Modelling and Simulation

Notes by Jane Hillston; course given by Kousha Etessami

Autumn Semester 2005

Contents

1 Modelling and Simulation

1.1 Introduction

This course teaches various aspects of computer-aided modelling with an emphasis on the performance evaluation of computer systems and communication networks. The performance of a system influences its design, procurement and subsequent use. Performance analysis may be carried out at any stage of that life-cycle. In general, the problem is to achieve the highest possible performance given the constraints on the system. Typical constraints might be the number of users, the bandwidth or the cost. Modelling and simulation are methods which are commonly used by performance analysts to represent constraints and optimise performance.

Many of the ideas of the course will be presented by examples; as well as computer systems and communication networks, other dynamic systems such as manufacturing and transport systems will sometimes be considered. With all the examples it will be as important to consider the underlying concepts as the details of the example itself.

1.2 Models as tools

The term *model* conjures up a variety of images. We will use it to mean something which represents a system in a form which allows us to make predictions about the behaviour of the system. How accurate these predictions are will depend on the detail we invest in the model. In some cases, when we only need an approximate or relative measure of the behaviour of the system, a very crude model will suffice. When the answers obtained from the model are crucial we will be prepared to expend more effort in developing our model and the representation of the system will be less abstract. During the course we will consider a range of modelling techniques from the very abstract, in which the behaviour of the system is captured by a single equation, the so-called *operational laws*, to the very detailed *simulation models* in which every aspect of the system's behaviour can be represented.

In all cases a model should be an abstraction of the system: an attempt to distill, from the mass of details that is the system itself, exactly those aspects that are essential to the system's behaviour. Once a model has been *defined* through this abstraction process, it can be *parameterised* to reflect any of the alternatives under study, and then *evaluated* to determine its behaviour under this alternative. Using a model to investigate system behaviour is less laborious and more flexible than direct experimentation, because the model is an abstraction that avoids unnecessary detail.

There are many reasons why investigations into the behaviour of a system cannot be carried out by direct experimentation on the system. At best experimentation will probably be disruptive, at worst, dangerous. For example, a system manager of a heavily loaded file server is unlikely to allow experimentation, or even simple monitoring, because the disruption to users will be too great. Systems such as nuclear reactor control systems or the London Underground control system are simply not safe to experiment with. However, timely response may be just as critical as correctness within such systems.

Another problem with direct experimentation may be that the internal behaviour which we wish to investigate may not be accessible in a working system. For example, in most operating systems it is difficult to obtain the exact timing of instruction level events. Experimentation, since it involves monitoring, is expensive (in terms of system resources) and time consuming. It is also difficult to use the results of direct experimentation to extrapolate to other scenarios of operation. For example, experimenting with one configuration of a communication network is unlikely to give us much insight into the expected performance if another configuration is used. This leads us to one of the most compelling reasons for using modelling instead of direct experimentation: it is often the case that the system which we wish to study does not yet exist.

The idea that a design should be subjected to some analysis to ensure that the resulting system will behave *correctly* is now widely accepted. Models are tools commonly used for this purpose: for example the process algebra CCS may be used to check that an implementation will satisfy conditions set out in a behavioural specification. Such models are sometimes called *qualitative models* because all definite values are abstracted away. In contrast, performance evaluation is concerned with ensuring that a system will satisfy constraints on timely behaviour and efficient resource utilisation. These constraints are sometimes expressed as *performance requirements* or a *performance specification*. These performance models are sometimes called *quantitative models*—if we wish to extract quantitative information (estimates of real values such as response time or throughput) then we must incorporate quantitative information in the model.

Throughout the course we will encounter a variety of modelling styles. The key notions of the course will recur whichever modelling style we are using. These notions are that

- a model can be constructed to represent some aspect of the dynamic behaviour of a system; and
- once constructed, such a model becomes a tool with which we can investigate the behaviour of the system.

1.3 Discrete event systems

A common feature of all the systems and models which we will consider is that they are *discrete event* systems. What this means is that our view of the system is characterised by variables that take distinct values and which change at distinct times or *events* in the system. For example, we might be interested in the number of nodes in a communication network which are currently waiting to send a message (N say). This number will change when one of two events occur: either a node, which was not previously waiting, generates a message and is now waiting to send ($N \to N + 1$), or a node, which was previously waiting, successfully transmits its message ($N \to N - 1$).

We will not consider *continuous* models. These are models of systems in which the variables characterising the system are constantly changing. An example of such a model would be the speed of a falling object or the drag experienced by a plane.

Of course, discrete systems and continuous systems are distinguished by the way in which we view them as much as by inherent properties of the situation they represent. In other words, for many systems the choice of whether to represent it as a discrete event or a continuous system will depend on several factors including the objectives of the study, the information available and the expertise of the modeller. However for some systems, such as computers and communication networks, the discrete event representation is generally the more natural one. One of our major tasks as modellers will be to choose the parameters which *characterise* the system under study.

Discrete time discrete event models only consider the system at *predetermined moments in time*, which are typically evenly spaced, eg. at each clock “tick”. State changes are observed only at these moments, which are not necessarily the moment at which the events themselves occur. Such models are often used to approximate systems with continuously changing states. *Continuous time* discrete event models, consider the system at the *time of each event*. Hence the time parameter in such models is conceptually continuous. At levels of abstraction above the hardware clock these models are generally appropriate for computer and communication systems.

1.4 Performance Evaluation

Performance evaluation is concerned with the description, analysis and optimisation of the dynamic behaviour of computer and communication systems. This involves the investigation of the flow of data, and control information, within and between components of a system. The aim is to understand the behaviour of the system and identify the aspects of the system which are sensitive from a performance point of view. More specifically, a performance analyst might be involved in any of the following tasks: specifying performance requirements, evaluating design alternatives, comparing two or more systems, determining the optimal value of a parameter (*system tuning*), finding the performance bottleneck (*bottleneck identification*), characterising the load on the system (*workload characterisation*), determining the number and sizes of components (*capacity planning*), and predicting the performance at future loads (*forecasting*).

These tasks address problems which are regularly faced by organisations during the life-time of a computer or communication system. For example, consider the following scenarios:

Design and Implementation A car manufacturing company is designing and building a computer-aided design system to allow thirty car designers simultaneous access to a distributed database through graphical workstations. Early in the design phase, fundamental decisions must be made on issues such as the database accessing mechanism and the process synchronisation and communication mechanism. The relative merits of various mechanisms must be evaluated prior to implementation.

Evolution of the configuration and workload A stock exchange intends to begin trading a new class of options. When this occurs, the exchange’s total volume of options transactions is expected to increase by a factor of three. Adequate resources, both computer and personnel, must be in place when the change is implemented.

We can use a variety of *measures*, or *indices*, to characterise the behaviour of the system. Perhaps the most commonly used are *response time*, *throughput* and *utilisation*. Each of these can be regarded as a random variable of the model. The response time is an estimate of the time which elapses between the arrival of a request to the system and the completion of the request by the system. Notice that this is quite a vague definition. This is because response time is a vague notion which should be defined more precisely in terms of the system and the goals of the study whenever it is used. Throughput is the number of requests satisfied per unit time. In both these definitions "request" is used as a general term which, again, should be made precise in the context of any particular model. Utilisation is the percentage of time that a system or component spends in use, as opposed to being idle.

Other measures of interest might be the total number of requests over a given time period, the mean number of requests waiting, or the mean time a request spends waiting while the system is otherwise occupied.

1.5 Probability Theory

Although computer systems by their very nature are deterministic systems the models we will be considering will be *stochastic*. Loosely speaking, a stochastic model is one in which behaviour is random. We use such models because the world that computers interact with is by no means deterministic, but instead full of random behaviour. Also, the stochastic nature of the models will allow us to consider typical or average behaviour rather than particular behaviour. We will rely heavily on basic probability theory when constructing and solving our models.

Probability theory deals with the analysis of events which are unpredictable when viewed individually, but which exhibit some regularity and predictability when viewed in large numbers. We are familiar with such ideas applied to tossing a coin or rolling a die. Here the unpredictability of an individual outcome is due to the non-deterministic nature of the situation. When we are considering a computer system the unpredictability of an individual outcome is likely to arise because of our inability to measure or specify the exact conditions under which the event occurs. This distinction, however, is largely philosophical; probability theory is equally applicable to both these situations.

There are three axioms of probability theory:

1. There is a *sample space*, Ω, which encompasses all possible observations or outcomes.
2. There is a collection of subsets of Ω, denoted E, and termed *events*[1]; these subsets are usually identified as sample points which satisfy some condition.
3. There is a *probability mapping*, Pr, from E to $\mathbb{R}$. Pr must satisfy three simple conditions:
   (a) For any event A, $A \in E$, the mapping Pr is defined and satisfies $0 \leq \Pr(A) \leq 1$.
   (b) $\Pr(\Omega) = 1$.
   (c) If A and B are mutually exclusive, that is, they contain no sample points in common, then $\Pr(A \cup B) = \Pr(A) + \Pr(B)$.

If we consider a mobile phone antenna using frequency division multiplexing the sample space is the complete set of observations. Example events might be that more than three frequencies are in use or that a particular user is making use of the facility. These events corresponds to subsets of the sample points.

[1]Note that this is a different use of the term "event" from discrete *event* system.

Various properties of probabilities can be derived from the axioms and simple set theory. For example, the probability of the union of two events A and B which are **not** mutually exclusive is $\Pr(A \cup B) = \Pr(A) + \Pr(B) - \Pr(A \cap B)$. Similarly, the probability of the complement of event A, denoted by $\neg A$, is $\Pr(\neg A) = 1 - \Pr(A)$.

1.5.1 Conditional Probabilities

If we know that some event has occurred then this may affect the probability that other events have occurred. The *conditional probability* of A occurring, given that B has occurred, is

$$\boxed{\Pr(A \mid B) = \frac{\Pr(A \cap B)}{\Pr(B)}}$$

Example: In a communication network, there are two routes A and B, between a particular pair of nodes which are chosen with equal probability. Message transmission time along route A is 1 second and along route B is 3 seconds. However, the probability of a transmission error is 0.2 on route A and 0.1 on route B. We wish to find the probability of a message being transmitted correctly in 1 second.

Let us denote the event of a successful message transmission by C. We wish to calculate the probability that a transmission takes route A and is successful, i.e. $\Pr(A \cap C)$. We know the conditional probability that a transmission is successful, given that it took route A, is $\Pr(C \mid A) = 0.8$ and the probability that route A is taken is $\Pr(A) = 0.5$. So it follows that $\Pr(A \cap C) = \Pr(C \mid A) \times \Pr(A) = 0.8 \times 0.5 = 0.4$.

If A and B are mutually exclusive $\Pr(A \mid B) = 0$. If B is a precondition for A, then $\Pr(A \cap B) = \Pr(A)$. Two events are *independent* if knowledge of the occurrence of one of them tells us nothing about the probability of the other, i.e. $\Pr(A \mid B) = \Pr(A)$. Thus, when two events are independent, the probability of them both occurring is

$$\boxed{\Pr(A \cap B) = \Pr(A) \times \Pr(B).}$$

1.5.2 Random Variables and Probability Distributions

Often when we conduct an experiment or investigation we are interested in some number associated with the experiment rather than the actual outcome. For example, if we consider the mobile phone antenna again, we are likely to be interested in the average number of frequencies in use rather than the details of each connection. Thus we naturally construct a mapping from the observations we make of the actual interactions, the *sample space*, to the value we are interested in. Such a function, from the sample space to the real or natural numbers is a continuous or discrete *random variable*, respectively. We are often interested in calculating, for each possible value, with what probability the random variable takes that value. Conversely, knowing these probabilities gives us a lot of information about the sample space modelled by the random variable.

When the random variable takes discrete values such as the natural numbers we associate probabilities with the random variable in an intuitive way. That is, the probability that a random variable X has the value v is equal to the probability of the union of all events that lead to $X = v$.

Suppose that random variable X has values $\{x_i \mid i \in I\}$ and that for all $i \in I$ the probability that $X = x_i$ is equal to p_i. Then $\{p_i \mid i \in I\}$ is the *probability distribution* of X. This mapping from x_i to p_i is sometimes called the *mass function.* In the simple example of rolling a fair die, the values of the random variable are clearly $\{1, 2, 3, 4, 5, 6\}$ and the probability distribution is $\{1/6, 1/6, 1/6, 1/6, 1/6, 1/6\}$. This is a *uniform* distribution.

The *expectation* of X is defined to be

$$\boxed{E[X] = \sum_{i \in I} x_i \times p_i.}$$

Informally, we can think of this as the average value that X will have. For rolling a fair die this will be

$$(1 \times 1/6) + (2 \times 1/6) + (3 \times 1/6) + (4 \times 1/6) + (5 \times 1/6) + (6 \times 1/6) = 21/6 = 3.5$$

For some sample spaces, discrete values of random variables are not appropriate. For example, if we are interested in the number of frequencies which are used each hour then a discrete random variable is ideal. However, if we are interested in the duration of each connection, the variable can take almost any non-negative value, so a continuous random variable is appropriate.

For continuous random variables we can no longer assign a probability to each sample point in a meaningful way. Instead we define the probability that the random variable takes a value which is less than a given value. This is done by a *distribution function*[2], F. For a continuous random variable X the distribution function is defined as

$$\boxed{F(x) = \Pr(X \leq x)}$$

Probability distribution functions satisfy four conditions:

1. $F(x_1) \leq F(x_2)$ if $x_1 \leq x_2$;
2. $F(x) \longrightarrow 0$ as $x \longrightarrow -\infty$;
3. $F(x) \longrightarrow 1$ as $x \longrightarrow \infty$;
4. $F(x)$ is continuous[3].

The derivative of the distribution is known as the *density function* $f(x)$,

$$\boxed{f(x) = \frac{dF(x)}{dx}.}$$

It is convenient to regard the infinitesimal quantity $f(x)dx$ as the probability that the value of X lies in the interval $(x, x + dx)$. The *expectation* of the continuous random variable X is defined as

[2]Some authors call this the *cumulative distribution function.*

[3]Strictly, $F(x)$ only needs to be *right continuous*, but we will assume that $F(x)$ is continuous for all the continuous random variables which we will encounter in this course.

$$\boxed{E[X] = \int_{-\infty}^{\infty} xf(x)dx.}$$

The expectation of the sum of a number of random variables is the sum of their individual expectations. This is true even if the variables are dependent in some way. In other words, for continuous random variables, X and Y, $E[X+Y] = E[X] + E[Y]$.

The expectation only gives us an idea of the average value assumed by a random variable, not how much individual values may differ from this average. The *variance*, $Var(X)$, gives us an indication of the "spread" of values:

$$\boxed{Var(X) = E[(X - E[X])^2] = E[X^2] - E[X]^2.}$$

1.5.3 Two important distributions

The most important continuous probability distribution for performance modelling purposes is the *(negative) exponential distribution* which has the distribution function

$$F(x) = 1 - e^{-\mu x} \quad x \geq 0 \qquad \text{and density function} \qquad f(x) = \mu e^{-\mu x} \quad x \geq 0$$

The expectation of an exponential random variable with parameter μ is $1/\mu$ and its variance is $1/\mu^2$. This random variable is often used to denote the time which elapses until some event occurs, such as the arrival of a task at a computer system, the arrival of a bus at a bus-stop or the outbreak of war.

Example: Suppose that the time for which a computer system is up is an exponential random variable with mean 20 days. What is the probability that the machine will stay up for more than 40 days?

Let X denote the random variable representing the up-time of the machine. Then we are interested in $\Pr(X > 40) = 1 - \Pr(X \leq 40)$ (i.e. the distribution function evaluated when $x = 40$ and $\mu = 1/20$):

$$\Pr(X > 40) = 1 - (1 - e^{-40/20}) = e^{-2} = 0.135 \qquad \text{to 3 significant figures}$$

The exponential distribution function is closely related to a discrete random variable, the *Poisson* distribution. This random variable takes values in the set $\{0, 1, 2, \ldots\}$ and has the mass function

$$p_i = e^{-\mu} \frac{\mu^i}{i!} \qquad i \geq 0.$$

The expectation of a Poisson random variable with parameter μ is also μ. The Poisson random variable arises frequently in computer and communication system modelling. It is typically used as a counting variable, recording the number of events that occur in a given period of time; for example, the number of messages transmitted in a communication network node in one minute or the number of users arriving at a terminal in one hour. If we observe a Poisson process with parameter μ for some time period of length h then:

- the probability that one event occurs is $\mu h + o(h)$.[4]

[4]The notation $o(h)$ denotes terms of small relative size; formally a function f is $o(h)$ if $\lim_{h \to 0} \frac{f(h)}{h} = 0$ In practice, what this means is that these are terms which we can ignore when h is small.

- the probability that two or more events occur is $o(h)$.
- the probability that no events occur is $1 - \mu h + o(h)$.

Thus if we observe a Poisson process for a infinitesimal time period dt the probability that an event occurs is μdt. If the occurrence of events is governed by a Poisson distribution then the inter-event times are governed by an exponential distribution with the same parameter, and vice versa. Therefore, if we know that the delay until an event is exponentially distributed then the probability that it will occur in the infinitesimal time interval of length dt, is μdt.

The reason that the exponential distribution is used extensively in performance modelling is because it has several attractive mathematical features. Here we will state those properties without proof, but the proofs can be found in any textbook on stochastic modelling, Markov processes or performance modelling.

- The exponential distribution is *memoryless*: if X is an exponentially distributed random variable, then

  $$\Pr(X > t + s | X > t) = \Pr(X > s) \qquad t > 0, s > 0$$

  If X represents the waiting time until an event, this implies that knowing that t time units have already elapsed without the event occuring does not give us any additional information about when the event will occur—the distribution of the further waiting time is the same as if no waiting time had already passed.
- If X_1 and X_2 are two exponentially distributed random variables, with parameters λ_1 and λ_2 respectively, and Y is defined to be their minimum, i.e. $Y = \min(X_1, X_2)$, then Y is also exponentially distributed, with parameter $\lambda_1 + \lambda_2$. This is sometimes called the *superposition* property.
- Let X be an exponentially distributed random variable, with parameter λ, representing time until an event. Suppose that the events are probabilistically divided into two categories, with events belonging to stream A with probability p_A and belonging to stream B with probability $p_B = 1 - p_A$. Then stream A and stream B are Poisson streams with parameters $\lambda \times p_A$ and $\lambda \times p_B$ respectively, and the waiting times between events of category A and between events of category B are exponentially distributed with parameters $\lambda \times p_A$ and $\lambda \times p_B$ respectively. This is sometimes called the *decomposition property.*

1.6 The rest of the course

The central theme of the course will be that a *model*, as well as being an abstract representation of a system, is a tool which we can exploit to derive information about the system. The more detail we invest in the model, the more sophisticated the information we can extract from it. As the course progresses the models will become increasingly detailed; the corresponding solution techniques will similarly become more complex, relying on increasing levels of computer assistance.

Jane Hillston ⟨jeh@inf.ed.ac.uk⟩. October 7, 2005.

2 Operational Laws

2.1 Introduction

Operational laws are simple equations which may be used as an abstract representation or model of the average behaviour of almost any system. One of the advantages of the laws is that they are very general and make almost no assumptions about the behaviour of the random variables characterising the system[1]. Another advantage of the laws is their simplicity: this means that they can be applied quickly and easily by almost anyone.

Based on a few simple observations of the system the performance analyst can, by applying these simple laws, derive more information. Using this information as input to further laws the analyst gradually builds up a more complete picture of the behaviour of the system. Note that although we will talk in this section about operational laws in the context of systems, the laws are equally applicable to the observations obtained from models and we will have occasion to use the laws in this way later in the course.

The foundation of the operational laws are *observable variables*. These are values which we could derive from watching a system over a finite period of time. We assume that the system receives *requests* from its environment. Each request generates a *job* or *customer* within the system. When the job has been processed the system responds to the environment with the completion of the corresponding request.

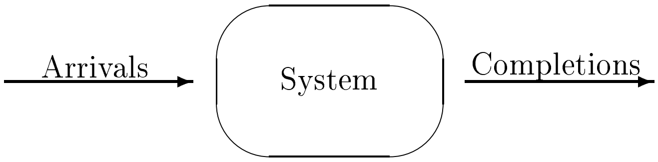


Figure 1: An Abstract System

If we observed such an abstract system we might measure the following quantities:

T, the length of *time* we observe the system;

A, the number of request *arrivals* we observe;

C, the number of request *completions* we observe;

B, the total amount of time during which the system is *busy* ($B \leq T$);

N, the average number of jobs in the system.

From these observed values we can derive the following four important quantities:

$\lambda = A/T$, the *arrival rate*;

$X = C/T$, the *throughput* or *completion rate*,

$U = B/T$, the *utilisation*;

$S = B/C$, the *mean service time* per completed job.

[1] In contrast, Markovian analysis relies on very strong assumptions about the distribution function of the random variables which are used.

We will assume that the system is *job flow balanced.* This means that the number of arrivals is equal to the number of completions during an observation period, i.e. $A = C$. Obviously this assumption is not true in all observation periods, but it is a *testable assumption* because an analyst can always test whether the assumption holds. Note that if the system is job flow balanced the arrival rate will be the same as the completion rate, that is, $\lambda = X$.

2.2 Little's Law

The best known and most commonly used operational law is Little's law. It is named after the man who published the first formal proof of the law in 1961, although it had been widely used before that time. Little's law is usually phrased in terms of the *jobs* in a system and relates the average number of jobs in the system N to the *residence time* W, the average time they spend in the system. Let X be the throughput, as above. Then Little's law states that

$$\boxed{N = XW}$$

The average number of jobs in a system is equal to the product of the throughput of the system and the average time spent in that system by a job.

Given a computer system, Little's law can be applied at many different levels: to a single resource, to a subsystem or to the system as a whole. A little care may be necessary if the law is applied in this way, as the definitions of the number of jobs, throughput and residence time used at the different levels must be compatible with each other. At different levels of detail, different definitions of "request" are appropriate. For example, when considering a disk, it is natural to define a request to be a disk access, and to measure throughput and residence time on this basis. When considering an entire transaction processing system, on the other hand, it is natural to define a request to be a user-level transaction, and to measure throughput and residence time on this basis. Each such transaction may generate several disk accesses. We will return to this idea of systems, or subsystems, within a system in the following subsection.

Example: Consider a disk that serves 40 requests/second ($X = 40$) and suppose that on average there are 4 requests present in the disk system (waiting to be served or in service) ($N = 4$). Then Little's law tells us that the average time spent at the disk by a request must be $4/40 = 0.1$ seconds. If we know that each request requires 0.0225 seconds of disk service we can then deduce that the average queueing time is 0.0775 seconds.

2.3 Forced Flow Law

It is often natural to regard a system as being made up of a number of devices or resources. Each of these resources may be treated as a system in its own right as far as the operational laws are concerned, with the rest of the system forming the environment of that resource. A request from the environment generates a job within the system; this job may then circulate between the resources until all necessary processing has been done; as it arrives at each resource it is treated as a request, generating a job internal to that resource.

Suppose that during an observation interval we count not only completions external to the system, but also the number of completions at each resource within the system. We define the *visit count*, V_i, of the ith resource to be the ratio of the number of completions at that resource to the number of system completions

$$V_i \equiv C_i/C.$$

More intuitively, we might think of this as the average number of visits that a system-level job makes to that resource. For example, if, during an observation interval, we measure 10 system completions and 150 completions at a specific disk, then on the average each system-level request requires 15 disk operations.

The *forced flow law* captures the relationship between the different components within a system. It states that the throughputs or *flows*, in all parts of a system must be proportional to one another. In other words, it relates the throughput at the individual resources ($X_i = C_i/T$) to the throughput at the complete system ($X = C/T$). It is stated as follows

$$\boxed{X_i = XV_i}$$

The throughput at the ith resource is equal to the product of the throughput of the system and the visit count at that resource.

An informal interpretation of this law is that, since the visit count defines the number of visits to a resource or device that each job needs in order to complete its processing, the resource must keep up a correspondingly scaled completion rate to ensure that the system completion rate is maintained.

Example: Consider a robotic workcell within a computerised manufacturing system which processes *widgets*. Suppose that processing each widget requires 4 accesses to the lathe and 2 accesses to the press. We know that the lathe processes 8 widgets in a minute and we want to know the throughput of the press. The throughput of the workcell will be proportional to the lathe throughput, i.e. $X = X_{lathe}/V_{lathe} = 8/4 = 2$. The throughput of the press will be $X_{press} = X \times V_{press} = 2 \times 2 = 4$. Thus the press throughput is 4 widgets per minute.

2.4 Utilisation Law

If we know the amount of processing that each job requires at a resource then we can calculate the utilisation of the resource. Let us assume that each time a job visits the ith resource the amount of processing, or *service*, time it requires is S_i. Note that service time is not necessarily the same as the residence time of the job at that resource: in general a job might have to wait for some time before processing begins. The total amount of service that a system job generates at the ith resource is called the *service demand*, D_i:

$$D_i = S_i V_i$$

The utilisation of a resource, the percentage of time that the ith resource is in use processing to a job, is denoted U_i. The *utilisation law* states that

$$\boxed{U_i = X_i S_i = X D_i}$$

The utilisation of a resource is equal to the product of the throughput of that resource and the average service requirement at that resource.

Example: Consider again the disk that is serving 40 requests/second, each of which requires 0.0225 seconds of disk service. The utilisation law tells us that the utilisation of the disk must be $40 \times 0.0225 = 90\%$.

2.5 General Residence Time Law

One method of computing the mean residence or response time per job in a system is to apply Little's law to the system as a whole. However, if the mean number of jobs in the system, N, or the system level throughput, X, are not known an alternative method can be used. Applying Little's law to the ith resource we see that $N_i = X_i W_i$, where N_i is the mean number of jobs at the resource and W_i is the average response time of the resource.

From the forced flow law we know that $X_i = X V_i$. Thus we can deduce that

$$N_i / X = V_i W_i.$$

The total number jobs in the system is clearly the sum of the number of jobs at each resource, i.e. $N = N_1 + \cdots + N_M$ if there are M resources in the system. We know from Little's law that $W = N/X$ and from this we arrive at the *general residence time*, or *general response time law*:

$$\boxed{W = \sum_{i=1}^{M} W_i V_i}$$

The average residence time of a job in the system will be the sum of the product of its average residence time at each resource and the number of visits it makes to that resource.

Example: A program running on a compute server requires 126 bursts of CPU time and makes 75 I/O requests to disk A and 50 I/O requests to disk B. On average each CPU burst requires 30 milliseconds (waiting + processing time). Monitoring has shown that the throughput of disk A is 15 requests per second and the average number in the buffer is 4 whilst at disk B the throughput is 10 requests per second and the average number in the buffer is 3. Using Little's Law we calculate the residence time at each of the disks (remembering that the number in the system is the number in the buffer +1):

$$W_{diskA} = \frac{N_{diskA}}{X_{diskA}} = \frac{5}{15/1000} = \frac{5000}{15} \qquad W_{diskB} = \frac{N_{diskB}}{X_{diskB}} = \frac{4}{10/1000} = \frac{4000}{10}$$

Then

$$\begin{aligned} W &= W_{CPU} V_{CPU} + W_{diskA} V_{diskA} + W_{diskB} V_{diskB} \\ &= 30 \times 126 + \frac{5000}{15} \times 75 + \frac{4000}{10} \times 50 = 3780 + 25000 + 20000 = 48780 milliseconds \end{aligned}$$

2.6 Interactive Response Time Law

The name of this law dates back to the time when most of the systems which were being modelled were mainframes processing both interactive jobs and batch jobs. The *think time*, Z, was quite literally the length of time that a programmer spent thinking at his terminal before submitting another job. More generally interactive systems are those in which jobs spend time in the system not engaged in processing, or waiting for processing: this may be because of interaction with a human user, or may be for some other reason. For example, if we are studying a cluster of workstations with a central file server to investigate the load on the file server, the think time might represent the average time that each workstation spends processing locally without access to the file server. At the end of this non-processing period the job generates a fresh request.

The key feature of such a system is that the residence time can no longer be taken as a true reflection of the response time of the system. The think time represents the time between processing being completed and the job becoming available as a request again. Thus the residence time of the job, as calculated by Little's law as the time from arrival to completion, is greater than the system's response time. The *interactive response time law* reflects this: it calculates the *response time*, R as follows:

$$\boxed{R = N/X - Z}$$

The response time in an interactive system is the residence time minus the think time.

Note that if the think time is zero, $Z = 0$ and $R = W$, then the interactive response time law simply becomes Little's law.

Example: Suppose that the library catalogue system, has 64 interactive users, that the average think time is 30 seconds, and that system throughput is 2 interactions/second. Then the interactive response time law tells us that the response time must be $64/2 - 30 = 2$ seconds.

2.7 Bottleneck analysis

The resource within a system which has the greatest service demand is known as the *bottleneck resource* or *bottleneck device*, and its service demand is $\max_i\{D_i\}$, denoted D_{max}. The bottleneck resource is important because it limits the possible performance of the system. This will be the resource which has the highest utilisation in the system.

The residence time of a job within a system will always be at least as large as the total amount of processing that each job requires—this will be the time that the job takes even if it never has to wait for a resource. The total amount of processing that a job requires is D, the total service demand, $D = \sum_{i=1}^{M} D_i$. In general, there will be some contention in the system meaning that jobs have to wait for processing so the residence time will be larger than this, i.e.

$$W \geq D$$

The throughput of a system will always be limited by the throughput at the slowest resource (think of the forced flow law); this is the bottleneck device. By the utilisation law, at this resource, let's call it b, $U_b = XD_{max}$. Therefore, since $U_b \leq 1$

$$X \leq 1/D_{max}$$

It follows that if we wish to improve throughput we should first concentrate on this resource—improving throughput at other resources in the system might have little effect on the overall performance.

Using Little's law or the interactive response time law, we can derive a tighter bound on the response time which applies when the system is heavily loaded (i.e. the mean number of jobs, N, is high). Applying the interactive response time law to the throughput bound, $X \leq 1/D_{max}$ we obtain:

$$R = N/X - Z \geq ND_{max} - Z$$

Applying Little's law we obtain $W \geq ND_{max}$. Thus the asymptotic bound for residence time or response time is:

$$\boxed{W \geq \max\{D, ND_{max}\}} \qquad \boxed{R \geq \max\{D, ND_{max} - Z\}}$$

Similarly the bound on the throughput of an interactive system may be made tighter when the system is lightly loaded (i.e. the mean number of jobs, N, is small). From the interactive response time law:

$$X = N/(R + Z) \leq N/(D + Z)$$

Applying Little's law (when $Z = 0$) we obtain $X \leq N/D$.

$$\boxed{X \leq \min\{1/D_{max}, N/(D + Z)\}}$$

Notice that the bottleneck depends on both resource parameters (X_i or S_i) and the workload parameters (V_i). If we change the number of visits that each job makes to a resource we might move the bottleneck.

2.8 Assumptions

As mentioned in the introduction, the operational laws do not rely on many assumptions. Indeed the only explicit assumption we have made is that the system is *job flow balanced*—the same number of requests are completed by the system as arrive at the system. We are also implicitly assuming that this holds at each of the resources or devices within a system. A consequence of this is that jobs are not created or destroyed anywhere in the system. This is sometimes called *conservation of work*.

We have also assumed that the system is *homogeneous*, that is, that the behaviour of jobs or resources within a system does not depend on the global state of the system. For example, a job cannot alter the number of visits it makes to a particular resource because that resource is heavily loaded.

3 Constructing and Solving Markov Processes

3.1 Stochastic Processes

Formally, a stochastic model is one represented as a *stochastic process*; a stochastic process is a set of random variables $\{X(t), t \in T\}$. T is called the *index set* and in all the models we consider T will be taken to represent time. Since we consider continuous time models $T = \mathbb{R}$, the set of real numbers. The *state space* of the process is the set of all possible values that the random variables $X(t)$ can assume. Each of these values is called a *state* of the process. These states will correspond to our intuitive notion of states within the system represented, although there will not necessarily be a one-to-one relation. Any set of instances of $\{X(t), t \in T\}$ can be regarded as a path of a particle moving randomly in the state space, S, its position at time t being $X(t)$. These paths are called *sample paths* or *realisations* of the stochastic process.

For example, if we consider the mobile phone antenna with frequency division multiplexing again, we might associate one random variable with each frequency and associate the values 1 and 0 with the frequency being used or not. At any time the random variables representing the system will be the set of random variables representing each of the frequencies. Thus the state of the system is an n-tuple, where n is the number of frequencies and the entry in each position represents the state of the individual frequency. The state will change whenever a new connection is made or an existing connection is terminated. Alternatively, we might consider the stochastic process characterised by a single random variable, M, which records the number of frequencies in use. Now a single state at the stochastic process level will correspond to several states of the system depending on which frequencies are in use. This example shows how the degree of abstraction influences the modelling process.

The stochastic processes we will be considering during the first half of the course will all have the following properties:

$\{X(t)\}$ is a Markov process. This implies that $\{X(t)\}$ has the *Markov* or *memoryless property*: given the value of $X(t)$ at some time $t \in T$, the future path $X(s)$ for $s > t$ does not depend on knowledge of the past history $X(u)$ for $u < t$, i.e. for $t_1 < \cdots < t_n < t_{n+1}$,

$$\Pr(X(t_{n+1}) = x_{n+1} \mid X(t_n) = x_n, \ldots, X(t_1) = x_1) = \Pr(X(t_{n+1}) = x_{n+1} \mid X(t_n) = x_n)$$

$\{X(t)\}$ is irreducible. This implies that all states in S can be reached from all other states, by following the transitions of the process.

$\{X(t)\}$ is stationary: for any $t_1, \ldots t_n \in T$ and $t_1 + \tau, \ldots, t_n + \tau \in T$ $(n \geq 1)$, then the process's joint distributions are unaffected by the change in the time axis and so,

$$F_{X(t_1+\tau)\ldots X(t_n+\tau)} = F_{X(t_1)\ldots X(t_n)}$$

$\{X(t)\}$ is time homogeneous: the behaviour of the system does not depend on when it is observed. In particular, the transition rates between states are independent of the time at which the transitions occur. Thus, for all t and s, it follows that
$\Pr(X(t + \tau) = x_k \mid X(t) = x_j) = \Pr(X(s + \tau) = x_k \mid X(s) = x_j)$.

3.2 Markov Processes

In this course, our primary objective with respect to a Markovian model will be to calculate the probability distribution of the random variable $X(t)$ over the state space S, as the system settles into a regular pattern of behaviour. This is termed the *steady state probability distribution*. From this probability distribution we will derive performance measures based on subsets of states where some condition holds. Markov processes are widely used because it is well-understood how to solve them.

The memoryless property means that once we have arrived at a particular state the future behaviour is always the same regardless of how we arrived in the state. The solution of Markov processes is closely related to their representation by matrices. Finding the average behaviour of the model then corresponds to solving a simple matrix equation. This will be discussed in detail in the remainder of this note.

Like any stochastic process, a Markov process is characterised by a random variable $X(t)$, indexed over time t, and a state space S such that $X(t) \in S$ for all t. The dynamic behaviour of the system being modelled is represented by the transitions between these states, and times spent in states, *sojourn times*. In general, these holding times represent the periods while processing is occurring in the system, the transitions represent events in the system.

If a state $i \in S$ is entered at time t and the next state transition takes place at time $t + T$, T is the sojourn time in state i. By the Markov property, at any time point τ, the distribution of the time until the next change of state is independent of the time of the previous change of state. In other words, sojourn times are memoryless. Since the only probability distribution function which has this property is the exponential distribution this tells us that the sojourn time in any state of a Markov process is an exponentially distributed random variable. Hence at time τ, the probability that there is a state transition in the interval $(\tau, \tau + dt)$ is $q_i dt + o(dt)$, where q_i is the parameter of the exponential distribution of the sojourn time in state i.

Suppose that when a transition out of state i occurs, the new state is j with probability p_{ij}. By the Markov property, this must only depend on i and j. Thus we have, for $i \neq j$, $i, j \in S$,

$$\Pr(X(t + dt) = j \mid X(t) = i) = q_{ij} dt + o(dt)$$

where the $q_{ij} = q_i p_{ij}$, by the decomposition property. The q_{ij} are called the *instantaneous transition rates*, and if the average time delay before a transition from i to j is exponentially distributed with parameter μ, then $q_{ij} = \mu$.

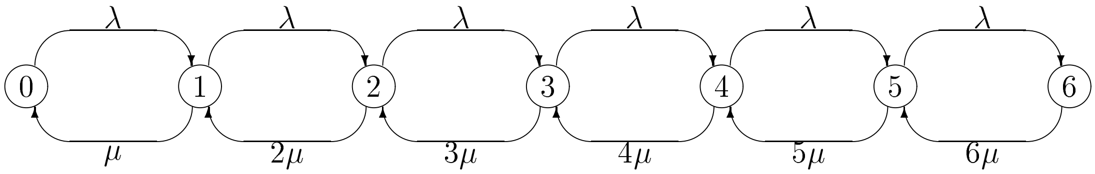
In practice, we are usually given the q_{ij} because we will know the parameter of the distribution governing the time delay associated with the event which is represented by the state transition. For example, we might be told that the average duration of a call using the mobile phone antenna is $1/\mu$ seconds. If we are using Markovian analysis, we then *choose to assume* that the duration is exponentially distributed with parameter μ.

From the q_{ij} we can derive the *exit rate*, q_i, and the transition probabilities, p_{ij}, as follows. The exit rate is the rate at which the system leaves state i, i.e. it is the parameter of the exponential distribution governing the sojourn time. This will be the minimum of the delays until any of the possible transitions occurs. Thus, by the superposition property it is the sum of the individual transition rates, i.e. $q_i = \sum_{j \in S} q_{ij}$. The transition probability p_{ij} is the probability, given that a transition out of state i occurs, that it is the transition to state j. By the definition of conditional probability, this is $p_{ij} = q_{ij}/q_i$.

3.2.1 State transition diagrams

For small Markov processes the simplest way to represent the process is often in terms of its state transition diagram. In such a diagram we represent each state of the process as a node in a graph. The arcs in the graph represent possible transitions between states of the process. The arcs are labelled by the *transition rates* between states. Since every transition is assumed to be governed by an exponential distribution, the rate of the transition will be the parameter of the distribution. Irreducibility implies that the state transition diagram must be strongly connected.

