From *Performance Evaluation*, vol. 24, no. 1, Oct./Nov. 1995.

THE *UltraSAN* MODELING ENVIRONMENT *

W. H. Sanders†, W. D. Obal II‡, M. A. Qureshi†, and F. K. Widjanarko‡

† Center for Reliable and High-Performance Computing
Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
Urbana, IL 61801
{whs, qureshi}@crhc.uiuc.edu

‡ Dept. of Electrical and Computer Engineering
University of Arizona
Tucson, AZ 85721
{obal, krisnadi}@ece.arizona.edu

ABSTRACT

Model-based evaluation of computer systems and networks is an increasingly important activity. For modeling to be used effectively, software environments are needed that ease model specification, construction, and solution. Easy to use, graphical methods for model specification that support solution of families of models with differing parameter values, are also needed. Since no model solution technique is ideal for all situations, multiple analysis- and simulation-based solution techniques should be supported. This paper describes *UltraSAN*, one such software environment. The design of *UltraSAN* reflects its two main purposes: to facilitate the evaluation of realistic computer systems and networks, and to provide a test-bed for investigating new modeling techniques. In *UltraSAN* models are specified using stochastic activity networks, a stochastic variant of Petri nets, using a graphical X-Window based interface that supports large-scale model specification, construction, and solution. Models may be parameterized to reduce the effort required to solve families of models, and a variety of analysis- and simulation-based solution techniques are supported. The package has a modular organization that makes it easy to add new construction and solution techniques as they become available. In addition to describing the features, capabilities, and organization of *UltraSAN*, the paper illustrates the use of the package in the solution for the unreliability of a fault-tolerant multiprocessor using two solution techniques.

Keywords: Stochastic Petri Nets, Stochastic Activity Networks, Model-Based Evaluation, Performance, Dependability, Performability.

*This work was supported in part by IBM, Motorola Satellite Communications, and US West Advanced Technologies.

I Introduction

Model-based performance, dependability, and performability evaluation have become integral parts of the design process. The primary strength of using models rather than a prototype system is the ease in evaluating competing design decisions. Creating a model usually costs less than building a prototype, and in most cases produces an accurate model that is much less complex than the system itself. This approach has the added benefit that once a model is created it may be saved for future use in evaluating proposed modifications to the system. So although measurement is and will remain an important method of system evaluation, modeling reduces the evaluation effort by focusing attention on the system characteristics having the largest impact on the performance measures of interest.

Due to these merits and the computational complexity of most model solution methods, the development of effective software tool support is an active research area. Many such tools exist, and are based on a wide variety of model specification techniques (e.g., queueing networks, Markov processes, object oriented languages, and stochastic Petri nets). Several surveys have been written on such tools, see, for example, [1, 2, 3]. Stochastic Petri nets and extensions have been the subject of significant recent research resulting in a number of software tools supporting the evaluation of systems represented in this way. These tools include among many DSPNexpress [4], FIGARO [5], GreatSPN [6], METASAN [7], RDPS [8], SPNP [9], SURF-2 [10], TimeNET [11], and *UltraSAN* [12]. Some of these tools have been built to illustrate particular classes of solution algorithms (e.g., DSPNexpress, analytic solution of SPNs with deterministic and exponential timings), while others have had very broad goals, implementing both analytic- and simulation-based solution methods (e.g., GreatSPN, SPNP, SURF-2, and *UltraSAN*).

This paper describes *UltraSAN*. In *UltraSAN*, models are specified using a variant of SPNs known as stochastic activity networks (SANs), and solution techniques include many analytic- and simulation-based approaches. The development of the software was guided by two goals: the desire to create an environment (test-bed) for experimenting with new model construction and solution techniques, and the goal of facilitating large-scale evaluations of realistic systems using both analytic- and simulation-based techniques. As will be seen in the next section, the first goal dictates that the software be constructed in a modular fashion so new construction and solution techniques can be easily added. The second goal dictates that many different solution techniques be available, since no single technique is ideal in all circumstances.

Since the initial release in 1991 [12], many new construction and solution techniques and user interface features have been added. In particular with respect to the user interface, a parameterized model specification method that facilitates solution of models with multiple sets of input parameter values has been developed. Furthermore, three new analytical solvers, as well as a terminating

simulation solver based on importance sampling, have been added to the package. The first new analytic solver solves models that have deterministic as well as exponential activities. The two remaining new analytic solvers compute the probability distribution function and mean, respectively, of reward accumulated over a finite interval. The importance sampling simulator provides an efficient solution for models that contain rare events which are significant relative to a measure in question. These enhancements have improved *UltraSAN*'s usefulness to others, and illustrated its effectiveness as a test-bed for new modeling techniques. The package has now been at academic and industrial sites for about four years, and much has been learned from its use.

Our purpose for describing *UltraSAN* is to provide insight into the most important aspects of the package and the reasons particular design decisions were made. Clearly, space does not permit an extended discussion of the theoretical developments made and incorporated in the package. However, reference is given to these developments and the interested reader is urged to consult the appropriate references.

The remainder of the paper is organized as follows. Section II of this paper gives an overview of the modeling process, as embodied in *UltraSAN*, including a brief review of SANs and an example illustrating the use of SANs in modeling a fault-tolerant multicomputer. The intent here is to provide a high-level overview of the capabilities and features of the package. Sections III–VI describe in detail each step of the modeling process. These sections provide more insight regarding the design choices made in each step of the process. Results for the example obtained using two different techniques (uniformization and importance-sampling based simulation) are given in Section VII. Finally, Section VIII summarizes the contributions and features of the tool, and suggests areas of future research and development.

II Overview

The organization of *UltraSAN* follows directly from the goals of the package. As shown in Figure 1, the package is constructed in a modular manner. All of the modules shown are coordinated by a central program called the control panel, that manages the interface between the user and the various modules, and provides basic consistency checking. In addition, the control panel offers several useful utilities, such as automatic documentation and convenient viewing of output data. The modules communicate via files which have well-defined formats. This configuration makes it easy to add new functionality, thus contributing to the goal of being a test-bed for developing new modeling techniques.

Figure 1 is divided into four sections, representing the four phases of the modeling process used in *UltraSAN*: *model specification*, *global variable assignment*, *study construction*, and *study solution*. These phases are briefly described here as an introduction to the package, with more

detailed descriptions of their functioning and underlying theory given in the following sections.

Model specification In the model specification phase, SANs are defined graphically using the SAN editor and then composed into a hierarchical model using the composed model editor. The performance variable editor is used to define the performance measures of interest in terms of a reward structure at the SAN level. Once a model is specified, it is typically solved many times for various values of the input parameters. To accommodate this activity, *UltraSAN* allows model specifications to be parameterized through the use of global variables in definitions of model components in the SAN editor.

Global variable assignment In the global variable assignment phase, values are assigned to global variables. A model specification, together with the assignment of a single value to each global variable, is called an experiment. A group of related experiments is called a study. *UltraSAN* provides the study editor to make it easy to specify groups of related experiments.

Study construction The study construction phase, is the process of converting a specified model and a global variable assignment into a format amenable to computer solution via analytic or simulation-based techniques.

Analytical models are constructed by automatically generating a Markov process representation of the model. The size of the generated Markov process is determined in part by the definition of state. In standard stochastic Petri net models, the standard definition of the state space is the set of tangible reachable markings. Realistic models of modern systems usually have very large state spaces. This problem has been widely recognized, and various approaches have been devised to handle the explosive growth [13, 3]. *UltraSAN* uses an innovative largeness-avoidance approach called reduced base model construction [14]. This technique uses information about the performance measures and symmetries in the model to directly generate a stochastic process that is often much smaller than the set of reachable markings, and yet is sufficient to support the specified performance measures. Experience has shown that reduced base model construction is effective for medium to large systems that have replicated subsystems [15].

When the developed model is intended to be solved using simulation, *UltraSAN* does not generate a state-level representation. For this solution method, study construction is accomplished by linking the model description files generated in model specification and global variable assignment to the simulation library. The result is an executable simulation of the model. Essentially, simulation is accomplished by assigning events to activity completions and using the rules of SAN execution to determine the next state. Data structures designed for reduced base model construction [16] make simulation more efficient. Symmetries in the model that allow lumping in state space construction

also help reduce the amount of future events list management necessary for each state change.

Study construction is carried out using the control panel. After verifying that the model description is complete, the control panel builds the reduced base model generator for each experiment to be solved analytically, and builds executable simulators for experiments where simulation-based solution is desired. The control panel can then be used to run the reduced base model generator for all experiments that are to be solved analytically. In this case, the control panel can distribute the jobs to run on a user-specified list of other available machines. This "multiple runs" feature makes it easy to construct a study with little effort on the part of the user.

Study solution The last phase of the modeling process is study solution. If analytic solution is intended, the reduced base model generator was executed during study construction to produce the reduced base model. This representation of the model serves as input to the analytic solvers. Since a single solution method that works well for all models and performance variables is not available, *UltraSAN* utilizes a number of different techniques for solution of the reduced base model. To meet the goal of providing a test-bed for research into new solution techniques, the interface between the reduced base model generator and the analytic solution modules is well-defined, making it easy to add new solvers as they become available.

The package currently offers six analytic solution modules. The direct steady state solver uses LU-decomposition. The iterative steady-state solver uses successive overrelaxation (SOR). The deterministic iterative steady-state solver is used to solve models that have deterministic as well as exponential activities. It uses SOR and uniformization. The transient instant-of-time solver uses uniformization to evaluate performance measures at particular epochs. All of these solvers produce the mean, variance, probability density, and probability distribution function. The PDF interval-of-time solver uses uniformization to determine the distribution of accumulated reward over a specified interval of time. Similarly, the expected interval-of-time solver uses uniformization to compute the expected value of interval-of-time and time-averaged interval-of-time variables.

