From *Proceedings of the Second International Workshop on Performability Modeling of Computer and Communication Systems,* Mont Saint-Michel, France, June 28-30, 1993.

SPECIFICATION AND CONSTRUCTION OF PERFORMABILITY MODELS[*]

John F. Meyer[†] and William H. Sanders[‡]

[†]Dept. of Electrical Engineering and Computer Science
University of Michigan
Ann Arbor, MI 48104 USA
jfm@eecs.umich.edu
+1 (313) 763-0037

[‡] Dept. of Electrical and Computer Engineering
University of Arizona
Tucson, AZ 85721 USA
whs@ece.arizona.edu
+1 (602) 621-6181

ABSTRACT

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 aspects of performability modeling, 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.

Keywords: Performability, Model-based evaluation, Stochastic activity networks, Stochastic Petri nets, Reduced base model construction.

[*]This work was supported in part by the Digital Equipment Corporation Faculty Program: Incentives for Excellence.

I 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), generally, such modeling efforts must 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) which interact with the object system during its use.

Given an interacting set of physical and human resources, the distinction between 1) and 2) (although often not made explicitly) depends on which subset of resources is being investigated as to its ability to perform. These comprise the object system, which in this context, is often referred to simply as the "system." The environment is then comprised of those remaining resources that interact with the object system to an extent that they affect its performability in some appreciable sense.

Accordingly, the above distinction is actually determined by just what aspects of the total system structure and behavior 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 [1], 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 recent text by Kant [2] that 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 behavior 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 [3] for a more thorough treatment of this distinction) can obviously affect an system's ability to perform in a designated environment.

Certain measures of such ability are based on the generic concept of *dependability*, i.e., that property of a system which allows "reliance to be justifiably placed on the the service it delivers" (again see [3]). 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 [4, 5]), measures of *performability* are needed to simultaneously address issues of both performance and dependability. 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 at the other.) Such measures can thus account for degraded levels of performance which, 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" the solution of each. A model with this property is thus referred to as a *base model* of the total system. In other words, based on its probabilistic nature, one is able to determine the value or values of each specified measure. With regard to the object system, such modeling considerations are fairly well understood. On the other hand, the importance of realistic environment modeling is often underestimated in this regard, 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 being autonomous (relative to the measures in question). For example, much of traditional structure-based reliability evaluation presumes autonomy of this type, in which case the environment part of the total system model is null. In most contemporary applications, however, the environment is nontrivial. In such cases, its modeling calls for the same kind of care and attention that's typically devoted to the specification and construction of an object system model.

In the evolution of performability evaluation (see [6], for example), the initial emphasis was on solution methods (see [7, 8], for example, for good surveys on these methods). 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. Increased attention in this regard (see [7, 9] for surveys that address this issue) is also due, in part, to the growing complexity of the systems being evaluated. With this growth, there is 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 becomes infeasible due to state space blow-up.

Motivated by the above considerations, the intent of the discussion that follows is to describe 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 endeavors. This is done via descriptions of generally defined concepts and techniques, followed by illustrative, concrete examples that presume the use of stochastic activity networks as a specification method, and the software tool *UltraSAN* for construction of the models.

Generally, just what distinguishes specification from construction is somewhat arbitrary. In high-level terms, 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 most all of the important technical issues that arise in more generally defined settings.

The ensuing discussion is organized accordingly, with Section 2 devoted to performability model specification and Section 3 to the the construction of models so specified. Although performability solution methods lie outside the scope of the discussion, just 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 2 and 3 must rely on certain assumptions regarding solution capability. Section 4 concludes the presentation with a summary and some pointers to applications of the specification and construction methods described herein.

II Performability Model Specification

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

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 indeed 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, just how this aspect is measured (in a probability-theoretic sense) is typically implied by the nature of the model, e.g., a performance measure that's 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 modeling framework introduced in [4, 5], 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 [4, 5] 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 $\mathit{Perf}(B) = P[Y \in B]$. 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 this more specific meaning, e.g., it refers to an arbitrary Y-measure or, in contexts where Y is understood, a particular Y-measure. A more detailed discussion of S1) is treated in the subsection that follows (Section 2.1).

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, once constructed, X 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, this more general view (of what's being specified) is advantageous since, among other things, it allows the same specification to be used for both analytic and simulation model construction. Details concerning S2) are described in Section 2.2.

S3) is perhaps the the most difficult aspect of performability model specification, particularly if the techniques employed are to apply to differing 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 [4, 5] 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. Secondly, even if support is possible, when fully constructed, X may be too complicated (e.g., have too many states) to deal with effectively in this regard. Hence, our development (Section 2.3) assumes that S3) is specified directly in terms of S1) and S2).

In addition to the above considerations, the nature of the overall specification must be such that the subsequent construction phase is indeed feasible, e.g., the base model X can be characterized within practical limits on 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 admit to feasible and, hopefully, efficient means of solving the specified Y-measures.

A 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 [10], for example). Given that elements of A express reward then, quite naturally, Y can be referred to alternatively as a *reward variable* (see [11, 12], for example).

With this unified view of accomplishment, most 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 2), it remains only to formally specify the time instant at which or time interval over which reward is being quantified by Y. Here, in concert with how 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, the limit as a time instant or interval duration approaches infinity. Specifically (again as discussed in [11, 12], 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.

A reward variable in the first category, called an *instant-of-time variable*, represents the reward experienced at a designated time during the object system's use. A variable in the second category, referred to as an *interval-of-time* variable, represents the total amount of reward accumulated over a specified interval of time. The third category, called *time-averaged interval-of-time* variables, are similar to 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 1.

Interval-of-time variables can then be further classified according to the nature of the interval. The first type represents the total or time-averaged reward accumulated during some interval $[t, t + \ell]$. The second is obtained as a limiting version of the first, 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, initially, the system is operating under steady-state conditions. The final type 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 stating how Y is to be measured in a probability-theoretic sense. Just which measures of the latter sort make sense will typically depend on how reward is interpreted for Y. To illustrate some choices in this regard, consider the following example of a small total system. (This

