Using Markovian Process Algebra to Specify Interactions in Queueing Systems

Nigel Thomas and Jane Hillston

Department of Computer Science
University of Edinburgh

{*nat,jeh*}*@dcs.ed.ac.uk*

Abstract: The advantages of using stochastic process algebra to specify performance models are well-documented. There remains, however, a reluctance in some quarters to use process algebras; this is particularly the case amongst queueing theorists. This paper demonstrates the use of a Markovian process algebra to represent a wide range of queueing models. Moreover, it shows that the most common interactive behaviours found in queueing systems can be modelled simply in a Markovian process algebra, and discusses how such specifications can lead to an automated numerical solution method. An additional objective of this work is to specify queueing models in a form which facilitates efficient solution techniques. It is intended that characterising the syntactic forms within the Markovian process algebra which give rise to these efficient solution methods will subsequently allow the methods to be applied to a much wider range of models specified in the process algebra.

Keywords: Markovian process algebra; queueing models; solution techniques

1 Introduction

A common misconception is that a process algebra is not a good way to specify queueing problems. One reason for this is that it is very easy to informally specify queueing problems using a common understanding amongst queueing theorists. This is achieved by relating any new problem to well known existing models, with the 'new' feature described in natural language. In doing this any notion of formally specifying a given problem is left until the underlying balance equations are given. There are a number of problems with this informal approach. Firstly, the lack of a higher level formal specification might lead to misunderstandings, particularly by those outside the peer group. This leads to an elitist view of performance modelling and a narrowing of viewpoints. Secondly, it is difficult to form relationships between different forms of models, both in terms of equivalence and potential approximate solutions. This in turn leads to a fragmented approach to solution where related models are often studied in isolation.

Another reason for the above misconception is that model specifications written in a process algebra are often lengthy and not always intuitive to those unfamiliar with their use. As will be demonstrated here, most actions in queueing systems can be defined recursively, hence the first objection cannot really hold. The second objection is a result of any shift in paradigm and cannot be discounted. However we have aimed to present a consistent approach to the specification of queueing systems that should provide a coherent reference point for any queueing theorist.

The advantages of using a stochastic process algebra to specify performance models are well documented (see Hillston [7] for example). In brief, a process algebra allows models to be compared e.g. for equivalence; to be analysed e.g. to reveal deadlocks or reducible structures; to be constructed at a higher level of abstraction, possibly by designers, rather than performance modellers; and in some cases to be solved automatically, either exactly or through the construction of approximations.

In the following sections we will give a brief overview of the Markovian process algebra PEPA, introduce some of the common types of interaction found in queueing systems, specify models involving these interactions and finally show how the structure of these models can be used to aid their numerical solution.

2 PEPA

In this paper we have specified models in PEPA (Performance Evaluation Process Algebra) [7], however we might equally well have used TIPP [4] or any other process algebra similarly enhanced to support performance modelling. PEPA is a Markovian Process Algebra, meaning that it only supports actions that occur with rates that are negative exponentially distributed. Specifications written PEPA represent Markov processes and can be mapped to a continuous time Markov chain (CTMC) for analytic or numerical solution. As such the specifications given here are limited to queues of the $M/M/n$ family.

The basic elements of PEPA are *components* and *activities*, corresponding to *states* and *transitions* in the underlying Markov process. Each activity has an *action type* (or simply *type*). Activities which are private to the component in which they occur are represented by the distinguished action type, τ. The duration of each activity is represented by the parameter of the associated exponential distribution: the *activity rate* (or simply *rate*) of the activity. This parameter may be any positive real number, or the distinguished symbol $\top$ (read as *unspecified*). Thus each activity, a, is a pair (α, r) where α is the action type and r is the activity rate. In a queueing system a component might be the number of jobs in a queue and the activities might be the arrival and service instances that change the number of jobs in a queue.

2.1 Syntax and informal semantics

PEPA provides a small set of combinators. These allow expressions, or terms, to be constructed defining the behaviour of components, via the activities they undertake and the interactions between them. The combinators, together with their names and interpretations, are presented informally below.

Prefix: $(\alpha, r).P$ Prefix is the basic mechanism by which the behaviours of components are constructed. The component carries out activity (α, r) and subsequently behaves as component P.

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

Cooperation: $P \underset{L}{\bowtie} Q$ The components proceed independently with any activities whose types do not occur in the *cooperation set* L. The activities not in L are *individual activities*. However, activities with action types in the set L require the simultaneous involvement of both components (*shared activities*). These activities are only enabled in $P \underset{L}{\bowtie} Q$ when they are enabled in both P and Q.

The published MPAs differ on how the rate of shared activities are defined [8]. In PEPA the shared activity occurs at the rate of the slowest participant. If an activity has an unspecified rate in a component, the component is passive with respect to that action type. This means that the component does not influence the rate at which any shared activity occurs. The cooperation combinator associates to the left but brackets may also be used to clarify the meaning. The parallel combinator $\|$ is used as shorthand to denote synchronisation with no shared activities, i.e. $P \| Q \equiv P \underset{\emptyset}{\bowtie} Q$.

Hiding: $P \backslash L$ The component behaves as P except that any activities of types within the set L are hidden, i.e. such an activity exhibits the unknown type τ and the activity can be regarded as an internal delay by the component. Such an activity cannot be carried out in cooperation with any other component: the original action type of a hidden activity is no longer externally accessible; the duration is unaffected.

Constant: $A \stackrel{\text{def}}{=} P$ Constants are components whose meaning is given by a defining equation; $A \stackrel{\text{def}}{=} P$ gives the constant A the behaviour of the component P. This is how we assign names to components (behaviours). There is no explicit recursion operator but components of infinite behaviour may be readily described using sets of mutually recursive defining equations.

The action types which the component P may next engage in are the current action types of P, a set denoted $\mathcal{A}(P)$. This set is defined inductively over the syntactic constructs of the language. For example, $\mathcal{A}(P + Q) = \mathcal{A}(P) \cup \mathcal{A}(Q)$. The activities which the component P may next engage in are the current activities of P, a multiset denoted $\mathcal{A}ct(P)$. When the system is behaving as component P these are the activities which are enabled. Note that the dynamic behaviour of a component depends on the number of instances of each enabled activity and therefore we consider multisets of activities as opposed to sets of action types. For any component P, the multiset $\mathcal{A}ct(P)$ can be defined inductively over the structure of P, as for $\mathcal{A}(P)$.

2.2 Execution strategy

A race condition governs the dynamic behaviour of a model whenever more than one activity is enabled. This has the effect of replacing the non-deterministic branching of classical process algebra with probabilistic branching. The probability that a particular activity completes is given by the ratio of the activity rate to the sum of the activity rates of all the enabled activities. Any other activities which were simultaneously enabled will be interrupted or aborted. The memoryless property of the exponential distribution makes it unnecessary to record the remaining lifetime in either case.

A much fuller explanation and a specification of the operational semantics of PEPA is given in [7].

3 The Nature of Interactions in Queueing Models

The literature on queueing models is perhaps the most extensive of any area of performance modelling. So many forms of queueing model have been studied that it would be virtually impossible to catalogue every type of interaction, but an attempt has been made here to summarise the most common forms. In general most performance measures of interest in queueing models are derived from the steady state queue size probabilities. Such measures might include relatively simple metrics such as average response time and job loss, as well as more complicated measures involving tail distributions, for instance the time that 95% of jobs complete within. In all these cases the aim of any model solution is generally to derive the steady state queue size probabilities in some form.

