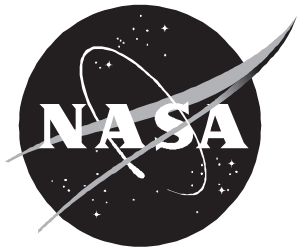
NASA Reference Publication 1348



Techniques for Modeling the Reliability of Fault-Tolerant Systems With the Markov State-Space Approach

Ricky W. Butler and Sally C. Johnson
Langley Research Center • Hampton, Virginia

National Aeronautics and Space Administration
Langley Research Center • Hampton, Virginia 23681-0001

September 1995

Available electronically at the following URL address: http://techreports.larc.nasa.gov/ltrs/ltrs.html

Printed copies available from the following:

NASA Center for AeroSpace Information
800 Elkridge Landing Road
Linthicum Heights, MD 21090-2934
(301) 621-0390

National Technical Information Service (NTIS)
5285 Port Royal Road
Springfield, VA 22161-2171
(703) 487-4650

Contents

Abstract

This paper presents a step-by-step tutorial of the methods and the tools that were used for the reliability analysis of fault-tolerant systems. The approach of this paper is the Markov (or semi-Markov) state-space method. The paper is intended for design engineers with a basic understanding of computer architecture and fault tolerance, but little knowledge of reliability modeling. The representation of architectural features in mathematical models is emphasized. This paper does not present details of the mathematical solution of complex reliability models. Instead, it describes the use of several recently developed computer programs—SURE, ASSIST, STEM, and PAWS—that automate the generation and the solution of these models.

1. Introduction

The rapid growth of digital electronics technology has led to the proliferation of sophisticated computer systems capable of achieving very high reliability requirements. Reliability requirements for computer systems used in military aircraft, for example, are typically in the range of $1 - 10^{-7}$ per mission, and reliability requirements of $1 - 10^{-9}$ for a 10-hr flight are often expressed for flight-crucial avionics systems. To achieve such optimistic reliability goals, computer systems have been designed to recognize and tolerate their own faults. Although capable of tolerating certain faults, these systems are still susceptible to failure. Thus, the reliability of these systems must be evaluated to ensure that requirements are met.

The reliability analysis of a fault-tolerant computer system is a complex problem. Lifetime tests are typically used to determine the reliability or "lifetime" of a diversity of products such as light bulbs, batteries, and electronic devices. The lifetime test methodology is clearly impractical, though, for computer systems with reliability goals in the order of $1 - 10^{-7}$ or higher; hence, an alternate approach is necessary. The approach generally taken to investigate the reliability of a highly reliable system is

1. Develop a mathematical reliability model of the system
2. Measure or estimate the parameters of the model
3. Compute system reliability based upon the model and the specified parameters

The estimated system reliability is consequently strongly dependent on the model itself. Because the behavior of a fault-tolerant, highly reliable system is complex, formulating models that accurately represent that behavior can be a difficult task. Mathematical models of fault-tolerant systems must capture the processes that lead to system failure and the system capabilities that enable operation in the presence of failing components. Current manufacturing techniques cannot produce circuitry that meets ultrahigh reliability requirements. Therefore, highly reliable systems use redundancy techniques, such as parallel redundant units or dissimilar algorithms for computing the same function, to achieve fault tolerance. Reconfiguration, the process of removing faulty components and either replacing them with spares or degrading to an alternate configuration, is another method often utilized to increase reliability without the overhead of more redundancy.

Fortunately, most of the detailed instruction-level activities of a system do not directly affect system reliability. Only the macroscopic fault-related events must be included in the reliability model. Furthermore, experimentally testing the correctness of the model would require at least as much experimentation as is required for life testing. Consequently, the best approach is to carefully develop the reliability model and subject it to scrutiny by a team of experts. The process of reliability modeling is thus not an

exact science, and at best, should be called an art. The goal of this paper is to look into this craft of reliability modeling.

The paper is structured in a tutorial style rather than as a catalog of reliability models. Consequently, elementary concepts are introduced first and are followed by increasingly more complex concepts. Thus, the paper begins with an overview of essential aspects of Markov state-space models. Next, the fundamental techniques that were used for modeling the nonreconfigurable systems are developed. Then, the basic techniques that were used in modeling reconfigurable systems are explored. Before examining more complicated models, the computer program SURE (Semi-Markov Unreliability Range Evaluator), which can be used to solve the reliability models numerically, is introduced (ref. 1). Next, two basic reconfigurations—degradation and sparing—are examined in more detail with the help of the SURE input language.

