i.
   (b) If there are no cycles (of length more than two) in the graph, then the graph is
      - a linear structure *or*
      - a tree structure

      and is reversible trivially.
   (c) If there are cycles in the graph then, following the approach of [8]:
      i. Create the spanning tree of the graph.
      ii. Identify the edges in the graph that are not in the spanning tree; these form the *fundamental circuits* of the graph.
      iii. Check Kolmogorov's criterion for each of the fundamental circuits.

      The fundamental circuits form a basis vector for the graph, so if each of the fundamental circuits is found to satisfy Kolmogorov's Criterion, then the graph represents a reversible process.

The state transition graph obtained could be either finite or infinite. In the finite case, the above method would detect a linear, tree-shaped or cyclic reversible process. In the infinite case some form of inductive analysis is necessary to identify simple unbounded reversible processes. Alternatively, from a definition given in terms of recursion equations, an equivalent definition could be sought in terms of higher-order functions. These can then be analysed more simply for reversibility, analogously to algebraic program transformation techniques [3]. In this way a pool of "basic" reversible MPA templates can be assembled.

The important property of compositionality of process algebras can now be employed. Successively larger reversible models can be created from smaller ones, starting with the pool of

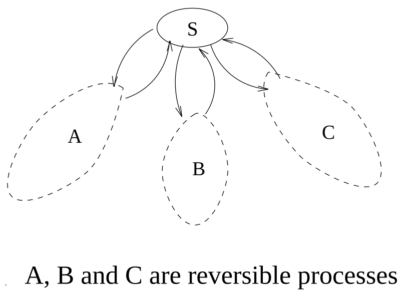

Figure 1: Composition of simple reversible structures

processes that have been recognised as reversible, and linked in such a manner as to preserve the reversibility property. In figure 1, graphs of the three reversible processes A, B, C are combined such that the detailed balance condition must hold on the arcs connecting each to the state S—only one in each direction—by considering an appropriate contour across which in- and out-flux must balance. Thus this more complex process, built from 3 reversible sub-processes, is also reversible since Kolmogorov's criterion is preserved; every cycle is built from smaller cycles satisfying the criterion. In this way, we can recognise ever more reversible processes and possibly add them to the "pool". Indeed, we can apply this methodology with just linear birth-death processes in the pool to deduce that single step, tree-like processes are also reversible.

For more complex combinations of reversible sub-processses, i.e. when a new cycle is created which is not formed solely out of known reversible cycles, the composite program has to be checked using the above analysis of fundamental circuits. However, given knowledge of the (reversible) fundamental circuits of the sub-processes, the analysis of those in the whole new graph can be simplified for particular types of interaction between these sub-processes. We will not address this topic further here, however.

Ideally, we would like to be able to recognise classes of MPA programs *syntactically* by working directly with the process algebra. Having identified a suitable pool of basic processes—at source level, but via semantic analysis at the level of the Markov graph—compositionality will facilitate this.

4 Analysis

4.1 The Derivation Graph

The semantics for MPA defines a labelled transition system that enables one to describe the evolution of an MPA program in the form of a *derivation tree*. A derivation tree is a directed, connected, acyclic structure. Each node represents a process. The children are those processes that the parent process can evolve into by one action, visible or otherwise. The arc connecting a parent and child node is labelled with the action that caused the corresponding process transition. This tree may be collapsed into a more general *derivation graph* if all syntactically identical nodes are grouped together. This provides a more compact notation for the derivation tree and in some instances leads to a graph that is actually isomorphic to the required Markov chain. We can formally define the derivation tree as the triple (V, E, L) where V is the set of vertices, representing processes, of the graph, E is a relation of type $Process \times Process$ and L is a labelling function of type $Process \rightarrow Process \rightarrow Event$ such that

$$\begin{aligned} EPQ &\Leftrightarrow P \xrightarrow{(a,r)} Q \\ L(PQ) = (a, r) &\Leftrightarrow P \xrightarrow{(a,r)} Q \end{aligned}$$

and $\mid E^{-1}Q \mid \leq 1$ where $E^{-1} : Process \longrightarrow \{Process\}$

The final constraint, that E^{-1} is a function, ensures that each node has only a single parent. This ensures the tree structure. Relaxing this constraint produces the definition for a derivation graph.

