From *IEEE Software*, Special Issue on Software for Performance Analysis, September 1991. Preliminary Version. Version as appeared in journal was edited further.

PERFORMABILITY MODELING WITH *UltraSAN*[*]

J. Couvillion, R. Freire, R. Johnson, W. D. Obal II, M. A. Qureshi,
M. Rai, W. H. Sanders, and J. E. Tvedt

Department of Electrical and Computer Engineering
The University of Arizona
Tucson, AZ 85721

(602) 621-6181
usan-project@ece.arizona.edu

ABSTRACT

Stochastic extensions to Petri nets have received growing attention during the past decade as a model for evaluating the performance, dependability, and performability of computer hardware, software, and networks. Their formal structure permits solution by analytic means in many cases. When this is not possible, they can facilitate the automatic generation of a simulation program to estimate system behavior. This paper describes an X-window based software tool for evaluating systems that are represented as stochastic activity networks, a variant of stochastic Petri nets. The tool, known as *UltraSAN*, incorporates the results of recent research to significantly reduce the size of state space that is considered for analytic solution, as well as the number of event types that are considered in simulation. Both of these results suggest that the tool will be able to solve significantly more complex models than previously possible. Throughout the paper, a simple local area network model is used to illustrate the concepts, user interface, and model construction and solution methods implemented in the package.

Keywords: Performance Evaluation, Dependability Evaluation, Performability Evaluation, Stochastic Petri Nets, Stochastic Activity Networks

[*]This work was supported in part by the Digital Equipment Corporation Faculty Program: Incentives for Excellence, Intel Corporation, Bell Communications Research, US West Advanced Technologies, the UA Foundation and the Office of the Vice President for Research, and by an equipment grant from AT&T.

I Introduction

Modern computer systems and networks must be carefully evaluated in order to insure acceptable performance and dependability during their use. This evaluation is needed in many phases of the life-cycle of a system, including design, implementation, and modification. While engineering "rules of thumb" are sufficient for simple systems, they cannot accurately predict the performance of today's complex systems, which may be made up of many components, each complex in itself. Because of this, more sophisticated methods have been developed to predict system performance and dependability. Broadly speaking, two distinct approaches have evolved: *testing* and *modeling*.

While evaluation via testing has been used successfully to evaluate small to moderate size systems, its sole use on large and complex systems is technically difficult, and economically infeasible. In particular, the size and complexity of modern computer software, hardware, and networks limit the use of testing as an evaluation method. Not only does testing require that the system be implemented, thus precluding testing during design, it may require that the use of the system under test be interrupted. Modeling is a promising alternative to testing. It can, in many cases, avoid the complexity problem by retaining important system information (relative to the desired measures of performance) while abstracting unnecessary details. Furthermore, the effect of design changes can be investigated easily by changing the model. However, traditional performance and dependability modeling techniques are limited in two ways: 1) they require that the performance and dependability of a system be evaluated separately, disregarding any dependencies that exist between these two aspects of system behavior, and 2) they can only be applied to small to moderate size systems. While these limitations may not be severe for simple systems, they do make the evaluation of complex systems, such as computer systems and networks, difficult by traditional means.

The concept of "performability" [1] allows us to overcome the first limitation. Informally, performability quantifies a system's "ability to perform" in the presence of faults. This ability is expressed formally by probabilities. A specific performability measure is obtained by defining just what "performance" means for the evaluation in question. The choices here are virtually limitless, including the binary-valued performance measures (success or failure) considered in reliability evaluation, discrete-valued variables such as the number of successful message transmissions during a fixed period of time, and continuous-valued performances measures such as packet transmission rates, product yield probabilities, and

times to carry out a particular operations.

While the concept of performability allows the behavior of complex systems to be described in general terms, it in itself does not suggest methods by which a performability model may be constructed. Although simple models can be constructed at the process level, realistic systems require more sophisticated representations to account for performance and dependability in a unified manner. Queueing networks, although useful in the context of strict performance evaluation, do not suffice since their structure is fixed and cannot account for fault-related behavior. Stochastic extensions to Petri nets (SPNs), on the other hand, permit the representation of both performance and dependability characteristics, depending on the interpretation given to tokens in the model.

One particular variant of SPNs, known as "stochastic activity networks (SANs)," has been used successfully to evaluate a wide variety of computer systems and networks. The utility of SANs in evaluating complex distributed systems was proved by their implementation in a software package known as METASAN[1] [2]. METASAN allowed testing of both the ease of representation of large systems as stochastic activity networks and the efficiency of current solution methods in solving the models. The experience gained has shown that stochastic activity networks may be used to evaluate a wide variety of systems. In particular, applications have included computer-communication networks [3, 4, 5, 6], computer systems [7], transaction processing systems, and automated manufacturing systems.

While each of these studies illustrated the utility of SANs in representing realistic systems, they also pointed out inefficiencies of traditional stochastic Petri net solution methods. Informally, the problem is, when traditional analytic solution techniques are used, that the complexity of the solution (in this case the size of the resulting stochastic process) grows extremely rapidly as the size of a system increases, quickly resulting in a situation where solution is no longer practical. Similarly, if the intended solution is via simulation, the run time necessary to obtain a statistically-significant solution increases rapidly, resulting in simulation run times that are unacceptable.

This motivated new work on construction and solution methods for SANs, and resulted in several innovations in model construction and solution methods for stochastic Petri nets. In particular, the work included:

1. Definition of a class of SAN-level performability variables that are common to both

[1] METASAN is a registered Trademark of the Industrial Technology Institute.

analytical and simulation-based solution methods [8],

2. Development of methods that make use of the performance variable choice and structure of the SAN to greatly reduce the size of the stochastic process required for an analytic solution [9], and

3. Development of methods that make use of the performance variable choice and structure of the SAN to reduce the number of activities that must checked on each state change and, hence, speed up solution by simulation [10].

In order to be useful, however, these techniques must be implemented in program form. This paper describes a recently developed software tool for evaluating systems represented as stochastic activity networks. The tool, known as *UltraSAN*, incorporates the results cited above in an easy-to-use, graphical, X-window based package. In the discussion that follows, Section 2 reviews the modeling framework used by the tool, including a brief discussion of stochastic activity networks, composed models, and associated performability variables. Section 3 presents an overview of the organization of the tool, while Section 4 discusses the user interface. This is followed, in Section 5, by a discussion of the implementation of model construction and solution methods, for both analysis and simulation. Finally, Section 6 offers conclusions and suggestions for future work. Throughout the paper, a simple local area network model is used to illustrate the concepts, user interface, and model construction and solution methods implemented in the package.

II Modeling Framework

Before describing *UltraSAN*, we describe the modeling framework on which the tool is based. This framework defines, in precise terms, the models and performability variables available to a user of the package.

A Stochastic Activity Networks

The models, known as *stochastic activity networks* (SANs) [11, 12, 13], are a stochastic extension to Petri nets. Structurally, they consist of *activities, places, input gates,* and *output gates.* Activities are of two types: *timed* and *instantaneous.* Timed activities represent activities of the modeled system whose durations impact the system's ability to perform.

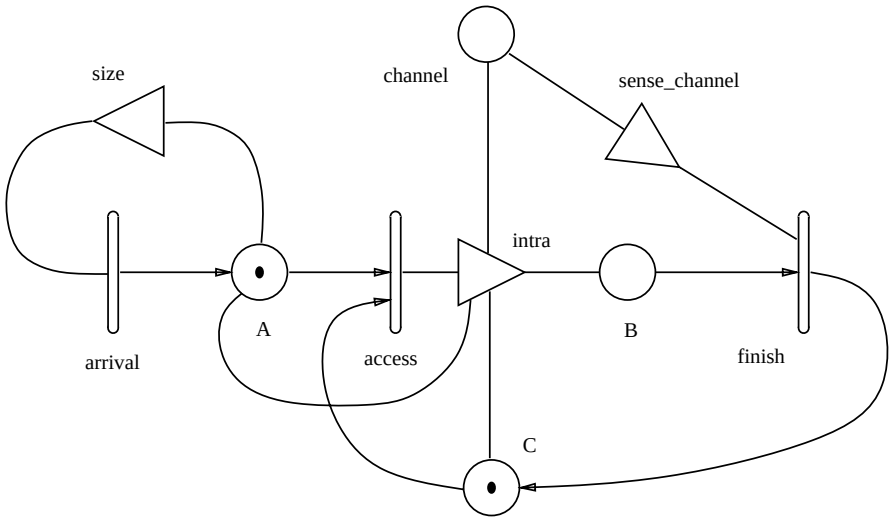


Figure 1: Station Submodel

To illustrate SAN components, we consider a simple model of a CSMA/CD station[2] In this model, as shown in Figure 1, timed activities are used to represent the following events: arrivals of frames from a higher-level protocol to be transmitted (activity *arrival*, in Figure 1), attempts to access the channel (activity *access*), and the time to transmit packets (activity *finish*). Instantaneous activities, on the other hand, represent system activities which, relative to the performability variables in question, complete in a negligible amount of time. *Cases* associated with activities (not used in this example, but represented as small circles on one side of an activity) permit the realization of uncertainty concerning what happens when an activity completes.

