Software Demonstration in
Proceedings of the Twenty-Third International Symposium on Fault-Tolerant Computing
Toulouse, France, June 22–24, IEEE Press, 1993
PMRL Technical Report 93-9

DEPENDABILITY EVALUATION USING *UltraSAN*[*]

William H. Sanders and W. Douglas Obal II

Department of Electrical and Computer Engineering
The University of Arizona
Tucson, AZ 85721 USA
whs@ece.arizona.edu and obal@ece.arizona.edu

Abstract

Dependability evaluation is an important, but difficult, aspect of the design of fault-tolerant computer systems. This software demonstration highlights UltraSAN, *a stochastic activity network-based package for model-based performance, dependability, and performability evaluation of such systems. The package supports solution by both simulation-based and analytic methods. Of particular importance in the evaluation of dependable systems by analytic means is the control of an often explosive state space growth.* UltraSAN *has the potential to do this, using recently developed reduced base model construction techniques which exploit symmetries inherent in the SAN structure during construction of the underlying stochastic process representation. Four analytic solvers are available: SOR and LU decomposition for steady-state variables, and uniformization for transient instant-of-time and interval-of-time variables. For models with characteristics that preclude analytic solution, transient and steady-state simulation can be used. This demonstration illustrates the use of* UltraSAN *and reduced base model construction by considering the evaluation of a fault-tolerant distributed system. Model generation via an X Windows based graphical interface, stochastic process construction, and model solution, using numerical methods, are illustrated.*

1 Introduction

As systems have become larger and more complex, model based evaluation has become an important component of the design process. In the early stages of the process, models can provide insight into the characteristics of the system that will have the greatest impact on its performance. Later, models of competing designs can be evaluated to determine which design is superior. Since testing requires a working prototype, its use as the sole method of evaluation is often limited to small systems. In the case of a modification to an existing system, adequate testing of proposed changes often requires that the system be made unavailable for normal use for an unacceptably long period of time. Model-based evaluation can thus help a designer answer his questions before money is spent for a prototype or service is disrupted.

Markov processes are a popular mathematical tool for evaluating dependable systems. However, modeling complex systems at the state space level can be very cumbersome, since such systems tend to have very large state spaces. The size of the state space poses a problem for both specification and solution of the model. To ease the task of specifying models, one would prefer a high-level representation from which the detailed Markov process could be derived. Ideally, such a representation would be compact but highly flexible.

Stochastic extensions to Petri nets have been widely recognized as a viable and convenient tool for representing complex systems. Examples of tools utilizing stochastic Petri net representations are DSPNexpress [1], GreatSPN [2], HARP [3], METASAN [4], SPNP [5], and SURF-2 [6]. Although each of these tools provides a more convenient representation, they still generate very large Markov processes. On the other hand, a recently developed technique called *reduced base model construction* [7] can provide a smaller state space Markov process directly from a stochastic Petri net representation. For this paper, we focus on a particular extension to Petri nets called *stochastic activity networks* (SANs) [8].

UltraSAN [9] is a software package designed to take advantage of stochastic activity networks and reduced base model construction techniques to facilitate the modeling and evaluation of discrete event systems. In this demonstration, we will show the utility of *UltraSAN* in the analytic evaluation of dependable systems. Since an understanding of SANs will help clarify the demonstration, a brief discussion of their basic concepts is now provided.

2 Stochastic Activity Networks

Stochastic activity networks are comprised of *places*, *activities*, *input gates*, and *output gates*. Figure 1 provides an example SAN that will help demonstrate the use of these components. The SAN in Figure 1 is a simple model of a memory module

*This work was supported in part by Bell Communications Research, the Digital Equipment Corp. Faculty Program: Incentives for Excellence, IBM, Intel, Motorola, US West Advanced Technologies, and an equipment grant from AT&T.

