Chapter 9

Specification and Construction of Performability Models

John F. Meyer
William H. Sanders

Model-based performability evaluation of computer and communication systems requires accurate and efficient techniques for model construction as well as model solution. Moreover, as the physical and logical complexity of such systems continues to grow, there is need for increased care in specifying just what is to be constructed in response to the aims of a given evaluation study. This presentation thus focuses on specification and construction of performability models, under the assumption that construction (and subsequent solution) are automated. Concepts and methods are described for each, with emphasis on specifying/realizing the relation between a base stochastic model and the measures it must support. Although stochastic activity networks are chosen as the vehicle for base model specification, many of the techniques employed convey principles that apply as well to other specification constructs.

9.1 Introduction

THE presentation that follows concerns model-based evaluation of computer and communication system performability, with emphasis on how models for this purpose are specified and constructed. Since contemporary systems of this type are seldom autonomous (closed), such modelling efforts must generally consider the representation of a *total system* composed of

1. an *object system*: the system that is the object of (model-based) evaluation, and

2. an *environment*: other systems (physical or human) that interact with the object system during its use.

Given an interacting set of physical and human resources, the distinction between these two things (although often not made explicitly) depends on which subset of resources is having its ability to perform investigated. This subset comprises the object system, which, in this context, is often referred to simply as the "system". The environment then comprises those remaining resources that interact with the object system to an extent that affects its performability in some appreciable sense.

Accordingly, the above distinction is actually determined by just what aspects of a total system's structure and behaviour are to be assessed via the evaluation process. In this regard, the discussion that follows presumes that such evaluation concerns an object system's "quality" (effectiveness) as opposed to its "cost". Given this qualification, we make some further distinctions that conform with current use of terminology in the computing field. With respect to a designated user-oriented or system-oriented service, *performance* usually refers to some aspect of service quality, assuming the system is correct. In other words, quoting [141], performance is generally indicative of " ... how well a system, assumed to perform correctly, works". (This use is not uniformly adhered to, however; e.g., the text by Kant [254] treats dependability measures as a special class of performance measures.) Performance evaluation and, specifically, model-based evaluation of measures such as throughput, response time and resource utilization, have long been recognized as important in the context of computer and communication system design. Moreover, in the development of

theory and techniques for this purpose, there has been remarkable progress over the past 20 years, particularly with regard to extensions and applications of queueing network models.

Although historically an equally long-lived concern, the need to evaluate effects of incorrectness in this context has received less attention. More specifically, we are speaking of incorrect behaviour due to either *design faults* (mistakes made by humans or automated tools in the process of specifying, designing, implementing or modifying a system) or *operational faults* (physical or human-made faults that subsequently occur during system operation). Both types of faults (see [273] for a more thorough treatment of the distinction) can obviously affect a system's ability to perform in a designated environment.

Certain measures of a system's ability to perform are based on the generic concept of *dependability*, i.e., the property of a system that allows "reliance to be justifiably placed on the service it delivers" (again see [273]). Measures of dependability thus quantify an object system's ability to perform with respect to some agreed-upon specification of desired service, where a *failure* of the (object) system occurs when delivered service no longer complies with this specification. Special attributes of dependability are defined according to the nature of failure occurrences and/or their consequences. These include *reliability* (continuity of failure-free service), *availability* (readiness to serve), *safety* (avoidance of catastrophic failures), and *security* (prevention of failures due to unauthorized access and/or handling of information).

However, if performance is degradable, then, as has been well documented in the literature (beginning with [314, 315]), measures of *performability* are needed to address issues of both performance and dependability simultaneously. Specifically, with respect to some designated aspect of system quality, a performability measure quantifies how well the object system performs in the presence of faults over a specified period of time. (This includes special cases of a single time instant at one extreme and an unbounded period of time at the other.) Such measures can thus account for degraded levels of performance that, according to failure criteria, remain satisfactory. They also permit simultaneous (i.e., within the same model) consideration of distinctions among users with respect to how failures and their consequences are perceived.

Given a specification of the measures of interest, a stochastic process or simulation model must represent the object system and environment in sufficient detail to "support" measure solution. A model with this property is thus referred to as a *base model* of the total system. In other words, based on a model's probabilistic nature, a user is able to determine the value or values of each specified measure. With regard to the object system, such modelling considerations are fairly well understood. On the other hand, the importance of realistic environment modelling is often underestimated, particularly when different aspects of the environment have differing effects on various components of the object system. Moreover, if faults are a concern, an environment model needs to account for more than just externally imposed workload, e.g., events such as transient fault occurrences, conditions such as temperature and humidity, and actions of external systems (either physical or human) in effecting fault repair and recovery. In certain cases, it is possible to view the object system as autonomous (relative to the measures in question). For example, much of traditional structure-based reliability evaluation presumes autonomy of this type, meaning that the environment part of the total system model is null. In most contemporary applications, however, the environment is nontrivial. In such cases, its modelling calls for the same kind of care and attention that is typically devoted to the specification and construction of an object system model.

In the evolution of performability evaluation (see [319, 320], for example), the initial emphasis was on solution methods (see [106, 463] for comprehensive surveys of such techniques). However, as a consequence of the above considerations, problems encountered in the specification and construction of performability models have become equally challenging from a technical point of view. Attention to this problem has increased considerably over the past decade (see the recent overviews of [106, Sec. 3] or [217], for example). In addition to the factors cited in the previous paragraph, emphasis on specification/construction issues has also been motivated by the great increase in complexity of the type of (total) systems being evaluated. In particular, with such growth there is a greater need to "match" a model to a few measures of interest, rather than attempt construction of a model that can support a host of measures. In the analytic case, the latter may require a state space that is infeasibly large, thus precluding its actual construction. On the simulation side, even if construction is possible, the

details required to support a multitude of measures may result in solution times that, for any given measure, are impractically long.

Prompted by these concerns, the intent of the discussion that follows is to describe, both conceptually and methodologically, what is meant by model "specification" and model "construction" in the context of performability evaluation. The nature of the presentation is tutorial in its style, with the hope of providing a better understanding of both endeavours. This is done via descriptions of generally defined concepts and techniques, followed by some illustrative, concrete examples. The latter presume a particular scheme that employs activity networks (SANs) for specification purposes and the software tool *UltraSAN* for model construction.

In general, the distinction between specification and construction is somewhat arbitrary. Stated informally, we take the view that model *specification* (in the "action" sense of this word) provides the "input" needed to construct and solve a model. Model *construction* is then the act of realizing the specified base model (analytic, simulation, or possibly a combination of both types) that, in turn, supports solution of the specified measures. We assume further that specification, in this context, is essentially a human activity, where the specifier(s) are sufficiently familiar with both (i) the total system being evaluated and (ii) the capabilities of the people and tools responsible for the construction and solution phases. Further, we assume that these phases are realized by a single software tool, thus permitting a more easily defined interface between model specification on the one hand, and model construction/solution on the other. Although this is admittedly a somewhat specialized form of the general problem, its consideration encompasses almost all of the important technical issues that arise in more generally defined settings.

The ensuing discussion is organized accordingly, with Section 9.2 devoted to performability model specification and Section 9.3 to the construction of models so specified. Although performability solution methods lie outside the scope of the discussion, the determination of what is to be solved (by such methods) is an essential part of the specification and must be accounted for during construction. Thus, material in Sections 9.2 and 9.3 relies on certain assumptions regarding solution capability. Section 9.4 concludes the presentation with a summary and some pointers to applications of the methods described herein.

9.2 Performability model specification

A specification of a performability model can be regarded as having three major ingredients.

S1. Specification of what is to be learned about the object system from its (model-based) evaluation, i.e., the performability measures of interest.

S2. Specification of a stochastic process on which the evaluation is to be based (a base model of the total system).

S3. Specification of how S2 relates to S1 in a manner that permits the base model (after construction) to support solution of the specified measures.

Moreover, given that the recipient of the above is a model-based evaluation tool, languages used to state S1–S3 must be sufficiently formal to permit their unambiguous interpretation and subsequent automated realization by the tool.

A measure, as this term applies to computer and communication system evaluation, typically refers to some aspect of (object) system quality, e.g., throughput, response time, time to failure, etc. However, the manner in which this aspect is measured (in a probability-theoretic sense) is typically implied by the nature of the model, e.g., a performance measure that is based on a queueing model usually refers to the expected value of some random variable under steady-state conditions. However, in the more general setting of performability measures, it is convenient to specify both the aspect of interest and the precise way it is to be measured.

Using terminology and notation of the modelling framework introduced in [314, 315], S1 thus takes the form of one or more random variables Y together with a specification of how each is to be measured. Such variables are generally referred to as *performance variables* (alternatively *performability variables*), where specification of a particular Y includes

(a) an interval of time (or an instant of time in the degenerate case) over which object system quality is being observed, and

(b) a set A in which Y takes its values (referred to in [314, 315] as the *accomplishment set*).

Specific interpretations of (a) and (b) express the intended meaning of Y and, in turn, what is to be learned about the object system via specified measures thereof. The latter can range from a complete quantification of performability, as supplied by the probability distribution function (PDF) of Y, to single-number measures such as moments of Y or, for a specified set B of accomplishment levels ($B \subseteq A$), a single performability value $Perf(B) = P[Y \in B]$. In what follows, the combination of a specific Y and its specified measure will be referred to simply as a *Y-measure*. When unqualified, the term "measure" will subsequently have the more specific meaning, e.g., it refers to an arbitrary Y-measure or, in contexts where Y is understood, a particular Y-measure.