At this point, the paper introduces a new language, ASSIST, for describing reliability models. This language is necessary because the models presented in the later sections are very large and complex. The expressiveness of the ASSIST language allows complex models to be defined in a succinct manner. Next, complex systems consisting of multiple triads that use various forms of reconfiguration are investigated. Then the techniques that were used to model transient and intermittent faults are presented. The next section explores the techniques that were used to model the components of control system architectures, which include sensors, buses, and actuators. Finally, some specialized topics such as sequence dependencies, phased missions, and nonconstant failure rate models are presented.

2. Introduction to Markov Modeling

Traditionally, the reliability analysis of a complex system has been accomplished with combinatorial mathematics. The standard fault-tree method of reliability analysis is based on such mathematics (ref. 2). Unfortunately, the fault-tree approach is incapable of analyzing systems in which reconfiguration is possible. (Work that augments fault-tree notation for the purpose of generating Markov models is not included in this statement.) Basically, a fault tree can be used to model a system with

1. Only permanent faults (no transient or intermittent faults)
2. No reconfiguration
3. No time or sequence failure dependencies
4. No state-dependent behavior (e.g., state-dependent failure rates)

Because fault trees are easier to solve than Markov models, fault trees should be used wherever these fundamental assumptions are not violated. (For more information see ref. 3.)

In reconfigurable systems, the critical factor often becomes the effectiveness of the dynamic reconfiguration process. It is necessary to model such systems by using the more powerful Markov modeling technique. A Markov process is a stochastic process whose behavior depends only upon the current state of the system. The particular sequence of steps by which the system entered the current state is irrelevant to its future behavior. Markov state-space models have four main categories:

1. Discrete space and discrete time
2. Discrete space and continuous time
3. Continuous space and discrete time
4. Continuous space and continuous time

The first two categories involve a discrete space; that is, the states of the system can be numbered with an integer. In the last two categories, the states of the system must be numbered with a real

number. In the first and the third categories, the system changes by discrete time steps and in the second and the fourth categories, the system transitions occur over continuous time.

The second category is the one most useful for modeling fault-tolerant systems and will always be used in this paper. Only models that contain a finite number of states will be used. However, the transition time between the states is not discrete and can take on any real value. For a more detailed mathematical description of Markov analysis, see reference 4.

The first step in modeling a system with a discrete-space and continuous-time Markov model is to represent the state of the system with a vector of attributes that change over time. These attributes are typically system characteristics such as the number of working processors, the number of spare units, or the number of faulty units that have not been removed. The more attributes included in the model, the more complex the model will be. Thus, the smallest set of attributes that can accurately describe the fault-related behavior of the system is typically chosen. The next step in the modeling process is to characterize the transition time from one state to another. Because this transition time is rarely deterministic, the transition times are described with a probability distribution.

Certain states in the system represent system failure, while others represent fault-free behavior or correct operation in the presence of faults. The model chosen for the system must represent system failure properly. Defining exactly what constitutes system failure is difficult because system failure is often an extremely complex function of external events, software state, and hardware state. The modeler is forced to choose between conservative or nonconservative assumptions about system failure. If the reliability of the system must be higher than a specific value, then conservative assumptions are made. For example, in a triple modular redundant (TMR) system of computers, the presence of two faulty computers is considered to be system failure. This assumption is conservative because the two faults may not actually corrupt data in a manner that would defeat the voter. This assumption simplifies the model because the probabilities of collusion of the faulty pair does not have to be modeled. In this paper, the conservative approach will be used exclusively.

It is important that all transitions in the reliability model be measurable. This measurability often is the primary consideration when developing a model for a system. Although a particular model may elegantly describe the behavior of the system, if it depends upon unmeasurable parameters, then it is useless.

Typically, the transitions of a fault-tolerant system model fall into two categories: slow failure transitions and fast recovery transitions. If the states of the model are defined properly, then the slow failure transitions can be obtained from field data and/or MIL-STD 217F calculations (ref. 5). The fast recovery transition corresponds to system responses to fault arrivals and can be measured experimentally with fault injection. The primary problem is to properly model the system so that the determination of these transitions is facilitated. If the model is too coarse, the transitions become experimentally unobservable. If the model is too detailed, the number of transitions that must be measured can be exorbitant.

This paper explores the methods and the assumptions that were used in the development of reliability models for fault-tolerant computer systems.