Three solution modules are available for simulation. The steady-state simulator uses the method of batch means to estimate the mean and variance of steady-state measures. The terminating simulator uses the method of independent replications to estimate the mean and variance of instant-of-time, interval-of-time, and time-averaged interval-of-time measures. The third simulator is useful when the model contains events that are rare, but significant with respect to the chosen performance variables. In this case, traditional simulation will be inefficient due to the rarity with which an event of interest occurs. For problems like this, *UltraSAN* includes an importance sampling simulation module [17] that estimates the mean of a performance measure. The importance sampling simulation module is flexible enough to handle many different heuristics, including balanced failure biasing, and is also applicable to non-Markovian models [18].

The study solution phase is made easier by the control panel multiple runs feature. After study construction, the control panel is used to run the reduced base model generator for all experiments solved analytically. After the state spaces for the experiments have been generated, all experiments can be solved by automatically distributing the solver or simulation jobs to run on all available machines.

These four phases, model specification, global variable assignment, study construction, and study solution are used repeatedly in the modeling process. All phases are supported by a graphical user interface, which facilitates specification of families of parameterized models and solution by many diverse techniques. The following four sections detail the underlying theory and steps taken in each phase of the process.

III Model Specification

The model specification steps are specification of SAN models of the system, composition of the SAN models together to form a composed model, specification of the desired performance, dependability, and performability variables and if needed, specification of the "governor" to be used in importance sampling simulation. Completion of these three or four steps results in a model representation that can be used to generate an executable simulation or state-level model representation, once values are assigned to global variables.

Input to each of the four editors is graphical and X-window based. Detailed instructions concerning the operation of each editor cannot be given due to space limitations, but can be found in [19]. However, a brief description of what must be specified for each step follows.

SAN model specification The SAN editor (see Figure 3) is used to create stochastic activity networks that represent the subsystems of the system being studied. Before describing its operation, it is useful to briefly review the primitives that make up a SAN and how these are used to build system and component models. SANs consist of places, activities, and gates. These components are drawn and connected using the SAN editor and defined through auxiliary editors that pop up on the top of the SAN editor.

The functions of the three model primitives are as follows. *Places* are as in Petri nets. Places hold tokens, and the number of tokens in each place is the *marking* of the SAN. The interpretation of the number of tokens in a place is left to the discretion of the modeler. Sometimes tokens are used to represent physical quantities such as the number of customers in a queue, or the number of operational components in a system. On the other hand, it is often convenient to use the marking of a place to represent the state of a network, or the type of a customer.

Activities are used to model delays in a system. *Timed* activities have a (possibly) marking-dependent *activity time distribution*, a (possibly generally distributed) probability distribution func-

tion that describes the nature of the delay represented by the activity. The special situation of a zero delay is represented by the *instantaneous* activity.

Cases are used to model uncertainty about what happens upon completion of an activity. Each activity has one or more cases and a marking-dependent case distribution. A *case distribution* is a discrete probability distribution over the cases of an activity. Using cases, the possible outcomes of the completion of a task can be enumerated and the probability of each outcome is specified. This modeling feature is useful for representing imperfect coverage or routing probabilities characteristics. They are closest in nature to the "probabilistic arcs" of generalized stochastic Petri nets, but differ in that the probabilities assigned to cases can be dependent on the global marking of the SAN.

Gates serve to connect places and activities, and are an important part of the modeling convenience offered by SANs. *Input* gates specify when activities are enabled, and what happens to the input places when the activity completes. The predicate of an input gate is a boolean function of its input places, which when true causes the activity to be enabled. Each input gate also has a function that specifies how the marking of the input places is updated when the activity completes. In most stochastic Petri nets (e.g., GSPNs), enabling conditions and marking updates are specified using networks of immediate transitions and places. Input gates allow functional descriptions of the enabling predicate and marking updates, which is more convenient for specifying complicated enabling conditions and token movements that are common in models of real systems. *Output* gates allow functional specifications of how the marking of output places is updated upon completion of an activity. This flexibility is enhanced when output gates are used in combination with cases. Introduced first in SANs in [20], the utility of gates has become apparent, and similar features have been incorporated into other extensions to stochastic Petri nets, such as the guards in SRNs [21].

Another important feature in the model specification tools is the incorporation of "global variable" support. In the SAN editor, global variables may be used in the definition of initial markings of places, activity time distributions, case distributions, gate predicates, and gate functions. Global variables can be thought of as constants (either integer or floating point) that remain unspecified during model specification, and are varied in different experiments. It is important to point out that the use of global variables is not limited to simple assignments. From its inception, *UltraSAN* has allowed SAN component definitions to make full use of the C programming language. The use of variables global to the model has been built on top of this capability. One ramification of this architecture is that the modeler is free to define SAN components in terms of complicated functions of one or more global variables or change the flow of control in a gate function, rather than simply assigning variables to component parameters.

The fault-tolerant multicomputer system discussed in [22] serves as a good example to illustrate the modeling process in *UltraSAN*. Although this system is a pure dependability model, it incorporates some interesting features that challenge a model description language. As in [22], a

multicomputer consists of multiple computer modules connected via a bus, where each module is as shown in Figure 2. Each computer module consists of three memory modules, three CPUs, two I/O ports, and an error handler. Each computer module is operational if at least two memory modules, two CPUs, one I/O port and the error handler are operational. The multicomputer is considered to be operational if at least one of the computer modules is operational. As given in [22], reliability is modeled at the chip level, and each chip has a failure rate of 1×10^{-7} failures per hour or 8.766×10^{-4} failures per year.

An interesting feature of the system is that different coverage factors have been assigned to failures at each level of redundancy. These coverage factors are listed in Table 1, along with the global variables used to represent them. For example, if a RAM chip fails and there are spares available, there is a 0.998 probability that it is successfully replaced by a spare. However, with probability 0.002, this failure causes the memory module to fail. The failure of a memory module is covered (assuming an available spare) with probability 0.95. But with probability 0.05, the memory module failure causes a computer to fail. Finally, if a computer fails, and there is an available spare, the multicomputer fails with probability 0.05. So, in the case where there are available spares at each level of redundancy, the failure of a RAM chip causes the entire multicomputer to fail with probability $0.002 \times 0.05 \times 0.05 = 5 \times 10^{-6}$.

However, if spares are not available at all levels then the coverage probability is different. If, for example, there are no spare RAM chips, the probability that a RAM chip failure causes the multicomputer to fail is $0.05 \times 0.05 = 0.0025$, since a RAM chip failure after the spares have been used causes the memory module to fail. For this system, the coverage probabilities clearly depend on the state of the system at the time of a component failure. Later, we will show how to model this system feature using marking-dependent case probability distributions.

Figure 3 shows the SAN model of the memory module in each computer module. The SAN has two activities, called *memory_chip_failure* and *interface_chip_failure*. The input places of *memory_chip_failure* are those connected to input gate *IG1*, namely *computer_failed*, *memory_failed*, and *memory_chips*. As can be seen from Table 2, *IG1* holds (its predicate is true) if there are at least thirty-nine tokens in *memory_chips*, at most one token in *computer_failed*, and at most one token in *memory_failed*. Under these conditions, a memory chip failure is possible, and activity *memory_chip_failure* is enabled. The function of input gate *IG1* is the identity function, because the marking of the input places of *memory_chip_failure* depends on the outcome of the failure. To accommodate this condition, the update of the marking is done by the output gates connected to the cases of the activity.

Each of the activities in the memory module SAN has multiple cases, representing the various failure modes of memory chips and interface chips. Connected to each case is an output gate that updates the marking according to the failure mode represented by the case. The output places of

activity *memory_chip_failure* are those places connected to the output gates that are connected to the cases of the activity. One can see from Figure 3 that all of the places in the memory module SAN are output places of *memory_chip_failure*.

Table 3 shows the parameterized case distribution for *memory_chip_failure*. Global variables have been used so that the impact of perturbations in the coverage factors can be evaluated (see Section VII). The first case corresponds to a covered RAM chip failure. The second case corresponds to a RAM chip failure that is not covered at the memory module level, but the failure of the memory module is covered at the computer level. In the third case, one of two events occurs as a result of an uncovered RAM chip failure. In the first event, a RAM chip failure that is uncovered at the memory module level propagates to the computer level, causing one computer to fail, but the computer failure is covered at the multicomputer level. In the second event, the memory module failure resulting from the uncovered RAM chip failure would have been covered, but exhausts the redundancy at the memory module level, and the resulting computer failure is not covered at the multicomputer level, leading to system failure. The two events were combined into one case in an effort to reduce the state space. Finally, the last case corresponds to a RAM chip failure that is not covered at any level, causing the entire multicomputer to fail.

The case probabilities are marking-dependent, because the coverage probabilities in the multicomputer system depend on the availability of spare components. To illustrate, consider the second case of *memory_chip_failure*. Using the coverage factors from Table 1 and assuming a spare memory module, the probability that the event associated with this case occurs is $(1 - RAM\_cov) \times mem\_cov = 0.002 \times 0.95 = 0.0019$ when there is at least one spare RAM chip available, but it increases to $mem\_cov = 0.95$ when there are no spare RAM chips left. There are forty one RAM chips, two of which are spare. The number of tokens in place *memory_chips* represents the number of working RAM chips. Therefore, a correct model of the coverage probabilities requires a case probability definition such as the one shown in Table 3. For the second case, if the marking of place *memory_chips* is equal to thirty nine, then there are no spares available, and the probability of this case is set to 0.95. Otherwise, there are spares available and the probability is set to 0.0019.

Table 4 shows an example of an output gate description. *OG2* is connected to the second case of activity *memory_chip_failure*, which was described above. The function of *OG2* sets the marking of places *memory_chips* and *interface_chips* to zero, disabling both activities in the memory module SAN. Then the marking of *memory_failed* is incremented, indicating a memory module has failed. If the marking of *memory_failed* is now greater than one, then the computer can no longer operate, since it has only one functioning memory module left, but requires two to be operational. This condition is checked by the *if* statement in the function of *OG2*, and *computer_failed* is incremented if more than one memory module has failed, indicating that the computer has failed.