Regarding S2, we assume that the base model being specified is a discrete-state stochastic process $X = \{X_t \mid t \in T\}$, where the index set (time base) T is continuous (typically the real interval $[0, \infty)$). For any $t \in T$, the value of the random variable X_t represents the state of the total system at time t. Note that X, once constructed, may be characterized in a form that is not literally a stochastic process, e.g., a computer program that simulates X. For specification purposes, however, the more general view of what is being specified is advantageous since, among other things, it allows the same specification to be used for both analytic and simulation model construction.

S3 is perhaps the most difficult aspect of performability model specification, particularly if the techniques employed must apply to different types of evaluation (including strict performance and dependability as well as performability) and, accordingly, a wide variety of specific base models and Y-measures. Its purpose is to specify how state trajectories (sample functions) of X map to values of Y, thus permitting solutions at the base model level (e.g., state occupancy probabilities) to determine the desired value(s) of a Y-measure. (Such a mapping is referred to in [314, 315] as a *capability function*.) Moreover, S3 should rely only on knowledge that is explicit in specifications S1 and S2 as opposed, say, to details that are revealed once the construction of the base model is completed. Although use of the latter might be possible, it is discouraged by two considerations. First, there is no guarantee that the resulting base model can support evaluation of the Y-measure(s) in question. Second, even if support is possible, X may be too complicated when fully constructed (e.g., have too many states) to be dealt with effectively in this regard (i.e., too difficult

for novice or less technical users, and too cumbersome for technical users). Hence, our development assumes that S3 is specified directly in terms of S1 and S2.

The overall specification, in addition to reflecting the above considerations, must be such that the subsequent construction phase is indeed feasible, e.g., the base model X can be characterized within practical limits of computer memory and compile time. In addition (although solutions are not an explicit concern of the material that follows), the complexity of the constructed model should allow feasible and, one hopes, efficient means of solving the specified Y-measures. Details concerning S1–S3 are described in the three subsections that follow.

9.2.1 Measure specification

If measure specification is to have general applicability, it must be done in a manner compatible with a general means of relating, per S3, the base model specification to the measure specification. To accomplish this, it is advantageous to view the accomplishment sets A of all performance variables Y as expressing value with respect to a common, uninterpreted unit of measure. In a stochastic process setting, a unit of this sort is typically referred to as a unit of "reward" (see [236], for example). Given that elements of A express reward, then, quite naturally, Y can be referred to alternatively as a *reward variable* (see [412], for example).

With this unified view of accomplishment, almost any aspect Y of object system performance (quality) can then be represented by giving reward a more specific interpretation. Moreover, by our earlier remarks concerning Y specification (see (a) and (b) at the outset of Section 9.2), it remains only to specify formally the time instant at which, or time interval over which, reward is being quantified by Y. Here, in concert with the way time is represented in the base model, time instants and durations associated with Y can either be elements of the real interval $[0, \infty)$ or, given that Y has a limiting distribution, be the limit as a time instant or interval duration approaches infinity. Specifically (again as discussed in [412], but without presuming an already-specified reward structure), three categories of reward variables can be usefully distinguished according to the nature of this time specification.

Reward variables in the first category, called *instant-of-time* variables,

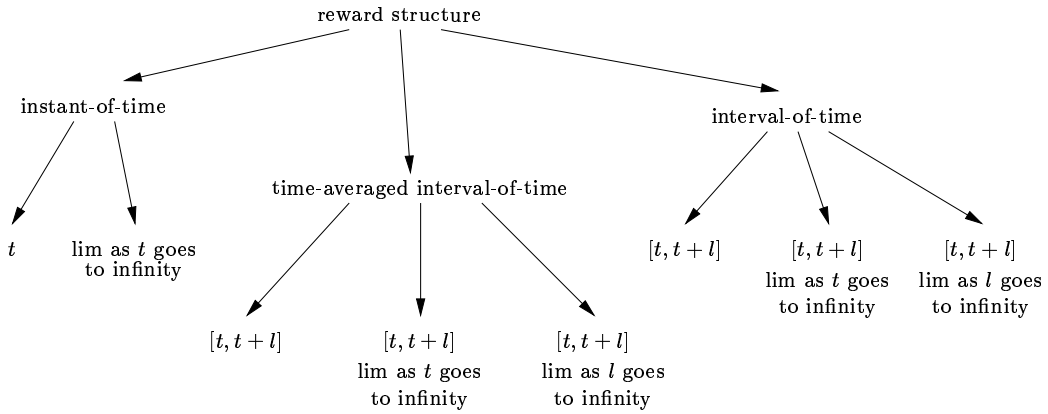

Figure 9.1: Variable specification

represent the reward experienced at a designated time during the object system's use. Variables in the second category, referred to as *interval-of-time* variables, represent the total amount of reward accumulated over a specified interval of time. Variables in the third category, called *time-averaged interval-of-time* variables, are similar to those in the second except that accumulated reward is now averaged over the duration of the specified interval. These three categories are at the first level of the tree depicted in Figure 9.1.

Interval-of-time variables can then be further classified according to the nature of the interval, again as shown in Figure 9.1. The first type represents the total or time-averaged total reward accumulated during some bounded interval $[t, t+\ell]$ (the leaves labelled $[t, t+\ell]$ in Figure 9.1). The second is obtained as a limiting version of the first (the middle leaves of the middle and right branch), where the duration ℓ remains finite and t goes to infinity. This type is useful in representing ability to perform over a bounded period when the system is initially operating under steady-state conditions. The final type (rightmost leaves) is similar to the second except that the initial instant t is fixed and the duration ℓ goes to infinity. For any such variable Y, the specification of a Y-measure is completed by a statement of how Y is to be measured in a probability-theoretic sense. The manner in which reward is interpreted for Y will typically determine which measures will make sense. The following example of a small total system illustrates some possible choices. It is complex enough to exemplify

much of what is discussed throughout the remainder of the presentation.

Example: Faulty multiprocessor Consider an object system consisting of N processing elements (PEs) that share a common input queue of finite capacity L. The environment is a serial stream of incoming tasks that arrive as a Poisson process with rate λ. Whenever queue capacity permits, an arriving task enters the object system; if the queue is full, an arriving task is rejected. Tasks are scheduled to processors on a FIFO basis, so that any "available" processor (the meaning of this will be explained in a moment) will attempt to access the task at the head of the queue. It is assumed that all of the available processors are equally likely to receive the task.

All PEs have identical structures and behave as follows. A PE, having accessed a task when idle, processes that task in exponentially distributed time with mean value $1/\mu$. Moreover, while busy with a single task, it can access a second task and accommodate both as if they were one, i.e., the two are processed together at the single-task rate μ and complete simultaneously. Accordingly, we regard a PE as being *available* if it is either idle or processing a single task; otherwise it is *unavailable* in the sense that additional loading is precluded.

In the case of double-task processing, however, all is not perfect. Specifically, there is interference, due to a fault in the design, that results in processing errors. Fortunately, these are detected at the time both tasks complete, with the likelihood of such errors being encountered given by the probabilities

$$\begin{aligned} p_1 &= \text{the probability that exactly one of the two tasks} \\ &\quad\ \text{is processed erroneously,} \\ p_2 &= \text{the probability that both tasks are processed erroneously,} \end{aligned}$$

where $0 \leq p_1 + p_2 \leq 1$. If a task completes with erroneous processing (we call this an *erroneous completion*), it is immediately returned to the PE for reprocessing. Accordingly, since both tasks may have erroneous completions, we assume further that $p_2 < 1$, so as to eliminate the possibility of live-lock. When the processing of a task completes without errors (an *error-free completion*), which may require several processing iterations, it departs the object system. Thus, any task that enters the system (i.e., is not rejected at the input) will be eventually accessed by some processor

and, in turn, will eventually enjoy an error-free completion. In the case in which only a single task resides in a PE during an iteration (either a task that was just accessed or one that was returned internally for reprocessing), everything works fine, and hence error-free completion is guaranteed.

For the total system just described, there are a number of Y-measures that might be considered. For example, the probability that the queue is full at some given time t is of obvious interest, due to the finite capacity of the queue and the slowdown due to reprocessing of tasks with erroneous completions. This measure could be specified, for example, in terms of an instant-of-time variable V_t, where reward is interpreted as the number of tasks in the queue. (Note that, although the reward variable part of the specification is referred to generically as a Y-measure, we take the liberty of using other symbols such as V and W; in particular, the symbol V is intended to suggest an instant-of-time variable.) Hence, when coupled with the interpretation of t,

$$V_t = \text{the number of tasks in the queue at time } t.$$

Recalling that L is the capacity of the input queue, if we take the (singleton) set $B = \{L\}$ to be the accomplishment set of interest, the associated measure is then the performability value

$$Perf(B) = P[V_t = L]\,.$$

Alternatively, this probability could be specified via the binary-valued reward variable (indicator variable)

$$V_t = \begin{cases} 1 & \text{if there are } L \text{ tasks in the queue at time } t \\ 0 & \text{otherwise} \end{cases}$$

coupled with its expected value (mean) as the associated measure. In other words, $E[V_t]$ (by virtue of properties of both V_t and E) likewise expresses the probability of a full queue.

To specify the steady-state probability of the queue being full, one instead uses a variable V_∞ whose PDF is the limiting distribution (provided it exists) of its corresponding V_t variable, e.g., for the first of the two variables considered above,

$$V_\infty = \text{the number of tasks in the queue as } t \to \infty.$$

Then $Perf(B)$, where again $B = \{L\}$, gives the desired measure. Since tasks are assumed to arrive as a Poisson process, $Perf(B)$ also specifies the steady-state probability that an arriving task will encounter a full queue and hence not enter the system.