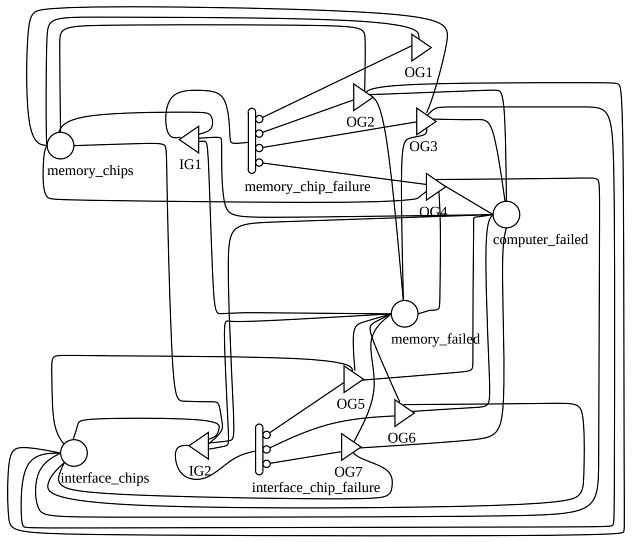


Figure 1: Memory module submodel.

Table 1: Memory Module Activity Time Distributions

Activity	Distribution
interface_chip_failure	exp(.0008766 * MARK(interface_chips))
memory_chip_failure	exp(.0008766 * MARK(memory_chips))

Table 2: Case Distribution for Activity 'memory_chip_failure'

Case	Probability
module_memory_chip_failure	
1	$if\ (MARK(memory\_chips)\ ==\ 39)$ $return(0.0);$ $else$ $return(0.998);$
2	$if\ (MARK(memory\_chips)\ ==\ 39)$ $return(0.95);$ $else$ $return(0.0019);$
3	$if\ (MARK(memory\_chips)\ ==\ 39)$ $return(0.0475);$ $else$ $return(0.000095);$
4	$if(MARK(memory\_chips)\ ==\ 39)$ $return(0.0025);$ $else$ $return(0.000005);$

Table 3: Predicate and Function for Input Gate 'IG1'

Gate	Enabling Predicate	Function
IG1	$(MARK(memory\_chips)\ >\ 38)$ && $(MARK(computer\_failed)\ <\ 2)$ && $(MARK(memory\_failed)\ <\ 2)$	*identity*

in the fault tolerant parallel processor system considered in [10]. Places hold *tokens*, which are used to represent system resources. For example, the place 'memory_chips' in Figure 1 initially holds 41 tokens, representing 41 available RAM chips. Other places in this model are 'interface_chips,' 'memory_failed,' and 'computer_failed.' The number of tokens in each place is the *marking* of the SAN. There are two kinds of activities. *Timed* activities represent delays in the system that impact its performance. The delay may be deterministic, or it may be described by a probability distribution function. In either case, the delay may depend on the marking of the SAN. Referring to our example, timed activity 'memory_chip_failure' represents the delay between failures of a memory chip. As shown in Table 1, the delay has been described by an exponential distribution with its rate parameter determined by the marking of 'memory_chips.' *Instantaneous* activities, on the other hand, are used to represent system tasks that are completed in a negligible amount of time relative to the performance measure of interest.

A SAN changes markings when an activity *completes*. When an activity is eligible to complete in a marking, we say that the activity is *enabled* in that marking. Upon the completion of an activity, the marking of the SAN is altered to reflect the new system state. Uncertainty about what happens upon completion of an activity is represented through its *cases*. The cases of an activity have a discrete probability distribution that may also be marking dependent. Activity 'memory_chip_failure' in Figure 1 has four cases with the probability distribution shown in Table 2. This marking dependent probability distribution models the coverage of a memory chip failure.

