Journal on Parallel and Distributed Computing, special issue Petri Net Models of Parallel Computers, vol. 15, no. 3, July 1992, pp. 238-254.

DEPENDABILITY EVALUATION USING COMPOSED SAN-BASED REWARD MODELS*

William H. Sanders and Luai M. Malhis

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

whs@ece.arizona.edu and malhis@ece.arizona.edu

ABSTRACT

Dependability evaluation is an important, but difficult, aspect of the design of fault-tolerant parallel and distributed computing systems. One possible technique is to use Markov models, but if applied directly to realistic designs, this often results in large and intractable models. Many authors have investigated methods to avoid this explosive state-space growth, but have typically either solved the problem for a specific system design, or required manipulation of the model at the state-space level. Stochastic activity networks (SANs), a stochastic extension of Petri nets, together with recently developed reduced base model construction techniques, have the potential to avoid this state space growth at the SAN level for many parallel and distributed systems. This paper investigates this claim, by considering their application to three different systems: a fault-tolerant parallel computing system, a distributed database architecture, and a multiprocessor-multimemory system. We show that this method does indeed result in tractable Markov models for these systems, and argue that it can be applied to the dependability evaluation of many parallel and distributed systems.

Keywords: Dependability evaluation, Parallel and distributed systems, Fault-tolerant systems, Stochastic Petri nets, Reduced base model construction, and Stochastic activity networks.

*This work was funded in part by the Digital Equipment Corporation Faculty Program: Incentives for Excellence, Intel Corporation, and US West Advanced Technologies.

I Introduction

Dependability evaluation of parallel and distributed systems is an important and difficult undertaking. The inherent component redundancy in such systems often permits them to continue operating in the presence of faults, and application of fault-tolerant system design techniques can reduce the probability of total system failure. This is especially important for those systems that have life critical applications, such as flight control systems, during a crucial but short mission time. Estimation of the degree of this fault tolerance is often not easy, due to the complexity of such systems. Many techniques have been developed to attempt to do this via modeling (for a good survey of these techniques, see [1].) Both simulation and analytic-based methods have been proposed. Among analytic methods, Markov process have been an important method, but suffer from the fact that the system to be modeled must be described at the state level, and the number of states that must be considered can be very large, on the order of hundreds of thousands of states or more.

Higher-level representations can be constructed to ease in the description process. Examples, among others, include the block diagrams used in SAVE [2], stochastic Petri nets used in HARP [3], SPNP [4], GreatSPN [5], and METASAN [6], dynamic queueing networks [7], production rules [8], and objects [9]. While each of these representations aided in the model description process, they, if not coupled with some state reduction technique, still lead to very large Markov models. Such large models make it difficult to perform transient and steady state analysis using current solution techniques and available computers. Thus, large effort has been devoted to state space reduction. Techniques to do this can be approximate, or exact. Examples of techniques that yield approximate results include model decomposition and state aggregation [10], state trimming [11], generating the most probable states [12], and the use of numerical techniques augmented by semi-Markov models [13].

Exact techniques are less common, but show promise for medium size problems. Most methods of this type are based on lumping theorems for Markov processes [14]. Unfortunately, direct application of such methods would require prior generation of the entire state space of the original model, which would be prohibitively expensive. Several recently developed techniques for stochastic Petri nets have the potential to avoid this problem. Most of these rely on using colored or high-level Petri nets to identify appropriate lumpings in the stochastic process [15, 16, 17, **?**, 19]. Another technique, known as reduced base model construction [20], makes use of both the structure of the net and the performance/dependabiliity/performability variables considered to determine an appropriate notion of state. Using this method, models (in this case a stochastic extension of Petri nets known as stochastic activity networks) are replicated and joined together to form a complete, or composed, model.

In addition to possible state space explosions, a second problem that must be dealt with in modeling fault-tolerant parallel and distributed systems is coverage [21]. Including coverage factors in a model is not a simple task [22]. Lee et al. [13] point out that most existing modeling tools fail to accurately model coverage factors at all levels in large, hierarchal systems. Stochastic activity networks have the potential to do this, through the use of their "case" construct.

The purpose of this paper is to investigate the applicability of stochastic activity networks and reduced base model construction to medium to large fault-tolerant parallel and distributed systems. We do this by considering the dependability evaluation of three systems: a fault-tolerant parallel computing system, a distributed database architecture, and a multiprocessor-multimemory system. Each of these systems has been considered before in the literature. In the case of the first two, only approximate solutions have been given, and in the case of the third, an exact answer was obtained by direct construction of a Markov process at the state level. In this paper, we obtain exact (to the numerical accuracy of the machine used) results using stochastic activity networks and reduced base model construction, as implemented in a software package known as *UltraSAN* [23]. We show that this method allows systems to be modeled using a convenient formalism (stochastic activity networks), produces tractable Markov models for the considered systems, and argue that it can be applied to the dependability evaluation of many parallel and distributed systems.

The remainder of this paper is organized as follows. Section II gives a review of the models and methods used: stochastic activity networks, composed SAN-based reward models, and reduced base model construction. Section III gives the analysis of the three systems considered. Finally, section IV gives remarks about the generality of these techniques in modeling fault-tolerant parallel and distributed systems.

II Modeling Framework

The models employed are hierarchical, with stochastic activity networks of individual components replicated and joined together with other models.

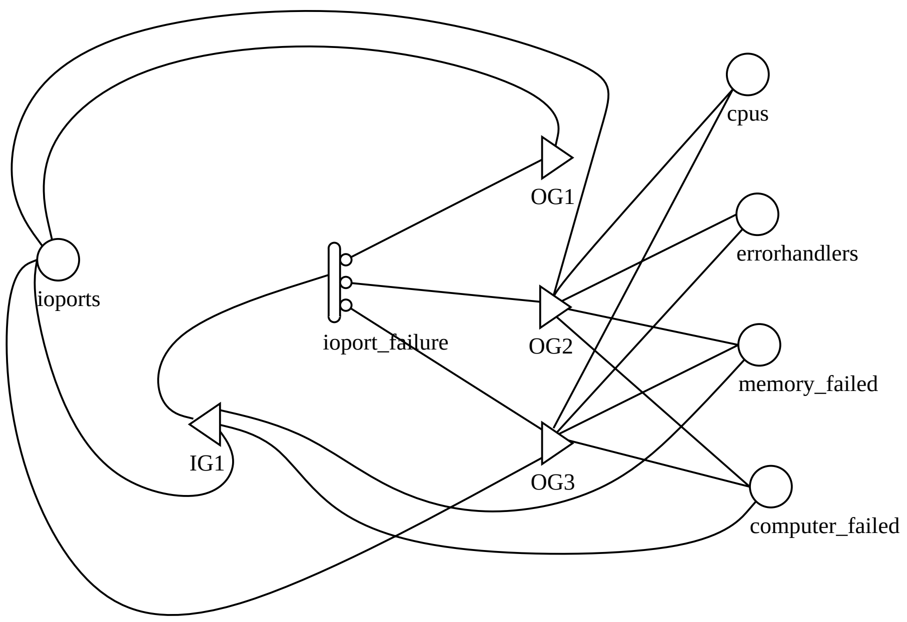

Figure 1: I/O Ports Submodel

A Stochastic Activity Networks

Stochastic activity networks (SANs) [24, 25, 26], are a stochastic extension to Petri nets. Structurally, they consist of *activities, places, input gates,* and *output gates.* Activities, which are similar to transitions in normal Petri nets, are of two types: *timed* and *instantaneous.* Timed activities represent activities of the modeled system whose durations impact the system's ability to perform. Instantaneous activities, on the other hand, represent system activities which, relative to the dependability variables in question, complete in a negligible amount of time.

To illustrate activities and other SAN components, we consider a simple model of I/O ports in a fault-tolerant parallel processor system, which will also be used in a later section (see Figure 1). In this model, a timed activity is used to represent an I/O port failure. As shown in table 1, times associated with activities can depend on the marking of the network. *Cases* associated with activities (represented as small circles on one side of an activity) permit the realization of uncertainty concerning what happens when an activity completes, such as whether a fault is covered, in this model. As with activity times, case probabilities can be (and typically are) dependent on place markings. For example, see table 2 which gives the case probabilities associated with activity *ioport_failure.*

Table 1: I/O Port Modules Activity Time Distributions

Activity	**Distribution**
$ioport\_failure$	expon(0.0052596 * MARK(ioports))

Table 2: I/O Port Modules Case Probabilities for Activities

Case	**Probability**
port_module_ioport_failure	
1	$if\ (MARK(ioports)\ ==\ 2)$ $\quad return(0.99);$ $else$ $\quad return(0.0);$
2	$if\ (MARK(ioports)\ ==\ 2)$ $\quad return(0.0095);$ $else$ $\quad return(0.95);$
3	$if\ (MARK(ioports)\ ==\ 2)$ $\quad return(0.0005);$ $else$ $\quad return(0.05);$

Table 3: I/O Port Modules Input Gate Predicates and Functions

Gate	**Enabling Predicate**	**Function**
$IG1$	$(MARK(ioports) > 0)$ && $(MARK(memory\_failed) < 2)$ && $(MARK(computer\_failed) < 2)$	*identity*

Places are used to represent the "state" of a system (e.g., place *ioports, cpus, errorhandlers, memory_failed* and *computer_failed*, in Figure 1) and may contain *tokens* (e.g., each small black dot in *ioports* is a token). For example, the number of tokens in *ioports* represents the number of operational I/O ports in the system. When an activity *completes*, one token is removed from each of the places directly connected (i.e., not through a gate) to the input of the activity and one token added to each of the places directly connected to the output of the activity.

Input gates and *output gates* permit greater flexibility in defining enabling and completion rules than with regular Petri nets. In particular, input gates have *enabling predicates* and *functions*, while output gates have only *functions*. The enabling predicate can be any computable predicate (taking on true and false values) of the places connected to it, and, as seen in that which follows, controls the enabling of an attached activity. The function associated with each input gate describes an action (change in marking) that will occur upon completion of the activity. As with predicates, gate functions can be any computable function on the places connected to the gate. Activities are *enabled* if there is at least one token in each of the places directly connected to the activity and if the predicate of each connected input gate is true (i.e., *holds*). For example, input gate *IG1* controls the enabling of activity *ioport_failure*. This gate specifies that the associated activity is to be enabled if there is at least one I/O port that has not failed and the computer that this module is associated with has not failed (i.e., the number of tokens in *memory_failed* and *computer_failed* is less than two.) The function associated with this gate is the identity function, i.e., no change is made to the places *ioports, memory_failed,* and *computer_failed* when this gate is executed.