Places are used to represent the "state" of a system (e.g., places A, B, C and *channel*) and may contain *tokens* (e.g., the small black dot in A is a token). For example, the number of tokens in A represents the number of frames awaiting transmission. When an activity *completes*, one token is removed from each of the places directly connected to the input of the activity (e.g., place C is connected to the input of activity *access*), and one token added to each of the places directly connected to the output of the activity (e.g., place A is connected to the output of activity *arrival*).

Input gates and *output gates* permit greater flexibility in defining enabling and completion rules. Input gates have *enabling predicates* and *functions*, while output gates have only

[2]Note that is this is a simple model, intended only to illustrate the functioning and use of SANs. More detailed CSMA/CD models exist.

Gate	Type	Enabling Predicate	Function
size	input	$MARK(A) < 2$	identity

intra	output	–	if (MARK(*channel*)==0) {
			MARK(*channel*) = 1;
			MARK(*B*) = 1; }
			else if (MARK(*channel*)==1) {
			MARK(*channel*) = 3;
			MARK(*C*) = 1;
			MARK(*A*) = MARK(*A*) + 1; }
			else if (MARK(*channel*)==2) {
			MARK(*C*) = 1;
			MARK(*A*) = MARK(*A*) + 1; }
			else if (MARK(*channel*)==3) {
			MARK(*C*) = 1;
			MARK(*A*) = MARK(*A*) + 1; }

sense_channel	input	MARK(*channel*)==2 || MARK(*channel*)==3	MARK(*channel*) = 0;

Table 1: Gates for Station Submodel

functions. The enabling predicate can be either true or false and, as seen in that which follows, controls the enabling of an attached activity. The function describes an action (change in marking) that will occur upon completion of the activity. Activities are *enabled* if there is at least one token in each of the places directly connected to the activity and if the predicate of each associated input gate is true (i.e., *holds*). For example, in the CSMA/CD model (see Table 1), input gate *size* holds, and hence activity *arrival* is enabled, when there are less than two tokens in place A. Note that the keyword *MARK(place)* is used to represent the number of tokens in *place*. Input gate *size* is thus used to model a finite queue on the interface between the media access and logical link control sublayers.

Output gates, together with directly connected output places, are used to specify the action to be taken upon completion of an activity. For example, the output gate (*intra*) connected to activity *access* represents the result of the attempt to access the channel. In this case, different numbers of tokens in place *channel* are used to represent the status of the bus; zero tokens represents an idle bus, one token an unpropagated frame (a frame which, due to the propagation delay of signals on the media, cannot yet be detected by all stations on the bus), two tokens a propagated frame, and three tokens a collision. Propagation of

the frame on the bus is a global action, and hence handled by a separate SAN submodel, which will be discussed later. As can be seen in Table 1, the status of the bus and station model is possibly changed by gate *intra* upon completion of activity *access*, depending on the marking of place *channel* at the time.

B Variable Specification

The formalism used to represent variables at the stochastic activity network level is an extension of the idea of a "reward model." Traditional reward models consist of three components: a stochastic process, a reward structure, and a performance variable defined in terms of the stochastic process and reward structure. The 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. We extend this idea to the SAN level, where impulse rewards can naturally be assigned to activity completions, and rate rewards can be assigned to particular numbers of tokens in places.

Performance, dependability, and performability variables can then be easily defined in terms of these rewards. In particular, we have defined a family of variables, distinguished by the intervals of time on which they depend (see Figure 2). Three categories of variables are distinguished: *instant-of-time variables*, which represent the status of the SAN at either a particular time t or in steady state, *interval-of-time variables*, which represent the total accumulated reward obtained from executing the SAN for a particular interval of time, and *time-averaged interval-of-time variables*, which represent the time-averaged accumulated reward obtained from executing the SAN for a particular interval of time. For the second and third categories, three types of variables are considered. The first type represents the total or time-averaged reward accumulated during some interval $[t, t+l]$. The second type corresponds to an interval of length l as t goes to infinity, and is useful in representing the reward that is accumulated during some interval of finite length in steady state. The final variable type corresponds to the total or time-averaged reward accumulated during an interval starting at t and of length l as $l \to \infty$.

Together, these variables and the reward structure discussed previously give us the ability to represent many traditional and non-traditional measures of performance, including queueing time, queue length, processor utilization, steady-state and interval availability, reliability, and productivity. In addition, if some high-level measure of "worth" is defined, it can be expressed as a particular reward structure of this type. For more details, and

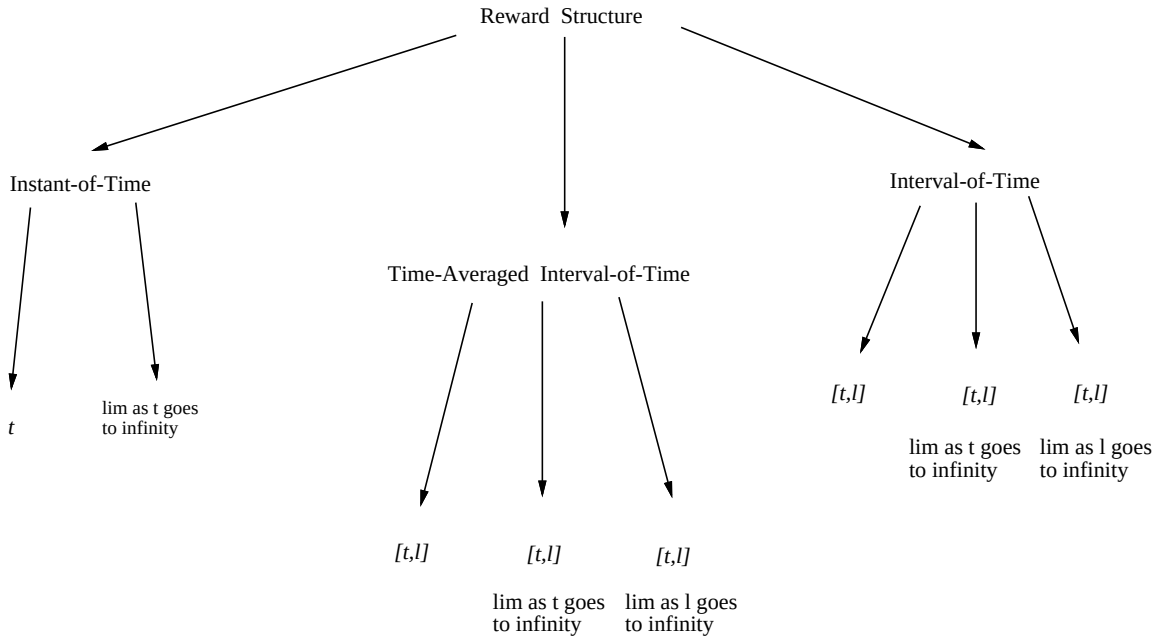


Figure 2: Types of Variables Considered

examples of the construction of several variables using this structure, see [8].

C Reduced Base Model Construction

Given the SAN and variable specification formalism just described, it is possible to investigate construction of small stochastic process representations that permit solution for a specified variable or variables. This is known as *model construction*. More precisely, it is the process of identifying a performance variable and determining a base model (stochastic process) that permits solution of that variable. *Model solution*, in turn, is the determination of the probabilistic nature of the selected performance variables.

Excellent progress in the development of model construction methods has been made by taking a more general view in the base model construction process, in which knowledge of the structure of the network and performance variable is used to determine the notion of state to use in the resulting stochastic process [9]. Traditional model construction methods for stochastic Petri nets and extensions do not use this approach. They typically obtain the base model stochastic process by choosing the reachable stable markings of the network to be the states of the process. To distinguish between these two approaches, a stochastic

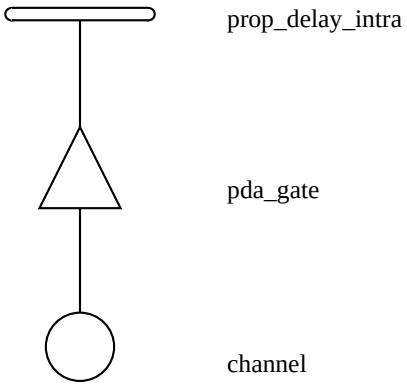
process which supports a large class of variables is referred to as a *detailed base model*, while a stochastic process constructed specifically to support a designated performance variable is a *reduced base model*.

The reduced base model construction methods developed in [9] are applicable for a restricted, but common class of stochastic activity networks and performance variables. This class includes stochastic activity networks that have some replicated components, such as processors in a multiprocessor or nodes in a computer network, and variables that are regular in the sense that they assign equal rewards to identical events and markings in different replicated components. These methods abstract unnecessary information from the base model without rendering it unsolvable.

Similarly, work has been done by the authors to speed up simulation. In discrete event simulation, a "future events list" is typically used to keep track of the timings of events which may occur some time in the future. The approach in [10] uses the structure of the SAN to manage a dynamically varying number of future event lists. In particular, since activities within replicated submodels are equivalent with respect to both their enabling conditions and associated rewards, they can be considered as a single event type in the simulation. This greatly reduces the amount of time spent updating the future event list.