Measures of productivity of an object system can typically be specified using interval-of-time variables. For example, relative to a particular processor, if a unit of reward is associated with each error-free task completion, then the interval-of-time variable $Y_{[0,\ell]}$ represents the number of error-free completions (for that processor) during the time interval $[0, \ell]$. If the rate of error-free completions, as averaged over $[0, \ell]$, is also of interest, this can be specified via the time-averaged interval-of-time variable

$$W_{[0,\ell]} = \frac{Y_{[0,\ell]}}{\ell},$$

where $Y_{[0,\ell]}$ is as above. Corresponding steady-state variables are obtained by considering their limits as $\ell \to \infty$. Performability measures associated with steady-state variables could range from simple mean values to full probability distributions.

9.2.2 Base model specification

Let us turn now to the second ingredient, namely the specification of a stochastic process X on which the evaluation (solution of the measures given by S1) is to be based. One means of doing this, of course, is to specify X directly, e.g., in the case of a finite-state, time-homogeneous Markov process, via a specification of X's initial-state distribution and its infinitesimal generator. However, the following discussion is aimed at higher-level specifications from which a base model X be can automatically constructed (by a tool that receives the specification).

Stochastic activity networks The most popular vehicle for accomplishing performability evaluation has been forms of stochastic Petri nets (SPN) [328, 349], which are typically referred to as "models". However, as we use them in the context of S2, such graphical models serve to specify lower-level stochastic models (base models); hence, when there is need to emphasize this distinction, we will refer to these graphical models more precisely as model specifications. Also, due to space limitations, we choose

to focus on an SPN-variant that is most familiar to us, namely stochastic activity networks (SANs) [333, 318, 405]. Finally, in keeping with this choice, the tool we use for construction purposes is UltraSAN (see [96, 415]). Among other things, this permits discussion and illustration of hierarchical specification/construction techniques that are not implemented in an earlier SAN-based tool (Metasan [409]).

Structurally, SANs have primitives consisting of *activities, places, input gates* and *output gates.* Activities ("transitions" in Petri net terminology) are of two types, *timed* and *instantaneous.* Timed activities represent actions of the object system or environment whose durations impact on the measures in question. Instantaneous activities, on the other hand, represent actions that, for modelling purposes (support of the measures), can be regarded as having negligible durations. Cases associated with activities permit the specification of two types of spatial uncertainty. Uncertainty about which activities are enabled in a certain state is specified by cases associated with intervening instantaneous activities. Uncertainty about the next state assumed upon completion of a timed activity is specified by cases associated with that activity. Places are as in Petri nets and may contain *tokens.* An assignment of numbers of tokens to places in the network is a *marking* of the network. Input gates each have an *enabling predicate* and *function* that, as will be seen below, control the execution of the network. Output gates have only *functions*, which define the change in marking of a network upon completion of activities. The use of gates permits greater flexibility in specifying enabling and completion rules than with ordinary stochastic Petri nets.

The stochastic nature of a SAN is described by associating an *activity time distribution function* with each timed activity and a *probability distribution* with each set of cases. Generally, both distributions can depend on the global marking of the network. The activity time distribution can be any probability distribution function and is dependent on the marking in which the activity is "activated" (see below). The probability distribution associated with cases, referred to as the *case distribution*, can depend on the marking of the network at completion time of the associated activity.

Before describing how a SAN executes in time, it helps to define a few related terms. In particular, a *stable* marking of a SAN is one in which no instantaneous activities are enabled. Conversely, a marking in which there is at least one instantaneous activity enabled is *unstable.* An

activity is *enabled* if the predicate of each of its input gates is true (*holds*, in SAN terminology), and there is at least one token in each of the directly connected input places (i.e., those places connected by a directed arc from the place to the activity).

Informally, SANs execute in time through completions of activities that result in changes in markings. Activities complete some period of time after they are *activated*, depending on their activity time distribution functions. (Activation of an activity occurs when the activity becomes enabled or when it completes while remaining enabled.) More specifically, an activity is chosen to *complete* in the current marking based on the relative priority among activities (instantaneous activities have priority over timed activities) and the activity time distributions of *enabled* activities. Selection of a case of the activity chosen to complete is then based on the probability distribution for that set of cases. These two choices uniquely determine the next (stable or unstable) marking of the network, which is subsequently obtained by executing the input gates connected to the chosen activity and the output gates connected to the chosen case.

Activities may also be restarted, or *reactivated* [318], under certain circumstances. In particular, for each marking in which an activity may be activated, a set of *reactivation markings* can be defined. If, prior to completion, one of these markings is reached, then the activity is *reactivated*, i.e., aborted and then immediately activated. This provides a mechanism for restarting activities, with either the same or a different activity time distribution. One can think of this mechanism as a generalization of the execution policies proposed for other forms of stochastic Petri nets, specified on a per-activity basis.

SAN specification of a single processor To illustrate the use of stochastic activity networks as specifications, let us again consider the faulty multiprocessor described in the previous section. We are interested in specifying an analytic base model that can support solution of measures of the type illustrated earlier in this section. Figure 9.2 depicts the graphical part of a SAN specification for a one-processor version of the faulty multiprocessor. In the figure, *arrival*, *access* and *processing* are timed activities. *Arrival* and *access* each have a single case, and *processing* has three cases. *Capacity* and *available* are input gates, and *correct* is an output gate. *Size*, *queue* and *num_tasks* are places.

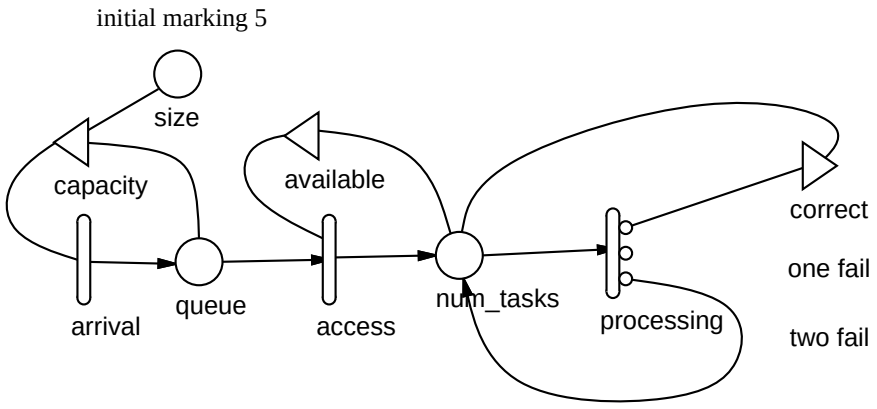

Figure 9.2: SAN specification

Table 9.1: Activity time distributions

Activity	Distribution	Parameter values
access	exponential	
	rate	*1000*
arrival	exponential	
	rate	*1.5*
processing	exponential	
	rate	*1*

This SAN diagram, together with its activity time distributions (Table 9.1), case probabilities (Table 9.2) and input and output gate definitions (Tables 9.3 and 9.4, respectively), precisely specify the object system, workload and fault environment described earlier. To see this, and to illustrate the use of SANs in specifying performability models, we now describe the functioning of the model. In particular, incoming task arrivals are modelled by completions of activity *arrival*. Upon arrival, a task is represented by a token in place *queue*, and the number of tokens in *queue* is the number of arrived tasks waiting for service. The finiteness of the queue is represented by input gate *capacity*, which holds whenever the queue is not full. This can be seen from the gate's predicate in Table 9.3, which specifies that the number of tasks in the queue must be less than the sys-

Table 9.2: Activity case probabilities

Activity	Case	Probability
processing	1	*if* $(MARK(num\_tasks) == 1)$ *return*(*1.0*); *else return*(*0.81*);
	2	*if* $(MARK(num\_tasks) == 1)$ *return*(*ZERO*); *else return*(*0.18*);
	3	*if* $(MARK(num\_tasks) == 1)$ *return*(*ZERO*); *else return*(*0.01*);

Table 9.3: Input gate definitions

Gate	Definition
available	Predicate $MARK(num\_tasks) < 2$
	Function /* *do nothing* */ ;
capacity	Predicate /* *has the buffer capacity been reached?* */ $MARK(queue) < MARK(size)$
	Function /* *do nothing* */ ;

Table 9.4: Output gate definitions

Gate	Definition
correct	/* *complete all tasks at the same time* */ $MARK(num\_tasks) = 0$;

tem capacity, for the attached activity to be enabled. When an activity completes, a case is chosen (for activity *arrival*, there is only one case, so it is chosen by default), the input gates of the activity are executed, one token is subtracted from each input place, the function of each output gate connected to the chosen case is executed, and one token is added to each output place of the chosen case. Activity *arrival*'s completion thus results in the execution of the function of input gate *capacity* (which does nothing, according to its specification) and the addition of one token to place *queue*, because of the output arc from the activity to place *queue*. The rate of activity *arrival* specifies the rate of task requests, which may or may not be serviced, depending on the number of tokens in place *queue*. Attempts of the PE to access a task from the queue are represented by activity *access*, whose activity time (see Table 9.1) represents the time to acquire a task from the queue, given that the processor is available (i.e., the PE is idle or processing a single task). The enabling of this activity is therefore controlled by input gate *available*. Activity *processing* represents the processing time of the PE. Specifically, when processing completes, the action taken depends on whether one or two tasks were processed. This choice is reflected in the cases of activity *processing* (see Table 9.2), whose probability distribution depends on the marking of place *num_tasks*. If only one task was processed, processing was correct with probability 1, and case 1 (the topmost case, in the diagram) is always chosen. When this occurs, one token is removed from *num_tasks* because of the input arc from the place, and output gate *correct* is executed.