Input gates and output gates provide flexibility in specifying the markings in which an activity is enabled, and the marking changes that occur when the activity completes. Input gates are comprised of a *predicate* and a *function*, while output gates have only a function. The predicate of an input gate is a Boolean expression. When the value of the predicate is true, the input gate *holds*. The default input gate, represented by a line drawn from a place to an activity, holds when there is at least one token in the input place. A user specified input gate, such as 'IG1' in Figure 1, supersedes the default input gate. An activity is enabled when all of its input gates hold. When an activity completes, the functions in the input gates and output gates are executed. Input gate functions are executed first. The function in the default input gate removes a token from the input place. The default output gate, represented by a line from an activity to a place, has a function that adds one token to its output place. 'OG1' through 'OG7' are examples of nontrivial output gates. As one can see from Table 4, gate functions are written as a sequence of statements affecting the

Table 4: Function for Output Gate 'OG3'

Gate	Function
$OG3$	$MARK(memory\_chips)\ =\ 0;$ $MARK(interface\_chips)\ =\ 0;$ $if\ (\ (MARK(memory\_failed)\ ==\ 1)\ \&\&$ $\quad (MARK(computer\_failed)\ ==\ 0))\ \{$ $\qquad MARK(memory\_failed)\ =\ 2;$ $\qquad MARK(computer\_failed)\ =\ 2;\}$ $else\ \{$ $\qquad MARK(memory\_failed)\ =\ 2;$ $\qquad MARK(computer\_failed) + +;\}$

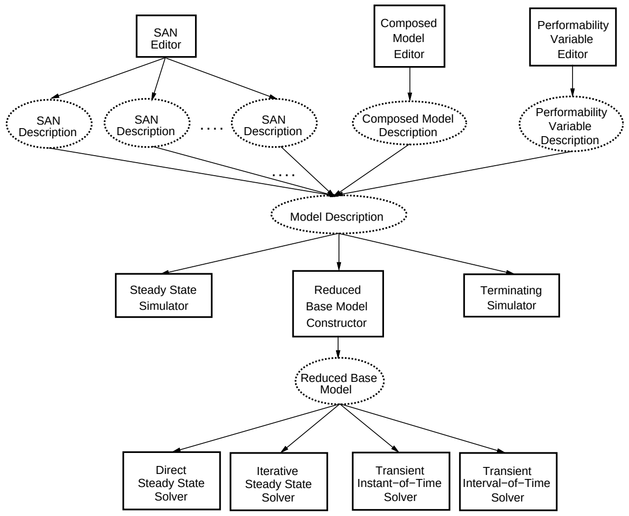

SAN Editor
Composed Model Editor
Performability Variable Editor
SAN Description
SAN Description
SAN Description
Composed Model Description
Performability Variable Description
Model Description
Steady State Simulator
Reduced Base Model Constructor
Terminating Simulator
Reduced Base Model
Direct Steady State Solver
Iterative Steady State Solver
Transient Instant-of-Time Solver
Transient Interval-of-Time Solver

Figure 2: *UltraSAN* organization

markings of places connected to the gate. Gate functions can, in general, use all of the statements available in the C programming language. In the memory module SAN example, the action taken upon completion of activity 'memory_chip_failure' depends on the coverage achieved. The action for each case has been specified in the output gate functions. Thus, the *identity* function is used in 'IG1' (see Table 3) to signify that no action is to be taken by that gate.

3 *UltraSAN*

UltraSAN is a tool for model-based evaluation of discrete event systems described by stochastic activity networks. It was implemented in C and C++ for workstations running UNIX[1] and the X Window System[2]. Currently, the software runs on DECstations, IBM RS6000s and Sun/4s. The organization of the package is shown in Figure 2. Modules in the package are divided into three main categories: model specification, reduced base model construction, and model solution.

[1]UNIX is a trademark of AT&T.

[2]X Window System is a trademark of the Massachusetts Institute of Technology.