The activity time distributions, case distributions, and gate predicates and functions are all

defined using auxiliary editors that pop up over the SAN editor. The definitions may be entered exactly as they are shown in the tables. The tables of component definitions in this paper have been created by the automatic documentation facility that is available from the control panel of *UltraSAN*. The tables are written in LaTeX and are generated from the model description files.

After SAN models of the subsystems have been specified, the composed model editor is used to create the system model by combining the SANs using replicate and join operations.

Composed Model Specification In the second step of model specification, the composed model editor is used to create a graphical description of the composed model. A composed model consists of one or more SANs connected to each other through common places via replicate and join operations.

The replicate operation creates a user-specified number of replicas of a SAN. The user also identifies a subset of the places in the SAN as common. The places not chosen to be common will be replicated so they will be distinct places in each replica. The common places are not replicated, but are shared among the replicas. Each replica may read or change the marking of a common place. The join operation connects two or more SANs together through a subset of the places of the SANs that are identified as common.

As in the SAN editor, auxiliary editors pop up over the composed model editor to allow one to specify the details of replicate and join operations. For example, the replicate editor lets one specify the number of replicas and the set of places to be held common among the replicas.

The composed model for the multicomputer system is shown in Figure 4. As can be seen from the figure, a model of a single computer is composed by joining three replicas of the memory module model with models of the CPU modules, I/O ports, and the errorhandler. The computer model is then replicated to produce a model of the multicomputer system. The place *computer_failed* is an example of a common place. It is held common at the system level, where it can be accessed by all subsystems, so that they may update the number of failed computers in the event of a coverage failure.

Performance, Dependability, and Performability Variable Specification The third step in model specification is to use the performability variable editor to build reward structures from which the desired performance measures may be derived. In short, to specify a performability variable one defines a reward structure and chooses a collection policy. Variables are typed and often referred to according to the collection policy. The three types of variables supported in *UltraSAN* are "instant-of-time," "interval-of-time," and "time-averaged interval-of-time." *Instant-of-time* variables are obtained by examining the value of the associated reward structure at a particular time t. *Interval-of-time* variables measure the reward accumulated in an interval $[t, t + l]$. *Time-averaged interval-of-time* variables divide the accumulated reward by the length of the interval. For a thor-

ough discussion of the concept and application of these variables, see [23].

For simulation models, "activity variables" can also be defined. *Activity variables* measure the time between completions of activities. Using an activity variable, one can measure the mean and variance of the time until first completion, the time between the tenth and twentieth completions, etc., using the terminating simulator. The steady-state simulator can be used to measure the mean and variance of the time between consecutive completions as time goes to infinity.

Importance Sampling Governor Specification For problems where traditional simulation methods are inefficient due to rare but significant events with respect to the chosen performance variables, importance sampling may be used.

Importance sampling is a variance reduction technique where the probability measure on the sample path space is replaced with another probability measure with the intent of improving the efficiency of the simulation. Typically this change of measure is induced by altering the stochastic components of the simulation model. To compensate for taking observations from an altered model, the observations are weighted by the likelihood ratio. This ratio is the ratio of the probability of the sample path under the orginal probability measure to the probability of the same sample path under the new probability measure. This weighting is required to retain an unbiased estimator. For the likelihood ratio to be valid, the new probability measure that is induced by the changes to the stochastic components of the simulation model must assign nonzero probability to all paths that have nonzero probability under the original measure.

In this way, importance sampling can be used to focus the simulation effort on sample paths leading to the event. The stochastic components of the simulation model are altered to increase the likelihood of encountering the rare event. A good example of a problem where importance sampling can help is estimating the unreliability of dependable systems [24, 25, 18]. In this case, the model is altered to accelerate component failures with the intent of increasing the probability of system failure. The cited research shows the utility of varying the acceleration applied to component failures depending on the evolution of a sample path. This approach is one heuristic for evaluating dependable systems. For simulations of other types of systems, different heuristics will be required, since no universally applicable heuristic has been discovered.

In *UltraSAN*, an importance sampling governor [17] is used to specify the heuristic. The governor is similar to a finite state machine, where each state corresponds to a different set of activity definitions. That is to say, in each governor state the user may redefine the activity time and case distribution of each timed activity in each SAN in the composed model. There are two restrictions that must be met for the simulation to be valid. All activity time distribution functions must have closed form complementary distribution functions, and case distributions cannot be altered to give zero probability to cases that have positive probability in the original model. Both of these

conditions are checked by the importance sampling simulator at run time.

The governor changes state based on the evolution of the simulation. Each state has a transition function consisting of a sequence of Boolean functions of the markings of the SANs in the composed model (called predicates), paired with a governor state name. When the marking of the model is updated, the transition function for the current governor state is checked. The first predicate that holds causes the governor to change to the associated governor state. When the governor changes state, the activity definitions associated with the new state are applied to the model. Activities whose definitions are changed by the governor transition are reactivated so that the new definition takes effect immediately. When an activity is reactivated its completion time is sampled from the new activity time distribution function. The details of this process are available in [17].

The utility of the governor is that dynamic importance sampling heuristics may be experimented with, where the bias on the model is altered according to the evolution of a sample path. Additional flexibility is available through the use of marking-dependent governor state definitions.

This approach differs from that taken in other tools, which are specialized to apply a single heuristic. The automatic application of a single heuristic is a feature found in more specialized tools such as SAVE [26], which is designed to handle Markovian dependability models. SAVE applies balanced failure biasing [27], a provably robust technique for importance sampling simulation of models representable in the SAVE model description language. Since *UltraSAN* is intended to handle a broader class of systems, the use of a single heuristic for automatic importance sampling is not practical.

After the model is specified, the next step is to assign values to the variables using the study editor.

IV Global Variable Assignment

The *study editor* facilitates assignment of numerical values to the global variables used in a model specification. In the process of assigning values, it provides a convenient way to organize the project in terms of multiple studies and experiments.

Conceptually, a *study* is analogous to a blank folder which can be used to collect experiments. The user can assign any meaning to a study. On the other hand, an experiment is a particular instantiation of the specified model. It consists of the model specification and a set of numerical values assigned to the global variables. Such a set of numerical values is called an *assigned set*.

Studies are created by invoking the study editor following model specification. In the study editor, studies can be added or deleted from the project, and experiments belonging to studies can be generated by assigning values to the global variables through either the range editor or the set editor. The *range editor* is used to generate experiments within a study where each global variable

takes on either a fixed value or a sequence of values over a range defined by an interval and a fixed increment. The sequence of numerical values is specified by selecting an initial value, a final value, and an increment factor, which can be additive or multiplicative. The range editor creates assigned sets by evaluating the cartesian product of the values specified for all global variables. Despite its limited scope, the range editor is very efficient for generating experiments for many of studies. When more flexibility is required, the set editor may be used.

The *set editor* is capable of generating experiments for all types of studies. It is capable of assigning any arbitrary sets of numerical values to the global variables. The set editor provides such a capability by allowing the user to directly add, delete or edit an assigned set. Furthermore, the set editor provides facilities for importing (reading) and exporting (writing) formatted files of assigned sets. This enables the user to assign sets to the global variables by editing a file (outside of *UltraSAN*, via "vi" for example) in a fixed format. Alternatively, the import file may be generated by a user's program. Once created, the import file can be read by the set editor, which then generates the corresponding experiments.

Upon reading assigned sets from an import file, the user can add, delete or edit any set. The export facility is useful when a user wants to use the assigned sets of one study in another study. This capability is desirable when the sets of values to be assigned to a study differ with the already assigned sets of another study in only a few elements. The user can first export the assigned sets of the existing study and then after making the changes can import the file for the new study.

After specifying the global variable assignments via the study editor, the next step is study construction.

V Study Construction

Depending on model characteristics, experiments in a study can be solved by analysis or simulation. Specifically, analytic solution can be used when 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. In this case, the underlying stochastic process for the model is a Markov process. In addition, mixes of exponential and deterministic activities can be handled, as long as no more than one concurrently enabled deterministic activity is enabled in any reachable stable marking. Otherwise, simulation must be used.

If analysis is the intended solution method, a stochastic process is constructed that supports solution of the variables in question. In most stochastic Petri net based tools, the set of reachable stable (also known as "tangible") markings is made to be the set of states in the Markov process. This choice causes two problems: 1) the state space generated for realistic systems is often unmanageably

large, and 2) not all variables can be written in terms of this process. In regard to the second problem, impulse rewards may be specified in a way that requires distinguishing between two different activity completions that result in the same transition in a Markov process constructed in this way.

One potential solution to this problem is to construct a process which keeps track (in its notion of state) of most recently completed activities, as well as reached stable markings. This process (called the activity-marking behavior, in [14]), supports all reward variables, but will result in state spaces that are very large. Alternatively, one can construct a stochastic process that is tailored to the variable in question, and exploits symmetries inherent in the model. When such symmetries occur, this approach, known as "reduced base model construction" can significantly reduce the size of the process that must be solved to obtain a solution. Symmetries are identified through the use of the replicate and join operation in the composed model editor (as described in the model specification section). This resulting state-level representation is known as a *reduced base model* [14].

The reduced base model is constructed from the model specification, which is known formally as a "composed SAN-based reward model." In order to understand the (non-standard) notion of state used in reduced base model construction, a more formal description of the model specification must be given. In particular, during model construction, SAN-based reward models are constructed by application of the replicate or join operations on input "SAN-based reward models" (SBRMs). Formally, a *SAN-based reward model* consists of a SAN, a reward structure defined on the SAN, and a set of common places (a subset of the set of places in the SAN). Reward structures are defined at the SAN level, rather than composed model, to insure that replicated SANs have the same reward structure, and hence behave symmetrically with respect to the specified performance variables. The resulting SBRM used in model construction is referred to as a *composed SAN-based reward model*, since it is the result of the composition of multiple SBRMs.