In general interactions can occur between two or more queues, between a given queue and another process, or a combination of both of these. These interactions may affect the service process, the jobs in a queue or the routing of jobs into a particular queue. For example such interactions include:

breakdowns (or vacations) : a server breakdown may be regarded as an interaction between the queue and an external process; indeed, in some server vacation models some external process is directly responsible for the absence of the server. The impact of the breakdown may go beyond the suspension of the service process and include, for example, the loss of jobs within the queue or the rerouting of subsequent arrivals. Several models of this type are presented in Section 4.1.

state-dependent routeing : when a job may be directed to one of a number of possible queues depending on their current status we have an indirect form of interaction between queues. Models of this type are discussed in Section 4.2.

prioritising different job types : if one job type has priority over another we have a form of interaction between jobs, or, in the case where the job types are held in separate queues, interaction between queues. An example of the latter form of model is given in Section 4.2.

jobs proceeding from one state of service to another : this is perhaps the most basic form of interaction within a queueing network, a *flow* or supply interaction: the service process associated with one queue forms the arrival process of another. Such models are discussed in Section 4.3.

In addition Davies [2] has identified two autonomous actions that can be performed by a customer (job); these are referred to as *jockeying* and *reneging*. Jockeying is where a customer transfers from one queue to another, presumably for some perceived advantage. The case where a customer leaves the system before service completion is referred to as reneging, which may be the result of a time-out. In the context of queueing models these behaviours may be captured as a form of interaction between jobs via the introduction of *negative customers* (see Harrison *et al.* [6] for example). In these models negative customers act as cancellation messages, removing "positive" customers from the queue. An example is discussed in more detail in Section 4.1.

4 PEPA Models of Queueing Systems

As stated earlier our aim is to present a range of queueing model examples represented in the Markovian process algebra PEPA in a consistent style. The advantages of adopting a consistent style should be immediately apparent. It allows us to readily compare models, to modify them and to extend them. In our chosen style each *stage* of a queueing model (normally referred to as a *node*) consists of a number of queues and a *scheduler*. Note that the server is not explicitly represented at all. The scheduler controls the arrivals and departures from each queue. Whenever possible the synchronised actions of the queues are outwardly passive and merely specify the legal set of arrivals and departures, e.g. arrivals cannot occur if the queues are full and departures cannot occur if the queue is empty, otherwise either may take place. We felt that it was desirable for consistency, clarity and ease of solution that no synchronised actions should be active in the queues (i.e. all rates of synchronised actions should be $\top$ in the queue components) but this proved to be a difficult condition to maintain, as demonstrated later.

The scheduler may have many states of operation. Within each state the actions performed on the queues and the transitions between scheduler states are listed and assigned rates. Although not necessary from a modelling point of view we have explicitly named each state of the scheduler: this aids the discussion of each model. The synchronisation of the queues and the scheduler over the given set of queue actions then defines the system. An example of this approach is given in Figure 1.

4.1 Models of a Single Queue

The example shown in Figure 1 is in fact a server vacation, or breakdown, model. The scheduler states S_1 and S_0 correspond to the server being either working or broken and the transitions $OneToZero$ and $ZeroToOne$ represent failure and subsequent repair. In this case the only effect of a failure is for the server to stop working, during repair periods

$$Queue_0 \stackrel{\text{def}}{=} (arrival, \top).Queue_1$$

$$Queue_j \stackrel{\text{def}}{=} (arrival, \top).Queue_{j+1} + (service, \top).Queue_{j-1} \quad , \qquad 1 \leq j \leq N-1$$

$$Queue_N \stackrel{\text{def}}{=} (service, \top).Queue_{N-1}$$

$$S_1 \stackrel{\text{def}}{=} (arrival, \lambda).S_1 + (service, \mu).S_1 + (OneToZero, \alpha).S_0$$

$$S_0 \stackrel{\text{def}}{=} (arrival, \lambda).S_0 + (ZeroToOne, \beta).S_1$$

$$Queue_0 \underset{(arrival,service)}{\bowtie} S_1$$

Figure 1: A PEPA model of a queue with two operational states

(state S_0) jobs do not leave the queue, but continue to arrive. Clearly it is a simple step to consider failures causing a change in the rate of arrivals by replacing λ with λ_1 and λ_0 in S_1 and S_0 respectively, or even stopping arrivals completely by omitting that term from S_0.

A failure may have much larger effect than just the suspension of service, for example causing the loss of a single job or the entire queue. Many models of this kind have been considered in the literature (see Thomas [12] for a survey of closed form results for infinite length queues). If changes in the scheduler state affect the number of jobs in the queue it is necessary to introduce further states into the queue to account for this.

In Figure 2 a PEPA specification is given for a system where the entire queue is lost on failure. Here an additional state has been added to the queue to account for the repair period, passive failure and repair actions have been added to the queue and synchronised and arrivals have been disallowed in scheduler state S_0, otherwise the model is identical to that given in Figure 1.

It should be noted that since only the repair action takes place in state S_0, the scheduler could be specified simply as,

$$S_1 \stackrel{\text{def}}{=} (arrival, \lambda).S_1 + (service, \mu).S_1 + (failure, \xi).(repair, \eta).S_1$$

The same approach could also be applied to the queue state *Broken*. Whilst this gives rise to a perfectly valid and equivalent PEPA specification of the model, the use of the states S_0 and *Broken* add clarity when reading the specification. In addition it should also be noted that this model can just as easily be specified with the transitions between queue states as active and the scheduler may be entirely passive. In fact, it is not even

$$Queue_0 \stackrel{\text{def}}{=} (arrival, \top).Queue_1 + (failure, \top).Broken$$

$$Queue_j \stackrel{\text{def}}{=} (arrival, \top).Queue_{j+1} + (service, \top).Queue_{j-1} + (failure, \top).Broken$$

$$, \quad 1 \leq j \leq N-1$$

$$Queue_N \stackrel{\text{def}}{=} (service, \top).Queue_{N-1} + (failure, \top).Broken$$

$$Broken \stackrel{\text{def}}{=} (repair, \top).Queue_0$$

$$S_1 \stackrel{\text{def}}{=} (arrival, \lambda).S_1 + (service, \mu).S_1 + (failure, \xi).S_0$$

$$S_0 \stackrel{\text{def}}{=} (repair, \eta).S_1$$

$$Queue_0 \underset{(arrival,service,failure,repair)}{\bowtie} S_1$$

Figure 2: A PEPA model of an $M/M/1$ queue with breakdowns leading to total job loss

necessary to use a scheduler in this model. In short there are in general a large number of equivalent ways of specifying any given model in PEPA. However the aim here is to present a consistent and clear approach to that specification and so we have kept to one form only. The advantage of this form of model should become evident when attempting to extract a numerical solution from the model specification.

It is possible to specify such an effect of failure using a model more similar to that in Figure 1. In this case the term $(OneToZero, \top).Queue_0$ is added to every state in the queue. In addition the term $(ZeroToOne, \top).Queue_0$ is added to state $Queue_0$ and the *arrival* term is omitted from S_0. Such a specification gives rise to an identical Markov chain to that given by the model in Figure 2. This specification has the advantage of being slightly more concise, but is also perhaps slightly less obvious.

All of the effects of failure at a single server queue considered in [12] can be specified in this way. For instance an extra queue state could be introduced for the case where the entire queue is lost except the head job, $N-1$ broken states could be defined in the queue to consider the case where the head job only is lost or extra states could be added to the scheduler as well to account for more than one type of failure. All these cases can also be tackled as extensions of the model in Figure 1, as with the previous model where the entire queue is lost. The more broken states it becomes necessary to define, the more attractive this method becomes as it negates the need for extra state definitions.