Output gates, together with directly connected output places, are used to specify the action to be taken upon completion of an activity. The function associated with each output gate can be any computable function on the connected places. For example, as shown in

Table 4: I/O Port Modules Output Gate Functions

Gate	**Function**
$OG1$	$if\ (MARK(ioports)\ ==\ 2)$ $\quad MARK(ioports)--;$
$OG2$	$MARK(cpus)\ =\ 0;$ $MARK(ioports)\ =\ 0;$ $MARK(errorhandlers)\ =\ 0;$ $MARK(memory\_failed)\ =\ 2;$ $MARK(computer\_failed)++;$
$OG3$	$MARK(cpus)\ =\ 0;$ $MARK(ioports)\ =\ 0;$ $MARK(errorhandlers)\ =\ 0;$ $MARK(memory\_failed)\ =\ 2;$ $MARK(computer\_failed)\ =\ 2;$

table 4, output gate *OG1* decrements the number of tokens in place *ioports* if the current marking of the place is equal to two.

Note the use of gates in the modeling of this subsystem as a stochastic activity network. It can be shown that SANs are equivalent to stochastic Petri net models without gates, in the sense that every gate can be replaced by a subnetwork of instantaneous activities, places, and arcs, preserving the behavior of the system. However, there are many logical actions and enabling conditions that are much more easily represented in functional form, rather than graphically, using logic implemented as instantaneous subnetworks. Gates allow the flexibility to use both specification methods: functions and predicates when they are more natural, and arcs, instantaneous activities, and places when they are more appropriate. While the use of gates does preclude using structural analysis of the type described in [27], we feel is a reasonable alternative for many models. If one desires to use structural analysis, the use of gates can be avoided.

B Variable Specification

The formalism used to represent variables at the stochastic activity network level is an extension of the idea of a "reward model" [28]. Traditional reward models consist of three components: a stochastic process, a reward structure, and a performance variable defined in terms of the stochastic process and reward structure. The reward structure typically consists of two types of rewards: an *impulse* reward that is associated with each state change, and a *rate* reward that is associated with the time spent in a state. We extend this idea to the SAN level, where impulse rewards can naturally be assigned to activity completions, and rate rewards can be assigned to particular numbers of tokens in places.

Dependability (as well as performance and performability [29]) variables can then be easily defined in terms of these rewards. In particular, we have defined a family of variables, distinguished by the intervals of time on which they depend. Three categories of variables are distinguished: *instant-of-time variables*, which represent the status of the SAN at either a particular time t or in steady state, *interval-of-time variables*, which represent the total accumulated reward obtained from executing the SAN for a particular interval of time, and *time-averaged interval-of-time variables*, which represent the time-averaged accumulated reward obtained from executing the SAN for a particular interval of time. For the second and third categories, three types of variables are considered. The first type represents the total or time-averaged reward accumulated during some interval $[t, t+l]$. The second type corresponds to an interval of length l as t goes to infinity, and is useful in representing the reward that is accumulated during some interval of finite length in steady state. The final variable type corresponds to the total or time-averaged reward accumulated during an interval starting at t and of length l as $l \rightarrow \infty$.

For example, the reliability for some mission time [0,t], when there is no repair from a failed state, can be formulated in terms of an instant-of-time variable at time t. Steady-state availability, on the other hand, can be determined in terms of an instant-of-time variable in the limit as $t \rightarrow \infty$. More generally, these variables and the reward structure discussed previously give us the ability to represent many traditional and non-traditional measures of performability, including queueing time, queue length, processor utilization, steady-state and interval availability, reliability, and productivity [31].

C Composed SAN-based Reward Models

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

Reduced based model construction proceeds by taking a more general view in the base model construction process, in which knowledge of the structure of the network and performance variable is used to determine the notion of state to use in the resulting stochastic process [20]. Traditional model construction methods for stochastic Petri nets and extensions do not use this approach. They typically obtain the base model stochastic process by choosing the reachable stable markings of the network to be the states of the process. To distinguish between these two approaches, a stochastic process which supports a large class of variables is referred to as a *detailed base model*, while a stochastic process constructed specifically to support a designated performance variable, dependability for example, is a *reduced base model*.

The reduced base model construction methods developed in [20] are applicable for a restricted, but common class of stochastic activity networks and performance or dependability variables. This class includes stochastic activity networks that have some replicated components, such as many parallel and distributed systems, and variables that are regular in the sense that they assign equal rewards to identical events and markings in different replicated components. These methods abstract unnecessary information from the base model without rendering it unsolvable.

To use this approach, a complete (or "composed") model is built from one or more SAN submodels using "replicate" and "join" operations. Formally, the resulting model is known as a *composed SAN-based reward model* (SBRM). The *replicate* operation replicates a SAN and associated reward structure a certain number of times, holding some subset of its places, called its "distinguished places" in [20], common to all resulting submodels. It is through these distinguished places that the replicated submodels interact. Each replica will have values for the impulse and rate rewards specified as in the original submodel. The replicate operation allows one to construct composed models that consist of several identical component submodels.

The combination of several different submodels is accomplished using the *join* operation. Informally, the effect of the operation is to produce a composed model which is a combination of the individual submodels. Again, distinguished places play an important role in the construction operation. In this case, however, a *list* of places is associated with each component submodel. The first place in each of the lists is merged to form a single place, the second place is merged to form another place, and so on. We allow particular elements on the lists to be null, permitting the case where certain places are created from a proper subset of the submodels joined. (See examples in section III.)

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

Reduced base model construction is accomplished by using a description of a composed model, including one or more SAN submodels and dependability variables, to build a compact state-level representation. This representation consists of a set of states, rates to transition between each pair of states, and a rate reward for each state. The notion of state employed is variable and depends on the structure of the composed model.

After a reduced base model is constructed, solution for the desired variables proceeds using known stochastic process solution techniques to obtain the appropriate state-occupancy probabilities. These probabilities, together with the rate reward calculated for each state while generating the reduced base model, are then used to generate the mean, variance, probability density function, and probability distribution function for each variable. Three analytical solvers are available in *UltraSAN* [23] for dependability evaluation: a direct steady-state solver, an iterative steady-state solver, and a transient solver. The two steady-state solvers can be used to determine the long-run behavior of a system such as the availability of the multiprocessor system. The direct steady-state solver is based on the LU decomposition technique, while the iterative steady-state solver is based on the successive over-relaxation technique. The transient solver uses a computationally stable version of the randomization technique developed by Gross and Miller [30]. A solver that can solve the probability distribution function of *interval-of-time variables* is under development.

III Dependability Evaluation Using Composed SAN-Based Reward Models

We now investigate the use of composed SAN-based reward models in the dependability evaluation of fault-tolerant parallel and distributed computer systems.

A Fault-Tolerant Parallel Computing System

The first system considered is a highly redundant fault-tolerant multiprocessor system, taken from Lee et al. [13] and shown in Figure 2. As shown here, at the highest level the

Table 5: Coverage Probabilities

Redundant Component	**Fault Coverage Probability**
RAM Chip	0.998
Memory Module	0.95
CPU Unit	0.995
I/O Port	0.99
Computer	0.95

system consists of 2 computers, where each computer is composed of 3 memory modules of which 1 is a spare module, 3 CPU units of which 1 is a spare unit, 2 I/O ports of which 1 is a spare port, and 2 non-redundant error-handling chips. Lee et al. consider a system that is similar but that has 10 computers at the highest level, but are only able to obtain an approximate solution, without bounds.

Internally, each memory module consists of 41 RAM chips, 2 of which are spare chips, and 2 interface chips. Each CPU unit and each I/O port consists of 6 non-redundant chips. The system is considered operational if at least 1 computer is operational. A computer is classified as operational if, of its components, at least 2 memory modules, at least 2 CPU units, at least 1 I/O port, and the 2 error-handling chips are functioning. A memory module is operational if at least 39 of its 41 RAM chips, and its 2 interface chips are working. Where there is redundancy (available spares) at any level of system hierarchy, there is a coverage factor associated with the component failure at that level. For example, following the parameter values used by Lee et al., if one CPU unit fails, there is a 0.995 probability that the failed unit will be replaced by the spare unit, if available, and the corresponding computer continues to operate. On the other hand, there is also a 0.005 probability that fault recovery procedure will fail and the corresponding computer will cease to operate. Table 5 shows the redundant components and the fault coverage probability associated with each component. Finally, the failure rate of every chip in the system, as in [13], is assumed to be 100 failures per billion hours.

In terms of the fault-tolerant parallel processor example, a composed model for the entire system is built by first defining SAN submodels to represent the failure of various components in the system (e.g., the model of the I/O system failure shown in Figure 1). We then use the replicate and join operations previously defined to construct a complete composed model. Figure 3 shows a composed model for a system with n identical memory