Example: Consider again the mobile phone antenna taking the simplest representation of the states of the system, i.e. the number of frequencies in use. Then, if there are 6 frequencies, the state space is $\{0, 1, 2, 3, 4, 5, 6\}$. The only possible transition from state 0 is to state 1, caused by a new connection. Similarly, from state 6 the only possible transition is to state 5, caused by a frequency being released. For any other state, j, $1 \leq j \leq 5$ there are two possible transitions, to states $j+1$ or $j-1$, caused by a new connection or release of an old one, respectively. Let us assume that the arrival of calls form a Poisson stream with parameter λ: the rate at which $j \to j+1$ transitions occur is λ. As above, we assume that the mean duration of a connection is $1/\mu$, so that the rate at which release transitions occur on each frequency is μ. Then the state transition diagram for this process is:



3.2.2 Generator matrices

A Markov process with n states can also be characterised by its $n \times n$ *infinitesimal generator matrix*, $\mathbf{Q}$. The entry in the jth column of the ith row of the matrix $(j \neq i)$ will be q_{ij}, the instantaneous transition rate. In other words, it will be the sum of the parameters labelling arcs connecting nodes i and j in the state transition diagram. The diagonal elements are chosen to ensure that the sum of the elements in every row is zero, i.e. $q_{ii} = -\sum_{j \in S, j \neq i} q_{ij}$.

Example: For the mobile phone antenna example, whose state transition diagram is shown in the previous section, the infinitesimal generator matrix is:

$$\mathbf{Q} = \begin{pmatrix} -\lambda & \lambda & 0 & 0 & 0 & 0 & 0 \\ \mu & -(\mu+\lambda) & \lambda & 0 & 0 & 0 & 0 \\ 0 & 2\mu & -(2\mu+\lambda) & \lambda & 0 & 0 & 0 \\ 0 & 0 & 3\mu & -(3\mu+\lambda) & \lambda & 0 & 0 \\ 0 & 0 & 0 & 4\mu & -(4\mu+\lambda) & \lambda & 0 \\ 0 & 0 & 0 & 0 & 5\mu & -(5\mu+\lambda) & \lambda \\ 0 & 0 & 0 & 0 & 0 & 6\mu & -6\mu \end{pmatrix}$$

3.3 Steady state distribution

Performance analysis is usually concerned with the behaviour of systems over an extended period of time. The running time of the system will be very large relative to the individual inter-event times within the system. From a modelling point of view this is important because we may have to choose an initial state of our model arbitrarily. Considering the long term probability distribution will overcome any bias introduced by the chosen starting state. This will have serious impact on how we use models when we consider simulation models, because each *run* of the model is a particular *sample path* of the model over its state space. For Markovian models we avoid the problems of bias because we do not consider individual sample paths but the probability distribution over the states of the process which we can deduce knowing the transition rates between states.

We assume that when the effects of initial bias have been overcome, the model, and the system, will enter into *equilibrium* or *steady state* behaviour. This does not mean that the model has entered one state and no longer makes transitions. It does mean that the model is assumed to exhibit regularity and predictability in its behaviour over the state space. Thus, the probability distribution of the random variable $X(t)$ over the state space S, will be no longer changing.

Let us denote by $\pi_k(t)$ the probability that if we observe the Markov process at time t it will be in state x_k, for $x_k \in S$. Equilibrium has been reached when, for all states $x_k \in S$, $\pi_k(t+\tau) = \pi_k(t)$, i.e. the time at which we observe the model does not influence the probability that it is in a particular state. Thus we denote the *steady state probabilities* without the time variable, e.g. $\boldsymbol{\pi_k}$, for the probability that the model is in state x_k.

The steady state distribution is useful to us because the following theorem tells us that, for the types of model we are considering, the steady state distribution is exactly the same as the long term probability distribution over states.

Theorem: A steady state probability distribution, $\{\boldsymbol{\pi_k}, x_k \in S\}$, exists for every time homogeneous, finite, irreducible Markov process. Moreover, this distribution is the same as the limiting distribution

$$\boxed{\lim_{t\to\infty} \Pr(X(t) = x_k \mid X(0) = x_0) = \boldsymbol{\pi_k}}$$

3.3.1 Global Balance Equations

In steady state, $\boldsymbol{\pi_i}$ is the proportion of time that the process spends in state x_i. Recall that the transition "rate" q_{ij} is the instantaneous probability that the model makes a transition from state x_i to state x_j. Thus, in an instant of time, the probability that a transition will occur from state x_i to state x_j is the probability that the model was in state x_i, $\boldsymbol{\pi_i}$, multiplied by the transition rate q_{ij}. This is called the *probability flux* from state x_i to state x_j.

When the model is in steady state, in order to maintain the equilibrium, we must assume that the total probability flux out of a state is equal to the total probability flux into the state. If this were not the case the distribution over states would change. So, for any particular state x_i, this implies

$$\boxed{\underbrace{\boldsymbol{\pi_i} \times \sum_{x_j \in S, j \neq i} q_{ij}}_{\text{flux } \textbf{out} \text{ of } x_i} = \underbrace{\sum_{x_j \in S, j \neq i} (\boldsymbol{\pi_j} \times q_{ji})}_{\text{flux } \textbf{into} \ x_i}}$$

The collection of these equations for all states $x_i \in S$ is called the *global balance equations*. If we recall that the diagonal elements of the infinitesimal generator matrix $\mathbf{Q}$ are the negative sum of the other elements in the row (i.e. $q_{ii} = -\sum_{x_j \in S, j \neq i} q_{ij}$) we can rearrange the equation above to be:

$$\sum_{x_j \in S} \boldsymbol{\pi_j} q_{ji} = 0.$$

Expressing the values $\boldsymbol{\pi_i}$ as a row vector $\boldsymbol{\pi}$, we can write this as a matrix equation:

$$\boxed{\boldsymbol{\pi} \ \mathbf{Q} = 0}$$

The $\boldsymbol{\pi_i}$ are unknown; they are the values we wish to find since they are the probability distribution. If there are n states in the state space there are n such equations in n unknowns. Unfortunately, this collection of equations is irreducible: we need another equation in order to solve the equations and find the unknowns. Fortunately, since $\{\boldsymbol{\pi_i}\}$ is a probability distribution we also know that the *normalisation condition* holds:

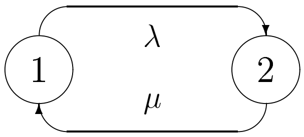
$$\boxed{\sum_{x_i \in S} \boldsymbol{\pi_i} = 1}$$

With these $n + 1$ equations we can solve to find the n unknowns, $\{\boldsymbol{\pi_i}\}$.

Example: Consider a system with multiple CPUs, each with its own private memory, and one common memory which can be accessed only by one processor at a time. The CPUs execute in private memory for a random time before issuing a common memory access request. Assume that this random time is exponentially distributed with parameter λ (the average time a CPU spends executing in private memory between two common memory access requests is $1/\lambda$). The common memory access duration is also assumed to be exponentially distributed, with parameter μ (the average duration of a common memory access is $1/\mu$). If the system has only one processor, it has only two states:

1. The processor is executing in its private memory;
2. The processor is accessing common memory.

The system behaviour can be modelled by a 2 state Markov process whose state transition diagram and generator matrix are as shown below:



$$\mathbf{Q} = \begin{pmatrix} -\lambda & \lambda \\ \mu & -\mu \end{pmatrix}$$

If we consider the probability flux in and out of state 1 we obtain: $\boldsymbol{\pi_1}\ \lambda = \boldsymbol{\pi_2}\mu$
Similarly, for state 2 [1]: $\boldsymbol{\pi_2}\ \mu = \boldsymbol{\pi_1}\lambda$

Finally, we know from the normalisation condition that: $\boldsymbol{\pi_1} + \boldsymbol{\pi_2} = 1$

It follows that the steady state probability distribution is $\boldsymbol{\pi} = \left(\frac{\mu}{\mu+\lambda}, \frac{\lambda}{\mu+\lambda}\right)$. From this information we can deduce, for example that the probability that the processor is executing in its private memory is $\mu/(\mu+\lambda)$.

3.3.2 Solving the global balance equations

Once we are considering models which have more than a few of states it is unlikely that we want to solve the equations ourselves by hand. Instead we take advantage of the matrix equation formulation of the global balance equations and use suitable software packages. Any linear algebra package will solve matrix equations of the form $\mathbf{Ax} = \mathbf{b}$, where $\mathbf{A}$ is an $n \times n$ matrix, $\mathbf{x}$ is a column vector of n unknowns, and $\mathbf{b}$ is a column vector of n values. There are two problems which have to be resolved before we can take advantage of such software:

1. Our global balance equation comes in the "wrong" form $\boldsymbol{\pi}\ \mathbf{Q} = 0$: the unknowns, $\boldsymbol{\pi}$ are given as a row vector not a column vector. This problem is resolved by transposing[2] the equation, i.e. $\mathbf{Q}^T\boldsymbol{\pi} = 0$, where the right hand side is now a column vector of zeros, rather than a row vector.
2. The second problem is the redundancy amongst the global balance equations—even after transposing there is insufficient information to solve for the unknowns. As already noted, we need the additional information of the normalisation condition. This problem is resolved by replacing one of the global balance equations by the normalisation condition. In the transposed matrix this corresponds to replacing one row by a row of 1's. We usually choose the last row and denote the modified matrix $\mathbf{Q}_n^T$. Similarly we change the "solution" vector, which was all zeros to be a column vector with 1 in the last row, and zeros everywhere else. We denote this vector, $\mathbf{e}_n$.

Now we can use any linear algebra solution package, such as `maple` or `xmaple` to solve the resulting equation[3]:

$$\mathbf{Q}_n^T\boldsymbol{\pi} = \mathbf{e}_n$$

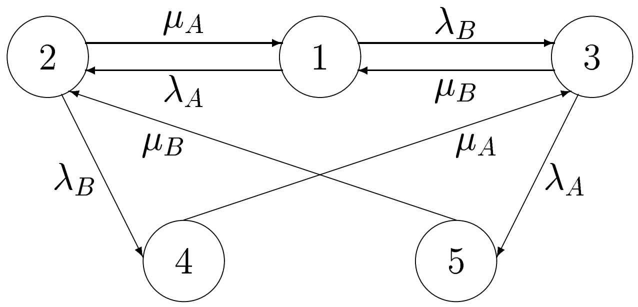
[1]Note, here the two equations are in fact the same. There will always be a redundant equation in the global balance equations which is why we need the additional information of the normalisation condition.

[2]i.e. rearranging the matrix so that the rows are columns and vice versa.

[3]Under DICE type `setup maple7` before typing `maple` or `xmaple`

Example: Consider the two-processor version of the system introduced in the previous section. We denote the processors A and B, respectively. We assume that the processors have different timing characteristics, the private memory access of A being governed by an exponential distribution with parameter λ_A, the common memory access of B being governed by an exponential distribution with parameter μ_B, etc. Now the state space becomes:

1. A and B both executing in their private memories;
2. B executing in private memory, and A accessing common memory;
3. A executing in private memory, and B accessing common memory;
4. A accessing common memory, B waiting for common memory;
5. B accessing common memory, A waiting for common memory;



$$\mathbf{Q} = \begin{pmatrix} -(\lambda_A+\lambda_B) & \lambda_A & \lambda_B & 0 & 0 \\ \mu_A & -(\mu_A+\lambda_B) & 0 & \lambda_B & 0 \\ \mu_B & 0 & -(\mu_B+\lambda_A) & 0 & \lambda_A \\ 0 & 0 & \mu_A & -\mu_A & 0 \\ 0 & \mu_B & 0 & 0 & -\mu_B \end{pmatrix}$$

This matrix is still small enough to be solved by hand but for the purposes of illustration we solve it using `maple`. The transcript of the `maple` session is given at the end of this note. After appropriate transformations the modified generator matrix is as follows:

$$\mathbf{Q}_n^T = \begin{pmatrix} -(\lambda_A+\lambda_B) & \mu_A & \mu_B & 0 & 0 \\ \lambda_A & -(\mu_A+\lambda_B) & 0 & 0 & \mu_B \\ \lambda_B & 0 & -(\mu_B+\lambda_A) & \mu_A & 0 \\ 0 & \lambda_B & 0 & -\mu_A & 0 \\ 1 & 1 & 1 & 1 & 1 \end{pmatrix}$$

If we choose the following values for the parameters:

$$\lambda_A = 0.05 \qquad \lambda_B := 0.1 \qquad \mu_A = 0.02 \qquad \mu_B = 0.05$$

solving the matrix equation, and rounding figures to 4 significant figures, we obtain:

$$\boldsymbol{\pi} = (0.0693, 0.0990, 0.1683, 0.4951, 0.1683)$$

3.4 Derivation of Performance Measures

Roughly speaking, there are three ways in which performance measures can be derived from the steady state distribution of a Markov process. These different methods can be thought of as corresponding to different types of measure:

- state-based measures, e.g. utilisation;
- rate-based measures, e.g. throughput;
- other measures which fall outside the above categories, e.g. response time.

State-based measures are those which clearly correspond to the probability that the model is in a state, or a subset of states, which satisfy some condition. For example, utilisation will correspond to those states where a resource is in use. If we consider the example above, the utilisation of the common memory, U_{mem}, is the total probability that the model is in one of the states in which the common memory is in use:

$$U_{mem} = \boldsymbol{\pi_2} + \boldsymbol{\pi_3} + \boldsymbol{\pi_4} + \boldsymbol{\pi_5} = 93.07\%$$

Other examples of state-based measures are idle time, or the number of jobs in a system. Some measures such as the number of jobs will involve a weighted sum of steady state probabilities, weighted by the appropriate value. For example, if we consider jobs waiting for the common memory to be queued in that subsystem, then the average number of jobs in the common memory, N_{mem}, is:

$$N_{mem} = (1 \times \boldsymbol{\pi_2}) + (1 \times \boldsymbol{\pi_3}) + (2 \times \boldsymbol{\pi_4}) + (2 \times \boldsymbol{\pi_5}) = 1.594$$

Rate-based measures are those which correspond to the predicted rate at which some event occurs. This will be the product of the rate of the event, and the probability that the event is enabled, i.e. the probability of being in one of the states from which the event can occur. Thus, to calculate the throughput of the common memory, the average number of accesses from either processor which it processes in unit time, X_{mem}, we use

$$X_{mem} = (\mu_A \times (\boldsymbol{\pi_2} + \boldsymbol{\pi_4})) + (\mu_B \times (\boldsymbol{\pi_3} + \boldsymbol{\pi_5})) = 0.0287$$

or, approximately one access every 35 milliseconds.

Finally, the other category of measures are those which do not readily fit into either of the above categories. In these cases, we usually use one of the operational laws to derive the information we need, based on values that we have obtained from solution of the model. Thus, applying Little's law to the common memory we see that

$$W_{mem} = N_{mem}/X_{mem} = 1.594/0.0287 = 55.54 \text{ milliseconds}$$

3.5 Assumptions

The most fundamental assumption we have made is the **stochastic hypothesis**: *"The behaviour of a real system during a given period of time is characterised by the probability distributions of a stochastic process."* Although it can never be conclusively proved, evidence suggests that this hypothesis is true. In contrast, the assumption that all delays and inter-event times are exponentially distributed, is less likely to fit with observations of real systems. We make this assumption nevertheless because of the nice mathematical properties of the exponential distribution, and because the exponential distribution is the only one we can use if we wish to assume that the stochastic process characterising our system is a Markov process. The Markov assumption that future behaviour of the system is only dependent on its current state, not on its past history *is* a reasonable assumption for computer and communication systems, if we choose our states carefully. We assume that the Markov process is finite, time homogeneous and irreducible: necessary conditions for a steady state distribution to exist, and coincide with the long term probability distribution.

3.6 Transcript of maple session

```
> with(linalg):
>
> lambdaA := 0.05;
> muA := 0.02;
> lambdaB := 0.1;
> muB := 0.05;
>
> size := 5;
>
> Q := array(sparse,1..size,1..size);
> e := array(sparse,1..size);
>
> Q[1,2] := lambdaA;
>
> Q[1,3] := lambdaB;
>
> Q[2,1] := muA;
>
> Q[2,4] := lambdaB;
>
> Q[3,1] := muB;
>
> Q[3,5] := lambdaA;
>
> Q[4,3] := muA;
>
> Q[5,2] := muB;
>
> for i to size do
>     s := 0:
>     for j to size do
>         s := s + Q[i,j]
>     od;
>     Q[i,i] := -s
> od;
>
> Qt := transpose(Q);
>
> for i to size do Qt[size,i] := 1 od;
>
> e[size] := 1;
>
> p := linsolve(Qt,e);
```

4 More Complex Markov Processes

4.1 Introduction

In this note we consider another example Markov process, modelling a more realistic system than those considered in the previous lecture notes. However we will quickly see that even for such moderately sized examples difficulties soon arise when we work directly at the level of the Markov process. Several attempts may be necessary before a correct characterisation of the state space is achieved; generating the state space and calculating the transition rates is time consuming and error prone; interpreting the results and deriving the performance measures is also a complex task. The last two problems are helped by using a high-level modelling paradigm, as we will discuss in the last section of the note.

4.2 Upgrading a PC LAN

4.2.1 Description of the system

The problem we consider is to determine the mean waiting time for data packets at a PC connected to a local area network, operating as a token ring. Such a network uses a transmission medium that supports no more than one transmission at any given time. To resolve conflicts, a token is passed round the network from one node to another in round robin order. A node has control of the medium, i.e. it can transmit, only whilst it holds the token. In a PC LAN every PC corresponds to a node on the network. Other nodes on the network might be peripheral devices such as printers or faxes but for the purposes of this study we make no distinction and assume that all nodes are PCs. There are currently four PCs (or similar devices) connected to the LAN in a small office, but the company has recently recruited two new employees, each of whom will have a PC. Our task is to find out how the delay experienced by data packets at each PC will be affected if another two PCs are added.

Each PC can only store one data packet waiting for transmission at a time, so at each visit of the token there is either one packet waiting or no packet waiting. The average rate at which each PC generates data packets for transmission is known to be λ (see Figure 2). We also know the mean duration, d, of a data packet transmission, and the mean time, m, taken for the token to pass from one PC to the next. It is assumed that if another data packet is generated, whilst the PC is transmitting, this second data packet must wait for the next visit of the token before it can be transmitted. In other words, each PC can transmit at most one data packet per visit of the token.

4.2.2 Modelling the system

The first stage of developing a Markov process to represent the system is to choose the random variables which will be used to characterise the **states** of the system. Firstly, considering the nodes, we can think of each PC as being in one of two states:

- the PC does not currently have a data packet to transmit; or

- the PC has a data packet waiting at the interface for transmission.

We map these two observations to the values 0 and 1 respectively.

For the token we can think of its current state being characterised by its current position. Thus, if there are N PCs in the network the states of the token correspond to the values $\{1, 2, \ldots N\}$.

It follows that when there are N PCs in the network each state can be represented as a $(N+1)$-tuple—one element for the state of each PC and one for the position of the token.

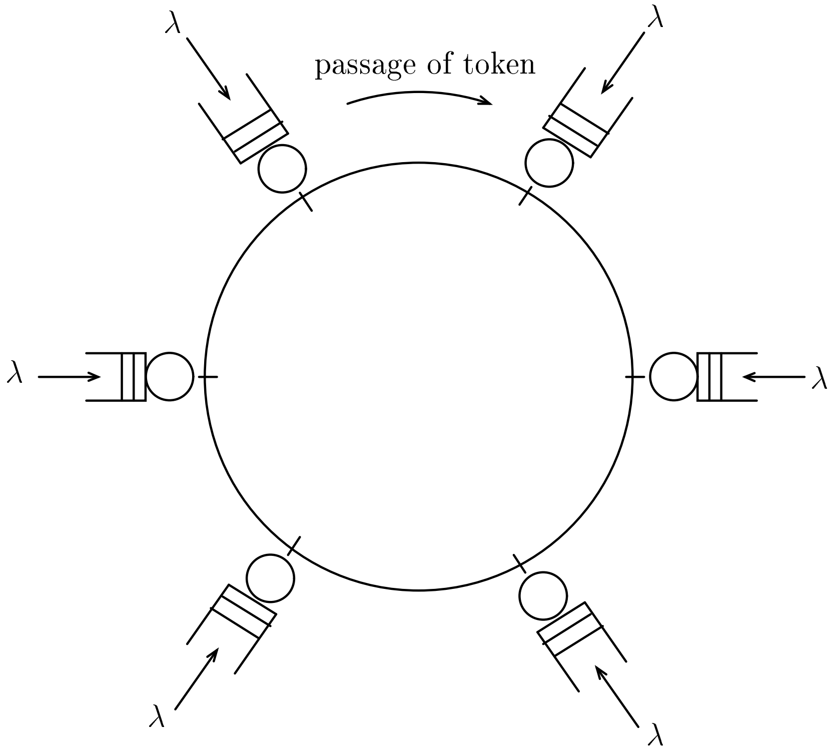


Figure 2: A six-node token ring network

The second stage is to decide on the **events** we wish to represent, and to assign parameters to the corresponding exponential distributions based on the information we have been given about the system. In this case, the events will be:

- the arrival of data packets—we know that at each PC data packets are generated at a rate of λ;
- the movement of the token between PCs—we set the rate of the movement, ω, to be the inverse of the mean time taken $\omega = 1/m$;
- the transmission of data packets—similarly, we set the rate of transmission, μ, to be the inverse of the mean transmission time $\mu = 1/d$.

It is always a good idea to test our reasoning at an early stage. In this case the most sensible approach is to consider a very simple version of the problem, for example, a network with just two PCs. For this system the states would be represented by the triple:

$$(\underbrace{\mathrm{PC}_1}_{\{0,1\}}, \underbrace{\mathrm{PC}_2}_{\{0,1\}}, \underbrace{\mathrm{token}}_{\{1,2\}})$$

where each of the elements has the possible values shown. This would give a model with eight states:

$$\begin{array}{cccc} (0,0,1) & (0,1,1) & (1,0,1) & (1,1,1) \\ (0,0,2) & (0,1,2) & (1,0,2) & (1,1,2) \end{array}$$

Let us consider the state $(1,1,1)$: the token is at PC_1 which has a data packet to transmit. The description of the system in Section 4.2.1 implies that in this state the only possible event is the transmission of PC_1's data packet. The state becomes $(0,1,1)$. If another data packet is now ready for transmission at PC_1, the state will once again become $(1,1,1)$ and a second transmission will be possible without the token visiting PC_2. This is contrary the behaviour of the system which was described.

The only conclusion is that our current version of the model is unsatisfactory and needs to be modified. There are several possibilities:

- One way to avoid the problem outlined above would be to ensure that after transmission of the data packet from PC_1 the token is moved to PC_2: $(1,1,1) \longrightarrow (0,1,2)$; and similarly after a transmission from PC_2. However, this ignores the waiting time while the token is passed between nodes of the network and amalgamates two events into one.
- Other solutions involve introducing more states into the model so that the state when a transmission has just occurred is distinguished from the state when the token has just arrived at a PC. There are two possibilities for doing this: introducing another state for each PC or introducing N more states for the token.

  In the former case, after transmission the PC would enter a kind of limbo state where it could not accept any new data packets into its interface until the token has moved on. This does not reflect the behaviour of the system as described.

  Alternatively, we can notionally move the token past PC_1 when it has completed a data transmission while recording that it still needs to be passed to PC_2. In this way the token is clearly not available for any more data transmission at PC_1 until it has completed a circuit of the ring, but it still has to be passed to PC_2. This is the solution we adopt.

Thus, the states of the token now become $\{1,\ldots,N,1+,\ldots,N+\}$ where $n+$ denotes the state of having completed a data transmission at PC_n. In the two node network, the states are now represented by triples:

$$(\underbrace{\mathrm{PC}_1}_{\{0,1\}}, \underbrace{\mathrm{PC}_2}_{\{0,1\}}, \underbrace{\mathrm{token}}_{\{1,2,1+,2+\}})$$

and the possible states are:

$$\begin{array}{cccccccc} (0,0,1) & (0,1,1) & (1,0,1) & (1,1,1) & (0,0,1+) & (0,1,1+) & (1,0,1+) & (1,1,1+) \\ (0,0,2) & (0,1,2) & (1,0,2) & (1,1,2) & (0,0,2+) & (0,1,2+) & (1,0,2+) & (1,1,2+) \end{array}$$

Using this new scheme, in general, for N PCs the state of the system will be described by an $(N+1)$-tuple:

$$(\underbrace{\mathrm{PC}_1}_{\{0,1\}},\ \ldots,\ \underbrace{\mathrm{PC}_i}_{\{0,1\}},\ \ldots,\ \underbrace{\mathrm{PC}_N}_{\{0,1\}},\ \underbrace{\text{token}}_{\{1,\ldots,N,1+,\ldots,N+\}}\ )$$

The number of states in the model, when there are N PCs, will be

$$2^N \times (2 \times N) = N \times 2^{N+1}.$$

Of course, in many offices there will be 20 or 30 PCs attached to a LAN rather than just 4 or 6. Even for a straightforward model such as the one we have constructed, systems of this size are beyond the capabilities of available tools to solve analytically. The size of the state space, $|S|$, for various values of N are shown in the table below:

N	2	3	4	5	6	8	10	20	30
$\lvert S\rvert$	16	48	128	320	768	4096	20480	4.194304×10^7	6.442450×10^{10}

4.2.3 Solving the model

The models with 4 nodes and 6 nodes were solved using `maple` (see transcript at the end of this note). The transition rates for these models were generated using a Standard ML program and are available from the web pages

`www.inf.ed.ac.uk/teaching/modules/ms/maple`

in `maple` input files `PC-LAN4.trans` and `PC-LAN6.trans` respectively. Similarly, the files `PC-LAN4.table` and `PC-LAN6.table`, provide the assignment of numbers (e.g. $1, 2, \ldots, 128$ and $1, 2, \ldots 768$) to states for each of the models.

The following values were assigned to parameters of the model in both cases (the unit of time is assumed to be milliseconds),

$$\lambda = 0.01 \qquad d = 10 \qquad \mu = 0.1 \qquad m = 1.0 \qquad \omega = 1.0$$

We wish to calculate the average waiting time of data packets at any PC in the network. Since all the PCs are statistically identical, we can arbitrary choose one as representative—we select PC_1. Waiting time, which is the residence time of a data packet at the interface minus the transmission time[1], cannot be derived directly from the steady state distribution since it is neither a state-based nor a rate-based performance measure. However, if we know the average number of data packets at the interface, and the average throughput, we can apply Little's law to calculate residence time and therefore the waiting time.

There will be a data packet waiting in the interface of PC_1 whenever the first entry in the state tuple is "1". Therefore the average number of data packets, $\mathcal{N}$, is the total probability of being in one of these states, i.e. the sum of the values in the elements of the steady state vector corresponding to these states. Similarly, a data transmission from

[1]Note that in the model a data packet remains at the PC until its transmission is complete, there is no state of being in transit. Thus its residence time includes the transmission time.

PC_1 can occur whenever there is a data packet waiting for transmission and the token is at PC_1. Therefore the average throughput X will be the rate at which transmission occurs, μ, multiplied by the total probability of being in one of these states. Finally let $\mathcal{W}$ denote the average waiting time:

$$\mathcal{W} = \frac{\mathcal{N}}{X} - 1/\mu$$

The `maple` expressions for calculating $\mathcal{N}$ and X for each of the models can be found in the files `NumberPC-LAN4`, `ThroughputPC-LAN4`, `NumberPC-LAN6` and `ThroughputPC-LAN6`, which are in the same directory as the files listed above. Based on these expressions and the calculated steady state distributions we find the following values: for 4 PCs

$$\text{average waiting time, } \mathcal{W} = \frac{0.1333}{0.008666} - \frac{1}{0.1} = 15.3862 - 10 = 5.3862$$

for 6 PCs

$$\text{average waiting time, } \mathcal{W} = \frac{0.1719}{0.00828} - \frac{1}{0.1} = 20.7678 - 10 = 10.7678$$

Thus the average waiting time for data packets will almost double when two more PCs are added to the network.

4.2.4 Summary

This example was intended to give you an impression of how even relatively small systems, such as a PC LAN connecting just four machines, can very quickly give rise to models with hundreds of states. You will get a good idea of the complexity of these, still relatively small, models if you look at the Maple files containing all the transitions (see above). Even with just three nodes in the network the model generates 48 states which is large enough to be very difficult to accurately generate by hand. Similarly, trying to extract even straightforward state-based performance measures from models of this size is time-consuming and error-prone.

Exercise: Try to list all the states and draw the state transition diagram for the three node model.

The PC LAN model also illustrates that our first attempt is not necessarily successful. Like any complex problem solving task, modelling is often an iterative process: only when we try to use the model do we become aware of a flaw in the abstraction used. In this case the initial representation of the states of the system was too abstract—we needed more information about the state of the token in order to capture the full behaviour of the system.

4.3 High-level modelling paradigms

Although Markov processes are a convenient modelling technique from some points of view, as noted in the introduction to this note, they can be difficult to work with once the number of states gets into hundreds rather than tens. Since most realistic models of systems involves hundreds of thousands, if not millions of states, constructing the complete state space and finding all the possible transitions becomes a serious problem. Obviously computer-assistance is needed. Moreover, for that assistance to be really worthwhile we need a more abstract description of system behaviour than direct manipulation of the state space.

High-level modelling paradigms are formalisms which provide the modeller with a means of abstracting system behaviour. Now the modeller describes the system in terms of the primitives of the paradigm rather than expressing the model directly in terms of the states and transitions of the Markov process. The formalism includes a procedure for generating the corresponding Markov process from a high-level model.

In this course we will consider three such high-level modelling paradigms, *stochastic Petri nets*, *stochastic process algebras* and *queues* and *queueing networks*. The primitives of stochastic Petri nets and stochastic process algebras are still quite low level, with transitions explicitly represented. In contrast, in queueing networks sophisticated primitives, which encode a lot of information, are used. The benefit of the simplicity of the stochastic Petri net/process algebra approach is much greater expressiveness—these models can capture many situations which cannot be represented using queueing networks.

Performance modelling tools based on high-level modelling paradigms generally provide several important features:

- support for model construction—in some cases this will include a graphical interface, with menus and icons;
- automatic generation of the corresponding Markov process—some tools will even recognise special cases of models which can be efficiently solved during this stage; others will rely on the modeller knowing when special cases apply;
- solution of the Markov process—either using the same basic approach that we have been applying or more sophisticated solutions based on particular model structures;
- expression of performance measures in terms of the high-level model and automatic derivation of the corresponding measures at the Markov process level.

4.4 Maple transcript for model PC-LAN6

```
with(linalg);                                   # load linear algebra package

lambda := 0.01;                                 # set values of parameters

mu := 0.1;

omega := 1.0;

Q := array(sparse,1..768,1..768);               # declare the generator matrix

b := array(sparse,1..768);                      # declare the solution vector

read `/home/jeh/CS4/MandS/maple/PC-LAN6.trans`; # read in the entries for Q

Qt := transpose(Q):                             # transpose the generator matrix

for i to 768 do Qt[768,i] := 1.0 od:            # replace the last equation by
                                                # the normalisation constant in
b[768] := 1.0;                                  # both Qt and b

p := linsolve(Qt,b);                            # solve to find the unknowns,
                                                # the steady state distribution

read `/home/jeh/CS4/MandS/maple/NumberPC-LAN6`;
                                                # derive the mean number of
                                                # data packets at PC1

read `/home/jeh/CS4/MandS/maple/ThroughputPC-LAN6`;
                                                # derive the mean throughput
                                                # of data packets at PC1

w = N/X - 1/mu;                                 # calculate waiting time as the
                                                # residence time (N/X) minus the
                                                # transmission time (1/mu)
```

Please note that you would need to adjust the pathnames to match your own location of the maple input files.

5 Stochastic Petri Nets

In this lecture note we consider an important class of high level performance modelling paradigms—stochastic extensions of *Petri nets*. These are Petri net formalisms into which random variables have been added to represent the duration of activities, or the delay until events. The basic extension, *Stochastic Petri Nets* (SPN), present a very straightforward mapping between events in the SPN model and events in the underlying Markov process. As in the Markov process, a delay, represented by a random variable, is associated with every event in the model. This straightforward mapping has the advantage that generating the Markov process from any SPN model is simple and easy to implement. The disadvantage is that models developed in this way tend to result in Markov processes which have a large number of states.

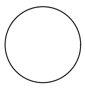
We will go on to consider *Generalised Stochastic Petri Nets* (GSPN). The generalisation involves adding some simple additional features to the set of modelling primitives provided for the modeller. As a result the mapping between events in the model and events in the underlying Markov process is more sophisticated. However the resulting state space is usually more compact, and often a closer representation of the behaviour of the system can be achieved.

Although Petri nets have been used for qualitative modelling of computer and communication systems since the 1960s, their use as a performance modelling paradigm started about twenty years ago. In particular they were found to be especially useful for modelling distributed and parallel systems. Such systems were difficult to model with queueing networks, which were the prevailing performance modelling paradigm at the time.

5.1 Petri Nets

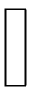
Petri nets provide a graphical notation for the formal description of the dynamic behaviour of systems. They are particularly well suited to systems which exhibit concurrency, synchronisation, mutual exclusion and conflict.

The primitives of the notation are the following:



PLACES

Places are used to represent conditions or local system states, e.g. a place may relate to one phase in the behaviour of a particular component.



TRANSITIONS

Transitions are used to describe events that occur in the system; these will usually result in a modification to the system state. The occurrence of the event in the system is represented by the *firing* of the corresponding transition in the Petri net.

TOKENS

Tokens are identity-less markers that reside in places. The presence of a token in a place indicates that the corresponding condition or local state holds.

ARCS Arcs specify the relationships between local states or conditions (places) and events (transitions). An arc *from* a place *to* a transition is termed an *input arc.* This indicates the local state in which the event can occur. An arc *to* a place *from* a transition is termed an *output arc.* This indicates the local transformations which will be induced by the event.

Tokens move between places according to the firing rules imposed by the transitions. A transition can *fire* when each of the places connected to it has at least one token; when it fires, the transition removes a token from each of these places and deposits a token in each of the places it is connected to. This is called the *firing rule.*

Sometimes a transition will require an input place to contain two or more tokens before it can fire. In this case, rather than draw more than one arc between the place and the transition, we denote the *multiplicity* of the arc by a small number written next to the arc. Similarly for output arcs.

The state of the system combines information about all the local states. Since each local state is represented by the number of tokens present in a particular place, the state of the system is represented by a tuple, with one entry for each place, and the value of the entries denoting the number of tokens in that place. This is termed the *marking* of the net.

A Petri net consisting of places and transitions linked by arcs is incomplete if it does not also have tokens in some places. This initial placing of tokens is called the *initial marking*—this represents the starting state of the system.

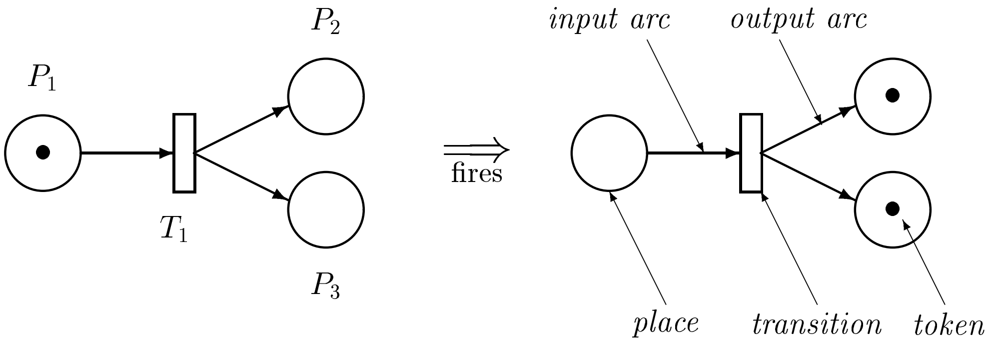


Figure 3: A Simple Petri Net Firing

Starting from an initial marking and following the firing rule we can progress through the states of the model. This is sometimes called *playing the token game.* Continuing in this way, recording all the states we see and stopping only when we can reach no states that we have not already seen, we obtain all the possible states of the model. This is called the *reachability set*; it is the set of all possible markings that a net may exhibit, starting from the initial marking and following the firing rules. Different initial markings might lead to different reachability sets as we will see in the example below. This is why the initial marking is an important part of the model definition.

If, when playing the token game, as well as all the states we come across, we record the transitions between those states, we obtain the *reachability graph.* This is a graph in which the nodes are the reachable markings and the arcs between nodes represent the possible transition firings which may move the model from one marking to the other.

Example: For the simple Petri net shown in Figure 3 the initial marking is $(1, 0, 0)$ and the final marking is $(0, 1, 1)$. These are the only possible markings. If the initial marking is changed to $(3, 2, 1)$ the set of reachable markings is: $\{(3, 2, 1),\ (2, 3, 2),\ (1, 4, 3),\ (0, 5, 4)\}$.
Exercise: Draw the reachability graph for this model with the second initial marking.

5.2 Stochastic Petri Nets

Recall that if we wish to extract timing information from a model we must represent timing information about the system in the model when it is constructed. In the case of Petri nets there has been a variety of suggestions of how to introduce timing information into Petri net notation.

If we consider the reachability graph of a Petri net model it resembles the state transition diagram of a Markov process. *Stochastic Petri Nets* (SPN) formalise this intuitive correspondence. Given a Petri net model (complete with initial marking):

- we associate a state in the Markov process with every marking in the reachability graph of the Petri net;
- we associate an event, or transition, in the Markov process with each firing of a transition in the Petri net which causes the corresponding change of marking.

Since an exponentially distributed delay is associated with the delay until each event in a Markov process, and transitions in the Petri net correspond to events, in an SPN model an exponentially distributed delay is associated with each transition in the net structure. Thus each transition in an SPN has a *firing rate* which is the parameter of the corresponding exponential distribution, and transitions are sometimes termed *timed transitions*.

How this works in practice is best illustrated by an example.