In many situations it is useful to consider a single queue that has more than one available server. Each server may only serve one job at a time, and so n available servers will mean that at most n jobs can be served at any one time. All services are considered to be independent and the servers identical. Such a multi-server queue, or $M/M/n$ queue, is given in Figure 3.

$$Queue_0 \stackrel{\text{def}}{=} (arrival, \top).Queue_1$$

$$Queue_j \stackrel{\text{def}}{=} (arrival, \top).Queue_{j+1} + (service, j\mu).Queue_{j-1} \quad , \qquad 1 \leq j \leq n$$

$$Queue_j \stackrel{\text{def}}{=} (arrival, \top).Queue_{j+1} + (service, n\mu).Queue_{j-1} \quad , \qquad n \leq j \leq N-1$$

$$Queue_N \stackrel{\text{def}}{=} (service, n\mu).Queue_{N-1}$$

$$S \stackrel{\text{def}}{=} (arrival, \lambda).S$$

$$Queue_0 \underset{(arrival)}{\bowtie} S$$

Figure 3: A PEPA model of an $M/M/n$ queue

In order to maintain the desired effect of no synchronised actions being active in the queue it was necessary to make the *service* action entirely part of the queue process. If this model were to be extended by the servers being considered subject to failure, then by far the simplest model would involve active rates of service in both the queue and the scheduler. Such a situation is well catered for in PEPA with the resultant synchronised rate being the minimum of the two given rates. Therefore, if the scheduler gave the service rate available from the number of working servers and the queue gave the desired service required from the number of jobs required, the minimum of these would be the actual amount of service given.

An alternative way of specifying this model is to consider each server separately, rather than collectively as in the model in Figure 3. This is perhaps most easily achieved by considering the system as a queue synchronised with a linear combination of servers, i.e.

$$Queue_0 \underset{L}{\bowtie} (Server_1 \| Server_2 \| \cdots \| Server_n)$$

Such an approach would generally increase the size of the specification. However it will provide greater flexibility in extending the model, particularly when considering servers with different characteristics. If servers were subject to failure then a scheduler might be employed to ensure that the fastest possible service is being delivered to jobs in the queue.

This is an important issue when the number of jobs in the queue is less than the number of available severs.

In a multi-queue environment there are many possible scenarios leading to customers voluntarily vacating a queue. In a single queue situation the main reason for reneging is due to time-outs. Modelling time-outs from the perspective of the queue, rather than the job, is difficult as every job in the queue will have its own time out process. It is not possible to model a deterministic time-out in PEPA, instead it is necessary to assume that a job will time-out after a negative exponentially distributed period with a given mean. Such a model is illustrated in Figure 4.

$$Queue_0 \stackrel{\text{def}}{=} (arrival, \top).Queue_1$$

$$Queue_j \stackrel{\text{def}}{=} (arrival, \top).Queue_{j+1} + (service, \top).Queue_{j-1} + (timeout, j\alpha).Queue_{j-1}$$

$$, \quad 1 \le j \le N-1$$

$$Queue_N \stackrel{\text{def}}{=} (service, \top).Queue_{N-1} + (timeout, N\alpha).Queue_{N-1}$$

$$S \stackrel{\text{def}}{=} (arrival, \lambda).S + (service, \mu).S$$

$$Queue_0 \underset{(arrival,service)}{\bowtie} S$$

Figure 4: A PEPA model of an $M/M/1$ queue where customers may time-out before service

As in the multi-server model in the previous figure, an action has been defined wholly within the queue process. In this case it is somewhat easier to justify as the rate at which jobs leave the queue is solely dependent on the number of jobs in the queue. An alternative way of viewing this model is that as well as the single server giving a conventional service, there are also an infinite number of servers generating 'time-out' services, hence the similarity to the previous case.

Negative customers are another useful way of modelling the departure of customers from the queue before service, i.e. reneging. In a negative customer model a second arrival process takes place that causes 'normal' customers to be lost. Such an arrival process can be interpreted as a cancelling message, indicating that a response is either no longer needed or possible. Clearly many such models are possible, even with an $M/M/1$ queue. In the simplest case the arrival of a negative customer causes the immediate departure of a single job from the queue, if a job is present. In such a scenario the arrival process of negative customers may be viewed merely as a secondary service process, albeit where job loss becomes an important performance measure. It is a small variation on this case that is given in Figure 5. Here a negative customer arrival causes the departure of two jobs from the queue, but the job in service is assumed exempt. If only two jobs are in the queue (including the job in service) then only the one not in service is lost.

$$Queue_0 \stackrel{\text{def}}{=} (arrival, \top).Queue_1$$

$$Queue_1 \stackrel{\text{def}}{=} (arrival, \top).Queue_2 + (service, \top).Queue_0$$

$$Queue_2 \stackrel{\text{def}}{=} (arrival, \top).Queue_3 + (service, \top).Queue_1 + (negative, \top).Queue_1$$

$$\begin{aligned} Queue_j \stackrel{\text{def}}{=}\ & (arrival, \top).Queue_{j+1} + (service, \top).Queue_{j-1} \\ & +(negative, \top).Queue_{j-2} \quad , \quad 3 \leq j \leq N-1 \end{aligned}$$

$$Queue_N \stackrel{\text{def}}{=} (service, \top).Queue_{N-1} + (negative, \top).Queue_{N-2}$$

$$S \stackrel{\text{def}}{=} (arrival, \lambda).S + (service, \mu).S + (negative, \alpha).S$$

$$Queue_0 \underset{(arrival, service, negative)}{\bowtie} S$$

Figure 5: A PEPA model of an $M/M/1$ queue with negative customers

This model might conceivably be thought of as providing an additional bulk service. As well as a 'normal' single job service there is a bulk service given by the negative arrival to serve 2 jobs simultaneously. Clearly many further variations on the negative customer model are possible, such as the entire queue being lost. Such models are often used as an alternative means of modelling failures where the repair time is either negligible or not of interest.

A further important extension is where negative customers are queued in the event that no positive customers are present to be removed. This can be modelled either as a separate queue, or perhaps a single queue for both types of customer with negative, as well as, positive states. In this model the arrival of negative customers can remove jobs from the queue (including the job in service) and, in the event of the queue being empty (non-positive) can be queued to instantly remove future arrivals. Such a model is specified in Figure 6.

The queueing of negative customers might be interpreted as a server suffering compound failures. Models such as these have been used to study maintenance strategies where

$$Queue_j \stackrel{\text{def}}{=} (arrival, \top).Queue_{j+1} + (service, \top).Queue_{j-1}$$

$$+(negative, \top).Queue_{j-2} \quad , \quad 1 \leq j \leq N-1$$

$$Queue_j \stackrel{\text{def}}{=} (arrival, \top).Queue_{j+1} + (negative, \top).Queue_{j-2} \quad , \quad 2-N \leq j \leq 0$$

$$Queue_N \stackrel{\text{def}}{=} (service, \top).Queue_{N-1} + (negative, \top).Queue_{N-2}$$

$$Queue_{-N} \stackrel{\text{def}}{=} (arrival, \top).Queue_{1-N}$$

$$Queue_{1-N} \stackrel{\text{def}}{=} (arrival, \top).Queue_{2-N} + (negative, \top).Queue_{-N}$$

$$S \stackrel{\text{def}}{=} (arrival, \lambda).S + (service, \mu).S + (negative, \alpha).S$$

$$Queue_0 \underset{(arrival, service, negative)}{\bowtie} S$$

Figure 6: A PEPA model of an $M/M/1$ queue with queued negative customers

repair can be both responsive (to failures) and preventative. The main interest in negative customers has been in networks of queues where a job completing service at one node may enter another queue as either a positive or negative customer. Many interesting results have been derived for such cases.