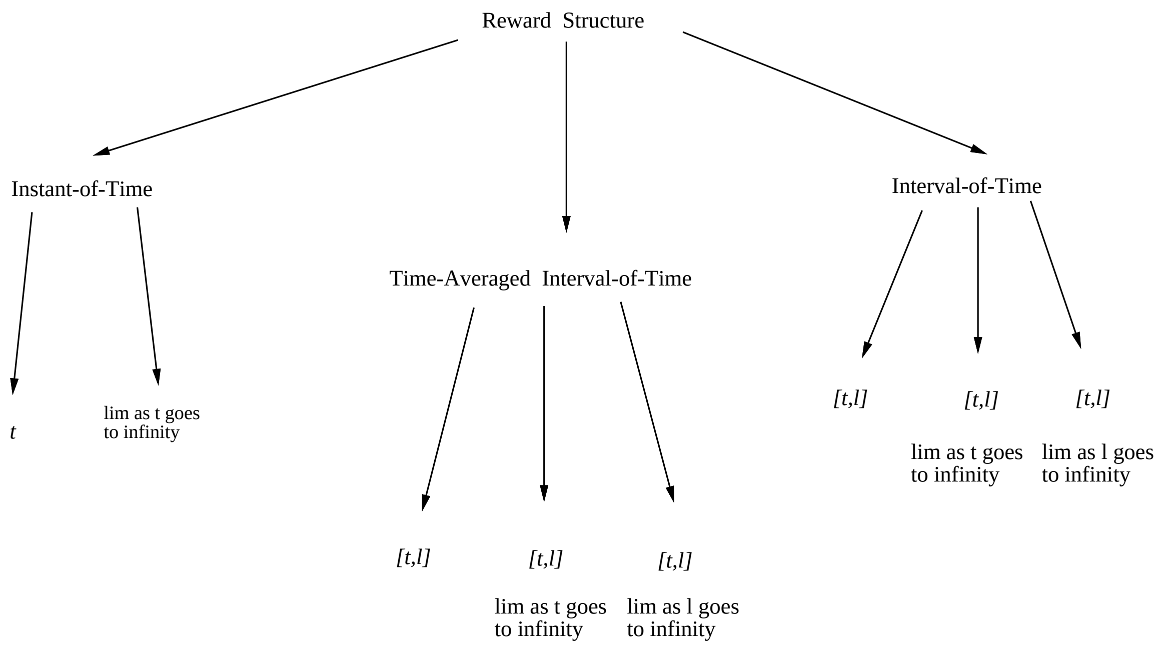
Reward Structure
Instant-of-Time
Time-Averaged Interval-of-Time
Interval-of-Time
t
lim as t goes to infinity
[t,l]
[t,l]
[t,l]
lim as t goes to infinity
lim as l goes to infinity
[t,l]
[t,l]
[t,l]
lim as t goes to infinity
lim as l goes to infinity

Figure 1: Variable Specification

example is nevertheless complex enough to exemplify much of what's discussed throughout the remainder of the presentation.)

Faulty Multiprocessor Example 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 which arrive as a Poisson process with parameter 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 PEs on a FIFO basis, where any "available" PE (the meaning of this will be explained momentarily) attempts to access the task at the head of the queue. Just which PE receives the task is assumed to be equally likely (among those PEs which are available).

All PEs have identical structure and behave as follows. A PE, having accessed a task when idle, can process it in exponentially distributed time with mean value $1/\mu$. Moreover, while busy with a single task, it can access a second task and, in turn, simultaneously complete the processing of both as if they were one, i.e., the two are then processed together at the single-task rate μ. Accordingly, we regard a PE as being *available* if it is idle or processing a single task; otherwise it is *unavailable*, i.e., it is engaged with two tasks and does not attempt to access a third.

In the case of double-task processing, however, all is not perfect. Specifically, there's 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 encountering errors being given by the probabilities:

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

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 jobs 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 PE and, in turn, will eventually enjoy an error-free completion via that PE. In the case where 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.

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

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

Letting $B = \{L\}$ be the accomplishment set of interest (a singleton set in this case), the associated measure is then the performability value

$$\mathit{Perf}(B) = P[V_t = L].$$

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, i.e., $E[V_t]$ (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 considers, instead, 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. If, further, tasks are assumed to arrive as a Poisson process, this 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 PE, 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 PE) 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 limits of these as $\ell \to \infty$. Performability measures associated with the above could range from simple mean values to full probability distributions.

B Base Model Specification

Let us turn now to the second ingredient, namely the specification of a stochastic process or simulation program 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, a specification of its 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 In the context of performability evaluation, the most popular vehicle for accomplishing this has been some form of stochastic Petri net (SPN) [13, 14], which, themselves, are typically called system 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's need to emphasize this distinction, we will refer to them 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 nets (SANs) [15, 16, 17]. Finally, in keeping with this choice, the tool we presume for construction purposes is *UltraSAN* (see [18]). Among other things, this permits discussion and illustration of hierarchical specification/construction techniques that are not implemented in an earlier SAN-based tool (METASAN [19]).

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 the measures in question. Instantaneous activities, on the other hand, represent actions which, for modeling 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 have an *enabling predicate* and *function* which, 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 (to the extent that's required to determine the base model it's specifying) 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 (known 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, markings in which there is at least one instantaneous activity enabled are *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 place 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* (i.e., the time at which an activity becomes enabled, or time of its completion, if it remains enabled), depending on their activity time distribution function. 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. A case of the activity chosen to complete is then selected based on the probability distribution for that set of cases. These two choices determine uniquely the next (stable or unstable) marking of the network, which is then obtained by executing the input gates connected to the chosen activity and the output gates connected to the chosen case. This procedure is repeated by considering the activities enabled in the new marking.

Activities may also be restarted, or *reactivated* [16], under certain circumstances. In particular, for each marking in which an activity may be activated, a set of *reactivation markings* can be defined. Then, if prior to completion, one of these markings is reached, the activity is *reactivated*, i.e., aborted and then immediately activated. This provides a mechanism for restarting activities, either with 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 Single-PE Faulty Multiprocessor To illustrate the use of stochastic activity networks in the specification process, we consider the faulty multiprocessor described in the previous section. More precisely, what we seek here is the specification of an analytic base model which can support solution of measures of the type illustrated

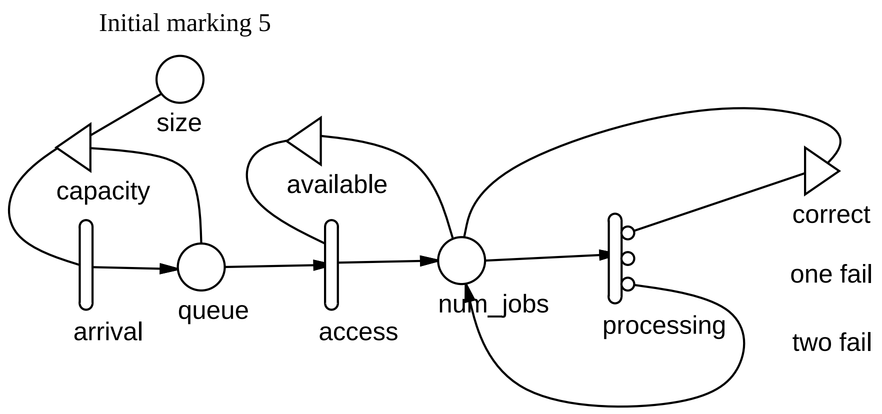

Figure 2: SAN Specification

Table 1: Activity Time Distributions

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

earlier in this section. Figure 2 contains a SAN specification of a 1 processor version of the faulty multiprocessor. In the figure, *arrival*, *access*, and *processing* are timed activities. *Arrival* and *access* have a single case, and *processing* has 3 cases. *Capacity* and *available* are input gates, and *correct* is an output gate. *Size*, *queue*, and *num_jobs* are places.