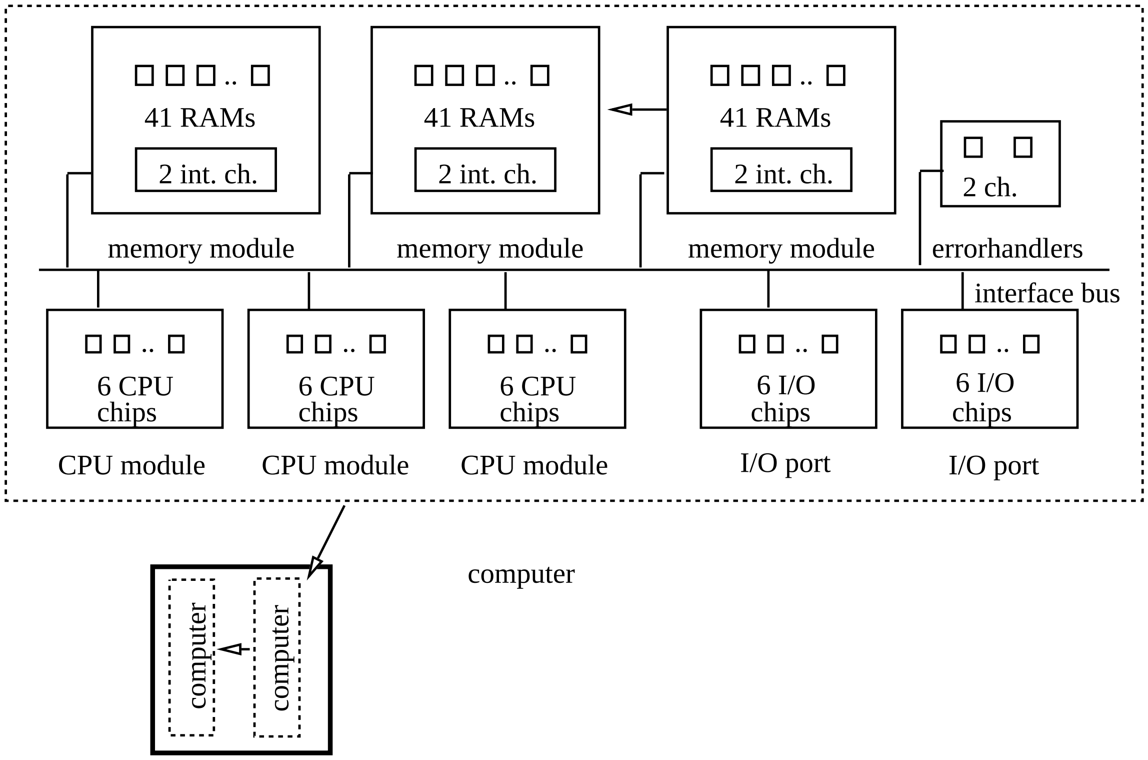

Figure 2: Fault-Tolerant Multiprocessor System

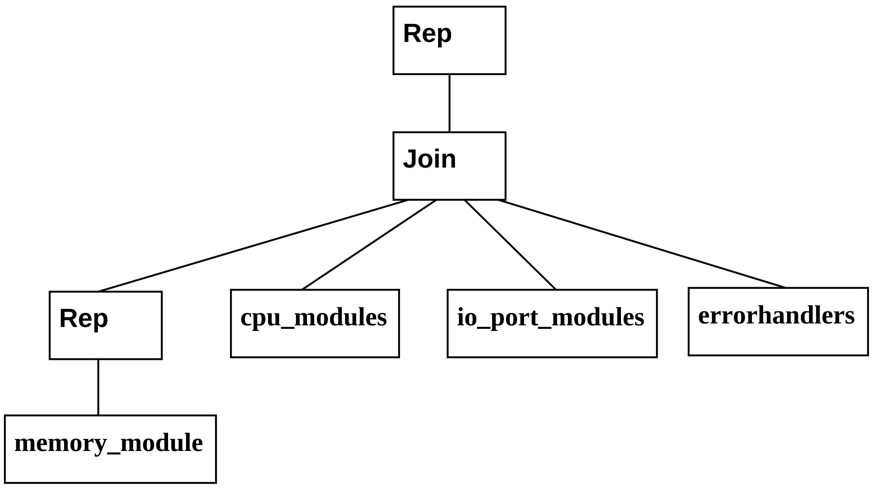

Figure 3: Composed Model for Fault-Tolerant Multiprocessor System

modules. The leaf nodes represent the individual submodels, together with their reward structures. The memory module is replicated n times with the place *computer_failed*, see Figure 5, held common among all replicas. This submodel is then joined to the I/O ports model (Figure 1), CPUs failure model (Figure 4), and error-handler model (Figure 6) by joining the place named *computer_failed* in each submodel to form a single new place.

The SAN-based reward model used to determine the reliability of the system is shown in Figure 3. The leaf nodes of the tree, which are labeled **memory_module**, **cpu_modules**, **io_port_modules**, and **errorhandlers**, correspond to the SAN submodels of the reliability of the memory module, the 3 CPU units, the 2 I/O ports, and the 2 error-handling chips, respectively. The internal node labeled **Rep** corresponds to replicating the memory module 3 times, which equals the number of memory modules in one computer. The internal node labeled **Join** joins the 4 SAN submodels to construct the SAN model of one computer. Finally, the joined SAN model of one computer is replicated 2 times to generate the final reliability SAN model of the multiprocessor system.

The SAN submodel of the CPUs is called **cpu_modules** and is shown in Figure 4. The places named *cpus* and *computer_failed* represent the current state of the CPUs, and the current state of the multiprocessor system, respectively. The number of tokens in *cpus* represents the number of operational CPUs in a given computer. The number of tokens in *computer_failed* indicates the number of computers that have failed in the system. In addition, the places labeled *ioports, errorhandlers,* and *memory_failed* are also included in

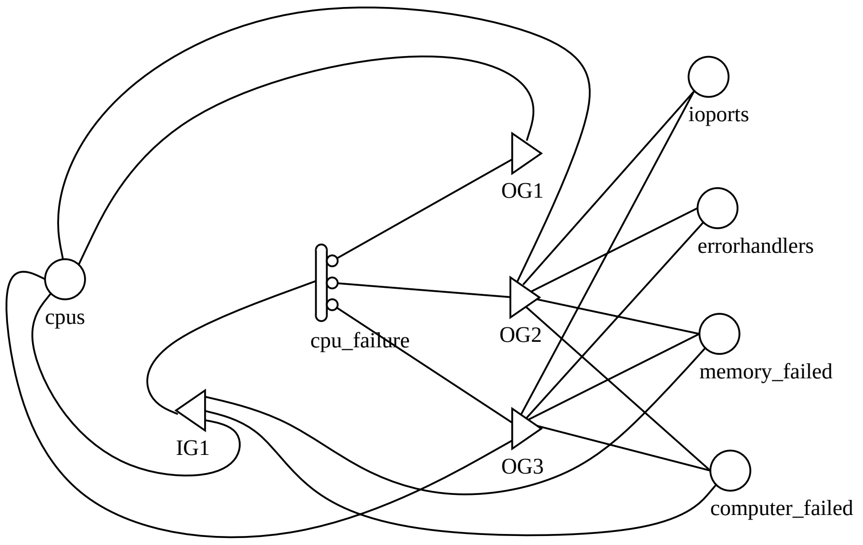

Figure 4: SAN Submodel of **cpu_modules**

this model to aid in reducing the state space size of the overall system model by lumping as many failed states together as possible.

Additional state lumping (with respect to that provided by the reduced base model construction method) can be achieved because once a computer fails, there is no need to keep track of which of its components failed that caused the computer to fail. More specifically, by assuming that all internal components of the failed computer have failed, then the states that represent a computer failure due to a CPU failure, a memory module failure, an I/O port failure, or an error-handling chip failure are combined into a single state. The marking of the combined state is reached by setting the number of tokens in each of the places *cpus, ioports,* and *errorhandlers* to zero, setting the number of tokens in *memory_failed* to 2, and incrementing the the number of tokens in *computer_failed.*

When the timed activity *cpu_failure* completes, one CPU unit is assumed to have failed. This activity completion rate is shown in table 6. If a spare CPU unit is available (i.e., the *MARK(cpus) == 3*), there are three cases associated with this activity completion. The first case represents a successful coverage of a CPU unit failure. If this case occurs, the failed CPU unit is replaced by the spare unit and its corresponding computer continues to operate. The second case represents the situation where a CPU unit failure occurs that is not covered, but where the failure of its corresponding computer is covered. If this occurs and a spare computer is available, the failed computer is replaced by the spare

Table 6: **cpu_modules** Activity Time Distributions

Activity	**Distribution**
cpu_failure	expon(0.0052596 * MARK(cpus))

computer and the system continues to operate. However, if no spare computer is available, the multiprocessor system fails. The third case is where neither the CPU failure nor the corresponding computer failure are covered, resulting in a total system failure.

On the other hand, if a spare CPU is not available (i.e., *MARK(cpus) == 2*), then a CPU unit failure causes a computer failure. In this marking, there are two possible actions associated with the completion of activity *cpu_failure*. The first situation is that in which a spare computer is available (i.e., *MARK(computer_failed) == 0*). In this case, the computer failure can be covered. The second case is that no spare computer is available, i.e., *MARK(computer_failed) == 1*, which, in turn, causes the system to fail. Table 7 shows the case numbers and the probabilities associated with each case. It is clear that these case probabilities are marking dependent since the coverage factors are dependent on the state of the system.

The input gate *IG1* is used to determine whether the timed activity *cpu_failure* is enabled in the current marking, hence, can complete. This activity is enabled only if at least 2 working CPU units are available and their corresponding computer and the system have not failed. Table 8 shows the predicate and function associated with this gate.

The output gates, *OG1, OG2* and *OG3*, are used to determine the next marking based on the current marking and the case chosen when *cpu_failure* completes. Table 9 lists the output gates and the function of each gate.

Another way to model the failure of CPU modules would be to model the failure of a single CPU module as a SAN and replicate this model three times. However, since the failure of any chip inside the CPU module causes the CPU to fail, and each chip is assumed to have an exponentially distributed failure rate, the failure rate of one CPU module is just the sum of the failure rates of the 6 CPU chips. Therefore, modeling the failure of one CPU module, then replicating this model three times, is equivalent to the **cpu_modules** submodel described above. Either way will generate an equivalent number of states. In contrast, a significant state space reduction can be achieved by modeling one memory

Table 7: **cpu_modules** Case Probabilities for Activities

Case	Probability
	module_cpu_failure
1	$if\ (MARK(cpus)\ ==\ 3)$ $\quad return(0.995);$ $else$ $\quad return(0.0);$
2	$if\ (MARK(cpus)\ ==\ 3)$ $\quad return(0.00475);$ $else$ $\quad return(0.95);$
3	$if\ (MARK(cpus)\ ==\ 3)$ $\quad return(0.00025);$ $else$ $\quad return(0.05);$

Table 8: **cpu_modules** Input Gate Predicates and Functions

Gate	Enabling Predicate	Function
$IG1$	$(MARK(cpus) > 1)$ && $(MARK(memory\_failed) < 2)$ && $(MARK(computer\_failed) < 2)$	*identity*

Table 9: **cpu_modules** Output Gate Functions

Gate	Function
$OG1$	$if\ (MARK(cpus)\ ==\ 3)$ $\quad MARK(cpus)--;$
$OG2$	$MARK(cpus)\ =\ 0;$ $MARK(ioports)\ =\ 0;$ $MARK(errorhandlers)\ =\ 0;$ $MARK(memory\_failed)\ =\ 2;$ $MARK(computer\_failed)++;$
$OG3$	$MARK(cpus)\ =\ 0;$ $MARK(ioports)\ =\ 0;$ $MARK(errorhandlers)\ =\ 0;$ $MARK(memory\_failed)\ =\ 2;$ $MARK(computer\_failed)\ =\ 2;$