Example: Consider again the simple multi-processor system which was described in Section 3.3.1. We make a slight modification to the system: we now explicitly represent the processor requesting and gaining access to the common memory. In the previous model we used a higher level of abstraction in which this action was ignored. A processor executes locally for some time (mean duration $1/\lambda$), and then requests access to common memory (gaining access has mean duration $1/r$). Once it has gained access, the duration of common memory access is assumed to be $1/\mu$ on average. We can model the behaviour of a single processor interacting with the common memory using an SPN, as shown in Figure 4. In this SPN

- place P_1 represents the local state of the processor when it is executing privately;
- place P_2 represents the local state of the processor when it is ready to commence access to the common memory;
- place P_3 represents the state when the process is using the common memory;
- place P_4 represents the local state of the common memory when it is not in use;
- transition T_1 represents the *action* of the processor *executing privately*; the rate of this transition is λ;

- transition T_2 represents the processor gaining access to the common memory; the rate of this transition is r;
- transition T_3 represents the processor accessing the common memory; the rate of this transition is μ.

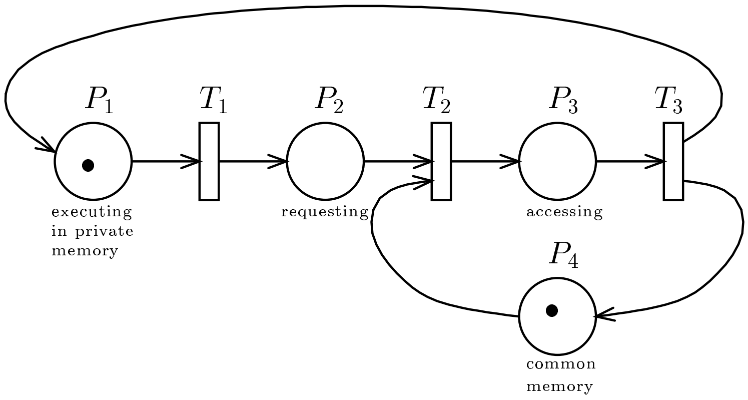
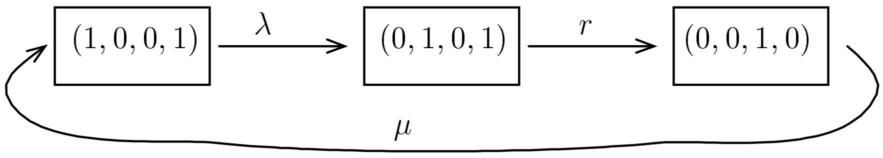


Figure 4: Stochastic Petri net representing a single processor in a shared memory system

The reachability set of this model is $\{(1,0,0,1),(0,1,0,1),(0,0,1,0)\}$, and the reachability graph is



This is also the state transition diagram of the underlying Markov process and thus we obtain the generator matrix:

$$\mathbf{Q} = \begin{pmatrix} -\lambda & \lambda & 0 \\ 0 & -r & r \\ \mu & 0 & -\mu \end{pmatrix}$$

Solving this we obtain the steady state probability

$$\boldsymbol{\pi} = \left(\frac{\mu r}{\lambda r + \mu\lambda + \mu r}, \; \frac{\mu\lambda}{\lambda r + \mu\lambda + \mu r}, \; \frac{\lambda r}{\lambda r + \mu\lambda + \mu r} \right)$$

If we consider the two-processor system, as we did in Section 3.3.2, the SPN model is as shown in Figure 5. Now we see why the action of gaining access needs to be explicitly represented in the SPN model. Without it we could not enforce the necessary mutual exclusion between the processors. If we assume the upper subnet represents processor A and the lower subnet represents processor B, the interpretations for transitions should be clear: T_1 and T_4 correspond to T_1 in the single processor model, T_2 and T_5 correspond to T_2, and T_3 and T_6 correspond to T_3. Similarly for the places. The rates of the transitions

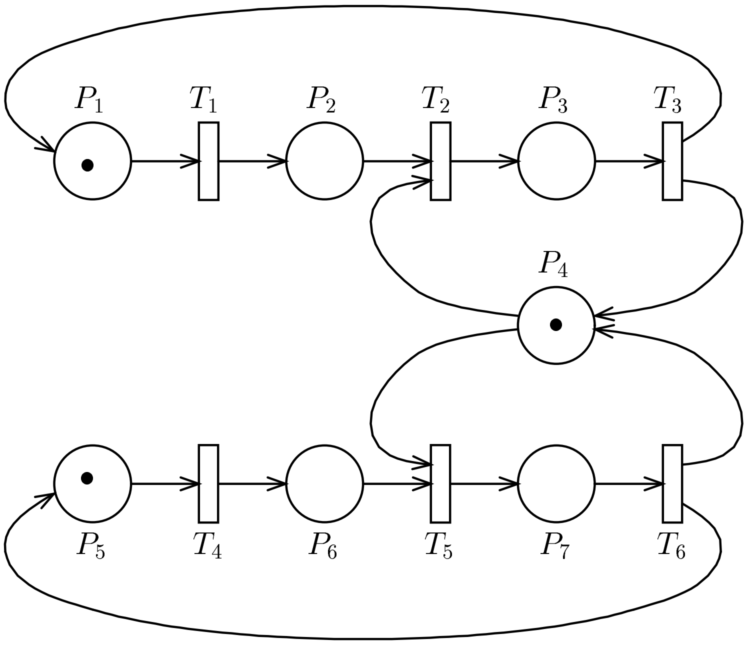


Figure 5: Stochastic Petri net representing a two-processor shared memory system

will be λ_A, r_A and μ_A for T_1, T_2 and T_3 respectively, and λ_B, r_B and μ_B for the transitions T_4, T_5 and T_6.

The reachability set is now[1]

$$\left\{ \begin{array}{llll} (1,0,0,1,1,0,0), & (1,0,0,1,0,1,0), & (1,0,0,0,0,0,1), & (0,1,0,1,1,0,0), \\ (0,1,0,1,0,1,0), & (0,1,0,0,0,0,1), & (0,0,1,0,1,0,0), & (0,0,1,0,0,1,0) \end{array} \right\}$$

Exercise: Draw the reachability graph (state transition diagram) for this model, and construct the generator matrix.

5.3 Generalised Stochastic Petri Nets

Although SPN provide a clear and intuitive formalism for generating Markov processes they do have the disadvantage that the models constructed in this way can soon become exceedingly large. One of the reasons for this is that actions which would not be explicitly represented if we were working directly at the Markov process level have to be represented by transitions. This has the effect of increasing the state space since we now have to consider the state of waiting for these actions to occur, whereas we would prefer to abstract away from these actions.

An example of this is the action of gaining access to the common memory in two-processor model of the previous section. When we modelled this system as a Markov process in Section 4.2.2 we did not represent this action because we constructed the state space directly and so could prohibit the state of both processors accessing the common

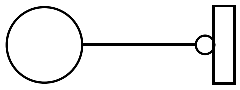
[1]Note that we now have eight distinct states as opposed to the five states we had in the Markov process when we modelled the system directly (Lecture Note 3).

memory at once. One of the disadvantages of using a high level modelling paradigm, such as SPN, to generate the state space automatically, is that we sometimes have to introduce mechanisms into the model to make sure that only "genuine" states are generated.

Generalised Stochastic Petri Nets (GSPN) represent an extension of the SPN formalism, which is designed to address this problem. Two new primitives are added to the notation, all others remaining the same and keeping the same interpretation. These new primitives are *immediate transitions* and *inhibitor arcs*.

IMMEDIATE TRANSITIONS

Immediate transitions are represented by a single bar, instead of the rectangle used to represent timed transitions. Immediate transitions are used to describe events which are assumed to take no time. When they are enabled in a model they fire immediately, taking precedence over any enabled timed transitions. In other respects the firing rule is exactly the same as for timed transitions. If two or more immediate transitions can be enabled at the same time, the probability that each of them is the one to fire must be declared in the model.



INHIBITOR ARCS

An inhibitor arc is used to indicate when a local state or condition *disables* a transition, rather than *enables* it. An inhibitor arc *from* a place *to* a transition, indicates that the transition cannot fire if there is a token in the place; it can fire when there is no token in the place if the places connected to its input arcs do contain tokens. In other words, the usual firing rule still applies; the inhibitor arcs impose an additional constraint to that rule. Inhibitor arcs may have multiplicities in the same way as ordinary arcs.

The types of events represented by immediate actions usually fall into two categories: *control actions* and *logical actions*. Control actions are ones which are necessary to ensure the correct behaviour of the model but which take negligible time to be executed, which means that they are unlikely to affect the performance of the system. For example, gaining access to the common memory in the two-processor model considered in the previous section can be regarded as a control action. The GSPN representing the system in this way is shown in Figure 6. In these cases immediate actions provide an additional tool for abstraction within the model.

Logical actions arise when the system makes a choice between two or more alternatives. In an SPN these actions must be represented as actions which take time. However it is more natural to think of them occurring instantaneously. In the GSPN a choice between two alternatives will be represented by two immediate transitions which have exactly the same input places (firing conditions), and the probabilities assigned to them will reflect the relative probabilities of the two choices.

5.3.1 Generating the Markov process from a GSPN model

One of the advantages of SPN models was the straightforward correspondence between the reachability graph of the SPN and the state transition diagram of the Markov process

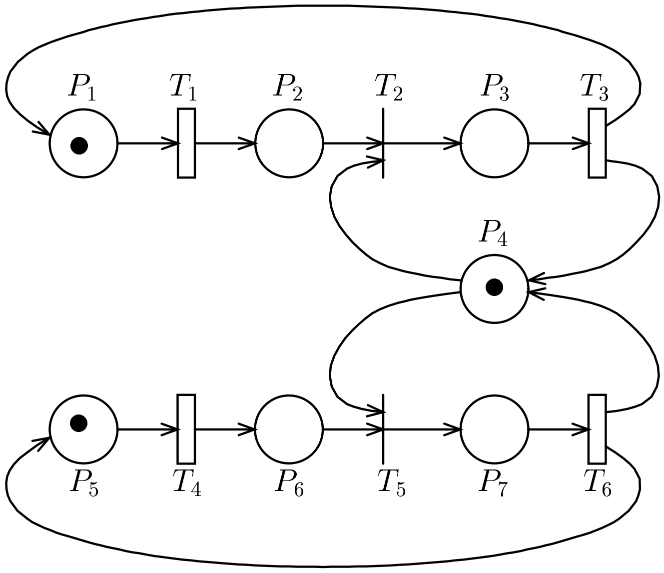


Figure 6: GSPN representing a two-processor shared memory system

it generated. The inclusion of inhibitor arcs within the GSPN notation does not affect this correspondence. The effect of the inhibitor arc will be to eliminate some arcs which might have been possible in the reachability graph. Thus the relationship between the Markov process and the reachability graph is unaffected. In contrast, the introduction of immediate actions has serious implications for the Markov process.

If there is an immediate transition in a GSPN model, in the reachability graph there will be arcs between nodes which represent immediate transitions between markings in the GSPN. Such arcs are labelled with rate ∞. In other words, in the reachability graph some transitions between states take place instantaneously, without an associated delay. Such instantaneous transitions are not possible in a Markov process. Therefore these transitions, and the states that give rise to them, must be eliminated from the reachability graph before the Markov process is generated.

Formally the transitions of a GSPN can be partitioned into two subsets—timed transitions and immediate transitions. Since immediate transitions always take precedence over timed transitions, we can also partition the marking of a GSPN into those which only enable timed transitions, and those which enable at least one immediate transition. The latter are called *vanishing markings*, since the model will always move on to another marking instantaneously. In contrast, the markings which enable only timed transitions are called *tangible markings*. It is the vanishing markings which must be eliminated from the reachability graph before the Markov process is generated. In the simple examples we will consider by hand it will be clear how this is done; for more complex GSPN models the elimination will be carried out automatically by software, as explained in Lecture Note 7.

Exercise: Draw the reachability graph of the GSPN model of the two-processor system and identify the vanishing markings and the tangible markings.

Jane Hillston ⟨jeh@inf.ed.ac.uk⟩. October 7, 2005.

6 More about GSPN Models

In this note we will consider two simple systems modelled by GSPN and in the course of doing so examine more closely the dynamics of these models with respect to timed and immediate transitions. At the end of the note we will summarise some of the key features of the `SPNP` modelling tool and describe how it may be used to generate and solve GSPN models.

6.1 System Dynamics

Recall that in an untimed Petri net the firing of a transition corresponded to an event in the system. In GSPN we recognised that these events might arise in two different ways:

- it may be induced by the completion of some activity; these activities can be further divided into two categories:
  - those which take significant time to complete.
  - those whose duration is negligible compared with other actions in the system, often called *control actions*.
- it may result from the verification of some logical condition of the system.

Timed transitions represent actions which take some time to be completed. Immediate transitions represent logical events and control actions.

For timed transitions, when they become enabled, i.e. all the input places of the transition become marked with the necessary number of tokens, we imagine the corresponding activity starting in the system. This activity will continue until it is completed or interrupted. We assume that each transition has a timer or local clock associated with it: when the transition is enabled the timer is set to a value. Since we are using a random variable to represent activity durations, this value is *sampled*[1] from the exponential distribution with the appropriate rate parameter. While the transition is enabled we assume that the timer decrements at a constant speed. When the timer reaches zero the transition fires.

For immediate transitions, the firing of the transition takes no time and is assumed to occur instantaneously. We also assume that enabled immediate transitions always have priority over any enabled timed transitions, so we fire all possible immediate transitions first. If only one immediate transition is enabled, it fires, and the new marking is produced. If several immediate transitions are enabled, some mechanism is necessary to identify which transition will fire. In fact the choice of transition to be fired is only important in cases where there is conflict between the enabled transitions, i.e. the firing of one will disable another. If the enabled immediate transitions are concurrent, they can be fired in any order. When enabled immediate transitions are in conflict GSPNs associate *weights*, or relative probabilities, with each transition. We can think of a distribution being sampled to determine which transition actually fires in each case.

[1] *Sampling* means randomly choosing a value, in such a way that if you kept on doing this over time and kept track of the values they would satisfy the probability distribution function. Sampling will be treated in more detail when we study simulation. Note that for GSPN models solved as a Markov process this is just a way of thinking about the dynamics of the system, we do not actually sample the distributions.

Bearing this view of the system dynamics in mind we will consider again the two-processor shared memory system from lecture note 5. For convenience it is shown again in Figure 7.

6.1.1 Example: The multiprocessor system revisited

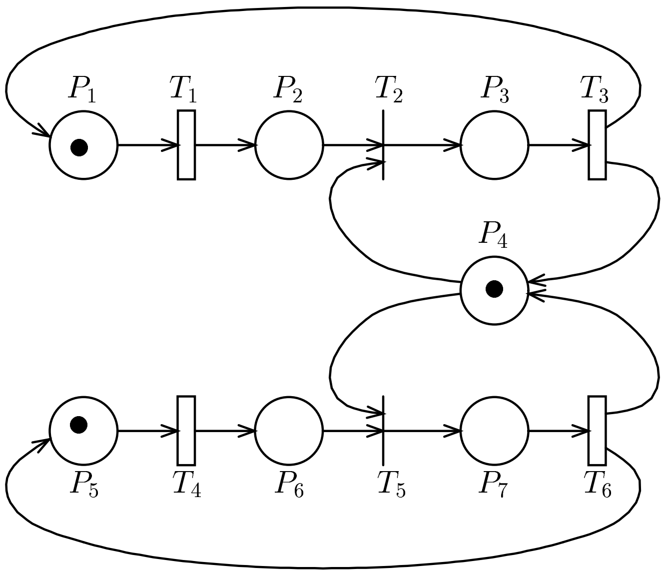


Figure 7: GSPN representing the two-processor shared memory system

In the initial marking of the system we see that both transitions T_1 and T_4 are enabled. Thus we think of them sampling their respective exponential distributions, with parameters λ_A and λ_B, and obtaining values V_A and V_B say. Which transition fires first will be determined by whose associated timer reaches the value zero first—this is called the *race policy*. If $V_A < V_B$, transition T_1 will be the one to fire and the marking changes as $(1,0,0,1,1,0,0) \to (0,1,0,1,1,0,0)$. If $V_B < V_A$, transition T_4 fires and the new marking is $(1,0,0,1,0,1,0)$. The case of $V_A = V_B$ can only occur with probability zero[2] so we do not need to worry about it. Of course each time the distributions are sampled different values may be obtained, and so which transition fires first will vary.

In either case we need to consider what happens to the remaining time on the timer associated with the transition which did not fire. There seem to be two possibilities: either it is forgotten or it is remembered. For example, in the two-processor system if $V_A < V_B$ and transition T_1 fires, the time remaining on the timer associated with T_4 will be $V_B - V_A$. Since T_4 remains enabled after T_1 has fired it would seem unreasonable for time the system has already spent on the corresponding activity to be forgotten. In contrast, if T_1 and T_4 had been in conflict, so that the firing of T_1 removed the enabling tokens for T_4 it might seem reasonable to forget the time that has already been spent on the corresponding activity, and imagine the system starting that activity again from the

[2]The probability that a sample extracted from an exponential distribution takes any specific value v is always zero, so if we know that the value sampled by the first timer is V_A the probability that the second timer will sample the same value is zero.

beginning whenever T_4 next becomes enabled. Note that in the case of conflict between timed transitions which transition will fire is always resolved by the race policy.

Fortunately, the memoryless characteristics of the exponential distribution mean that we do not need to differentiate between the situations when a timed transition should forget its previous work and those when it should remember. Indeed, whether the timer is reset or not, the probability distribution of the time remaining until the firing of a transition will always be the same, and distributed according to the exponential distribution when the transition was first enabled. Thus we can model either of these situations in the same way in a GSPN.

Despite initial impressions from inspecting the net, the two immediate transitions in the GSPN of the two-processor system can never both be enabled at the same time. Thus at most one immediate transition is enabled in any marking and this transition will fire instantaneously leading to the following marking. For example, if T_1 fires as described above, leading to the marking $(0, 1, 0, 1, 1, 0, 0)$, transition T_2 is now enabled and fires at once, leading to the marking $(0, 0, 1, 0, 1, 0, 0)$.

6.1.2 Example: The reader-writer system

The GSPN in Figure 8 is a representation of what is sometimes referred to as the *reader-writer system*. The model represents a system in which there is a set of processes who share access to a common database. On any particular access a process may wish to perform a read or a write. Any number of readers may access the database concurrently; in contrast, a writer requires exclusive access to the database. In between accesses to the database each process will undertake some processing independently.

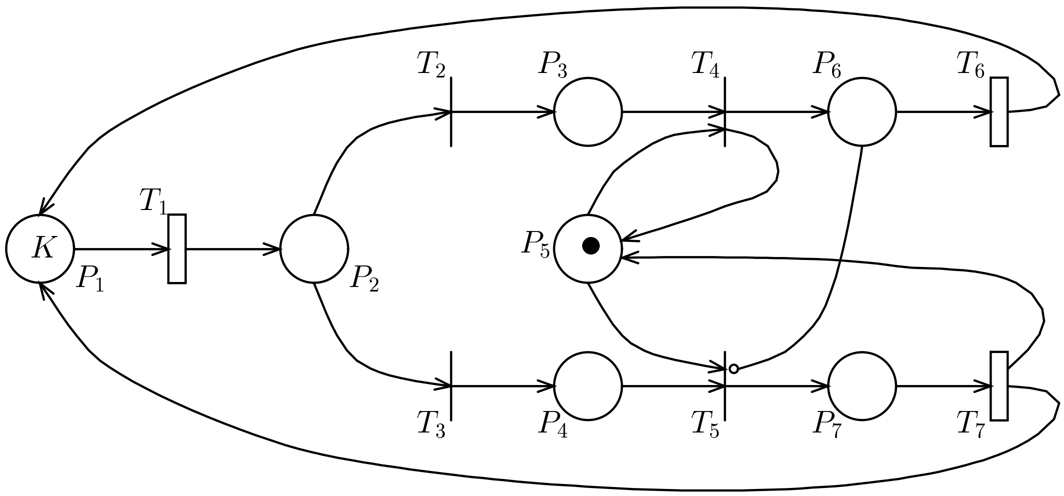


Figure 8: GSPN representing the simple reader-writer system

Timed transitions are used to represent the reading and writing actions as well as the local activities performed by the processes between two subsequent accesses to the database. Immediate transitions are used to implement the decision of which action a process wants to perform on the database (T_2 and T_3) and to implement the scheduling between reading and writing activities (T_4 and T_5). The interpretations of the places and transitions in the model shown in Figure 8 are given below.

- P_1 represents a process undertaking local processing.
- P_2 represents a process ready to commence access to the database.
- P_3 and P_4 model processes ready to read and ready to write respectively.
- P_5 enforces unique access to the database for writing; it can be thought of as representing the condition *no write access is in progress*.
- P_6 and P_7 represent processes currently reading or currently writing respectively.
- T_1 models the independent processing, not related to the database access; each process undertakes local processing for a mean duration of $1/\lambda$ milliseconds.
- T_2 and T_3 model the "decision" of the process to make a read access or a write access; the switching probabilities are chosen to reflect the relative frequencies of these types of access.
- T_4 represents a process gaining access to the database to start a read access.
- T_5 represents a process gaining access to the database to start a write access; note that the transition is inhibited whenever a read is already in progress (P_6).
- T_6 models the database read activity; the average read access lasts for $1/r$ milliseconds.
- T_7 models the database write activity; the average write access lasts for $1/w$ milliseconds.

This system is commonly used in texts about modelling because it is small and easily understood but yet it exemplifies three interesting features which are often important in models: *concurrency of events* (two or more processes may be concurrently accessing the database for reading), *mutual exclusion* (only one process at a time may access the database for writing) and *choice* (an access can either be a read or a write). This version of the model is based on the one presented in the book by Ajmone Marsan *et al.*[3]; in other books variations on this model appear.

Note that the initial marking of the GSPN has a single token in place P_5, representing that the condition is currently satisfied, and a parameter K in place P_1. The *parametric marking* represents the situation that there is some number of processes, K, within the system, but the value of K has not yet been decided. A value must be assigned to K before the model can be solved because, as we have already seen, the number of tokens in the initial marking can drastically change the reachability set of a model.

Let us first consider the immediate transitions of the model. Whenever there is a token in place P_2 the two immediate transitions T_2 and T_3 are both enabled. These transitions are in conflict because the firing of one will disable the other. Thus weights, or relative probabilities, must be assigned to the transitions to resolve the conflict. If we know that 60% of database accesses are read accesses and 40% are write accesses, we associate weights 0.6 and 0.4 with T_2 and T_4 respectively.

[3] *Modelling with Generalized Stochastic Petri Nets*, M. Ajmone Marsan, G. Balbo, G. Conte, S. Donatelli and G. Franceschinis, John Wiley, Series in Parallel Computing, 1995

If transition T_2 fires, and P_5 contains a token, transition T_4 is now enabled and fires immediately—no other immediate transitions are enabled and the immediate transition takes priority over any timed transitions which might be enabled, such as T_1. Thus two immediate transitions may fire one after another without any time progressing in our model. If T_3 fires a token appears in place P_4, but transition T_5 will only be enabled if there is no token in place P_6—this is the effect of the inhibitor arc—and there is a token in place P_5.

Let us now consider the timed transitions of the model. If there are K tokens in place P_1 it means that there are K processes currently involved in local processing, each progressing at a rate λ. Thus the total rate at which processes complete local processing is $K \times \lambda$. In the model we represent this by saying that the rate of the transition T_1 is dependent on the marking of transition P_1: the rate of the transition is λ multiplied by the number of tokens in P_1 (by the superposition principle). Such transitions are said to have *marking dependent firing rate.*

Similarly when more than one process is engaged in a read access, the rate at which a single read access is completed will depend on the number of processes currently reading—the marking of place P_6—and the rate at which reading occurs—the transition rate of transitions T_6 (r).

In general, whenever a transition's enabling condition is satisfied more than once, e.g. K tokens in place P_1 when only one is required to enable transition T_1, there are two possibilities with respect to the firing rate of the transition. We can imagine the transition progressing each set of enabling input tokens *serially.* In this case we only consider the transition to be enabled or not, and when it is enabled it always works at the same rate. This is called *single server semantics* and it is the default case for GSPN models. Alternatively we can think of the transition progressing each complete set of enabling input tokens *concurrently.* In this case the rate at which the transition works will depend upon its *enabling degree.* This is called *infinite server semantics* and in GSPN models it is achieved by setting a marking dependent firing rate.

In the reader-writer model the enabling degree of the transition T_1 in the initial marking is K. Similarly the enabling degree of transitions T_6 can have any value between 1 and K. In contrast, enabling degree of T_5 is at most 1: even though the place P_4 may have up to K tokens, the enabling degree will not be greater than 1 because there is only one token in place P_5.

6.2 The SPNP package

The Division (of Informatics) has several Petri net solution packages but the one which we use on this course is `SPNP` (Stochastic Petri Net Package). The package is composed of several C files. The GSPN to be studied must be described in a language called CSPL (C-based Stochastic Petri Net Language). This language consists of various predefined functions which can be used to specify the structure of the GSPN and the desired outputs of the model. The following steps can be identified in the performance analysis of a system using a GSPN model:

1. the model must be constructed to reflect the behaviour of the system;

2. the performance measures of interest must be defined in terms of GSPN markings and of transition firings;
3. the reachability set and the reachability graph must then be generated, to obtain the Markov process associated with the GSPN;
4. the Markov process must be solved;
5. the performance measures must be computed from the Markov process solution.

Using SPNP the modeller is responsible for the first two steps; the model and the performance measures are specified in the CSPL file. Steps 3-5 are carried out automatically by the SPNP package. The CSPL file (*filename.c*) is compiled, linked to the other files of the package, and run by typing the command

`make -f /home/jeh/SPNP/obj/Makerun SPN=`*filename*
or `make -f /home/jeh/SPNP/obj/Makerun SPN=`*pathname/filename*

The intermediate files generated by the package and the final results will be placed in the same directory as the CSPL file. Files have different extensions, according to the kind of information that they contain. If your CSPL file is named `model.c` then the following files may be generated. Some output files are optional and will only be produced if a flag in the source file is set to the appropriate value.

`model.o` obtained when compiling `model.c`

`model.out` contains some default information and the requested output (according to what is specified in `model.c` using the provided functions)

`model.spn` executable file obtained by linking the package object files together with `model.o`

`model.rg` contains the reachability graph information: composition of each marking and the transition firings between them.
(`IOP_PR_RGRAPH = VAL_YES`)

`model.mc` contains the (numerical) Markov process corresponding to the GSPN.
(`IOP_PR_MC = VAL_YES`)

`model.prb` contains the (numerical) results of the analysis of the underlying Markov process: the steady state probability for each tangible marking.
(`IOP_PR_PROB = VAL_YES`)

`model.log` contains the output messages produced by the package during model solution.

The syntax and semantics of CSPL are based on the C language: a correct CSPL file will be a correct C file too. What distinguishes CSPL from C is the set of predefined functions available for the definition of GSPN entities. Every CSPL file must specify the following functions: `parameters, net, assert, ac_init, ac_reach` and `ac_final`. We briefly describe the significance of each function below. You should look at the provided example files for more details (`/home/cs4/www/modsim/spnp-examples`).

Function `void parameters(void)` This function is used to set the various flags which control options in the package. There are two option-setting functions `iopt` or `fopt` according to whether the option being set has integer or floating point value.[4] In addition there are two input functions `input()` and `finput()` which accept integer and floating point values respectively.

Function `void net(void)` This defines the structure and initial marking of the GSPN, together with firing rates (for timed transitions) and firing probabilities (for immediate transitions). The most important predefined functions are:

`place` **and** `trans` are used to declare the names of the places and transitions used in the GSPN respectively.

`init` gives the initial number of tokens in the named place. By default, places are assumed to be empty.

`rateval` **and** `ratedep` define the transition rate for timed transitions with single-server and infinite-server semantics respectively. When a dependent rate is assumed the place on which the rate depends must be given.

`probval` **and** `priority` define the probability and priority associated with an immediate transition. SPNP assumes all timed transitions have priority level 0 and will not allow immediate transitions to be given priority levels less than 1; it will however allow higher levels as a way of enforcing an ordering between conflicting immediate transitions.

`iarc, oarc, harc` these specify input, output and inhibitor arcs respectively. In each case the transition and the place involved are named, in that order. All these arcs are assumed to have multiplicity one.

`miarc, moarc, mharc` as above, but for arcs with multiplicity greater than one. In each case the transition, place and multiplicity must be specified in that order.

Function `int assert(void)` It is often desirable to carry out some form of model validation to ensure that the model behaves correctly with respect to the system.[5] SPNP supports this in the CSPL file by allowing the modeller to make assertions about the model which are checked when the state space is generated. For example, in the reader-writer system we know that at most one process should have write access to the database at any time, i.e. the marking of place P_7 is less than or equal to 1. Similarly, we know that it should not be possible for transitions T_6 and T_7 to both be enabled.

```
assert(){
   if (mark("p7") > 1 || enabled("t6") && enabled("t7"))
     return(RES_ERROR);
   return(RES_NOERR);
}
```

[4]A full list of options and values is in the SPNP manual but you do not need to know these and can probably safely copy the parameter section of one of the example files into any CSPL program you write.

[5]We will learn more about model validation techniques later in the course.

If we make these assertions in the CSPL file (as shown above), and they are violated when the reachability set is generated, SPNP will stop the execution and report the error.

Functions `void ac_init(void)`, `void ac_reach(void)` **and** `void ac_final(void)` These are called before reachability graph construction, after reachability graph construction and for solution of the Markov process, respectively. The function `ac_final` is designed to give the user flexibility with respect to the output produced by the model solution. Much of this flexibility is beyond the scope of this course. In addition this function is used to control the performance measures calculated in the model (see lecture note 7).

6.3 A GSPN model of the PC LAN with 4 nodes (8 nodes)

The model below is a GSPN representation of the PC LAN with four nodes considered in lecture note 4. Note that the inhibitor arc from place P_{i2} to transition T_{bi} (for $i = 1, \ldots, 4$) ensures that a token does not bypass a PC in which a data packet is waiting for transmission. In the CSPL file `/home/cs4/www/modsim/spnp-examples/PC-LAN.c` you will find the SPNP model of the LAN with eight PC nodes.

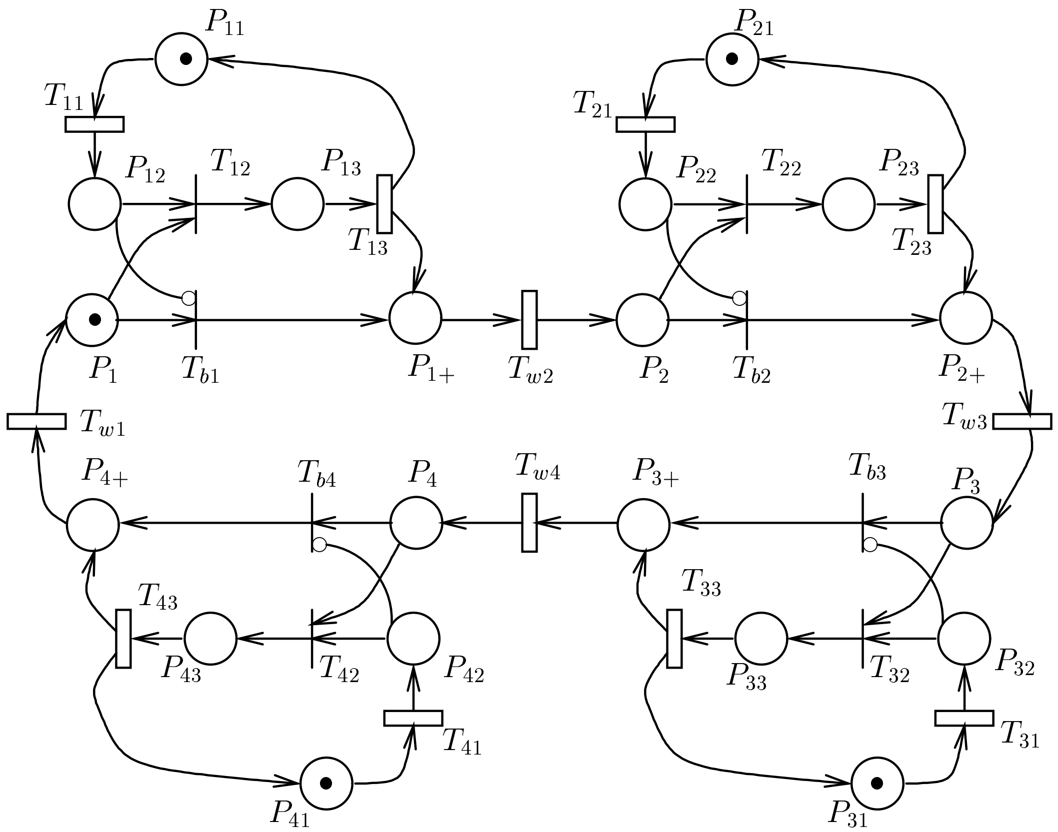


Figure 9: GSPN representing the PC LAN with four nodes

7 Using a GSPN for Performance Evaluation

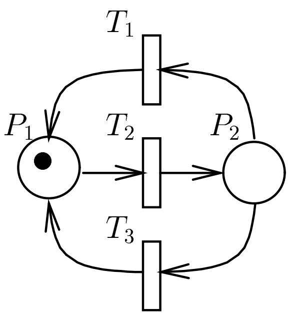
In this note we will consider two aspects of using a GSPN model of a system once it has been constructed: generating and solving a corresponding Markov process, and deriving performance measures. At the end of the note we will summarise the assumptions we are making when we use GSPN models.

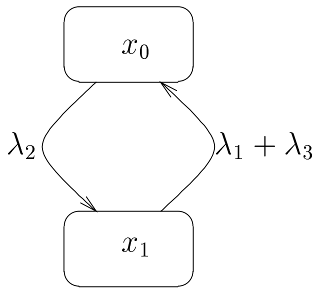
7.1 Generating and solving the corresponding Markov process

As stated in Section 5.2 generating the Markov process underlying an SPN model is very straightforward. We take advantage of the isomorphism between the reachability graph of the SPN and the state transition diagram of the Markov process. If the markings of the SPN are $\{M_0, M_1, \ldots, M_N\}$ (where M_0 is the initial marking), then the states of the Markov process will be $\{x_0, x_1, \ldots, x_N\}$ generated as follows:

- we associate a state, x_i, in the Markov process with every marking, M_i, in the reachability graph of the SPN;
- the transition rate from state x_i (corresponding to marking M_i) to state x_j (M_j), is obtained as the sum of the firing rates of the transitions that are enabled in M_i and whose firings generate marking M_j.

If we consider the very simple SPN model shown below, $M_0 = (1, 0)$ and $M_1 = (0, 1)$, are the only possible markings. Suppose that the firing rates of transitions T_1, T_2 and T_3 are λ_1, λ_2 and λ_3 respectively.





$$\mathbf{Q} = \begin{pmatrix} -\lambda_2 & \lambda_2 \\ \lambda_1 + \lambda_2 & -(\lambda_1 + \lambda_2) \end{pmatrix}$$

We associate states x_0 and x_1 with the markings M_0 and M_1. The transition rate from x_0 to x_1 is λ_2 because T_2 is the only transition enabled in M_0 whose firing results in M_1. The transition rate from x_1 to x_0 is $\lambda_1 + \lambda_3$ since both T_1 and T_3 are enabled in M_1, and the firing of either of them will result in the marking M_0.

Once the Markov process corresponding to an SPN model has been generated, it is solved in exactly the same way as Markov processes which are constructed directly during modelling. The steady state probability distribution is found by solving the global balance equations, together with the normalisation condition.

Two additional features were added to SPN notation to give GSPN notation: inhibitor arcs and immediate transitions. The effect of inhibitor arcs in a GSPN model is to alter the reachability graph of model: some markings and transition firings which would have been possible in the absence of the inhibitor arcs may no longer be possible. However, the effect of this new type of arcs only impacts on the generation of the reachability graph, not on the subsequent generation of the underlying Markov process. Once the reachability graph has been constructed obeying the constraints of the inhibitor arcs, they may be forgotten as far as generating the Markov process is concerned. Unfortunately, the same is not true for immediate transitions.

Recall that in a Markov process the delay spent in a state must be exponentially distributed. However, in a GSPN the delay in a vanishing marking is of length zero because the immediate transition will fire at once, moving the GSPN on to another marking. As far as the Markov process is concerned this means that these markings must be eliminated from the reachability graph before the state space of the Markov process is generated. Moreover, if immediate transitions from a marking can lead to two or more different markings, the transition rates to these markings need to be adjusted (by the decomposition principle), as explained below.

Let us consider first the simplest case: when a vanishing marking, M_v, enables a single immediate transition. In this case the next marking will clearly be the one resulting from firing the immediate transition. We will call this the *successor marking*, M_s. First we delete the vanishing marking from the reachability graph; similarly, we delete the arc *from* M_v *to* M_s. In principle, there may have been arcs *to* M_v *from* any other marking, M_i, in the reachability set. Each such arc is now taken directly from M_i to M_s; the name of the eliminated immediate transition is added to the label of the arc, but the rate or probability associated with the arc is unchanged.

If a vanishing marking, M_v enables more than one immediate transition it represents a conflict state, and there will be more than one arc leaving the marking in the reachability graph, each arc leading to a different possible successor marking. We will call these markings the *successor set*. As before, we delete M_v from the reachability graph. We also delete all arcs *from* M_v *to* markings in the successor set. For each arc which did come *to* M_v, we now form an arc to every marking in the successor set. Thus, if there are three markings in the successor set (i.e. the vanishing marking enabled three immediate transitions) then a single arc to the vanishing marking which has now been deleted will be replaced by three arcs, one to each of the three successor markings. As in the case of a single immediate transition, the name of the transition which has been removed is added to each of these new arcs. In addition, if these arcs represent timed transitions the rate associated with the arc must be adjusted in each case to represent the probability of that successor marking. For example, if the rate from some marking M_i to the vanishing marking, M_v, was r, and M_v enabled two immediate transitions, whose firing weights were equal (i.e. they were equally likely to fire) then the rate to each of the two successor markings will be $r/2$. Similarly, if the arcs represent immediate transitions then the relative probability of each of the new arcs must be chosen to reflect the original probability of the arc multiplied by the probability of the immediate transition which has been removed. For example, if the probability of the transition from some marking M_i to the vanishing marking, M_v, was p, and, as above, M_v enabled two immediate transitions whose firing weights were equal, then the probability of the transitions from M_i to the two successor markings would be $p/2$.