For example, if we define an M/M/1 queue by:

$$\begin{aligned} Q_0 &= (arr, \lambda).Q_1 \\ Q_n &= (arr, \lambda).Q_{n+1} + (dep, \mu).Q_{n-1}, n > 0 \end{aligned}$$

we obtain the derivation graph shown in Figure 2

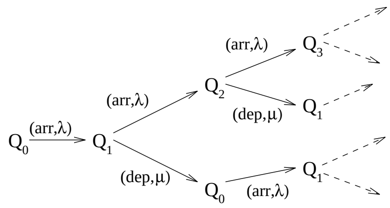

Figure 2: Derivation tree for simple queue

We can identify identical states which we can use to fold the tree to give us a derivation graph as in Figure 3.

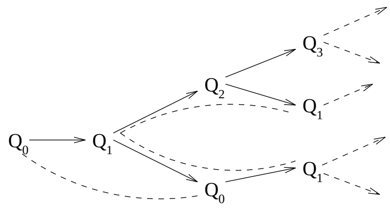

Figure 3: Identify identical states

Thus the tree can be folded along the dotted lines to give the following (familiar looking) graph as in Figure 4.

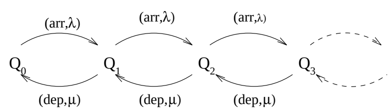

Figure 4: Resulting derivation graph

The size of the derivation tree may be unbounded, such as for the queue example above. This would be the case as well for any other non-terminating or recursive process. This may present us with a problem in implementation of an algorithm that relies on the direct analysis of the derivation tree. There are ways around this problem; for example one could choose a lazy functional programming language for the basis of the implementation. The graph corresponding to an unbounded tree may itself be bounded and instead contain a cycle. However, the finite nature of the graph would simplify its analysis if we ensure that nodes that have been analysed are marked as such.

4.2 The Mapping Function

We now introduce a function F of type $Process \rightarrow num$ that maps a process algebra program to an associated (numerical) state of some underlying continuous time Markov chain with countable state space. It is a partial function in that some processes may not have a defined Markov state associated with them. As we have already discussed, this is typically true of processes involved in sequences of immediate τ actions. These represent internal, instantaneous states in the Markov chain, and are therefore not wanted when analysing steady-state behaviour. Moreover, there may be some detail in the definition of a process that is of no interest in the Markov analysis. For example, the explicit occupancy of each of a set of buffers, where we are really only interested in the *number* of occupied buffers.

The process of choosing the function F could be automated in an MPA with an extended form of weak bisimulation equivalence. This would have to equate states not only up to τ-actions but also *any* internal actions which occur at infinite rate. Moreover, the resulting set of equivalence classes would have to be isomorphic to the desired state space. This would not necessarily be the case, as noted above.

By applying the function to a node in the tree and to its immediate children, we can determine the states that are the immediate transitions out of any current Markovian state, i.e we can define the associated Markov chain.

Take as an example, a section of the derivation tree for our simple queue shown in Figure 5. Here we have $F(Q_n) = n\ for\ all\ n \geq 0$.

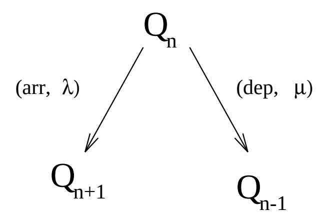

Figure 5: Section of derivation tree for simple queue

We can apply the state function to all nodes in this tree segment. The result is that if we are in a Markovian state n, upon the event (arr, λ) occuring, we move to Markovian state $n+1$ and upon event (dep, μ) we move to state $n-1$ (for n $>$ 0). We rely on the fact that actions in our MPA are indivisible. They are either single actions or synchronisations. This precludes the possibility that our system could evolve through several Markov states during the course of a single MPA transition.