To use this approach, a complete (or "composed") model is built from one or more SAN submodels using "replicate" and "join" operations. Formally, the resulting model is known as a *composed SAN-based reward model* (SBRM). The *replicate* operation replicates a SAN and associated reward structure a certain number of times, holding some subset of its places, called its "distinguished places" in [9], common to all resulting submodels. It is through these distinguished places that the replicated submodels interact. Each replica will have values for the impulse and rate rewards specified as in the original submodel. The replicate operation allows one to construct composed models that consist of several identical component submodels.

The combination of several different submodels is accomplished using the *join* operation. 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 construction 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. We allow particular elements on the lists to be null, permitting the case where certain places are created from a proper



Gate	Type	Enabling Predicate	Function
pda_gate	input	MARK(*channel*)=1	MARK(*channel*)=2;

Activity	Distribution Type	Parameter (Rate)
prop_delay_intra	exponential	20

Figure 3: Network Submodel

subset of the submodels joined. This extension (from that in [9]) presents no problem with either solvability or support, but provides more flexibility in defining composed models.

In terms of our running LAN example, a model for the entire network is built by first defining a SAN submodel to represent the propagation of signals on the bus, as shown in Figure 3. We then use the replicate and join operations just defined to construct a complete composed model. In particular, Figure 4 shows a composed model for a CSMA/CD network with n identical stations. The leaf nodes represent the individual submodels, together with their reward structures (each C is a function representing the impulse rewards for the model; each R represents the rate rewards). The station submodel is replicated n times with the place *channel* held common among all replicas. This submodel is then joined to the network submodel by joining the place named *channel* in each submodel to form a single new place. The resulting composed model can then be solved by both analysis and simulation, as will be seen in the following sections.

III *UltraSAN* Organization

UltraSAN was developed for the UNIX operating system for DEC, Sun, and AT&T

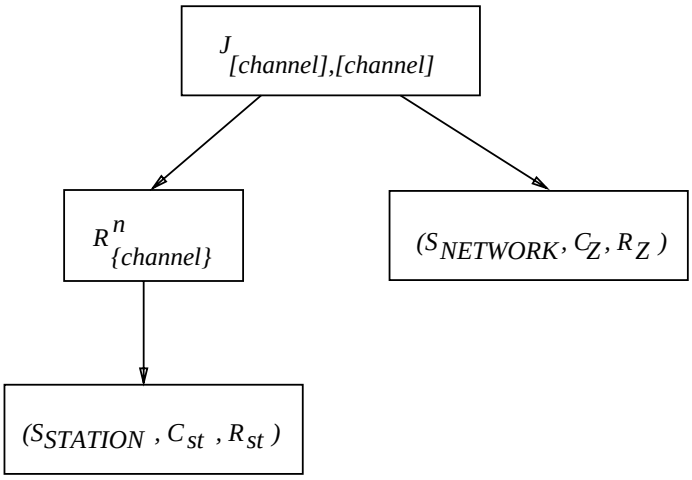


Figure 4: Composed Model

workstations, using the X window interface for portability. The software was developed in a modular manner, as shown in Figure 5. In this figure, the boxes represent executable programs, and the ovals represent data files which are either generated by, or serve as input to, the constituant tools. Broadly speaking, the tools can be classified as either model specification or model construction and solution tools. The model specification tools were written in C++ and utilize the InterViews object oriented library interface to X. The model construction and solution routines were written in C. In all, *UltraSAN* consists of about 70,000 lines of code.

Model specification is often the most tedious part of the modeling process, so every attempt was made to simplify and expedite this task in *UltraSAN*. Three main tools are used for model specification: the SAN editor, the composed model editor, and the performance variable editor. The SAN editor expedites the specification of SAN submodels by allowing the user to enter the SAN graphically, almost exactly as he would draw it on paper. The composed model editor is used to draw a tree representing the connection of the submodels. Each node in the tree is a *replicate* or *join* operation, and each leaf a submodel. Finally, the performance variable editor is used to specify reward variables. Rewards may be specified for activity completions or may be based on specific markings of the model.

The model construction and solution modules require no direct interaction from the user. All of the information needed for the solution of a model is generated from the user's specification. If an analytic solution method is chosen, the reduced base model

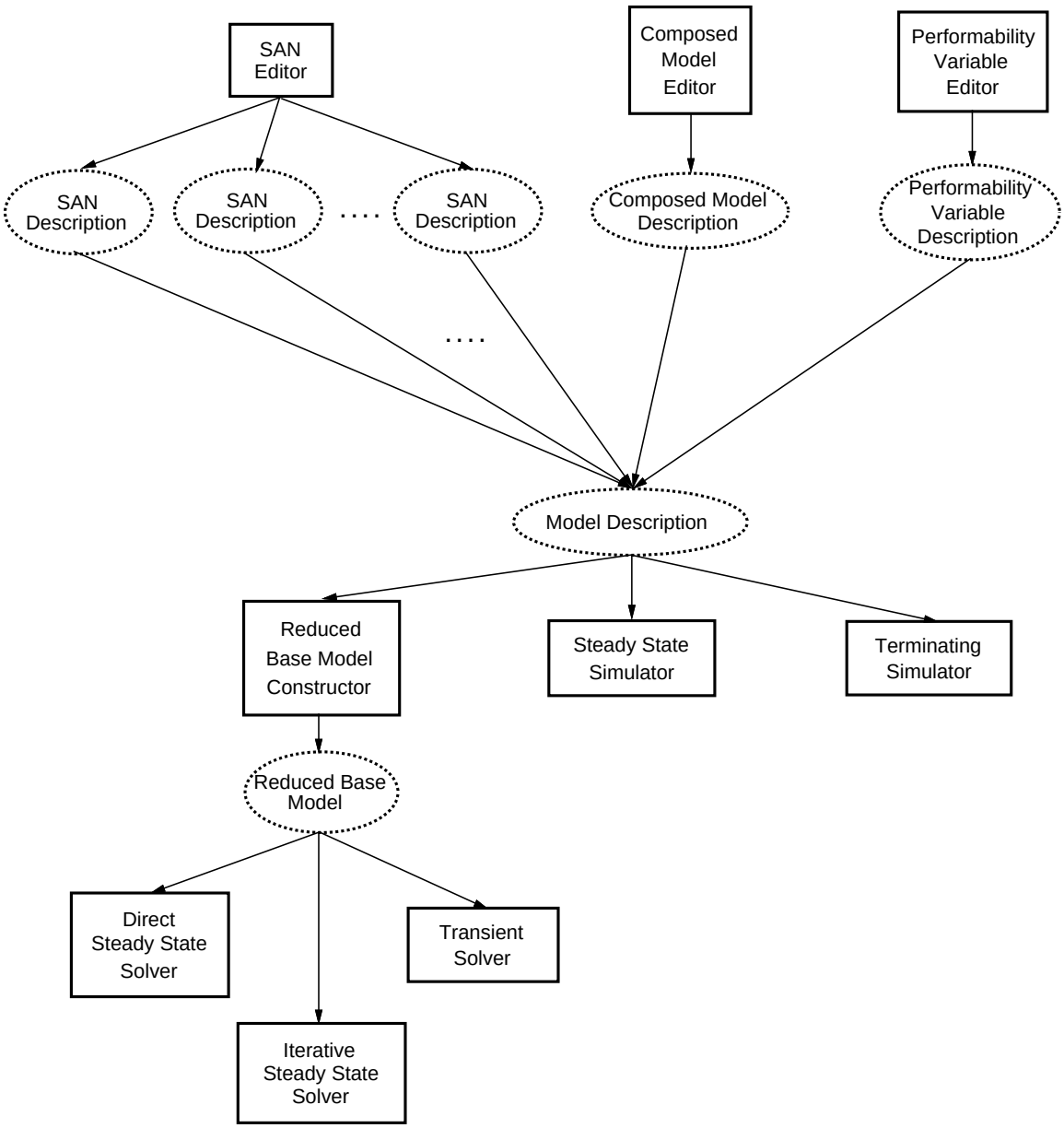


Figure 5: Organization and Data Flow of *UltraSAN*

constructor is used to generate the reduced base model for the model. Following the model construction, the selected solver is executed on the generated process. Two analytic solvers for steady-state (or long-run) variables are provided: a direct solver based on the LU-decomposition technique and an iterative solver based on successive over-relaxation. If a value for a performance variable at some particular time is desired, the transient solver may be employed. If simulation is preferred or model characteristics make analytic solution impossible, the simulation solvers can be used. The terminating simulator solves for variable values at specific points or intervals of time, while the steady state simulator functions exactly as its name implies. Each of these solvers will be discussed in more detail in Section V.

UltraSAN was constructed in a modular manner, thereby facilitating addition or replacement of package components. The major components are coupled only through a set of files with a specified format, as shown in Figure 5. By adhering to these specifications, it is possible to add to or replace any of the main components in the tool. For example, since the output of the reduced base model constructor is known, additional analytical solvers are easily added to the package. If alternative specification tools were desired, any or all of the current tools could be replaced as long as the description files retained the same format.