This procedure is systematically applied to all the vanishing markings in the reachability graph. In the end all arcs in the modified reachability graph will have a rate originating from a timed transition associated with every arc. As explained above, this rate may have been adjusted during the elimination of vanishing markings to reflect the relative probability of immediate transitions enabled after the timed transition. It is this modified reachability graph that is used to generate the Markov process underlying a GSPN model. Each tangible marking corresponds to a state in the Markov process, and the transition rate between states of the Markov process are derived as described above for SPNs. Once the Markov process is generated, it is solved as before, via the generator matrix.

7.1.1 Example: The reader-writer system revisited

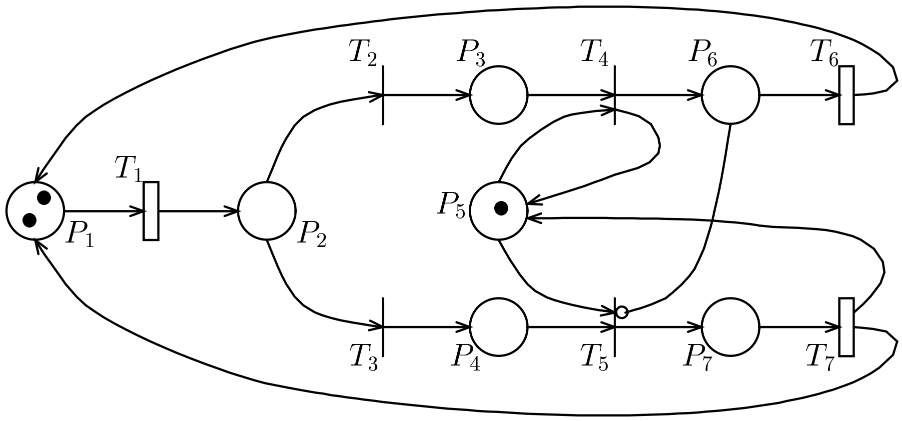


Figure 10: GSPN representing the simple reader-writer system

M_0	$(2,0,0,0,1,0,0)$	tangible
M_1	$(1,1,0,0,1,0,0)$	vanishing
M_2	$(1,0,1,0,1,0,0)$	vanishing
M_3	$(1,0,0,1,1,0,0)$	vanishing
M_4	$(1,0,0,0,1,1,0)$	tangible
M_5	$(1,0,0,0,0,0,1)$	tangible
M_6	$(0,1,0,0,1,1,0)$	vanishing
M_7	$(0,0,1,0,1,1,0)$	vanishing
M_8	$(0,0,0,1,1,1,0)$	tangible
M_9	$(0,0,0,0,1,2,0)$	tangible
M_{10}	$(0,1,0,0,0,0,1)$	vanishing
M_{11}	$(0,0,1,0,0,0,1)$	tangible
M_{12}	$(0,0,0,1,0,0,1)$	tangible

Table 1: Table showing the markings of the reader-writer GSPN model

We will now illustrate the elimination of vanishing markings for the reachability graph of the reader-writer GSPN model, shown in Figure 10. Note that the parametric marking of place P_1, considered in the previous lecture note, has now been replaced in the initial marking by two tokens in P_1.

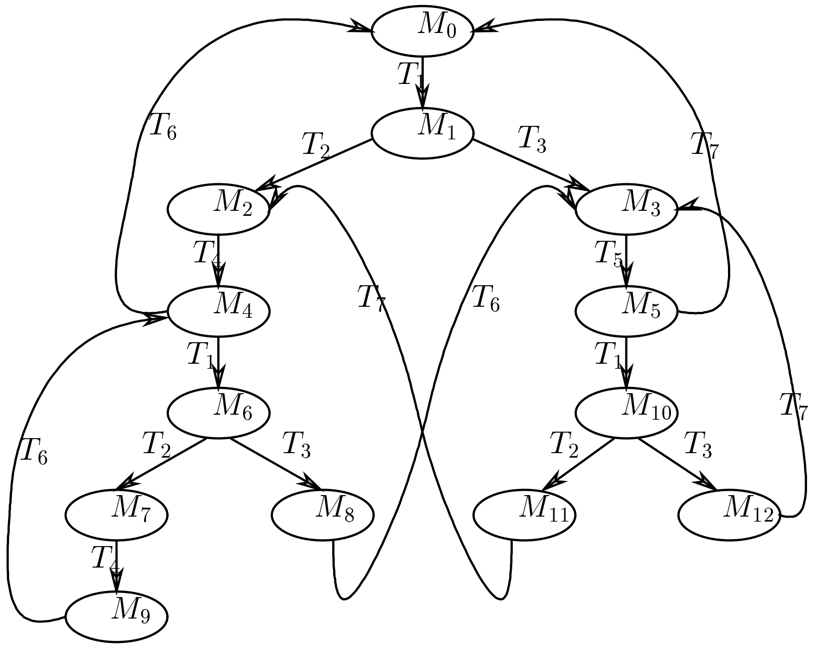


Figure 11: Reachability graph of the reader-writer GSPN model

The reachability set of the model is given in Table 1, together with an indication of whether each marking is vanishing or tangible, i.e. whether it enables an immediate transition or not. Note that there are 13 markings; if we considered the untimed Petri net with the same structure it would have 19 states. The reduction is caused by the priority of immediate transitions.

The full reachability graph, including vanishing markings, is shown in Figure 11.

Exercise: In this, and each of the modifications of the reachability graph, the transition rates have been omitted from the arcs—*you should write them in.* (Recall that the rate at which each process undertakes independent work is λ, the rate at which a process can perform a read access is r and the rate at which a process can perform a write access is w. The probability that a process chooses to read is 0.6, while the probability that it chooses to write is 0.4.)

In Figure 12 the effect of removing the vanishing markings which enable the choice between T_2 and T_3 is shown. For example, marking M_1 is now eliminated. The arc which previously went from M_0 to M_1 is now replaced by two arcs, one from M_0 to M_2 and one from M_0 to M_3. These arcs are labelled by the transitions which must fire to make this transformation of the marking occur: in this case, $T_1 + T_2$ and $T_1 + T_3$. The rates labelling arcs will also be adjusted. For example in M_0 the rate of the transition to M_1 will be $2 \times \lambda$ since independent processing takes place concurrently. The choice of transition T_2 occurs with probability 0.6. Therefore the rate which will be associated with the arc from M_0 to M_2 in the modified reachability graph is $1.2 \times \lambda$. Similarly, the arc from M_4 to M_7 will now be labelled with rate $0.6 \times \lambda$.

Once the vanishing markings which enabled the choice between T_2 and T_3 have been

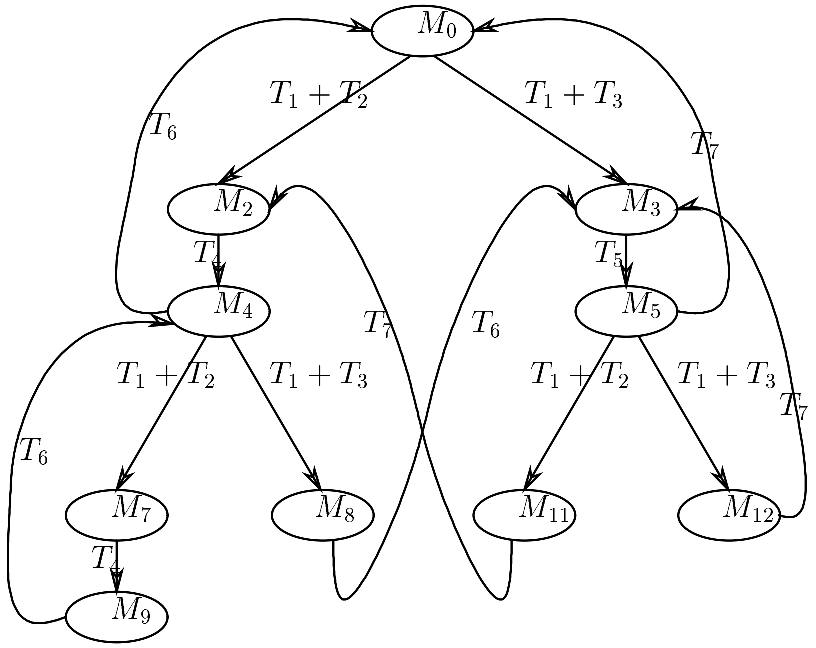


Figure 12: Reachability graph of the reader-writer model with vanishing markings which enable choices removed

eliminated, it remains to remove those markings which enable T_4 and T_5. Note that it is always the case that if one of these transitions is enabled in a marking it is the only transition enabled. Therefore elimination of the corresponding vanishing markings is simpler in this case. For example, M_2 enables T_4. In Figure 13 we can see that this marking has been deleted. The arc from M_0 to M_2 (introduced in the last step) is now deleted and replaced by an arc from M_0 to M_4. Similarly arcs from M_{11} and M_9 are now redirected to M_4. In each case the label of the arc is changed appropriately.

7.2 Deriving performance measures

The steady state probability distribution, $\boldsymbol{\pi}$, is still the basis of performance evaluation. In other words the aim is to derive performance characteristics of the system based on the steady state probability of being in certain states, or markings. Recall that when we were modelling directly at the Markov process level, defining performance measures, such as the average number of data packets waiting at a PC, involved examining the state representation of each state. Now, however, we can identify the states we are interested in by their characteristics at the GSPN level.

At the GSPN level the states (markings) which we are interested in can usually be identified either by whether a particular transition is enabled, or by whether a particular place is marked. To derive a performance measure we then associate a value with each of the markings we are interested in, just as we did when working directly at the Markov process level (cf. Section 3.3). For example, to derive the utilisation of the database in the reader-writer system, we associate a value 1 with any marking in which transitions T_6 or T_7 are enabled, and a value 0 with all other markings. In the CSPL file for SPNP

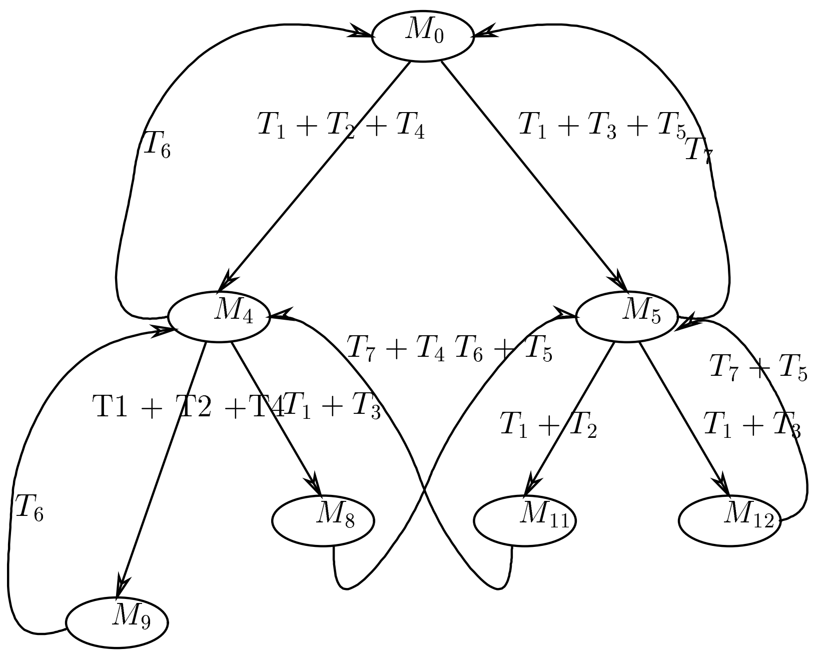


Figure 13: Reachability graph of the reader-writer model with all vanishing markings removed

this is expressed as:

```
 double util()
{ if (enabled("t6") || enabled("t7")) return(1.0); else return(0.0); }
```

The value associated with each marking, 1.0 or 0.0 in the case of `util`, is generally termed a *reward*. Different rewards can be used to calculate different measures. Indeed, typically a reward over all markings must be defined for each performance measure to be calculated. For example, if we wanted to derive the throughput of accesses to the database we associate a reward of $w(= 0.01)$ with all markings in which T_7 is enabled, and a reward of $r(= 0.05)$ multiplied by the number of readers (number of tokens in P_6) with all markings in which T_6 is enabled. In CSPL this is expressed as:

```
double tput() {
    if (enabled("t7"))
        return(0.01);
    else if (enabled("t6"))
        return(0.05*mark("p6"));
    else
        return(0.0);
}
```

The expected values of these rewards is generated and printed in the output file by the appropriate calls to `pr_expected` in the `ac_final` function:

```
void ac_final() {
```

```
    pr_mc_info();
    pr_std_average();

    pr_expected("Utilisation",util);
    pr_expected("Throughput",tput);
}
```

7.3 Assumptions

Since we are using the GSPN to generate a Markov process which we then solve numerically using the techniques discussed in lecture note 3 the assumptions we need to make about our model are the same ones as are needed for Markov processes in general. However, since we are now modelling at the GSPN level rather than the state space level, it is perhaps more natural to consider what these assumptions tell us about the GSPN.

Recall that in order to ensure that the stochastic process we were considering was a Markov process we made the assumption that all delays and inter-event times within the model were exponentially distributed. As we have seen, in a GSPN this condition is violated because some inter-event times have no duration: in a vanishing marking the time until the next event (transition firing) is zero. However, as we have demonstrated that it is possible to get round this problem by eliminating the vanishing markings before the Markov process is generated.

Assumptions we previously made to ensure that the steady state probability distribution coincided with the long-term probability distribution are that the model is finite, time homogeneous and irreducible. Since we still rely on the steady state probability distribution to derive performance measures we are making the same assumptions for GSPN.

Finite implies that the number of markings in the reachability set of a model (both tangible and vanishing markings) is finite. It can be shown that a GSPN is finite if it is *bounded*. A place in a Petri net is k-bounded if the number of tokens in the place will never exceed k. A Petri net is bounded if every place in the net is K-bounded for some finite value K.

Time homogeneity implies that the firing characteristics and system dynamics of a model do not change depending on when you observe it. This does not necessarily mean that these characteristics are static—we have seen that marking dependent rates and infinite server semantics may mean that the rate at which a transition fires may vary—but that firing rate will change only as dictated by the state of the model, not by how long it has been running.

Irreducibility implies that it is possible to reach an arbitrary state from every other state. In particular it must be possible to reach the initial marking from every reachable marking of the GSPN, and this implies that the reachability graph is strongly connected.

Jane Hillston ⟨jeh@inf.ed.ac.uk⟩. October 7, 2005.

8 Stochastic Process Algebra

In this lecture note we consider another class of performance modelling paradigms—stochastic extensions of *process algebras*. Like queueing networks and stochastic Petri nets, and their variants, these formal languages can be regarded as high-level model specification languages for low-level stochastic models. As we will see, the development of SPA has been very similar to that of SPN: in both cases an untimed formalism, used for studying the correct functional behaviour of systems, is extended by associating exponential delays with actions and reachability analysis is used to construct a corresponding Markov process. The advantages of SPAs are that they incorporate the attractive features of process algebras and thus bring to the area of analytic performance modelling several attributes which are not offered by the existing formalisms. Perhaps the most important such feature is the compositionality which is inherent in the models and can be exploited during their analysis. In this note we present some simple examples which illustrate this and several other aspects of the SPA approach to performance modelling. Several stochastic process algebras have appeared in the literature but they are all broadly similar. Here we will concentrate on PEPA (Performance Evaluation Process Algebra).

8.1 Process Algebras

Process algebras are abstract languages used for the specification and design of concurrent systems. The most widely used process algebras are Milner's Calculus of Communicating Systems (CCS) and Hoare's Communicating Sequential Processes (CSP) and the SPAs take inspiration from both these formalisms. Models in CCS and CSP have been used extensively to establish the correct behaviour of complex systems by deriving *qualitative* properties such as *freedom from deadlock* or *livelock*.

In the process algebra approach systems are modelled as collections of entities, called *agents*, which execute atomic *actions*. These actions are the building blocks of the language and they are used to describe sequential behaviours which may run concurrently, and synchronisations or communications between them.

In CCS two agents communicate when one performs an action, a say, while the other performs the complementary action $\overline{a}$. The resulting communication action is regarded as an *internal* action that is invisible to the environment. Agents may proceed with their internal actions simultaneously but it is important to note that the semantics given to the language imposes an interleaving on such concurrent behaviour, i.e. it is not possible for two actions to occur simultaneously. The grammar of the language makes it possible to construct an agent which has a designated first action (prefix); has a choice over alternatives (choice); or has concurrent possibilities (composition).

The communication mechanism is different in CSP as there is no notion of complementary actions: this is a major distinction between CCS and CSP. In CSP two agents communicate by simultaneously executing actions with the same label. Since during the communication the joint action remains visible to the environment, it can be reused by other concurrent processes so that more than two processes can be involved in the communication. This is the communication mechanism adopted in the SPA languages.

8.2 PEPA

Process algebras offer several attractive features which are not necessarily available in existing performance modelling paradigms. The most important of these are *compositionality*, the ability to model a system as the interaction of its subsystems, *formality*, giving a precise meaning to all terms in the language, and *abstraction*, the ability to build up complex models from detailed components but disregarding internal behaviour when it is appropriate to do so. Queueing networks offer compositionality but not formality; SPN and GSPN offer formality but not compositionality; neither queueing networks nor Petri nets offer abstraction mechanisms.

PEPA extends classical process algebra by associating a random variable, representing duration, with every action. These random variables are assumed to be exponentially distributed and this leads to a clear relationship between the process algebra model and a Markov process.

PEPA models are described as interactions of *components*. Each component can perform a set of actions: an action a is described as a pair (α, r), where α is the *type* of the action and $r \in \mathbb{R}^+$ is the parameter of the exponential distribution governing its duration. Whenever a process P can perform an action, an instance of a given probability distribution is sampled: the resulting number specifies how long it will take to *complete* the action. A small set of combinators is used to build up complex behaviour from simpler behaviour. The combinators are familiar: sequential composition (*prefix*), *choice*, synchronisation (*cooperation*) and abstraction(*hiding*). We explain each of them below, in terms of a extremely simple model of a web-based information system.

Prefix (.): A component may have purely sequential behaviour, repeatedly undertaking one activity after another and eventually returning to the beginning of its behaviour. A simple example is a web server, which allows one data transfer at a time. Each browser requiring web pages etc. will need to acquire access to the server and only when the transfer is complete will the server be released and available again for acquisition.

$$Server \stackrel{\mathit{def}}{=} (get, \top).(download, \mu).(rel, \top).Server$$

In some cases, as here, the rate of an action is outside the control of this component. Such actions are carried out jointly with another component, with this component playing a passive role. For example, the server is passive with respect to the *get* action and this is recorded by the distinguished symbol, $\top$ (called "top").

Choice (+): A choice between two possible behaviours is represented as the sum of the possibilities. For example, if we consider a browser in the information system, displaying the current data may have two possible outcomes: demand for access to data available in the local cache (with probability p_1) or demand for access to data stored at the remote server (with probability $p_2 = 1 - p_1$). These alternatives are represented as shown below:

$$\begin{aligned} Browser \stackrel{\mathit{def}}{=} \; & (display, p_1\lambda).(cache, m).Browser \; + \\ & (display, p_2\lambda).(get, g).(download, \top).(rel, r).Browser \end{aligned}$$

A race condition is assumed to govern the behaviour of simultaneously enabled actions and the continuous nature of the probability distributions ensures that the actions cannot occur simultaneously. Thus a sum will behave as either one summand or the other. When an action has more than one possible outcome, e.g. the *display* action in the browser, it is represented by a choice of separate actions, one for each possible outcome. The rates of these actions are chosen to reflect their relative probabilities (decomposition principle). Note that in a GSPN we would represent this situation by a single timed transition to represent the *display* action, which when it fired enabled two immediate transitions with weights p_1 and p_2 to reflect the different possible outcomes.

Cooperation ($\bowtie_L$): We have already anticipated that the browser and the server in the example will be working together within the same system. This will require them to *cooperate* when the browser needs to download data which is not available locally. In contrast, the local activities of the browser can be carried out independently of the server. Cooperation over given actions is reflected in the cooperation by the *cooperation set*, $L = \{get, download, rel\}$ in this case. Actions in this set require the simultaneous involvement of both components. The resulting action, a *shared* action, will have the same type as the two contributory actions and a rate reflecting the rate of the action in the slowest participating component. Note that this means that the rate of a passive action will become the rate of the action it cooperates with.

If, for simplicity, we assume that the system consists of just two browsers, the system is represented as the cooperation of the browsers and the server as follows:

$$WEB \stackrel{def}{=} (Browser \parallel Browser) \bowtie_L Server \qquad L = \{get, download, rel\}$$

The combinator $\parallel$ is a degenerate form of the cooperation combinator, formed when two components behave completely independently, without any cooperation between them, as in the case of the two browsers. This pure parallel combinator can be thought of as cooperation over the empty set: $(Browser \bowtie_{\emptyset} Browser)$.

Abstraction (/): It is often convenient to hide some actions, making them private to the component or components involved. The duration of the actions is unaffected, but their type becomes hidden, appearing instead as the unknown type τ. Components cannot synchronise on τ. For example, as we further develop the model of the information system we may wish to hide the access of a browser to its local cache. This might lead to a new representation of the browser:

$$Browser' \stackrel{def}{=} Browser/\{cache\}$$

and a corresponding new representation of the system:

$$WEB' \stackrel{def}{=} (Browser' \parallel Browser') \bowtie_L Server \qquad L = \{get, download, rel\}$$

Use of the hiding combinator in this way has two implications. Firstly, it ensures that no components added to the model at a later stage can interact, or interfere, with this action of the browser. Secondly, private actions are deemed to have no contribution to the performance measures being calculated and this might subsequently suggest simplifications to the model.

Throughout the simple example above we have used constants (names) such as $Server$ to associate identifiers with behaviours. Using recursive definitions we are able to describe components with infinite behaviours.

Representing the components of the system as separate components means that we can easily extend our model. Now we may want to consider a system consisting of more than two browsers which act independently of each other but compete for the use of the web server. This extension may be achieved compositionally by combining more instances of the $Browser$ component already described. For example, in the case of four browsers we have:

$$WEB4 \stackrel{def}{=} (Browser \parallel Browser \parallel Browser \parallel Browser) \underset{L}{\bowtie} Server$$

where, as before, $L = \{get, download, rel\}$.

8.3 Model analysis

The formality of the process algebra approach allows us to assign a precise meaning to every language expression. This implies that once we have a language description of a given system its behaviour can be deduced automatically. The meaning, or semantics, of a PEPA expression is provided by a formal semantics, in the structured operational style, which associates a labelled transition system with every expression in the language. This form of directed graph shows the possible evolutions of the model.

Formally, a labelled transition system $(S, T, \{\xrightarrow{t} \mid t \in T\})$ consists of a set of states S, a set of transition labels T and a transition relation $\xrightarrow{t} \subseteq S \times S$. For PEPA the states are the syntactic terms in the language, the transition labels are the actions ($(type, rate)$ pairs), and the transition relation is given by the semantic rules[1].

Based on the transition relation, a transition diagram, called the *derivation graph*, can be associated with each language expression. This graph describes all the possible evolutions of any component. It is completely analogous to the reachability graph of an SPN or GSPN. There the firing rule defined the semantics of the model; here the semantics is given by a set of transition rules, at least one for each combinator. An example derivation graph is shown in Figure 14 where the derivation graph of the PEPA model $WEB1$, consisting of a single browser accessing the web server, is shown. For clarity, in the upper part of the figure the derivatives of the components $Browser$ and $Server$ have been expanded. Inspection of the derivation graph allows us to derive qualitative properties of the model. In this case, for instance, we can see that the PEPA model is free from deadlock. Moreover, the Markov process underlying the PEPA model is obtained directly from the derivation graph: a state of the Markov process is associated with each node of the graph and the transitions between states are defined by considering the rates labelling the arcs. Since all activity durations are exponentially distributed, the total transition rate between two states will be the sum of the activity rates labelling arcs connecting the corresponding nodes in the derivation graph. Starting from the derivation graph of Figure 14, the derivation of the corresponding Markov process is straightforward

[1]Note that in contrast to CCS, the number of instances of a transition (action) is significant since it can affect the timing behaviour of a component.

$$
\begin{aligned}
Browser &\stackrel{def}{=} (display, p_1\lambda).Browser_1 + (display, p_2\lambda).Browser_2 \\
Browser_1 &\stackrel{def}{=} (cache, m).Browser \qquad Browser_2 \stackrel{def}{=} (get, g).Browser_3 \\
Browser_3 &\stackrel{def}{=} (download, \top).Browser_4 \qquad Browser_4 \stackrel{def}{=} (rel, r).Browser \\
Server &\stackrel{def}{=} (get, \top).Server_1 \qquad Server_1 \stackrel{def}{=} (download, \mu).Server_2 \\
Server_2 &\stackrel{def}{=} (rel, \top).Server \\
WEB1 &\stackrel{def}{=} Browser \underset{L}{\bowtie} Server \qquad L = \{get, download, rel\}
\end{aligned}
$$

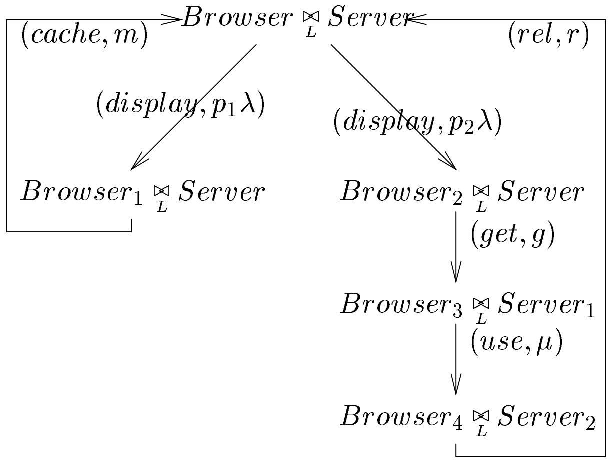


Figure 14: Derivation graph underlying Sys_4.

and results in the following generator matrix.

$$
\boldsymbol{Q} = \begin{pmatrix} -\lambda & p_1\lambda & p_2\lambda & 0 & 0 \\ m & -m & 0 & 0 & 0 \\ 0 & 0 & -g & g & 0 \\ 0 & 0 & 0 & -\mu & \mu \\ r & 0 & 0 & 0 & -r \end{pmatrix}
$$

As usual, the probability distribution of the states of the model is generally not the ultimate goal of performance analysis. In order to define performance measures such as throughput and utilisation the modeller can define a reward structure over the Markov process at the level of the process algebra. Just as in a GSPN, using CSPL, we defined rewards in terms of the marking of places and the enabling of transitions, when using PEPA we associate rewards with actions being enabled[2]. The reward associated with a state is then the total reward attached to the actions it enables. Note that no reward can be attached to internal, τ, actions.

8.4 A PEPA model of the PC LAN Example with 4 nodes

The model below is a PEPA representation of the PC LAN with four nodes considered in lecture note 4 (and again in lecture note 6). Each PC is represented by a separate

[2]There is no equivalent to *the number of tokens in a place* within a PEPA model.

component which, when empty (e.g. PC_{n0}) can either have an arrival or will allow the token to "walk" past ($walkon_n$). If the PC has a data packet waiting (PC_{n1}) it remains in that state until served by the token. The token moves round the LAN deciding what to do at each node on the basis of the actions offered by the corresponding PC. Thus although it looks as if it has a choice at each node, in fact only one of the actions $walkon_n$ and $serve_n$ will be enabled at any given time (these actions must be carried out in cooperation with the appropriate PC).

$$
\begin{aligned}
PC_{10} &\stackrel{def}{=} (arrive, \lambda).PC_{11} + (walkon_2, \omega).PC_{10} \\
PC_{11} &\stackrel{def}{=} (serve_1, \mu).PC_{10} \\[1em]
PC_{20} &\stackrel{def}{=} (arrive, \lambda).PC_{21} + (walkon_3, \omega).PC_{20} \\
PC_{21} &\stackrel{def}{=} (serve_2, \mu).PC_{20} \\[1em]
PC_{30} &\stackrel{def}{=} (arrive, \lambda).PC_{31} + (walkon_4, \omega).PC_{30} \\
PC_{31} &\stackrel{def}{=} (serve_3, \mu).PC_{30} \\[1em]
PC_{40} &\stackrel{def}{=} (arrive, \lambda).PC_{41} + (walkon_1, \omega).PC_{40} \\
PC_{41} &\stackrel{def}{=} (serve_4, \mu).PC_{40}
\end{aligned}
$$

$$
\begin{aligned}
Token_1 &\stackrel{def}{=} (walkon_2, \omega).Token_2 + (serve_1, \mu).(walk_2, \omega).Token_2 \\
Token_2 &\stackrel{def}{=} (walkon_3, \omega).Token_3 + (serve_2, \mu).(walk_3, \omega).Token_3 \\
Token_3 &\stackrel{def}{=} (walkon_4, \omega).Token_4 + (serve_3, \mu).(walk_4, \omega).Token_4 \\
Token_4 &\stackrel{def}{=} (walkon_1, \omega).Token_1 + (serve_4, \mu).(walk_1, \omega).Token_1
\end{aligned}
$$

$$
\begin{aligned}
LAN &\stackrel{def}{=} (PC_{10} \parallel PC_{20} \parallel PC_{30} \parallel PC_{40}) \underset{L}{\bowtie} Token_1 \\
&\quad \text{where } L = \{walkon_1, walkon_2, walkon_3, walkon_4, serve_1, serve_2, serve_3, serve_4\}.
\end{aligned}
$$

Here we have arbitrarily chosen a starting state in which all the PCs are empty and the Token is at PC1.

8.5 The PEPA Workbench

The PEPA Workbench, is a suite of tools, intended to help a modeller to solve and experiment with PEPA models. It comes in various versions, but the one installed on the system under the alias `pepa` is a Java version which offers a number of different solution techniques. Information about the tool and a user guide (including a *quick start* section) can be found on the module web page.

It is important to note that there are two stages to solving a model, *mymodel*`.pepa`:

1. In the first stage the state space of the PEPA model is generated using the `Run` command. This results in a file *mymodel*`.states`.

2. In the second state this state space is interpreted as a Markov process and solved for steady state, using the appropriate solver. This results in the file *mymodel*`.steadystate`.

Models to be loaded into the tool must be in a file with extension `.pepa` and in a version of the syntax which can be recreated in ASCII. The most obvious change is that instead of $P \underset{\{a,b\}}{\bowtie} Q$ we would type `P <a, b> Q`. Additionally if symbolic rates (e.g. `r, s` etc.) have been used in the model they must be explicitly assigned values at the start of the file. Similarly probabilities must be explicit values within the `.pepa` file.

```
lambda = 0.05;
m=0.1;
g =0.01;
r=0.001;
mu= 1.0;

Browser= (display,0.4*lambda).(cache,m).Browser +
        (display,0.6*lambda).(get,g).(download,infty).(rel,r).Browser;
Server = (get,infty).(download,mu).(rel,infty).Server;
Browser<get,download,rel>Server
```

Figure 15: `WEB1.pepa` file

An example PEPA file, corresponding to the small *WEB1* example discussed earlier, is shown in Figure 15. Examples of the `.states` and `.steadystate` files, corresponding to this model, are shown in Figures 16 and 17 below[3].

```
1 Browser <{rel, get, download}> Server
2 (cache,m=0.1).Browser <{rel, get, download}> Server
3 (get,g=0.01).(download,1.0*Top).(rel,r=0.0010).Browser
                                             <{rel, get, download}> Server
4 (download,1.0*Top).(rel,r=0.0010).Browser <{rel, get, download}>
                                    (download,mu=1.0).(rel,1.0*Top).Server
5 (rel,r=0.0010).Browser <{rel, get, download}> (rel,1.0*Top).Server
```

Figure 16: `WEB1.states` file

8.6 Efficient Solution

PEPA models are prone to the problem of state space explosion: even moderately sized system descriptions can generate a huge state space making analysis of the system infeasible. It is an area of on-going research to investigate when the component structure of the model can be used in order to gain efficient solution of the underlying Markov process, i.e. when can we consider the components of the model separately and yet still derive answers about the model as a whole.

[3]More information about the PEPA Workbench and associated tools is available via the WWW— `http://www.dcs.ed.ac.uk/pepa/`.

```
1  0.0292141396435875
2  0.005842827928717507
3  0.08764241893076259
4  8.764241893076253E-4
5  0.8764241893076249
```

Figure 17: `WEB1.steadystate` file

As we will see in a later lecture note efficient *product form solution* is one of the major attractions of queueing networks for performance modelling purposes. These models rely on a form of interaction between nodes in a network which allows them to be solved in isolation, since they behave as if independent, up to normalisation. In some circumstances PEPA models (and GSPN models) can also be solved in this way—each component is solved in isolation and the steady state probability vector of the complete state space is formed from the product of the individual steady state distributions of the components.

8.7 Assumptions

The assumptions that we need to make about our models are the same ones that we needed to make about Markov processes in general. But just as when we were considering GSPNs we were able to express some of these assumptions in terms of the Petri net, we can express some of the assumptions in terms of PEPA.

In order to ensure that the underlying Markov process is ergodic, i.e. does have a steady state probability distribution, the derivation graph of a PEPA model must be strongly connected. Necessary conditions for ergodicity, have been defined in terms of syntactic rules for PEPA models; e.g. if cooperation occurs it must be the highest level combinator—you cannot have a choice between two components which are themselves cooperation expressions.

9 Queueing Networks

9.1 Introduction

Mathematicians have studied queues for approximately 100 years, and one of their first applications was to telephone exchanges (*Erlang's loss formula*). However they gained popularity with Computer Scientists about 35 years ago when it was realised that single queues, and networks of queues could be used as performance models of computer systems. More recently, developments in many of those systems have grown beyond the expressiveness of queueing networks. Nevertheless, for many people *performance analysis* is synonymous with *queueing theory*.

Queueing theory was first used in the late 1960s to model time-sharing computer systems. In particular single queues were used to study allocation policies for CPUs. Analysis of the single queues led to a new qualitative, as well as quantitative, understanding of some aspects of operating system and disk management system design. However, these single queue models were too restrictive to represent systems of interacting computing devices, as computers came to be viewed.

Later developments in queueing theory therefore studied the interaction between service centres or queues—*queueing networks*. Contemporary computers at the time could be viewed as a set of loosely coupled hardware components through which weakly interacting *jobs* or *transactions* were circulating. The success of queueing theory as a performance modelling paradigm relied on this view of computers. Computer and communication systems today do not fit into this model so readily; however, queueing networks remain a useful performance modelling paradigm, in some circumstances.

9.2 Single queues

The basic scenario for a single queue is that *customers*, who belong to some *population* arrive at the *service facility*. The service facility has one or more *servers* who are capable of performing the service required by customers. If a customer cannot gain access to a server it must join a queue, in a *buffer*, until a server is available. When service is complete the customer departs, and the server selects the next customer from the buffer according to the *service discipline*.

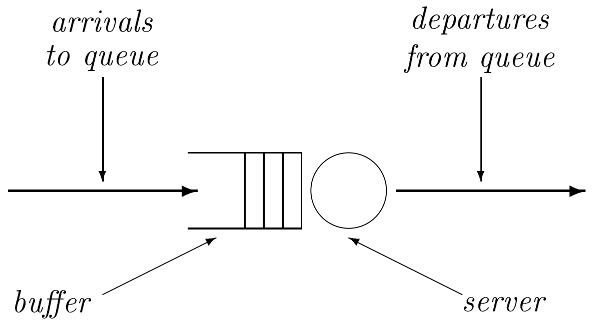


Figure 18: A Simple Queue

In order to describe a service facility accurately we need to know details about each of the terms emphasised above:

Arrival Pattern of Customers The ability of the service facility to provide service for an arriving stream of customers depends not only on the mean arrival rate λ, but also on the pattern in which they arrive, i.e. the *distribution function* of the inter-arrival times. In this course we will only be considering queues in which the times between arrivals are assumed to be exponentially distributed[1]. However, you should be aware that there are other possibilities.

Service Time Distribution The service time is the time which a server spends satisfying a customer. As with the inter-arrival stream, the important characteristics of this time will be both its average duration and the distribution function. Similarly, in this course we will only consider service times which obey an exponential distribution. If the average duration of a service interaction between a server and a customer is $1/\mu$ then μ is the *service rate*.

Number of Servers If there is more than one server in the service facility the processing of customers can go on concurrently: one customer at each server. It is assumed that the servers are indistinguishable from each other, so that it does not matter which server actually serves a customer—the results, and the service characteristics, will be the same.

The most commonly considered cases are of a *single server* or an *infinite server*. In the first case, we assume that the service facility only has the capability to serve one customer at a time; waiting customers will stay in the buffer until chosen for service; how the next customer is chosen will depend on the service discipline. In an infinite server facility we assume that there are always at least as many servers as there are customers, so that each customer can have a dedicated server as soon as it arrives in the facility. There is no queueing, (and no buffer) in such facilities. Between these two extremes there are *multiple server* facilities. These have a fixed number of servers (usually denoted c), each of which can service a customer at any time. If the number of customers in the facility is less than or equal to c there will no queueing—each customer will have direct access to a server. If there are more than c customers, the additional customers will have to wait in the buffer.

Buffer Capacity Customers who cannot receive service immediately that they require it must wait in the buffer and wait for a server to become available. Clearly the number of customers waiting may grow, filling the places of the buffer. If the buffer has finite capacity there is the possibility that it may become full. There are two possibilities in this case:

- either, the fact that the facility is full is passed back to the arrival process and arrivals are suspended until the facility has spare capacity (created by the completion of a customer who is currently being served);
- or, arrivals continue and arriving customers are turned away and lost until the facility has spare capacity again.

[1] Equivalently, the arrival stream obeys a Poisson distribution.

In some systems the buffer capacity is so large as to never affect the behaviour of the customers; in this case the buffer capacity is assumed to be infinite.

Population The characteristic of the population which we are interested in is usually the size. Clearly, if the size of the population is fixed, at some value N, no more than N customers will ever be requiring service at any time. When the population is finite the arrival rate of customers will be affected by the number who are already in the service facility. The arrival rate will be zero when all the population is already in the facility. Sometimes the size of the population is so large that this situation will never arise. In this case we assume that the population is infinite—this simplifies the mathematics.