More formally, the Markov transition graph corresponding to the derivation tree (V,E,L) is given by (V_M, E_M, L_M) where V_M is the set of states defined by $V_M = \{s \mid s = Fp, p \in V\}$, E_M is the edge relation of type $state \times state$ defined by $E_M RS \Leftrightarrow EPQ \ \wedge \ FP = R \ \wedge \ FQ = S$ and L_M is the labelling function of type $state \rightarrow state \rightarrow rate$ defined by $L_M RS = \sum_{i:(a,x_i)\in\{LPQ \ \mid \ FP=R, FQ=S\}} x_i$. Here we have to sum all of the rates going from one state into another to get the total flow between the two states. Since we are dealing with exponentially distributed rates we can just sum them due to the superposition property.

This definition is actually not complete since during analysis we may be faced with the situation that a given node has an undefined Markov state associated with it. This means that the application of F to that process should also be undefined. In this situation we can conclude that the system is in an *implementation specific transient state.* The process is evolving through a stage which is needed to allow implementation of the system. Such a transient state can be discarded and replaced by its children, provided its holding time is zero.

Consider an implementation of a birth-death process:

$$\begin{aligned}
C &= (arr, \lambda).(C^\frown C) + (dep, \mu).D \\
D &= (\overline{d}, \top).C + (\overline{z}, \top).B \\
B &= (arr, \lambda).(C^\frown B) + (z, \top).B \\
System &= C^\frown C^\frown \ldots C^\frown B
\end{aligned}$$

where $P^\frown Q$ represents $(P \mid (Q[d\backslash dep, \top\backslash\mu])\backslash\{arr\})\backslash\{d, z\}$. This defines a system which is the same system as the 'stack' defined in [10] of C processes linked together to form a chain tailed off by B processes. Let us assume we are interested in the number of C processes linked together. Informally looking at the system we see that upon an (arr, λ) event, the lead C process (the only process that can accept the event) splits into two, increasing the length of the chain by one. However, upon a (dep, μ) event the leading C process becomes a D process which can now synchronise with the second C process in the chain to swap places and so on down the chain until it comes to a B process which will absorb it, effectively decreasing the total number of C processes in the chain by one. We therefore define the state function thus :

$$\begin{aligned}
F(C^\frown Rest) &= 1 + F Rest \\
F(B^\frown Rest) &= 0 \\
FB &= 0
\end{aligned}$$

Thus the function recursively counts the number of C processes in the chain. Note that there is no mention of process D in the definition for F. Whilst the D process is rippling down the chain after a (dep, μ) event the state of the system is undefined. Therefore, whilst the D process is rippling down the chain, we would like to think that our system is moving instantaneously through a number of implementation specific transient states. In an ideal definition for the system, the length of the chain would decrease by one in just one transition but this is not the case here.

Moreover, situations can arise within the system that we do not address here. For example, whilst the D processes are rippling down the chain, the leading C process can still continue to accept (arr, λ) or (dep, μ) events. This confuses the system beyond our interest but one should be aware of this shortcoming in the definition. [1]

The definitions we have thus given for the Markov graph in terms of the definition graph therefore have to be modified to take these transient states into account.

Thus, we give new definitions for V_M, E_M and $\mathrm{L_M}$

$$\begin{aligned}
V_M &= \{s \mid s = Fp, p \in V\}\backslash\{\perp\} \\
E_M RS &\Leftrightarrow EPQ \;\wedge\; FP = R \;\wedge\; FQ = S \\
&\quad\vee\; E^*PQ \;\wedge\; \forall Z[E^*PZ \;\wedge\; E^*ZQ \Rightarrow FZ = \perp] \\
L_M RS &= \sum_{i:(a,x_i)\in\{LPQ \;\mid\; FP=R, FQ=S\}} x_i
\end{aligned}$$

where E^* is the transitive closure of E, i.e. $E^*PQ \Leftrightarrow P \longrightarrow^* Q$. Thus V_M has the undefined state removed from it and two states in the Markov graph have an edge between them if there is a path from one process, corresponding to the origin state, to a process with corresponding target state, in the derivation tree such that either the path is single step or that all intermediate processes have no state associated with them.