Another consideration for a multi-server single queue system might be where the service of jobs is replicated to ensure a correct result. One such common system is *tri-modular redundancy*, or TMR. There are 3 servers in a TMR system and each performs the same service for each job. On completion of service the results are compared and a majority verdict delivered. It is possible that a decision could be made if the first two results computed are identical. It is assumed that a correctly working server will always give a correct result and a faulty server an incorrect one. Clearly such models are only of interest when the servers are subject to failure. It is highly unlikely that more than one server would be faulty at any given time — such an event would be regarded as 'absolute failure' and would be avoided at all costs. This model is clearly somewhat different to that in Figure 3 as there is at most 1 job being served, albeit at 3 servers simultaneously. The most suitable form for such a model is probably of the kind presented in Figure 1. In this case the queue would remain essentially passive, although the scheduler would be greatly expanded to account for the amount of successful service given and potential service available.

There are a great many possible models involving replication, far too many to cover here. However one further point is worth making. In all the failure models considered so far the occurrence of a failure has become evident immediately. The comparison of results in this model opens up a further possibility, namely that a processor is only discovered to be faulty when it delivers a false result. From the point of view of the performance model the only effect this would have is to delay the onset of the repair process until after the completion of the service immediately following the breakdown. In addition the result obtained by the faulty processor will have to discounted and provision will need to be made to catch the possibility of two or more processors failing before service completion. The same effect might also be considered for the failure models considered above, the assumption being that a faulty processor will give a noticeably erroneous result.

4.2 Models of Queues in Parallel

The simple breakdown model, illustrated in Figure 1, can be thought of as a special case of a more general model involving two queues and a single server. In this model there are arrivals of two different types of job, which are queued separately. One type of job carries a higher priority and will always receive service in preference to the other type of job. Thus jobs of the lower priority type will only receive service when the higher priority queue is empty. Furthermore, the service of a lower priority job will be interrupted if a higher priority request is received. The simple breakdown model is a special case of this model with the maximum length of higher priority queue set at 1 job (breakdown). Hence a model involving two priority queues can be interpreted as a compound breakdown model. More conventionally priority queueing models are used to study cases where one or more arrival sources are considered more urgent than others. One such scenario might be where an organisation offers open access to a system, e.g. a database or WWW server, but wishes to retain priority access for its own members.

To specify this model in PEPA it is necessary only to know whether the higher priority queue is empty or not, i.e. the scheduler will need only two states. This can be achieved by the use of a trigger action for the service from the higher priority queue when it contains 1 job only. If service is being performed on a job from the lower priority queue then it will be interrupted immediately the moment a higher priority job arrives. This model is shown in Figure 7, with the second queue having the higher priority.

This model may be extended in several ways, for instance by the addition of more job types. In a variation of this model a lower priority job may be allowed to complete its service even if a higher priority job arrives. Such an extension would merely require the addition of an extra scheduler state to which control would pass when a higher priority job arrives, but which would still serve the head lower priority job, thus:

$$S_1 \stackrel{\text{def}}{=} (service1, \mu_1).S_1 + (arrival2, \lambda_2).S_3$$

$$S_3 \stackrel{\text{def}}{=} (service1, \mu_1).S_2 + (arrival2, \lambda_2).S_3$$

A further extension along the same lines might be to consider the two job types to be of equal priority. This can be achieved by defining the state $Queue1_1$ in the same way as

$$Queue1_0 \stackrel{\text{def}}{=} (arrival1, \lambda_1).Queue1_1$$

$$Queue1_j \stackrel{\text{def}}{=} (arrival1, \lambda_1).Queue1_{j+1} + (service1, \top).Queue1_{j-1} \ , \ \ 1 \leq j \leq N-1$$

$$Queue1_N \stackrel{\text{def}}{=} (service1, \top).Queue1_{N-1}$$

$$Queue2_0 \stackrel{\text{def}}{=} (arrival2, \top).Queue2_1$$

$$Queue2_1 \stackrel{\text{def}}{=} (arrival2, \top).Queue2_2 + (last, \top).Queue2_0$$

$$Queue2_j \stackrel{\text{def}}{=} (arrival2, \top).Queue2_{j+1} + (service2, \top).Queue2_{j-1} \ , \ \ 2 \leq j \leq M-1$$

$$Queue2_M \stackrel{\text{def}}{=} (service2, \top).Queue2_{M-1}$$

$$S_1 \stackrel{\text{def}}{=} (service1, \mu_1).S_1 + (arrival2, \lambda_2).S_2$$

$$S_2 \stackrel{\text{def}}{=} (service2, \mu_2).S_2 + (arrival2, \lambda_2).S_2 + (last, \mu_2).S_1$$

$$(Queue1_0 \parallel Queue2_0) \underset{(arrival2, service1, service2, last)}{\bowtie} S_1$$

Figure 7: A PEPA model of two prioritised queues with a single server

$Queue2_1$ in Figure 7, in order to provide a service trigger, $last1$ say, to signal that queue 1 is empty. Arrivals of the higher priority type can be moved from the scheduler to $Queue2_j$ (and therefore not synchronised) and S_1 will have the additional term $(last1, \mu_1).S_2$. In this case the server would serve jobs from a queue until it became empty, at which point it would exhaustively serve the next queue and so on. Related to this is the idea of the polling system, where a job is served from each queue in turn. Polling systems have already been modelled with PEPA in some detail by Hillston [7].

There are many possibilities for interaction when queues are arranged in parallel offering equivalent services. The most commonly studied form of interaction in this case is state-dependent routing. The state in question in this case being either the operational state (when servers are subject to failure for example), the number of jobs in the various queues, or a combination of the two. The first of these cases is much the simpler to model as there are generally fewer changes in routing and these changes are triggered from outside the queue (see for example Thomas and Mitrani [13, 14] or Mitrani and Wright [10]). Such models fit easily into the queues / scheduler framework described in Section 3. A model of this type is illustrated in Figure 8.

$$Queue1_0 \stackrel{\text{def}}{=} (arr1, \top).Queue1_1$$

$$Queue1_j \stackrel{\text{def}}{=} (arr1, \top).Queue1_{j+1} + (serv1, \top).Queue1_{j-1} \quad , \quad 1 \leq j \leq N-1$$

$$Queue1_N \stackrel{\text{def}}{=} (serv1, \top).Queue1_{N-1}$$

$$Queue2_0 \stackrel{\text{def}}{=} (arr2, \top).Queue2_1$$

$$Queue2_j \stackrel{\text{def}}{=} (arr2, \top).Queue2_{j+1} + (serv2, \top).Queue2_{j-1} \quad , \quad 1 \leq j \leq M-1$$

$$Queue2_M \stackrel{\text{def}}{=} (serv2, \top).Queue2_{M-1}$$

$$S_0 \stackrel{\text{def}}{=} (arr1, \lambda_1).S_0 + (arr2, \lambda_2).S_0 + (repair1, \eta_1).S_1 + (repair2, \eta_2).S_2$$

$$S_1 \stackrel{\text{def}}{=} (serv1, \mu_1).S_1 + (arr1, \lambda_1 + \lambda_2).S_1 + (fail1, \xi_1).S_0 + (repair2, \eta_2).S_3$$

$$S_2 \stackrel{\text{def}}{=} (serv2, \mu_2).S_2 + (arr2, \lambda_1 + \lambda_2).S_2 + (fail2, \xi_2).S_0 + (repair1, \eta_1).S_3$$

$$S_3 \stackrel{\text{def}}{=} (arr1, \lambda_1).S_3 + (arr2, \lambda_2).S_3 + (serv1, \mu_1).S_3 + (serv2, \mu_2).S_3$$
$$\quad + (fail1, \xi_1).S_2 + (fail2, \xi_2).S_1$$

$$(Queue1_0 \parallel Queue2_0) \underset{(arr1, arr2, serv1, serv2)}{\bowtie} S_3$$

Figure 8: A PEPA model of two $M/M/1$ queues in parallel with state dependent routing

In this model (taken from Thomas and Mitrani [14]) jobs are shared between two queues, each of which has a single server attached. When both servers are operational, state S_3, jobs are directed to either queue. However if one or other fails, states S_1 and S_2, then all jobs are directed towards the operational server. If both servers are broken, state S_0, then jobs are once more shared as in the fully operational state. The failure of a server suspends the service of jobs from that queue, but no jobs are lost from the queue. Clearly this model is easy to extend, either by adding more parallel queues (and scheduler states) or changing the effect of failures (as in Section 4.1).