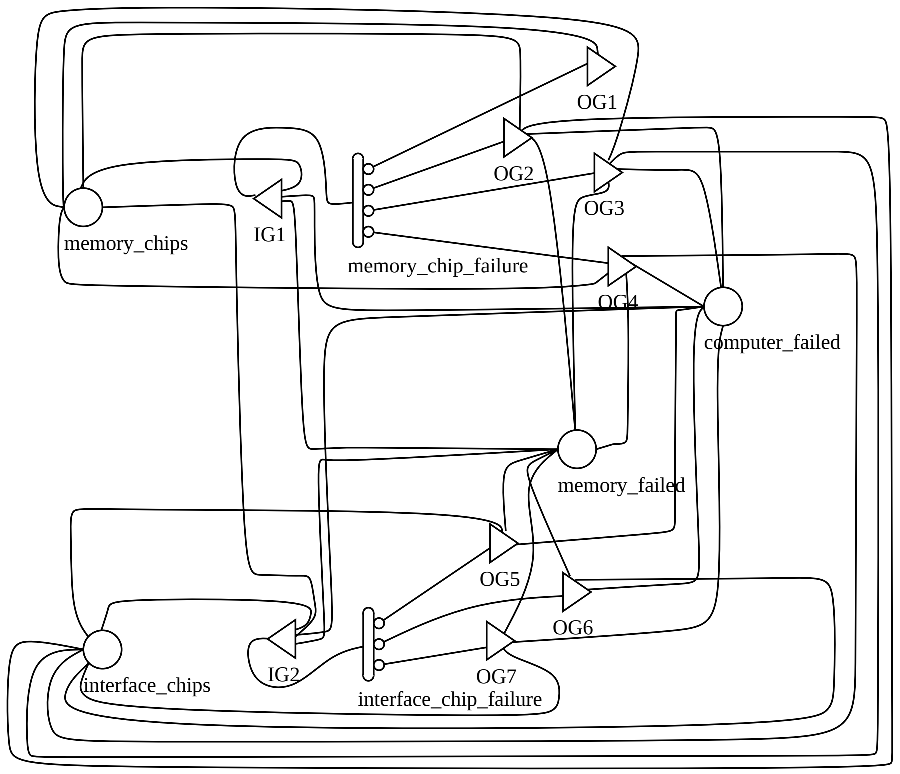

Figure 5: SAN Submodel of the **memory_module**

module as a SAN and replicating this model three times as compared to modeling the failure of the three memory module in one SAN. This is because the failure of a single RAM chip does not cause the memory module to fail and, hence, a memory module can not be modeled as a single entity.

The SAN submodel of the I/O ports, memory module, and the two error-handling chips are shown in Figures 1, 5, and 6, respectively. The line of reasoning followed in modeling each of these components is similar to that followed in modeling the **cpu_modules**. (Note similarity between 1 and 4.) In the joined model of one computer, places that have the same name are joined together, hence, treated as single places among all system submodels. The tables associated with each submodel are also shown in Appendix A.

Using the reward structure formalism defined in section II, we define the multiprocessor system reliability in terms of the following rate reward

$$\mathcal{C}(a) = 0, \quad \forall \;\; activities \; a$$

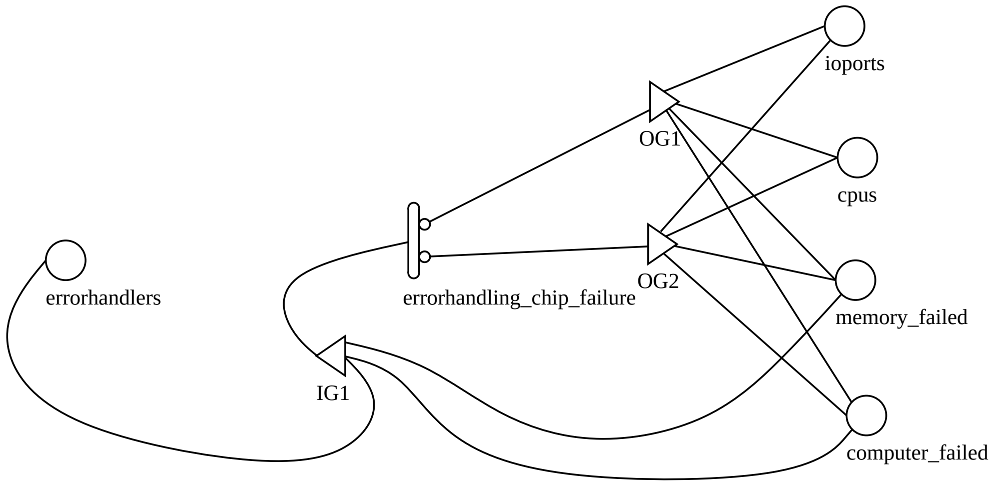

Figure 6: SAN Submodel of the **errorhandlers**

$$\mathcal{R}(\nu) = \begin{cases} 0 & \text{if } \nu = \{(computer\_failed, 2)\} \\ 1 & \text{otherwise,} \end{cases}$$

where $\mathcal{C}(a)$ is the impulse reward associated with the the completion of activity a, and $\mathcal{R}(\nu)$ is a rate reward of zero when the marking of $computer\_failed$ is 2, and one, otherwise. Then, if we define V_t as the instant-of-time variable at time t and $V_{t\to\infty}$ as the limit of this variable at $t \to \infty$, $E[V_t]$ is the reliability at time t.

The reliability of the system was then evaluated using *UltraSAN* transient solver, for 20 years mission time. The result of this evaluation is shown in Figure 7. The state space of the reduced base model consists of 10114 states. In addition, the reliability of 6 other design variations of the design described above are measured. Table 10 lists the 7 different designs, and the size of the reduced base model, and reliability of each design assuming 10 years mission time.

B Distributed Database Architecture

The second system considered is a distributed architecture for a database system, as shown in Figure 8. This example, which is a modified version of the examples given in [12] and [32], is intended to illustrate the use of reduced base model construction technique in evaluating database and file systems. The database consists of 6 disk clusters, 2 sets of disk controllers, and 2 processors. Each disk cluster consists of 4 disks. Data on each disk is replicated such that one third of the data is on each of the other three disks in the same

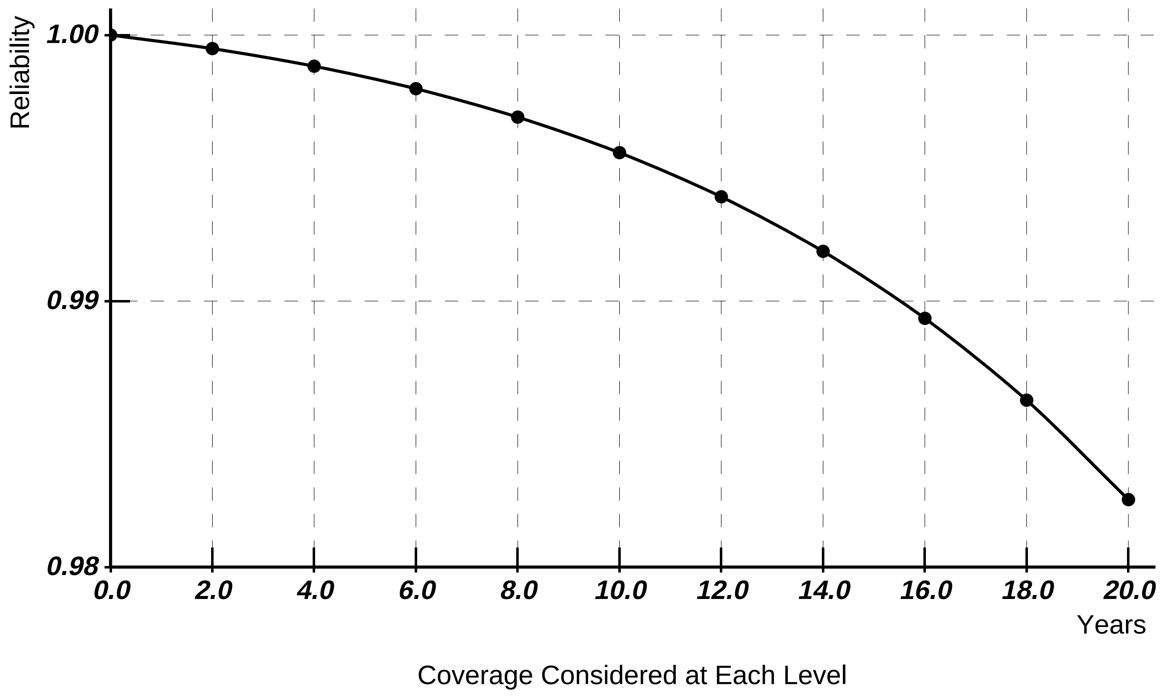

Figure 7: Reliability vs. Mission Time

Table 10: Reliability for Different System Designs

Design Description	**State Space Size**	**Reliability (10 Years Mission Time)**
100% coverage at all levels	4278	0.999539
Non-perfect coverage considered at all levels	10114	0.995579
Non-perfect coverage considered at all levels, no spare memory module	1335	0.987646
Non-perfect coverage considered at all levels, no spare CPU module	3299	0.973325
Non-perfect coverage considered at all levels, no spare I/O port	3299	0.985419
Non-perfect coverage considered at all levels, no spare memory module, CPU module, or I/O port	511	0.935152
100% coverage at all levels, no spare memory module, CPU module, I/O port, or RAM chips	6	0.702240

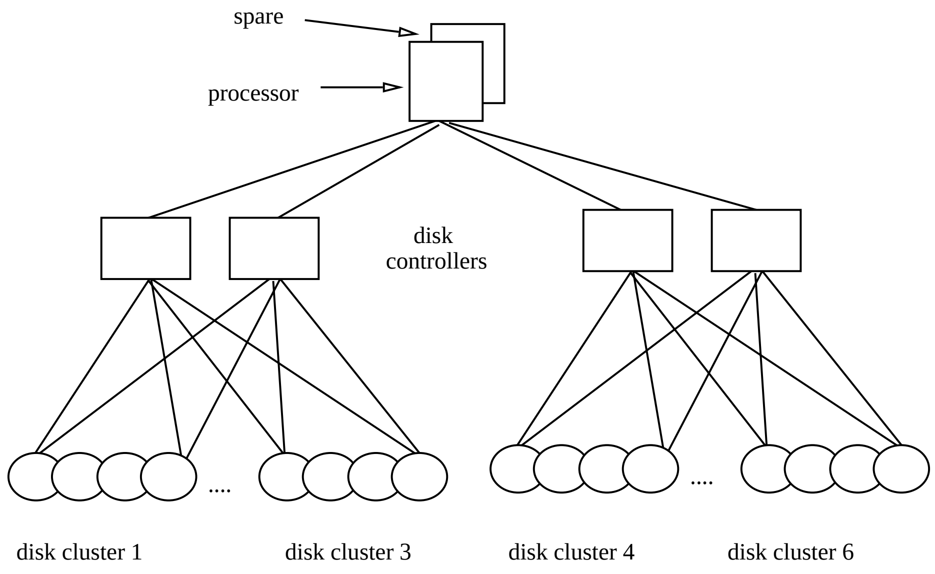

Figure 8: Database System

cluster. The disk clusters are placed into two groups, where each group contains 3 disk clusters. One set of controllers is connected to each group of disk clusters. Each processor can access the data on any disk cluster through the disk controllers.