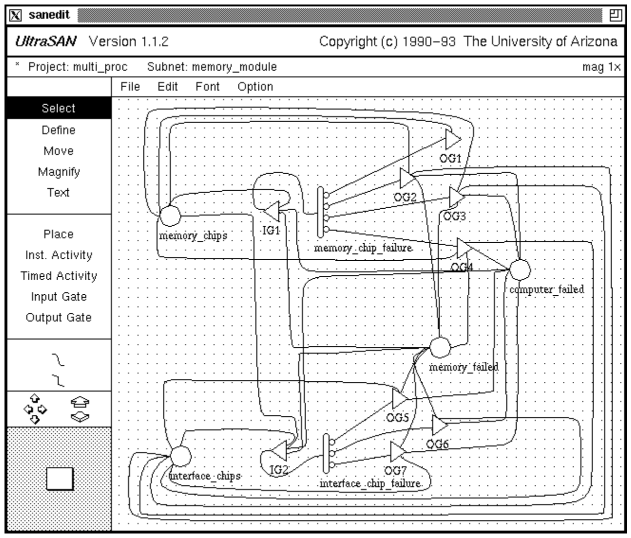


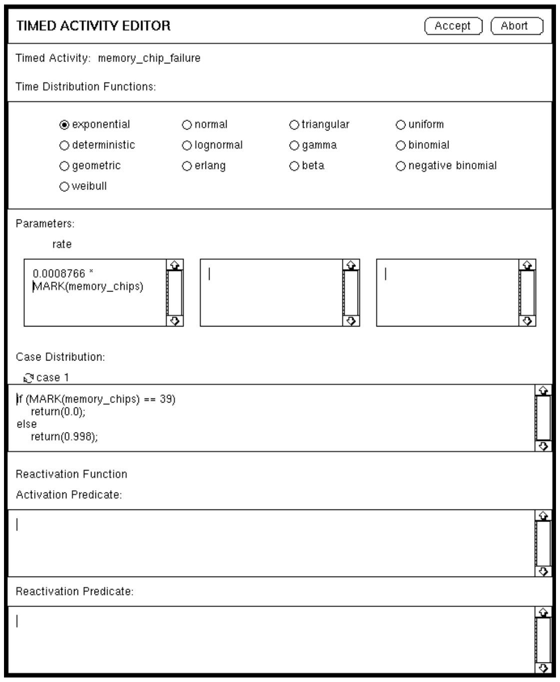
Figure 3: The memory module SAN in `sanedit`.

3.1 Model Specification

To ease the task of specifying a model for a complex system, *UltraSAN* provides a graphical user interface. Using the SAN editor (`sanedit`), one simply draws the SAN. Figure 3 is a screen dump showing the SAN model, as specified in `sanedit`, for the memory module component of the fault-tolerant parallel computer system introduced earlier. The details of a SAN component are specified through pop up editors. For example, Figure 4 shows the pop up editor for defining the parameters of the timed activity 'memory_chip_failure.' Pop up editors are also used to define gate predicates and functions. For example, Figure 5 shows the output gate editor for 'OG3' in the memory module SAN shown Figure 3.

UltraSAN provides hierarchical modeling capabilities in the form of a *composed model* [7]. A composed model is a tree created from SANs, *join* nodes, and *replicate* nodes. The composed model editor (`compedit`), shown in Figure 6, provides a graphical user interface that allows one to build composed models by simply drawing the tree. Each leaf of the tree is a SAN. All other nodes are either replicate nodes or join nodes.

A join node allows different SANs to communicate through a set of places identified as *common* among the SANs. To specify a join node, one must specify the places that will be shared. This operation is a convenient way to partition a large, complicated system into more manageable components. Each subsystem can be modeled separately and then joined. In *UltraSAN*, the definition of a node in the composed model tree is done through pop up editors. The join editor, shown in Figure 7, presents the places common at the nodes directly below it. One then chooses the places he would like to have shared among the nodes connected to the join node. Places can be shared by



Timed Activity: memory_chip_failure

Time Distribution Functions: [✓ exponential] normal triangular uniform deterministic lognormal gamma binomial geometric erlang beta negative binomial weibull