The scheduler states are a representation of the binary system operational state, where broken is 0 and not broken is 1, e.g. if only server i is working the operational (scheduler) state is 2^i and so on. As such the scheduler states can also be generated recursively. An alternative approach to modelling such a model might be to consider each of the servers as separate processes, rather than collectively in the scheduler. Such an approach would conceivably give rise to fewer states in the specification when the number of servers is large. However, it would clearly be difficult to construct a mechanism to reroute jobs without full knowledge of the operational state of the system.

The question of routing dependent on the number of jobs in the parallel queues is clearly somewhat more difficult to model. It would appear that in order to direct jobs towards, or away from, a queue because of the number of jobs it contains a scheduler would need to 'know' most of, if not all, the system state. Clearly it seems nonsensical to implicitly specify every state in the system, this would be the equivalent of writing out the balance equations and destroy any advantage of compositionality. In most cases however it is necessary for the scheduler to know only that a queue is in one of a very small number of states (e.g. full or empty, as in the priority queues model above) or to know the relative number of jobs in two or more queues.

In the case of breakdowns the change of state of the scheduler was triggered by an action that was specific to the operational state, i.e. a failure or repair. In queue-size dependent routing the triggers for scheduler state change are arrivals into and departures from the queue, but not every arrival or departure will cause the scheduler state to change. Such a mechanism was used in the priority queues model in Figure 7 to indicate to the scheduler when the higher priority queue was empty. The first such case we will consider is where jobs are directed away from a full queue in order that job-loss is minimised. Clearly all a scheduler will need to know is whether each queue is full or not; if a queue is full then jobs will be directed elsewhere. If all the queues are full then all arriving jobs will be lost until a service occurs. In order to change the scheduler state a different arrival action is used to fill the final place in the queue, but with the same rate of course. This model is given in Figure 9.

With this model it is possible to envisage a number of possible scenarios, for instance a primary server with a backup at times of high load. It is also possible to envisage many extensions to this model. One possible extension might be to route all jobs to an empty queue, if one exists. This could be done in the same way as in the model in Figure 7, with triggers used to signal the completion of the last job in each queue and the addition of 3 scheduler states, $BothEmpty$, $Q1Empty$ and $Q2Empty$.

Another interesting problem arising when routing is dependent on queue size is that of queue balancing. In this model jobs are shared between those queues which contain the least number of jobs. In the two queue case this means that jobs are shared only when the queues are of equal size, otherwise all jobs are sent to the smaller queue. The state space for the scheduler is obviously going to be large for this model. The scheduler state is in fact dependent on the relative sizes of the two queues and therefore has $2N+1$ states, where N is the maximum number of jobs in either queue. This model is illustrated in Figure 10.

Clearly this specification gives rise to a large number of states, $N+1$ in each queue

$$Queue1_0 \stackrel{\text{def}}{=} (arr1, \top).Queue1_1$$

$$Queue1_j \stackrel{\text{def}}{=} (arr1, \top).Queue1_{j+1} + (serv1, \top).Queue1_{j-1} \quad , \quad 1 \leq j \leq N-2$$

$$Queue1_{N-1} \stackrel{\text{def}}{=} (lastArr1, \top).Queue1_N + (serv1, \top).Queue1_{N-2}$$

$$Queue1_N \stackrel{\text{def}}{=} (serv1, \top).Queue1_{N-1}$$

$$Queue2_0 \stackrel{\text{def}}{=} (arr2, \top).Queue2_1$$

$$Queue2_j \stackrel{\text{def}}{=} (arr2, \top).Queue2_{j+1} + (serv2, \top).Queue2_{j-1} \quad , \quad 1 \leq j \leq M-2$$

$$Queue2_{M-1} \stackrel{\text{def}}{=} (lastArr2, \top).Queue2_M + (serv2, \top).Queue2_{M-2}$$

$$Queue2_M \stackrel{\text{def}}{=} (serv2, \top).Queue2_{M-1}$$

$$NotFull \stackrel{\text{def}}{=} (arr1, \lambda_1).NotFull + (arr2, \lambda_2).NotFull + (serv1, \mu_1).NotFull$$
$$+(serv2, \mu_2).NotFull + (lastArr1, \lambda_1).Q1Full + (lastArr2, \lambda_2).Q2Full$$

$$Q1Full \stackrel{\text{def}}{=} (arr2, \lambda_1 + \lambda_2).Q1Full + (lastArr2, \lambda_1 + \lambda_2).BothFull$$
$$+(serv1, \mu_1).NotFull + (serv2, \mu_2).Q1Full$$

$$Q2Full \stackrel{\text{def}}{=} (arr1, \lambda_1 + \lambda_2).Q2Full + (lastArr1, \lambda_1 + \lambda_2).BothFull$$
$$+(serv2, \mu_2).NotFull + (serv1, \mu_1).Q2Full$$

$$BothFull \stackrel{\text{def}}{=} (serv1, \mu_1).Q2Full + (serv2, \mu_2).Q1Full$$

$$(Queue1_0 \parallel Queue2_0) \underset{(arr1, arr2, serv1, serv2, lastArr1, lastArr2)}{\bowtie} NotFull$$

Figure 9: A PEPA model of two $M/M/1$ queues in parallel with routing away from a full queue

$$Queue1_0 \stackrel{\text{def}}{=} (arr1, \top).Queue1_1$$

$$Queue1_j \stackrel{\text{def}}{=} (arr1, \top).Queue1_{j+1} + (serv1, \top).Queue1_{j-1} \quad , \quad 1 \leq j \leq N-1$$

$$Queue1_N \stackrel{\text{def}}{=} (serv1, \top).Queue1_{N-1}$$

$$Queue2_0 \stackrel{\text{def}}{=} (arr2, \top).Queue2_1$$

$$Queue2_j \stackrel{\text{def}}{=} (arr2, \top).Queue2_{j+1} + (serv2, \top).Queue2_{j-1} \quad , \quad 1 \leq j \leq N-1$$

$$Queue2_N \stackrel{\text{def}}{=} (serv2, \top).Queue2_{N-1}$$

$$S_0 = (arr1, \lambda_1).S_1 + (arr2, \lambda_2).S_{-1} + (serv1, \mu_1).S_{-1} + (serv2, \mu_2).S_1$$

$$S_j = (arr2, \lambda_1 + \lambda_2).S_{j-1} + (serv1, \mu_1).S_{j-1} + (serv2, \mu_2).S_{j+1} \ , \ 1 \leq j \leq N$$

$$S_j = (arr1, \lambda_1 + \lambda_2).S_{j+1} + (serv1, \mu_1).S_{j-1} + (serv2, \mu_2).S_{j+1}$$

$$, \quad -N \leq j \leq -1$$

$$(Queue1_0 \parallel Queue2_0) \underset{(arr1, arr2, serv1, serv2)}{\bowtie} S_0$$

Figure 10: A PEPA model of two $M/M/1$ balanced queues in parallel

plus $2N+1$ in the scheduler. However, this is a significant saving over listing every state in the underlying Markov chain when N is relatively large. There are $(N+1)^2$ states in the underlying Markov chain and $4N+3$ states in the model specification. Therefore this specification has fewer states than the underlying Markov chain if $N \geq 3$. Furthermore, whilst the underlying Markov chain suffers from exponential growth, the specification is linear.