IV User Interface

The user interface provides a simple method for describing the system to be evaluated as well as the performance, dependability, and/or performability measures to be determined. The system is described as one or more stochastic activity network modules, organized in a hierarchical manner using replicate and join operations. Using this method, the user refers to a given modeling study as a "project," which may consist of one or more "subnets." Each subnet is a stochastic activity network together with a performance variable specification. These subnets are then combined together to form a "composed model."

At the highest level, a user interacts with the system through three user interfaces: the SAN editor, the composed model editor, and the variable editor. Each of these editors is described in more detail in the following.

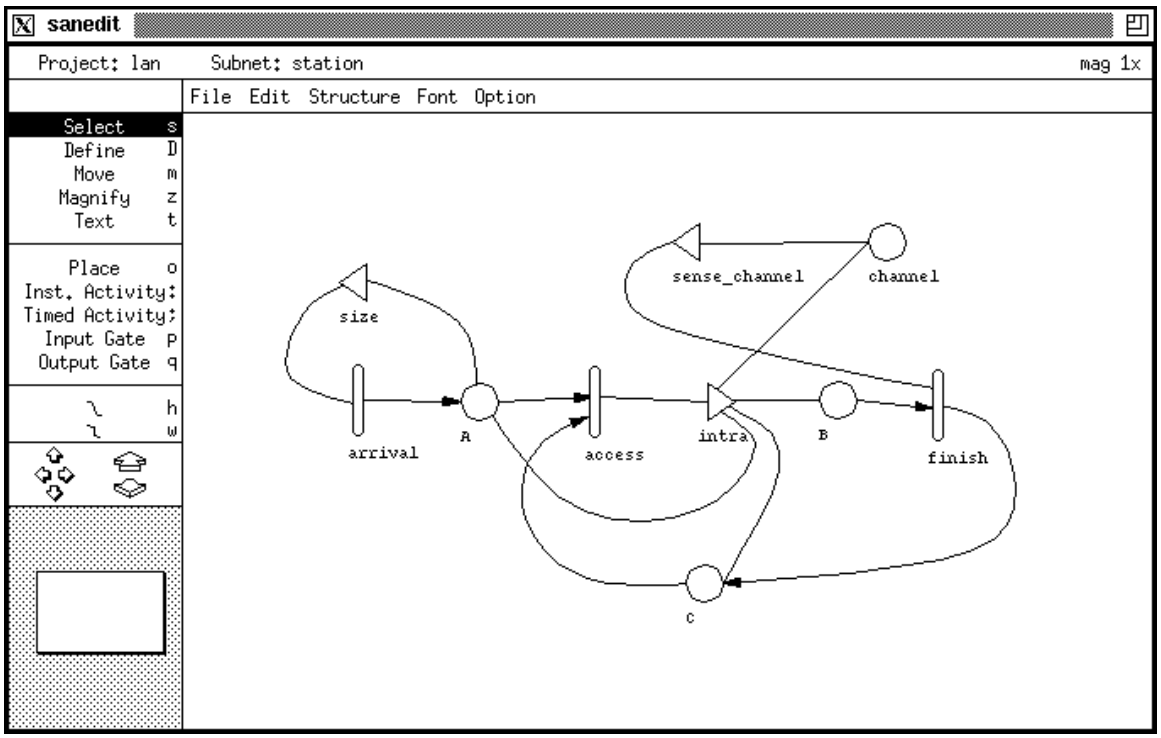


Figure 6: SAN Editor

A SAN Editor

The SAN editor (`sanedit`) is used to specify subnets. Input to the SAN editor is graphical. Figure 6 depicts the editor representation of the station submodel described earlier. As shown here, a user of `sanedit` simply draws a SAN corresponding to the subsystem he wishes to model, using the predefined model primitives on the left of the figure. Zooming and panning are supported (via the magnify button and the zoom and pan arrows near the lower left corner of the screen). Attributes for each of the model components can be edited by clicking the mouse on the define tool and then clicking on the component itself. When this is done, the appropriate form will pop up. For example, if the user selected the activity *finish*, the form shown in Figure 7 would appear on the screen. As can be seen, attributes for an activity can be easily specified. In particular, the activity time distribution for the activity can be selected by clicking on a particular distribution. When this is done, the appropriate parameters for the distribution will appear in the window (in this case, rate, since the exponential distribution was selected), and the user may enter values for these parameters. Next, if there had been more than one case for this activity, there would be

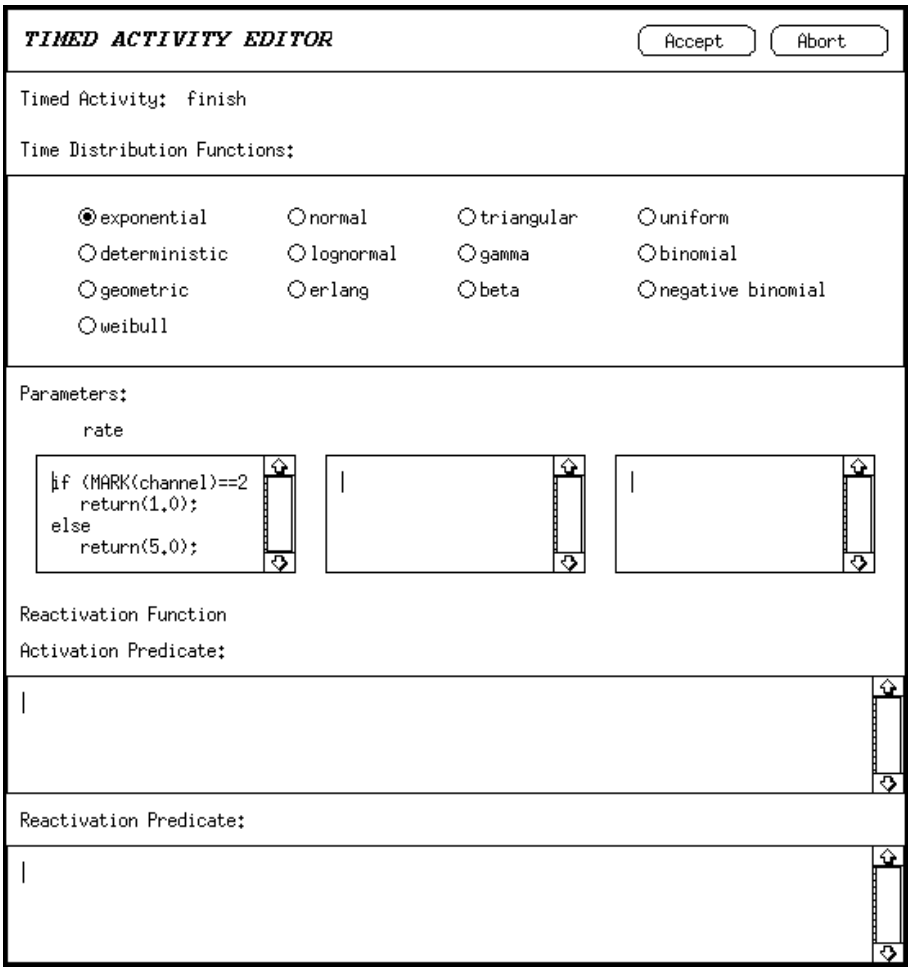


Figure 7: Timed Activity Editor

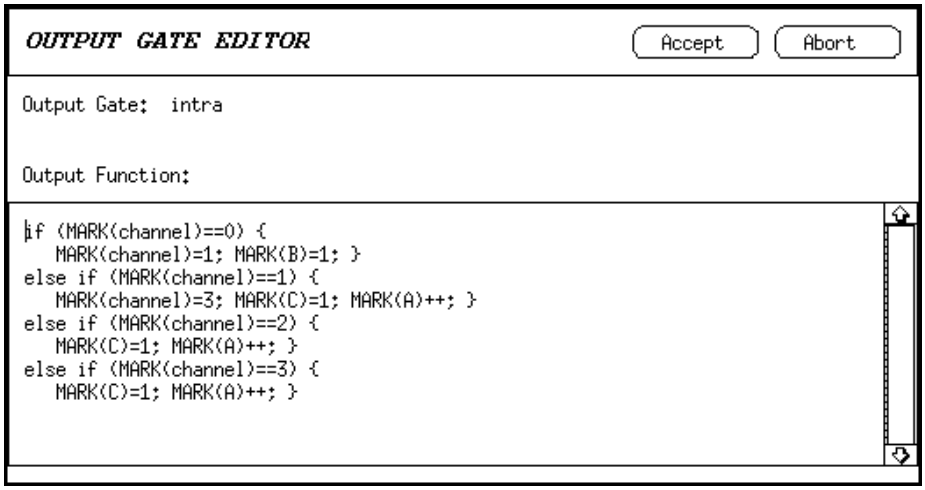


Figure 8: Output Gate Editor

a text editor for each case, to allow specification of its probability. These probabilities can be a simple value, or a complex marking dependent function. Reactivation functions are an advanced aspect of SANs (see [13] for information, if desired). They are specified in a manner similar to activity time parameters, by entering predicates defining activation markings and reactivation markings in the text windows shown. If no entry is given in either of these windows, the system assumes the default situation where an activity is never reactivated.