As alluded to in the model specification section, the replicate operation is useful when a system has two or more identical subsystems. The effect of the operation depends on a chosen set of places which is a subset of the common places of the input SBRM. These places are not replicated when the operation is carried out (informally, they are "held common" in the operation), and hence allow communication between the replicated submodels. The resulting SBRM consists of a new SAN, reward structure, and set of common places. The SAN is constructed by replicating the input SAN some number of times, holding the chosen set of places common to all SANs. The reward structure is constructed by assigning: 1) an impulse reward to each replicate activity equal to its value in the original model and, 2) a reward rate to each marking in the new SBRM equal to the sum of the rates assigned to the marking of each replica of the input SBRM. Finally, the set of common places in the new SBRM is taken to be the set of places held common during the operation.

Composition of multiple distinct SBRMs is done using the join operation. The *join* operation acts on SBRMs and produces a SBRM. The new SBRM is produced by identifying certain places

in the different input SANs as common in the new SAN. These common places allow arbitrary communication between the SANs that are joined; the only restriction being that the initial marking of the places to be combined is identical. The reward structure in the new SBRM is constructed by assigning: 1) an impulse reward to each activity in the joined SBRM equal to the reward of the corresponding activity in the input SBRM, and 2) a rate reward to each marking in the joined SBRM equal to the sum of the rate rewards of the markings in the input SBRM whose union constitutes the marking of the joined SBRM. The set of common places of new SBRM is the subset of the set of common places of each input SAN-based reward model that is made common, with zero or more models that are joined together.

The composed SAN-based reward model constructed in this way is input to the reduced base model construction procedure to generate the reduced base model. The notion of state employed is variable and depends on the structure of the composed model. One can think of the state as a set of impulse and rate rewards plus a state tree [16], where each node on the state tree corresponds in type and level with a node on the composed model tree (as specified in the composed model editor). Each node in a state tree has associated with it a subset of common places of the corresponding node in the composed model diagram. These places are those that are common at the node, but not at its parent node. The saving in state space size, relative to the standard state generation approach, is due to the fact that at each replicate node in the tree, we keep track of the *number* of submodels in particular markings, rather than the marking of *each* submodel. Proof that this notion of state produces Markov processes whenever the standard state generation algorithm can be found in [14].

Figure 5 shows the state tree of a particular state of the reduced base model for the multicomputer example. Places *interface_chips* and *memory_chips* are not common at the replicate node *R2*. Therefore, as shown in Figure 5, they are unique for each memory module, and can be different for different replica copies. At the join node, the set of common places is the union of the common places associated with the SANs of cpu_module, errorhandlers, io_port_module, and replicate node R2. Finally, among these places, *computer_failed* is common to all computer modules. The other places: *memory_failed*, *cpus*, *errorhandlers*, and *ioports* are not common, and hence unique to each computer module.

A reduced base model employing this notion of state can be generated directly without first generating the detailed (marking or activity-marking) behavior. This is done by first creating the state tree corresponding to the initial marking of the SAN, and then generating all possible next states by executing each activity that may complete in the state. A new state tree and impulse reward are generated corresponding to each possible next state that may be reached. For each possible next state, a non-zero rate (or probability, if a deterministic activity is enabled in the state, see [28]) from the original state to the reached state is added to the set of rates to other states from the originating state. If the reached state is new, then it is added to the set of states which need

to be explored. Generation of the reduced base model proceeds by selecting states from the set of unexplored states and repeating the above operations. The procedure terminates when there are no more states to explore. The resulting reduced base model serves as input to the analytic solvers, which determine the probabilistic nature of the selected performability variables.

Unlike the analytic solvers, which require a reduced base model, the simulators execute directly on the assigned model specifications. As depicted in Figure 1, three types of simulation programs can be generated: a *steady-state simulator* to solve long-run measures of performability variables, a *terminating simulator* to solve instant-of-time or interval-of-time performability variables, and an importance sampling terminating simulator. When simulation is the intended solution method, in the study construction phase the control panel links experiments (model descriptions) with the appropriate simulation library to generate the selected executable simulation programs. In the case of importance sampling simulation, the control panel links the governor description created by the importance sampling editor together with the experiment and importance sampling simulation library.

The next section details how the analytic and simulation solvers are used to determine the probabilistic behavior of the desired performability variables.

VI Study Solution

Solution of a set of base models, corresponding to one or more experiments within a study, is the final step in obtaining values for a set of performability variables specified for a model. The solution can be either by analytic (numerical) means or simulation, depending on model characteristics and choices made in the construction phase of model processing. This section details the solution techniques employed in *UltraSAN* to obtain analytic and simulative solutions.

A Analytic Solution Techniques

Analytic solvers can provide steady-state as well as transient solutions for the reduced base model generated during study construction. Figure 1 illustrates the six solvers available for analytic solutions. Three of the solvers, specifically, the *direct steady-state solver*, *iterative steady-state solver*, and *deterministic iterative steady-state solver* provide steady-state solutions for the instant-of-time variables. The other three solvers, the *transient instant-of-time solver*, *PDF interval-of-time solver* and *expected interval-of-time solver*, provide solutions for transient instant-of-time and interval-of-time variables. Except for the two interval-of-time solvers, each solver calculates the mean, variance, probability density function, and probability distribution function of the variable(s) for which it is intended to solve. The PDF interval-of-time solver calculates the probability distribution function of reward accumulated during a fixed interval of time $[0, t]$. Likewise, the expected interval-of-time

solver calculates the expected reward accumulated during a fixed interval of time. All solvers use a sparse matrix representation appropriate for the solution method employed.

The direct steady-state solver uses LU decomposition [29] to obtain the steady-state solution for instant-of-time variables. As with all steady-state solvers, it is applicable when the generated reduced base model is Markovian, and contains a single, irreducible, class of states. LU decomposition factors the transition matrix Q into the product of a lower triangular matrix L and an upper triangular matrix U such that $Q = LU$. Crout's algorithm [30] is incorporated to compute L and U in a memory efficient way. Pivoting in the solver is performed by using the improved generalized Markowitz strategy, as discussed by Osterby and Zlatev [31]. This strategy performs pivoting on the element for which the minimum fill-in is expected, subject to constraints on the magnitude of the pivot. Furthermore, the matrix elements that are less than some value (called the drop tolerance) are made zero (i.e., *dropped*) to retain the sparsity of the matrix during decomposition. In the end, iterative refinement is used to correct the final solution to a user-specified level of accuracy.

The iterative steady-state solver uses successive-overrelaxation to obtain steady-state instant-of-time solutions. Like the direct steady-state solver, it acts on the Markov process representation of a reduced base model. However, being an iterative method, it avoids fill-in problems inherent to the direct steady-state solver and therefore requires less memory. However, unlike LU decomposition, SOR is not guaranteed to converge to the correct solution for all models. In practice however, this does not seem to be a problem. The implementation solves the system of equations directly, without first adding the constraint that the probabilities sum to one, and normalizes only at the end and as necessary to keep the solution vector within bounds, as suggested in [32]. Selection of the acceleration factor is left to the user and defaults to a value of 1 (resulting in a Gauss-Seidel iteration).

The method used in the deterministic iterative steady-state solver was first developed for deterministic stochastic Petri nets [33, 34] with one concurrently-enabled deterministic transition, and later adapted to SANs in which there is no more than one concurrently-enabled deterministic activity [28]. In this solution method, a Markov chain is generated for marking in which a deterministic activity is enabled, representing the states reachable from the current state due to completions of exponential activities, until the deterministic activity either completes or is aborted. The solver makes use of successive-overrelaxation and uniformization to obtain the solution.

The transient instant-of-time solver and expected interval-of-time solver both use uniformization to obtain solutions. Uniformization works well when the time point of interest is not too large, relative to the highest rate activity in the model. The required Poisson probabilities are calculated using the method by Fox and Glynn [35] to avoid numerical difficulties.

The PDF interval-of-time solver [36] also uses uniformization to calculate the probability distribution function of the total reward accumulated during a fixed interval $[0, t]$. This solver is unique

among the existing interval-of-time solvers in that it can solve for reward variables containing both impulse and rate rewards. The solver calculates the PDF of reward accumulated during an interval by conditioning on possible numbers of transitions that may occur in an interval, and possible sequences of state transitions (paths), given a certain number of transitions have occurred. Since the number of possible paths may be very large, an implementation that considers all possible paths would require a very large amount of memory. This problem is avoided in *UltraSAN* by calculating a bound on the error induced in the desired measure by not considering each path, and discarding those with acceptably small (user specified) errors. In many cases as illustrated in [36], selective discarding of paths can dramatically reduce the space required for a solution while maintaining acceptable accuracy. Calculation of the PDF of reward accumulated, given that a particular path was taken, can also pose numerical difficulties. In this case, the solver makes selective use of multiprecision math to calculate several summations. Experience with the implementation has shown that this selective use of multiprecision operations avoids numerical difficulties, while maintaining acceptably fast execution.

B Simulative Solutions

As shown in Figure 1, three simulation-based solvers are available in *UltraSAN*. All three simulators exploit the hierarchical structure and symmetries introduced by the replicate operation in a composed SAN-based reward model to reduce the cost of future event list management. More specifically, multiple future events lists are employed, one corresponding to each leaf on the state tree or, in other words one corresponding to each set of replica submodels in a particular marking. These multiple future event lists allow the simulators to operate on "compound events," corresponding to a set of enabled replica activities, rather than on individual activities. By operating on compound events, rather than individual activities, the number of checks that must be made upon each activity completion is reduced. The details and a precise algorithm can be found in [16]. This algorithm is the basis of the state-change mechanism for all three simulators.

The steady-state simulator uses an iterative batch means method to estimate the means and/or variances of steady-state instant-of-time variables. The user selects a confidence level and a maximum desired half-width for the confidence interval corresponding to each variable, and the simulator executes batches until the confidence intervals for every performability variable satisfy the user's criteria. The terminating simulation solver estimates the means and/or variances for performability variables at a specific instant of time or over a given interval of time. It uses an iterative replication method to generate confidence intervals that satisfy the criteria specified by the user.