Parameters:

rate

```
0.0008766 *
MARK(memory_chips)
```

Case Distribution:

case 1

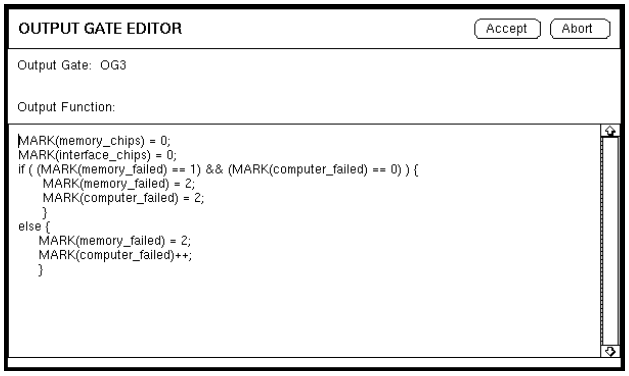
```
if (MARK(memory_chips) == 39)
    return(0.0);
else
    return(0.998);
```

Reactivation Function

Activation Predicate:

Reactivation Predicate:

Figure 4: The timed activity editor.



Output Gate: OG3

Output Function:

```
MARK(memory_chips) = 0;
MARK(interface_chips) = 0;
if ( (MARK(memory_failed) == 1) && (MARK(computer_failed) == 0) ) {
      MARK(memory_failed) = 2;
      MARK(computer_failed) = 2;
      }
else {
     MARK(memory_failed) = 2;
     MARK(computer_failed)++;
     }
```

Figure 5: The output gate editor.

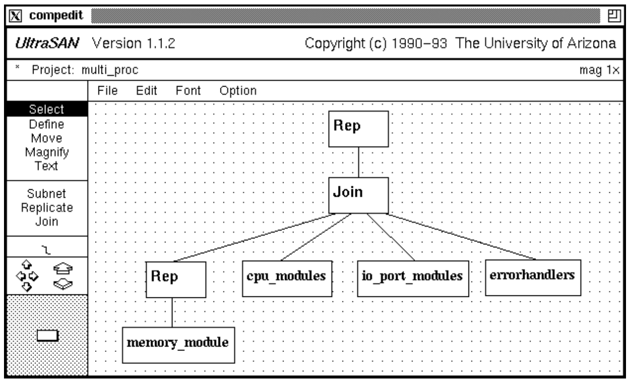
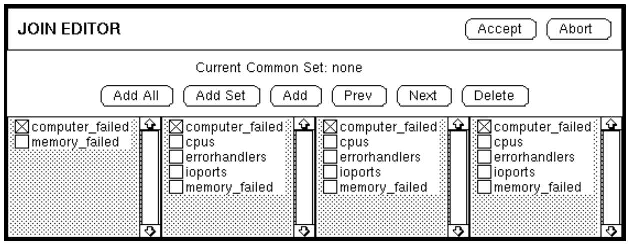


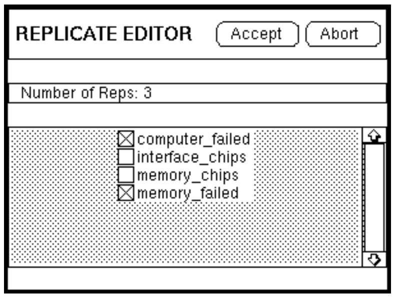
Figure 6: The composed model editor.



Current Common Set: none

☒ computer_failed	☒ computer_failed	☒ computer_failed	☒ computer_failed
☐ memory_failed	☐ cpus	☐ cpus	☐ cpus
	☐ errorhandlers	☐ errorhandlers	☐ errorhandlers
	☐ ioports	☐ ioports	☐ ioports
	☐ memory_failed	☐ memory_failed	☐ memory_failed

Figure 7: The join editor.



Number of Reps: 3

☒ computer_failed
☐ interface_chips
☐ memory_chips
☒ memory_failed

Figure 8: The replicate editor.

all of the connected nodes, or one may pick and choose which nodes will have access to which places.

A replicate node duplicates a SAN, holding a subset of the places in the SAN common to all replicas. To specify a replicate node, one identifies the set of common places and chooses the number of duplicates. Each place in the set of common places is shared by all replicas, while the other places are distinct within each replica. The replicate editor, shown in Figure 8, supplies the interface for specifying the common places and selecting the number of replicas. The replicate operation provides a straightforward method for noting symmetries in the system, thus providing a structure that can be used by *UltraSAN* to generate a smaller state space than possible with traditional SPN tools. In the presence of such symmetries, the reduced base model construction technique can produce a significant reduction in the state space of the model. We elaborate on reduced base model construction in the next subsection.