This SAN diagram, together with its activity time distributions (Table 1), case probabilities (Table 2), and input and output gate definitions (Tables 3 and 4, respectively), precisely specify the object system, workload, and fault environment described informally earlier. To see this, and illustrate the use of SANs in specifying performability models, we now describe the functioning of the model. In particular, incoming task arrivals are modeled 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 3, which specifies that the number of jobs in the queue must be less than the system 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

Table 2: Activity Case Probabilities

Activity	*Case*	*Probability*

processing	1	*if (MARK(num_jobs) == 1)*
		    return(1.0);
		else return(0.81);

	2	*if (MARK(num_jobs) == 1)*
		    return(ZERO);
		else return(0.18);

	3	*if (MARK(num_jobs) == 1)*
		    return(ZERO);
		else return(0.01);

Table 3: Input Gate Definitions

Gate	*Definition*
available	Predicate *MARK(num_jobs) < 2*

	Function
	  / do nothing */*
	  ;

capacity	Predicate */* has the buffer capacity been reached? */* *MARK(queue) < MARK(size)*

	Function
	  / do nothing */*
	  ;

Table 4: Output Gate Definitions

Gate	*Definition*
correct	*/* complete all jobs at the same time */* *MARK(num_jobs) = 0;*

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 to access the PE are represented by activity *access*, whose activity time (see Table 1) represents the time to assign a task to a PE. The enabling of this activity is controlled by input gate *available*, which enables the activity if the processor is "available." Activity *processing* represents the processing time of *tasks*, as described in the informal description of the model. Specifically, when processing completes, the action taken depends on whether one or two jobs were processed. This choice is reflected in the cases of activity *processing* (see Table 2), whose probability distribution depends on the marking of place *num_jobs*. If only one job 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_jobs*, because of the input arc from the place, and output gate *correct* is executed.

If there were two jobs that completed their processing, the action is probabilistic, as per the informal specification given earlier. In this situation, the three cases of activity *process* correspond to the three possible actions:

1. both jobs were processed correctly, and hence leave the system,
2. one job was processed correctly, and hence leaves, while the other must remain to be reprocessed, or
3. both jobs were processed incorrectly, and hence must remain to be reprocessed.

The probabilities associated with each of these actions is given in Table 2, and the actions are carried out by the arcs and output gate connected to the activity. For example, if two jobs complete processing, with probablity .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_jobs* 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 a SAN 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 hence make use of. 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 hierarchial 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

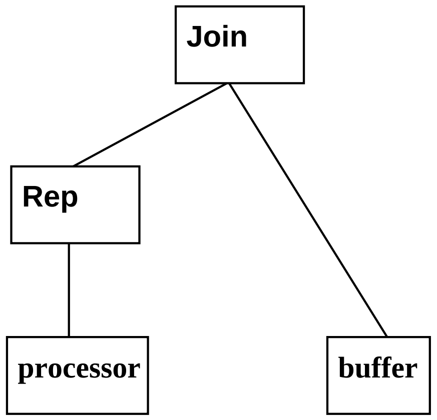

Figure 3: Composed N-PE Faulty Multiprocessor Specification

in the subsequent construction process. Formally, the resulting specification is known as a *composed SAN-based reward model* (composed SBRM) [20].

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, and although identical in their specification, can each have their 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 which 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 place in each of the lists is merged to form a single place, the second place is merged to form another place, and so on. Particular elements on the lists can be null, permitting the case where certain places are created 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 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 (**queue**) and the processing at a single PE (**processor**), as shown in Figure 4. Note the similarity with the single processor version described earlier. Since we now consider a N-processor version, we must replicate the processor specification N times. Replication is done via the “**Rep**” node in Figure 3. The **Rep** specifies that its child node (**processor**) should be replicated N times, holding the place *queue* common among all the replica submodels. 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 3. This node specifies that the place *queue*, in the N-PE submodel, is to be made common with place *queue* in submodel **buffer**.

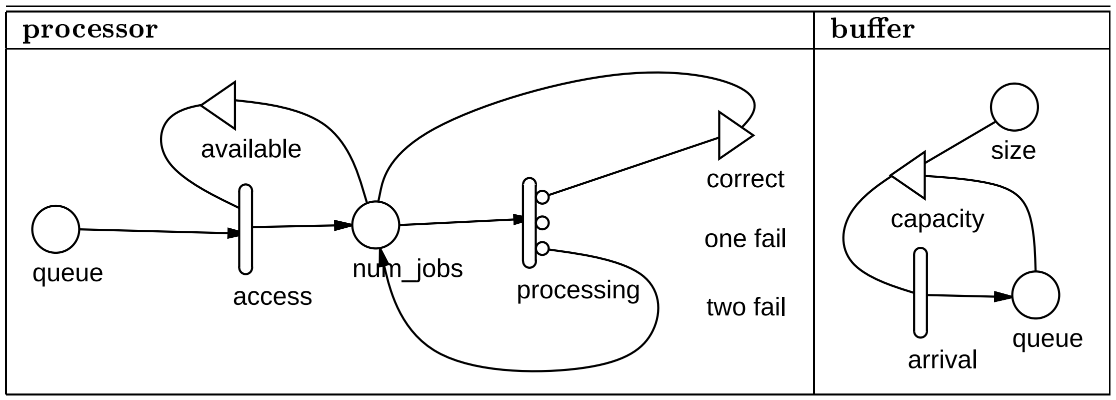

Figure 4: SAN Submodels in Composed Model Specification

C Reward Structure Specification

This section describes the link between a SAN specification and the measure specification method outlined earlier. The link is made using a *reward structure*, similar to that used for Markov reward models, but specified at 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 [11, 12] to the SAN level, where impulse rewards can be assigned to activity completions, and rate rewards can be assigned to particular numbers of tokens in places. A similar extension was made by Ciardo, Blakemore, and Chimento [21] in defining "stochastic reward nets."