If the processing of two tasks was completed (simultaneously), the outcome is probabilistic, as per the informal specification given earlier. In this situation, the three cases of activity *process* correspond, respectively, to the following alternatives.

1. Both tasks were processed correctly and hence depart the system.
2. One task was processed correctly, thus departing, while the other must be reprocessed.
3. Both tasks were processed incorrectly, and hence both must be reprocessed.

The probabilities associated with each of these actions are given in Table 9.2, and the actions are carried out by the arcs and output gate connected

to the activity. For example, if two tasks complete processing, with probability 0.01 they were both processed incorrectly. If this occurs, the third case of the activity is chosen, and the token that was removed from place *num_tasks* by the input arc attached to *processing* is returned via the output arc attached to the case.

The example just presented illustrates the specification of a single processor system as a SAN. While multiple processor versions could be specified directly as SANs through the use of additional activities, places and gates, this would be cumbersome for large systems. Furthermore, structures (such as symmetries) in the SAN that could be exploited in the construction process would be difficult to detect and use profitably. Composition methods for SAN specifications address both of these concerns and are now discussed.

Composed models As pointed out in the previous paragraph, large systems are cumbersome to express directly in terms of a single SAN specification and can lead to models whose solution is difficult, due to the complexity of the resulting specification. Composed model specifications permit hierarchical composition of SAN models, and their corresponding reward variable specifications (to be discussed in the next section), in an iterative manner. This composition is done using two operations: *replicate* and *join*. The composition acts on the reward structure(s) of a SAN, as well as the SAN itself, to preserve properties needed in the subsequent construction process. Formally, the resulting specification is known as a *composed SAN-based reward model* (composed SBRM) [413].

The *replicate* operation replicates a SAN and associated reward structure a certain number of times, holding some subset of its places, called its "distinguished" or "common" places, common to all resulting submodels. Replicated submodels interact through these common places; although the submodels are identical in their specifications, each of them has its own marking. Each replica will have the same reward structure(s) (see the next section for their specification) as the original submodel.

The *join* operation allows the combination of several different submodels. Informally, the effect of the operation is to produce a composed model that is a combination of the individual submodels. Again, distinguished places play an important role in the operation. In this case, however, a *list* of places is associated with each component submodel. The first places

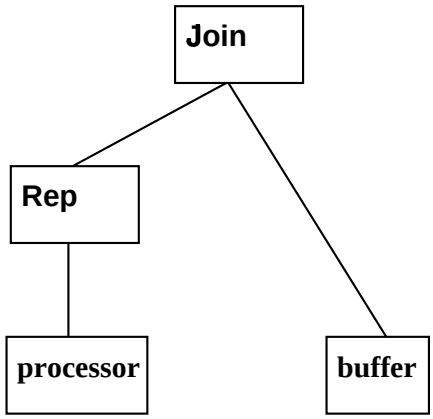

Figure 9.3: Composed N-PE faulty multiprocessor specification

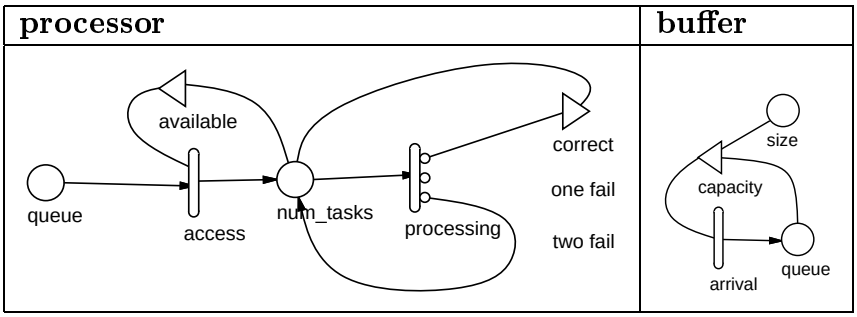

Figure 9.4: SAN submodels in composed model specification

in all of the lists are merged to form a single place, the second places are merged to form another place, and so on. Particular elements on the lists can be null, making it possible to create certain places from a proper subset of the joined submodels.

Composed model for N-PE faulty multiprocessor A composed model specification of an N-PE faulty multiprocessor is given in Figure 9.3. The nodes at the leaves of the tree are SANs, together with their reward structures. These SANs represent the workload offered to the multiprocessor (**buffer**) and the processing at a single PE (**processor**), as shown in Figure 9.4. Note the similarity with the single-processor version described earlier. Since we now consider an N-processor version, we must

replicate the processor specification N times. Replication is done via the "**Rep**" node in Figure 9.3. The **Rep** specifies that its child node (**processor**) should be replicated N times, taking *queue* to be a common place for all the replica submodels (again see Figure 9.4). In other words, this operation creates a specification consisting of N processors, all sharing a common place *queue*. Specification of the model is completed by joining the N processor specification to the **buffer** submodel, using the **Join** node in Figure 9.3. This node specifies that the place *queue* in the N-PE submodel should be common with place *queue* in the **buffer** submodel.

9.2.3 Reward structure specification

We now describe how a SAN specification of a base model is linked to measure specifications of the type described in Section 9.2.1. This connection is provided by a *reward structure*, which is similar to that used for Markov reward models, but specified at a level that coincides with the base model's specification (the *SAN level*). A reward structure typically consists of two types of rewards: an *impulse* reward that is associated with each state change and a *rate* reward that is associated with the time spent in a state. This idea was extended in [412] to permit its formulation at the SAN level, where impulse rewards are assigned to activity completions and rate rewards are assigned to particular numbers of tokens in places. A similar extension was made by Ciardo *et al.* [77] in defining "stochastic reward nets".

To describe such structures more formally, let $\mathbb{N}$ denote the set of natural numbers and let $\mathcal{P}(P, \mathbb{N})$ be the set of all partial functions from P to $\mathbb{N}$. Then, as in [412], an *activity-marking-oriented reward structure* of a SAN, with places P and activities A, is a pair of functions

$$\mathcal{C}: A \to \mathbb{R},$$

where, for all $a \in A$, $\mathcal{C}(a)$ is the reward obtained due to completion of activity a, and

$$\mathcal{R}: \mathcal{P}(P, \mathbb{N}) \to \mathbb{R},$$

where, for all $\nu \in \mathcal{P}(P, \mathbb{N})$, $\mathcal{R}(\nu)$ is the rate of reward obtained when, for each $(p, n) \in \nu$, there are n tokens in place p. In this context, an element $\nu \in \mathcal{P}(P, \mathbb{N})$ is referred to as a *partial marking* of the SAN, signalling that

this marking refers only to places in a subset of P, namely the domain of the partial function ν.

Informally, impulse rewards are associated with activity completions (via $\mathcal{C}$), and rates of reward are associated with numbers of tokens in sets of places (via $\mathcal{R}$). We adopt the convention, in practice, that rewards associated with activity completions and partial markings are taken to be zero if they are not otherwise explicitly assigned. Performance, dependability and performability variables can then be easily defined in terms of these rewards, using the measure specification method discussed earlier in this section (see Figure 9.1).

Reward structure specification for faulty multiprocessor To illustrate the use of reward structures, consider those described in Table 9.5 for the composed N-PE faulty multiprocessor. Following the convention employed in *UltraSAN*, we specify rate rewards using multiple predicate-function pairs. The interpretation of each predicate-function pair is as follows. When the predicate is true, reward is earned at the rate specified by the function. The total rate at which reward is accumulated by a variable is then the *sum* of the rewards contributed by all of the predicate-function pairs. Furthermore, from the nature of the replicate operation described in the previous subsection, a reward structure is replicated along with its SAN. The contribution to the total reward by the replicated SAN is thus obtained by summing the rewards associated with all of the replicates.

For example, consider Table 9.5's first reward structure, which specifies a reward structure for an indicator random variable. This variable is used to determine *probability non-blocking*, i.e., the probability that the input queue is not full. In that case, reward is accumulated at rate 1 whenever the queue is not full and at rate 0 when it is full. Using this structure for an instant-of-time variable, we obtain the status of the queue either at a particular time t or in steady state. Since it is an indicator variable, its expected value provides the desired non-blocking probability. The second variable in this table illustrates the use of a reward structure to specify the utilization of the N-PE system, where utilization is defined to be the fraction of PEs that are processing at least one task, i.e., if there are k such PEs ($0 \leq k \leq N$), then the utilization is k/N (having value 1 if the system is fully utilized). Since the SAN (subnet) **processor** is replicated, the corresponding reward structure is likewise replicated (via reward sum-

Table 9.5: Reward rate specification for faulty multiprocessor (impulse rewards are not used in the example, hence omitted from the specification)

Variable	Definition
probability non-blocking	
	Rate rewards Subnet = *buffer* Predicate: $MARK(queue) < MARK(size)$ Function: *1*
utilization	
	Rate rewards Subnet = *processor* Predicate: $MARK(num\_tasks) > 0$ Function: *1.0 / N*
number of tasks in queue	
	Rate rewards Subnet = *buffer* Predicate: *1* Function: $MARK(queue)$
number of tasks in system	
	Rate rewards Subnet = *buffer* Predicate: *1* Function: $MARK(queue)$ Subnet = *processor* Predicate: *1* Function: $MARK(num\_tasks)$
completions of processing	
	Rate rewards none

ming; see above). Hence, as indicated in Table 9.5, the reward structure specification for *utilization* is the contribution of a single PE (either 0 or $1/N$ according to whether the PE is idle or busy) to the utilization of the entire system.