5 Topological structures

5.1 Linear structure

The canonical example of a system that has a linear structure is the birth-death process, in which the population increases on a birth, at rate λ and decreases on a death, at rate μ. This example has been covered by the M/M/1 queue specification given in section 4.1.

[1] Actually this can be ignored if we employ the rule that $\top$ rated events occur in preference to ones with defined rates; similar to immediate and stochastic transitions in GSPNs

Of course, many linear systems are not birth-death processes since they have multi-step jumps. In this case the Markov graph contains cycles and the Kolmogorov criteria must be checked. In simple cases, e.g. where the jumps follow a regular pattern specified inductively along with their rates, such checking is straightforward, involving only one or two generic cycles.

5.2 Tree-shaped structure

Consider a model of a network distribution protocol. Messages are sent to each child node from the root. Replies are recieved from each of the child nodes upwards. The permormance measure of interest in the model is the load at each of the nodes in the network. The rate at which messages are sent from the root node to its children is λ and the rate at which the results are received back is μ. The process algebra for the system would look like:

$$
\begin{aligned}
\mathrm{Node(1,1)} &= (\mathrm{send},\lambda).(\mathrm{Node(2,1)}+\mathrm{Node(2,2)}) \\
\mathrm{Node(n,x)} &= (\mathrm{send},\lambda).(\mathrm{Node(n+1,2x-1)}+\mathrm{Node(n+1,2x)}) \\
&+ (\overline{\mathrm{resend}},\mu)(\mathrm{Node(n-1,}\frac{1}{2}\mathrm{x}+\frac{1}{2}),\mathrm{if\ x\ mod\ 2} \neq 0 \\
& \quad (\overline{\mathrm{resend}},\mu)(\mathrm{Node(n-1,}\frac{1}{2}\mathrm{x}),\mathrm{otherwise}
\end{aligned}
$$

The function mapping process to state is defined in the obvious way:

$$F(\mathrm{Node}(n,x)) = (n,x)$$

since the Markov state is again isomorphic to the MPA process. The resulting state transition graph is then Figure 6. Obviously this definition can be extended recursively to yield various

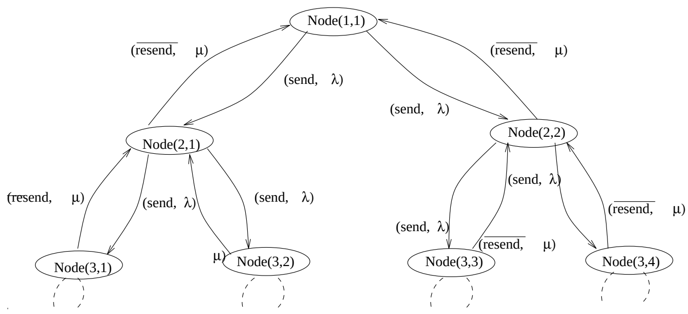

Figure 6: Tree-shaped graph

tree-structures of arbitrary size, and reversibility is immediate since all transitions are single step.

5.3 Cyclic structure

Consider now a ring-based network protocol, in which messages can travel to and fro, from sender to receiver. We present 2 alternative process algebras [2] to represent the process.