Ideally, members of the population are indistinguishable from each other. When this is not the case we divide the population into *classes* whose members all exhibit the same behaviour. A class will represent a set of customers who share the same characteristics. Different classes differ in one or more characteristics, for example, arrival rate or service demand.

Service Discipline When more than one customer is waiting for service there has to be a rule for selecting which of the waiting customers will be the next one to gain access to a server. The commonly used service disciplines are

- FCFS *first-come-first-serve* (or FIFO *first-in-first-out*).
- LCFS *last-come-first-serve* (or LIFO *last-in-first-out*).
- RSS *random-selection-for-service*.
- PRI *priority*. The assignment of different priorities to elements of a population is one way in which classes are formed.

We will only be considering systems which obey the FCFS service discipline.

A shorthand notation for these six characteristics of a system is provided by *Kendall's notation* for classifying queues. In this notation a queue is represented as $A/S/c/m/N/SD$:

A denotes the arrival process; usually M, to denote Markov (exponential), G, general, or D, deterministic distributions.

S denotes the service rate and uses the distribution identifiers as above.

c denotes the number of servers available in the service facility.

m denotes the capacity of the **service facility** (buffer + server(s)), infinite by default.

N denotes the customer population, also infinite by default.

SD denotes the service discipline, FCFS by default.

Example: Consider a network gateway, at which measurements have shown that packets arrive at a mean rate of 125 packets per second, and are buffered until they can be transmitted. The gateway takes 2 milliseconds on average to transmit a packet. The buffer currently has 13 places, including the place occupied by the packet being transmitted and packets that arrive when the buffer is full are lost. We wish to find out if the buffer

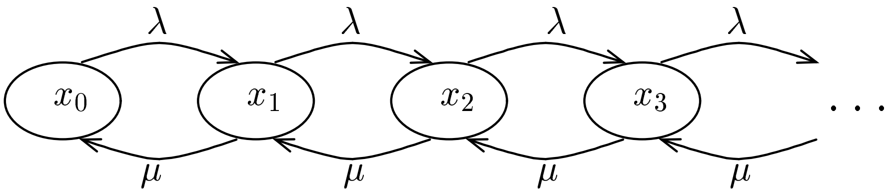


Figure 19: The state transition diagram for a simple $M/M/1$ queue

capacity is sufficient to ensure that less than one packet per million gets lost. Making exponential assumptions about the arrival rate and the service rate we would model the gateway as an $M/M/1/13/\infty/FCFS$ queue.

Example: Consider the mobile phone antenna discussed in Sections 1.5 and 3.2.1. Assuming that we are interested in the simpler characterisation of the system as a random variable, whose value denotes the number of frequencies in use, we can now see that this is a $M/M/6/6/\infty/FCFS$ queueing system. Since the last two characteristics are default values we would usually omit them and talk about an $M/M/6/6$ queue.

Example: A PEPA model of a $M/M/1/N$ queue is as follows:

$$\begin{aligned}
Arrivals &\stackrel{def}{=} (arrive, \lambda).Arrivals \\[1em]
Queue_0 &\stackrel{def}{=} (arrive, \top).Queue_1 \\
Queue_i &\stackrel{def}{=} (arrive, \top).Queue_{i+1} + (depart, \mu).Queue_{i-1} \qquad 1 \le i \le N-1 \\
Queue_N &\stackrel{def}{=} (depart, \mu).Queue_{N-1} \\[1em]
System &\stackrel{def}{=} Arrivals \underset{\{arrive\}}{\bowtie} Queue_0
\end{aligned}$$

9.3 Traffic Intensity

The two most important features of a single queue are the arrival rate of customers λ, and the service rate of the server(s), μ. These are combined into a single parameter which characterises a single or multiple server system[2], the *traffic intensity*.

$$\boxed{\text{traffic intensity, } \rho = \frac{\lambda}{c \times \mu}}$$

As we will see in the next lecture, many of the important performance measures associated with a queue can be expressed in terms of ρ. These are measures such as *utilisation* (the proportion of time the server is busy), *residence time* (the average time spent at the

[2] $c = 1$ in the single server case

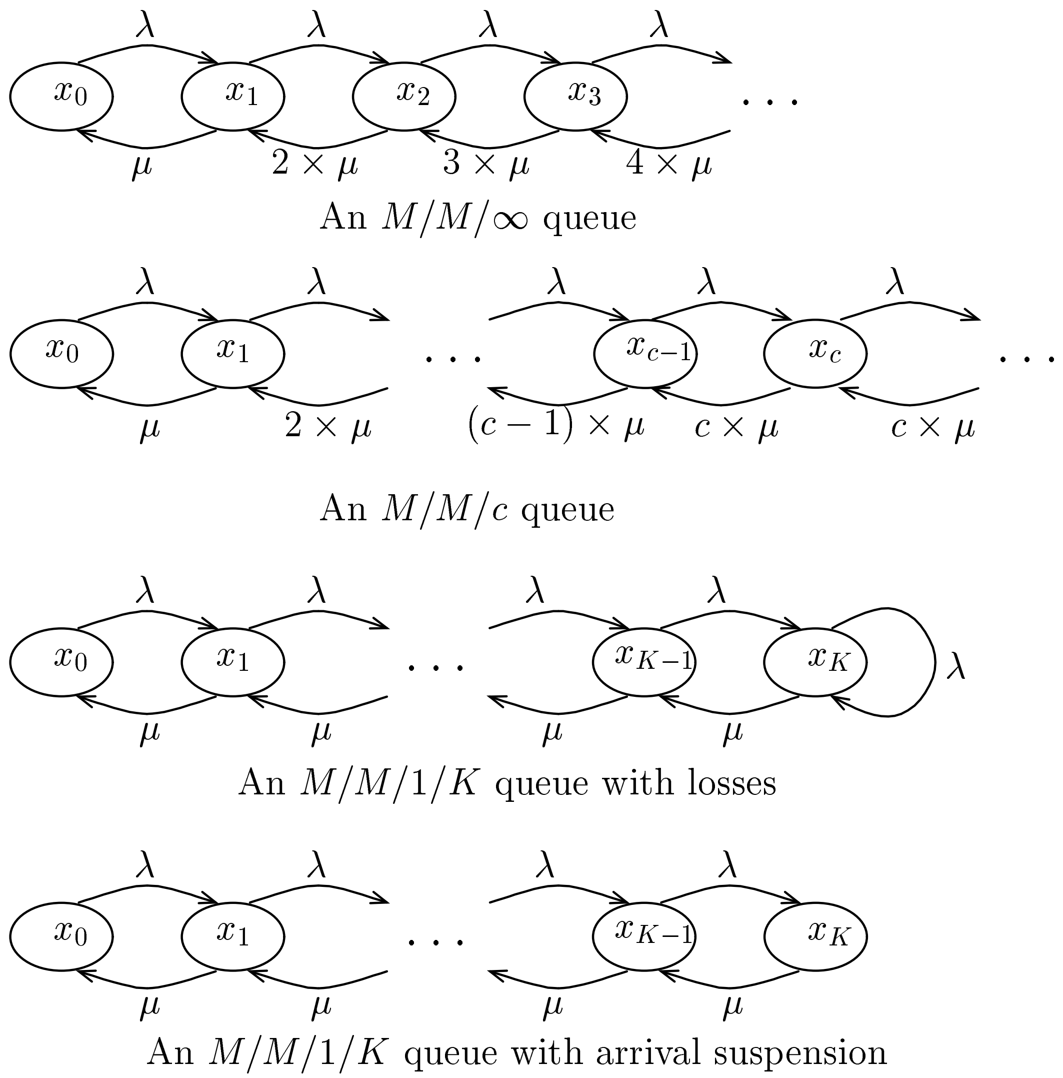


Figure 20: State transition diagrams for some other commonly occurring queues

service facility by a customer, both queueing and receiving service), *waiting time* (the average time spent queueing), *queue length* (the average number of customers at the service facility, both waiting and receiving service), and *throughput* (the rate at which customers pass through the service facility). For the moment we just point out that for the system to be *stable*, ρ must be less than 1: that is the arrival rate of customers must be less than the rate at which they are served. The alternative, if the arrival rate is greater than the service rate, would result in a queue which grew unboundedly.

9.4 Networks of Queues

If we view a computer as an interaction of devices, which jobs visit sequentially, then it is natural to model the system as a *queueing network*. Each device or resource of the system is represented by a separate service facility, now termed a *service centre*. Customers in the queueing network correspond to the users or transactions in the computer system.

It is assumed that after service at one service centre a customer may progress to other service centres in the system, following some previously defined pattern of behaviour, representing the task which the customer wishes to achieve. Note that this is very similar to the view of systems taken in operational analysis.

The characteristics of each service centre are analogous to the single queue case, i.e. the arrival rate, the service rate, the number of servers, the buffer capacity, the population and the service discipline. However, note that these characteristics are not completely independent for the service centres within a queueing network. For example, the population will usually be the same for all service centres, and the arrival rate at one service centre will often depend on the arrival rate at another. If a service centre with finite capacity buffer becomes full, service may be suspended at the service centres which send customers to this queue.

Defining a queueing network model of a particular system is made relatively straightforward by the close correspondence between the attributes of queueing network models and the attributes of the systems.

A queueing network can be represented as a directed graph in which the nodes represent the service centres (see Figure 21). Various conventions have been developed for this representation but you should note that this is an informal representation. In the case of GSPN the graph representation had a formal interpretation from which the dynamic behaviour of the system could be deduced. In contrast, the diagram of a queueing network is schematic, representing the topology of the system, and the route that customers take between resources. In the diagram the routing of customers is represented by the *routing probabilities*, which label branches in the paths between service centres. For example, in the network shown in Figure 21, a customer leaving the first service centre goes to the second service centre with probability p and leaves the system with probability $1 - p$.

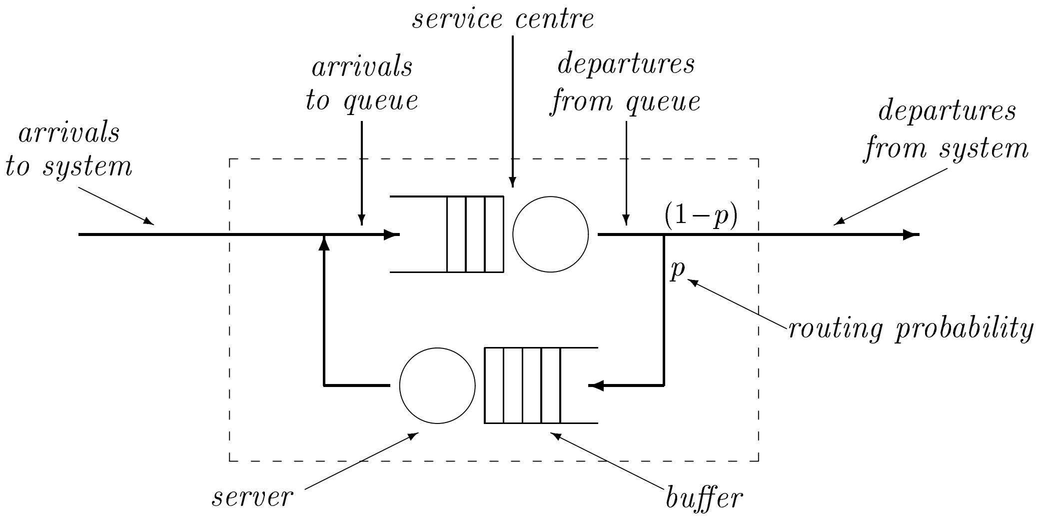


Figure 21: A Simple Open Queueing Network

The state of the system is typically represented as the number of customers currently occupying each of the service centres. There may be a number of different *classes* of customers each exhibiting different characteristics within the network. In this case the state is the number of customers of each class at each service centre. As well as differences in arrival rate and/or service demand, classes may be distinguished by the routes which customers take through the network.

A network may be *closed*, *open* or *mixed* depending on whether a fixed population of customers remain within the system; customers may arrive from, or depart to, some external environment; or there are classes of customers within the system exhibiting open and closed patterns of behaviour respectively.

9.5 Expressiveness

As mentioned in the Introduction, contemporary computer and communication systems often have features which are not easily represented by queueing networks. Some of the most important of these features are listed below. In some cases ways of adapting queueing models to represent, and provide solutions for, systems with such features have been found. However, these are new approaches and largely remain topics for research.

Simultaneous resource possession In a computer system a job may continue to compute while its I/O requests are progressing in parallel. Thus, the job may be simultaneously using two or more resources of the system.

Fork and Join Fork and Join primitives are used in computer systems to create and synchronise subprocesses. This causes the number of jobs in the system to change and also invalidates the assumption of independence among jobs.

Bulk and train arrivals In communication networks the arrival of packets may occur in batches (bulk arrivals) or in quick succession (train arrivals). Such situations do not fit assumptions of independence between customers, nor exponential inter-arrival times.

Load dependent arrivals Computer networks and distributed systems have intelligent load-balancing policies that cause new jobs or packets to go to one of a number of devices depending upon the load observed in the recent past.

Defections from the queue Communication systems sometimes set a maximum limit on the time that a packet or request is able to stay in a queue. Thus, the packets that have been in the queue for a long time are dropped from the queue on the assumption that they may have already been retransmitted by higher protocol layers.

10 Solving Queueing Models

10.1 Introduction

In this note we look at the solution of systems of queues, starting with simple isolated queues. The benefits of using predefined, easily classified queues will become apparent: many performance measures can be calculated directly from the parameters of the model. Obviously the situation becomes more complicated when queues are connected together. However we see that even in this case deriving performance measures can be very straightforward as we are able to consider each queue in isolation. Finally we summarise the assumptions which we need to make in order to obtain these solutions.

10.2 Single Queues

Since we assume that all the queues which we consider have a Markovian arrival process and a Markovian service process the queue can be modelled as a Markov process. The state transition diagram for a single-server queue with infinite capacity is shown in Figure 22. Note that unlike the Markov processes which we have considered earlier in the course this process has an infinite state space.

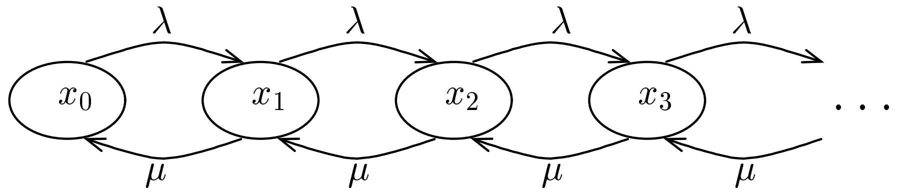


Figure 22: The state transition diagram for a simple $M/M/1$ queue

If we write the global balance equations for this system we can soon recognise a regular pattern emerging.

$$\begin{aligned}
\lambda \boldsymbol{\pi_0} &= \mu \boldsymbol{\pi_1} \\
(\lambda + \mu)\boldsymbol{\pi_1} &= \lambda \boldsymbol{\pi_0} + \mu \boldsymbol{\pi_2} \\
(\lambda + \mu)\boldsymbol{\pi_2} &= \lambda \boldsymbol{\pi_1} + \mu \boldsymbol{\pi_3} \\
&\vdots
\end{aligned}$$

Using simple algebra we can rewrite these as shown below:

$$\begin{aligned}
\boldsymbol{\pi_1} &= \frac{\lambda}{\mu}\boldsymbol{\pi_0} \\
\boldsymbol{\pi_2} &= \frac{\lambda}{\mu}\boldsymbol{\pi_1} = \left(\frac{\lambda}{\mu}\right)^2 \boldsymbol{\pi_0} \\
\boldsymbol{\pi_3} &= \frac{\lambda}{\mu}\boldsymbol{\pi_2} = \left(\frac{\lambda}{\mu}\right)^3 \boldsymbol{\pi_0} \\
&\vdots
\end{aligned}$$

Remembering that *traffic intensity*, ρ is defined to be λ/μ (for the single server case, $\lambda/(c \times \mu)$ in the general case) we can see that for an arbitrary state x_i

$$\boldsymbol{\pi_i} = \rho^i \boldsymbol{\pi_0}$$

and using the normalisation condition we see that

$$1 = \sum_{i=0}^{\infty} \boldsymbol{\pi_i} = \boldsymbol{\pi_0} \sum_{i=0}^{\infty} \rho^i = \boldsymbol{\pi_0} \frac{1}{1-\rho} \qquad \text{if } \rho < 1$$

From this we can deduce that

$$\boxed{\boldsymbol{\pi_0} = 1 - \rho \qquad \text{and} \qquad \boldsymbol{\pi_i} = (1-\rho)\rho^i \qquad \text{for all } i > 0.}$$

Thus we have a symbolic evaluation of the steady state distribution for any $M/M/1$ queue, i.e. the steady state distribution expressed in terms of the parameters of the model, λ and μ. This means that for any particular model we can find the steady state distribution simply by evaluating the expressions above with our particular value of $\rho = \lambda/\mu$. More importantly, we can derive symbolic expressions for the important performance measures we are likely to want to derive from a queue, and then evaluate those measures for a particular model *without having to carry out any numerical solution of global balance equations.*

Utilisation, U The queue is being utilised whenever it is non-empty; in other words the utilisation, U, is $1 - \boldsymbol{\pi_0}$.

$$\boxed{\textbf{Utilisation} \qquad U = \rho}$$

Mean number of customers in the queue, N This is the expectation of the number of customers in the service facility as a whole, i.e.

$$N = \sum_{n=1}^{\infty} n \times \boldsymbol{\pi_n} = \sum_{n=1}^{\infty} n \times (1-\rho)\rho^n = \frac{\rho}{(1-\rho)}$$

$$\boxed{\textbf{No. in queue} \qquad N = \frac{\rho}{1-\rho}}$$

Mean number of customers waiting, N_b This is the expectation of the number of customers in the buffer i.e.

$$N_b = \sum_{n=1}^{\infty} (n-1) \times \boldsymbol{\pi_n} = \sum_{n=1}^{\infty} (n-1) \times (1-\rho)\rho^n = \rho \sum_{n=1}^{\infty} n \times (1-\rho)\rho^n = \frac{\rho^2}{(1-\rho)}$$

$$\boxed{\textbf{No. in buffer} \qquad N_b = \frac{\rho^2}{1-\rho}}$$

Mean response time, R Using Little's Law we can calculate the mean response time of the queue to be the mean number in the queue N, divided by the arrival rate λ, i.e. $R = N/\lambda = \dfrac{\rho}{1-\rho} \times \dfrac{1}{\lambda} = \dfrac{1/\mu}{1-\rho} = \dfrac{1}{\mu(1-\rho)}$.

$$\boxed{\textbf{Response time} \qquad R = \frac{1}{\mu(1-\rho)}}$$

Other common performance measures can be calculated in a similar way. Similarly we can derive symbolic steady state distributions, and expressions for performance measures, for the other standard queues. Thus, given almost any single queue model, in order to derive a performance measure it is only necessary to select the appropriate formula from a table and evaluate it using the parameters of your model. Some examples are shown in Table 2. Most textbooks on performance models will contain these formulae.

Performance Measures	$M/M/1$	$M/M/c$	$M/M/1/K$
Traffic Intensity ρ	$\frac{\lambda}{\mu}$	$\frac{\lambda}{c \times \mu}$	$\frac{\lambda}{\mu}$
Utilisation U (per server)	ρ	ρ	$\rho(1 - \frac{(1-\rho)\rho^K}{1-\rho^{K+1}})$
Prob. system is idle $\boldsymbol{\pi_0}$	$1-\rho$	$\left(1 + \frac{(c\rho)^c}{c!(1-\rho)} + \sum_{n=1}^{c-1} \frac{(c\rho)^n}{n!}\right)^{-1}$	$\frac{1-\rho}{1-\rho^{K+1}}$
Prob. buffer non-empty B	ρ^2	$\frac{(c\rho)^c}{c!(1-\rho)}\boldsymbol{\pi_0}$	
Mean no. in system N	$\frac{\rho}{1-\rho}$	$c\rho + \rho B/(1-\rho)$	$\frac{\rho}{1-\rho} - \frac{(K+1)\rho^{K+1}}{1-\rho^{K+1}}$
Mean no. in buffer N_b	$\frac{\rho^2}{1-\rho}$	$\rho B/(1-\rho)$	$\frac{\rho}{1-\rho} - \rho\frac{1+K\rho^K}{1-\rho^{K+1}}$
Mean response time R	$\frac{1}{\mu(1-\rho)}$	$\frac{1}{\mu}\left(1 + \frac{B}{c(1-\rho)}\right)$	$\frac{N}{\lambda(1 - \frac{(1-\rho)\rho^K}{1-\rho^{K+1}})}$

Table 2: Common performance measures

Example: Consider again the network gateway discussed in the previous note. Measurements have shown that packets arrive at a mean rate of 125 packets per second, and are buffered until they can be transmitted. The gateway takes 2 milliseconds on average to transmit a packet. The gateway currently has 13 places (including the packet being transmitted) and packets that arrive when the buffer is full are lost. We wish to find out if the buffer capacity is sufficient to ensure that less than one packet per million gets lost.

We represent the gateway as a $M/M/1/13$ queue, with $\lambda = 125$ and $\mu = 1/0.002 = 500$. The utilisation of the gateway will be $\rho = \lambda/\mu = 0.25$.
The loss rate is the arrival rate multiplied by the probability that the system is full, i.e. $\lambda \times \boldsymbol{\pi_K}$. The proportion of packets that are lost is simply $\boldsymbol{\pi_K}$ (expected number lost per time unit divided by the expected number arriving per time unit)

$$\boldsymbol{\pi_K} = \frac{(1-\rho)\rho^K}{1-\rho^{K+1}} = (0.75\times 0.25^{13})/(1-0.25^{14}) = (1.1176\times 10^{-8})/0.99999999 = 1.12\times 10^{-8}$$

Thus the expected proportion of packets lost is 0.0112 every million packets, well within the requirement.

Exercise: Using maple or otherwise, investigate how much the buffer size can be reduced before the loss rate becomes unacceptable. What happens if the arrive rate increases to 175 packets per second?

10.3 Product Form Queueing Networks

Once we consider a network of queues rather than a single queue the possible forms of the state spaces of the underlying Markov process become much more diverse. Therefore we would not anticipate being able to derive symbolic steady state distributions of wide applicability in the same way as we have done for single queues. However, for a large class of queueing networks a straightforward and efficient means of solving models has been found. These networks are known as *product form networks.*

The term *product form* comes from the fact that the steady state distribution of these models can be derived as the product of the steady state distributions of each of the constituent service centres.

Recall that the state of a single service facility can be characterised by the number of customers currently in the system. In a queueing network the state of the system is characterised by the number of customers waiting at each of the service centres. This is usually represented as a tuple. For example, in a simple queueing network with two service centres, such as the one shown in Figure 23, the state $\boldsymbol{(n_1, n_2)}$ indicates that there are n_1 customers in service centre 1 (queueing or in service) and n_2 customers in service centre 2.

For a model such as this one to have a product form steady state distribution means that the distribution can be expressed as a product of terms representing the steady states of each of the service centres considered in isolation, e.g.

$$\boldsymbol{\pi}(n_1, n_2) = \boldsymbol{\pi_1}(n_1) \times \boldsymbol{\pi_2}(n_2)$$

In terms of the system what this means is that the two service centres reach their equilibrium behaviours and then behave independently of each other. (Recall that a probability

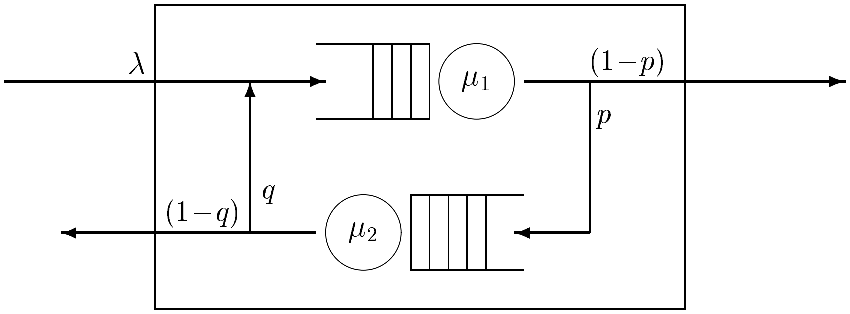


Figure 23: Open Queueing Network

of a coincidence of events can be expressed as the product of the probabilities of each event if and only if those events are independent.)

The class of queueing networks in which service centres exhibit this form of independent behaviour in equilibrium excludes some interesting and important system features as was discussed in the previous lecture note. However when the necessary conditions are satisfied, performance measures can be derived without resorting to the underlying Markov process—just as in the case of a single queue, because in effect we only solve the network one queue at a time.

10.3.1 Traffic equations

Using the already established formulae for individual service centres we only need to know two parameters: the arrival rate at the queue, which we will denote by λ_i for the ith queue, and the service rate at the queue, denoted μ_i. For a queueing network we will be given μ_i for each queue in the network. However, we will only usually be given the arrival rate of arrivals to the network from the environment (λ in the example shown in Figure 23). Fortunately the following theorems, the decomposition principle of Poisson/exponential distributions, and simple algebra, can help us to work out the arrival rate at each node in the network.

Burke's Theorem stated that a Poisson arrival process at a service centre with exponential service rates generates a Poisson process of departures. Moreover the rate of departure is the same as the arrival rate. This result implies that each service centre in a chain of simple exponential service centres with Poisson arrivals can be analysed independently using results from simple queues.

Jackson's Theorem generalised Burke's Theorem to consider an arbitrary network of exponential service centres each of which is driven by a Poisson arrival process. Although the presence of feedback paths destroys the Poisson nature of the service centre arrivals, Jackson showed that they still *behave* as if they were Poisson driven. Moreover the relationship between the arrival stream and the output stream is maintained. The networks considered by Jackson were *open*.

Again, this implies that each service centre in the queueing network can be analysed independently.

Gordon and Newell's Theorem considered a modification of Jackson's networks in which the network is *closed* rather than open. Closed queueing networks are not driven by any external Poisson arrival process. Instead, the number of customers in the network is fixed, K. Whenever a customer departs one service centre in the network it immediately requests service from another one. Hence customers never leave the network. Gordon and Newell's product form equation has the form:

$$\boldsymbol{\pi}(n_1, n_2, \ldots, n_k) = \frac{1}{G(K)} \prod_{i=1}^{n} \boldsymbol{\pi_i}(n_i)$$

where $G(K)$ is a *normalisation constant* chosen to ensure that the steady state probabilities sum to 1.

Using these results we know that if the arrival rate at queue i is λ_i then the departure rate will also be λ_i. Therefore, if all the departures from queue i go directly to queue $i+1$ the arrival rate at queue $i+1$ will also be λ_i, i.e. $\lambda_{i+1} = \lambda_i$.

However, in general the Poisson departure stream from a queue may be split between different queues and/or the environment, according to the routing probabilities. By the decomposition principle we know that when the departure stream is split in this way each of the resulting arrival streams will also be Poisson, with a rate obtained by multiplying the departure rate by the probability of this branch of the routing probability.

If we analyse all the service centres of a queueing network in this way, expressing its input stream as a sum of output streams from the environment or other service centres, we obtain what are known as the *traffic equations*. For the simple network shown in Figure 23, the traffic equations are as follows:

$$\begin{aligned} \lambda_1 &= \lambda + q \times \lambda_2 \\ \lambda_2 &= p \times \lambda_1 \end{aligned}$$

We obtain one equation for each service centre, and we have one unknown for each service centre, so we end up with n equations in n unknowns. Therefore we can solve the traffic equations and find the arrival rate at each service centre. In the case above, assume that $\lambda = 10$, $p = 0.5$ and $q = 0.4$. If we substitute $\lambda_2 = 0.5\lambda_1$ into the first equation we obtain

$$\begin{aligned} \lambda_1 &= 10 + 0.2\lambda_1 \\ 0.8\lambda_1 &= 10 \end{aligned}$$

Therefore $\lambda_1 = 12.5$ and substituting back we deduce that $\lambda_2 = 6.25$.

The great advantage of solving traffic equations rather than the global balance equations is that the number of equations we need to solve grows linearly with the number of service centres, rather than exponentially, which is the case for global balance equations.

10.4 Assumptions

Queueing network modelling is inherently a *top-down* process. The underlying philosophy is to begin by identifying the principal components of the system and the ways in which they interact, then supply any details that prove to be necessary. This philosophy inevitably leads to a large number of assumptions being introduced while a modelling study is being conducted.

Many of these assumptions are the general ones which we have been making throughout our study of Markov processes, although since we are avoiding the numerical solution of the global balance equations we *can* now consider models which have an infinite number of states. There are five assumptions which we must make about the behaviour of the model to ensure that a product form solution exists. These are:

Each service centre is **flow balanced**; this means that the number of customers that arrive at each centre is equal to the number of customers who depart.

The system exhibits **one step behaviour**; this means that no two customers in the system "change state" at exactly the same time. Here changing state may mean a customer finishing at one service centre and progressing to the next, or a customer arriving at, or departing from the system.

The system has **routing homogeneity**. This means that the probability that a customer completing service at service centre j now proceeds to service centre k is independent of the current queue lengths at j or k, for all service centres in the network.

The system has **device homogeneity**. The rate of completions of customers from a service centre may vary with the number of jobs at that centre, but otherwise cannot depend on the number or placement of customers within the network.

The system experiences **homogeneous external arrivals**; that is, the times at which arrivals from outside the network occur may not depend on the number or placement of customers within the network.

11 Simulation Models—Introduction and Motivation

So far in the course the stochastic models which we have considered have been solved *analytically*. What this means is that by carrying out analysis of the system we have been able to deduce the steady state behaviour of the corresponding stochastic process, expressed as a probability distribution over the possible states. Such models can be regarded as a *mathematical abstraction* of the system. The analytic model is a representation which can be analysed mathematically to deduce the behaviour of the system.

In contrast, a stochastic simulation model can be regarded as an *algorithmic abstraction* of the system—a simulation model gives a representation which when *executed* reproduces the behaviour of the system.

We still assume that the system is characterised by a family of random variables $\{X(t), t \in T\}$. As the value of time increases, and in response to the "environment" (represented by random variables within the model) the stochastic process progresses from state to state. Any set of instances of $\{X(t), t \in T\}$ can be regarded as a path of a particle moving randomly in a state space, S, its position at time t being $X(t)$. These paths are called *sample paths*. Using the analytic approach of Markov processes we characterised all possible sample paths by the global balance equations. Using simulation we investigate the sample paths directly. In other words we allow the model to trace out a sample path over the state space. Each *run* of the simulation model will generate another, usually distinct, sample path.

There are many reasons why simulation may be preferable to analytic modelling.

Level of Abstraction As we have seen, Markovian modelling relies on many assumptions and abstractions—these may not be appropriate for the system being studied. For example, it may be unrealistic to assume that only one event can happen at any time, or that the inter-event times are all exponentially distributed. Simulation models allow us to represent a system at arbitrary levels of detail, although you should remember that there is always a trade-off between how elaborate the model is and how long it takes to produce a statistically significant run.

Transient Analysis In some cases we are not interested in the steady state behaviour of a system, but in its *transient* behaviour. Some systems never reach a steady state. Even those that do will usually have a "warm-up" period while the behaviour settles into the regular pattern which characterises steady state. The analytic solutions we have considered earlier in the course ignore this period since the global balance equations only capture the behaviour after steady state has been reached. However it is not always possible to extract all the information we want about a system from its steady state behaviour. A sample path derived from a simulation model will clearly represent transient behaviour in addition to steady state[1].

Size of State Space In most cases solving a model analytically involves constructing and storing the complete state space of the model. For example, for a Markov process with N states solving the global balance equations involves (at least) an $N \times N$ matrix (the generator matrix) and a vector with N elements (the steady

[1] For Markov processes transient behaviour can be derived analytically but this generally requires more mathematically sophisticated analysis techniques than those we have considered in this course.

state distribution). When N becomes very large this becomes infeasible. In contrast, in a simulation model the state space is generated "on-the-fly" by the model itself during execution so it does not need to be all stored at once.

11.1 Constructing simulation models

Simulation models are complex computer programs. They can be developed in any programming language but there are distinct advantages to using a language or package specifically designed for simulation. Languages with simulation features built-in, such as SimJava, provide facilities for many of the routine features of a simulation model. These features are common to all models, regardless of the system being represented. This allows the performance analyst to concentrate on the issues specific to the system being modelled and to not worry about issues which are general to all simulations. Using a general language the modeller might be able to write a more efficient program. However, this is likely to take much longer to develop because the program will be responsible for the simulation management, as well as the representational aspects of the model itself.

Some of the common features of simulation management are listed below.

Event scheduler The events in the system change the state and therefore drive the simulation. An event scheduler keeps track of the events which are waiting to happen, usually as a linked list, and allows them to be manipulated in various ways. For example,

- schedule event E at time T;
- hold event E for a time interval ∂t;
- cancel a previously scheduled event E;
- hold event E indefinitely (until it is scheduled by another event);
- schedule an indefinitely held event.

The event scheduler is one of the most frequently executed components of a simulation model. It is called before every event, and it may be called several times during one event to schedule other new events. Therefore it is very important that it is implemented efficiently.

Simulation clock and time management Every simulation model must have a global variable representing the *simulated* time. The event scheduler is usually responsible for advancing this time, either one unit at a time or, more commonly, directly to the time of the next scheduled event. This latter approach is called *event-driven* time management.

System state variables Since a simulation model generates a random walk over the state space of the system it is essential that the model has variables to capture the state of the system at each step. If a simulation run is stopped in the middle, it can be restarted later if, and only if, the values of all state variables are known.

Event routines Each event in the system brings about a state change; thus, in the simulation model the effect of each event must be represented in a way which updates the system state variables, and in some cases, schedules other events. How the event routines are generated will depend on the simulation modelling paradigm used to construct the model.

Random number/random variate generator Random numbers play a crucial role in most discrete event simulations. For example the impact of the environment on the system, e.g. inter-arrival times, is usually represented by random variables of some specified distribution. A random number generator is used to generate a sequence of random values between 0 and 1. These values are then transformed to produce a sequence of random values which satisfy the desired distribution. This second step is sometimes called random variate generation.

Report generator Performance measures are derived from a simulation run by observing the values of parameters of interest during the execution. Most simulation modelling languages and packages contain built-in routines to calculate statistics from these observations and generate a report when the run is completed.

Trace routines A trace of the system can be a useful tool for debugging (sometimes called *verifying*) and validating a model. It is a time-ordered list of events, state variable values or output parameter values. Most simulation languages provide routines to generate traces which can be switched on or off in a particular run of the model. Since trace generation is usually very inefficient it is generally only used during model development.

Dynamic memory management The number of active entities during the execution of a simulation model will vary continuously as new entities are created and old ones become obsolete. Most simulation languages provide automatic garbage collection to remove obsolete entities.

11.2 Simulation modelling paradigms

There are a number of approaches to discrete event simulation; the two most commonly used are *event based modelling* and *process based modelling*. Each of these modelling styles is briefly described below but all the example simulation programs which we will consider in the notes are developed using the process based approach.

Note that the high level modelling paradigms which we have already considered in the course—SPN, GSPN, PEPA and queueing networks—can be used to generate simulation models as well as Markov processes.

11.2.1 Event Based Simulation

Event based simulation focusses the modeller's attention on the individual events which can occur within the system. An event within the system may generate several actions in the model—these are grouped together in an event routine. The event scheduler maintains a pointer to the appropriate event routine, and this is executed when the event reaches the head of the event list.

A good example of a system suited to event based simulation would be a simple queue with deterministic service times and customer defections. The events are a customer arrives, A; a customer defects, D; a customer begins service, B; and a customer ends service, E.

At each event time as well as the processing of the event to represent the behaviour of the system some processing internal to the model might be done. This can be event tracing or statistical collection—whatever has been specified in the model description. For example, if the queue above were being modelled to calculate the average queue length and throughput the events would be implemented as follows.

An event A **at time** t will result in the following actions

- add one to the state variable representing queue length, and record the time at which the change occurred,
- schedule an event D at the time $t + d$, where d is the length of time a customer will wait without defecting,
- schedule an event B to occur as soon as possible, depending on the availability of the server,
- schedule another event A at time $t + a$ where a is the inter-arrival time.

An event D **at time** $t + d$ will

- decrease the queue length by one and the record the time at which the change occurred,
- de-schedule event B on hold since time t.

An event B **at time** $t + w$ $(w < d)$ will

- decrease the queue length by one and record the time at which the change occurred,
- de-schedule the event D at time $t + d$,
- schedule an event E at time $t + w + s$, where s is the service time,

An event E **at time** $t + w + s$ will

- increment the busy time of the service centre by s,
- add one to the total number of customers served,
- activate the first event B waiting in the event list.

11.2.2 Process Based Simulation

The process based approach to simulation modelling collects events together into related sequences which are ordered by time. These sequences are related in the sense that they all involve the same entity within the system; they are termed *processes*. For example, above we could consider each customer to be a process within the system, since it generates a sequence of related events, and track its progress through the queue.

The process based approach views a system as a web of concurrent, interacting entities—the processes. All the actions associated with an entity's behaviour are grouped together to form a life-cycle for entities of that type. The event scheduler still maintains a list of scheduled events centrally but this will now generally be in the form of a pointer to a process/object. The process will maintain a record of its current state and which action it should perform when next scheduled. This style of modelling maps very well onto object-oriented programming where a class is associated with each type of entity; objects belonging to the class then represent instances of the entity.

For the example of a queue with deterministic inter-arrival and service times (but no defections), we would define a class to represent the arrival process. This class (`Source`) would generate the event representing a customer entering the queue and then delay until the arrival time of the next customer. A second class would represent the server (`Server`). It is passive in the sense that it first waits to be notified of an event (the arrival of a customer) and then represents the service of the customer as a delay. This simple deterministic queue represented in SimJava is shown in Figure 24.

In this case each entity class is instantiated just once in the body of the simulation program, which is itself declared as a class `Queue`. We will go into the SimJava language and the facilities it provides in more detail in the next lecture note.