3. Modeling Nonreconfigurable Systems

The simplest systems to model are nonreconfigurable systems. This section introduces the basic elements of reliability modeling by describing how to model simple nonreconfigurable systems that range from a single simplex computer through a majority-voting *N*-modular-redundant (NMR) system.

3.1. Simplex Computer

The first example is a system consisting of a single computer. First, let T be a random variable representing the time to failure of the computer. Next, define a distribution for T, say $F(t)$. Typically,

electronic components, and consequently computers, are assumed to fail according to the exponential distribution

$$F(t) = \text{Prob}[T < t] = 1 - e^{-\lambda t}$$

The parameter λ completely defines this distribution. An important concept in reliability modeling is the failure rate (or hazard rate), $h(t)$ defined as follows:

$$h(t) = F'(t) / [1 - F(t)]$$

For the exponential distribution, the hazard rate $h(t) = \lambda$. The exponential distribution is the only distribution with a constant hazard rate. The Markov model representing this system is given in figure 1.

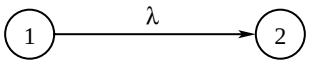


Figure 1. Model of simplex computer.

In this Markov model, state (1) represents the operational state in which the simplex computer is working, state (2) represents the system failure state in which the simplex computer has failed, and the transition from state (1) to state (2) represents the occurrence of the failure of the simplex computer. The transitions of a Markov model are exponential, and thus, can be labeled by the constant hazard rate.

For reliability modeling purposes, electronic components are generally assumed to fail according to the exponential distribution. Some immature devices may exhibit a somewhat higher failure rate due to insufficient testing before product delivery; however, mature devices have been shown experimentally to fail according to the exponential distribution (ref. 6). The MIL-STD 217F handbook offers a more complete discussion on the problem of estimating the reliability of electronic components. Once the reliability of each component (e.g., a chip) in a computer is known, the failure rate of the computer is simply the sum of the failure rates of the individual components. For example suppose $\lambda_1, \lambda_2, ..., \lambda_n$ represent the failure rates of the components in the computer. Letting T be a random variable representing the time of failure of the computer and T_i represent the time the ith component fails, the distribution of failure for the computer $F_c(t)$ is determined as follows:

$$\begin{aligned} F_c(t) &= \text{Prob}[T < t] \\ &= \text{Prob}\left[\min\{T_1, T_2, ..., T_n\} < t\right] \\ &= 1 - \text{Prob}\ [T_1 > t, T_2 > t, ..., T_n > t] \end{aligned}$$

Assuming that the components fail independently, then

$$\begin{aligned} F_c(t) &= 1 - \prod_{i=1}^{n} \text{Prob}[T_i > t] \\ &= 1 - \prod_{i=1}^{n} \exp(-\lambda_i t) \\ &= 1 - \exp + \left(-\sum_{i=1}^{n} \lambda_i t\right) \end{aligned}$$

which is an exponential distribution with failure rate $\sum_{i=1}^{n} \lambda_i$.

This technique does not work for parallel redundant systems. The time of failure of a redundant system is not merely the time that the first component fails. Such systems will be examined in the following sections.

3.2. Static Redundancy

The triple modular redundant (TMR) system is one of the simplest fault-tolerant computer architectures. The system consists of three computers; all performing exactly the same computations on exactly the same inputs. The computers are assumed to be physically isolated so that a failed computer cannot affect another working computer. Mathematically, therefore, the computers are assumed to fail independently. It is further assumed that the outputs are voted prior to being used by the external system (not included in this model), and thus, a single failure does not propagate its erroneous value to the external world. Thus, system failure does not occur until two computers fail. The model of figure 2 describes such a system.

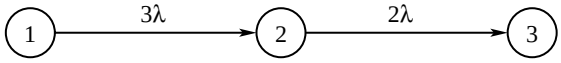


Figure 2. Model of TMR system.

State (1) represents the initial condition of three working computers. The transition from state (1) to state (2) is labeled 3λ to represent the rate at which any one of the three computers fails. Because all the computers are identical, the failure rate λ is the same for each computer.

The system is in state (2) when one processor has failed. The transition from state (2) to state (3) has rate 2λ because only two working computers can fail. State (3) represents system failure because a majority of the computers in the system have failed. The processor failure rate is $\lambda = 10^{-4}$/hr.

In figure 3, the probability of system failure as a function of mission time is shown.