Input gates and output gates are specified in a similar manner, i.e. by applying the define tool to pop up a specification form. For example, consider the output gate *intra*, which updates the state of the channel after an access attempt. Figure 8 shows the code that is entered to specify *intra*'s output function. The specified code can be any valid sequence of C statements, where the keyword "MARK" is used to refer to the marking of a place.

B Composed Model Editor

After input of each of the subnets, the user then specifies the composed model structure. This is also done graphically, using the composed model editor (`compedit`) as pictured in Figure 9. As depicted in the figure, the composed model graph consists of the three types of nodes described earlier: replicate nodes, join nodes, and subnet nodes. Recall that subnet nodes refer to the subnets the user has already created. Similarly, a replicate node operates on the submodel that it is connected to, producing a new submodel that consists of a specified number of copies of the original submodel, holding a subset of places common

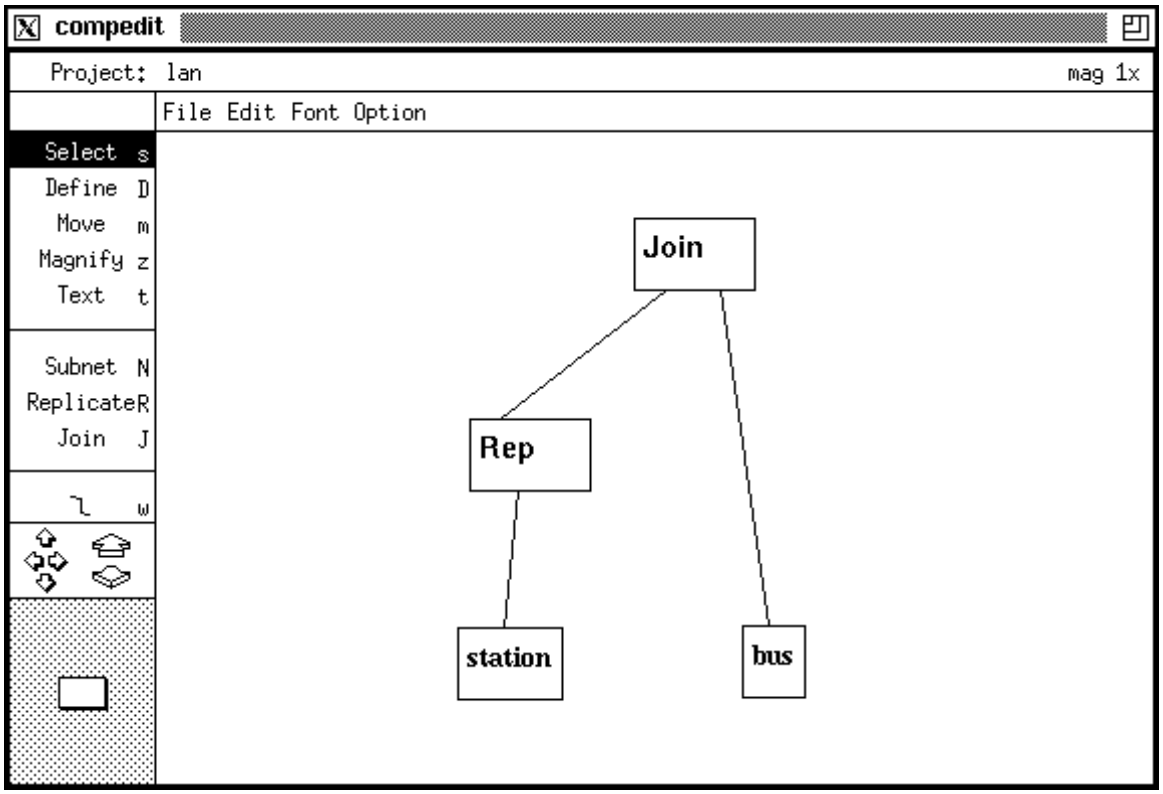


Figure 9: Composed Model Editor

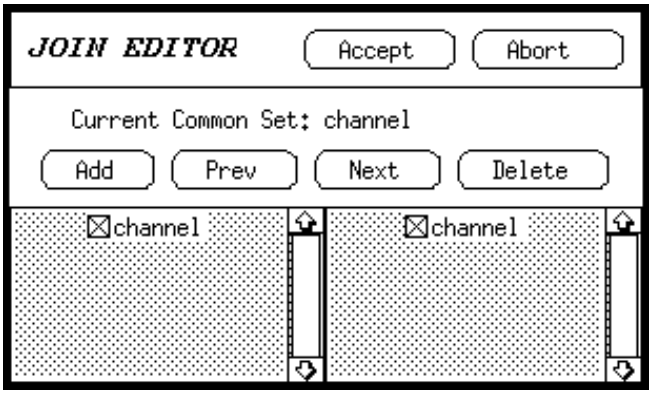


Figure 10: Join Editor

to all the replicas. Finally, join nodes are used to merge two or more dissimilar models, connecting some subset of places in each submodel with places in the other submodels. It is important to note that both the join and replicate operations can be applied multiple times in an iterative manner. In other words, the result of an operation on a submodel is itself a submodel on which further operations can be performed.

As in `sanedit`, a user of `compedit` specifies the composed model by simply drawing the graph corresponding to a model, using the composed model primitives on the left of the figure. Specification of the distinguished places associated with each replicate and join node is done with pop-up forms, similar to the method in which SAN components are specified. In particular, the define tool is applied to a node, and the appropriate node editor pops up. For example, to specify the details of the join node in Figure 9, the join editor shown in Figure 10 is used. To identify a place in one subnet with a place in another, one clicks the mouse on the boxes next to the place names. For example, Figure 10 indicates the join editor configuration if the place *channel* in the *station* subnet is to be identified with the place *channel* in the *network* subnet. Multiple connections between different subnets are allowed by the join editor but a particular place can only be used in one connection. Like the join editor, the replicate editor uses check boxes to allow simple denotation of places that are to be common among the replicas. This structure description method makes it easy to represent large, complicated systems composed of heterogeneous subsystems. Furthermore, it provides the formal structure necessary for the efficient solution of the models using either mathematical analysis or simulation.

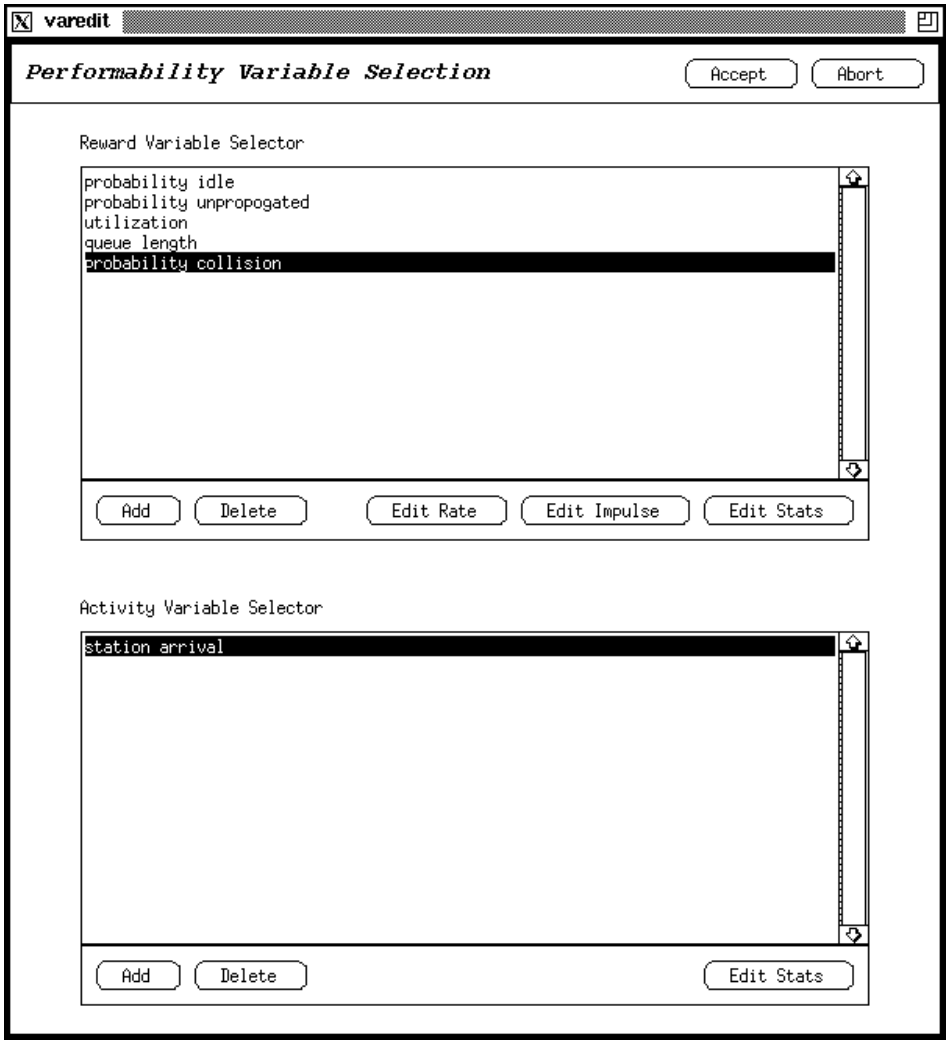


Figure 11: Performability Variable Editor

C Performability Variables and the Performability Variable Editor

In *UltraSAN*, variables are specified through the use of the performability variable editor (`varedit`). As shown in Figure 11, the initial screen of `varedit` provides variable management functions. Two classes of variables are supported: the reward variables disucessed earlier, and variables which estimate the time between completions of activities (called "activity variables," in the following). The top portion of the editor screen is used to add, delete, edit, and specify simulator statistics for the reward variables, while the bottom is used to add, delete, and specify simulator statistics for activity variables.