11.3 Common mistakes in simulation studies

Inappropriate level of detail Because simulation modelling allows the modeller to incorporate arbitrary levels of detail it is sometimes tempting to represent too much in the model. This will have a cost in terms of execution time. The best strategy is to start with the simplest possible model and only add detail as it becomes necessary (sensitivity analysis).

Unverified models As already mentioned, simulation models are complex programs and as such are prone to bugs in the same way that any complex program is. Verification is intended to make sure that the model behaves as it was intended to.

Invalid models Validation is intended to make sure that the model is a good representation of the system. A model may behave correctly in the sense that it contains no bugs, but if the assumptions which have been made during its construction are inappropriate the results obtained from the model will still be invalid.

Too short simulation runs Especially when a model has a large state space, the model must be executed for a long (simulated) time to ensure that the sample path which is generated is statistically valid.

Single simulation runs Each run of the simulation represents only one sample path through the state space of the model, corresponding to a particular sequence of random numbers. In order for results to be statistically valid they should be based on several sample paths obtained using different sequences of random numbers.

Poor random-number generators Random number generators are used extensively in simulation models to produce values for random variables modelling the environment or internal working of the system. A poor random-number generator may introduce correlation and/or bias into the value of those random variables.

```
class Source extends Sim_entity {
  private Sim_port enqueue;

  public Source(String name) {
    super(name);
    enqueue = new Sim_port("enqueue");
    add_port(enqueue);
  }

  public void body() {
    while (Sim_system.running()) {
      sim_schedule(enqueue,0.0,0);
      sim_pause(10.0);
    } } }

class Server extends Sim_entity {
  private Sim_port arrival;

  public Server(String name) {
    super(name);
    arrival = new Sim_port("arrival");
    add_port(arrival);
  }

  public void body() {
    Sim_event next = new Sim_event();
    while(Sim_system.running()) {
      sim_wait(next);
      sim_process(8.5);
      sim_completed(next);
    } } }

class Queue {
  public static void main(String args[]) {
    Sim_system.initialise();
    Source customers = new Source("Customers");
    Server service = new Server("Service");
    Sim_system.link_ports("Customers", "enqueue", "Service", "arrival");
    Sim_system.set_termination_condition(Sim_system.EVENTS_COMPLETED,
                                         "Service", 0, 10, false);
    Sim_system.run();
  }
}
```

Figure 24: A simple SimJava program representing the deterministic queue in the process-oriented style

12 SimJava

As discussed in the previous lecture note, process based simulation focuses primarily on entities within the system, rather than events. The model is developed as the series of interactions between these entities. In the language we will consider, interactions are represented by messages; in other languages other mechanisms are used to capture interactions. But in all process based simulations the key elements of any model are the entities and their interactions.

SimJava2[1] is a Java API intended to aid in the development of discrete event simulation models. SimJava2 augments Java with building blocks which are designed to be useful in modelling a wide range of systems. In particular, the package includes a class definition `Sim_entity` and all entities within the system are represented by subclasses of this class. It is assumed that entities each have their own pattern of behaviour which is why the general class `Sim_entity` needs to be specialised for each case.

The SimJava2 web pages (`http://www.dcs.ed.ac.uk/home/simjava`) provide documentation and a tutorial for SimJava2 as well as several example models. You will also find examples on the module web page. In this course you will be expected to be able to understand, modify and write SimJava2 programs, and to generate, understand and interpret the output from such programs. Thus you need to develop a basic familiarity with the language.

12.1 Example: Variations on a Queue

In this section we consider several modified versions of the deterministic queue presented at the end of the previous lecture note. The objective is to introduce various features of SimJava2. In a later lecture note we will consider a more sizeable model. First we go through the deterministic queue in a little more detail, explaining the elements of the `simjava` package.

```
import eduni.simjava.*;

class Source extends Sim_entity {
  private Sim_port enqueue;

  public Source(String name) {
    super(name);
    enqueue = new Sim_port("enqueue");
    add_port(enqueue);
  }

  public void body() {

    while (Sim_system.running()) {
      sim_schedule(enqueue,0.0,0);
```

[1] SimJava2 is the latest release of SimJava which has much improved statistical support. This is the version we will be using in this course.

```
      sim_pause(10.0);
    } } }
```

All SimJava2 models start with the line: `import eduni.simjava.*;` which imports all the classes of the `simjava` package. Each entity in the model is a subclass of the provided class `Sim_entity`; this includes both entities representing "real things" in the system being modelled and modelling entities which are included to control the system. In this simple queueing system there are just two entities: the arrival process which generates customers and the server which handles service requests. In this model the "queue" in the queueing system is modelled implicitly by an internal queue of the simulation.

The arrival process is modelled as a subclass of `Sim_entity` named `Source`. Each subclass of `Sim_entity` must have a constructor which calls the constructor for `Sim_entity`, passing the string which is the identifier for this instance as a parameter. The rest of the constructor is used to set up the attributes (ports, distributions, local parameters) of the entity. In this case there is just a single port which will be used to communicate with the server. As discussed in the last lecture note the arrival of a customer is represented by an event, generating a message from the arrival process to the server. The port `enqueue` is first instantiated and then added to the entity.

The second part of the entity declaration is the `body` method. This is called when the simulation is run, and details the behaviour of the entity. The behaviour of the entity is controlled by a while loop conditioned on a call to the boolean function `Sim_system.running()`. As long as the condition is true the behaviour will be repeated, i.e. another customer will be generated. On each iteration a customer is generated, represented by an event, dispatched from the `enqueue` port. The generation of an event is scheduled (using `sim_schedule`) to occur after *delay* 0.0, and is given a *tag* value 0 in this case. Tags are integer values that can be used to distinguish different types of events. This can be useful if an entity may receive events from several sources and needs to distinguish, for example, the order in which they are handled. The method `sim_pause(10.0)` blocks the entity for 10.0 simulation time units; implicitly this removes the entity from the head of the event list and reschedules it for the current time + 10.0.

```
class Server extends Sim_entity {
  private Sim_port arrival;

  public Server(String name) {
    super(name);
    arrival = new Sim_port("arrival");
    add_port(arrival);
  }

  public void body() {
    Sim_event next = new Sim_event();

    while(Sim_system.running()) {
      sim_wait(next);
      sim_process(8.5);
```

```
    sim_completed(next);
  } } }
```

The server entity is declared analogously. Instead of generating arrival events, it waits for such events: `sim_wait`. In this case we are not interested in any specifics of the event, since all events are customer arrivals, but in general, having obtained the event `next` we can then access its attributes using accessor methods such as `next.get_tag()` and `next.event_time()`. As in the source the server entity has a delay, in this case `sim_process(8.5)` rather than `sim_pause` indicating that the entity is busy during the delay rather than idle. When service is complete `Sim_system` is notified that the processing of the event is finished using the call `sim_completed(next)`.

```
class Queue {
  public static void main(String args[]) {
    Sim_system.initialise();
    Source customers = new Source("Customers");
    Server service = new Server("Service");
    Sim_system.link_ports("Customers", "enqueue", "Service", "arrival");
    Sim_system.set_termination_condition(Sim_system.EVENTS_COMPLETED,
                                           "Service", 0, 10, False);
    Sim_system.run();
  } }
```

The final part of the program, is the actual "model": here entities are instantiated and ports are linked together, controlling which entities can communicate[2]. This takes place in a class declaration; this class declaration shares the name of the file and has a `main` method. First, `Sim_system` object is initialised. This is followed by the entity instantiations. Port connections are uni-directional single links, and are specified by *Sending Object Name, Sending Port Name, Receiving Object Name, Receiving Port Name*. The *termination* or *stopping* condition for the simulation needs to be set[3]. Termination conditions can be defined in terms of

- the number of event completions which have been observed in the model;
- the length of (simulation) time for which the model has been running; or
- the estimated accuracy of the measures being collected (see Lecture 13).

In this case we stipulate that the simulation should run until 10 service events have been completed. Since the simulation engine cannot judge when an event can be judged to be complete the modeller is responsible for marking when events are completed (see the statement `sim_completed(next);` at the end of the body of the `Server` entity). Finally the call `Sim_system.run()` starts the simulation; this method will exit when there are no more events to process. Each entity will cease to generate events when the termination condition is satisfied.

[2]Not strictly true since events can be sent outside the port-based messaging system, for example in animated simulations when you do not want an explicit link in the display.

[3]If this is not set the simulation will run indefinitely until its process is killed by the user or it crashes. When animation is being used it can make sense to leave the control of the termination of the simulation to the user who can make a decision based on what he/she observes on the screen.

```
class Source extends Sim_entity {
  private Sim_port enqueue;
  private Sim_negexp_obj src_hold;

  public Source(String name) {
      .........
    src_hold = new Sim_negexp_obj("iatime", 10.0);
  }

  public void body() {
    while (Sim_system.running()) {
      sim_schedule(enqueue,0.0,0);
      sim_hold(src_hold.sample());
    } } }

class Server extends Sim_entity {
  private Sim_port arrival;
  private Sim_negexp_obj svr_hold;

  public Server(String name) {
      .........
    svr_hold = new Sim_negexp_obj("stime", 8.5);
  }

  public void body() {
    Sim_event next = new Sim_event();
    while(Sim_system.running()) {
      i++; if(i>10) break;
      if (sim_waiting() > 0)
        sim_select(Sim_system.SIM_ANY, next);
      else
        sim_wait(next);
      sim_process(svr_hold.sample());
      sim_completed(next);
    } } }
```

Figure 25: Source and Server declarations for a Markovian queue

12.1.1 Using random variables

Since we are primarily interested in stochastic systems in this course the first generalisation we make of the simple queueing system is to change the interarrival time (the `sim_pause()` at the end of the body of `Source`) and the service time (the `sim_process()` at the end of the body of `Server`). To do this we define appropriate random variables in each constructor. In the source we define an exponentially distributed random variable `src_hold`. This is a private attribute of the entity and is instantiated in the constructor:

```
src_hold = new Sim_negexp_obj("iatime",10.0);
```

The first parameter is an identifier of the random variable, the second is the *mean* of the exponential distribution. Note that the value used to specify the exponential distribution is its *mean*, **not** its *parameter*, e.g. the distribution $F(t) = 1 - e^{rt}$ would have an instantiation like:

```
new Sim_negexp_obj("iatime",1/r)
```

In the modified model, each time the source delays between arrivals instead of the deterministic value 10.0 it takes an exponentially distributed amount of time with mean 10.0. Sampling a value for the random variable is generated by the method call `scr_hold.sample()`. The declaration and use of `svr_hold` in the server is analogous. (see Figure 25.)

Now that the delays are no longer deterministic it cannot be guaranteed that the server is ready to receive each arrival as it is sent from the source. If an event arrives at an entity when it is not ready to handle it, the event is entered into a *deferred queue*. Events can be extracted from the deferred queue using `sim_select(Sim_predicate pred, Sim_event ev)`; predicates are used to select specific events from the queue. A method `sim_waiting()` is provided to test the number of events currently in the deferred queue for that entity. In the body of the server, when the server is ready to begin a new service it now first checks to see if any events are waiting in the deferred queue (`if (sim_waiting() > 0)`) and if there are any selects one of them (`sim_select(Sim_system.SIM_ANY, next);`); alternatively it waits for the next event to be sent to it as in the previous version of the model (`sim_wait(next);`). In fact the construction:

```
if (sim_waiting() > 0)
   sim_select(Sim_system.SIM_ANY, ev);
else sim_wait(ev);
```

appears so often that it is implemented as a method `sim_get_next(ev)`

`Sim_system.SIM_ANY` is a wildcard predicate which matches any event. It is also possible to select just the first event that has a given tag using `sim_select(new Sim_type_p(tag), ev)`. There is a corresponding form of `sim_get_next()` which takes a predicate and a tag as arguments.

12.1.2 Reporting

SimJava2 provides support for statistics gathering and reporting of both default measures and custom measures. Default measures can be regarded as common to all SimJava2 entities and events, whatever is actually being modelled. For default measures the modeller does not need to explicitly tell SimJava2 when measurements need to be taken, only which default measures are of interest. Note, however, that it is up to the modeller to ensure that the default measures do make sense in the context of the chosen entity in their model. The Server entity of the Queue model extended with default measures is shown in Figure 26.

```
class Server extends Sim_entity {
  private Sim_port arrival;
  private Sim_negexp_obj svr_hold;
  private Sim_stat stat;

  public Server(String name) {
    super(name);
    arrival = new Sim_port("arrival");
    add_port(arrival);
    svr_hold = new Sim_negexp_obj("stime", 8.5);
    add_generator(svr_hold);
    stat = new Sim_stat();
    stat.add_measure(Sim_stat.UTILISATION);
    stat.add_measure(Sim_stat.SERVICE_TIME);
    stat.add_measure(Sim_stat.QUEUE_LENGTH);
    stat.calc_proportions(Sim_stat.QUEUE_LENGTH, new double[] {0, 1, 2});
    stat.set_efficient(Sim_stat.SERVICE_TIME);
    stat.set_efficient(Sim_stat.QUEUE_LENGTH);
    set_stat(stat);
  }

  public void body() {
    Sim_event next = new Sim_event();
    while (Sim_system.running()) {
      sim_get_next(next);
      sim_process(svr_hold.sample());
      sim_completed(next);
    }
  }
}
```

Figure 26: Server entity from the `Queue_Stats.java` model

In contrast, custom measures are particular to each model, and depend on the meaning of the entities and events. In this case, as well as declaring the measures, the modeller must explicitly tell SimJava2 when to update them. Both custom and default measures can be recorded in detail or efficiently. Recording a measures efficiently means taht less data is stored and this improves the efficiency of the simulation run. However, there is also much less flexibility about how the measure can subsequently be manipulated and displayed. Further details can be found in the SimJava2 tutorial.

```
class Source extends Sim_entity {
  private Sim_port enqueue;
  private Sim_negexp_obj src_hold;
  private Sim_stat stat;

  public Source(String name) {
    super(name);
    enqueue = new Sim_port("enqueue");
    add_port(enqueue);
    src_hold = new Sim_negexp_obj("iatime", 10.0);
    add_generator(src_hold);
    stat = new Sim_stat();
    stat.add_measure("Generation rate", Sim_stat.RATE_BASED);
    set_stat(stat);
  }

  public void body() {
    while (Sim_system.running()) {
      stat.update("Generation rate", Sim_system.sim_clock());
      sim_schedule(enqueue, 0.0, 0);
      sim_pause(src_hold.sample());
    }
  }
}
```

Figure 27: Source entity from the `Queue_Stats.java` model

All custom measures must be classified as

Rate based measures: these record the occurrences of an event over a period of time and derive a rate for the event accordingly.

State based measures: these reflect the proportion of time an entity may be in a certain state. A state based measure is considered to be *continuous* if all states are of interest, and *non-continuous* if there are some states which can be ignored from a measurement point of view.

Interval based measures: these are concerned with time intervals, usually as experienced by events rather than entities. An example might be the waiting time of an event at an entity.

At the end of each experiment a report file (`sim_report`) is produced with details of the default and custom measures together with general information about the experiment[4], such as the time it took to complete and the conditions used. The report generated by the `Queue_Stats.java` model is shown in Figures 28 and 29 .

[4]How an experiment can be constructed from a number of model runs will be explained in Lecture note 13.

```
############################################################
#                                                          #
#                    SIMULATION REPORT                     #
#                                                          #
############################################################

Version: SimJava 2.0

Simulation date:         November 12, 2002
Simulation start time:   11:25:18 AM GMT
Simulation end time:     11:25:18 AM GMT

############################################################
#             Overall simulation run information           #
############################################################

Total simulated time:    107.52726357025585
Total transient time:    0.0
Total steady state time: 107.52726357025585
Transient condition:     None
Termination condition:   10 event completions at Service
Output analysis method:  None
```

Figure 28: Report generated by the `Queue_Stat.java` model (part 1)

Additionally SimJava2 can also produce graphs which can be viewed by the companion tool, the SimJava2 Graph Viewer (SJGV). This will be discussed in more detail in a later lecture note.

SimJava2 also includes the `simdiag` package which is a collection of JavaBeans based classes intended for displaying timing diagrams and graphs of results. However the graphs are based on data gathered from single simulation runs makes them inappropriate for the type of stochastic modelling we are doing in this course, so we will not consider `simdiag`.

12.1.3 Animation

One of the original motivations for the development of SimJava was the provision of easy facilities to develop animations of simulations based on applets. These facilities are provided by the `simanim` package, which is tightly integrated with the `simjava` package. The model is augmented with special trace commands and information about the visual appearance of entities and ports. The traces have two forms, which have differing results on the appearance of the running model.

- `S` traces are used to display the sending of an event by `sim_schedule()`. They have the standard format:

```
"S <port_name> <event_data_string>"
```

In the animation this will display a blue dot (the standard visual representation of a message), with the event data string written next to it, moving along the link from the port named on the scheduling entity.

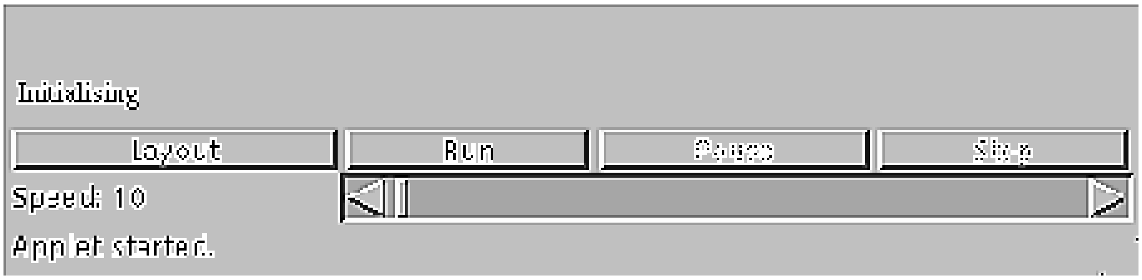
- `P` traces are used to display parameter values associated with entities; these parameters may be dynamically updated as the model evolves. They have the standard format:

  `"P <param1> <param2> ... <paramN>"`

  when the current entity has N parameters. For example the method call `sim_trace(1, "P busy "+3)`, will result in the entities first parameter being updated to value `busy` and the second to value 3. These parameter values may be displayed explicitly in the animation or may result, for example, in a change in the icon used to represent an entity.

`simanim` and `simjava` run concurrently: the trace commands produced by the `simjava` simulation code are fed into `simanim` to produce the animation.

The role of `Sim_system` is now subsumed within a skeleton applet. Two methods of `simanim` create the animation: `anim_init()` is overridden to add buttons, controls and parameter fields, and `anim_layout()` is extended to position the entity icons on the screen and draw ports. To build a simple animated model the modeller gives a `.gif` image and (x,y) coordinates for each entity. Providing additional `.gif` images allows the entity's icon to change according to its internal state, as described above. (see `AppletQ.java` which is a basic animation of the stochastic queue.) `simanim` provides a standard panel for controlling the simulation run:



This allows you to adjust the speed at which the animation runs.

```
############################################################
#              Simulation run measurements                 #
############################################################
------------------------ Customers -------------------------
- Generation rate

Sample mean:          0.10229963671318439
Event count:          11

------------------------- Service --------------------------
- Utilisation

Sample mean:          0.43670097481090003

- Service time

Sample mean:          4.281873568872536
Sample variance:      Not available in efficient mode
Sample std deviation: Not available in efficient mode
Maximum:              8.104271795447396
Minimum:              0.3959786030920327

- Queue length

Sample mean:          0.13959070039740498
Sample variance:      Not available in efficient mode
Sample std deviation: Not available in efficient mode
Maximum:              2.0
Minimum:              0.0
Exceedence proportions:
      0.0 < Queue length <= 1.0 : 0.10242505109839929
      1.0 < Queue length <= 2.0 : 0.018582824649502838
      2.0 < Queue length : 0.0
```

Figure 29: Report generated by the `Queue_Stat.java` model (part 2)

```
class Source extends Sim_entity {
  private Sim_port enqueue;
  private Sim_negexp_obj src_hold;

  public Source(String name, double iatime) {
    super(name, "source", 15, 100);
    enqueue = new Sim_port("enqueue", "port", Anim_port.RIGHT, 30);
    add_port(enqueue);
    add_param(new Anim_param("Count", Anim_param.VALUE, "0", 10,10));
    src_hold = new Sim_negexp_obj("iatime", iatime);
    add_generator(src_hold);
  }

  public void body() {
    int count;

    while (Sim_system.running()) {
      sim_schedule(enqueue, 0.0, 0);
      sim_trace(1,"S enqueue");
      sim_trace(1,"P ‘‘+(++count));
      sim_pause(src_hold.sample());
    }
  }
}
```

Figure 30: The modified `Source` entity to work with animation.

12.2 Compiling and running SimJava2 models

If you want to run SimJava2 on a home machine you can download all the code from the SimJava2 web page (`http://www.dcs.ed.ac.uk/home/simjava`) and follow the instructions which come with the zip file. Please make sure that you download **SimJava2**, not SimJava1.2.

To compile and run a SimJava2 model under DICE you need to add the `simjava` package to your `CLASSPATH` environment variable. This can be done simply by typing:

```
export CLASSPATH=/usr/java/lib/simjava/classes/:$CLASSPATH
```

To compile a model in a file `Model.java` type:

```
javac Model.java
```

This will produce a class file for each class within the model.
The model can then be executed by typing:

```
java Model
```

When you want to run an animation in the appletviewer you need to add a symbolic link to the animation classes in the directory from which you will be running the appletviewer. This can be done by typing:

```
ln -s /usr/java/lib/simjava/classes/eduni eduni
```

As usual, you are strongly encouraged to develop your understanding by experimenting with the example models provided via the module web page.

13 Random Variables and Simulation

In this lecture note we consider the relationship between random variables and simulation models. Random variables play two important roles in simulation models.

We assume that within our models some delays (`pause()` or `process()` in SimJava2) will not have deterministic values, but instead will be represented by random variables; similarly when a choice must be made within the behaviour of an entity we will sometimes want the decision to be made probabilistically. Both cases will involve sampling a probability distribution to extract a value each time this part of the entity's behaviour is reached. For delays we will use continuous random variables (e.g. `Sim_normal_obj` in SimJava2) and for a choice we will use a boolean random variable (via `Sim_random_obj` in SimJava2). As we will see below, both cases rely on the random number generator.

As with the models we have considered earlier in the course, we assume that the variables characterising the behaviour of the system/model, the performance measures or output parameters, are also random variables. For example, a measure might be the number of packets lost per million in a communication network, or the average waiting time for an access to a disk. In general, each run of the simulation model provides a single estimate for these random variables. If we are interested in steady state values the longer we run a simulation the better our estimate will be. However, it still remains a single observation in the sample space. We need more than a single estimate in order to draw conclusions about the system. Thus we use output analysis techniques to improve the quality of an estimate from a single model run, and to develop ways of gaining more observations without excessive computational cost. Realistic simulation models take a long time to run—there is always a trade-off between accuracy of estimates and execution time.

13.1 Random Number Generation

Generating random values for variables with a specified random distribution, such as an exponential or normal distribution, involves two steps. First, a sequence of random numbers distributed uniformly between 0 and 1 is obtained. Then the sequence is transformed to produce a sequence of random values which satisfy the desired distribution. This second step is sometimes called *random variate generation*.

To obtain a sequence of uniform random numbers, it is sufficient to be able to generate a sequence X_k of integers in the range $[0, M-1]$ since the sequence $X_k/(M-1)$ will then be approximately uniformly distributed over $(0, 1)$. In 1951, D.H. Lehmer discovered that the residues of successive powers of a number have good randomness properties. He obtained the kth number in the sequence by dividing the kth power of an integer a by another integer M and taking the remainder.

$$X_k = a^k \bmod M$$

This can be expressed as an iteration:

$$X_k = (a \times X_{k-1}) \bmod M$$

The parameters a and M are called the *multiplier* and the *modulus* respectively. Random number generators of this form are called *Lehmer generators*, or *multiplicative linear-congruential generators*.

There are several desirable properties for a random number generator:

It should be efficiently computable. Since simulations typically require several thousand random numbers in each run, the processor time required to generate these numbers should be small.

It should be pseudo-random. Given the same seed the random number generator should produce exactly the same sequence of numbers. Without this property it would not be possible to recreate experiments.

The cycle should be long. A short cycle may cause the random number sequence to recycle, resulting in a repeated event sequence. This may limit the useful length of simulation runs.

Successive values should be independent and uniformly distributed. The correlation between successive numbers should be small. Correlation, if significant, indicates dependence.

Research has shown that Lehmer generators obey these properties provided a and M are carefully chosen. However care is needed. In the early 1970s most university mainframes were using a linear-congruence generator known as `RANDU`. It used the values $a = 65539$ and $M = 2^{31}$. Although the output looked random, detailed statistical analysis showed that there was significant correlation in the output.

Nevertheless this form of generator continues to be used, if somewhat more warily. These generators are particularly efficient if M is chosen to be a power of 2. In this case finding the residue amounts to simply truncating the result of the multiplication. However a modulus of the form 2^k results in a shorter cycle: 2^{k-2} at best.

Care must be taken in the implementation of random number generators. For example, properties such as the length of cycle are only maintained if all computations are done precisely without round-off errors. Thus the computations should use only integer arithmetic. Similarly, care must be taken to ensure that the multiplication $a \times X_{k-1}$ does not cause overflow—intermediate calculations may need to be carried out using extended precision arithmetic operations.

The random number generator used in SimJava2 is a Lehmer generator with parameters: $a = 742,938,285$ and $M = 2^{31} - 1$. This has been demonstrated to have good statistical properties.

Generation algorithms for values of the commonly used probability distributions, based on a uniformly distributed stream of values between 0 and 1, can be found in many books on simulation and performance modelling[1]. Inverse transformation algorithms are based on the observation that for any probability distribution with distribution function $F(x)$, the value of $F(x)$ is uniformly distributed between 0 and 1. Thus, using values from the random number stream, $u = X_k$, the function is inverted to find the next value of x: $x = F^{-1}(u)$. For example, given a random number u, we generate the next value in an

[1] For example, *The Art of Computer Systems Performance Analysis*, R. Jain, Wiley, 1991.

exponential distribution with parameter λ as[2]

$$x = -\frac{1}{\lambda}\ln(u)$$

Boolean valued distributions which are used to make decisions within a model take a single real parameter, p, such that $0 \leq p \leq 1$. This represents the probability of a "positive" outcome. Then each time the branching point in the model is reached, the next random number in the stream is generated $u = X_k$. If $u \leq p$ the positive branch is taken; if $u > p$ the other branch is selected.

Similarly, for discrete probability distributions a technique known as *aliasing* is used. Here a cumulative distribution function is calculated based on the probability mass function, once some order is imposed on possible outcomes. Random values between 0 and 1 are then compared with the cumulative function, and the appropriate outcome selected based on the interval into which the value falls.

One of the benefits of using a simulation package is that at least some of these algorithms are provided for us. SimJava2 provides classes for many commonly used continuous probability distributions: e.g. `Sim_normal_obj`, `Sim_negexp_obj`, and `Sim_uniform_obj` and discrete probability distributions: e.g. `Sim_bernoulli_obj`, `Sim_binomial_obj`, and `Sim_poisson_obj`. Each time such an object is instantiated the seed for the random number generator can be set explicitly. However if these seeds are not *well-spaced* there may be overlap between the sequences of random numbers used by the generators resulting in correlation between the samples used in the simulation. Consequently SimJava2 also provides an automatic seeding mechanism which will seed each distribution with a distinct seed which is far in the cycle from other seeds currently in use. This avoids the problem of overlap and correlation.

13.2 Simulation output analysis

In performance modelling our objective in constructing a simulation model of a system is to generate one or more performance measures for the system. In the Markov models we considered earlier in the course such measures were derived from the steady state probability distribution, after the model had been solved. In contrast, in a simulation model measures are evaluated directly during the execution of the model. It is part of model construction to make sure that all the necessary counters and updates are in place to allow the measures to be collected as the model runs. This is sometimes called *instrumentation* of a model as it is analogous to inserting probes and monitors on a real system.

However the data from a simulation model is generated it is important to remember that each run of a model constitutes a single trajectory over the state space. Consequently, in general, any estimate for the value of a performance measure generated from a single run constitutes a single observation in the possible sample space. To gain an accurate measure of the performance of the system we should not base our results on a single observation.

[2]Strictly speaking, the equation should be $x = -1/\lambda \ln(1-u)$ but since u is uniformly distributed between 0 and 1, $1-u$ will be uniformly distributed between 0 and 1 and the generation algorithm can be simplified.

When carrying out steady state analysis we should bear in mind that the averages we calculate from data collected during execution will always be an approximation of the unknown true long-term averages that characterise the system performance. Important issues are:

- choosing the starting state of the simulation;
- choosing the *warm-up* period that is allowed to elapse before data collection begins;
- choosing a run length that ensures that the calculated averages are representative of the unknown true long term average.

Statistical techniques can be used to assess how and when the calculated averages approximate the true average, i.e. to analyse the accuracy of our current estimate. This is often done in terms of a *confidence interval*. A confidence interval expresses probabilistic bounds on the error of our current estimate. So a confidence interval (c_1, c_2) with *confidence level* $X\%$, means that with probability $X/100$ the real value v lies between the values c_1 and c_2, i.e.

$$\Pr(c_1 \leq v \leq c_2) = X/100$$

$X/100$ is usually written in the form $1 - \alpha$, and α is called the *significance level*, and $(1 - \alpha)$ is called the *confidence coefficient*. Usually performance modellers will run their simulation models until their observations give them confidence levels of 90% or 95% and a confidence interval which is acceptably tight. Calculation of the confidence interval is based on the variance within the observations which have been gathered. The greater the variance the wider the confidence interval; the smaller the variance the tighter the bounds.

In SimJava2 the confidence level can be used to control the period over which a model is run or the number of times it is run for, as we will see later in this note.

13.2.1 Steady state measures

For some modelling studies the length of time for which a simulation model should be run is defined by the problem itself. For example, if we wish to investigate how many messages can be processed by a dealers' transaction processing system *in the first hour of trading* then it makes sense to run the model for 3600 seconds. However, if the question is how many messages can be processed by the dealers' transaction processing system *in an average hour* then running the model for 3600 seconds is unlikely to be enough.

The first question identifies the simulation as a *transient* or *terminating* simulation. It is said to have a *cold-start*: the system is initially empty which is not its usual state but we still include this data in the observation period. For this type of simulation the question becomes how many times the simulation must be repeated (with different random number streams) to achieve a required confidence interval.

In the second scenario we are interested in the *steady state* behaviour of the system. As in Markovian modelling we associate steady state behaviour with long term behaviour. In other words we are theoretically interested in the observations obtained from runs of the model which are infinitely long. However, in practice we are interested in finite run lengths and estimating the steady state distribution of the measures we are interested in from finitely many samples.

The initial conditions of the model, its *starting state*, influence the sequence of states through which the simulation will pass, especially near the start of a run. In a steady state distribution the output values should be independent of the starting state. Thus the modeller must make some effort to remove the effect of the starting state, sometimes termed *bias*, from the sample data used for estimating the performance measure of interest. Unfortunately it is not possible to define exactly when the model has moved from transient behaviour to steady state behaviour. This initial period before steady state is reached is sometimes called the *warm-up period*. Therefore, although several techniques exist, they are all heuristics. The common techniques are

1. Long runs.
2. Proper initialisation.
3. Truncation.
4. Initial data deletion.
5. Moving average of independent replications.
6. Batch means.

The last four techniques are all based on the assumption that variability is less during steady state behaviour than during transient behaviour.

In SimJava2 the end of warm-up period can be set using a condition: steady state measures will only be calculated based on data collected after the condition became true. The method `Sim_system.set_transient_condition()` is used to set the condition to be used. There are three possibilities:

- The condition is expressed in terms of event completions based on an event tag at a given entity, e.g. `Sim_system.set_transient_condition(Sim_system.EVENTS_COMPLETED, "Server", 0, 5);` in the `AppletQ_Stats` example marks the end of the warm-up period as when 5 jobs have been served at the server.

- An estimate the length of the warm-up period is used to express the condition as a simulation time, e.g. `Sim_system.set_transient_condition(Sim_system.TIME_ELAPSED, 50000);` in `Queue_Other` SimJava2 will start collecting data when 50000 time units have elapsed.

- No condition is set by the modeller but after the completion of a run SimJava2 applies the minimum-maximum truncation method to identify the end of the transient period. Note however that this is a crude technique which generally under-estimates the length of the warm-up period.

13.2.2 Termination Conditions

How the simulation run is terminated can also have an effect on the accuracy of the data collected. There are two generally used options for choosing when to stop the simulation of a performance model:

Option 1

- begin the simulation at time 0
- begin data collection at specified time $w \geq 0$
- complete data collection at specified time $w + t$
- terminate execution of the simulation at time $w + t$
- calculate summary statistics based on sample path data collected in the time interval $(w, w + t)$.

Option 2

- begin the simulation at time 0
- begin data collection when the Mth job completes all service
- complete data collection when the $(M + N)$th job completes all service
- terminate execution of the simulation when the $(M + N)$th job completes all service
- calculate summary statistics based on sample path data collected in the time interval (t_M, t_{M+N}), where t_j is the time at which the jth job completes all service.

Option 1 implies that the simulated time $(w, w + t)$ for data collection is predetermined but the number of job completions is random. Conversely, Option 2 implies that the time period for data collection is random but the number of job completions is predetermined. Option 1 is preferable for calculating queue lengths and resource utilisations, whereas Option 2 is preferable for calculating waiting times.

Both options are supported in SimJava2, and as with the transient condition, the termination condition can be based on either event completions or elapsed time. For example in the example model `Queue.java` the simulation is set to terminate when 10 jobs have been processed at the Server:

```
Sim_system.set_termination_condition(Sim_system.EVENTS_COMPLETED,
                                                  "Service", 0, 10, false);
```

A third option is to use confidence interval accuracy as the controlling factor.

13.2.3 Variance reduction techniques

Assume that we are running a simulation model in order to estimate some performance measure M. During the ith execution of the model we make observations of M, o_{ij} and at the end of the run we calculate the mean value of the observations O_i. Note that the observations o_{ij} in most simulations are *not* independent. Successive observations are often correlated. For example, if we are interested in the delay of messages in a packet-switching network, if the delay of one message is long because the network is heavily congested, the next message is likely to be similarly delayed. Thus the two observations are not independent. This is why, in general, a simulation model must be run several times.

Independent Replications If independent replications are used the model is run m times in order to generate m independent observations. In order for the runs to be independent the seeds used for the random number generator must be carefully chosen to ensure that they are independent.

If transient or short term behaviour is being investigated all the data collected during the ith run will be used to calculate the ith observation, i.e. O_i is the mean value over all o_{ij}.

If steady state or long term behaviour is being investigated the data relating to the warm-up period must be discarded. The observation for the run, O_i will be the mean value over o_{ij} such that $j > k$, where $\{o_{i1}, \ldots o_{ik}\}$ are the observations made during the warm-up period.

In either case let O denote the mean value of the observations, O_i, after m runs. Then the variance over all observations is calculated as shown below:

$$V = \frac{1}{m-1} \sum_{i=1}^{m} (O_i - O)^2$$

For steady state analysis independent replication is an inefficient way to generate samples, since for each sample point, O_i, k observations, $\{o_{i1}, \ldots, o_{ik}\}$, must be discarded.

Batch Means In the method of batch means the model is run only once but for a considerable period. The run is divided into a series of sub-periods of length ℓ, and measures are collected over each sub-run to form a single point estimate. If the observations made during the run form a set $\{o_i\}$, the set is partitioned into subsets

$$S_i = \{o_j \mid o_j \text{ observed between } (i-1) \times \ell \text{ and } i \times \ell\}$$

Now each sample point O_i is the mean generated from a subset of observations S_i, and O is the mean generated from the O_i. Variance is calculated as above.

This method is unreliable since the sub-periods are clearly not independent. However it has the advantage that only one set of observations $\{o_i \ldots o_k\}$ needs to be discarded to overcome the warm-up effects in steady state analysis.

Regeneration It is sometimes possible within the run of a simulation model to identify points in the trajectory where the model returns to exactly equivalent states. These are called *regeneration points*. Periods between regeneration points are genuinely independent sub-runs. The simplest example to consider is the state when a queue becomes empty. Whatever the distribution used to generate the behaviour of the model the trajectory (queue length, waiting time etc) after a visit to this state does not depend on the previous history of the model in any way. The duration between two successive regeneration points is called a *regeneration cycle*.

The variance computation using regeneration cycles is a bit more complex than that in the method of batch means or the method of independent replications. This is because the regeneration cycles are of different lengths, whereas in the other two methods the batches or replications are all of the same length. Suppose that we divide a sample into m cycles of length $n_1, \ldots, n_m$ respectively. The mean for each cycle, generating a single observation sample can be calculated as expected: $O_i = \frac{1}{n_i}\sum_{j=1}^{n_i} o_{ij}$. However the overall mean O, is not the arithmetic mean of the O_i. Instead we use a different approach to calculate O:

$$O = \frac{\sum_{i=1}^{m}\sum_{j=1}^{n_i} o_{ij}}{\sum_{i=1}^{m} n_i}$$

Variance of the summed observations is then calculated as shown below:

$$V_s = \frac{1}{m-1}\sum_{i=1}^{m}\left(n_i(O_i - O)\right)^2$$

Notice that, unlike the previous two methods, the method of regeneration does not require any transient observations to be removed. Unfortunately not all models have easily defined regeneration states, and even when they exist they can be computationally expensive to identify. Another disadvantage is that it is not possible to define the length of a simulation run beforehand. However, research suggests that this method gives the most accurate results and so it is the method of choice when it is feasible.

SimJava2 supports both independent replications and batch means and use of these can be specified via the `Sim_system.set_output_analysis()` method. If the method is not used no output analysis will be performed. The method call allows the modeller to set the technique to be used and its parameters, such as the number of replications and the desired confidence level for confidence intervals. The output analysis is carried out automatically. For example, a call

```
Sim_system.set_output_analysis(Sim_system.IND_REPLICATIONS, 5, 0.95);
```

will run the model for 5 independent replications and confidence level 95%. At the completion of each run the model will be reset to its initial starting state and all generators will be re-seeded. If an entity contains mutable objects or non-final static fields the modeller must include code in the `body()` method which will reset them as this will not be handled automatically (see the SimJava2 tutorial for more details).