The third simulation-based solver is a terminating simulator that is designed to use importance sampling to increase the efficiency of simulation-based evaluation of rare event probabilities. The importance sampling terminating simulator estimates the mean of transient variables that are affected

by rare events by simulating the model under the influence of the governor (described in Section III), so that the interesting event is no longer rare. The simulator then compensates for the governor bias by weighting the observations by the likelihood ratio. The likelihood ratio is calculated by evaluating the ratio of the likelihood that a particular activity completes in a marking, and that the completion of this activity results in a particular next marking, in the original model to the likelihood of the same event in the governed model. The likelihood calculation is based on the work in [37], adapted for SAN simulation and implemented in *UltraSAN* [17]. It works for models where all activities have closed-form activity time distribution functions. The probability distribution function is needed to handle the reactivations of activities when their definitions are changed by the governor.

VII Example Results

The reliability of the multicomputer system example discussed earlier has been evaluated in [22] using a hierarchical evaluation technique based on numerical integration, and using SANs and the *UltraSAN* transient instant-of-time solver in [15]. Both studies focus on long mission times. In this section we discuss the evaluation of the unreliability of the multicomputer for shorter mission times, using the transient instant-of-time solver and the importance sampling terminating simulator.

Since there is no repair capability in the model of the multicomputer system, the system failed state is an absorbing state. Therefore, an interval-of-time variable such as unreliability can be evaluated as an instant-of-time variable. The reward structure for unreliability is simple, consisting of a rate reward of one assigned to the system failed state, with all other states having rate reward zero. Also, all impulse rewards are zero.

To evaluate the unreliability of the multicomputer for a given mission time using the transient instant-of-time solver, one simply generates the reduced base model and runs the solver. The transient instant-of-time solver is designed to handle multiple time points, so it is easy to evaluate the unreliability at a sequence of possible mission times. The answers for earlier instants of time are obtained with little additional effort compared to obtaining the answer for the last time point.

If importance sampling is to be used to evaluate the unreliability, a governor must be constructed to guide the importance sampling simulation. In this case, we chose a heuristic called "approximate forcing with balanced failure biasing [25]." Approximate forcing is based on the forcing technique developed in [24] to ensure that a failure event occurs before the mission is completed. In approximate forcing, failure events are accelerated so that a component failure occurs before the end of a mission with probability 0.8. When combined with balanced failure biasing, not only is the total probability of a failure event before mission completion increased, but the conditional probabilities of particular failure events, given a failure event occurs, are made equal.

A governor that implements approximate forcing with balanced failure biasing consists of two

states. In the first state, all failure events are given equal probability of occurring, and the total probability of a component failure before the mission time is increased to 0.8. Since the example model is Markovian, the new activity parameters are not difficult to calculate. The total rate of all failure events must be $\rho = -\log 0.2/t$, where t is the mission time in years, since that was the chosen time scale for this model. To assign equal probability to every possible failure event means equal probability must be assigned to every case of every failure activity.

Therefore, in the first governor state, the case distribution for each activity corresponding to a component failure event is the discrete uniform distribution. The rate parameter for a given failure activity is calculated using the formula $\lambda = c\rho/C$, where c is the number of cases on the activity, and C is the total number of cases in the model. For the multicomputer model with two computers, $C = 58$. For a mission time of thirty days, an activity with four cases would be assigned a rate $(4 \times -\log 0.2 \times 365.25)/(58 \times 30)$. Note that by balancing the probabilities of all possible failure events, we are increasing the conditional probability of a system failure due to an uncovered component failure, relative to the probability of system failure through exhaustion of available spares.

The second governor state is unbiased. Since there is no repair in the model, there are no activities competing with the failure activities, so it is sufficient to allow the model to evolve naturally from this point forward. An overly aggressive biasing scheme at this point would cause the simulation to waste time exploring very unlikely paths to system failure.

The multicomputer model with two computer modules was evaluated for missions from one to 390 days in length, using the transient instant-of-time solver and the importance sampling terminating simulator with the governor described above. The accuracy requested in the solver was 1×10^{-9}, while the simulator was set to run until the estimated half width of a 99% confidence interval was within 10% of the point estimate. The results from the two different solution approaches are plotted in Figure 6. As can be seen from the figure, the results are in close agreement. There is one errant point from the simulator, but the result obtained from the transient instant-of-time solver is still within the confidence interval.

Using importance sampling, with the same governor as before, the impact of increasing the redundancy at the computer level was evaluated for a mission time of thirty days. As shown in Figure 7, for a thirty day mission, increasing the redundancy led to an increase in unreliability. Due to the short mission time, coverage failures dominate, so increasing the number of components will make the system less reliable. This is because it is now more likely that some component will fail before the mission ends, which in turn increases the probability of a coverage failure.

The impact of perturbations in the coverage factors for RAM chip failures and computer module failures was evaluated using the transient instant-of-time solver for the multicomputer with two computer modules. These studies were carried out using the study definitions shown in Table 5. As can be seen from Figure 8, the unreliability is not very sensitive to variations in the RAM chip

failure coverage. However, variations in the coverage of computer module failures had a large impact on the unreliability, as shown in Figure 9.

The presented results indicate the importance of accurate estimates of coverage at the computer level, and constitute a strong argument for improvement of this coverage factor. On the other hand, perhaps money could be saved by using a less expensive fault tolerance mechanism to cover RAM chip failures, since the system unreliability seems relatively insensitive to this factor when mission times are short.

VIII Conclusions

We have described *UltraSAN*, an environment designed to facilitate the specification, construction, and solution of performability models represented as composed stochastic activity networks. *UltraSAN* has been in use for a period of about four years at several industrial and academic sites. Feedback from these users, and the development of new solution methods, has motivated continuous improvement of the software. As it stands today, *UltraSAN* is an environment that incorporates features and capabilities that do much to automate and simplify the task of model-based evaluation, and as evidenced by its application (e.g., [38, 39, 40, 15]), is capable of evaluating the performance, dependability, and performability of "real-world" systems. In addition to this capability, the tool serves as a test-bed in the development of new model construction and solution algorithms.

In particular, the control panel provides a simple interface to the package that enhances productivity and provides useful utilities such as automatic documentation. Models are specified as SANs, a model description language capable of describing a very large class of systems. Modeling conveniences such as cases and gates make it easier to specify the complicated behavior common in realistic models of modern computer systems and networks. The generation and solution of large numbers of production runs for a model is made easier through support for model specifications parameterized by global variables. Moreover, hierarchical composed models, created using the replicate and join operations, allow the modeling problem to be broken down into smaller, more manageable pieces. Wherever system symmetries exist, as identified by the replicate operation, they can be exploited during model construction to produce a smaller base model than possible with traditional techniques.

In the area of model construction, reduced base model construction allows *UltraSAN* to handle the typically large state spaces of models of "real-world" systems. The set of systems that can be evaluated analytically using *UltraSAN* has been enlarged by the recent addition of the capability to generate reduced base models for composed SANs that contain a mix of exponential and deterministic timed activities. Construction of models parameterized by global variables is automated through the control panel, which is capable of distributing model construction jobs across a network to utilize

other available computing resources.

Both analytic and simulation based solution methods have been implemented in *UltraSAN*. For analytic solution, six different solvers are available to compute performance, dependability and performability measures at an instant of time or over an interval of time, for models with mixes of exponential and deterministic timed activities, and for transient or steady-state. If simulation is to be used, three simulation-based solvers are available. The first two are traditional simulators; a terminating simulator for transient measures and an iterative batching simulator for steady-state measures. The third simulator is an importance sampling terminating simulator that implements importance sampling heuristics represented as governors. Using the control panel, model solution jobs can also be distributed across a network of workstations and carried out in parallel, improving productivity. The fact that three of the six solvers, and the importance sampling terminating simulator, were added after the initial release of *UltraSAN* demonstrates the success of the environment as a test-bed for the development and implementation of new solution techniques.

Acknowledgments

The success of this project is the result of the work of many people, in addition to the authors of this paper. In particular we would like to thank Burak Bulasaygun, Bruce McLeod, Aad van Moorsel, Bhavan Shah, and Adrian Suherman, for their contributions to the tool itself, and the members of the Performability Modeling Research Lab who tested the pre-release versions of the software. We would also like to thank the users of the released versions who made suggestions that helped improve the software. In particular, we would like to thank Steve West of IBM Corporation, Dave Krueger, Renee Langefels, and Tom Mihm of Motorola Satellite Communications, Kin Chan, John Chang, Ron Leighton and Kishore Patnam of US West Advanced Technologies, Lorenz Lercher of the Technical University of Vienna, and John Meyer, of the University of Michigan in this regard. We would also like to thank Sue Windsor for her careful reading of the manuscript.

REFERENCES

[1] R. Geist and K. Trivedi, "Reliability estimation of fault-tolerant systems: Tools and techniques," *Computer*, pp. 52–61, July 1990.

[2] B. R. Haverkort and I. G. Niemegeers, "Performability modelling tools and techniques." To appear in *Performance Evaluation*, 1994.

[3] K. S. Trivedi, B. R. Haverkort, A. Rindos, and V. Mainkar, "Techniques and tools for reliability and performance evaluation: Problems and perspectives," in *Computer Performance Evaluation Modelling Tools and Techniques* (G. Haring and G. Kotsis, eds.), (New York), pp. 1–24, Springer-Verlag, 1994.

[4] C. Lindemann, "DSPNexpress: A software package for the efficient solution of deterministic and stochastic Petri nets," in *Proceedings of the Sixth International Conference on Modelling Techniques and Tools for Computer Systems Performance Evaluation*, (Edinburgh, Great Britain), pp. 15–29, 1992.

[5] M. Bouissou, H. Bouhadana, M. Bannelier, and N. Villatte, "Knowledge modeling and reliability processing: The FIGARO language and associated tools," in *Proceedings of the Twenty-Third Annual International Symposium on Fault-Tolerant Computing*, (Toulouse, France), pp. 680–685, June 1993.

[6] G. Chiola, "A software package for analysis of generalized stochastic Petri net models," in *Proceedings of the International Workshop on Timed Petri Net Models*, (Torino, Italy), pp. 136–143, July 1985.