As reviewed earlier, reward variables have rates and impulses associated with them,

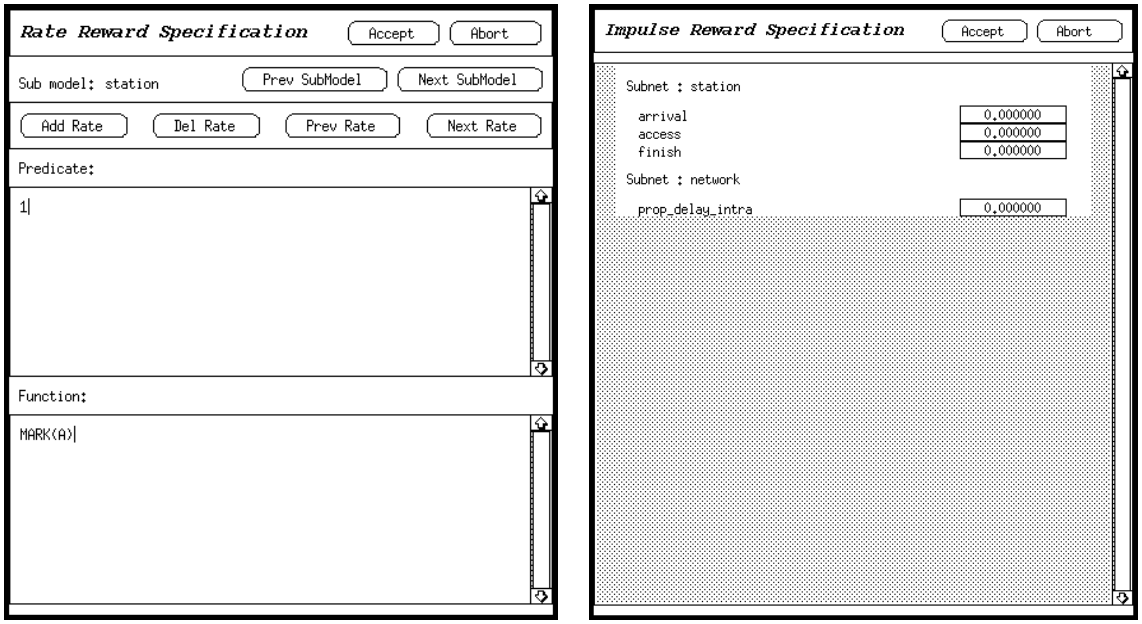


Figure 12: Rate and Impulse Reward Editors

and can be obtain either by analysis or simulation, depending on model characteristics. To specify an impulse reward associated with the completion of an activity, the user simply clicks on the "Edit Impulse" button to pop up the impulse reward editor. The impulse reward editor displays all of the activities in the composed model, organized by submodel. Adjacent to each activity is a box where the impulse reward may be entered. For example, if the number of times a particular activity, say *arrival*, completed during some interval is of interest, an impulse reward of one should be assigned to that activity. Rate-based rewards are entered through the rate reward editor, shown in Figure 12. Rate rewards are specified as pairs consisting of a predicate and a rate function. If the predicate of a given pair is true for a marking, the rate determined by the rate function is earned while the model is in that marking. For example, in Figure 12, a rate reward equal to the marking of place A is earned at all times. In terms of the LAN example, this variable corresponds to the queue length of a station. Note that the specified predicate is 1, indicating that the rate reward will be earned for all markings. Multiple pairs can be defined; they are accessed by pressing the "Next Rate" and "Prev Rate" buttons near the top of the form. The total rate earned while in a marking is then the sum of the rates for all pairs whose predicates are true in that marking. Since rate rewards are specified on a per subnet basis, the "Prev Subnet" and "Next Subnet" buttons are provided to allow the user to scroll through the

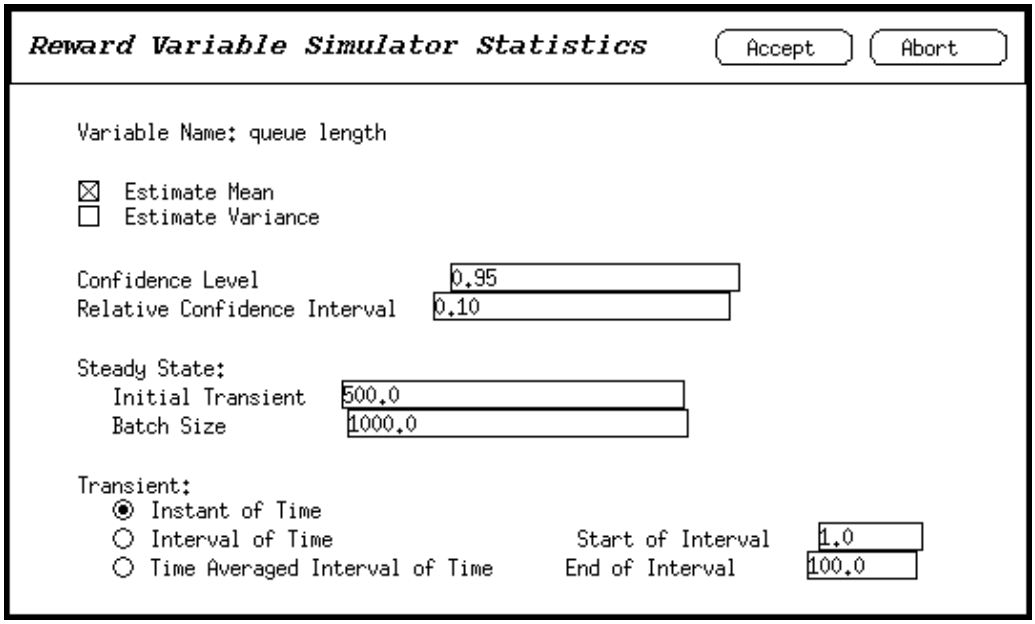


Figure 13: Simulator Statistics Editor

subnets in the composed model. The "Edit Stats" button is used to specify characteristics of the variables that are specific to simulation. As seen in Figure 13, the desired estimators for a variable can be specified, along with a desired confidence level and relative confidence interval width. Furthermore, for steady state simulation, the initial transient and batch size are specified. For the terminating simulation, the type of variable, along with the time point or interval associated with the variable are specified.

Activity variables are specified in the lower portion of the main `varedit` screen, and can currently only be estimated via simulation. They are specified in a manner similar to the reward variables, except that no rate or impulse need be specified. Instead, the user simply gives the subnet and activity name of the activity for which the estimation of the time between completions is desired. This information is then passed to the simulator, which collects the necessary data to estimate the mean or variance of the time between completions of the activity.

V Model Construction and Solution

After input of the system description and measures, the solution can be obtained by either analysis or simulation, depending on system characteristics. Informally, stochastic

activity networks can be solved via analytic methods when all activity time distributions are exponential, activities are reactivated often enough to ensure that their rates depend only on the current state, and the state space is not too large relative to the capacity of the machine.

A Reduced Base Model Construction

If analysis is the intended solution method, the internal representation of the system is passed to the reduced base model constructor (see Figure 5). As outlined in the Section II, this module takes a description of a composed model, including one or more SAN submodels and performability variables, and builds a state-level representation of the model. The state-level representation consists of a set of states, rates to transition between each pair of states, and a set of impulse and rate rewards for each state. The notion of state employed is variable and depends on the structure of the composed model. One can think of the state as an impulse and rate reward plus a *state tree* [10], where each node on the state tree corresponds in type and level with a node on the composed model tree. Furthermore, each node in a state tree has associated with it a subset of the distinguished places of the corresponding node in the composed model diagram. These places are those that are distinguished at the node, but not at its parent node.

Nodes are connected by directed arcs. An arc that connects a parent node i to a node j has associated with it an integer $n_{i,j}$, which represents the number of occurrences of the submodel j in the current state of the composed model. By definition, each outgoing arc from a join node has $n_{i,j}$ equal to one, since there is one copy of each submodel in the join operation. Arcs originating from replicate nodes may have values greater than one. Furthermore, the sum of the values of all arcs from a replicate node is the number of times that submodel was replicated in the composed model.