It is interesting to note that the model of the balanced queue described above is only the optimal routing strategy if the two servers are identical, i.e. $\mu_1 = \mu_2$. If one server is faster than the other then the shortest average queue size average (hence minimum average response time) will be achieved by sending more jobs to the faster server, even when its queue is considerably the larger of the two. There may, however, be an advantage in sending jobs to the slower server if its queue is empty (or at least nearly so). Clearly the optimal routing strategy in this case will form a model dependent on the actual size of both queues and as such would be extremely complicated to specify in PEPA, or indeed any other way. Another interesting extension to the queue balancing model is to incorporate simple breakdowns that suspend service but do not affect jobs in the queue.

The final part of this section is devoted to systems where jobs transfer from one queue to another. There are many possible scenarios leading to a job leaving one queue and joining another, but in most cases there will be some perceived advantage to be gained in doing so. One exception to this is where a job may receive multiple services at the same stage in a network. In this case a job may complete service only to be fed back, possibly with a different priority. The question of feedbacks and multiple services is dealt with in the next section.

There may be many possible reasons that make it advantageous for a job to move queues, for instance a server may breakdown or an alternative queue may have fewer jobs in it. In every case the advantage is that it is probable the job in question will complete its service sooner if it moves to another queue. In general there are two forms of movement between queues (other than the feedback case already mentioned), these are instantaneous response to a change in system state, e.g. a breakdown or idle server, and a more gradual (timed) migration.

The migration of jobs from a longer queue to a shorter one might be modelled as an extension of the model in Figure 10. In this case activities, $one2two$ and $two2one$, might be added to the queues such that $one2two$ removes a job from queue 1 and adds one to queue 2 and $two2one$ performs the reverse. The active rates for these activities are then given in the scheduler such that $one2two$ is carried out when $1 \leq j \leq N$ and $two2one$ is carried out when $-N \leq j \leq -1$. The rates of these activities might be proportional to the difference in queue size, j. A further extension of this model might be to add states to the scheduler to show when one or other of the queues is empty and adjust the migration rates accordingly.

The situation where jobs instantaneously move from one queue is somewhat more difficult to model directly. It is also a case where some consideration of how this might be achieved in practice is useful, for instance such a system might be better modelled as an $M/M/n$ queue. Alternatively it might be useful to consider the case where servers may interchange their queues, such that an idle server might swap its empty queue for the non-empty queue of a broken partner. Such a situation is easily handled by the scheduler simply by swapping service activities.

4.3 Models involving Many Services

It is in models where jobs receive more than one service that the compositional nature of PEPA gives a great advantage in the specification of queueing models. In general there are two mechanisms whereby this may occur: firstly a job may complete service at one server and then progress to be queued at another; alternatively on completion a job may be fed back to rejoin the same queue (or set of queues that comprise a node). In Figure 11 a model of two queues in tandem is given where some jobs feedback.

$$
\begin{aligned}
Queue1_0 &\stackrel{\text{def}}{=} (arrival, \lambda).Queue1_1 + (feedback, \top).Queue1_1 \\
Queue1_j &\stackrel{\text{def}}{=} (arrival, \lambda).Queue1_{j+1} + (service1, \mu_1).Queue1_{j-1} \\
&\qquad\qquad + (feedback, \top).Queue1_1 \quad , \quad 1 \leq j \leq N-1 \\
Queue1_N &\stackrel{\text{def}}{=} (service1, \mu_1).Queue1_{N-1} + (feedback, \top).Queue1_N
\end{aligned}
$$

$$
\begin{aligned}
Queue2_0 &\stackrel{\text{def}}{=} (service1, \top).Queue2_1 \\
Queue2_j &\stackrel{\text{def}}{=} (service1, \top).Queue2_{j+1} + (feedback, q\mu_1).Queue2_{j-1} \\
&\qquad\qquad + (leave, (1-q)\mu_1).Queue2_{j-1} \quad , \quad 1 \leq j \leq N-1 \\
Queue2_N &\stackrel{\text{def}}{=} (service1, \top).Queue2_N + (feedback, q\mu_1).Queue2_{N-1} \\
&\qquad\qquad + (leave, (1-q)\mu_1).Queue2_{N-1}
\end{aligned}
$$

$$
Queue1_0 \underset{(arrival, service1, feedback)}{\bowtie} Queue2_0
$$

Figure 11: A PEPA model of two queues in tandem with feedback

The first thing to note about the model in Figure 11 is that there is no need to use a scheduler in order to specify this model. Given a more complicated model it would be desirable to include a scheduler in order to improve the clarity and ease of solution. However, this model is sufficiently simple for this not to be necessary, indeed to do so would unnecessarily complicate the model specification. In general it is desirable to include a scheduler whenever there are processes other than service and arrivals. In this case the $feedback$ process is clearly a service at one node and an arrival at another.

A further point of interest about this model is the presence in the Nth terms of the queues of passive instances of the $feedback$ and $service1$ processes. Such terms might not be expected since the queues are full in these instances. However, they are necessary in order to allow the activity in the other queue to proceed. Such a definition is equivalent to saying that a job leaving queue 1 and finding queue 2 full will leave the system (and vice versa). Alternatively the model could be defined such that a service cannot complete until there is space in the succeeding queue.

There are several alternative ways of handling jobs that are being sent to a full queue. For instance, service at a given server may not be initiated unless its succeeding queue is not full. Alternatively a server may suspend service of any subsequent jobs until the completed job can be accepted at the succeeding queue. The first of these cases is the easiest to model. Where jobs may only proceed from one queue to another it is sufficient only that the acceptance of jobs is suppressed when the succeeding queue is full, i.e. in the model in Figure 11 the $service1$ term is omitted from the expression for $Queue2_N$. In this situation a job arriving at a full queue is lost from the system. In the case where there is a choice of destinations there are more options. It might be that all service is suspended until all the specified destinations are available. This is not a very efficient solution, more practically it might be decided to suspend service at a given server only when service is requested by a job that is intended for a full queue. Alternatively all jobs might be directed away from full queues (as in the model in Figure 9), perhaps with modified service rate. Possibly jobs intended for full queues are not served until those queues are available, but other jobs might be served, perhaps by using different job types as in the model in Figure 7. It should be noted that if service is always suspended then there is a potential for deadlock when both queues are full.

Many of the above comments also apply to multi-stage services where nodes suffer failure. In the breakdown model considered in Figure 1 jobs could still be queued, giving a possible alternative to rerouting. However in the model in Figure 2 the queue also suffered complete failure and so jobs would either need to be rerouted or wait for repair. Clearly there are a great many possible scenarios governing the routing of jobs between stages, but potential means of specifying them should be clear from the models presented earlier. The compositional nature of PEPA models in general, and the structures we have used in earlier models in particular, makes specifying staged service models relatively straightforward.

5 Numerical Solution

The first observation that should be made here is that all the specifications in the preceding sections are given for queues with a finite maximum length. This gives rise to finite state irreducible Markov chains, which are always theoretically possible to solve.