The third variable in the table illustrates the use of a reward structure whose predicate is "true" (specified as "1" in *UltraSAN*) for all markings, and whose function changes depending on the marking. This structure is defined in terms of the SAN **buffer** and, through its specification, supports an instant-of-time variable Y_t whose value is the number of tasks in the queue at time t (or in steady state if $t \to \infty$).

The fourth variable has a reward structure involving both the **buffer** and **processor** components of the base-model specification, demonstrating the use of multiple predicate-function pairs that relate to different SANs. Because tokens are counted both in place *queue* and in place *num_tasks* (since the latter are summed, this accounts for all the tasks currently being processed by some PE), the instant-of-time variable obtained is the number of tasks in the N-PE faulty multiprocessor system.

The final variable illustrates the use of impulse rewards. In this variable, one "unit" of reward is accumulated each time activity *processing* completes in any of the **processor** submodels. An interval-of-time variable can be defined with this structure to determine the mean, variance or PDF of the number of processor completions in some fixed interval of time. It is also possible to define more complicated variables that make use of both rate and impulse rewards.

9.3 Performability model construction

The construction of a performability model from a given specification can take many forms, depending on the amount of detail required in the base model and the type of solution method employed. Broadly speaking, model solution can be accomplished by either numerical analysis or simulation. In the case of analysis, the model to be constructed is a stochastic process that can support a feasible solution of the specified Y-measure(s). If solutions are to be obtained by simulation, the end product of the construction phase is a discrete-event simulator, which, when executed, produces the desired estimators of the specified measures. In this case, the simulator

itself serves as the base model.

With either form of solution, there are many feasible base models for a given performability model specification, differing in the level of detail they preserve relative to the specification. At one extreme, the models can be very detailed, preserving all the details of the base model's specification and supporting any Y-measure that is specifiable in the sense described in Sections 9.2.1 and 9.2.3. At the other extreme, a base model can be the least refined model that feasibly permits a particular specified Y-measure to be solved (by the designated type of solution method). Models at the detailed end of this spectrum are referred to, quite naturally, as *detailed base models*; following the terminology of [413], those near the other end are *reduced base models*. In the remainder of this section, we first describe algorithms that are necessary and common to the construction of both the detailed and reduced varieties. This is followed by a further discussion of construction techniques for each type, along with examples that illustrate their use.

9.3.1 Marking-transition algorithms

Regardless of the type of model construction method employed, there are several algorithms that are common to both detailed and reduced base model construction methods. Generally speaking, these algorithms convert execution of the model's specification (a hierarchical composition of SANs) in terms of its marking behaviour (as discussed in Section 9.2.2) to a less detailed representation that remains appropriate for the intended use of the model. As with most decisions regarding model construction, what is "appropriate" is determined by what we would like to know about the object system. In this regard, examination of the methods employed for the specification of Y-measures (Section 9.2.1) and their corresponding reward structures (Section 9.2.3) suggests that we would like to preserve information regarding timed-activity completions and times spent in stable markings. On the other hand, there is no need to account for particular sequences of instantaneous activities that might complete, or about sojourns in unstable markings, since neither of these occurrences contributes to the value of a reward variable.

Thus, for the purpose of construction, it is enough to consider the possible sequences of timed-activity completions, along with the sequences of

stable markings that result from the completions. More precisely, for each stable marking, each timed activity that may complete in that marking, and each case that may be chosen, it suffices to determine the probability distribution of the next stable marking. After the completion of a timed activity and choice of a case, the next stable marking can be reached in one of two ways. If execution of the gates and arcs connected to the chosen case results immediately in another stable marking, then no additional work is required; the probability associated with this possible next stable marking is simply the probability associated with the case. However, if an unstable marking is reached upon execution of the gates and arcs associated with the chosen case, one must address the following two questions.

1. Does there exist a sequence of subsequent instantaneous activity completions and unstable markings such that a stable marking is never reached?

2. If more than one instantaneous activity is enabled, does the probability distribution across next stable markings depend on which instantaneous activity is chosen to complete first?

The answer to the first question determines whether a next stable marking distribution exists for this marking, timed activity completion and case selection. In SAN terminology, if the answer is *no* for all reachable stable markings, activities that may complete in these markings and cases of these activities, we say that the SAN is *stabilizing* [405]. Unfortunately, this question is undecidable, i.e., there is no decision algorithm for deciding whether a SAN, together with a specified initial marking, is stabilizing (again see [405]). While this is disappointing from a theoretical viewpoint, a computer with finite memory is unable to account for arbitrarily long sequences of intervening instantaneous activity completions and resulting unstable markings. In practice, therefore, one simply prescribes an upper bound on the number of subsequent instantaneous activities that will be considered for some SAN in a given initial marking. If all such sequences have lengths less than or equal to this bound, then the SAN is declared (conservatively) to be stabilizing for that initial marking.

If a SAN is stabilizing in the sense just defined, then the second question becomes important. It asks whether the next stable marking distribution is unique or depends on instantaneous activity choices that are made while

the next stable marking distribution is being generated. Recall that, with SANs, choices made upon completion of an activity are determined by its cases. This is in contrast, for example, with the GSPN notion of a "random switch" (see [5], for example) for which a probability distribution is specified on completion of one or more instantaneous activities. Cases are therefore advantageous, since they permit a clear distinction between the alternatives that need to be accounted for by the model, and those that are incidental. In keeping with this distinction, we would like the probability distributions for next stable markings to be independent of the completion choices among concurrently enabled instantaneous activities. If this independence holds for all such distributions, we say that a SAN is *well-specified* [405]. Accordingly, a well-specified SAN has a complete probabilistic specification in the sense that its designated activity time distributions and case distributions suffice to describe its probabilistic behaviour completely. Since solution of the specified Y-measures relies on the knowledge of such behaviour, only well-specified SANs are employed for the purpose of base model specification.

Fortunately, it is decidable whether a SAN is well-specified. Moreover, algorithms [405] exist that are constructive in the following sense. If a SAN in some initial stable marking is well-specified, then the algorithm determines the probability distribution of the next stable marking; if it is not well-specified and, moreover, the SAN violates the well-specified property, then the algorithm returns a *yes* answer to Question 2. Informally, determining whether such choices are influential involves enumerating all possible "paths" (i.e., sequences of instantaneous activities and unstable markings) that are traversed before a stable marking is reached. A relation is then defined on this path set, where two paths are related if the choices made among concurrently enabled instantaneous activities are common. It can be shown that this is a compatibility relation on the path set; that is, it is both reflexive and symmetric. However, it need not be transitive, and hence it is generally not an equivalence relation.

Once the maximal compatibility classes have been determined for the relation, the set of possible next stable markings is computed for each class; this is done by listing the stable markings that result from paths in a given class. The probability distribution for that class is then computed by summing, for each resulting stable marking, the probabilities of the paths that terminate in that marking. Finally, the probability distributions

so determined for each class are compared. If they are identical, then there is no dependence on the choice distinctions; moreover, the common distribution becomes the next stable marking distribution for the timed activity, case and initial marking considered. If they are not identical, then the SAN is not well-specified.

Given that a SAN is stabilizing, the above algorithm provides a method for moving from stable marking to stable marking, in which at each step, the timed-activity completion that caused the transition is recorded. If the SAN is not well-specified, the algorithm will detect this fact and halt the computation of the state-level representation. Note that this computation can be done on the fly, with no need to record the unstable markings.

Finally, note that these algorithms have recently been generalized to the class of "path-based" reward structures [380]. In these reward structures, impulses can be associated with individual activities or sequences of instantaneous activities, and the well-specified check algorithms must be modified to check that the distribution of impulse reward obtained when transitioning from one stable marking to another is invariant over different activity choices. A similar algorithm, in the context of more standard stochastic Petri nets, has been proposed by Ciardo and Zijal [85].

The next step in the construction is to determine an appropriate notion of state for the target model. This choice determines whether the resulting base model will be detailed or reduced.

9.3.2 Detailed base model construction

Detailed base models are constructed directly from the base model specification, without regard to the performability measures in question. Construction of base models from stochastic extensions to Petri nets is typically done in this manner, where most often (see [217, 5], for example), the state of the base model is taken to be the stable (also known as "tangible") markings of the network. More detailed notions of state, which keep track of the most recently completed timed activity and the stable marking that is reached when that activity completes, can also be considered [413]. Both of these notions of state are "detailed" in the sense that the exact marking of the specifying SAN is preserved in the definition of state for the target base model. In the case of analytic solution methods, this marking becomes part of the state of the resulting stochastic process; if simulation is

used, it is the notion of state that is employed for the generated trajectory when the simulator executes. Distinctions between the needs of analysis and simulation are discussed in the paragraphs that follow.

Construction for analytic solution If a SAN is well-specified and the possible stable markings are taken to be states, then their evolution with time is a stochastic process. More precisely, under these conditions, the *marking behaviour* of a SAN is the stochastic process

$$(R, T, L) = \{(R_n, T_n, L) \mid n \in \mathbb{N}\},$$

where T_n is the time of the nth timed-activity completion, R_n is the stable marking reached after the nth timed-activity completion, and L is the total number of state transitions of the process through time T_n (including the one made at T_0), given that $R_0 = \mu_0$ and $T_0 = 0$. Note that since separate random variables are used for the marking entered and the time of entry, the number of activity completions during an interval can be counted. Similarly, activity completions that do not change the marking of the network can be detected, since successive times of activity completions are recorded by T_n. Note that we have made no assumptions about the nature of this stochastic process, which may be Markov, semi-Markov, Markov regenerative, or even more general, depending on the activity time distributions chosen and the structure of the SAN and composed model.