Variance reduction can also be used as a termination condition. For example in the model `Queue_Variance_Reduction.java` a confidence level is set at 95% and batch means analysis is used to control the length of the run until this confidence level is achieved.

```
Sim_system.set_transient_condition(Sim_system.TIME_ELAPSED, 50000);
Sim_system.set_termination_condition(Sim_system.INTERVAL_ACCURACY,
      Sim_system.BATCH_MEANS, 0.95, 0.1, "Service", Sim_stat.UTILISATION);
```

14 Model Validation and Verification

14.1 Introduction

Whatever modelling paradigm or solution technique is being used, the performance measures extracted from a model will only have some bearing on the real system represented if the model is a *good* representation of the system. Of course, what constitutes a *good* model is subjective, but from a performance modelling point of view our criteria for judging the goodness of models will be based on how accurately measures extracted from the model correspond to the measures which would be obtained from the represented system.

By its nature a model is more abstract than the system it represents (even a simulation model). Viewed in one way, abstraction, and assumptions we make to achieve it, eliminate unnecessary detail and allow us to focus on the elements within the system which are important from a performance point of view; viewed in another way, this abstraction process introduces inaccuracy. Some degree of inaccuracy may be necessary, desirable even, to make the model solution tractable and/or efficient. Inevitably some assumptions must be made about the system in order to construct the model. However, having made such assumptions we must expect to put some effort into answering questions about the goodness of our model. There are two steps to judging how good a model is with respect to the system. We must ascertain whether the model implements the assumptions correctly (model verification) and whether the assumptions which have been made are reasonable with respect to the real system (model validation).

We have already seen examples of both model verification and model validation in models which we have considered earlier in the course. For example, in the Markov process model of the PC-LAN we carried out a walk-through of the simplified model with just two nodes in the network to check the characterisation of state which we were using. This was model verification. The level of abstraction which we had first chosen did not allow us to represent the behaviour of the system and make the assumptions about memoryless behaviour. The walk-through allowed us to detect that we needed to distinguish states more finely, leading to a modification of the model. The SPNP modelling package allows us to include assertions within the program representing a model. These assertions are used to encode invariants about the behaviour of the system which we know should be true at all times. For example, in the reader-writer system we were able to assert that it should never be the case that a reader and a writer had access to the database at the same time. This can be used for both model verification and model validation.

It is important to remember that validation does not imply verification, nor verification imply validation. However, in practice, validation is often blended with verification, especially when measurement data is available for the system being modelled. If a comparison of system measurements and model results suggests that the results produced by the model are close to those obtained from the system, then the implemented model is assumed to be both a verified implementation of the assumptions and a valid representation of the system.

14.2 Model Verification

Verification is like debugging—it is intended to ensure that the model does what it is intended to do. Models, especially simulation models, are often large computer programs. Therefore all techniques that can help develop, debug or maintain large computer programs are also useful for models. For example, many authors advocate modularity and top-down design. Since these are general software engineering techniques we will not discuss them in detail here; modifications of such techniques to make them suitable for modelling, and modelling-specific techniques are discussed below.

Antibugging Antibugging consists of including additional checks and outputs in a model that may be used to capture bugs if they exist. These are features of the model which do not have a role in representing the system, or even necessarily in calculating performance measures. Their only role is to check the behaviour of the model.

A common form of antibugging is to maintain counters within a simulation model which keep track of the number of entities which are generated and terminated during the evolution of the model. For example, in a communication network simulation we might keep track of the number of messages entering the network, the number successfully delivered and the number lost. Then, at any time, the number of messages which have entered should be equal to the number which have been delivered plus the number that have been lost plus the number which are currently in transit.

In Markovian models it is not possible to maintain counters in this explicit way as it would break the irreducibility assumption about the state space. However, it is still sometimes possible to include antibugging structures within a Markovian model. For example, an extra place could be included in a GSPN model into which a token was inserted for each entity created in the model and a token was deleted for each entity deleted. Care should be exercised in applying such techniques to ensure that you are not creating *state space explosion*. The SPNP assertions can also be used for antibugging since they allow a check to be made in each state of the model, as the state space is constructed.

Structured walk-through/one-step analysis Explaining the model to another person, or group of people, can make the modeller focus on different aspects of the model and therefore discover problems with its current implementation. Even if the listeners do not understand the details of the model, or the system, the developer may become aware of bugs simply by studying the model carefully and trying to explain how it works. Preparing documentation for a model can have a similar effect by making the modeller look at the model from a different perspective.

In the absence of a willing audience the model developer should try to carry out the same sort of step-by-step analysis of the model to convince himself or herself that it behaves correctly. For a GSPN model this would amount to playing the token game; in a queueing network, stepping through the possible customer transitions. Some modelling packages provide support for doing this.

Simplified models It is sometimes possible to reduce the model to its minimal possible behaviour. We have already seen examples of this when we considered the "multiprocessor" example with only one processor, to make sure that the interaction between the processor and the common memory was correct, and when we considered the PC-LAN with just two PCs. Similarly in a closed queueing network model we might consider the model with only a single customer, or in a simulation model we might only instantiate one entity of each type. Since one-step analysis can be extremely time consuming it is often applied to a simplified model.

Of course, a model that works for simple cases is not guaranteed to work for more complex cases; on the other hand, a model which does not work for simple cases will certainly not work for more complex ones.

Deterministic models (simulation only) For simulation models the presence of random variables can make it hard for the modeller to reason about the behaviour of a model and check that it is as expected or required. Replacing random variables which govern delays or scheduling with deterministic values may help the modeller to see whether the model is behaving correctly. Only when we are satisfied that the behavioural representation of the entities is indeed correct should we introduce random variables to represent inter-event times using continuous time distributions.

Note that this technique is only appropriate for simulation models—Markovian models can only be solved with exponential distributions.

Tracing (simulation only) Trace outputs can be extremely useful in isolating incorrect behaviour in a model, although in general other techniques will be used to identify the presence of a bug in the first place. Since tracing causes considerable additional processing overhead it should be used sparingly in all except the simplest models.

SimJava2 allows you to trace models in terms of specific events or entities, as well as a default trace which will can be produced automatically. Default tracing can be switched on and off using the method `Sim_system.set_auto_trace()` which takes a boolean parameter. The entity trace is produced when an entity includes `sim_trace` commands. The first parameter sets a notional level for the trace message and the level of tracing for any particular run is set in the model body using the method `Sim_system.set_trace_level(l)` where `l` is an integer used as a bit mask with respect to entity trace messages.

Event traces are produced using the method `Sim_system.track_event(s)()`. If a single integer is passed as an argument events with that tag value are traced; if an array of integers is passed (`track_events`) events of all types listed will be traced.

The method `Sim_system.set_trace_detail(b1, b2, b3)` which takes 3 boolean parameters, lets the modeller choose what form of tracing to have in a particular run. For example `Sim_system.set_trace_detail(false, true, false)` switches off the default trace, switches on entity tracing and switches off event tracing. Note that tracing is very expensive in terms of simulation time and the default is to have no tracing if none of the tracing specific methods of `Sim_system` are called. The model fragments in Figure 31 show a variant on the M/M/1 model with user defined tracing.

Animation (simulation only) From a verification perspective, animation is similar to tracing but provides the information about the internal behaviour of the model in a graphical form. Some modelling packages with graphical interfaces, like SimJava, provide a dynamic display of model behaviour whilst the model is executing. In some systems the display will represent high level information about the current value of the performance measures of interest shown as dials or meters which change as the values change. The SimJava animation view is rather low level, showing events moving around the system capturing the interactions between entities. This low level view is particularly suited to verification. Animation can take the form of automated one-step analysis, if the animation facilities allow the view of the model to advance one event at a time. Graphical stochastic Petri net and queueing network tools often provide a animated form of one-step analysis in which tokens or customers can be seen moving around the network.

Handling the display as well as the evolution of the model slows down the simulation considerably. As with tracing, animation is most useful for isolating an error once its presence has been established.

Seed independence (simulation only) The seeds used for random number generation in a simulation model should not significantly affect the final conclusion drawn from a model, although there will be variation in sample points as seeds vary. If a model produces widely varying results for different seed values it indicates that there is something wrong within the model. Seed independence can be verified by running the simulation with different seed values, something which is probably necessary in any case. In SimJava2 the root seed value to be used in a run can be set explicitly; for example calling the method `Sim_system.set_seed(23533)` sets the root seed to 23533. `Sim_system` will still give each separate distribution used in the simulation a distinct well-spaced seed.

Continuity testing At an abstract level all systems and models can be thought of as generating a function from input values to output values, and in most cases we expect that function to be continuous. In other words, in most cases, we do not anticipate that a slight change in an input value will result in very large changes in the corresponding output value.

Continuity testing consists of running a simulation model, or solving a Markovian model, several times for slightly different values of input parameters. For any one parameter, a slight change in input should generally produce only a slight change in the output. Any sudden changes in the output are taken to be an indication of a possible error which should be investigated unless this is known behaviour of the system.

Degeneracy testing Degenerate cases for a model are those values of input parameters which are at the extremes of the model's intended range of representation. Degeneracy testing consists of checking that the model works for the extreme values of system and workload (input) parameters. Although extreme cases may not represent typical cases, degeneracy testing can help the modeller to find bugs that would not otherwise have been discovered.

Consistency testing For most models and systems it is reasonable to assume that similarly loaded systems will exhibit similar characteristics, even if the arrangement of the workload varies. Consistency tests are used to check that a model produces similar results for input parameter values that have similar effects. For example, in a communication network, two sources with an arrival rate of 100 packets per second each should cause approximately the same level of traffic in the network as four sources with arrival rate of 50 packets per second each. If the model output shows a significant difference, either it should be possible to explain the difference from more detailed knowledge of the system, or the possibility of a modelling error should be investigated.

14.3 Model Validation

Validation is the task of demonstrating that the model is a reasonable representation of the actual system: that it reproduces system behaviour with enough fidelity to satisfy analysis objectives. Whereas model verification techniques are general the approach taken to model validation is likely to be much more specific to the model, and system, in question. Indeed, just as model development will be influenced by the objectives of the performance study, so will model validation be. A model is usually developed to analyse a particular problem and may therefore represent different parts of the system at different levels of abstraction. As a result, the model may have different levels of validity for different parts of the system across the full spectrum of system behaviour.

For most models there are three separate aspects which should be considered during model validation:

- assumptions
- input parameter values and distributions
- output values and conclusions.

However, in practice it may be difficult to achieve such a full validation of the model, especially if the system being modelled does not yet exist. In general, initial validation attempts will concentrate on the output of the model, and only if that validation suggests a problem will more detailed validation be undertaken.

Broadly speaking there are three approaches to model validation and any combination of them may be applied as appropriate to the different aspects of a particular model. These approaches are:

- expert intuition
- real system measurements
- theoretical results/analysis.

In addition, as suggested above, ad hoc validation techniques may be established for a particular model and system.

Expert intuition Essentially using expert intuition to validate a model is similar to the use of one-step analysis during model verification. Here, however, the examination of the model should ideally be led by someone other than the modeller, an "expert" with respect to the system, rather than with respect to the model. This might be the system designer, service engineers or marketing staff, depending on the stage of the system within its life-cycle.

Careful inspection of the model output, and model behaviour, will be assisted by one-step analysis, tracing and animation, in the case of simulation models, and the full steady state representation of the state space in the case of Markovian models. In either case, a model may be *fully instrumented*, meaning that every possible performance measure is extracted from the model for validation purposes regardless of the objectives of the performance study.

Real system measurements Comparison with a real system is the most reliable and preferred way to validate a simulation model. In practice, however, this is often infeasible either because the real system does not exist or because the measurements would be too expensive to carry out. Assumptions, input values, output values, workloads, configurations and system behaviour should all be compared with those observed in the real world. In the case of simulation models, when full measurement data is available it may be possible to use *trace-driven* simulation to observe the model under exactly the same conditions as the real system.

Theoretical results/analysis In the case of detailed Markovian models or simulation models it is sometimes possible to use a more abstract representation of the system to provide a crude validation of the model. In particular, if the results of an operational analysis, based on the operational laws coincide with model output it may be taken as evidence that the model behaves correctly.

Another possible use for the operational laws is to check consistency within a set of results extracted from a simulation model. If a model is behaving correctly we would expect the measures extracted during the evolution of a model to obey the operational laws provided the usual assumptions hold. Failure of the operational laws would suggest that further investigation into the detailed behaviour of the model was necessary. For example, the general residence time law can provide us with a simple validation of the model output values obtained for residence times at individual components if we know their respective visit counts, jobs behave homogeneously and we expect the model to be job flow balanced.

At slightly more detail a simulation model may also be validated by comparing its output with a simple queueing network model of the same system, and conversely, (in academic work) Markovian models are often validated by comparing their outcome with that of a more detailed simulation model.

Validation of models against the results or behaviour of other models is a technique which should be used with care as both may be invalid in the sense that they both may not represent the behaviour of the real system accurately.

Another analytic approach is to determine invariants which must hold in every state of the system. For example, these invariants might capture a mutual exclusion condition or a conservation of work condition. Showing that the model always satisfies such an invariant is one way of increasing confidence in the model, and providing support for its validity. The assertions in SPNP do this, and we could imagine including similar checks within an entity in a SimJava model. The disadvantage of such an approach is that it can be computationally expensive to carry out the necessary checks regularly within a model.

```
class Source extends Sim_entity {
  ............
  public Source(String name) {
   ...........
  }

  public void body() {
    int i = 0;
    while (Sim_system.running()) {
      sim_trace(1, "about to generate arrival " + (i++));
      sim_schedule(enqueue, 0.0, 0);
      sim_pause(src_hold.sample());
}  }  }

class Server extends Sim_entity {
  ............
  public Server(String name) {
   ...........
  }

  public void body() {
    Sim_event next = new Sim_event();
    int i = 0;
    while (Sim_system.running()) {
      if (sim_waiting() > 0) {
        sim_select(Sim_system.SIM_ANY, next);
      } else {
        sim_wait(next);
      }
      sim_process(svr_hold.sample());
      sim_trace(2, "completed service of arrival " + (i++));
      sim_completed(next);
}  }  }

class Queue_Trace {
  public static void main(String args[]) {
    Sim_system.initialise();
    ............
    Sim_system.set_auto_trace(true);
    Sim_system.set_trace_level(7);
    Sim_system.track_event(0);
    Sim_system.set_trace_detail(false, true, true);
    Sim_system.run();
  }
}
```

Figure 31: SimJava model demonstrating user-defined tracing (`Queue_Trace.java`)

15 The PC LAN as a SimJava2 Model

15.1 Introduction

In this note we consider again the PC LAN first presented in lecture note 4 as a Markov process, and represented both as a GSPN and a PEPA model in subsequent lecture notes. Using simulation we are able to make the model somewhat more realistic in several respects:

- inter-event times within the model are no longer restricted to be exponentially distributed;
- since the size of the state space does not concern us quite so much we can allow more realistic buffer sizes;
- we can study the transient behaviour of the model (from "cold start" when the system is empty) just as easily as steady state measures;
- moreover, in steady state we are not restricted to considering the means but can also find, for example, the maximum and minimum value of a random variable.

Just as when we were deriving the Markovian model, we might not necessarily arrive at the correct characterisation of the system on our first attempt. However, it is often harder to establish confidence in a simulation model than a Markov process model, requiring the use of specific validation and verification techniques as described in lecture note 14.

15.1.1 Description of the system

Initially, as in lecture note 4, we aim to determine the mean waiting time for data packets at a PC connected to a local area network, operating as a token ring. We relax some of the assumptions that we made in the earlier model, but retain the restriction that a PC can transmit only when in possession of the token, and that the token circulates in *round robin* order. We now assume that whilst the inter-arrival time of packets is still exponentially distributed, packet transmission time is normally distributed and token circulation time is deterministic. In the model presented here we still assume that the PCs connected to the LAN all have the same characteristics. However, if you study the model you will see that it would be straightforward to parameterise the entity definitions in such a way as to differentiate between them.

Without the state space restrictions imposed by numerical solution of the Markov process we no longer impose that at most one packet is waiting to be transmitted from each PC. As previously though, we assume that packets that arrive when the token is already at the node must wait for the next visit of the token. Various policies are used in practice. Our assumption represents what is termed *gated service*: this means that when the token arrives at the node a notional gate is closed. The token will remain at the node long enough to transmit all those packets in front of the gate (i.e. those packets that were waiting when the token arrived) but it will leave those behind the gate (i.e. those packets that arrived while the token was at the node) until the next visit.

15.2 Modelling the system

The first task is to identify the entities to be represented in the SimJava2 model, and then the events which constitute interactions between them. An initial consideration of the system description would perhaps identify the following potential entities:

- PCs,
- Buffers,
- LAN,
- Token.

In this model some of these entities are represented implicitly. We distinguish between the PC, whose role is to generate packets, but which is otherwise independent of the network, and the "Node" which holds waiting packets, carries out the transmission and passes on/receives the token. In the Markov process we did not need to distinguish these entities but now it is necessary for two reasons. Firstly, we need to record the number of packets waiting at a node: using two distinct entities we can use the deferred queue of the node as an implicit representation of the buffer. Secondly, we wish to model concurrent packet generation and transmission. When Markovian assumptions are made we could use an interleaving approach due to the memoryless property of the exponential distribution. Since each entity has its own (single) thread of control, concurrent activity must generally be represented in distinct entities[1].

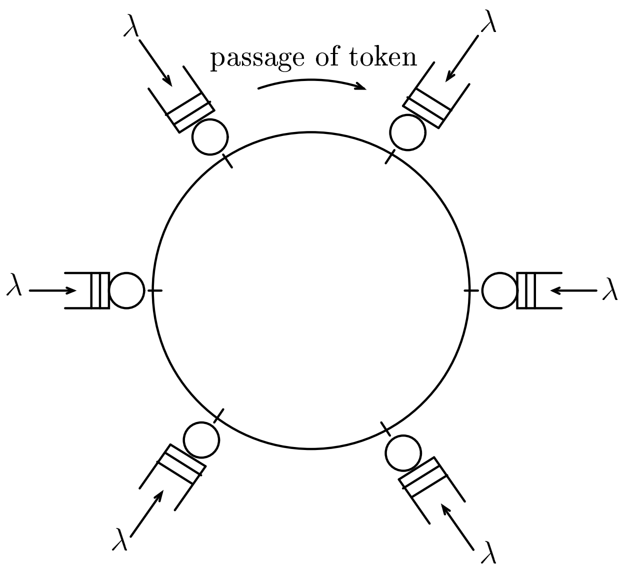


Figure 32: A six-node token ring network

The LAN is modelled only implicitly as the collection of nodes and the links between them. Once we have defined the PCs and the Nodes there is no behaviour left to be represented which indicates that having an entity to represent the LAN itself is unnecessary.

[1]The exception is that scheduling an event with a delayed time will allow the entity to concurrently proceed with its next activity.

In general, we aim to keep the number of entities within a model to a minimum. This aids both the understandability of our models and the execution times: there is significant overhead in switching between threads.

The token is modelled as an event or message rather than as an entity. In the GSPN and PEPA models of the LAN, the token was represented more explicitly and participated in a synchronised transition/activity. In SimJava (and discrete event simulation packages generally) it is difficult to represent shared activity so directly. If two entities undertake joint work, in the model it is represented (as a delay) in only one of the entities, the other being passivated until the work is complete. In SimJava this would be represented by the passive entity first sending a message to say it was ready to proceed with the shared work and then waiting for a message from the active partner, indicating that the work had been completed. Other languages, such as DEMOS, distinguish this form of interaction, by including control structures in the language which allow one entity to temporarily be treated as a resource by another entity.

Thus the SimJava entities which we use to represent the system are just `PC` and `Node`. The `Sim_entity` `PC` is shown in Figure 33, and described in detail below.

```
class PC extends Sim_entity {
  private Sim_port enqueue;
  private Sim_negexp_obj pc_hold;
  private double iatime;

  public PC(String name, int x, int y, double iatime) {
    super(name, "pc", x, y);
    this.iatime = iatime;
     enqueue = new Sim_port("enqueue", "port", Anim_port.BOTTOM, 15);
    add_port(enqueue);
    pc_hold = new Sim_negexp_obj("iatime", iatime);
    add_generator(pc_hold);
  }

  public void body() {
    int i;

    while (Sim_system.running();) {
      sim_schedule(enqueue, 0.0, 0);
      sim_trace(1,"S enqueue");
      sim_pause(pc_hold.sample());
    }
  }
}
```

Figure 33: SimJava2 entity representing the PC (generator of packets)

The entity has several parameters: in addition to the name and location coordinates, there is the mean interarrival time for the data packets `iatime`, which is used to instantiate the exponentially distributed random variable used to generate the delays between packet arrivals. Note that in general the user of a model is not necessarily the modeller and so it can be a good idea to allow the user to modify parameters of the model without editing the SimJava2 code.. The `body()` of the entity is simple—until the end of the run data packets are generated and sent as events from the port `enqueue` (to the corresponding node) and the entity waits for a sample of the random variable representing the interarrival time.

```
class Node extends Sim_entity {
  private Sim_port buffer, token_in, token_out;
  private Sim_normal_obj trans_hold;
  private int num_packets;
  private double transmean, transvar, walk;
  private boolean lr;

  public Node(String name, int x, int y, double transmean, double transvar,
                                                  double walk, boolean lr) {
    super(name, "node", x, y);
    this.transmean = transmean;    this.transvar = transvar;
    this.walk = walk;              this.lr = lr;

    buffer = new Sim_port("buffer", "port", Anim_port.TOP, 10);
    add_port(buffer);
    if (lr) {
      token_in = new Sim_port("token_in", "port", Anim_port.LEFT, 10);
      add_port(token_in);
      token_out = new Sim_port("token_out", "port", Anim_port.RIGHT, 10);
      add_port(token_out);
    } else {
      token_in = new Sim_port("token_in", "port", Anim_port.RIGHT, 10);
      add_port(token_in);
      token_out = new Sim_port("token_out", "port", Anim_port.LEFT, 10);
      add_port(token_out);
    }
    add_param(new Anim_param("num_packets", Anim_param.VALUE, "0", 1,10));
    trans_hold = new Sim_normal_obj("trans_hold",transmean,transvar);
    add_generator(trans_hold);
    stat = new Sim_stat();
    stat.add_measure(Sim_stat.WAITING_TIME);  set_stat(stat);
  }
```

Figure 34: Constructor for the `Node` entity

The entity representing the `Node` is represented in Figures 34 and 35.

Again parameters of the random variable are passed as parameters to the entity's constructor. The parameter `lr` is used to determine whether the node appears in the animation in *left-right* orientation: accepting the token on the left and passing it on to the right. Thus the constructor uses this parameter to determine just where the ports are added to the icon representing the node. Of course such visual niceties have nothing to do with the simulation aspects of the code.

```
  public void body() {
    Sim_type_p token = new Sim_type_p(2);
    while (Sim_system.running()) {
      Sim_event control = new Sim_event();
      sim_get_next(token, control);
      if (sim_waiting() > 0) {
        num_packets = sim_waiting();
        sim_trace(1,"P "+num_packets);
        for (int i = num_packets-1; i >= 0; i--) {
          Sim_event ev = new Sim_event();
          sim_get_next(ev);
          sim_process(trans_hold.sample());
          sim_completed(ev);
          sim_trace(1, "P "+i);
        }
        num_packets = 0;
      }
      sim_schedule(token_out,walk,2);
      sim_trace(1,"S token_out");
    }
  }
}
```

Figure 35: The `body()` of the `Node` entity

This model has been set up just to record the average waiting time for packets. As we can see in the `body()` in Figure 35 a default measure can be used for this and no explicit updating is necessary.

Since the `Node` maintains a single deferred queue for events, the token events (tag 2) must be selected from amongst the events representing incoming data packets (tag 0). This is done using a predicate. SimJava provides various predicate constructors. Here we use `Sim_type_p(int tag)`, which constructs a predicate to test whether an event has the specified tag value (in our case 2). This predicate can then be passed as an additional argument to several of the event handling methods. In our case we use the method `sim_get_next()` with this predicate. This has the effect of selecting an event from the deferred queue if it can find one with the matching tag, or waiting for an event with the matching tag to arrive. Events in the deferred queue with other tag values are ignored. In `Node` this means that the packets are allowed to accumulate in the deferred queue until the token next arrives.

When the token does arrive the length of the deferred queue is checked (`sim_waiting()`)—this is the number of packets currently in the buffer, the number of packets to be transmitted on this visit of the token. If this number is zero the token is immediately passed on to the next node. In fact the transmission time of the token (`walk`) is represented as a delay in the scheduled event:

```
sim_schedule(token_out, walk, 2);
```

If the value is non-zero a `for` loop is then used to send each of the packets. The animation parameter `num_packets` records how many packets are remaining to be transmitted on this visit of the token. When all packets have been transmitted the token is passed on to the next node through the port `token_out` as described above.

At the end of the simulation a special entity, the `Terminator` (see Figure 36), is used to gather the `stat` objects from each of the nodes and calculate an aggregate average waiting time. To keep it passive through the simulation run it is set to wait for a message although none will be sent. The Terminator is a modelling entity and does not represent anything in the system so it is not desirable to have it visible in the animation. Thus the animation form of the `Sim_entity` constructor is not used. On construction the `Terminator` is

```
class Terminator extends Sim_entity {
  private int LANsize;

  public Terminator(String name, int LANsize) {
    super(name);
    this.LANsize = LANsize;
  }

  public void body() {
    Sim_event next = new Sim_event();
    while (Sim_system.running()) { // kept passive by waiting
      sim_wait(next);              // for an event until the
    }                              // end of the simulation run
    double sum = 0.0;
    for (int i=1; i <= LANsize; i++) {
      sum +=
    ((Sim_system.get_entity("Node"+i)).get_stat()).average(Sim_stat.WAITING_TIME);
    }
    PC_LAN.WTavg.setText("Average Waiting Time: "+(sum/LANsize));
  }
}
```

Figure 36: The `Terminator` entity

passed a parameter `LANsize` so it knows how many nodes to collect statistics from.

The only thing remaining to complete the model is the token. The nodes have been defined in such a way as to receive the event representing the token, transmit the waiting packets (if any) and then pass on the token through the `token_out` port. However to start the simulation a token must be placed somewhere within the LAN. To achieve this we introduce an additional modelling entity, the `Starter` (Figure 37). This entity has the minimal constructor, simply calling the `Sim_entity` constructor. Its `body()` method schedules only one event, sending the token to the first node in the LAN. Since this is a modelling entity we do not wish it to have a port visibly linked to the node's icon. SimJava provides an alternative form of `sim_schedule()` to handle such cases. Instead of scheduling an event on a port, one entity can send an event directly to another if it knows the `id` of the other entity. Each entity is assigned an id when it is instantiated in the body of the simulation program (`anim_layout` in this case). The method

`Sim_system.get_entity_id()` returns the integer id of an entity given the string used as the name of the entity in its constructor. This is used in `Starter` to find the `id` of `Node1`. The starter then sends an event representing the token to that node. The entity would then be eligible for garbage collection.

```
class Starter extends Sim_entity {
  public Starter(String name) {
    super(name);
  }
  public void body() {
    sim_schedule(Sim_system.get_entity_id("Node1"), 0.0, 2);
  }
}
```

Figure 37: The `Starter` entity

The method `anim_init()` sets up the user interface to the simulation with appropriate fields and choices for user-defined parameter values. In the method `anim_layout()` values are extracted from the user interface and used to parameterise the constructors of the various entities. The `lr` parameter of the `Nodes` is used to arrange the nodes as a circuit for animation purposes. Note that the `PCs` are linked to the `Nodes` and the `Nodes` to each other but the modelling entities `Starter` and `Terminator` are not linked.

Finally, in order to set the transient and termination conditions of the run the `Anim_applet` method `sim_setup` method is overriden. This is how simulation conditions, an output analysis method and other simulation parameters are set in an animated model.

```
  // Setup the simulation's parameters
  public void sim_setup() {
    Sim_system.set_transient_condition(Sim_system.TIME_ELAPSED, warmuptime);
    Sim_system.set_termination_condition(Sim_system.TIME_ELAPSED, runtime, true);
  }
```

You should experiment with running the model under a variety of parameters. It is available in the gzipped tar file on the module web page as file `PC_LAN.java`, and can be viewed using the appletviewer and file `PC_LAN.html`.

15.3 SimJava Graph Viewer

As explained in the previous lecture note, whilst having an animation of a simulation model is a good way of gaining confidence in the model, it makes running the simulation relatively slow. Particularly when a model contains a large number of entities, this becomes an important consideration: remember that long runs and/or many replications may be necessary in order to get measures with any accuracy. Therefore it is common practice to develop a model as an animation initially and then, once it has been verified, reduce it to an application, stripping away the animation features. One of the other advantages of this approach is that report files and graph outputs can be produced from the model, which are not possible when the model is running in an applet.

SimJava2 provides support for generating graphs from any of the measures which have been collected in non-efficient mode during the simulation run. This support takes the form of a separate tool, the SimJava Graph Viewer, or SJGV, and it can greatly aid the modeller in interpreting the output from a model.

In the model `Queue_Other.java` the queue model is modified to include graph data generation. The entities are unchanged (cf `Queue_Stats.java` discussed in Lecture note 12) but the model portion of the program is as follows:

```
class Queue_Other {
  public static void main(String args[]) {
    Sim_system.initialise();
    Sim_system.set_seed_sequence(100000, 7311);
    Source customers = new Source("Customers");
    Server service = new Server("Service");
    Sim_system.link_ports("Customers", "enqueue", "Service", "arrival");
    Sim_system.set_transient_condition(Sim_system.TIME_ELAPSED, 50000);
    Sim_system.set_termination_condition(Sim_system.EVENTS_COMPLETED,
                                                "Service", 0, 500, false);
    Sim_system.set_output_analysis(Sim_system.IND_REPLICATIONS, 5, 0.95);
    Sim_system.set_report_detail(true, false);
    Sim_system.generate_graphs("test");
    Sim_system.run();
  }
}
```

Once the model has completed the five replications, it will generate the graph data (a serialised object) in the file `test.sjg`. This can then be viewed and manipulated by the modeller using the command

```
java -classpath /usr/java/lib/simjava/classes/SGJV SJGV test.sjg
```

This will start the tool with the graph data loaded. (Note that for long runs with several measures this may take some time). The tool has three panels.

- The top panel can be used to display summary information about the run(s) used to generate the data and summary information about the measurements calculated. Menus at the right of the panel allow the user to select the entity and measure of interest.

- The middle panel displays the graph for the chosen entity and measure. Note that the points on the graph are not the value of an individual observation made at the corresponding simulation time but the value that would be calculated for the measure had the run terminated at the corresponding simulation time.

- The lower panel offers the modeller further control over the displayed graph data. There is a button which allows the modeller to select to show the transient period for the data (based on the min-max truncation method), or the steady state. If replications have been used there is an option to choose to view one replication at a time or to have all superposed on a single graph.

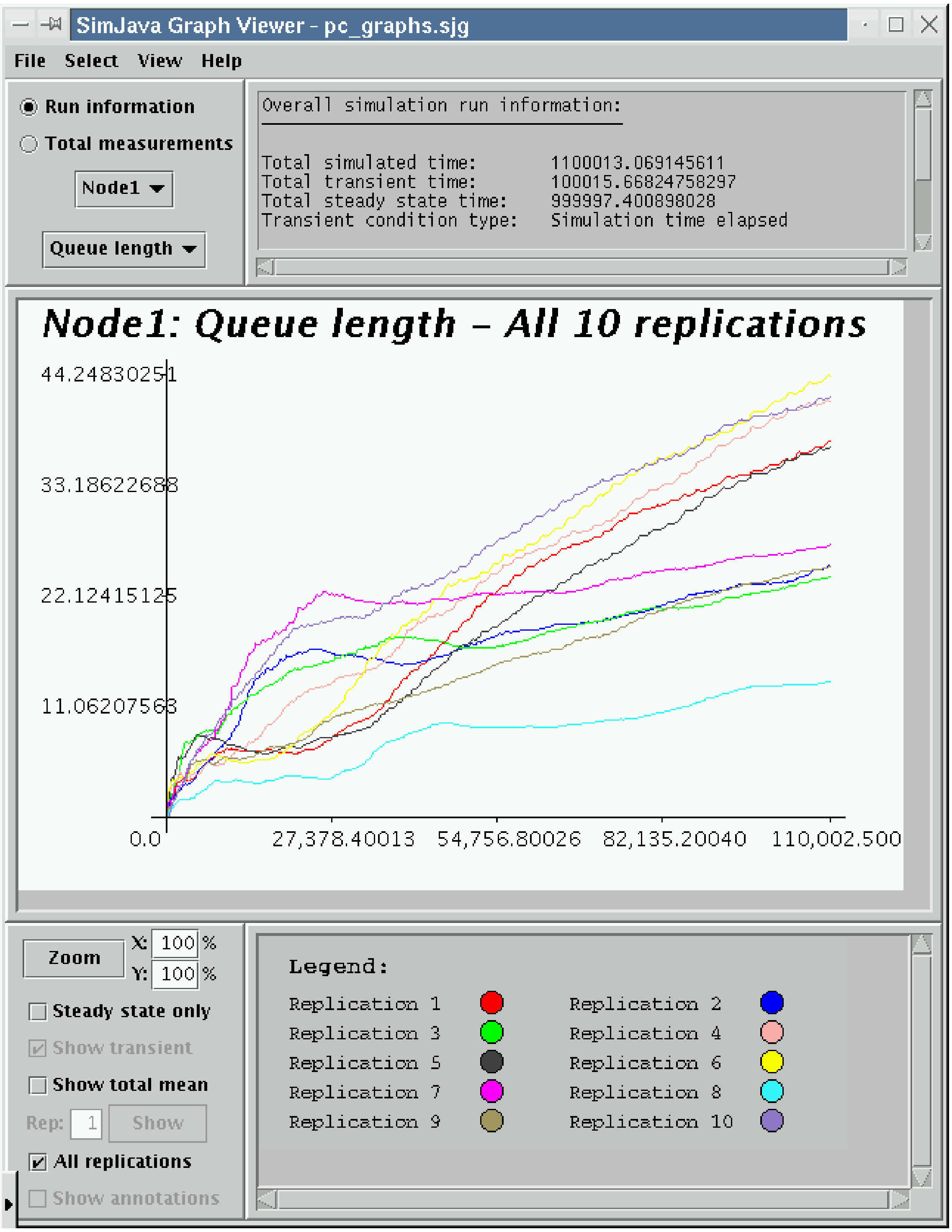


Figure 38: SJGV displaying results from the PC LAN model run as an application

Figure 38 show a screenshot of the tool displaying output from the PC LAN model run as an application. In this case the graph is displaying the queue length (average number of data packets waiting) at node 1 of the LAN. The model has been run for 10 replications and the results from each replication are shown as a separate line. The tool also offers the possibility to save a graph as in a gif file suitable for inclusion in documents (see Figure 39) .

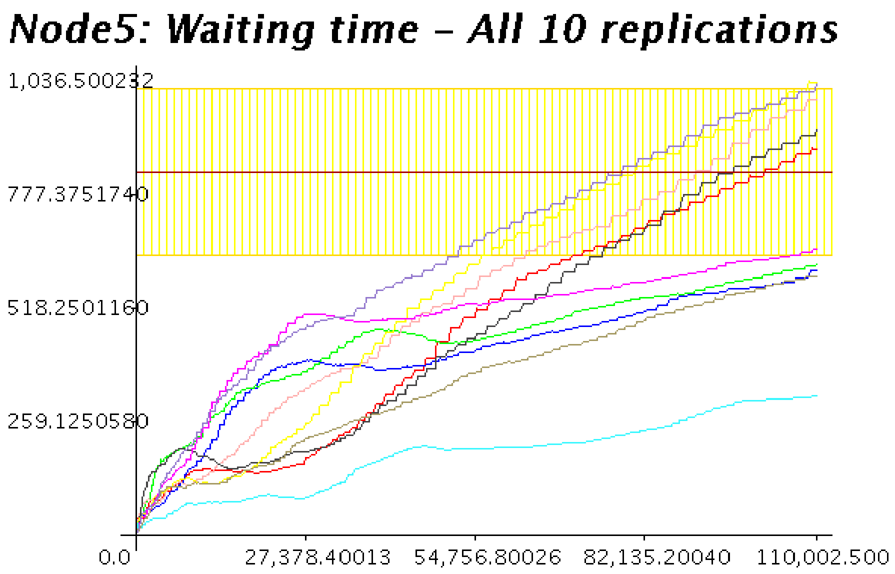


Figure 39: Graph saved from SJGV showing waiting time in PC LAN model

The PC LAN model as an application is included below. Note that the `Terminator` entity is no longer needed as we no longer need to make summary statistics available in the applet.

```
/*
 * The simple PC LAN
 */
import eduni.simjava.*;
import eduni.simjava.distributions.*;

class Pc extends Sim_entity {
    private Sim_port enqueue, test;
    private Sim_negexp_obj pc_hold;
    private double iatime;

    public Pc(String name, double iatime) {
        super(name);
        this.iatime = iatime;
        enqueue = new Sim_port("enqueue");
        add_port(enqueue);
        pc_hold = new Sim_negexp_obj("iatime", iatime);
```

```
        add_generator(pc_hold);
    }
    public void body() {
        while (Sim_system.running()) {
            sim_schedule(enqueue, 0.0, 0);
            sim_pause(pc_hold.sample());
}   }  }

class NODE extends Sim_entity {
    private Sim_port buffer, token_in, token_out;
    private Sim_normal_obj trans_hold;
    private Sim_stat stat;
    private int num_packets;
    private double transmean, transvar, walk;

    public NODE(String name, double transmean, double transvar, double walk) {
        super(name);
        this.transmean = transmean;
        this.transvar = transvar;
        this.walk = walk;

        buffer = new Sim_port("buffer");
        add_port(buffer);
        token_in = new Sim_port("token_in");
        add_port(token_in);
        token_out = new Sim_port("token_out");
        add_port(token_out);
        trans_hold = new Sim_normal_obj("trans_hold", transmean, transvar);
        add_generator(trans_hold);
        stat = new Sim_stat();      stat.add_measure(Sim_stat.WAITING_TIME);
        stat.add_measure(Sim_stat.QUEUE_LENGTH);     set_stat(stat);
    }
    public void body() {
        Sim_type_p token = new Sim_type_p(2);

        while (Sim_system.running()) {
            Sim_event control = new Sim_event();
            sim_get_next(token, control);
            if (sim_waiting() > 0) {
                num_packets = sim_waiting();
                for (int i = num_packets-1; i >= 0; i--) {
                    Sim_event ev = new Sim_event();
                    sim_get_next(ev);
                    sim_process(trans_hold.sample());
                    sim_completed(ev);
                }
```

```
                num_packets = 0;
            }
            sim_schedule(token_out,walk,2);
}    }    }

class Starter2 extends Sim_entity {
    public Starter2(String name) {
        super(name);
    }
    public void body() {
        sim_schedule(Sim_system.get_entity_id("Node1"), 0.0, 2);
    }
}
public class PC_LAN2 {
  public static void main(String args[]) {
    int num_pcs=6, i;
    double iatime=25.0, transmean=4.2, transvar=0.5, walk=1.0;
    int warmuptime=10000, runtime=110000;

    Sim_system.initialise();
    // Add entities
    Starter2 starter = new Starter2("Starter");
    for (i=1; i <= num_pcs; i++) {
        Pc pc = new Pc("PC"+i, iatime);
        NODE node = new NODE("Node"+i, transmean, transvar, walk);
    }
    // Link entities
    for (i=1; i <= num_pcs; i++) {
        Sim_system.link_ports("PC"+i, "enqueue", "Node"+i, "buffer");
        if (i < num_pcs)
            Sim_system.link_ports("Node"+i, "token_out", "Node"+(i+1), "token_in");
        else
            Sim_system.link_ports("Node"+i, "token_out", "Node1", "token_in");
    }
    // Setup the simulation's parameters
    Sim_system.set_transient_condition(Sim_system.TIME_ELAPSED, warmuptime);
    Sim_system.set_termination_condition(Sim_system.TIME_ELAPSED, runtime, true);
    Sim_system.set_output_analysis(Sim_system.IND_REPLICATIONS, 10, 0.95);
    Sim_system.set_report_detail(true, false);
    Sim_system.generate_graphs("pc_graphs");
    Sim_system.run();
}    }
```

16 Parameterisation and Workload Characterisation

So far in this course we have been concentrating on constructing a good representation of the system, appropriate for the investigation we wish to carry out. However it is not just the physical system which is represented within the model but also the effect of the environment upon the system. The most obvious influence of the environment upon the system is to produce *work* for the system to do. Therefore it is just as important to have a good representation of the workload of the system as to have a good representation of the system itself. Indeed performance measures are not, in general, absolute for a given system: performance is predicted on the basis of a given workload. This was reflected in lecture note 14 when validation of the input parameters was given equal importance with validation of the assumptions used within the behaviour of the model. Finding suitable and/or realistic values for input parameters is often termed *workload characterisation*.