Using the PEPA Workbench [3] gives the underlying Markov chain expressed in sparse block tri-diagonal matrix form. This matrix can then be evaluated by conventional Gaussian methods, on the condition that the sum of all probabilities is one. However, in practice the underlying Markov chain can become extremely large, thus it becomes difficult to handle the size of the matrices. As matrix size increases the time taken to derive a solution rapidly becomes excessive. More significantly, very large amounts of memory are required to perform large matrix calculations. Often the limiting factor on model size therefore is the amount of memory available. Clearly another method is needed and this has long been a major focus of research into Markov chains.

One of the attractions of studying queueing models is that they invariably give rise to recurrent structures that facilitate a product form solution. In [5] Harrison and Hillston have generalised such results from queueing models to propose a method for the solution of a corresponding class of models expressed in PEPA. This class of models is defined by the state of the system being governed by input and output processes. In queueing models these processes are the arrivals and departures of jobs. A similar technique has also been employed by Bhabuta et al [1]. In this method structures are identified that are *quasi-reversible* (or *reversible* in [1]).

A product form solution means that the probability of a state of a system can be expressed as the product of the probabilities of the corresponding sub-states. In a queueing network this is normally interpreted as the joint probability of the state of the network can be expressed in terms of the marginal probability of the state of individual queues. In general interactions of the kind described in this paper relate the behaviour of queues to each other and to other processes in the system so that a product form solution does not, in general, exist. Alternative notions of product form which relate to Petri net models have been applied to PEPA models by Sereno [11].

For an example of product form, consider the model in Figure 11 when the queue is unbounded. This model is an example of a Jackson network and has product form. Thus each queue can be considered as an independent $M/M/1$ queue with arrival rate $\lambda/(1-q)$ and service rate μ_i. The product of the marginal queue size probabilities gives the joint queue size probability. If either of the queues is bounded then the model does not have product form. If the feedback loop is removed ($q = 0$) then the model has product form if and only if the first queue is unbounded.

A further example of product form is given by a variation of the model in Figure 1. If arrivals are not permitted during repair (scheduler state S_0) then this model has product form such that the queue size probabilities and the state of the server (scheduler) are independent. In this case the marginal queue size probabilities are given by an $M/M/1$ queue (without breakdowns) with arrival rate λ and service rate μ. The probabilities of the operational state of the server are given by $\alpha \times prob[S_1] = \beta \times prob[S_0]$. This result holds regardless of whether the queue is bounded or not. If actions are permitted in S_0 that change the number of jobs in the queue then the resulting model will not have product form. This model is reversible, whereas the model in Figure 11 considered above is quasi-reversible.

The existence of multiple boundary conditions often makes it extremely difficult to obtain analytical results for finite length queues. Conversely it is much easier in general to obtain analytical results for single infinite queues. For example, consider the model in Figure 1. The first observation that might be made is that changes in scheduler state are independent of the number of jobs in the queue, but the converse is not true. Thus,

$$\beta \times prob[S_0] = \alpha \times prob[S_1]$$

gives the marginal probability of the state of the scheduler. This result holds regardless of the length of the queue. Clearly the majority of the states in the specification are defined recursively, with the exceptions being the boundary conditions of the queue, $Queue_0$ and $Queue_N$. Furthermore, it is clear that,

$$\lambda \times prob[Queue_i] = \mu \times prob[S_1 \,\&\, Queue_{i+1}] \qquad 0 \leq i \leq N-1$$

also,

$$(\beta + \lambda) \times prob[S_0 \,\&\, Queue_{i+1}] = \alpha \times prob[S_1 \,\&\, Queue_{i+1}] + \lambda \times prob[S_0 \,\&\, Queue_i]$$

where, $0 \leq i \leq N-1$, and,

$$(\beta + \lambda) \times prob[S_0 \,\&\, Queue_0] = \alpha \times prob[S_1 \,\&\, Queue_0]$$

These balance equations easily give rise to an expression for the probability generating function when the queue in unbounded, i.e. $N \to \infty$. It is also possible to find expressions for the queue size probabilities when the queue is finite, although evaluating the boundary conditions becomes somewhat more problematical.

It should be noted that the above analysis relies on finding expressions for the balance equations of the underlying Markov chain. Such a method is not strictly in the traditions of the use of process algebra. Also this method would be extremely difficult to automate, even given a consistent model structure such as that used here. A much better method for infinite length queues therefore is the one adopted by Mitrani et al in [9] to link a Markovian process algebra (TIPP) to a matrix geometric solution method (spectral expansion).

For large finite queues there are improvements that can be made to the current matrix method used by the PEPA workbench. For instance, the balance equations of the form above can be generated in a sequential manner. In this way the steady state probability of every state of the chain is defined in terms of two preceding states. Given an initial condition relating only two states, all the states can be expressed in terms of just one of these initial states. This initial state can then be evaluated using the normalising equation, $\sum_{\forall i} p_i = 1$. Hence all the steady state probabilities can be found. This method, whilst more difficult to implement than standard matrix Gaussian methods, has the advantage that it forms only a vector of size $K-1$ rather than a matrix of size K^2 (where K is the number of states in the chain).

Most of the models presented here are multi-dimensional, their birth-death processes operate on more than one numerical line. In all cases the state of the queues is dependent on the actions of the scheduler, they are not separable. However, the queues themselves can often be considered independent of each other and still maintain the same overall behaviour. By considering the queues this way it is possible to find the marginal queue size probabilities, which can then be used to find most performance measures of interest. Such a method is used in [10] and [14] and is referred to as *quasi-separability*.

Such a simplification is easily expressed in PEPA. For instance, in the two queue parallel system given in Figure 8 the actions $arr1$ and $serv1$ apply only to queue 1 and the actions $arr2$ and $serv2$ apply only to queue 2. Hence the expression,

$$(Queue1_0 \,\|\, Queue2_0) \underset{(arr1, arr2, serv1, serv2)}{\bowtie} S_3$$

can be rewritten as,

$$(Queue1_0 \underset{(arr1, serv1)}{\bowtie} S_3) \underset{(arr2, serv2)}{\bowtie} Queue2_0$$

or,

$$(Queue2_0 \underset{(arr2,serv2)}{\bowtie} S_3) \underset{(arr1,serv1)}{\bowtie} Queue1_0$$

Thus, evaluating the first part of each of these expressions,

$$Queue(i)_0 \underset{(arr(i),serv(i))}{\bowtie} S_3$$

will give an exact result for the marginal queue size probabilities for queue i. Clearly if models have been specified with passive actions in the scheduler then the scheduler expressions will need to be rewritten before evaluating the synchronised process. The condition by which this method may be applied is that the state of the scheduler cannot be determined by the state of the queues.

It is important to note that this approach does not give rise to expressions for the joint queue size probabilities since, as stated, this system is not separable. However the average number of jobs in the system is given by the product of the average number of jobs in each queue which can be found from the marginal queue size probabilities. Other performance measures of interest can also be derived from these marginal probabilities. It is clear therefore that this approach does not give a product form solution, but is very useful nevertheless.

A further extension of this method is to consider queues as behaving partially independently even when they clearly are not. Such a method was used in [12] to study a pipeline of queues suffering breakdowns. Here the number of jobs in a queue is dependent on the state of every preceding queue and server, but a good approximation can be made by considering only the effect of the preceding stage of service. This method is an example of common approximation techniques of *lumping* states together and smoothing out the behaviour of the resulting super-state by assuming that it is Poisson.