1. The communication is illustrated by the following definition for a single cycler:

$$
\begin{aligned}
C &= a.C' + c.C' \\
C' &= b.C + d.C \\
C_i &= C[f_i] \\
C'_i &= C'[f_i] \\
\mathrm{where} f_i &= (a_i/a, \overline{a_{i+1}}/b, \overline{b_i}/d, b_{i+1}/c).
\end{aligned}
$$

[2] all addition is mod n where n is the number of nodes in the ring

Then a ring of three cyclers is $C^3 = (C_1|C_2|C_3)\backslash\{a_1, a_2, a_3, b_1, b_2, b_3\}$. To permit interaction with the outside world we use a more complex process with "probes" a, a' (we also include rates) :

$$\begin{aligned} C &= (a,\mu).C' + (a',\lambda).C'' \\ C' &= (b,\mu).C'' + (b',\lambda).C \\ C'' &= (c,\mu).C + (c',\lambda).C' \end{aligned}$$

To form the ring of 3 cyclers, as in [10]:

$$\begin{aligned} C_i &= C[f_i] \\ C_i' &= C'[f_i] \\ C_i'' &= C''[f_i] \end{aligned}$$

$$\begin{aligned} \text{where } f_i = \; & \{((a_i,\mu_i)/(a,\mu), (b_{i+1},\mu_i)/(b,\mu), \\ & (\overline{b_i},\mu_i)/(c,\mu)), (a_i',\lambda_i)/(a',\lambda), \\ & (b_{i+1}',\lambda_i)/(b',\lambda), (\overline{b_i'},\lambda_i)/(c',\lambda)\} \end{aligned}$$

$$\text{and finally } C^3 = (C_1|C_2''|C_3'')\backslash\{b_1, b_2, b_3, b_1', b_2', b_3'\}$$

2. Alternatively, we can simply define

$$\begin{aligned} C_i &= (x_i,\lambda_i).C_{i+1} + (y_i,\mu_i).C_{i+2} \\ Ring &= C_1 + C_2 + C_3 \end{aligned}$$

We obtain the state transition graph, Figure 7. Since this contains cycles—actually only one

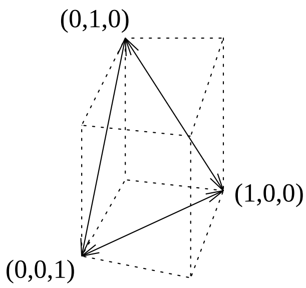

Figure 7: Graph for a 1 customer case

of length greater than two—we apply Kolmogorov's criterion to obtain the reversibility condition $\lambda_1\lambda_2\lambda_3 = \mu_1\mu_2\mu_3$.

6 Algorithm

We now present an algorithm to deduce the Markov process associated with an MPA system. Initially, all we are presented with is the source for the MPA system and the definition for the state function F. From this we would like to systematically derive our Markov graph.

In our terminology, we have the set V, the relation E, and the functions L and F. From these we would like to derive V_M, E_M and L_M. Towards this goal, we can define two new functions E_f and E_{Mf} of type $Process \rightarrow \{Process\}$ and $state \rightarrow \{state\}$ respectively. These correspond to the relations E and E_M i.e. $E_f P = \{Q \mid EPQ\}$ and similarly $E_{Mf} P = \{Q \mid E_M PQ\}$

We next define T, which is similar to E_f but only relates processes with defined states:

$$\begin{aligned} TP &= \{Q \mid Q \in E_f P \ \wedge \ FQ \neq \perp \\ &\vee \ \ Q \in E_f^* P \ \wedge \ \forall Z[E_f^{-1*} Z = P \ \wedge \ E_f^{*-1} Q = Z \Rightarrow FZ = \perp]\} \end{aligned}$$

Then we can define E_{Mf} by $E_{Mf} R = \{S \mid FP = R \ \wedge \ FQ = S \ \wedge \ Q \in TP\}$ since by the definition of T, we are guaranteed the constraint on paths. Thus we can define the function E_{Mf} if we can find the set $\{S \mid FP = R \ \wedge \ FQ = S \ \wedge \ Q \in TP\}$ for a given R. It can be seen that this can be accomplished by the function $(\mathsf{map}\ (F)) \circ (\mathsf{fold}\ \cup) \circ (\mathsf{map}\ (T)) \circ F^{-1}$ which when presented with a state derives all processes that map onto that state by application of F, finds all processes with defined states that the process can immediately evolve into and returns the states associated with those processes. Thus the function basically finds the states out of a given Markov state. We can use the function as the definition for E_{Mf} and as a basis for an algorithm for its construction. The construction for the function L_M is similar. The only problem with this definition is that for most cases, F^{-1} is not easy or even possible to find. In this case we can only piecewise build up E_{Mf} by examining processes, P, such that $FP = R$. We are not guaranteed completeness, i.e. to find all of the state transitions out of a given Markov state since we cannot know if we have examined all processes that map onto a given state for their contribution to the graph. Only pathological systems seem to possess the property that for any two processes, P and Q, then if $FP = FQ$, we do not get identical state transition information from both, this means that usually only examining one of those processes will yield as much information as examining all of them.