Formally, as in [12], an *activity-marking oriented reward structure* of a SAN with places P and activities A is a pair of functions: $\mathcal{C} : A \to I\!\!R$ where for $a \in A$, $\mathcal{C}(a)$ is the reward obtained due to completion of activity a, and $\mathcal{R} : \mathcal{P}(P, I\!\!N) \to I\!\!R$ where for $\nu \in \mathcal{P}(P, I\!\!N)$, $\mathcal{R}(\nu)$ is the rate of reward obtained when for each $(p, n) \in \nu$, there are n tokens in place p. $I\!\!N$ is the set of natural numbers and $\mathcal{P}(P, I\!\!N)$ is the set of all partial functions between P and $I\!\!N$. An element $\nu \in \mathcal{P}(P, I\!\!N)$ is referred to as a *partial marking.* The marking is partial in the sense that natural numbers are assigned some 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 use the convention, in practice, that rewards associated with activity completions and partial markings are taken to be zero if not explicitly assigned otherwise. 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 1).

Reward Structure Specification for Faulty Multiprocessor To illustrate the use of reward structures in specifying measures, consider the reward structures in Table 5, for the composed N-PE faulty multiprocessor. Following the convention used in *UltraSAN*, we specify rate rewards using multiple predicate-function pairs. The interpretation of each

Table 5: Reward Structure Specification for Faulty Multiprocessor

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

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 reward contributed by each predicate-function pair. Furthermore, as per the functioning 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 reward from each individual SAN. Impulse rewards (although not illustrated here), specify fixed amounts of reward to be accumulated upon an activity's completion.

For example, consider the first reward structure in Table 5, which specifies a reward structure for an indicator random variable. This variable is used to determine the probability that the queue is not full (i.e., non-blocking). In this case, a reward is earned at rate 1 whenever the queue is not full, and at rate 0 when the queue is full. Using this structure to build an instant-of-time variable, we obtain the status of the queue at a particular time t, or in steady-state. The expected value of this variable is thus (since it is an indicator r.v.) the probability that the queue is not full. The second variable illustrates the use of a reward structure to specify the utilization of the N-processor system. Utilization is defined such that it is 1 (fully utilized) if all processors are processing at least one job, and k/N, if $k \leq N$ processors are processing at least one job. This definition is directly reflected in the reward structure of Table 5.

The third variable in Table 5 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 reward structure is used to obtain the number of jobs that are waiting in the queue, and is defined on the buffer submodel. Together with an instant-of-time variable, the reward structure results in a variable whose value takes on the the number of tasks in the queue at a particular time or in steady-state. The final variable illustrates the use of multiple predicate-function pairs in different SANs, to determine the number of tasks in the N-PE faulty multiprocessor system. Counting the number of tokens in place *queue*, and all the places *num_jobs* determines this number.

III Performability Model Construction

Given a specification of a performability model, its construction 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 either be by numerical analysis, or simulation. In the case of numerical analysis, model construction implies building a feasible stochastic process (one that supports the desired variables, and is solvable) from the specification. In the case of solution by simulation, it implies building an appropriate discrete event simulator, which when executed, produces the desired estimators of the specified performance variables. In this case, the simulator itself is the base model.

With either solution technique, 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 SAN and composed model, and supporting all possible variables specifiable within the specification method. At the other extreme, a base model can be the least refined model that supports the specified performance variable, and is solvable. Models which are at the detailed end of the spectrum are called, appropriately, “detailed base models,” and those near the other end of the spectrum are called “reduced base models.”

In the remainder of this section, we first describe algorithms that are necessary and common to both detailed and reduced base model construction. We will then discuss particular examples of detailed and reduced base model construction methods.

A Marking Change 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 the execution of SAN and composed model from the very detailed level a SAN executes at (discussed in the previous section), to a level of detail appropriate for constructing base model representations. As with most decisions regarding model construction algorithms, just what is “appropriate” is motivated by what we would like to know about the specified system. Examination of the performance variable specification method and, in turn, method for specifying reward structures suggests that we would like to preserve information regarding timed activity completions and times spent in stable markings. Information about particular sequences of instantaneous activities which might complete, or sojourns in unstable markings need not be preserved, since neither of these occurrences contribute to the value of a reward variable.

Thus, we are interested, at the most detailed level, in possible sequences of timed activity completions and reached stable markings. We would like to determine, for each stable marking, each timed activity which may complete in that marking, and each case that may be chosen, the probability distribution on possible next stable markings. After the completion of a timed activity and choice of a case, a next stable marking can be reached in two ways. In the first, execution of the gates and arcs connected to the chosen case results immediately in another stable marking, so no additional work is required. The probability associated with this possible next stable marking is simply the probability associated with the case.

The situation is much more complicated if an unstable marking is reached upon execution of the gates and arcs associated with the chosen case. In this situation, two questions must be addressed:

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 on 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 which may complete in these markings, and cases of these activities, we say that the SAN is *stabilizing* [17]. Unfortunately, it is not decidable if a SAN in a particular initial marking is stabilizing [17]. While this is unfortunate from a theoretical viewpoint, it is not particularly disturbing, since arbitrary length sequences of intervening instantaneous activity completions and resulting unstable markings could not be held in a computer with finite memory anyway. In practice, then, one just sets a limit on the number of subsequent instantaneous activities that will be allowed, and declares a SAN to be stabilizing if no sequences longer than this length exist.

Given that a SAN is stabilizing, then, the answer to the second question becomes important. In this case, the answer determines whether the next stable marking distribution is unique, or depends on instantaneous activity choices that are made while generating the next stable marking distribution. Recall that with SANs, choices made upon completion of an activity are made using cases, not by specifying a distribution on completion of one or more instantaneous activities enabled in a marking (a "random switch," in GSPN terminology [22]). This cleanly separates choices that the modeler is aware of, and should be probabilistically specified, and choices that he is not aware of, and should not matter. Thus, we would like the probability distribution on next stable markings to not depend on choices among concurrently enabled instantaneous activities. When such choices do not matter, we say that a SAN is *well specified* [17]. SANs that are well specified are completely probabilistically specified, in the sense that the chosen activity time distributions and case distributions completely specify the probabilistic behavior of the SAN. Since we wish to use probabilistic methods to evaluate a SAN's behavior, being well specified is a prerequisite to constructing a base model.

Fortunately, algorithms to determine whether a given SAN in some initial marking is well specified do exist, and are constructive, in the sense that while determining whether instantaneous activity choices matter, they determine that probability distribution, if it exists [17]. Informally, determining whether such choices matter involves enumerating all possible "paths," i.e. sequences of instantaneous activities and unstable markings, that are reached before reaching a stable marking, and grouping together those where the same choices among concurrently enabled instantaneous activities were made. Note that this grouping defines a compatibility relation on the set of possible paths, not an equivalence relation. Once the grouping is done, the set of possible next stable markings is computed for each compatibility set, by looking at the stable marking that results from each path's execution. The probability distribution within each compatibility set is then computed by summing the probability of each path that results in a particular stable marking within the set. Finally, the probability distributions of each compatibility set are compared with one another. If they are identical, then possible instantaneous activity choices that are made do not matter, and this common distribution becomes the next stable marking distribution for the original timed activity and case in the starting marking. If they are not identical, then the choices do matter, and the SAN is not well specified [17].