The initial state tree for the LAN example is shown in Figure 14. Initially, all three station submodels are in the same marking (where the marking of *channel* is 0, the marking of A is 1, B is 0, and C is 1). Places A, B, and C are at the leaf node corresponding to the network model, since they are local to that model. Place *channel* is at the root node, since it is common to all station submodels and the network submodel. Note that each leaf on the state tree represents a submodel type in a particular marking. In this case, since the tree represents the initial marking, all submodels of a given type are in the same marking.

After the initial state is constructed, generation of the reduced base model proceeds as

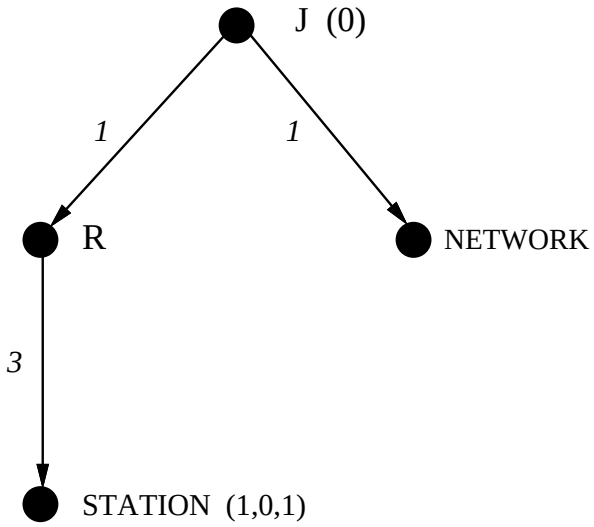


Figure 14: Initial State Tree for LAN Model

follows. First, each activity that may complete in the initial state is completed, generating a new state tree and impulse reward corresponding to each possible next state 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 to other states for 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 reduced base model 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 [9].

The result of the procedure, if it terminates successfully, is a set of reduced base model states, rates between these states corresponding to activity completions in the stochastic activity network, and for each variable defined, an impulse and rate reward for each state. This state-level reward model serves as input to the analytical solvers.

B Analytical Solvers

After a reduced base model is constructed, solution for the desired variables proceeds using known stochastic process solution techniques to obtain the appropriate state-occupancy probabilities. These probabilities, together the impulse and rate reward calculated for each state while generating the reduced base model, are then used to generate the mean, variance, probability density function, and probability distribution function for each variable. An understanding of the workings of these techniques is not necessary to use the package. However, for those familiar with such techniques, we briefly review the choices we have made in their implementation as well as the options available to users at run time. Three analytical solvers are provided: a direct steady-state solver, an iterative steady-state solver, and a transient solver. The two steady-state solvers can be used to determine the long-run behavior of a system. Examples of variables in this category include the average response time of a computer network, the utilization of a computer network or machine, and the availability of a network node or machine. The transient solver solves for system characteristics at particular times. For example, one may be interested in the probability that a network node or machine is up at a particular time, or the status of a process at a given time.

The direct steady-state solver is based on a technique known as LU decomposition. It permits the calculation of the expected value, variance, probability density function, and probability distribution function of steady-state instant-of-time variables. Two methods are used to reduce the fill-in of the matrix during solution. The first is the improved generalized Markowitz strategy, which selects a next pivot element based on an heuristic that can reduce fill-in. The second is a technique advocated by Osterby and Zlatev, in which elements that are less than some value (the *drop tolerance*) are set to zero during the solution algorithm. *Iterative refinement* is then used to obtain a correct final solution. The drop tolerance can be specified by the user at run time, and can be set to zero if use of the technique is not desired. Likewise, the pivoting mechanism can be turned off if desired. In any case, an estimate of the error in the final solution is given. When iterative refinement is used, the norm of the residual is used as the error estimate. When it is not used, a method by Cline is used to estimate the condition number and, in turn, estimate the error of the solution. Output from this, and all of the analytic solvers, is both textual and graphical. The mean and variance of each variable is given in textual form, while the probability density and

distribution functions are given in graphical form.

The iterative steady-state solver is based on the successive over-relaxation technique. This method is applicable to state-spaces much larger than the direct technique, due to its more modest space requirements, but is not guaranteed to converge for all reduced base models and initial conditions. As with the direct steady-state solver, it is intended to be used to solve for steady-state instant-of-time variables. The optimal acceleration factor is impractical to compute, even for a given problem, and hence left to the user to select. Values that are approximately optimal can by determined by trial and error or by monitoring the rate of convergence and adaptively changing the acceleration factor. A second solver parameter that may be set by the user is whether to attempt to make the process matrix diagonally dominant. Although not proven, Golub implies that the more dominant the diagonal elements, the faster the solution will converge. If turned on, this option will attempt to make the process matrix close to diagonally dominant. A method must also be used to terminate the solver if divergence is suspected. As implemented, the solver is terminated if the maximum number of iterations is exceeded or if the error between the iterates gets very large. Each of these values have defaults which can be overridden by a user. As with the direct solver, an estimate of the error in the solution is provided, by computing the norm of the differences between the last two iterates.

The transient solver provides solutions for instant-of-time variables at particular times, using randomization. The technique used is a computationally stable version of one by Gross and Miller. The method is based on the idea of subordinating a Markov process to a Poisson process. This method is well suited to processes such as these for several reasons, including computational efficiency, preservation of matrix sparsity, and the ability to solve to user-specified tolerances. Simultaneous solution for multiple time points is supported. This approach is significantly more efficient than solving for multiple time points iteratively. The technique, as implemented, is very memory efficient. In particular, by computing the probability for each time point one state at a time, only one column of the state transition probability matrix, raised to various powers, need be kept in memory at any time. As with the other two analytic solvers, the transient solver yields means, variances, probability density functions, and probability distribution functions for each variable solved.

C Simulation Solvers

When system characteristics preclude mathematical analysis, simulation can be used as the solution method. Simulation is performed directly using the internal SAN, composed model, and performance variable representation created by the user interface module. Again, as with the stochastic process generator, the constructed simulation program makes use of the structure of the SAN and choice of performance variables to speed up solution of the system. Two simulation solvers are available: a terminating (transient) simulator to solve for variables that are a function of the behavior of a SAN at a particular time or interval of time, and a steady-state simulator to solve for long-run measures of performance, dependability, and performability. Both simulation solvers provide estimates of the accuracy of the result, using an iterative method (batch means, in the steady-state solver, and replication in the terminating solver) for estimating the confidence interval at a user specified level.

As with the reduced base model constructor, the simulators operate on state trees. Recall that each leaf on the state tree represents one or more submodels in a particular marking. This fact can be used to reduce the number of activities that are checked for a change in their "status" (i.e., enabled or disabled) from one state to another. This is possible, since, if an activity a has a particular status in some marking, then all replicas of this activity whose input places in the same marking as the input places of a will have the same status as a.

Building upon this idea, we define a *representative activity* as an activity that "represents" the set of replicated activities $a_1 \in A_1$, $a2 \in A_2$, ..., $a_i \in A_i$, ..., $a_n \in A_n$, where A_i is the set of activities of a submodel i, and i is one of a set of n replicated submodels of a particular type in identical markings. Each representative activity is an *event type* in the new simulation technique, whereas activity completions are events.

As seen earlier, the state tree is organized in a manner that allows for identification of sets of replicas in a particular marking, as well as the number in that marking. During simulation, instead of checking every replicated activity for its status in the current marking, we use the representative activities. These checks are then reduced to a single check per set of replicated activities for a set of submodels in identical markings. The events for each of these replicated activities can be grouped into a list related to the representative activity. We call this list of potential completion times a "compound event". More formally, we define

a *compound event*, e_a, for representative activity, a, as the list of potential completion times $\{t_1, t_2, \ldots, t_n\}$, where n is the number of activities represented by a.

The number of submodels in a particular marking (n) can be found for every leaf, using the state tree, by multiplying the numbers on the arcs on the route to the leaf. The set of compound events can be then built from the list of enabled activities for each set of submodels represented by a leaf node.

The LAN example is useful to illustrate a possible state tree and corresponding multiple future events lists. Specifically, consider the composed model of Figure 4, where *STATION* is the submodel of Figure 1. *STATION* is replicated three times holding place *channel* as common. The result is then joined with the network submodel of Figure 3, holding the same place, *channel*, as common.

Figure 14 illustrates the initial state tree for the LAN model. The vectors beside nodes represent the marking of the set of places at the nodes. Figure 15 shows the multiple future events lists associated with this state. The compound events are represented by the names of the corresponding representative activity. One leaf is associated with the set of compound events E_1 and the other with E_2. E_1 has two compound events scheduled, *arrival* and *access*. The first number of the subscript of t relates it to a particular set, and the second identifies it with a representative activity in SAN submodel of type *STATION*. Since there are three replicas of type *STATION* 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, *arrival* has three potential completion times. The compound event *access* similarly has the same number of times. E_2 has no compound events, since there are no activities enabled in *NETWORK* in this marking.