The algorithm to piecewise build up a Markov graph when we cannot find F^{-1} is presented as a functional program in the style of Miranda [3]. It works by simply searching the derivation tree of a given process to find processes with defined states, thus showing which states can be reached in one Markovian transition from the state defined by the initial process.

6.1 Example : The M/M/1 queue

We might define a single server queue at the "implementation level" as a server and buffer process working in tandem:

$$\begin{aligned} C &= (\text{arrival}, \top).(C \frown C) + (\text{service}, \top).D \\ D &= (\text{arrival}, \top).C \\ S &= (\overline{\text{service}}, \top).(\text{departure}, \mu).S \\ Q &= (D|S)\backslash\{s\} \end{aligned}$$

A C process can be thought of as being a single customer. On an arrival event it becomes chained to another C process, thus reflecting another arrival. Thus the buffer of customers waiting to be served can be viewed as the chained list of C processes. A D process can be thought of as the space left after a customer has been served. On an arrival event it becomes a C process. An S process can be thought of as being the server part of the queue. It accepts service events from the buffer and sends departures to the environment.

Again, being interested in queue length, we can thus define the state function as:

$$\begin{aligned} F_1(C \frown Rest) &= 1 + (F_1 Rest) \\ F_1 C &= 1 \\ F_1 D &= 0 \\ F_2 S &= 0 \\ F_2((d,\mu).S) &= 1 \\ F((P_1|P_2)\backslash L) &= F_1 P_1 + F_2 P_2 \end{aligned}$$

[3] Miranda is a trademark of Research Software Ltd.

Here we have a more complex derivation tree thus we will present it as a group of labelled transitions as opposed to a data-structure. let A^n represent $A|A|\cdots$ n times starting with process Q at the root we have the labelled transition : $Q \xrightarrow{(\tau,\lambda)} (C|S)\backslash\{s\}$
this gives us transitions $Q = (0, [(1, \lambda)])$ Thus we now have state 0 and its transition out to state 1 at rate λ as part of the Markov graph.

We can analyse the new process further to build up the rest of the Markov transition graph. The next level in the derivation tree has just the node for the process $(C|S)\backslash\{s\}$, with labelled transitions :

1. $(C|S)\backslash\{s\} \xrightarrow{(\tau,\lambda)} ((C^2)|S)\backslash\{s\}$
2. $(C|S)\backslash\{s\} \xrightarrow{(\tau,\top)} (D|(\text{departure}, \mu).S)\backslash\{s\}$

giving us the transition : $(1, [(2, \lambda), (1, \top)])$ we now also have the transition from state 1 to state 2 at rate λ. We also get an instantaneous transition from state 1 back to itself (a customer moving from the buffer into service). This does not contribute to the Markov transition graph, since we are looking at the underlying *continuous time* Markov chain.

Proceeding to examine the next level of the graph :

1. $(C^2|S)\backslash\{s\} \xrightarrow{(\tau,\lambda)} ((C^3)|S)\backslash\{s\}$
2. $(C^2|S)\backslash\{s\} \xrightarrow{(\tau,\top)} ((C^\frown D)|(\text{departure}, \mu).S)\backslash\{s\}$
3. $(D|(\text{departure}, \mu).S)\backslash\{s\} \xrightarrow{(\tau,\mu)} (D|S)\backslash\{s\}$

giving us the transitions :

1. $(2, [(3, \lambda), (2, \top)])$
2. $(1, [(0, \mu)])$

We could proceed in this manner to constructively build up the remaining Markov graph to any desired size (since it has an infinite number of nodes). However, after analysis to this point we can 'see' how the queue process is developing and thus choose to examine it in a 'general configuration' in order to give us an inductive definition for the remaining Markov graph. Indeed, theorem proving techniques exist that mechanise this process.