Given that a SAN is stabilizing and well specified, the above algorithms give us a method for jumping from stable marking to stable marking, recording the timed activity completion that caused the transition. Note that this computation can be done on the fly; there is no need to keeps sets of stable and unstable markings, which are later used to compute a state-transition matrix, as is done with GSPNs [22]. The next step in developing performability model construction methods is to determine an appropriate notion of state, and this choice is the basic distinction between detailed and reduced base model construction methods.

B Detailed Base Model Construction

Detailed base models are constructed directly from SAN and composed model representations, without regard to specified performance variables. Construction of base models from stochastic extensions to Petri nets is typically done this way, where most often, e.g., [9, 22], 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, that keep track of the most recently completed timed activity, as well as the stable marking that is reached when that activity completes, can also be used [20]. Both of these notions of state are "detailed," in the sense that they preserve the exact marking of the specified SAN in the constructed base model. In the case of analytical methods, this marking becomes part of the state of the generated stochastic process, and in simulation methods, it is the notion of state that is employed on the generated trajectory when the simulator executes. Both of these detailed model construction methods are discussed in the following.

Construction for Analytic Solution When a SAN is well specified, and the possible stable markings are taken to be states, the resulting process is known as the "marking behavior" of the SAN. The *marking behavior* is, more formally, a 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 stable marking 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 $R_0 = (\mu_0)$ and $T_0 = 0$. Note that since we maintain a separate random variable for markings entered and the time of entry, the number of activity completions during an interval can be counted. Similarly, activity completions which do not change the marking of the network can be detected, since successive times of activity completions are recorded by T_n.

When this level of detail is not needed, the "minimal marking behavior" can serve as a detailed base model. More formally, the *minimal marking behavior* of a SAN with marking behavior (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. This behavior is minimal, relative to the marking behavior, in the sense that it is the behavior with the minimum number of state transitions that corresponds to the marking behavior. Hence, activity completions that do not result in a change of marking cannot be detected. The minimal marking behavior is the behavior that is typically derived from a stochastic Petri net, e.g., [13, 14, 22].

The resulting marking and minimal marking behaviors are Markov when all the activities in the model are exponential, and activities are reactivated often enough that their rate (if marking dependent) depends only on the current state. It is important to note, however, that there are SANs with all exponentially distributed activities that are not 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 specific model, it may be the case that an exponentially distributed activity time depends on a *past marking* for its rate, and is, hence, not Markov. Other stochastic Petri nets definitions do not allow this behavior, and implicitly assume that rates adjust to follow the current marking of the net. While this is one reasonable behavior (which can be specified using reactivation functions) that will result in Markov behavior, there are others, as well. Furthermore, if all activities are not exponential, it is not reasonable to assume that rates "adjust" as markings change, since the delay behavior of these activities is not memoryless. For more information, and a precise definition of the class of SANs that exhibit Markov behavior, see [17].

Given the behavior of a SAN is Markov, we can generate the marking behavior, represented as a state-transition-rate diagram, as follows. First, each activity that may complete in the initial state is completed, generating a potential new state corresponding 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 which 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 marking behavior then proceeds by selecting states from the list of unexpanded states and repeating the above operations. The procedure terminates when there are no more unexpanded states (signifying that the state space is finite and has been generated), or when the capacity of the machine to store additional states is exhausted. In this case, the state space is infinite or too large to be computed. For a more precise description of this algorithm, see [20].

Marking Behavior of Faulty Multiprocessor Model To illustrate the marking behavior of a composed SAN-based reward model, consider again the faulty processor model. Suppose we are interested in the simple case where there is a single buffer, and two PEs. In this case, the marking behavior of the process has 18 states, and has the state transition rate diagram given in Figure 5. In the figure, circles represent possible marking states, and the numbers within the circles refer to the markings of places in that state. 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. The marking of place *size*, in the buffer submodel, is not shown, since it is constant and equal to the number of buffers in the system, for all states. The marking for each SAN is such that the places are in alphabetical order. Rates are not shown on the arcs to reduce the complexity of the figure, but are generated via the marking behavior generation process just described. As can be seen from the figure, the marking behavior of even a small configuration of the example system is quite complex. Furthermore, as shown in Table 6, it grows in size rapidly as the number of buffers and processors considered increases. Dashes in the table represent state spaces that were too large to generate.

In spite of this fairly rapid growth in size, the marking behavior is not always sufficient to support a variable that depends on knowing *which* activity completed in a particular marking. This is true, since there may be more than one activity which may complete in a marking that results in the same next stable marking. These multiple activities induce the same state transition in the marking process, and hence cannot be distinguished.

If this can happen, 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 is known as the "activity-marking behavior" of a SAN [17]. Formally, the *activity-marking behavior* is a 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 am-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 $T_0 = 0$. An "am-state" is a possible stable marking of the network