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

[8] G. Florin, P. Lonc, S. Natkin, and J. M. Toudic, "RDPS: A software package for the validation and evaluation of dependable computer systems," in *Proceedings of the Fifth IFAC Workshop Safety of Computer Control Systems (SAFECOMP'86)* (W. J. Quirk, ed.), (Pergamon, Sarlat, France), pp. 165–170, October 1986.

[9] G. Ciardo, J. Muppala, and K. S. Trivedi, "SPNP: Stochastic Petri net package," in *Proceedings of the Fourth International Workshop on Petri Nets and Performance Models*, (Kyoto, Japan), pp. 142–151, December 1989.

[10] C. Béounes, M. Aguéra, J. Arlat, S. Bachmann, C. Bourdeau, J.-E. Doucet, K. Kanoun, J.-C. Laprie, S. Metge, J. M. de Souza, D. Powell, and P. Spiesser, "SURF-2: A program for dependability evaluation of complex hardware and software systems," in *Proceedings of the Twenty-Third International Symposium on Fault-Tolerant Computing*, (Toulouse, France), pp. 668–673, June 1993.

[11] R. German, C. Kelling, A. Zimmermann, and G. Hommel, "Timenet: A toolkit for evaluating non-markovian stochastic petri-nets," tech. rep., Technische Universitat Berlin, Germany, 1994. Submitted for publication.

[12] 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, pp. 69–80, September 1991.

[13] M. Siegle, "Reduced Markov models of parallel programs with replicated processes," in *Proceedings of the Second Euromicro Workshop on Parallel and Distributed Processing*, (Malaga), pp. 1–8, January 1994.

[14] 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, pp. 25–36, January 1991.

[15] W. H. Sanders and L. M. Malhis, "Dependability evaluation using composed SAN-based reward models," *Journal of Parallel and Distributed Computing*, vol. 15, pp. 238–254, July 1992.

[16] W. H. Sanders and R. S. Freire, "Efficient simulation of hierarchical stochastic activity network models," *Discrete Event Dynamic Systems: Theory and Applications*, vol. 3, pp. 271–300, July 1993.

[17] W. D. Obal II and W. H. Sanders, "Importance sampling simulation in *UltraSAN*," *Simulation*, vol. 62, pp. 98–111, February 1994.

[18] W. D. Obal II and W. H. Sanders, "An environment for importance sampling based on stochastic activity networks," in *Proceedings of the 13th Symposium on Reliable Distributed Systems*, (Dana Point, California), pp. 64–73, October 1994.

[19] Center for Reliable and High Performance Computing, Coordinated Science Laboratory, University of Illinois, *UltraSAN Reference Manual*, 1995.

[20] A. Movaghar and J. F. Meyer, "Performability modeling with stochastic activity networks," in *Proceedings of 1984 Real-Time Systems Symposium*, (Austin, Texas), pp. 215–224, December 1984.

[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*, vol. 48 of *IMA Volumes in Mathematics and its Applications*. New York: Springer-Verlag, 1992.

[22] D. Lee, J. Abraham, D. Rennels, and G. Gilley, "A numerical technique for the hierarchical evaluation of large, closed fault-tolerant systems," in *Dependable Computing for Critical Applications 2,* Vol. 6 in Dependable Computing and Fault-Tolerant Systems (eds. A. Avižienis, H. Kopetz, J. C. Laprie) (J. F. Meyer and R. D. Schlicting, eds.), New York: Springer-Verlag, 1992.

[23] W. H. Sanders and J. F. Meyer, "A unified approach for specifying measures of performance, dependability, and performability," in *Dependable Computing for Critical Applications* (A. Avizienis and J. C. Laprie, eds.), vol. 4 of *Dependable Computing and Fault-Tolerant Systems*, pp. 215–238, Vienna: Springer Verlag, 1991.

[24] E. E. Lewis and F. Böhm, "Monte Carlo simulation of Markov unreliability models," *Nuclear Engineering and Design*, vol. 77, pp. 49–62, January 1984.

[25] V. F. Nicola, P. Heidelberger, and P. Shahabuddin, "Uniformization and exponential transformation: Techniques for fast simulation of highly dependable non-Markovian systems," in *Proceedings of the Twenty-Second Annual International Symposium on Fault-Tolerant Computing*, (Boston, Massachusetts), pp. 130–139, 1992.

[26] A. M. Blum, P. Heidelberger, S. S. Lavenberg, M. Nakayama, and P. Shahabuddin, "System availability estimator (SAVE) language reference and user's manual version 4.0," Tech. Rep. RA 219 S, IBM Thomas J. Watson Research Center, June 1993.

[27] P. Shahabuddin, *Simulation and Analysis of Highly Reliable Systems.* PhD thesis, Stanford University, 1990.

[28] B. P. Shah, "Analytic solution of stochastic activity networks with exponential and deterministic activities," Master's thesis, The University of Arizona, 1993.

[29] J. E. Tvedt, "Matrix representations and analytical solution methods for stochastic activity networks," Master's thesis, The University of Arizona, 1990.

[30] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling, *Numerical Recipes in C: The Art of Scientific Computing.* Cambridge University Press, 1988.

[31] O. Osterby and Z. Zlatev, *Lecture Notes in Computer Science: Direct Methods for Sparse Matrices.* Heidelberg: Springer-Verlag, 1983.

[32] U. R. Krieger, B. Müller-Clostermann, and M. Sczittnick, "Modeling and analysis of communication systems based on computational methods for Markov chains," *IEEE Journal on Selected Areas in Communications*, vol. 8, no. 9, pp. 1630–1648, 1990.

[33] C. Lindemann, "An improved numerical algorithm for calculating steady-state solutions of deterministic and stochastic Petri net models," in *Proceedings of the Fourth International Workshop on Petri Nets and Performance Models*, (Melbourne, Australia), pp. 176–185, 1991.

[34] M. A. Marsan and G. Chiola, "On Petri nets with deterministic and exponentially distributed firing times," in *Advances in Petri Nets 1986* (G. Rozenberg, ed.), Lecture Notes in Computer Science 266, pp. 132–145, Springer, 1987.

[35] B. L. Fox and P. W. Glynn, "Computing Poisson probabilities," *Communications of the ACM*, vol. 31, pp. 440–445, 1988.

[36] M. A. Qureshi and W. H. Sanders, "Reward model solution methods with impulse and rate rewards: An algorithm and numerical results," *Performance Evaluation*, vol. 20, pp. 413–436, August 1994.

[37] V. F. Nicola, M. K. Nakayama, P. Heidelberger, and A. Goyal, "Fast simulation of dependability models with general failure, repair and maintenance processes," in *Proceedings of the Twentieth Annual International Symposium on Fault-Tolerant Computing*, (Newcastle upon Tyne, United Kingdom), pp. 491–498, June 1990.

[38] L. M. Malhis, W. H. Sanders, and R. D. Schlichting, "Analytic performability evaluation of a group-oriented multicast protocol," Tech. Rep. PMRL 93-13, The University of Arizona, June 1993. Submitted for publication.

[39] B. D. McLeod and W. H. Sanders, "Performance evaluation of N-processor time warp using stochastic activity networks," Tech. Rep. PMRL 93-7, The University of Arizona, March 1993. Submitted for publication.

[40] W. H. Sanders, L. A. Kant, and A. Kudrimoti, "A modular method for evaluating the performance of picture archiving and communication systems," *Journal of Digital Imaging*, vol. 6, pp. 172–193, August 1993.

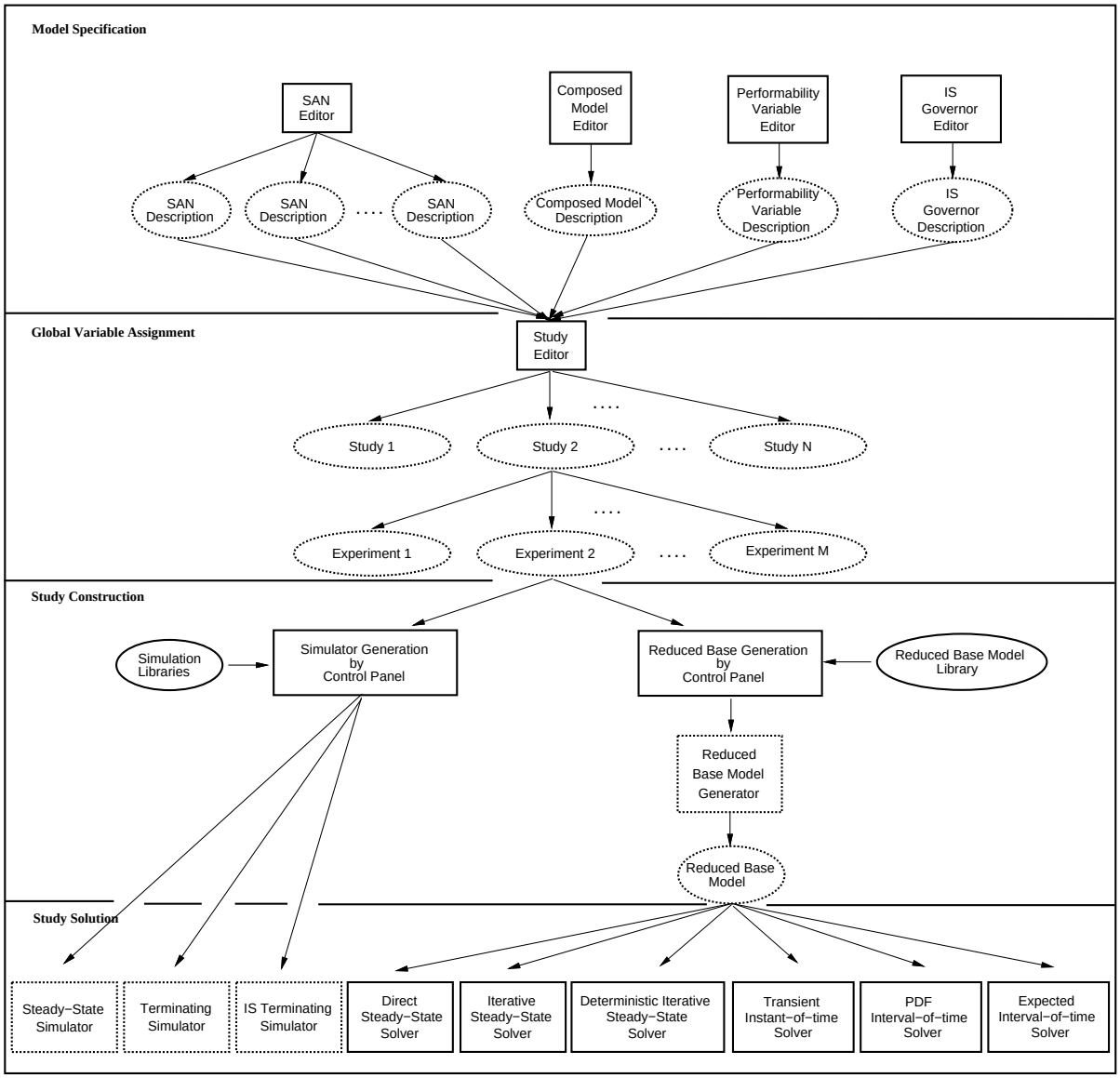

Figure 1: Organization of *UltraSAN*.