When this amount of detail is not needed, the "minimal marking behaviour" can serve as a detailed base model. More formally, the *minimal marking behaviour* of a SAN with marking behaviour (R, T, L) is the stochastic process

$$Z = \{Z_t \mid t \in \mathbb{R}^+\},$$

where Z_t is the stable marking at time t. By ignoring activity completions that leave the marking unchanged, this behaviour is minimal in the sense that it has a minimum number of state transitions relative to the original marking behaviour. Of course, with such simplification, these ignored completions can no longer be detected. For many applications, such detection is not essential; indeed, the minimal marking behaviour is the stochastic process that is typically associated with stochastic Petri nets (see [328, 349, 5], for example).

The resulting marking and minimal marking behaviours are Markov (continuous-time, finite-state, time-homogeneous) if all the activity times

are exponentially distributed and, further, activities are reactivated often enough to ensure that their rates (if marking-dependent) depend only on the current state. It is important to note, however, that if the latter condition does not hold, then even if all the activity times are exponential, the minimal behaviour may not be Markov. This possibly surprising fact follows from the execution rules for SANs. In particular, recall that activity times are determined at activation time and may be marking-dependent. Therefore, depending on the nature of the specification, it may be that an exponentially distributed activity time depends on a *past marking* whose rate differs from that of the current marking; if this occurs, then Z is not Markov. Other stochastic Petri net definitions preclude this behaviour, implicitly assuming that rates are determined by the current marking of the net. While this is a reasonable restriction for the purpose of obtaining Markov behaviour (and is specifiable with SANs via reactivation functions), there are other conditions that can ensure that Z is Markov. Furthermore, if all activities are not exponential, it unreasonable to assume that rates "adjust" as markings change, since the delay behaviour of these activities is not memoryless. For additional discussion of this issue, along with a precise definition of the class of SANs that exhibit Markov behaviour, see [405].

Given that the behaviour of a SAN is Markov, a state-transition-rate diagram for its marking behaviour can be determined as follows (see [413] for a more precise description of this algorithm). First, each activity that may complete in the initial state is completed, generating potential new states that correspond to each possible next stable marking that may be reached from the initial marking. If a potential next state already exists, a non-zero rate from the original state to the reached state is added to the list of rates associated with the originating state. If the reached state is new, then it is added to the list of states that need to be expanded. A rate from the original state to the new state is then added to the list of rates for the original state. Generation of the state-transition-rate structure then proceeds by selection of states from the list of unexpanded states and repetition of the above operations. The procedure terminates when (i) there are no more unexpanded states (signifying that the state space is finite and has been completely determined) or (ii) the machine has reached its capacity and cannot store additional states. In the latter case, the state space is either infinite or too large to be computed.

Table 9.6: State-space sizes for detailed base model construction methods

	Marking behaviour				Activity-marking behaviour			
	L				L			
N	1	5	10	20	1	5	10	20
1	6	18	33	63	12	44	84	164
2	18	54	99	189	58	214	409	799
5	486	1,458	2,673	5,103	3,483	12,555	23,895	46,575
7	4,374	13,122	24,056	45,972	43,012	292,330	-	-
10	118,098	-	-	-	-	-	-	-

Marking behaviour of the faulty multiprocessor example To illustrate the marking behaviour associated with a SAN-specified base model, consider again the N-PE example addressed in Section 9.2, with particular reference to the composed SAN specification of its base model (described in Section 9.2.2). To restrict the example to a manageable number of states, let us suppose further that $N = 2$ (the number of PEs) and $L = 1$ (the capacity of the input queue). The result of applying the construction method described above to those choices is an 18-state process having the state-transition-rate diagram depicted in Figure 9.5. In the figure, circles represent possible marking states, and the numbers within the circles refer to the markings of places in those states. The two numbers on the first line refer to the marking of the first processor, the two on the second line the marking of the second processor, and the one on the third the marking of the buffer subnet. The marking of place *size* in the buffer subnet is not shown, since it is constant and equal to the capacity $L = 2$ for all states. The marking for each SAN is such that the places are in alphabetical order. To reduce the complexity of the figure, rates are not shown on the arcs, but they are generated via the algorithm just described. As can be seen from the figure, the marking behaviour for even a small instance of the example system is quite complex. Furthermore, as shown in Table 9.6, the state-space size grows rapidly as the values of N and L increase. Dashes in the table represent state spaces that were too large to generate.

In spite of this fairly rapid growth in size, the marking behaviour is not always sufficient to support a variable that depends on knowledge of *which* activity completed in a particular marking. This is due to the fact

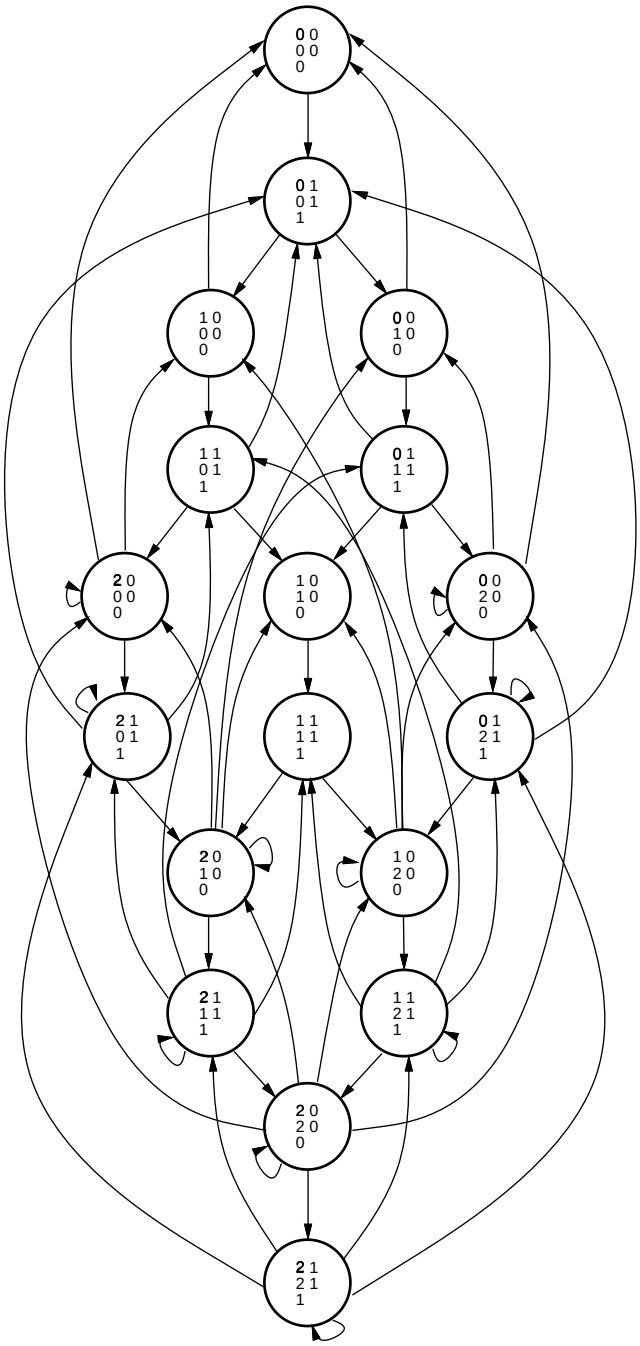

Figure 9.5: Marking behaviour for a 2-PE system with buffer capacity 1

that multiple timed activities may complete in a marking, all resulting in the same next stable marking. These multiple activities induce the same state transition in the marking process and hence cannot be distinguished.

If the marking behaviour is not sufficient, one must keep track of the most recently completed timed activity, as well as stable marking, as part of the state description. The resulting process, called the *activity-marking behaviour* of a SAN [405], is the stochastic process $(R, T, L) = \{R_n, T_n, L\}$, where T_n is the time of the nth timed-activity completion, R_n is the state reached after the nth timed-activity completion, and L is the total number of transitions of the process (including the one made at T_0), given that $R_0 = (\bigtriangledown, \mu_0)$ and $T_0 = 0$. States of the activity-marking behaviour are called *am-states* and are denoted by a pair (a, μ) where μ is a marking and a is a timed activity whose completion can result in μ. The activity-marking behaviour thus distinguishes states with respect to the timed activity that caused a marking to be reached, as well as the marking itself. An am-state is a possible stable marking of the network together with a timed activity that may result in that marking when it completes. $\bigtriangledown$ is a fictitious activity that is assumed to complete, bringing the network into its initial marking.

It can be shown [413] that the activity-marking behaviour of a network is Markov whenever the marking behaviour is Markov, and that the activity-marking behaviour supports all performance variables that can be defined using the reward variable specification method described in Section 9.2. However, this construction method results in base models that become extremely large as the size of a system grows. For example, Table 9.6 shows that the size of the resulting activity-marking space of the faulty multiprocessor model grows very rapidly as the number of processors and buffer stages is increased. Even if one considers only the marking behaviour, the state space grows quickly and becomes unmanageable for most realistic applications. This motivates the development of reduced base model construction methods, which we will discuss in the next section. Before doing so, however, we outline a method for simulating SAN-based reward models, using the notions of state just discussed.

Construction for solution by simulation In a simple approach to discrete-event simulation, each activity is an event type and every activity completion is an event (e.g., [405, 72, 461]). Such an approach must be employed if one wishes to preserve the complete marking or am-state dur-

ing execution. Simulation of stochastic extensions to Petri nets is typically done in this manner, although simulation of a generated stochastic process representation has also been proposed [137]. Use has also been made of the structural relationship between activities (transitions in GSPNs) [72] to minimize the number of event types that need to be checked for a change in status while retaining the detailed notion of state.