The database system is considered to be operational if at least one processor can access the data in each disk cluster. Thus, the criteria for correct operation is that at least one processor, at least one disk controller in each set of controllers, and at least 3 disks in each disk cluster are operational. System repair is carried out by having one repair facility for each disk cluster, one repair facility for each set of controllers, and one repair facility for the two processors. Furthermore, we assume that components can continue to fail even if the system is in a failed state. Each repair facility repairs one component at a time. Repair on a given component starts as soon as its corresponding repair facility becomes free. If the system is in a failed state, it goes back into a working state when enough components have been repaired for the system to be in an operational state. For our numerical solution, we let the failure and repair rates for the components be, as in [32], the values listed in table 11.

For this system, both reliability and steady-state availability can be evaluated. We first consider the steady-state availability of the database. The composed model of the database system availability is shown in Figure 9. Replicate and Join operations are applied on three availability submodels, **disks**, **controllers**, and **processors**, to generate the composed

Table 11: Components Per Hour Failure and Repair Rates for the Database System

Component	**Failure Rate**	Repair Rate
Processor	1/2000	1
Disk controller	1/2000	1
Disk	1/6000	1

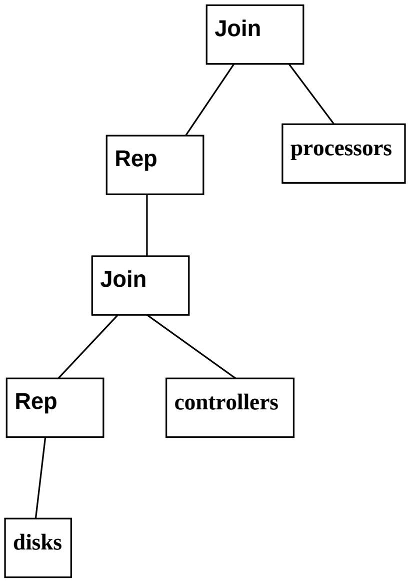

Figure 9: Composed Availability Model for Database System

model. The submodel **disks** represents the failure and repair of disks in a given disk cluster. The submodel **controllers** represents the failure and repair of disk controllers in one set of controllers. The submodel **processors** models the failure and repair of the two processors. The **disks** submodel is first replicated 3 times, then joined with one set of controllers. This composed submodel represents one-half of the database, controlled by a replicated disk controller. This submodel is then replicated 2 times before it is joined with the processors submodel to generate the composed model of the database system used to determine steady-state availability.

The SAN submodel of a single disk cluster is shown in Figure 10. This submodel contains three places, *disks_working, disks_failed*, and *system_fail.* Disk failure and repair is represented by the timed activities *disk_failure,* and *disk_repair*, respectively. The marking of *disks_working* indicates the current number of disks that are operational in a given disk

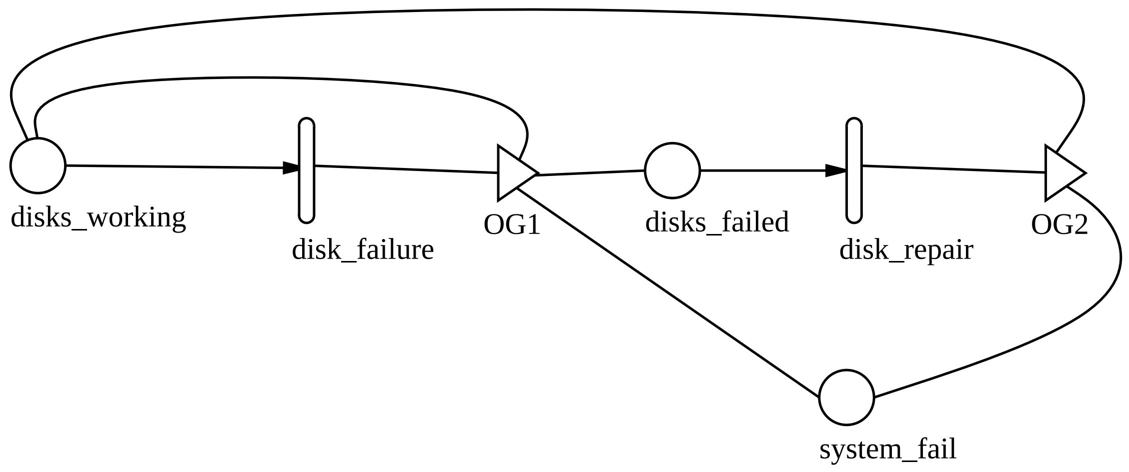

Figure 10: SAN Availability Submodel of a Disk Cluster

Table 12: Disks Availability Submodel Activity Time Distributions

Activity	**Distribution**
disk_failure	expon(1/6000.0 * MARK(disks_working))
disk_repair	expon(1.0)

cluster, initially 4. If a disk fails, i.e. the activity *disk_failure* completes, then the marking of *disks_working* is decremented by one and the marking of *disks_failed* is incremented by one. If the number of operational disks in a given disk cluster becomes less than three, the marking of *system_fail* is incremented by one.

Whenever a failed disk's repair is complete (i.e., activity *disk_repair* completes) the status of the corresponding disk cluster is evaluated to see whether it should become operational, if it was not. More specifically, if the disk cluster was in a failed state, e.g., *MARK(disks_working) == 2*, and now it is in a working state, i.e., *MARK(disks_working) == 3*, the marking of *system_fail* is decremented by one. The marking of *system_fail* can be as high as 9 due to the failure of the 6 disk clusters, the two sets of controllers, and the two processors. The system is considered operational only if the marking of *system_fail* is zero. The completion rates of the timed activities in each disk cluster submodel are listed in table 12. The input gate *predicates and functions* for this SAN submodel are also listed in table 13.

The SAN submodels for the disk controllers, the two processors, and the tables asso-

Table 13: Disks Availability Submodel Output Gate Functions

Gate	Function
$OG1$	$MARK(disks\_failed)++;$ $if\ (MARK(disks\_working)\ ==\ 2)$ $\quad MARK(system\_fail)++;$
$OG2$	$MARK(disks\_working)++;$ $if\ (MARK(disks\_working)\ ==\ 3)$ $\quad MARK(system\_fail)--;$

ciated with each submodel are shown in appendix A. The construction of these models is similar to that for the disk cluster and, hence, will not be discussed here. Using the complete composed model, the steady-state availability was determined by using *UltraSAN* to construct the reduced base model for the composed model, and solving it using available state-state Markov solution methods. The following rate reward variable is defined for the database system availability,

$$\mathcal{C}(a) = 0, \quad \forall\ \ activities\ a$$

$$\mathcal{R}(\nu) = \begin{cases} 1 & \text{if } \nu = \{(system\_fail, 0)\} \\ 0 & \text{otherwise,} \end{cases}$$

where $\mathcal{R}$ and $V_{t\to\infty}$ are as defined previously. The steady-state availability is $E[V_{t\to\infty}]$. For the parameters used the steady state availability was evaluated using the steady-state solver in *UltraSAN* and found to be 0.999997. The reduced base model size is 16695 states.

The reliability of the database system discussed above can also be determined. To do this, we use the same composed model structure as that used to determine steady-state availability, but must change the SAN submodels which represent the individual system components. Specifically, in determining the reliability of the system, it is not necessary to consider component repair. The SAN submodel used for the reliability of a disk cluster is shown in Figure 11. The tables associated with this submodel are given in table 14 and 15. As is readily apparent, this model is a simplification of that used to determine steady-state availability, in which repair times are not considered. The remaining SAN submodels used (for each set of controllers and the two processors), and the tables associated with them are also given in appendix A.