Table 1: Coverage Probabilities in Multicomputer System

Component	Coverage Probability	Global Variable Name
RAM Chip	0.998	RAM_cov
Memory Module	0.95	mem_cov
CPU Unit	0.995	CPU_cov
I/O Port	0.99	io_cov
Computer	0.95	comp_cov

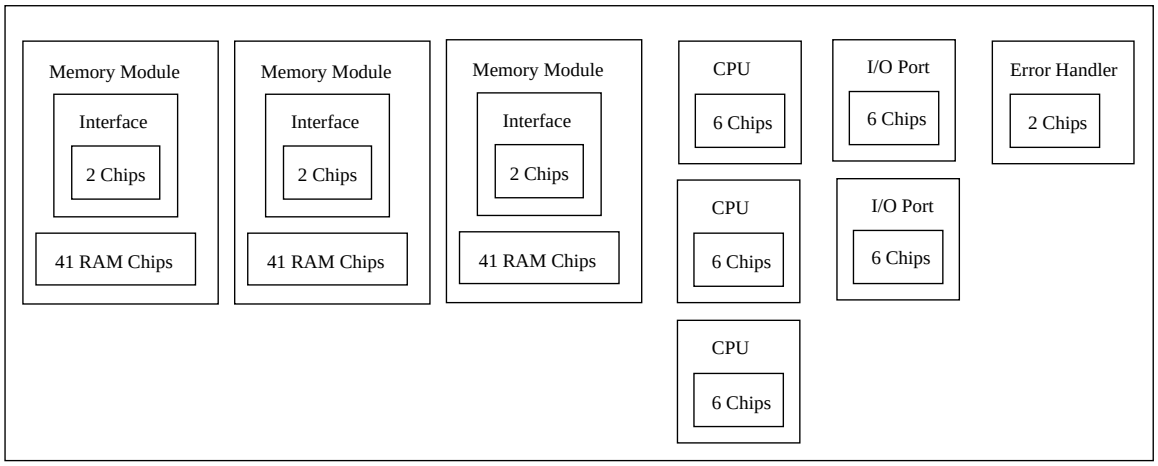

Figure 2: Block diagram of fault-tolerant computer

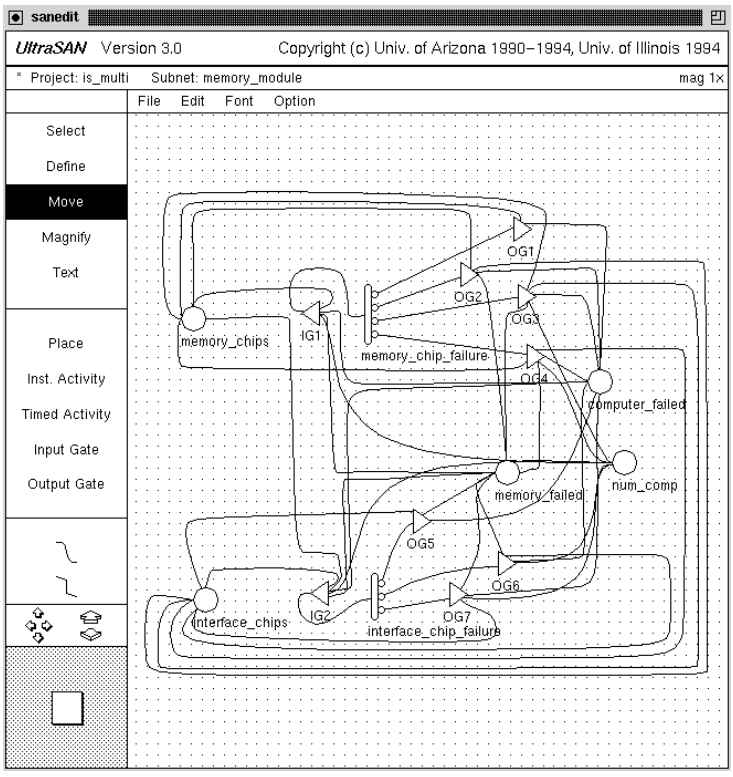

Figure 3: SAN model of a memory module, as specified in the SAN editor.

Table 2: Example Input Gate Definitions

Gate	*Definition*
IG1	Predicate $(MARK(memory\_chips) > 38)$ && $(MARK(computer\_failed) < 2)$ && $(MARK(memory\_failed) < 2)$
	Function *identity*
IG2	Predicate $(MARK(interface\_chips) > 1)$ && $(MARK(memory\_failed) < 2)$ && $(MARK(computer\_failed) < 2)$
	Function $MARK(memory\_chips) = 0;$

Table 3: Activity Case Probabilities for SAN Model *memory_module*

Activity	*Case*	*Probability*
interface_chip_failure	1	$GLOBAL\_D(mem\_cov)$
	2	$(1.0 - GLOBAL\_D(mem\_cov)) * GLOBAL\_D(comp\_cov)$
	3	$(1.0 - GLOBAL\_D(mem\_cov)) * (1.0 - GLOBAL\_D(comp\_cov))$
memory_chip_failure	1	*if* $(MARK(memory\_chips) == 39)$ $return(0.0);$ *else* $return(GLOBAL\_D(RAM\_cov));$
	2	*if* $(MARK(memory\_chips) == 39)$ $return(GLOBAL\_D(mem\_cov));$ *else* $return((1.0 - GLOBAL\_D(RAM\_cov)) * GLOBAL\_D(mem\_cov));$
	3	*if* $(MARK(memory\_chips) == 39)$ $return((1.0 - GLOBAL\_D(mem\_cov)) * GLOBAL\_D(comp\_cov));$ *else* $return((1.0 - GLOBAL\_D(RAM\_cov)) * (1.0 - GLOBAL\_D(mem\_cov)) * GLOBAL\_D(comp\_cov));$
	4	*if* $(MARK(memory\_chips) == 39)$ $return((1.0 - GLOBAL\_D(mem\_cov)) * (1.0 - GLOBAL\_D(comp\_cov)));$ *else* $return((1.0 - GLOBAL\_D(RAM\_cov)) * (1.0 - GLOBAL\_D(mem\_cov)) * (1.0 - GLOBAL\_D(comp\_cov)));$

Table 4: Example Output Gate Definition

Gate	*Definition*
OG2	$MARK(memory\_chips) = 0;$ $MARK(interface\_chips) = 0;$ $MARK(memory\_failed)++;$ *if* $(MARK(memory\_failed) > 1)$ $MARK(computer\_failed)++;$

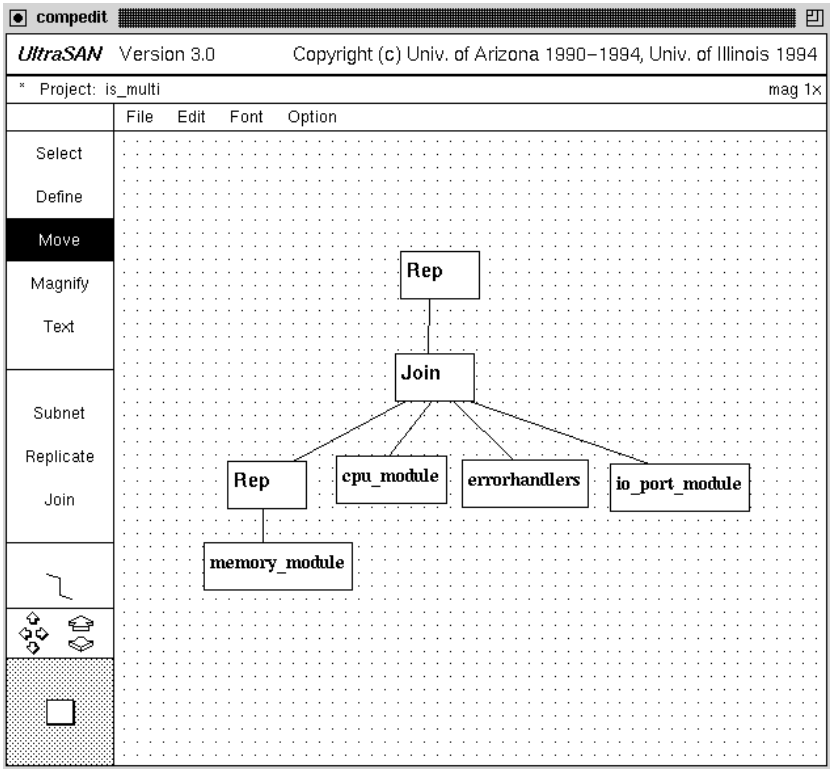

Figure 4: Composed model for multicomputer, as specified in the composed model editor.