In such a procedure (for example, as implemented in [409]), detailed state trajectories are generated by repeatedly completing the earliest activity scheduled to complete and updating a single future event list, which keeps track of activities scheduled to complete. At every activity completion, a new stable marking for the model is chosen probabilistically from the set of next stable markings generated, using a procedure similar to that discussed in subsection 9.3.1 of this section. It is then necessary to check all scheduled events to see if they are still enabled in the new marking. The events that remain enabled must be kept on the future events list, unless the current marking is a reactivation marking for the activity. If it is, the activity is first removed from the list, and then added back to the list, with the new scheduled completion time. Finally, all activities that are not currently on the future event list must be checked to see if they are now enabled and hence should be added to the list.

While this procedure is simple, it does have some drawbacks. These stem primarily from the fact that as the number of activities (or transitions, in stochastic Petri net terminology) grows, the work that must be done to update the future event list upon each state change also increases. This work relates primarily to the fact that after a change in marking, a large number of activities need to be checked to see if their status (enabled or disabled) has changed. This problem can be solved by using ideas from reduced base model construction and using structural information to minimize the number of activities that must be checked upon each state change [72]. This will be discussed in the next subsection.

9.3.3 Reduced base model construction

Detailed base model construction methods can lead to base models that are extremely large if analytic solution methods are employed, or if simulation is used and a large number of events need to be considered during each state change. To avoid these problems, we make use of the SAN and composed

model structure to develop a less-refined notion of state that does not distinguish between replicate submodels. This notion of state is adaptive, and depends on the structure of the composed model tree and choice of performance variables. By exploiting symmetries present in the composed model (identified via the replicate node), the procedures generate base models for analytic solution that often consist of many fewer states than would otherwise be necessary. If simulation is used, the procedures reduce the number of events that must be processed upon each state change. This subsection will first describe the notion of state we employ, which is represented graphically as a "state tree". It will then describe how state trees are used in analytical and simulation-based construction methods.

State trees State trees [96, 406] are closely related to the graphical representation of composed SAN-based reward models (SBRMs), described in Section 9.2. Recall that a composed SBRM is a result of operations on SBRMs that may themselves be composed models. This composition is represented graphically by a tree. A notion of "state" can then be determined at each level of the tree structure. At a join operation, we keep a vector of "states" for each joined submodel. At a replicate operation, the number of replicas in each existing submodel "state" is kept. Finally, at the lowest level, the "state" of a SAN model is its normal marking. The complete state is then the impulse reward due to the last activity completion and the composed state formed as above. The notion of state thus preserves the identity of all submodels at a join node, but only keeps track of the *number* of submodels in particular states at replicate nodes.

State trees are a graphical representation of the marking portion of a reduced base model state. They consist of three types of nodes: *join nodes*, *replicate nodes* and *SAN nodes*. All leaves of a state tree are of type *SAN*. Nodes that are not leaves are of type *join* or *replicate*. A node of type *SAN* has a *sub-type* that relates the node to a particular SAN model. Each node in the state tree has a corresponding node in the composed model diagram. A node on a particular level on a state tree corresponds in type and level with a node on the composed model diagram. In both the state tree and the composed model diagram, nodes related to SANs are at the leaves of the tree.

Furthermore, each state tree node has associated with it a subset of the distinguished places of the corresponding node in the composed model

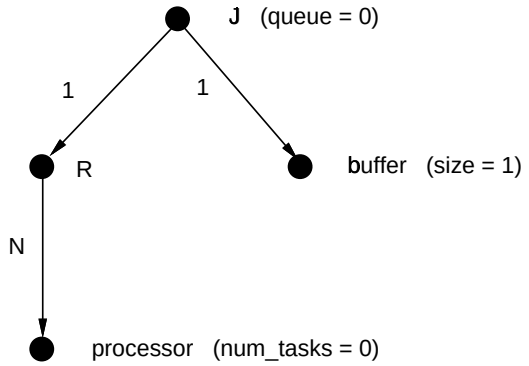

Figure 9.6: Initial reduced base model state

diagram. This subset consists of those places that are distinguished at the node, but not at its parent node. For convenience, we use the expression "a place at node i" to denote this relationship between nodes in the state trees and places in the composed SAN-based reward model. Given this assignment of places to nodes in the state tree, we define μ_i as the restriction of the global marking to the places at node i. The marking μ_i appears next to a node i and is ordered according to the alphabetical order of the places at that node.

Nodes in a state tree are connected by directed arcs. An arc that connects a parent node i to a node j has an associated integer $n_{i,j}$, where $n_{i,j}$ is the number of occurrences of the marking of the SBRM represented by node j at node i. By definition, each outgoing arc j from a join node i has $n_{i,j}$ equal to one, since one copy of each constituent SBRM is used in the join operation. Outgoing arcs from replicate nodes can have cardinality ranging from 1 to the degree of replication defined in the corresponding composed model node, and represent the number of replicas in a particular sub-state.

State trees for the faulty multiprocessor model To illustrate the use of state trees, consider again the N-processor faulty multiprocessor model introduced in Section 9.2. The state tree for the initial reduced base model state of this model, when the number of buffer stages is 1, is given in Figure 9.6. Note that the common place *queue* is at the top level join node (denoted by "**J**" in the figure), since it is common to all processor

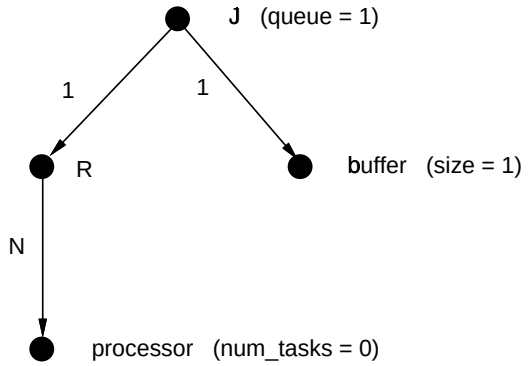

Figure 9.7: Second reduced base model state

.

replicas and the buffer. No places are at the replicate node (denoted with an "**R**"), since the only place common at that level (*queue*) is also common at the next higher level. The "**N**" on the outgoing arc from the replicate node denotes that all N replicas of the processor are in a single marking and the number of tasks at the processor is zero.

Only one activity, *arrival* in submodel buffer, is enabled in the initial marking. It will therefore eventually complete, resulting in the state tree shown in Figure 9.7. Note that the structure of the tree has not changed, only the marking of place queue at the join node. In this marking, the queue is full, so activity *arrival* is no longer enabled. Activity *access* is now enabled in all processor submodels, which "compete" for the task. Eventually, one of these activities will complete, resulting in the state tree shown in Figure 9.8.

Note that the structure of the tree has now changed, with the processor submodels split into two groups: $N-1$ models that remain in the idle state, and one model that now has a token in place *num_tasks*, indicating that it is processing a single task. It is important to note that the particular processor submodel in which activity *access* completed is not recorded, only the fact that activity *access* completed in some processor submodel. This observation gives insight into the cause of the reduction in number of states in a reduced base model. For a replicate node with N competitively enabled timed activities, this abstraction results in an N-fold reduction of possible next stable markings. If the composed model consists of multiple

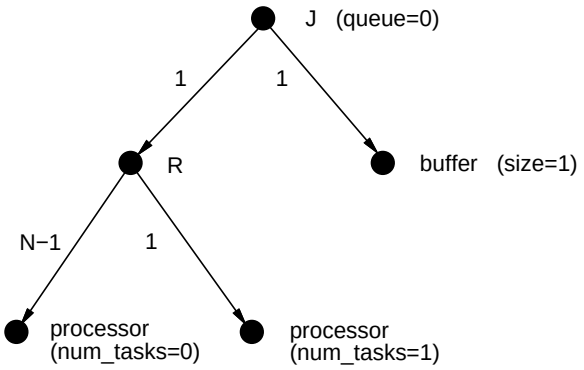

Figure 9.8: Third reduced base model state

replica nodes, arranged in a hierarchical fashion, the reduction can be even greater.

Construction for analytical solution A state tree is combined with the impulse reward of the most recently completed timed activity to form a *complete state* in a reduced base model. Note that the leaves of the state tree are submodels in a particular marking, and that the number of submodels in that marking can be determined by multiplying together the weights associated with each of the arcs on the path from the root of the tree to the submodel.

A reduced base model then can be generated, conceptually, by taking the reachable complete states as the states in a stochastic process and computing transitions between these states. The process to generate a reduced base model is very similar to that used to generate a detailed base model, except that the algorithm operates on complete states. The rates from one complete state to another are determined as described earlier in this section, except that we consider completions of activities from sets of *replica activities*, which are sets of identical activities in different replicas of a submodel in a particular marking. The SAN will transition from a state when the first activity in some set of replica activities completes, so the departure rate assigned to a set of replica activities in a leaf in the state tree is equal to the rate assigned to that activity in the SAN, multiplied by the number of submodels that are in the marking. Using these rates, and the generated state transition probabilities, transitions occur directly

from one reduced base model state to another.

Moreover, it can be shown (see [413]) that the resulting reduced base model has the following properties.

1. It is Markov if the corresponding detailed base model is a Markov process.

2. It supports the specified performance variable.

The first fact is established by formally specifying a mapping from each am-state of the model to its corresponding reduced base model state and showing that this mapping defines a strong lumping on the am-behaviour (activity-marking behaviour) of the model. Reduced base models thus produce exact results and can be solved using Markov methods whenever detailed base models can. It is important to note that while the proof of the Markov nature of the reduced base model relies on lumping arguments, the generation process described above does not. Instead, it generates the reduced base model directly from the composed SAN specification, without ever generating the activity-marking behaviour of the model.