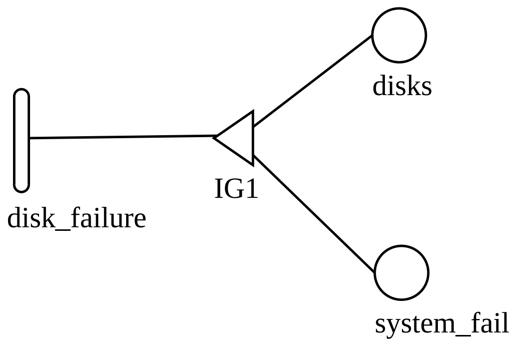

Figure 11: SAN Reliability Submodel of a Disk Cluster

Table 14: Disk Cluster Reliability Submodel Activity Time Distributions

Activity	**Distribution**
disk_failure	expon(1/6000.0 * MARK(disks))

Table 15: Disk Cluster Reliability Submodel Input Gate Predicates and Functions

Gate	**Enabling Predicate**	**Function**
$IG1$	$(MARK(disks) > 2)$ && $(MARK(system\_fail) == 0)$	$MARK(disks) - -;$ $if\ (\ MARK(disks) == 2)\ \{$ $MARK(system\_fail) = 1;$ $MARK(disks) = 0;$ $\}$

Table 16: Database System Reliability for Different Designs

Design Description	**State Space Size**	**Reliability (5 Weeks Mission Time)**
6 disk clusters, 2 controller sets, and 2 processors	268	0.425082
6 disk clusters + 1 spare disk per cluster, 2 controller sets, and 2 processors	1510	0.650810
6 disk clusters + 1 spare disk per cluster, 2 controller sets + 1 spare controller per set, and 2 processors	3075	0.769799

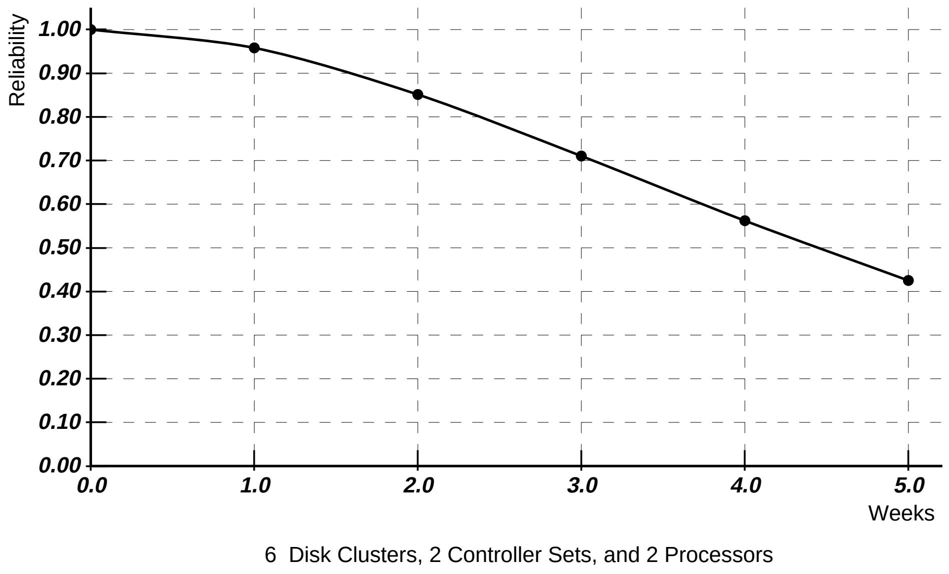

Figure 12: Database System Reliability vs. Mission Time

Using the same reward structure used for database system availability and reward variable $E[V_t]$ to define database system reliability, the result is plotted for a 5 week mission time in Figure 12. The state space of the reduced base model used to model reliability consists of 268 states. In addition, for comparison, the reliability of several other design variations are also evaluated and the results are tabulated in table 16.

C Multiprocessor-Multimemory System

The third system considered is a multiprocessor-multimemory system (MPMMS), shown in Figure 13 [33]. In this system there are 16 processors that share 16 memories. The processors are connected to the memories through a 2 stage omega network which is constructed

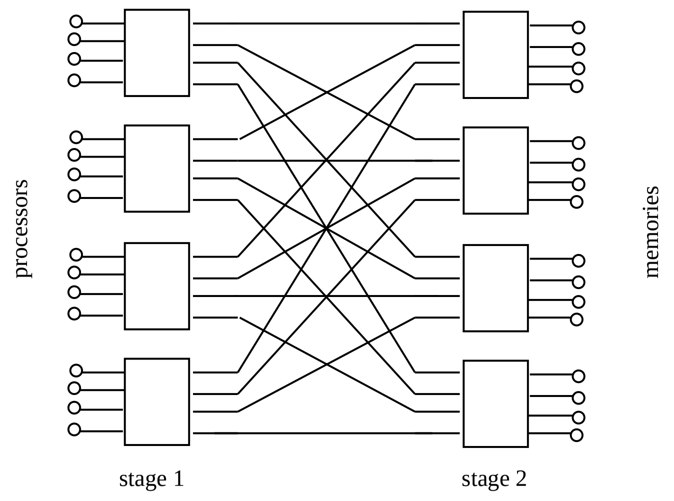

Figure 13: 16 Processors and 16 Memories Interconnected Using an Omega Network With Eight 4x4 Switching Elements

Table 17: Components Failure and Repair Rates for MPMMS

Component	**Failure Rate**
Processor	0.0000689
Memory	0.0002241
Switch	0.00001012

from eight 4x4 switching elements (SEs). Through the switching elements any processor can communicate with all memories. The system is considered operational as long as at least U processors can access at least V memories. To illustrate the evaluation of a system such as this, we consider the case where U = V = 4. The failure rate per hour for the components in this system are shown in table 17.

The composed model used to determine the reliability of the MPMMS is shown in Figure 14. Each of the SAN submodels in the composed model (i.e., **stage1** and **stage2**) is replicated 4 times. The replicated submodels are then joined together to generate the complete composed model of the MPMMS used to determine its reliability.

The SAN submodel of the first stage of switching elements (i.e., **stage1**) is shown in Figure 15. In this SAN, the failure of a switch and the failure of the processors connected to this switch are modeled. In particular, completion of timed activity *processor_failure* represents the failure of a processor. Similarly, when *SE_failure* completes, a switch fails and,

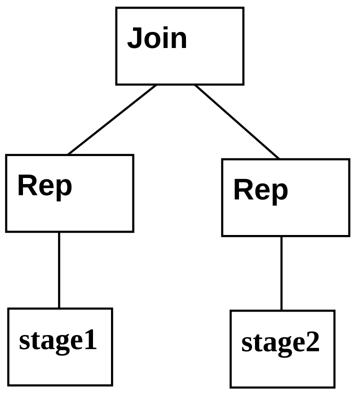

Figure 14: Composed Reliability Model of the MPMMS

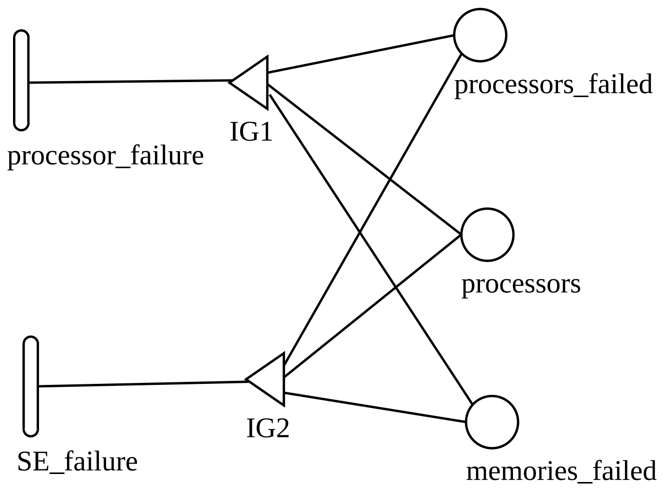

Figure 15: SAN Reliability Submodel of **stage1**

hence, all working processors that are connected to this switch lose the ability to communicate (see table 19). The system fails if the number of failed processors or failed memories is greater than 12 (i.e., there are at least four functioning processors and memories).

The SAN submodel **stage2**, shown in appendix A, is similar to the reliability submodel of stage 1. In the **stage2** SAN, the failure of a switch and the failure of the 4 memories connected to this switch are modeled. The tables associated with this submodel are also shown in appendix A.

The following reward variable can be used to define the MPMMS reliability,

$$\mathcal{C}(a) = 0, \quad \forall \ \textit{activities } a$$

Table 18: **stage1** Reliability Submodel Activity Time Distributions

Activity	**Distribution**
$SE\_failure$	expon(0.00001012)
$processor\_failure$	expon(0.0000689 * MARK(processors))

Table 19: **stage1** Reliability Submodel Input Gate Predicates and Functions

Gate	**Enabling Predicate**	**Function**
$IG1$	$(MARK(processors) > 0)$ && $(MARK(processors\_failed) < 13)$ &&$(MARK(memories\_failed) < 13)$	$MARK(processors) - -;$ $MARK(processors\_failed) + +;$
$IG2$	$(MARK(processors) > 0)$ && $(MARK(processors\_failed) < 13)$ && $(MARK(memories\_failed) < 13)$	$MARK(processors\_failed) = MARK(processors\_failed) + MARK(processors);$ $MARK(processors) = 0;$

$$\mathcal{R}(\nu) = \begin{cases} 0 & \text{if } \nu = \{(processors\_failed, i), (memory\_failed, j)\}, \quad \forall\, i,j \;\; 12 < i < 17 \text{ } or \text{ } 12 < j < 17 \\ 1 & \text{otherwise,} \end{cases}$$

where $E[V_t]$ is the reliability at time t.

The reliability of the MPMMS for 12000 hours mission time is plotted in Figure 16, for various mission times. The state space size for this model is 4851 states. In addition, the reliability of the MPMMS was also evaluated for different values of K=U=V, the results are shown in table 20.

Table 20: MPMMS Reliability for Different Values of K=U=V

K	**State Space Size**	Reliability (1000 Hours Mission Time)
4	4851	0.999856
8	3081	0.994883
12	480	0.747444
16	5	0.008489

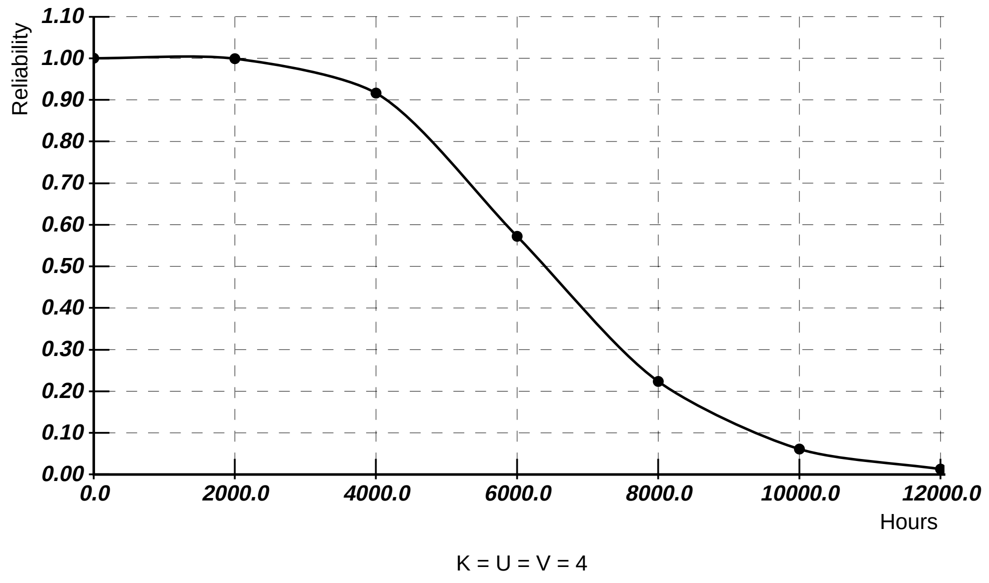

Figure 16: MPMMS Reliability vs. Mission Time

IV Conclusions

The purpose of this paper was to investigate the applicability of stochastic activity networks and reduced base model construction to modeling medium to large fault-tolerant parallel and distributed systems. In order to do this, we considered their applicability to three representative systems: a fault-tolerant parallel computing system, a distributed database architecture, and a multiprocessor-memory system. We showed that models of these systems can be easily constructed using stochastic activity networks, and that the processes that resulted from the reduced base model construction procedure were solvable. The results obtained were exact, to the numerical accuracy desired. Approximate techniques can solve larger problems than can be solved using these techniques, but the example systems considered illustrate that these methods indeed can be used to solved models of many realistic systems.