There have been well-documented problems arising from badly parameterised models. The Hubble Space Telescope was rigorously simulated prior to its launch, as a cheaper alternative to testing. However, once positioned it was found that it could not produce a sharp image because the main mirror was badly flawed. The error was traced to the erroneous input data used in the simulations. Some corrective optics have now been added to the telescope in space (at a cost much higher than ground-based testing of the mirror), but the telescope will never perform as well as planned.

16.1 Parameterisation

Parameterisation is the process of assigning values to variables within a model in order to ensure that it is as accurate as feasible or appropriate. In a simulation model as well as the actual values to be used, there will be some consideration of the appropriate distribution to be used for generating the values. It is important that we consider the availability of data to assist in parameterising our models from the outset of model design and construction. For example, the level of abstraction at which work is represented must correspond to the level of abstraction at which the system is represented; i.e. since values must eventually be assigned to all input parameters, it would be a waste of effort to develop a very detailed model if there is not data available from which to construct a similarly detailed workload characterisation. Conversely, there is no need to invest a lot of effort in sophisticated statistical analysis of workload data to parameterise a crude model which is only intended to give rough estimates.

The first step is to choose which parameters to include in the model. As usual this will be influenced by the objectives of the study, and by consideration of what is likely to affect performance. If we examine the workload of a system it is likely to have many different characteristics, such as inter-arrival time of jobs, type of job, resource required per job, or size of job. However not all of these will have an impact on the performance of a system. For example, if we are modelling a router, within a communication network, at which the packet size does not affect the packet forwarding time then this characteristic of the workload (packets) does not need to be represented.

In the models we have considered during the course we have usually chosen the parameters implicitly at the same time as choosing the appropriate level of abstraction, and we have generally assumed a *homogeneous workload*, i.e. that all jobs or customers within the system behave identically. Assigning values to parameters has also been simple within our models as they have not been based on real systems. However in practice a considerable part of the effort of a modelling study might be devoted to choosing appropriate values for input parameters.

Of course the model will also contain parameters which represent aspects of the system behaviour. Many of these will be *static* parameters which contain information about the system configuration. For existing systems these are often available from published materials (manuals, specification documents, webpages etc.). In contrast *dynamic* parameters usually characterise information about the working system which must be extracted or predicted from records produced during system operation. Moreover, the actual parameter used in a model may not directly match what is measured, but may need to be derived from static and dynamic measures.

For example, in a PEPA model of an application program, access to memory is represented as a single activity, with time delay t_{avg}. Direct measurement of this time on a real machine, however, could be problematic because the detailed steps of the memory hierarchy are not typically observable. However it is easy to see that the expression

$$t_{avg} = ht_c + (1 - h)t_m$$

represents the variable, where t_c is the time to access cache and t_m is the time to access main memory, whereas h is the hit ratio. The values of t_c and t_m will be available from the manufacturer's specification, and only h must be derived from measurements.

16.2 Measurement and Monitoring

Most approaches to measurement gathering are based on some notion of *event*. An event is a pre-defined change in system state—dependent on the metric being measured—which triggers the recording of data. Example events might be memory references, disk accesses or network communication operations.

The measures or *metrics* that a performance analyst might want to record can be classified as follows:

Event-count metrics For this class of metrics, what is being measured is quite simply the number of occurrences of some type of event. Examples might be the number of cache misses, or the number of disk I/O requests made by a program.

Secondary-event metrics Here the occurrence of an event stimulates the recording of some other system parameters which are used to define the metric of interest. For example, if we are interested in the number of messages in a buffer, the occurrence of an `enqueue` event or a `dequeue` event will stimulate the recording of the current queue length.

Profile These are not so directly related to individual events, but aim to build a complete picture of the overall behaviour of a program or system by an aggregate measure. In this case, a wide variety of events may be used to initiate the recording of data.

Dynamic information about the operation of a system under a workload can sometimes be extracted from the information recorded for general purposes such as accounting and logging. However, in general the information required for performance analysis is more sophisticated and requires specific monitoring and measurement of the system.

Software performance monitors analyse the behaviour of the system during operation and write records describing the resource usage and the performance status of the system. For example, at specified intervals, queue lengths or device state indicators may be sampled and the results written to the record. Alternatively, certain events which are considered to be significant (such as swapping a page) may be documented in the record. In general the volume and form of the data recorded by software monitoring means that it is not possible to use it directly to parameterise the model. Instead it must be processed and analysed to produce suitable summaries. Most software monitors will include a reporting component which will undertake at least the initial stages of this task.

Hardware monitors are also available for some systems, and have the advantage of being "external" to the system under observation: they do not perturb system operation. However, in general the type of information which they can record is less detailed and so favour a homogeneous view of workload.

Some commonly occurring subsystems, such as databases, have given rise to specialised application software monitors. This is necessary because the subsystem has a certain amount of autonomy from the host operating system which makes access to its internal behaviour difficult.

There are four basic strategies for collecting measurement data. Each has advantages and disadvantages, and in particular the analyst should be aware of the extent to which the chosen strategy perturbs the system under observation.

Event-driven This strategy is best-suited to event-count metrics and secondary-event metrics for which the event frequency is low. Every occurrence of the event causes data to be recorded. For example if the desired metric is the number of page faults that occur in the execution of a program, the performance analyst can modify the page-fault-handling routine in the operating system to increment a counter whenever the routine is entered. An additional mechanism must also be provided to dump the contents of the counter when the program finishes execution.

This strategy has the advantage that the monitoring overhead is only incurred when the event of interest occurs. However if the frequency of the event of interest is high the overall perturbation of the behaviour of the system will be substantial.

Tracing This is essentially an enhanced version of the above strategy in which in addition to recording simple measures at each event, details of system state are also recorded so that each event can be identified. Thus in the example above, for each page fault we would record the address which caused the fault. This increases the time overhead of the strategy and can, additionally, lead to storage problems when the event of interest is frequent.

Sampling This is a general statistical measurement technique whereby a subset of the members of a population being examined is selected at random. It assumes that since the subset is chosen at random, its characteristics will approximate those of the total population.

In practical terms this means that at fixed time intervals a portion of the system state, necessary for calculating the metric of interest, is recorded. In this case the overhead incurred is independent of the event frequency and is instead a function of the sampling frequency. The sampling frequency will depend on the resolution needed to capture the events of interest, i.e. if the event happens only rarely far more samples are likely to be needed. Furthermore, each run of a sampling-based experiment is likely to produce a different result since the samples occur asynchronously with respect to the system's execution. Nevertheless, while the exact behaviour may differ, the statistical behaviour should remain approximately the same.

Indirect An indirect strategy must be used for measures which are not directly measurable. The general approach is to define an alternative metric from which the metric of choice can be deduced or derived. In these cases the appropriate metric and measurement strategy will often need to be defined on an ad hoc basis and will depend on the ingenuity and creativity of the analyst.

16.3 Workload characterisation

Workload characterisation is the process of selecting the workload or workloads on which to base the performance study. Difficult questions arise even in considering an existing computing environment: What constitutes a "typical" workload? How should a measurement interval, or *time window*, be selected? Should data from several measurement intervals be averaged? These uncertainties are compounded in considering an environment that cannot be measured directly, for example in contemplating the movement of an existing workload to a new system, or the introduction of a new workload to an existing system. This problem is exacerbated by users who will often change their usage patterns dependent on the service available.

The workload has to be broken down into *workload components*. These are the significantly different tasks the system may be required to undertake. The choice of workload components can have significant impact on the results of a performance study. For example, if the packets in two networks are generally a mixture of two sizes—short and long—the workload to compare the networks should consist of short and long packet sizes. Using the wrong workload components will lead to inaccurate conclusions.

In most of the models we have considered in the course there has been only one workload component but most systems have many more. For example, a workstation in an academic environment might need to address all the following workload components: compilers, editors, file utilities, communication, scientific computation, graphics, basic services, text processing, and more. However for most performance studies, based on the objective of the studies we can abstract details of the workload components down to a minimal set. For example, if we wished to study the impact of scientific computation on communication we might consider a model with just three workload components: `scientific computation`, `communication` and `other`. If we were developing a queueing network model each of these workload components would be represented a distinct class of customers. For each class we would then need to know the service demand of customers of that class at each of the resources in the system, as well as the number of such customers (for a closed class) or the arrival rate (for an open class). If we were developing a GSPN model each class of customer would be represented in distinct cycles within the model (only closed classes can be represented in a GSPN) competing for the resources via synchronisations (cf. the reader-writer model).

Often the resources we are interested in within our systems are quite low level, e.g. CPU and disks, whereas our workload components are expressed in terms of user applications. We must therefore derive the *devolved workload*. This involves viewing the system as a series of layers each built one on top of another. A layer offers services to the layer above and generates workload to the layer below. The devolved workload is then the amount of workload requested from the lower layer in order to satisfy one unit of workload from the layer above. For example, this technique might be used to derive the typical usage of a communication gateway involved in one `ftp` connection.

Most systems display varying behaviour over time although the time periods may differ. For example, a web server which presents pages offering advice on filling tax returns will exhibit varying patterns of use over a year, whilst most human-centric systems, such as a bank ATM, will have a workload which varies throughout each 24 hour period. In most cases, the time window which exhibits peak use is chosen for performance studies on the understanding that performance will only improve during the rest of the cycle.

When the objective of the performance study is to predict the behaviour of a system under a future workload, direct measurement of current workload characteristics is merely the starting point. The parameter values in the model will be based on a workload model which is itself predicted from the measured data. There are two major ways in which this prediction can be carried out.

Trend Analysis If historical workload measurements exist, the current workload measurements may be considered in relation to these and examined for trends. *Moving averages* and *exponential smoothing* are both techniques for extrapolating future values from a series. In this way the expected future values of the workload parameters are predicted.

Key Value Indicators Users or system managers often find it easier to predict the future in terms of their business than in terms of the service demands or load intensity of the resulting computer system workload. The key value indicator (KVI) is a quantifiable business variable such as the number of customers. The current workload is then related to the current value of the KVI and the users' predicted future computing needs in terms of KVI are then related back to give future workload estimates.

As explained in the previous section monitoring software or hardware can produce substantial amounts of information about the workload on the system. Parameterisation should have identified the key characteristics of that data for this model and modelling study. There are then various techniques which can be used to extract the necessary values from the data. For Markovian models, where we have already made an assumption that all delays within the model will be exponentially distributed, the analysis will generally be less sophisticated than what would be used to find input values for a simulation model. Commonly used techniques include the following:

Averaging Based on the sample data the *mean* value is calculated. In addition to the mean, the *median* and the *mode* are also sometimes useful ways of characterising a particular parameter as a single value.

Finding variability Variability is sometimes represented by the *coefficient of variation* which is the ratio of the standard deviation within a sample to the mean. For simulation models, this can then be used to fit a distribution to the sample points. The corresponding parameter in the model will then be drawn from that distribution with appropriate parameters.

Single parameter histograms A histogram shows the relative frequencies of various values, or range of values, of a parameter. Some simulation packages allow this sample data to be stored and used to provide varying values for the corresponding parameter during model execution via a look-up table.

Multi-parameter histograms Single parameter histograms fail to capture any correlation which may exist between different parameters of the workload. For example, short jobs may create a lower number of disk I/O requests and take a smaller amount of CPU time than long jobs. However, considering the parameters *number of disk I/O requests* and *CPU time* separately might allow jobs in the model with short CPU time and a high number of disk I/O requests—something not generally possible in reality. Considering multiple parameters at once attempts to overcome this problem. In practice correlated input parameters should usually be represented as functions of a single parameter.

Cluster analysis Most large computer systems have workloads consisting of several identifiable components or classes as in the packet size example above. There are two key goals when the observations of the system are analysed to identify these classes. Clearly, customers or jobs within a class should exhibit similar patterns of behaviour within the system. But also, if the objectives of the performance study require separate performance predictions for different workload components, they must be represented in different classes.

Cluster analysis is the principal way in which components or classes are identified within workload data. This technique is described in detail below.

16.3.1 Cluster Analysis

The data about a system generated by software monitoring may contain several thousand "user" profiles. The objective of cluster analysis is to reduce this data to a few key characteristics, identifying the different types of users in the system. Users who share characteristics are gathered together into clusters; eventually each cluster will be represented by one class in the model. The steps of cluster analysis are as follows:

1. Sample the workload data.
2. Select important workload parameters.
3. Transform parameters, if necessary.
4. Remove outliers.
5. Select a distance measure.
6. Perform clustering.
7. Interpret results.

8. Change parameters, or number of clusters, and repeat steps 3 to 6.
9. Select representative components from each cluster.

If there are n parameters of interest for each sample point (user observation) we can imagine that the selected sample points are mapped into n-dimensional space; for each sample point its position along the i axis represents its value for the ith parameter. The objective of cluster analysis is to form groups of sample points which are related with respect to their parameter values. In other words points which are close together spatially. The variance between points can be measured as their distance apart so the objective becomes to identify groups such that the variance within groups is minimised while the variance between groups is maximised.

Hierarchical approaches to cluster analysis are often taken and these can be *agglomerative* or *divisive*, depending on whether they start with as many different clusters as sample points and successively merge them, or whether they start with a single cluster containing all sample points which is successively split.

17 Comparison of Techniques

17.1 Introduction

In this note we review the various approaches to representing a system which we have considered during the course, and try to identify their relative strengths and weaknesses. In doing so we gain some insight into when it is appropriate to use each of the techniques. During the course the following ways of representing the performance related aspects of the behaviour of computer and communication systems have been discussed:

- Operational laws
- Markov processes, and related high level modelling paradigms:
  - Stochastic Petri Nets
  - Stochastic Process Algebras
  - Queues and Queueing Networks
- Simulation
- Measurement

The criteria on which we will compare the different approaches are the following:

- Time and skill requirements
- Expressiveness/assumptions
- Diagrammatic representation
- Solution
- Deriving performance measures
- Particular strengths or weaknesses

However, you should note that not all these criteria are applicable to all the approaches, and so they will not all be considered in every case.

17.2 Operational laws

Time and skill requirements The key advantage of operational laws is their minimal requirements in terms of time and skill. This makes them especially useful for calculating bounds on performance measures rather than specific values. For example, worst and best timing assumptions can be quickly evaluated to give an estimate of the range of a performance measure for a system. The resulting bounds can contribute significantly to the analyst's understanding of the system. The calculations involved are simple, and can often even be done by hand.

Expressiveness/assumptions The assumptions which are necessary for the application of the operational laws are that the system is *job flow balanced* (*conservation of work*), and that the workload and the system are *homogeneous*.

The extremely abstract view taken of the system means that the expressiveness is poor: many aspects of system behaviour cannot be represented. For example, it is impossible to include any notion of relative ordering between jobs, synchronisation, or exclusive access. This can be viewed both as a strength and as a weakness.

Solution and deriving performance measures The calculations involved are simple, and can often even be done by hand and unlike other techniques there is no distinction between solving the model and deriving performance measures.

Particular strengths or weaknesses The main strength of the operational laws is the speed and ease with which they can be formulated and evaluated.

17.3 Markov Processes

Time and skill requirements Modelling directly at the level of the Markov process is time consuming and error-prone for all but the simplest models, even with software support. However no sophisticated modelling techniques need to be learned and the solution procedure is a straightforward application of linear algebra, once the global balance equations have been formulated.

The time required to develop a model is often greatly reduced if a high-level modelling paradigm is used rather than working directly at the level of the Markov process.

Expressiveness/assumptions Perhaps the most important assumption for a Markov process is the *memoryless property*: that future behaviour is independent of past behaviour. This assumption does not seem unreasonable for essentially deterministic systems like computer and communication systems but it may influence our chosen level of abstraction. For solution purposes we also assume that the process is *finite*, *time homogeneous* and *irreducible*: again, reasonable assumptions from the perspective of what we are planning to model, which are unlikely to restrict the expressiveness of our models.

Thus there are few restrictions on expressiveness when we model directly in terms of states and events, except the assumption that all timings are exponentially distributed.

Diagrammatic representation The state transition diagram of a small Markov process provides a clear diagrammatic representation of the system/model. Indeed, if it is correctly labelled it contains all the information necessary to solve the Markov process and can be regarded as a formal representation. However, for models with hundreds or thousands of states, constructing and viewing the state transition diagram becomes infeasible and the generator matrix is the preferred representation of the system.

Note that the state transition diagram represents the dynamic behaviour of the system, but not its structure in terms of components or subsystems.

Solution Markov processes are solved *numerically* giving the steady state probability distribution. This is an *exact* solution. The main problem with this approach is one of tractability: the generator matrix, and the steady state probability vector must be manipulated and stored throughout the solution process. This can cause memory problems even with moderately sized models. This problem is often referred to as *state space explosion*.

For all but the smallest system computer-assistance is essential. However, this does not need to involve specialist software: any general package which supports the solution of linear algebra equations can be used.

Deriving performance measures State-based and rate-based measures can be calculated directly from the steady state probability distribution. This can be formalised in terms of associating rewards of appropriate values with states of interest and calculating the expected values of those rewards. Defining and identifying the states of interest can be difficult, using this direct approach, for large models. Other measures, such as response times, are usually derived using Little's law.

Particular strengths and weaknesses From a model construction point of view the main weakness of all Markov process based modelling techniques is the dependence on the exponential distribution for all inter-event times or delays. However, this is also the main strength of the approach from a model solution point of view—only memoryless systems can be solved directly in terms of the global balance equations and the memoryless property derives from the assumption of exponential delays.

State space explosion and problems of size are the major handicaps to the extensive use of Markovian based techniques in practice, and extensive research has been dedicated to finding efficient approaches to solving these models.

17.3.1 Stochastic Petri nets

Time and skill requirements The time required for model construction is often greatly reduced by using a high level modelling paradigm such as stochastic Petri nets. However, some additional skill is required because the modeller must be aware of both the notation and the semantics of the nets. The time and skill required for solution are approximately the same as when using a Markov process directly.

Expressiveness/assumptions The Petri net notation is simple with only a few primitives. This notation can be regarded as being at a low level, close to the Markov process being specified. As a result expressiveness is similar to that for Markov processes—fairly unrestricted. Stochastic Petri net models are therefore capable of representing a large class of systems.

No additional assumptions are made for stochastic Petri net models, although conditions on the underlying Markov process, such as finiteness, time homogeneity and irreducibility, can be re-expressed in terms of the net formalism.

Diagrammatic representation Stochastic Petri net formalisms are a graphical notation: the meaning of the graphical representation is formally defined and can be used to generate the underlying Markov process, via the reachability graph. In the case of GSPN the graphical notation must include annotation to distinguish any transitions which have infinite-server firing semantics.

As in the state transition diagram, the Petri net representation captures the dynamic behaviour of the system but provides little insight into the structure of the system.

Solution - generating the underlying Markov process Generating the underlying Markov process is achieved by generating the reachability graph of the Petri net. In the case of SPNs this is quite straightforward, but in the case of GSPNs it is complicated by the presence of immediate transitions and vanishing markings. Computer-assistance is essential for this task in all but the simplest models. Once the Markov process is generated solution proceeds numerically.

Deriving performance measures Although deriving measures is handled more or less in the same way as for Markov processes, identifying the states of interest can be easier in stochastic Petri net models. Here the states are identified by their net characteristics, which relate to system characteristics.

Particular strengths or weaknesses Stochastic Petri nets are based on a formal system description technique: untimed Petri nets. Consequently these models may also be analysed to investigate the functional, or qualitative, aspects of system behaviour. One disadvantage is that it can be deceptively easy to draw a SPN or GSPN model which looks innocent, but which very rapidly generates more states than the solution tools can readily handle.

17.3.2 Stochastic Process Algebras

Time and skill requirements For the novice, stochastic process algebras such as PEPA, are perhaps less intuitive to use than stochastic Petri nets. However, the component-based approach greatly simplifies the task of model construction by allowing the modeller to focus on each subsystem individually and then model their interactions. Moreover, the use of previously defined components can speed up model construction.

Basic performance measures may be derived from the solution of the underlying Markov process without detailed knowledge of the algebra, and the time and skill requirements are approximately the same as when using a Markov process directly. But, as with stochastic Petri nets, to take full advantage of the formalism the modeller should be aware of the stochastic process algebra both as a system description technique and in terms of its underlying semantics, equivalence relations, etc.

Expressiveness/assumptions Stochastic process algebra languages are very simple: systems are described as components who undertake actions and a small number of combinators define how such components and interactions between them can be built up. Since the notation is again low level the expressiveness is close to that of Markov processes themselves. It has been formally proved that the expressiveness of PEPA is equivalent to the expressiveness of bounded SPN models. The necessary assumptions are just those for Markov processes.

Solution - generating the underlying Markov process Generation of the underlying Markov process of any stochastic process algebra model is formally defined based on the operational semantics of the language. Tool support exists to do this automatically—to do it manually would be time-consuming and error-prone. Each state in the *derivation graph* is associated with a node of the state transition diagram of the underlying Markov process. Once the Markov process is generated solution proceeds numerically.

Deriving performance measures As with stochastic Petri nets, it is often possible to derive performance measures via reward structures, using characteristics of the model to identify those states to which rewards should be attached. This is much less error prone than examining the state space directly.

Particular strengths or weaknesses Stochastic process algebra models bring several new features to performance modelling, because of their basis in the formal description technique of process algebra. Primary amongst these is the compositional structure readily apparent within models. Not only does this aid model construction, but it can, in some circumstances, be exploited during model solution. Other novel features are equivalence relations which allow models to be compared and abstraction mechanisms which allow unnecessary detail to be disregarded. The main weakness of stochastic process algebras is the problem of state space explosion.

17.3.3 Queues and queueing networks

As earlier in the course, we only consider queueing networks which have a *product form* solution. Other queueing networks require more sophisticated techniques, relying on skill and experience of the modeller, or must be expanded out into the related Markov process.

Time and skill requirements For many applications, queueing networks achieve a favourable balance between accuracy and efficiency. Both single queue and simple queueing network models can be constructed, parameterised and evaluated relatively easily and without detailed knowledge of the underlying theory. Since solutions for most commonly occurring Markovian queues are widely published, single queues and simple product form queueing networks can be used almost as readily as the operational laws.

Expressiveness/assumptions Queueing networks are a compact notation in which many systems may be represented concisely. The behaviour of each service centre can be simply expressed, with a few descriptors, based on the six basic characteristics used in Kendall's notation.

The penalty for the compact notation is the limited expressiveness of the language. Most notably, queueing models cannot represent systems in which more than one resource must be simultaneously retained or in which there is internal concurrency. Work has been done to extend product form queueing networks to such systems, but the results are not generally applicable.

Diagrammatic representation The diagrammatic representation of a queueing network is largely schematic, and does not have a formal interpretation in the same way that a stochastic Petri net does. However, the structure of a queueing network will often bear a close resemblance to the physical structure of the system being modelled. The diagram usually also captures a high-level view of the dynamic behaviour of the system.

Solution and derivation of the performance measures Queueing networks are solved analytically, but generally without recourse to direct solution of the global balance equations. For most commonly occurring queues, expressions or algorithms for calculating performance measures directly from the parameters of the queue have been widely published. Product form queueing networks are such that each queue may be solved separately once the appropriate arrival rate has been found. Arrival rates at all the service centres in the network are calculated using the *traffic equations*. These have the advantage of being *linear* in the number of service centres in the network, whereas the global balance equations are *exponential* in the number of service centres in the network. This computationally efficient method of solution is largely responsible for the popularity of queueing networks for performance modelling.

Particular strengths and weaknesses Restriction to the subset of product form queueing networks implies certain assumptions about the system under study. On the one hand, these assumptions seldom are satisfied strictly. On the other hand, the inaccuracies resulting from violations of these assumptions typically are, at worst, comparable to those arising from other sources (e.g. inadequate measurement data).

17.4 Simulation

Time and skill requirements Simulation models generally are expensive to define, because this involves writing and debugging a complex computer program. They can be expensive to parameterise, because a highly detailed model typically requires a large number of parameters. Finally, they are expensive to evaluate because running a simulation requires substantial computational resources, especially if narrow confidence intervals are desired. Developing a good simulation model will often require detailed knowledge of the system being represented as well as a thorough understanding of statistical techniques.

Expressiveness/assumptions There are few restrictions on the behaviour that can be simulated, so a computer system can be represented at an arbitrary level of detail. Similarly any distribution can be used to represent delays and other distributions within the system. However, this expressiveness does not come without a cost: complex models take a long time to solve. Even a detailed representation may embody subtle assumptions about the behaviour of the system but often these assumptions will be implicit.

Solution A simulation is an *algorithmic abstraction* of the system, rather than an *algebraic abstraction*. This means that the model is executed to reproduce the behaviour of the system, rather than being solved to analyse the behaviour. There are several disadvantages to this: in order to ensure that a run is representative of all aspects of system behaviour a long execution may be necessary; all "results" are merely observations from this particular run; repeated executions are necessary in order to generate good estimates. In contrast, the advantages of this approach are the greater freedom offered to model important aspects of system behaviour in detail and the ability to consider models who state space exceeds the capabilities of numerical analysis.

Deriving performance measures Because simulation models are *run* rather than *solved*, performance measures are *observed* rather than *derived*. As with all other aspects of modelling, simulation offers the modeller a great deal of flexibility in how measures can be defined. However, a single observation should not be regarded as conclusive: variance reduction techniques should be employed to generate independent observations of the measures.

Particular strengths and weaknesses The principal strength of simulation is its flexibility; its principal weakness is its relative expense. Generating transient measures is straightforward for simulation models whereas it can be problematic for Markovian models.

17.5 Measurements

In this course we have only really considered measurements of a system as a means to develop a workload characterisation and thus values for input parameters. However, it is sometimes useful to remember that a set of measurements may also be used as a representation, or model, of a system.

Time and skill requirements Measurements generally take longer than Markovian models to develop but less time than simulation models. If specific monitoring software or hardware needs to be developed, obtaining measurements becomes very demanding both in terms of skills and time. Analysing and summarising the collected data also requires knowledge of statistical techniques and, often, a substantial investment of time.

Expressiveness/assumptions Which aspects of system behaviour can be represented by measurements will be influenced by many factors, most obviously whether the system actually exists or not. Even if the system exists monitoring a real workload may be too disruptive leading to the use of synthetic workloads in isolation. Unless monitoring software is especially written, the data collected may not represent the desired characteristics of system behaviour.

At the concrete extreme, running a benchmark experiment is in some sense using the system as a detailed simulation of itself.

Solution Although it may seem that measurements should be able to produce very accurate results this is not always true. The results may be inaccurate because many of the environmental parameters, such as system configuration, type of workload, and the time of the measurement, may be unique to this experiment.

Deriving performance measures Choosing performance measures to summarise measurement data can be as difficult as parameterising a performance model. Workload analysis can help to identify which parameters are significant. However care is needed, especially when comparing two systems on the basis of measures. For example, the performance of CPUs may be compared on the basis of their throughput, measured in MIPS (millions of instructions per second). However, if one CPU has a RISC architecture and the other has a CISC architecture the comparison in terms of MIPS is meaningless.

Particular strengths and weaknesses Obviously measurements are only possible if the system, or a similar system, exists and appropriate monitoring software or hardware is available.

Using real user workloads to generate measures raises issues about the analyst's ability to control or repeat the measures; on the other hand, using synthetic workloads such as *benchmarks* raises issues of how realistic the measurements are.

All measurements are as susceptible to experimental errors, bugs or misinterpretation as the other modelling techniques.

17.6 Some concluding remarks

Skill in introducing and assessing assumptions is the key to conducting a successful modelling study. In general, it is important to be explicit about the assumptions that are made, the motivations for their introduction, and the arguments for their plausibility. This allows the analyst's reasoning to be examined, and facilitates evaluating the sensitivity of the results to the assumptions.

Sometimes it is unavoidable that questionable assumptions must be introduced. For example, if inadequate measurement data is available the modeller may have to estimate parameter value on the basis of very little information. In this case, *sensitivity analysis* can be used to determine the extent to which such assumptions cast doubt on the conclusions of the study. This will commonly take the form of evaluating the model a number of times for variations in the assumption, and comparing the results. Thus the analyst is able to judge how influential the assumption is on the results of the model.

It is important for any modelling study that the modeller has a thorough understanding of the system under consideration, but also a clear understanding of the objectives of the study. The modeller's understanding of both is likely to improve as the model develops, which is one of the reasons why conducting a modelling study is often an iterative process. As a general rule of thumb, detail should only be added into a model when it is found to be necessary. Many system characteristics that would need to be represented in a fully general model may be irrelevant in a particular study. Identifying these characteristics leads to a simpler model and a simpler modelling study.

Finally, however detailed the model, and however long the modelling study has been, performance analysts should always bear in mind the limitations of their approach. In general, the results of performance analysis will be *evidence* rather than *fact*, and should be presented as such (cf. the Hubble Space Telescope discussed in Note 16).

Jane Hillston ⟨jeh@inf.ed.ac.uk⟩. October 7, 2005.

Modelling and Simulation: Annotated Booklist

Modelling with Generalized Stochastic Petri Nets
M. Ajmone Marsan, G. Balbo, G. Conte, S. Donatelli and G. Franceschinis, Wiley, 1995.
(JCML QA 76.9.D5 Mar. **reserved**)

This is a very good book—the essential *reference for GSPN modelling. The early chapters of the book cover basic Petri nets, extensions to nets with timing and nets with priority, and then GSPNs. Later chapters concentrate on particular application areas such as Manufacturing Systems, Communication Systems and Concurrent Programs.*

Probability, Statistics, and Queueing Theory with Computer Science Applications (Second Edition)
A.O. Allen, Academic Press 1990.
(JCML QA 273 All.)

This is quite a good book although it suffers from numerous typos. It goes at quite a gentle pace, and uses lots of examples, many of them humourously presented. As the title suggests it covers probability theory and queueing theory. Although Markov processes are discussed, it is only really as a stepping stone to queueing models. Much of the material in the book is beyond the scope of this course. The third part of the book is about statistical inference–a topic we won't cover at all.

Performance Modelling of Communication Networks and Computer Architectures
P.G. Harrison and N.M. Patel, Addison-Wesley, 1993.
(unfortunately no longer in print but the JCML has several copies: TK 5105.5 Har)

This book is a detailed textbook on performance modelling using queueing networks and it covers many sophisticated techniques which are beyond the scope of this course. The introductory chapters present a rigourous development of much of the same material as the first part of the course.

The Art of Computer Systems Performance Analysis
R. Jain, John Wiley, 1991.
(JCML QA 76.9.E94 Jai.**reserved and short loan**)

This book is primarily aimed at computer professionals who may need to know about performance analysis. It provides concise and clear introductions to many topics without going into them in depth. These topics include measurement, monitoring, workload characterisation, simulation, queues and queueing networks, and the operational laws—many of the topics we will cover in the course. I did not recommend it as the course text because the treatment is often superficial, it does not address Markovian analysis at all and only discusses simulation in a very abstract way, and also because it is expensive.

Computer and Communication Systems Performance Modelling
P.J.B. King, Prentice Hall, 1990.
(JCML QA 76.9.E94 Kin.)

This is book is intended to be similar in outlook to Harrison and Patel and so also mostly covers material which is beyond the scope of this course. It has quite a terse style and not many examples: this makes it good for looking things up in if you want to be reminded of a definition but not so useful for finding an explanation of something you don't understand.

Statistical Tools for Simulation Practitioners
J.P.C Kleijnen, Marcel Dekker 1987.
(JCML 51.62:51.92 Kle)

A very detailed account of the statistical aspects of simulation modelling. For example, in this book you can find much more detail on replication, batch means *and* regeneration, *than I will have time to cover in the course.*

System Simulation: programming styles and languages
W. Kreutzer, Addison-Wesley, 1986.
(JCML 681.3.06 Kre)

This book aims to present several different simulation modelling languages and styles by repeatedly considering the same or similar systems represented in different styles. These examples are fairly high-level and light-hearted. DEMOS is one of the languages considered.

Quantitative System Performance: computer system analysis using queueing network models
E.D. Lazowska, J.Zahorjan, G.S. Graham and K.C. Sevcik, Prentice Hall, 1984.
(No longer in print, and regrettably the library no longer has a copy; however the full text is freely available over the WWW:
`http://www.cs.washington.edu/homes/lazowska/qsp/`)

This is a good book about queueing networks which also has a very clear explanation of the operational laws and whole section on parameterisation. Like most of the other books on queueing networks, however, much of the material covered by the book is beyond the scope of this course.

Measuring Computer Performance
David J. Lilja, Cambridge University Press 2000.
`copy on order for JCML`

Despite the title this book does consider analytic modelling (in the form of queueing networks) and simulation as well as measurement. However the emphasis is on practical measurement techniques, such as defining metrics, measurement strategies, and related statistical analysis — much more depth than we go into in this course.

Performance Modelling with Deterministic and Stochastic Petri Nets
C. Lindemann, Wiley 1998.
(JCML QA 76.58 Lin)

This recently published book has four parts. The first presents a concise overview of performance modelling and stochastic modelling formalisms (similar to notes 1-10 of our course); the second discusses Petri nets in which transitions have stochastic and deterministic delays in some detail; the third presents a series of case studies based on multiprocessor architectures; the fourth presents the Petri net based modelling tool developed by the author, DSPN-express. *Performance modelling is given a higher profile in this book than in the book by Ajmone Marsan et al., but the net formalism and associated solution techniques are more complicated, due to the presence of deterministic timings, and so go beyond the scope of this course.*

Simulating Computer Systems: techniques and tools
M.H. MacDougall, MIT Press, 1987.
(JCML QA 76.9.A73 Macd.)

This book teaches simulation for performance evaluation based on a series of case studies modelled in a `C` *based simulation package* `SMPL`. *The explanations can be a bit long-winded at times, but it does address statistical techniques as well as model validation and verification.*

Capacity Planning and Performance Modelling: from mainframes to client-server systems
D.A.Menasce, V.A.F. Almeida and L.W. Dowdy, Prentice Hall, 1994.
(JCML QA 76.9.C63 Men)

This book takes a practical, example-led approach to introducing the capacity planning aspect of computer performance evaluation. It mostly concentrates on models based on queueing networks, and goes into this topic in much greater depth than we do within the course. However it also has sections on operational laws, model validation and verification and workload characterisation.

Probabilistic Modelling
I. Mitrani, Cambridge University Press, 1998.
(JCML QA 273 Mit.)

This is an excellent introduction to probability theory, Markov processes and queueing networks. You might find the early chapters useful revision of material from Maths 2Y. It covers most of the material in the first half of the course, with the exception of GSPN modelling, often in more detail than we have time for. The more extended discussions may help clarify ideas for you, so this could be a good book to read along side lecture notes 1 to 10.

Computer Networks and Systems: Queueing Theory and Performance Evaluation (Second Edition)
T.G. Robertazzi, Springer-Verlag, 1994.
(JCML TK 5105 Rob.)

As the title suggests this book is mostly concerned with product form queueing networks. However, it does also have chapters on numerical solution and on Stochastic Petri Nets. The discussion of SPN is restricted to a subset of SPN models which have a particular structure which means that they also exhibit a product form steady state probability distribution. This is beyond the scope of this course.

Introduction to the Numerical Solution of Markov Chains
W.J. Stewart, Princeton University Press, 1994.
(JCML QA 274.7 Ste.)

This is a very detailed book about solution methods for both discrete time and continuous time Markov chains. The first chapter contains a comprehensive introduction to Markov chains, and a section on queueing networks. The rest of the book is concerned with efficient techniques for finding the steady state probability distribution, especially for models with large state spaces.