Note that in Figure 6 join nodes and replicate nodes have been used to operate on subtrees, not simply SANs. For example, the memory module SAN is replicated, and the result is joined with other system modules. While all of the places in the **cpu_modules**, **io_port_modules**, and **errorhandlers** SANs are eligible to be denoted as common at the join node, only the places made common in the replicate node are available from the **memory_module** SAN. Similarly, when the entire system is replicated by the replicate node at the root of the tree, only those places made common at the join node are eligible to be shared by the replicas.

The definition of the performance variables of in-

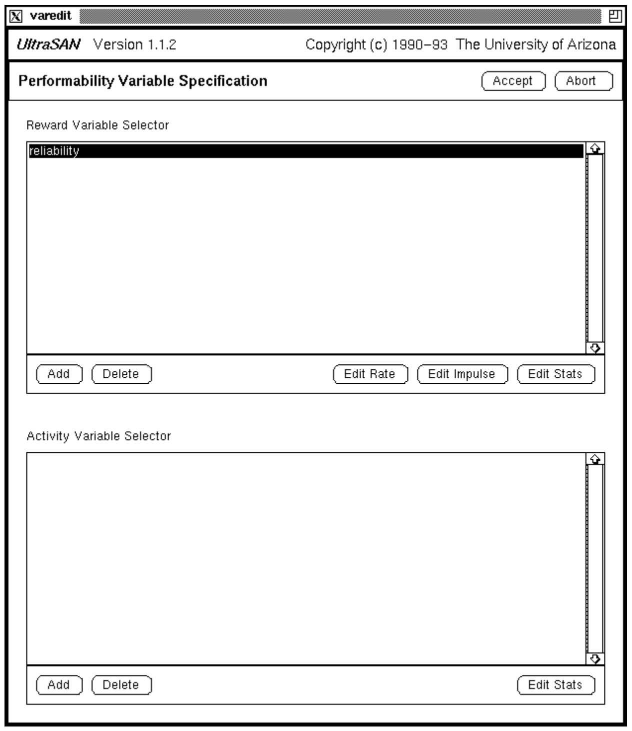


Figure 9: The variable editor.

terest is the last step of the model specification process in *UltraSAN*. The formalism used here is an extension of a "reward model" to the SAN level. There are two types of reward: *rate* rewards and *impulse* rewards. Rate rewards are associated with the marking of a SAN, while impulse rewards are associated with activity completions. The set of rate an impulse rewards associated with a SAN is referred to as a *reward structure* [11]. A performance variable is defined by a reward structure and a collection policy. For example, one might be interested in examining the reward at an instant of time, or he may prefer to look at the reward accumulated over an interval of time. Performance variables are specified in *UltraSAN* using the variable editor (`varedit`), shown in Figure 9. In `varedit`, rate rewards are specified as predicate/function pairs. The predicate is a Boolean expression that determines when the reward should be collected. The function is an expression that evaluates to the rate at which reward accumulates while the predicate holds.

3.2 Reduced Base Model Construction

Once the model has been completely specified, the next task is to solve the model for the performance variables of interest. Analytic methods can be applied to SAN models as long as all activities have exponentially distributed delays, the rate parameter of each exponential distribution depends only on the current marking of the SAN, and the state space size does not overwhelm the available hardware. The size of the state space of the stochastic process generated depends on the method by which it is constructed.

Traditionally, stochastic process representations of models specified using stochastic extensions to Petri nets have been derived by assigning one process state to each stable reachable marking of the Petri net. We call a process obtained through this method a *detailed base model.* A detailed base model has a rich structure, capable of supporting many performance variables. However, this rich structure is not free. Detailed base models can have very large state spaces.