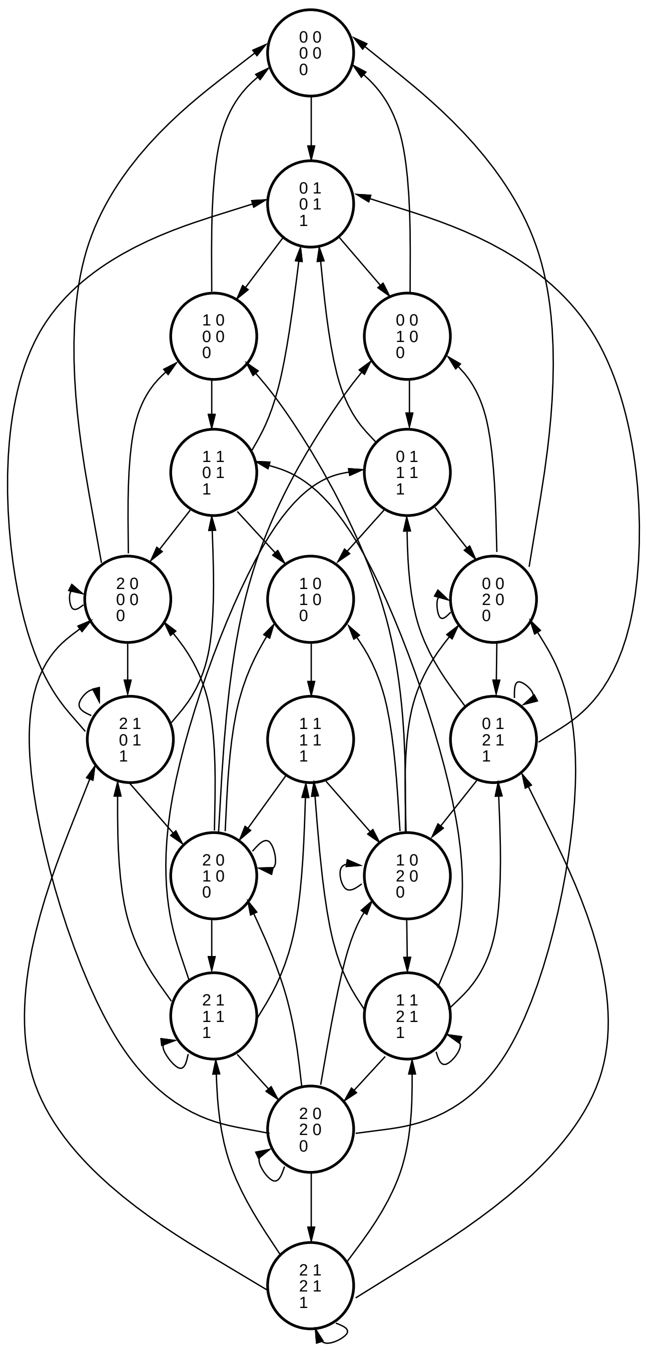

Figure 5: Marking Behavior for Faulty Processor with 1 Buffer and 2 PEs

Table 6: State Space Sizes for Detailed Base Model Construction Methods

#	Marking Behavior				Activity-Marking Behavior			
Proc.	Number of Buffers				Number of Buffers			
	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	-	-	-	-	-	-	-

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 [20] that the activity-marking behavior of a network is Markov whenever the marking behavior is Markov, and that the activity-marking behavior supports all performance variables that can be defined using the reward variable specification method described in Section II. However, this construction method results in base models that become extremely large as the size of a system grows. For example, Table 6 shows that the size of the resulting activity-marking space of the faulty multiprocessor model grows extremely rapidly as the number of processors and buffer stages is increased. Even if one considers only the marking behavior, the state space grows quickly, and becomes unmanageable for most realistic applications. This motivates the development of reduced base model construction methods, which will be discussed 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 A simple approach to discrete-event simulation, in which each activity is an event type and every activity completion is an event (e.g., [17, 23, 24]), must be employed if one wished to preserve the complete marking or am-state during 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 [25]. Use has also been made of the structural relationship between activities (transitions in GSPNs) [23] 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 [19]), detailed state trajectories are generated by completing the earliest activity scheduled to complete, iteratively, 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 A 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. Making use of structural information to minimize the number of activities that may have changed their status [23] together with reduced base model construction solves this problem, and will be discussed in the next subsection.

C Reduced Base Model Construction

Detailed base model construction methods can lead to base models that are extremely large, if analytic solution methods are employed, and where a large number of events need be considered during each state change, if simulation is used. To avoid these problems, we make use of the SAN and composed model structure to develop a less-refined notion of state, which 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 that often consist of many fewer states for analytic solutions. They also result in fewer events that must be processed upon each state change if simulation is used. This subsection will first describe the notion of state employed (and 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 [18, 26] are closely related to the graphical representation of composed SAN-based reward models, described in Section II. Recall that a composed SBRM is a result of operations on SBRMs that themselves may 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 *numbers 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* which 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. On both the state tree and the composed model

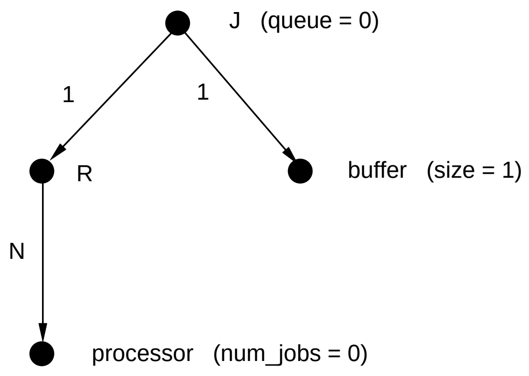

Figure 6: Initial Reduced Base Model State

diagram, nodes related to SANs are at the leaves of the tree.

Furthermore, each state tree node has an associated subset of the distinguished places of the corresponding node in the composed model diagram. These are the 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 characterize 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 there is one copy of each constituent SBRM used in the join operation. Outgoing arcs from replicate nodes can have cardinality ranging from one 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 processor model introduced in Section II. 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 6. Note the common place *queue* is at the top level join node (denoted by "**J**" in the figure), since it is common to all processor replicas and the buffer. No places are at the replicate node (denoted with a "**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, where the number of jobs 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 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 are "competing" for the job. Eventually,

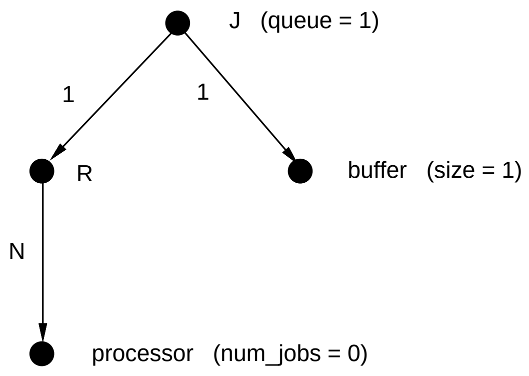

Figure 7: Second Reduced Base Model State

.

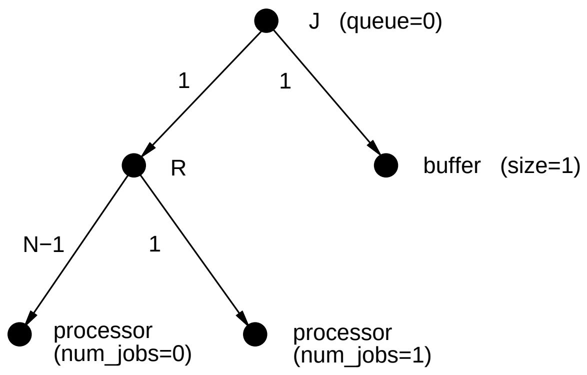

Figure 8: Third Reduced Base Model State

one of these activities will complete, resulting in the state tree shown in Figure 8.

Note that the structure of the tree has now changed, with the processor submodels split into two groups, $N-1$ models which remain in the idle state, and one model, which now has a token in place *num_jobs*, indicating that it is processing a single job. 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 replica nodes, arranged in a hierarchical fashion, the reduction can be even greater.

Construction for Analytical Solution A state tree, together with the impulse reward of the timed activity that most recently completed, form a *complete state* in a reduced base model. Note that the leaves of the state tree are submodels in a particular marking, and the number of submodels in that marking can be determined by multiplying the weights associated with 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* – that is, sets of identical activities in different replica 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, we can transition directly from one reduced base model state to another.