Figure 16 shows a possible state tree resulting from the completion of activity *access* in one of the submodels of type *STATION*. Because this caused a change in the marking of a place at the lowest level, there is one more leaf. Each leaf represents a set of submodels of the same type in identical markings. For instance, the leftmost leaf is representing a set of two submodels of type *STATION* in identical markings. The compound events for the future events list associated with this leaf have, therefore, two elements each.

Precise algorithms for generating new state trees and future event lists can be found in [10]. From the example, however, one can see that multiple future event lists are employed, each corresponding to one or more submodels of a particular type in a specific marking. This makes it possible to use compound events to represent the completion times of several

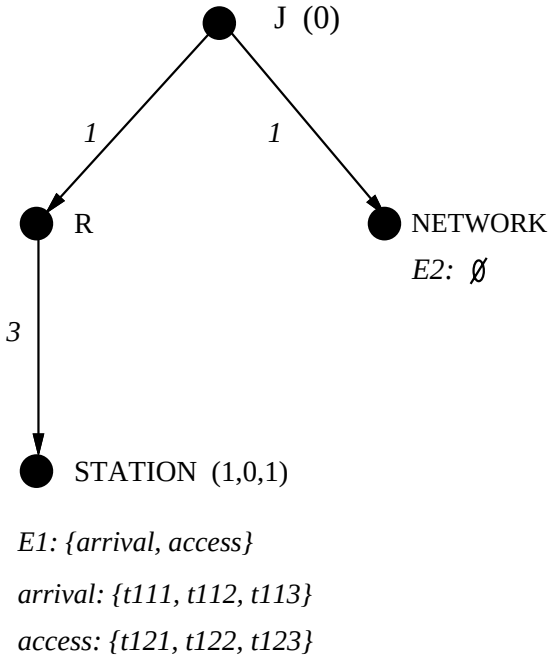


Figure 15: Future Events Lists for Initial State Tree

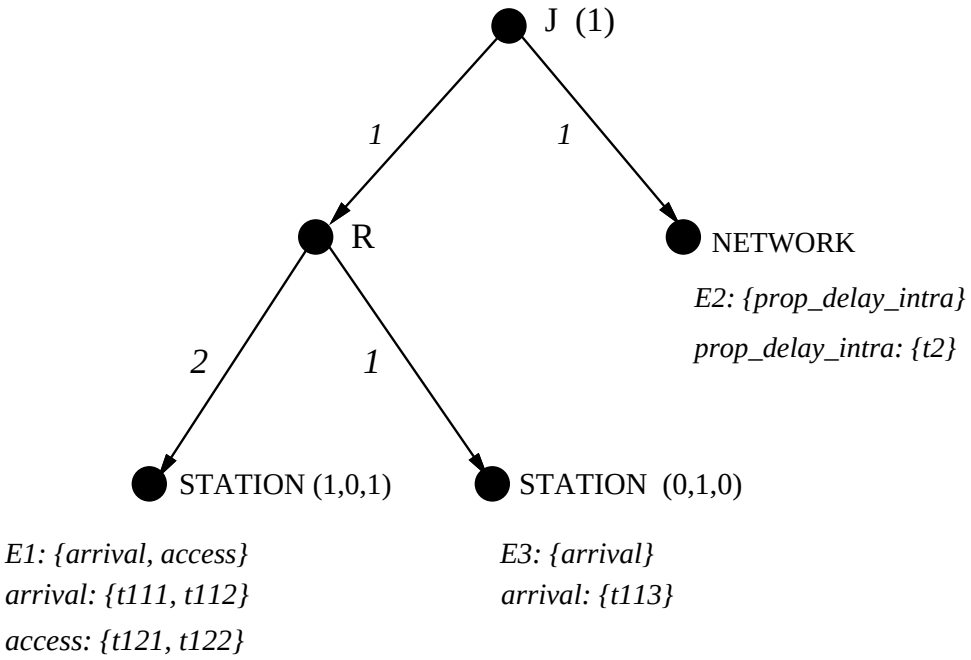


Figure 16: Future Events Lists for a Possible Next State Tree

activities of a single type. In the common situation where the composed model has replicate nodes, the algorithm achieves a significant reduction in the number of event types (i.e., activities) that must be managed.

For example, consider the case, as is shown in Figure 16, where one station has just begun to transmit, and the two other stations have a packet to transmit (i.e., activity *access* is enabled), but have not yet attempted transmission. In this case, a collision will occur if *either* of the two other stations begin transmission before packet being transmitted is propagated (i.e., activity *prop_delay_intra* completes). This possibility can be represented by the single compound event *access* in the left-most future event list on the tree, instead of two separate events, as would be needed if this approach was not used. While the savings in this case is modest, since our example is small, the savings will increase dramatically as the number of replicated submodels in the composed model increases.

These algorithms provide the basis for the state-change mechanism for both terminating and steady-state simulation. They are used, together with iterative batch and replication methods, to generate the trajectories necessary to estimate the specified performance variables. In all cases, the user specifies a desired confidence interval size and level, and the simulators attempt to reach the specified accuracy.

VI Conclusions

As argued in the introduction, stochastic Petri nets and extensions are a truly useful model class for representing distributed systems. In particular, they can be used to evaluate the performance, dependability, and performability of a design within a single modeling framework. Furthermore, because of their formal structure, they can provide analytic as well as simulation-based solutions. However, traditional model construction and solution methods for these models limit their usefulness, due to the extremely rapid growth of the size of the stochastic process used in model solution. Likewise, simulation times can be extremely long due, in part, to the large number of event types that are typically considered.

This paper reports on a new software package that alleviates these problems, using recently developed model construction and solution techniques for stochastic activity networks. As currently implemented, the package has two main purposes. The first is to serve as a test-bed during the development of new model construction and solution methods. In this spirit, we are currently investigating methods to analytically solve for interval-of-

time variables, using an extension of the randomization technique proposed by De Souza e Silva and Gail, as well as investigating new base model construction techniques that are applicable to a wider class of variables. The second purpose is to provide an easy-to-use, graphical, X-window based software package, which can be used by systems engineers and researchers to evaluate the performability of specific systems. This has been accomplished, and the package is now being used in both industry and academia to evaluate new and existing computer hardware, software, and network designs.

Acknowledgement

The authors would like to thank Mike Woodbury, of Bellcore, Rose Mary Owen, of Intel, and Hemal Shah and Abijhit Kudrimoti, of the University of Arizona, for using early versions of this tool, and providing feedback that has guided its future development. Their suggestions have been invaluable in improving the tool.

REFERENCES

[1] J. F. Meyer, "On evaluating the performability of degradable computing systems," *IEEE Trans. Comput.*, vol. C-22, pp. 720–731, Aug. 1980.

[2] W. H. Sanders and J. F. Meyer, "METASAN: A performability evaluation tool based on stochastic activity networks," *Proc. ACM-IEEE Comp. Soc. 1986 Fall Joint Comp. Conf.*, Dallas, TX, November 1986.

[3] J.F Meyer, K.H. Muraldihar and W.H. Sanders, "Performability of a token bus network under transient fault conditions," in *Proc. 19th Int. Symp. on Fault-tolerant computing*, Chicago, June 1989.

[4] B. E. Aupperle and J. F. Meyer, "Fault-tolerant BIBD networks," in *Proc. Int. Symp. on Fault-tolerant Computing,* Tokyo, Japan, 1988.

[5] W. H. Sanders, R. Martinez, Y. Alsafadi, and J. Nam, "Performance evaluation of a picture archiving and communication system using stochastic activity networks," to appear in *IEEE Trans. on Medical Imaging.*

[6] K. H. Prodromedes and W. H. Sanders, "Performabilty evaluation of CSMA/CD and CSMA/DCR protocols under transient fault conditions," presented at the *10th International Symp. on Reliable Distributed Systems,* Pisa, Italy, Sept. 30 – Oct. 2, 1991.

[7] J.F. Meyer and L. Wei, "Influence of workload on error recovery in random access memories," in *IEEE Transactions on Computers, Vol. 37, No. 4,* April 1988.

[8] W. H. Sanders and J. F. Meyer, "A unified approach for specifying measures of performance, dependability, and performability," in *Dependable Computing for Critical*

Applications, Vol 4: of Dependable Computing and Fault-Tolerant Systems (ed., A. Avizienis and J. Laprie), Springer-Verlag, 1991.

[9] W. H. Sanders and J. F. Meyer, "Reduced base model construction methods for stochastic activity networks," *IEEE Journal on Selected Areas of Communications*, Jan. 1991.

[10] R. S. Freire, "A technique for simulating composed SAN-based reward models," Master's Thesis, Dept. of Electrical and Computer Engineering, University of Arizona, Dec. 1990.

[11] A. Movaghar and J. F. Meyer, "Performability modeling with stochastic activity networks," in *Proc. 1984 Real-Time Systems Symp.*, Austin, TX, Dec. 1984.

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

[13] W. H. Sanders, "Construction and solution of performability models based on stochastic activity networks," Computing Research Laboratory Technical Report CRL-TR-9-88, The University of Michigan, Ann Arbor, MI, August 1988.