More specifically, with respect to the first example, we provided an exact solution for the reliability of the fault-tolerant multiprocessor system for the case of two computers at the highest level. The technique discussed in [13] was used to solve this example for up to ten computers at the highest level, but was approximate, with no bounds given on the approximation. Architectures similar to the second example, a distributed database system, have been investigated by other researchers using importance sampling and simulation [32] and a state truncation technique [12]. The state-truncation technique does not require that the failure rates on identical subsystems be the same, but as with the technique in [13] is approximate. Finally, the third example, a multiprocessor-multimemory system, has been considered by [33], who provided an exact solution, but built the state space directly at the state level. This works for small systems or those with a very regular structure, but is much too tedious for many systems.

In summary, stochastic activity networks and the reduced base model construction technique work well, and provide an exact solution, for many medium size, but realistic systems. They can be used when the system considered has some replicated components, as is the case with most fault-tolerant parallel and distributed systems. Models can be constructed at the network, rather than state, level and many design specifics, such as coverage, can be incorporated that can have a direct impact on system dependability.

REFERENCES

[1] A. M. Johnson, Jr. and M. Malek, "Survey of software tools for evaluating reliability, availability and serviceability," *ACM Computing Surveys*, vol. 20, pp. 227–269, Dec. 1988.

[2] A. Goyal, W. C. Carter, E. de Souza e Silva, S. S. Lavenberg and K. .S. Trivedi , "The system availability estimator," in *Proceedings of FTCS-16*, pp. 84–89, July, 1986.

[3] S. J. Bavuso, J. B. Dugan, K. S. Trivedi, E. M. Rothmann 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] G. Ciardo, J. Muppala and K. S. Trivedi, "SPNP: Stochastic Petri net package," in *Proceedings of the Third International Workshop on Petri-nets and Performance Models*, pp. 142–151, 1989.

[5] G. Chiola "A software package for the analysis of generalized stochastic Petri net models," in *Proceedings of the International Workshop on Timed Petri nets*, pp. 136–143, 1985.

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

[7] B. R. Haverkort and I. G. Niemegeers, "Performability modeling using dynamic queueing networks," in *Performance Evaluation Review*, vol. 17, p. 225, 1989.

[8] J. A. Carrasaco and J. Figueras , "METFAC: Design and implementation of a software tool for modeling and evaluation of complex fault-tolerant computing systems," in *Proceedings of FTCS-16*, pp. 424–429, 1986.

[9] S. Berson, E. de Souza e Silva and R. R. Muntz, "An object oriented methodology for the specification of Markov models," in *The First International Conference on the Numerical Solutions of Markov Chains*, pp. 2–29, 1990.

[10] R. A. Sahner and K. S. Trivedi, "Reliability modeling using SHARPE," *IEEE Transactions on Reliability*, vol. R-36, pp. 186–193, June 1987.

[11] A. L. White and D. L. Palumbo, "State reduction of semi-Markov reliability models," *1990 Proceedings annual Reliability and Maintainability Symposium*, pp. 280–285, 1990.

[12] R. R. Muntz, E. de Souza e Silva and A. Goyal, "Bounding availability of repairable computer systems,"*Performance Evaluation Review*, vol. 17, pp. 29–38, May 1989.

[13] D. Lee, J. Abraham, D. Rennels and G. Gilley, "A numerical technique for the evaluation of large, closed fault-tolerant systems," in *Dependable Computing for Critical Applications 2*, eds. J.F. Meyer and R.D. Schlichting, Springer-Verlag, Wien, pp. 95–114, 1992.

[14] J. C. Kemeny and J. L. Snell, *Finite Markov Chains*, Princeton: D. Van Nostrand Co., Inc., 1969.

[15] A. Zenie, "Colored stochastic Petri nets," in *Proc. International Workshop on Timed Petri Nets*, pp. 262–271, Torino, Italy, July 1985.

[16] C. Lin and D. C. Marinescu, "Stochastic high-level Petri nets and applications," *IEEE Trans. on Computers*, vol. C-37, no. 7, pp. 815–825, July 1988.

[17] C. Dutheillet and S. Haddad, "Regular stochastic Petri nets," *Proc. Tenth European Workshop on Application and Theory of Petri Nets*, Bonn, W. Germany, June 1989.

[18] G. Chiola and G. Franceschinis, "Colored GSPN models and automatic symmetry detection," *Proc. Third International Workshop of Petri Nets and Performance Models*, Kyoto, Japan, Dec. 1989.

[19] G. Chiola, C. Dutheillet, G. Franceschinis, and S. Haddad, "On well-formed coloured nets and their symbolic reachability graph", *Proc. Eleventh International Conference on Application and Theory of Petri Nets*, Paris, France, June 1990.

[20] W. H. Sanders and J. F. Meyer, "Reduced base model construction methods for stochastic activity networks," *IEEE Journal on Selected Areas in Communications*, vol. 9, pp. 25–36, Jan. 1991.

[21] T. F. Arnold, "The concept of coverage and its effect on the reliability model of a repairable system," *IEEE Transactions on Computers* , vol. C-22, pp. 251–254, Mar. 1973.

[22] J. B. Dugan and K. S. Trivedi, "Coverage modeling for dependability analysis of fault-tolerant systems," *IEEE Transactions on Computers* , vol. 38, pp. 775–787, June 1989.

[23] J. A. 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*, pp. 69–80. Sept. 1991.

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

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

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

[27] M. A. Marsan, G. Balbo, G. Chiola, and G. Conte,"Generalized stochastic Petri nets revisited: random switches and priorities," in *Proc. International Workshop on Petri nets and performance models*, Madison, Wisconsin, August 1987, pp. 44–53.

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

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

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

[31] W. H. Sanders and J. F. Meyer, "A unified approach for specifying measures of performance, dependability and performability," in *Dependable Computing for Critical Applications,* eds. A. Avizienis and J. C. Laprie, Springer-Verlag, Wien, pp. 216–237, 1991.

[32] V. F. Nicola, M. K. Nakayama, P. Heidelberger and A. Goyal "Fast simulation of dependability models with general failure and maintenance processes," *The Twentieth International Conference of Fault-Tolerant Computing*, pp. 491–498, June 1990.

[33] J. T. Blake, A. L. Reibman and K. S. Trivedi "Sensitivity analysis of reliability and performability measures for multiprocessor system," *Proceedings of the 1988 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pp. 177–186, May 1988.

V Appendix A

Table 21: **memory_module** Activity Time Distributions

Activity	**Distribution**
interface_chip_failure	expon(0.0008766 * MARK(interface_chips))
memory_chip_failure	expon(0.0008766 * MARK(memory_chips))

Table 22: **memory_module** Case Probabilities for Activities

Activity	**Case**	**Probability**
$interface\_chip\_failure$	1	0.95
	2	0.0475
	3	0.0025

Table 23: **memory_module** Case Probabilities for Activities

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 24: **memory_module** Input Gate Predicates and Functions

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

Table 25: **memory_module** Output Gate Functions

Gate	Function
$OG1$	$if\ (MARK(memory\_chips)\ >\ 39)$ $MARK(memory\_chips) - -;$
$OG2$	$MARK(memory\_chips)\ =\ 0;$ $MARK(interface\_chips)\ =\ 0;$ $MARK(memory\_failed) + +;$ $if\ (MARK(memory\_failed)\ >\ 1)$ $MARK(computer\_failed) + +;$
$OG3$	$MARK(memory\_chips)\ =\ 0;$ $MARK(interface\_chips)\ =\ 0;$ $if\ (\ (MARK(memory\_failed)\ ==\ 1)\ \&\&\ (MARK(computer\_failed)\ ==\ 0)\ )\ \{$ $MARK(memory\_failed)\ =\ 2;$ $MARK(computer\_failed)\ =\ 2;$ $\}$ $else\ \{$ $MARK(memory\_failed)\ =\ 2;$ $MARK(computer\_failed) + +;$ $\}$
$OG4$	$MARK(memory\_chips)\ =\ 0;$ $MARK(interface\_chips)\ =\ 0;$ $MARK(memory\_failed)\ =\ 2;$ $MARK(computer\_failed)\ =\ 2;$
$OG5$	$MARK(interface\_chips)\ =\ 0;$ $MARK(memory\_failed) + +;$ $if\ (MARK(memory\_failed)\ >\ 1)$ $MARK(computer\_failed) + +;$
$OG6$	$MARK(interface\_chips)\ =\ 0;$ $if\ (\ (MARK(memory\_failed)\ ==\ 1)\ \&\&\ (MARK(computer\_failed)\ ==\ 0)\ )\ \{$ $MARK(memory\_failed)\ =\ 2;$ $MARK(computer\_failed)\ =\ 2;$ $\}$ $else\ \{$ $MARK(memory\_failed)\ =\ 2;$ $MARK(computer\_failed) + +;$ $\}$
$OG7$	$MARK(interface\_chips)\ =\ 0;$ $MARK(memory\_failed)\ =\ 2;$ $MARK(computer\_failed)\ =\ 2;$

Table 26: **errorhandlers** Activity Time Distributions

Activity	Distribution
$errorhandling\_chip\_failure$	expon(0.0008766 * MARK(errorhandlers))