Reduced base model construction [7] utilizes symmetries in the structure of the composed model, and knowledge of the specified performance variables, to determine a compact notion of state that is appropriate to the problem at hand. The generated state-level representation contains a set of states, transition rates between states, and the rate rewards for each state. The stochastic process representation derived using this technique supports only the specified performance variables, but can be much smaller than the detailed base model. Hence, we refer to the resulting stochastic process as the *reduced base model.* Once the reduced base model has been built, the state occupancy probabilities can be obtained via known stochastic process solution techniques. Together with the specified reward structure, these probabilities are used to determine the properties of the performance variables.

3.3 Model Solution

UltraSAN offers four analytic and two simulation-based model solution methods. If the model is analytically tractable, the user has a choice of two different solvers for steady-state analysis, a solver for transient instant-of-time variables, and a recently developed solver for transient analysis of interval-of-time variables. The first three solve for expected value, variance, probability density, and probability distribution function for each variable. The transient solver for interval-of-time variables provides the probability distribution function of reward accumulated over a fixed interval of time. The direct steady-state solver uses LU decomposition, while the iterative steady-state solver relies on successive over-relaxation to solve the process. For transient analysis of instant-of-time variables, a computationally stable version of the uniformization technique described by Gross and Miller [12] is employed. Transient solution for interval-of-time variables is done using the uniformization approach described in [13].

For models with characteristics that preclude analytic techniques, *UltraSAN* has two simulation-based solvers [14]. Both simulators take advantage of the structure of the composed model to speed the solution process, and both simulators can estimate the mean and/or variance of a performance variable. The steady-state simulator uses an iterative batching technique after a user specified initial transient to collect samples of instant-of-time variables in the steady-state. The terminating simulator uses the method of independent replications to obtain samples of instant-of-time, interval-of-time, and time-averaged interval-of-time variables. A replication ends when all variables have been sampled once. Replications are carried out until all variable estimators lie within their specified relative precision.

4 Planned Demonstration

We plan to demonstrate the utility of *UltraSAN* for evaluating dependable systems. Using the fault-tolerant parallel computer that has served as an example throughout this paper, we will show how models are specified using the *UltraSAN* user interface. The process of combining individual SAN models into a composed model will be discussed, along with the benefits and limitations of the replicate and join operations. Next, the performance variable editor will be presented, and reward structures for several performance variables will be built. When the model is completely specified, the reduced base model will be generated and its format discussed. Finally, several solvers will be employed to obtain exact answers for the presented performance variables.

5 Acknowledgments

We would like to acknowledge the hard work and support of the additional members of the *UltraSAN* development team: Bruce D. McLeod, M. Akber Qureshi, and Fransiskus K. Widjanarko. In addition, we would like to acknowledge the help of Luai Malhis, who developed the *UltraSAN* model used in this demo.

References

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

[2] 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, July 1985.

[3] S. J. Bravuso, J. B. Dugan, K. S. Trivedi, E. M. Rothman, and W. E. Smith, "Analysis of typical fault-tolerant architectures using HARP," *IEEE Transactions on Reliability*, vol. R-36, pp. 176–185, June 1987.

[4] 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.

[5] 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*, pp. 142–151, Kyoto, Japan, December 1989.

[6] K. Kanoun, "SURF-2 demonstration," in *Proceedings of the 22nd Annual International Symposium on Fault-Tolerant Computing*, Boston, Mass., July 1992. Tool demonstration.

[7] 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.

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

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

[10] 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.

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

[12] D. Gross and D. R. Miller, "The randomization technique as a modelling tool and solution procedure for transient Markov processes," *Operations Research*, vol. 32, no. 2, pp. 343–361, March-April 1984.

[13] M. A. Qureshi and W. H. Sanders, "Reward model solution methods with impulse and rate rewards: An algorithm and numerical results," Submitted to *Performance Evaluation*, September 1992.

[14] W. H. Sanders and R. S. Freire, "Efficient simulation of hierarchical stochastic activity network models," To appear in *Discrete Event Dynamic Systems: Theory and Applications*.