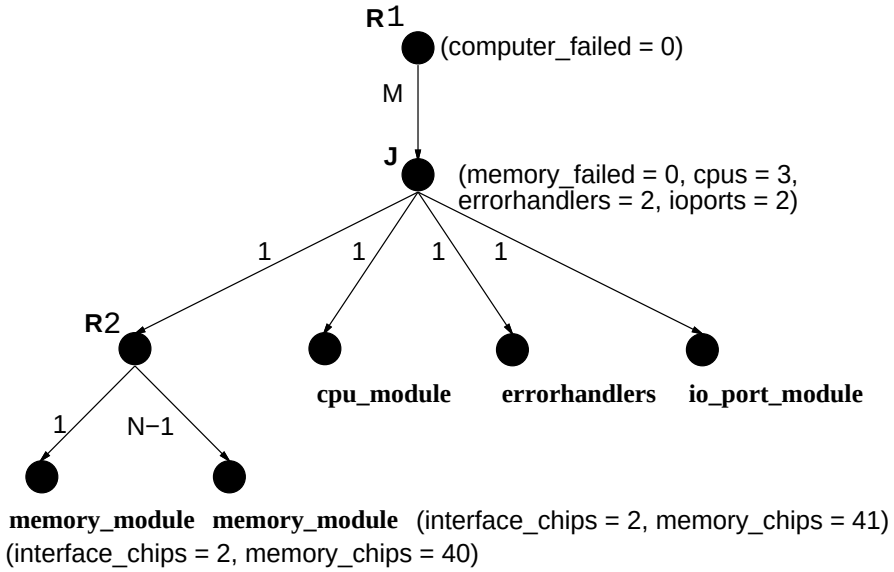

Figure 5: Example state tree for multicomputer model.

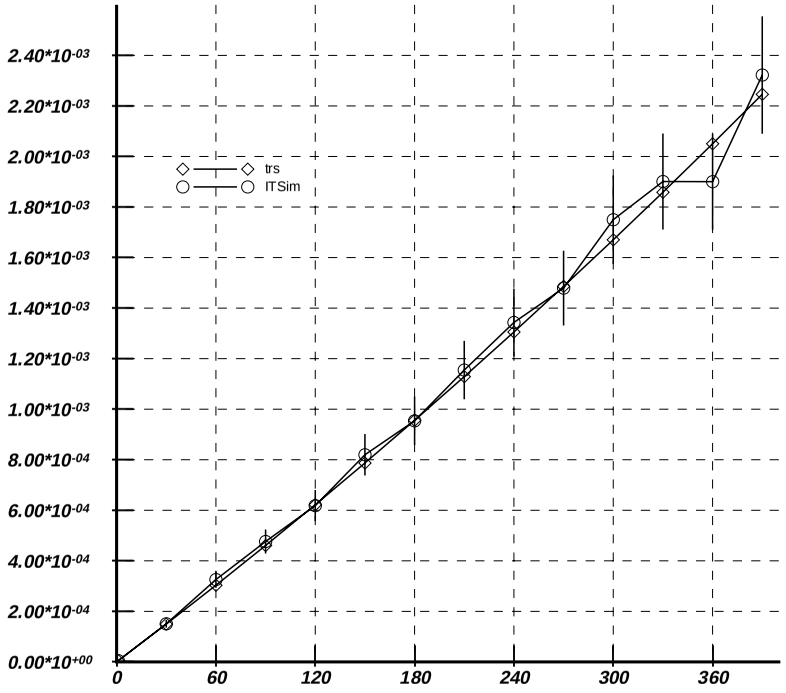

Figure 6: Unreliability versus mission time (in days).

Table 5: Study Editor Range Definitions for Project *multi_proc*

Study	Variable	Type	Range	Increment	Add./Mult.
vary_RAM_coverage					
	CPU_cov	double	0.995	–	–
	RAM_cov	double	[0.75, 1.0]	0.05	Add.
	comp_cov	double	0.95	–	–
	io_cov	double	0.99	–	–
	mem_cov	double	0.95	–	–
vary_comp_cov					
	CPU_cov	double	0.995	–	–
	RAM_cov	double	0.998	–	–
	comp_cov	double	[0.75, 1.0]	0.05	Add.
	io_cov	double	0.99	–	–
	mem_cov	double	0.95	–	–

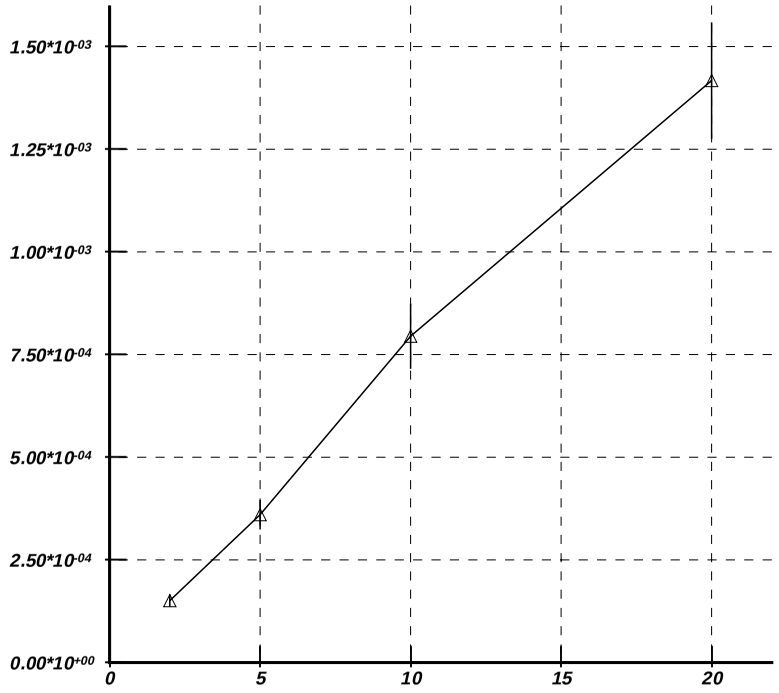

Figure 7: Unreliability versus number of computer modules.

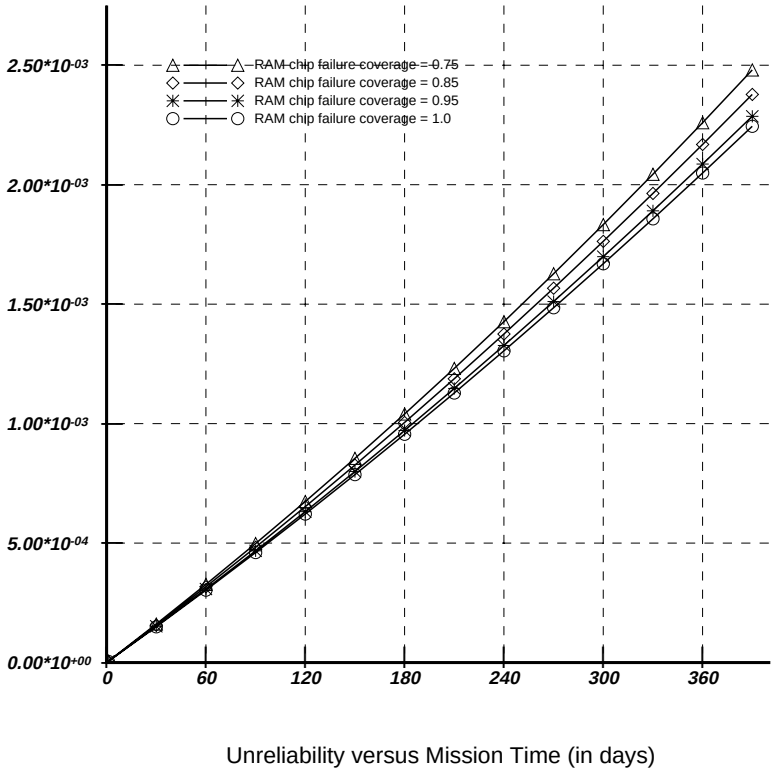

Figure 8: Impact of perturbation in RAM chip failure coverage factor.

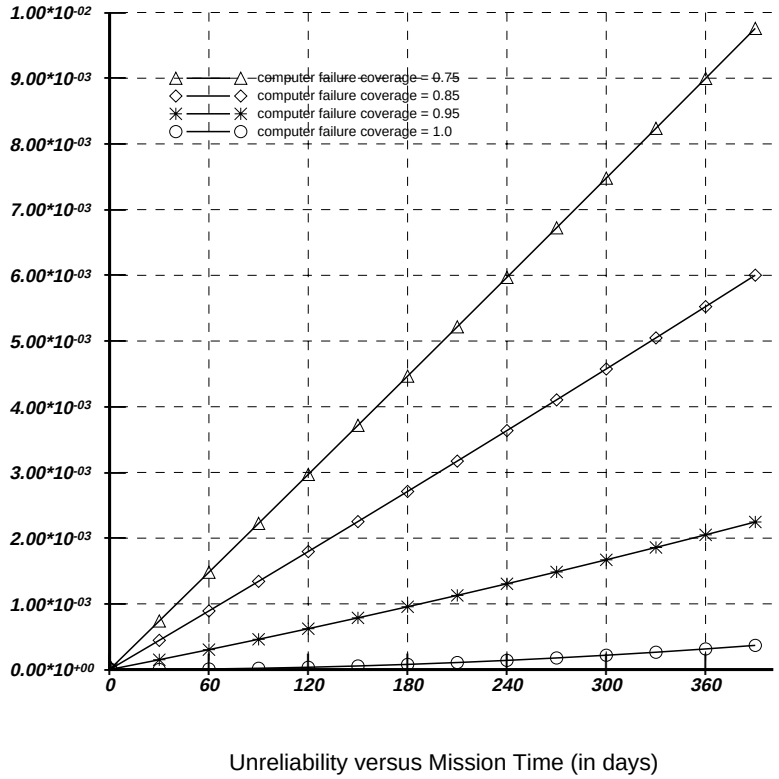

Figure 9: Impact of perturbation in computer failure coverage factor.