1. $((C^n{}^\frown D^j)|S)\backslash\{s\} \xrightarrow{(\tau,\top)} ((C^{n-1}{}^\frown D^{j+1})|(\text{departure}, \mu).S)\backslash\{s\}$
2. $((C^n{}^\frown D_j)|S)\backslash\{s\} \xrightarrow{(\tau,\lambda)} ((C^{n+1}{}^\frown D_j)|S)\backslash\{s\}$
3. $((C^n{}^\frown D^j)|(\text{departure}, \mu).S)\backslash\{s\} \xrightarrow{(\tau,\mu)} ((C^n{}^\frown D^j)|S)\backslash\{s\}$

examining these we now get the following transitions defining the Markov graph:

1. gives us no useful information, i.e. $(n, n, \top)$
2. gives us $(n, n+1, \lambda)$
3. gives us $(n+1, n, \mu)$

Thus we find that we have the same underlying Markov process as for the more simply defined queue in section 4.1. Its reversibility is immediate.

7 Conclusion

Product-form stochastic models provide an efficient means for performance modelling and so they are much sought after, either as an exact representation or an approximation for physical systems. Markovian process algebra has good expressive power but in general leads to inefficient solutions based upon the direct solution of the underlying Markov chain. We have begun to develop a methodology for detecting and synthesising product form Markov models by considering their simplest subset, reversible processes.

Although only a small fraction of product form processes are reversible, the compositionality of process algebras offers the prospect of mechanically identifying a large proportion of them. Moreover, by developing ways of syntactically combining processes to preserve their *product form* property, a far greater proportion of such processes will become identifiable. This idea has been applied to *quasi-reversible* processes, which define a much more substantial subset of product form models, in [4]. Our work on reversibility will provide an important pool of base cases upon which to build a significant collection of processes with easily computable product form solutions. Noting that neither reversibility nor quasi-reversibility implies the other property, all that will be necessary will be to determine when any process we recognise as reversible is also quasi-reversible. This is a relatively simple task; e.g. the birth-death process is quasi-reversible by virtue of its Poisson arrival process.

References

[1] Harrison P G and Patel N M. *Performance modelling of Communication Networks and Computer Architectures.* Addison-Wesley, 1993.

[2] Harrison P G and Strulo B. *Process Algebra for Discrete Event Simulation.* Workshop on Formalisms, Principles and state-of-the-art, Erlangen, September 1993.

[3] Grant-Duff Z and Harrison P G. *Homomorphisms for parallel program synthesis.* Research Report, Department of Computing, Imperial College, 1995.

[4] Harrison P G and Hillston J. *Exploiting quasi-reversible structures to find product form solutions in Markovian process algebra models.* Proceedings of the 3rd workshop on Process Algebras and Performance Modelling, June 1995.

[5] Hillston J. *A Compositional approach to Performance Modelling* PhD thesis, University of Edinbrough, 1994.

[6] Hillston J. *The Nature of Synchronisation* Proceedings of the 2nd workshop on Process Algebras and Performance modelling, August 1994.

[7] Kelly F. P.*Reversibilty and Stochastic Networks* John Wiley and Sons, 1978.

[8] Koukoulidis V. N. and Comeau M. A. *Graph-Theoretic Characterisation and Modeling of Product-Form Queueing Networks* Preliminary report, Concordia University, Canada.

[9] Lamport L. LaTeX. Addison-Wesley, 1986.

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

[11] Strulo B. *Process Algebra for Discrete Event Simulation* PhD thesis, Imperial College, 1993.