Table 27: **errorhandlers** Case Probabilities for Activities

Activity	**Case**	**Probability**
$errorhandling\_chip\_failure$	1	0.95
	2	0.05

Table 28: **errorhandlers** Input Gate Predicates and Functions

Gate	**Enabling Predicate**	**Function**
$IG1$	$(MARK(errorhandlers)\ ==\ 2)$ && $(MARK(memory\_failed)\ <\ 2)$ && $(MARK(computer\_failed)\ <\ 2)$	$MARK(errorhandlers)\ =\ 0;$

Table 29: **errorhandlers** Output Gate Functions

Gate	**Function**
$OG1$	$MARK(cpus)\ =\ 0;$ $MARK(ioports)\ =\ 0;$ $MARK(memory\_failed)\ =\ 2;$ $MARK(computer\_failed)++;$
$OG2$	$MARK(cpus)\ =\ 0;$ $MARK(ioports)\ =\ 0;$ $MARK(memory\_failed)\ =\ 2;$ $MARK(computer\_failed)\ =\ 2;;$

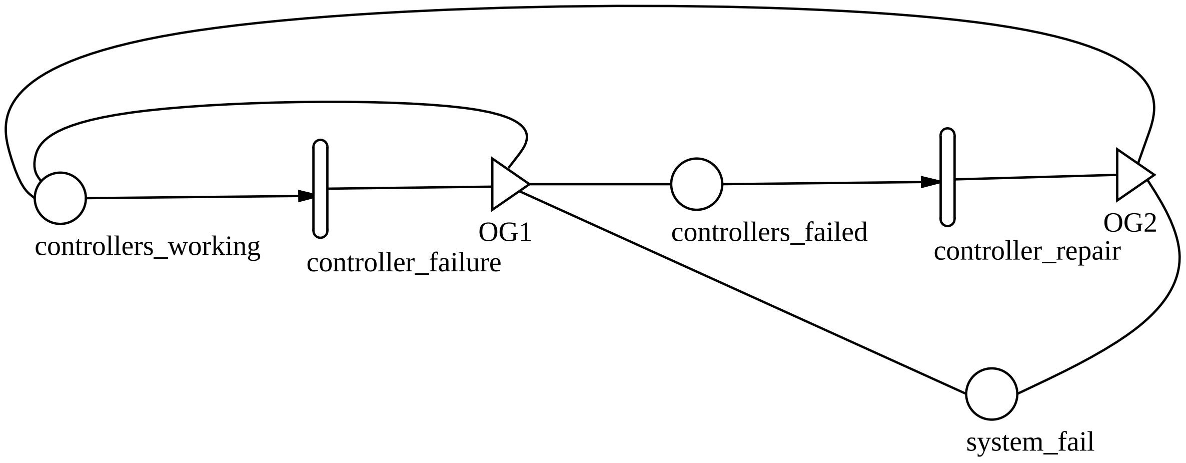

Figure 17: SAN Availability Submodel of a Set of Controllers

Table 30: A Set of Controllers Availability Submodel Activity Time Distributions

Activity	**Distribution**
controller_failure	expon(1/2000.0 * MARK(controllers_working))
controller_repaire	expon(1.0)

Table 31: A set of Controllers Availability Submodel Output Gate Functions

Gate	**Function**
$OG1$	$MARK(controllers\_failed)++;$ $if(MARK(controllers\_working) == 0)$ $MARK(system\_fail)++;$
$OG2$	$MARK(controllers\_working)++;$ $if(MARK(controllers\_working) == 1)$ $MARK(system\_fail)--;$

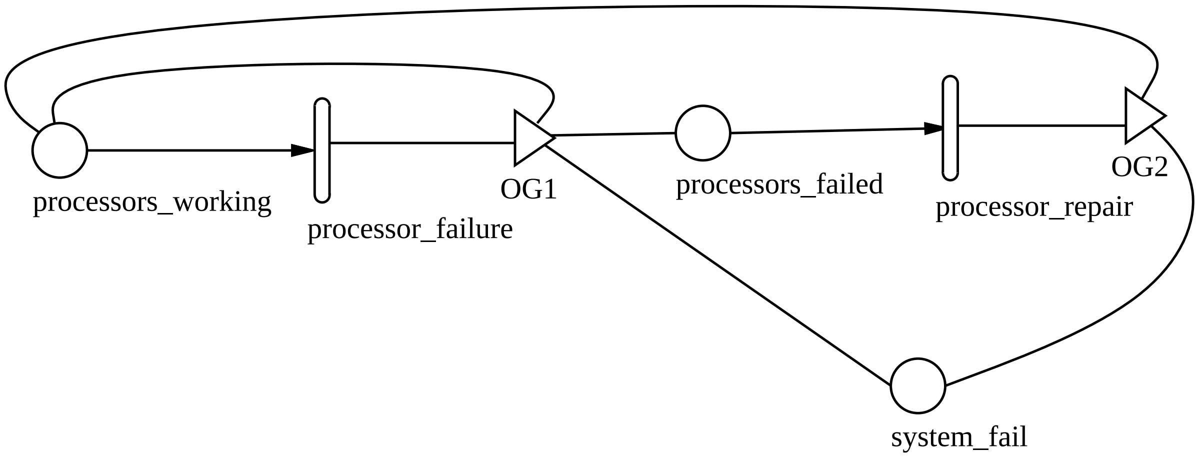

Figure 18: SAN Availability Submodel of the Two Processors

Table 32: **processors** Availability Submodel Activity Time Distributions

Activity	**Distribution**
processor_failure	expon(1/2000.0 * MARK(processors_working))
processor_repair	expon(1.0)

Table 33: **processors** Availability Submodel Output Gate Functions

Gate	**Function**
$OG1$	$MARK(processors\_failed)++;$ $if(MARK(processors\_working) == 0)$ $MARK(system\_fail)++;$
$OG2$	$MARK(processors\_working)++;$ $if(MARK(processors\_working) == 1)$ $MARK(system\_fail)--;$

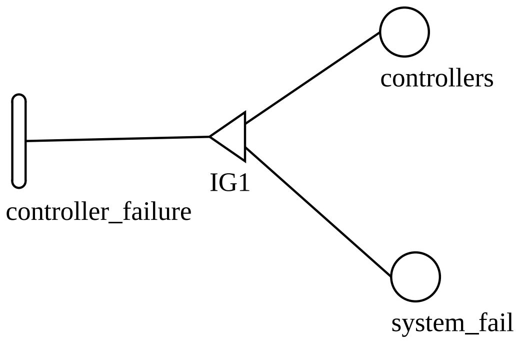

Figure 19: SAN Reliability Submodel of a Set of Controllers

Table 34: Set of Controllers Reliability Submodel Activity Time Distributions

Activity	**Distribution**
controller_failure	expon(1/2000.0 * MARK(controllers))

Table 35: Set of Controllers Reliability Submodel Input Gate Predicates and Functions

Gate	**Enabling Predicate**	**Function**
$IG1$	$(MARK(controllers)\ >\ 0)$ && $(MARK(system\_fail)\ ==\ 0)$	$MARK(controllers) - -;$ $if\ (MARK(controllers)\ ==\ 0)$ $MARK(system\_fail)\ =\ 1\ ;$

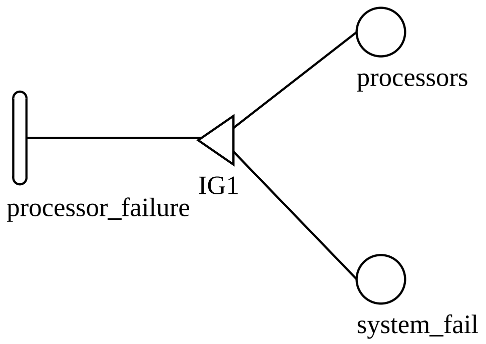

Figure 20: SAN Reliability Submodel of the Two Processors

Table 36: **processors** Reliability Submodel Activity Time Distributions

Activity	**Distribution**
processor_failure	expon(1/2000.0 * MARK(processors))

Table 37: **processors** Reliability Submodel Input Gate Predicates and Functions

Gate	Enabling Predicate	Function
$IG1$	$(MARK(processors)\ >\ 0)$ && $(MARK(system\_fail)\ ==\ 0)$	$MARK(processors)--;$ $if(MARK(processors)\ ==\ 0)$ $MARK(system\_fail)\ =\ 1;$

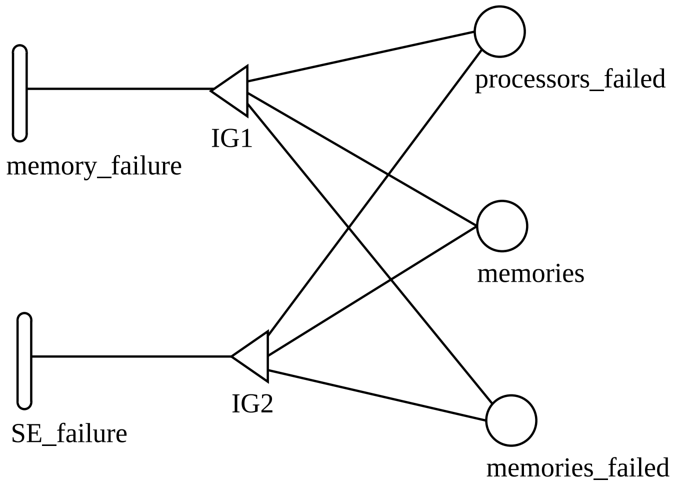

Figure 21: SAN Reliability Submodel of **stage2**

Table 38: **Stage2** Reliability Submodel Activity Time Distributions

Activity	Distribution
$SE\_failure$	expon(0.00001012)
$memory\_failure$	expon(0.0002241 * MARK(memories))

Table 39: **stage2** Reliability Submodel Input Gate Predicates and Functions

Gate	Enabling Predicate	Function
$IG1$	$(MARK(memories) >\ 0)$ && $(MARK(processors\_failed)\ <\ 13)$ && $(MARK(memories\_failed)\ <\ 13)$	$MARK(memories)--;$ $MARK(memories\_failed)++;$
$IG2$	$(MARK(memories)\ >\ 0)$ && $(MARK(processors\_failed)\ <\ 13)$ && $(MARK(memories\_failed)\ <\ 13)$	$MARK(memories\_failed)\ =$ $MARK(memories\_failed)\ +$ $MARK(memories);$ $MARK(memories)\ =\ 0;$