The second fact follows from the variable-specification method described in Section 9.2. Since the impulse and rate rewards are specified in terms of a SAN, they are identical for all replica SANs defined by the replicate operation. Hence, all such SANs have identical rewards when in the same marking. This implies that the reward obtained while executing a composed SAN model depends only on the number of replica SANs in a particular marking, not the fact that a particular replica is in some marking.

Reduced base model for faulty multiprocessor model To illustrate analytic reduced base model construction, consider once again the faulty multiprocessor introduced in Section 9.2. Figure 9.5 gave the marking behaviour of this model when the number of buffer stages is one and there are two PEs; Figure 9.9 shows the reduced base model for the same model parameters, when all activities have identical impulse rewards. Replica submodels in identical markings are not distinguished (as per the state tree concept), and hence, a state can be labelled by the distinct markings of each SAN, and the number of SANs in each of these markings. As before, markings of SANs are shown as vectors of places in alphabetical order,

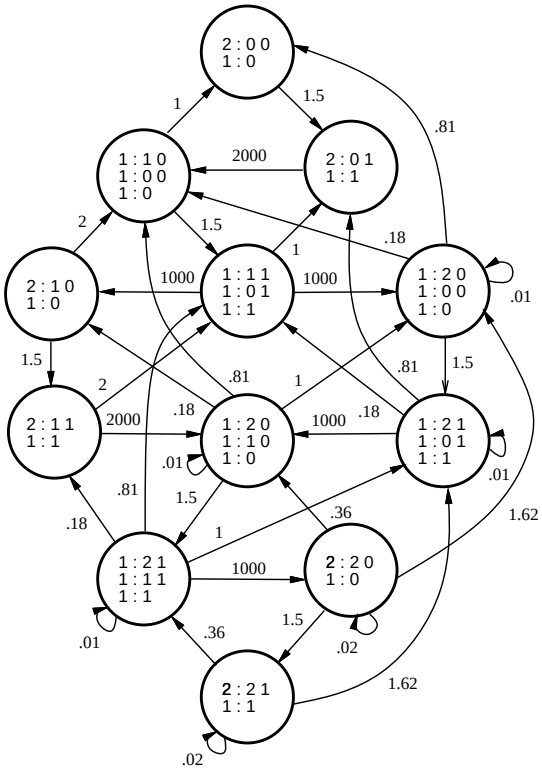

Figure 9.9: Reduced base model for 2-PE system with buffer capacity 1

Table 9.7: Detailed vs. reduced base model construction methods

	Marking behaviour				Reduced base model			
	L				L			
N	1	5	10	20	1	5	10	20
1	6	18	33	63	6	18	33	63
2	18	54	99	189	12	36	66	126
5	486	1,458	2,673	5,103	42	126	231	441
7	4,374	13,122	24,056	45,972	72	216	396	756
10	118,098	-	-	-	132	396	726	1,386
15	-	-	-	-	272	816	1,496	2,856
20	-	-	-	-	462	1,386	2,541	4,851
50	-	-	-	-	2,652	7,956	14,586	27,846
100	-	-	-	-	10,302	30,906	56,661	108,171
250	-	-	-	-	63,252	189,756	347,886	-
500	-	-	-	-	251,502	-	-	-

and the marking of place *size* in submodel buffer is not shown, since it is constant for all states. The number of submodels in a particular marking is given by the number before the colon on a line, and the vector after the colon represents the marking of that number of submodels. The numbers on the arcs are rates between states, calculated as described earlier.

The top state in the diagram thus represents the case in which both processors are idle, and the buffer is empty. The state below and to the left represents the situation in which one of the processor submodels has a single token in place *num_tasks*, one processor submodel has no tokens in place *num_tasks*, and there are zero tokens in place *buffer*. Since we require that reduced base models support variables that depend on activity completions, the impulse reward of the activity that most recently completed would normally also be part of the state label; however, since this impulse is the same for all activities, the label is not needed. Note that the reduced base model consists of 12 states, while the marking behaviour consisted of 18 states. The saving comes from the fact that we do not distinguish between the two processor submodels in the reduced base model.

The differences in state-space size are dramatic for larger systems, as shown in Table 9.7. As with Table 9.6, dashes represent state spaces that

were too large to generate. As can be seen from the table, generating detailed base models becomes impractical after only 10 processors, but models for up to 500 processors can be generated when reduced base model construction methods are used.

Construction for solution by simulation Recall that when detailed base model construction methods are used, activities in SANs are event types, and activity completions are events in the discrete event simulation. For large models, this leads to situations in which there are a very large number (possibly thousands) of event types, and the number of activities whose "status" (i.e., enabled or disabled) must be checked upon each activity completion is correspondingly large. In simulation, reduced base model construction methods make use of the state tree to perform future event list management more efficiently [406].

This efficiency is achieved by reducing the number of activities that are checked for changes in their status during the transition from one reduced base model state to another. The reduction can be achieved since all replicas of each activity will have the same status, since they all have their input places in the same marking. By definition, this will be the case for all replica activities at a particular leaf in the state tree, and hence we only need to check the status of one activity in a set of replica activities at a leaf in the state tree. More formally, we define a *representative activity* as an activity that "represents" the set of replica activities $a_1 \in A_1$, $a_2 \in A_2$, ..., $a_i \in A_i$, ..., $a_n \in A_n$, where A_i is the set of activities of the ith replica in a set of n replicas of a particular submodel in identical markings. Each representative activity is an *event type* in the new simulation technique, whereas activity completions are events.

During simulation, we operate on representative activities, instead of all activities, when performing future event list management. Status checks on activities are then reduced to a single check per set of replica activities for a set of submodels in identical markings. The events for each of these replica activities can be grouped into a list related to the representative activity. We call this list of sampled completion times a "compound event". More formally, we define a *compound event* e_a for representative activity a as the list of sampled completion times $\{t_1, t_2, \ldots, t_n\}$, where n is the number of activities represented by a. As argued in the previous section, n can be found by multiplying together the numbers on the arcs on the path

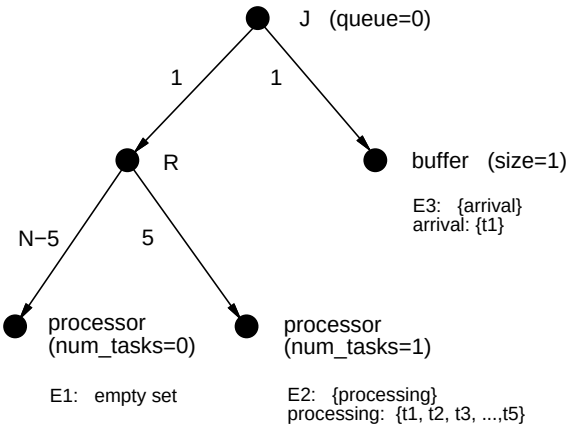

Figure 9.10: State tree with future event lists

from the root to the SAN under consideration. Using this information, it is possible to build compound events, each with n elements, from the list of future events for each set of submodels represented by a leaf node.

Simulation of faulty multiprocessor model The faulty multiprocessor model is useful for illustrating the use of compound events in simulation using state trees and multiple future event lists. Specifically, consider the state tree of Figure 9.10, which has each leaf node augmented with a list of compound events. This state represents the situation in which five PEs are processing a single task. In this state, there are three sets of compound events: $E1$, $E2$ and $E3$. $E1$ has no events scheduled, since there are no activities enabled in this state in this submodel. $E2$ has one compound event scheduled, which corresponds to activity *processing*. Because the integers at each arc on the route from the node at the highest level to the leaf have been multiplied, there are five replicas of type processor in the same marking; as a result, *processing* has five sampled completion times. $E3$ has one event scheduled, since the queue is now empty; hence, activity *arrival* is enabled. There is only one time scheduled for this event, since the submodel corresponding to this leaf is not replicated.

The algorithms to effect the transition from one state tree (augmented with future events lists) to another are quite complicated, since trees may have multiple replicate nodes, each of which may split or join in a given state change. A detailed description of the algorithms, which also make

use of structural information to detect activities that may have changed their statuses, can be found in [406].

9.4 Summary

As argued in the introduction, model specification and construction are important aspects of performability evaluation. With appropriate abstractions and tools, they facilitate the specification of complex behaviours that would be extremely difficult to represent at the state level. Early specification of performance measures is also very important, since they can guide the specification of the environment and object system models. Furthermore, a specification formalism of the type described can have a considerable influence on model construction. In particular, it suggests how base models can be tailored to the measures in question and, by identifying symmetries prior to construction, permits automated reduction of the resulting model. Accordingly, when compared with more conventional approaches, the base models obtained are typically smaller and easier to solve.

While a relatively simple system was used to illustrate some of these advantages, they have all been substantiated through application to a wide variety of more realistic examples. These have served to validate the utility of the specification/construction techniques and, when limits were reached, to motivate additional work. Early (*circa* 1986) techniques of this type were employed by Metasan; subsequently, all the methods described herein have been implemented in UltraSAN. Both tools have been used to evaluate various systems with respect to measures of performance and dependability as well as performability. Examples of the latter include the performability evaluation of computers (e.g., [16, 322]), communication systems (e.g., [15, 321, 375, 294, 255, 379]), databases (e.g., [408, 407]) and software systems (e.g., [295, 453]). The results reveal that stochastic activity networks are indeed an appropriate method for specifying complex system models. Moreover, they have shown that for many practical examples, reduced base model construction techniques result in base models that can be solved using readily available computing resources.

Acknowledgement

This work was supported in part by the Digital Equipment Corporation Faculty Program: Incentives for Excellence and NASA Grant NAG 1-1782.