This approach can be illustrated by considering the model of two prioritised queues given in Figure 7. In the first instance this system can be treated as quasi-separable, hence the expression,

$$(Queue1_0 \parallel Queue2_0) \underset{(arrival2,service1,service2,last)}{\bowtie} S_1$$

can be rewritten as,

$$(Queue2_0 \underset{(arrival2,service2,last)}{\bowtie} S_1) \underset{(service1)}{\bowtie} Queue1_0$$

The first part of this can be evaluated to give the marginal queue size probabilities for the higher priority queue. In fact, the higher priority queue can be regarded as an entirely independent $M/M/1$ queue. Unlike the example used above to illustrate quasi-separability, the same approach cannot be used to derive the marginal queue size probabilities for the lower priority queue. The service of the lower priority queue is dependent on the higher priority queue being empty. In the above expression this is represented by the synchronisation of the actions $arrival2$ and $last$. The action $arrival2$ occuring in scheduler state S_1 causes the scheduler to change state to state S_2. This transition is never blocked since in state S_1 queue 2 is always empty, hence it may be treated independently of the process $Queue2_i$. In scheduler state S_2 queue 2 is always non-empty, hence the action $service2$ will not be blocked. However, the action $arrival2$ will be blocked if queue 2 is full and the action $last$ will be blocked unless queue 2 contains exactly 1 job. In order to treat the queues as independent this synchronisation between the scheduler and queue 2 must be relaxed to remove blocking. This can be done by modifying the rates of the actions $arrival2$ and $last$ by multiplying them with the probabilities that queue 2 is not full or contains exactly 1 job respectively. These probabilities are calculated from the previous expression for the marginal queue size probabilities at queue 2. Thus,

$$S_2 \stackrel{\mathrm{def}}{=} (service2, \mu_2).S_2 + (arrival2, \lambda_2 \times \sum_{i=1}^{M-1} prob[Queue2_i \mid S_2]).S_2$$
$$+(last, \mu_2 \times prob[Queue2_1 \mid S_2]).S_1$$

where $prob[Queue2_i \mid S_2]$ is the probability that queue 2 contains exactly i jobs given that queue 2 is not empty (scheduler state S_2). Thus the expression,

$$Queue1_0 \underset{(service1)}{\bowtie} S_1$$

can be evaluated to give an approximation to the marginal queue size probabilities for queue 1. It is important to note that there is more than one way to solve this model of prioritised queues exactly and that this method is used here merely as an example.

The presence of feedbacks, such as in the model in Figure 11, requires a slightly more involved solution method, except when a product form exists. Such a method might involve making an initial estimate of the feedback rate and repeatedly obtaining solutions to update this estimate, until a steady state is reached. Not all such models are easily solved and a poor initial estimate can lead to a diverging, rather than converging, result. In complex models a combination of the above methods will invariably be needed.

A large proportion of the literature on queueing systems is concerned with infinite length queues, that is, queues with no maximum length. Whilst such an unboundedness is clearly a physical impossibility it is a useful method for considering many scenarios. The current version of the PEPA workbench can only handle finite state Markov chains, however in [9] TIPP was linked to spectral expansion, a powerful matrix-geometric solution method for evaluating infinite state Markov chains. The results derived for TIPP can easily be applied to PEPA, hence all the relevant models specified above could easily be rewritten to account for infinite length queues. Such a specification would merely mean removing the upper bound of the queue and extending the index j to infinity.

In some situations models with infinite length queues are easier to solve than the same model with finite length queues. This arises from the added complexity of finite length queues caused by the queues becoming full. Many results exist that treat networks of infinite length queues as separable to obtain exact solutions. It should be noted that infinite state Markov chains can only be solved if they exhibit reversible or quasi-reversible behaviour. Also the spectral expansion method applies only to chains which are infinite in one dimension only. Very few results are available for systems that are infinite in more that one dimension except where those dimensions (queues) are essentially independent.

6 Conclusions

We have attempted to include here as many common types of queueing scenarios of a generic nature as possible. Many hundreds of useful models can be formed by considering variations on, and combinations of, these examples. Clearly there are many more possible behaviours than can ever be tackled in a survey paper such as this one. In addition we are seeking to extend the application of queueing model results further in PEPA. This paper is an early stage in that effort and therefore much remains to be done that will greatly extend the numerical solution of PEPA models.

The motivation for carrying out this work was to investigate the expressiveness of PEPA in an area that has well documented problems and solutions. We have successfully demonstrated that a wide range of interactions in queueing models can be expressed in PEPA. In particular we have specified priority systems despite the lack of explicit priorities in PEPA. Also we have shown how actions that appear to be immediate can be modelled in PEPA by the use of triggers. Our ultimate aim is to apply the methods outlined in the numerical solutions section to a much wider range of models than have previously been considered by working directly from specifications.

Finally, it is worth observing that the form used here to specify models is directed very much towards a particular goal of implementation. It is stated in many places in the text that there are numerous alternative ways to express these models. Similarly there are many alternative approaches to numerical solution. The particular form of models given here could be very different if the intended solution method or objective changed. For these reasons such models cannot be said to have canonical form. Rather there are many possible equivalent specifications that are equally valid given different solution objectives.

References

[1] M. Bhabuta, P. Harrison and K. Kakani, Detecting Reversibility Structures in Markovian Process Algebras, in: M. Merabti, M. Carew and F. Ball eds., *Performance Engineering of Computer and Telecommunication Systems*, Springer-Verlag, 1995.

[2] R.M. Davies, Discrete Simulation of Human Systems, in: *Proceedings of the 1990 UKSC Conference on Computer Simulation*, Brighton, UK, September 1990.

[3] S. Gilmore and J. Hillston, The PEPA Workbench: A Tool to Support a Process Algebra-based Approach to Performance Modelling, in: G. Haring and G. Kotsis eds., *Proceedings of the Seventh International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, Lecture Notes in Computer Science, vol. 794, pp. 353–368, Springer Verlag, 1994.

[4] N. Götz, U. Herzog and M. Rettelbach, TIPP - a language for timed processes and performance evaluation, in: J. Hillston and F. Moller eds., *Proceedings of the Workshop on Process Algebra and Performance Modelling*, University of Edinburgh, UK, May, 1993.

[5] P. Harrison and J. Hillston, Exploiting quasi-reversible structures to find product form solutions in Markovian process algebra models, *The Computer Journal* **38**(7), 1995.

[6] P. Harrison, N. Patel and E. Pitel, Negative Customers Model Queues with Breakdowns, in: M. Merabti, M. Carew and F. Ball eds., *Performance Engineering of Computer and Telecommunication Systems*, Springer-Verlag, 1995.

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

[8] J. Hillston, The Nature of Synchronisation, in: U. Herzog and M. Rettelbach eds., *Proceedings of the 2nd Workshop on Process Algebra and Performance Modelling*, Erlangen, Germany, 1994.

[9] I. Mitrani, A. Ost and M. Rettelbach, TIPP and the Spectral Expansion Method, in: F. Baccelli, A. Jean-Marie and I. Mitrani, eds., *Quantitative Methods in Parallel Systems*, Springer-Verlag, 1995.

[10] I. Mitrani and P.E. Wright, Routing in the Presence of Breakdowns, *Performance Evaluation* **20** pp.151–164, 1994.

[11] M. Sereno, Towards a product form solution for stochastic process algebras, *The Computer Journal* **38**(7), 1995.

[12] N. Thomas, Performance and Reliability in Distributed Systems, PhD Thesis, University of Newcastle-upon-Tyne, 1997.

[13] N. Thomas and I. Mitrani, Routing among different stages, in: F. Baccelli, A. Jean-Marie and I. Mitrani, eds., *Quantitative Methods in Parallel Systems*, Springer-Verlag, 1995.

[14] N. Thomas and I. Mitrani, Routing Among Different Nodes Where Servers Break Down Without Losing Jobs, *Proceedings of IEEE International Computer Performance and Dependability Symposium*, pp. 246–255, 1995.