It can be shown (see [20]) that the resulting reduced base model:

1. is Markov whenever the corresponding detailed base model is, and

2. 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-behavior of the model. Thus, reduced base models 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 rely on lumping. Instead, it generates the reduced base model directly from the composed SAN specification, without ever generating the activity-marking behavior of the model.

The second fact is established by noting that, according to the variable specification method described in Section II, the impulse and rate rewards are specified in terms of a SAN, and are thus identical for all replica SANs defined by the replicate operation. Since all such SANs have identical rewards when in the same marking, 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 II. Figure 5 showed the marking behavior of this model when the number of buffer stages is one and there are two PEs; Figure 9 shows the reduced base model for the same model parameters, when all activities have identical impulse rewards. In the figure, states are notated as follows. Replica submodels in identical markings are not distinguished (as per the state tree concept), and hence, a state can be labeled 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, 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, and were calculated as described earlier.

The top state in the diagram thus represents the case where both processors are idle, and the queue is empty. The state below and to the left represents the situation where one

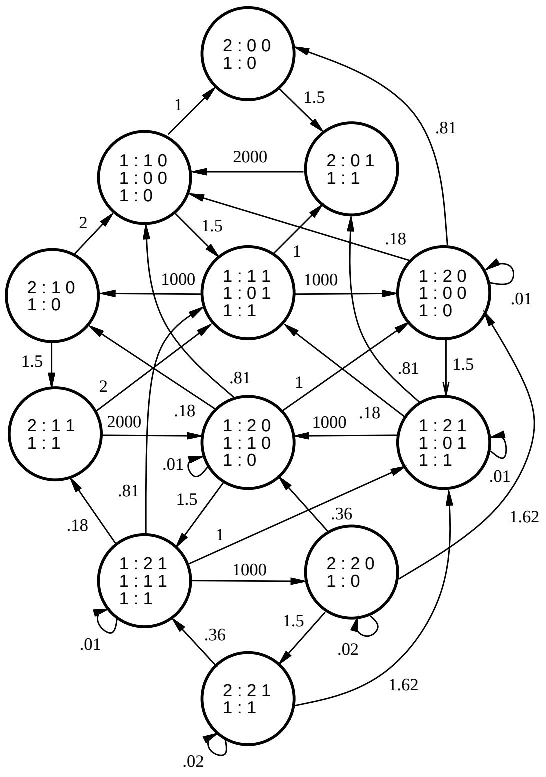

Figure 9: Reduced Base Model for Faulty Processor with 1 Buffer and 2 PEs

Table 7: Detailed vs. Reduced Base Model Construction Methods

#	Marking Behavior				Reduced Base Model			
Proc.	Number of Buffers				Number of Buffers			
	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	-	-	-

of the processor submodels has a single token in place *num_jobs*, one processor submodel has no tokens in place *num_jobs*, and there are zero tokens in place *buffer*. Normally (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 also be part of the state label, but 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 behavior consisted of 18 states. The savings 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 7. As with Table 6, dashes represent state spaces that were too large to generate. As can be seen from the table, generating detailed base models become impractical after only ten processors, but models for up to 500 processors can be generated when using reduced base model construction methods.

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 where there are a very large number (possibly 1000's) 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 more efficiently perform future events list management [26].

This efficiency is achieved by reducing the number of activities that are checked for a change 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 which have their input places in the same marking, will have the same status. By definition, this will be the case for all replica activities at a particular leaf in the state tree, and hence we only need check the status of one activity in a set of replica activities at a leaf in the state tree. More

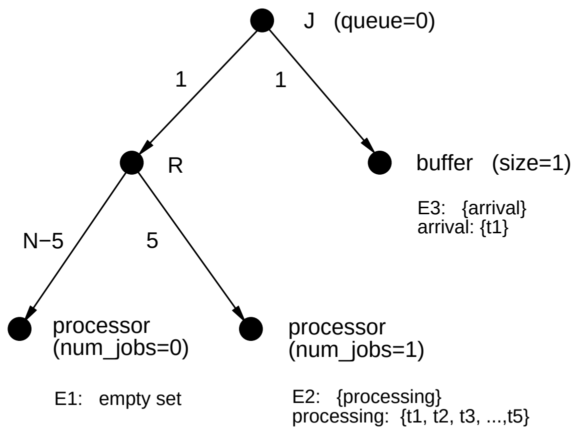

Figure 10: State Tree with Future Event Lists

formally, we define a *representative activity* as an activity that "represents" the set of replica activities $a_1 \in A_1$, $a_2 \in A_2$, $\ldots$, $a_i \in A_i$, $\ldots$, $a_n \in A_n$, where A_i is the set of activities of the i^{th} 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 the numbers on the arcs on the path from the root to the SAN under consideration. Using this information, compound events can be built, 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 to illustrate the use of compound events in simulation using state trees and multiple future event lists. Specifically, consider the state tree of Figure 10, which has each leaf node augmented with a list of compound events. This state represents the situation where 5 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, corresponding to activity *processing*. Since there are five replicas of type processor in the same marking (the result of multiplying the integers at each arc on the route from the node at the highest level to the leaf), *processing* has five sampled completion times. $E3$ has one event scheduled, since the queue is now empty, and hence, activity *arrivals* 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 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 which may have changed their status, can be found in [26].

IV Summary

As argued in the introduction, model specification and construction are important aspects of performability evaluation. With proper abstractions and tools, they make possible the specification of complex behaviors, which would be extremely difficult to represent at the state level. Early specification of performance measures is also very important, since they can guide development of the specification of the environment and object system. Furthermore, a carefully chosen specification formalism can greatly aid in the construction process, by suggesting base models that are tailored to the measures in question or that exploit symmetries that are made evident by the specification method. Use of such techniques can result in base models that are smaller, and can be solved more efficiently than if they were not used.

While the simple example presented herein illustrated some of these advantages, the "proof" comes through application to realistic, practical examples. Such applications can validate the utility of specification and construction techniques and (when limits are reached) serve to motivate new work. Early (circa 1986) specification and construction methods were implemented in METASAN, and all methods described in this paper have been implemented in *UltraSAN*. Both tools have been used for a wide variety of performance, dependability, and performability evaluations. In particular, with regard to performability evaluation, they have been used to evaluate both computer (e.g., [27, 28]) and communication (e.g., [29, 30, 31, 32]) systems. The results show that stochastic activity networks are indeed an appropriate method for specifying complex system behaviors, and that, for many practical examples, reduced base model construction methods generate base models that can be solved using readily available computing resources.

REFERENCES

[1] D. Ferrari, *Computer Systems Performance Evaluation*, Prentice-Hall, 1978.

[2] K. Kant, *Introduction to Computer System Performance Evaluation*, McGraw-Hill, 1992.

[3] J.-C. Laprie, editor, *Dependability: Basic Concepts and Terminology*, volume 5 of *Dependable Computing and Fault-Tolerant Systems*, Springer-Verlag, 1992.

[4] J. F. Meyer, "On evaluating the performability of degradable computing systems," in *Proc. 8th Int'l Symp. on Fault-Tolerant Computing*, pp. 44–49, Toulouse, France, June 1978, IEEE Computer Society Press.

[5] J. F. Meyer, "On evaluating the performability of degradable computing systems," *IEEE Trans. Computers*, vol. C-29, no. 8, pp. 720–731, August 1980.

[6] J. F. Meyer, "Performability: A retrospective and some pointers to the future," *Performance Evaluation*, vol. 14, no. 3/4, pp. 139–156, 1992.

[7] E. de Souza e Silva and H. R. Gail, "Performability analysis of computer systems: From model specification to solution," *Performance Evaluation*, vol. 14, no. 3/4, pp. 157–196, 1992.

[8] K. S. Trivedi, J. K. Muppala, and S. P. Woolet, "Composite performance and dependability analysis," *Performance Evaluation*, vol. 14, no. 3/4, pp. 197–216, 1992.

[9] B. R. Haverkort and K. S. Trivedi, "Specification techniques for Markov reward models," *Discrete Event Dynamic Systems: Theory and Application*, vol. 3, no. 2/3, , July 1993.

[10] R. A. Howard, *Dynamic Probabilistic Systems, Vol. II: Semi-Markov and Decision Processes*, Wiley, 1971.

[11] W. H. Sanders and J. F. Meyer, "A unified approach for specifying measures of performance, dependability, and performability," in *Dependable Computing for Critical Applications*, pp. 87–94, Santa Barbara, CA, August 1989.

[12] W. H. Sanders and J. F. Meyer, "A unified approach for specifying measures of performance, dependability, and performability," in *Dependable Computing for Critical Applications*, pp. 215–238, Springer-Verlag, 1991.

[13] M. K. Molloy, "Performance analysis using stochastic Petri nets," *IEEE Transactions on Computers*, vol. C-31, pp. 913–917, September 1982.

[14] S. Natkin, *Reseaux de Petri Stochastiques*, PhD thesis, CNAM-PARIS, June 1980.

[15] A. Movaghar and J. F. Meyer, "Performability modeling with stochastic activity networks," in *Proceedings of 1984 Real-Time Systems Symposium*, Austin, Texas, December 1984.

[16] J. F. Meyer, A. Movaghar, and W. H. Sanders, "Stochastic activity networks: Structure, behavior, and application," in *Proceedings of International Workshop on Timed Petri Nets*, pp. 106–115, Torino, Italy, July 1985.

[17] W. H. Sanders, *Construction and Solution of Performability Models Based on Stochastic Activity Networks*, PhD thesis, University of Michigan, 1988.

[18] J. Couvillion, R. Freire, R. Johnson, W. D. Obal II, M. A. Qureshi, M. Rai, W. H. Sanders, and J. Tvedt, "Performability modeling with *UltraSAN*," *IEEE Software*, vol. 8, no. 5, pp. 69–80, September 1991.

[19] W. H. Sanders and J. F. Meyer, "METASAN: A performability evaluation tool based on stochastic activity networks," in *ACM-IEEE Fall Joint Computer Conference*, pp. 807–816, Dallas, TX, November 1986, IEEE Computer Society Press.

[20] W. H. Sanders and J. F. Meyer, "Reduced base model construction methods for stochastic activity networks," *IEEE Journal on Selected Areas in Communications*, vol. 9, no. 1, pp. 25–36, January 1991.

[21] G. Ciardo, A. Blakemore, P. F. J. Chimento, J. K. Muppala, and K. S. Trivedi, "Automated generation and analysis of Markov reward models using Stochastic Reward Nets," in *Linear Algebra, Markov Chains, and Queueing Models*, C. Meyer and R. J. Plemmons, editors, volume 48 of *IMA Volumes in Mathematics and its Applications*, Springer-Verlag, 1993.

[22] M. Ajmone Marsan, G. Conte, and G. Balbo, "A class of generalized stochastic Petri nets for performance evaluation of multiprocessor systems," *ACM Trans. Computer Systems*, vol. 2, no. 2, pp. 93–122, May 1984.

[23] G. Chiola, "Simulation framework for timed and stochastic petri nets," *Int. Journal in Computer Simulation*, vol. 1, pp. 153–168, 1991.

[24] A. A. Törn, "Simulation nets, a simulation modeling and validation tool," *Simulation*, vol. 45, no. 2, pp. 71–75, August 1985.

[25] J. B. Dugan, *Extended stochastic Petri nets: Applications and analysis*, PhD thesis, Duke University, 1984.

[26] W. H. Sanders and R. S. Freire, "Efficient simulation of hierarchical stochastic activity network models," *Discrete Event Dynamic Systems: Theory and Application*, vol. 3, no. 2/3, pp. 271–300, July 1993.

[27] B. E. Aupperle, J. F. Meyer, and L. Wei, "Evaluation of fault-tolerant systems with nonhomogeneous workloads," in *Proc. 19th Int'l Symp. on Fault-Tolerant Computing*, pp. 159–166, Chicago, IL, June 1989, IEEE Computer Society Press.

[28] J. F. Meyer and L. Wei, "Influence of workload on error recovery in random access memories," *IEEE Trans. Computers*, vol. C-37, no. 4, , April 1988.

[29] B. E. Aupperle and J. F. Meyer, "Fault-tolerant BIBD networks," in *Proc. 18th Int'l Symp. on Fault-Tolerant Computing*, pp. 306–311, Tokyo, Japan, June 1988, IEEE Computer Society Press.

[30] J. F. Meyer, K. H. Muralidhar, and W. H. Sanders, "Performability of a token bus network under transient fault conditions," in *Proc. 19th Int'l Symp. on Fault-Tolerant Computing*, pp. 175–182, Chicago, IL, June 1989, IEEE Computer Society Press.

[31] K. H. Prodromides and W. H. Sanders, "Performability evaluation of CSMA/CD & CSMA/DCR protocols under transient fault conditions," *IEEE Transactions on Reliability*, vol. 42, no. 1, pp. 116–127, March 1993.

[32] L. M. Malhis, W. H. Sanders, and R. D. Schlicting, "Analytic performablity evaluation of a group-oriented multicast protocol," Technical Report PMRL 93-13, The University of Arizona, June 1993. Submitted for Publication.