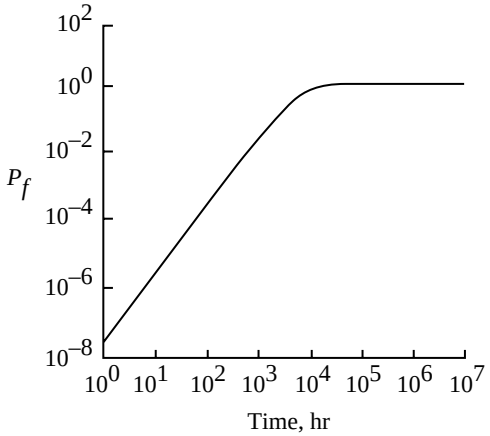


Figure 3. Failure probability as function of mission time.

It can be seen that high reliability is strongly dependent on a short mission time.

The system was implicitly assumed to start with no failed components (Probability in state (1) at time 0 = 1). This probability is equivalent to assuming perfect maintenance between missions. The probability of system failure as a function of the processor failure rate λ is plotted in figure 4.

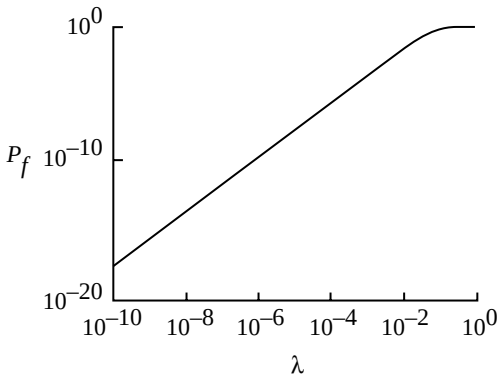


Figure 4. Failure probability as function of λ.

This probability was calculated for a mission time of 10 hr. Throughout this paper, unless otherwise stated, a mission time of 10 hr will be used.

The model shown in figure 2 uses the technique of state "aggregation" to reduce the number of states. The model in figure 5 also shows a TMR system, but without state aggregation.

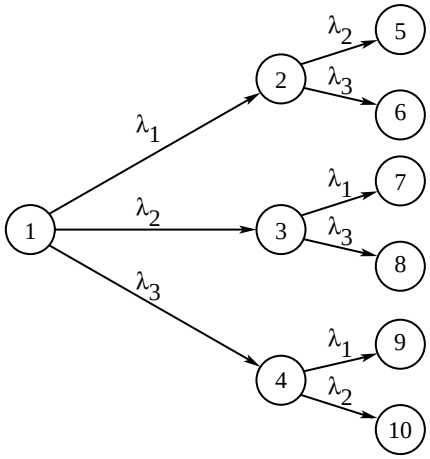


Figure 5. Nonaggregated model of TMR system.

This model does not take advantage of the inherent symmetry of the system. Each component is given a separate failure transition. Thus, three transition rates λ_1, λ_2, and λ_3 are used. In a TMR system, these three rates are all equal because the redundant channels are identical. A comparison of the model in figure 2 with the model in figure 5 shows the significant reductions in the number of states that can be obtained by use of aggregation. Throughout the rest of the report, aggregated models will be used unless an asymmetry exists in the system that prevents this usage.

3.3. Analytic Solution to TMR Model

In this section, the basic technique for solving a Markov model analytically is presented. A detailed understanding of this section is not necessary to understand the rest of this paper. This section may be omitted on the first reading.

The solution of a Markov model is conceptually simple, although the details can be cumbersome. An *n*-state Markov model leads to a system of *n*-coupled differential equations. These equations can

most simply be represented with vector notation. Let $P(t)$ be a vector that gives the probability of being in each state at time *t*. For the three-state Markov model in figure 2

$$P(t) = [p_1(t), p_2(t), p_3(t)]$$

The system of differential equations is given by

$$P'(t) = P(t)\,\mathbf{A}$$

where

$$\mathbf{A} = \begin{bmatrix} -3\lambda & 3\lambda & 0 \\ 0 & -2\lambda & 2\lambda \\ 0 & 0 & 0 \end{bmatrix}$$

In nonmatrix form

$$P_1'(t) = -3\lambda P_1(t)$$

$$P_2'(t) = 3\lambda P_1(t) - 2\lambda P_2(t)$$

$$P_3'(t) = 2\lambda P_2(t)$$

The matrix **A** is easily constructed by thinking of the Markov model in terms of flow in and flow out. One begins with the off-diagonal components. Because there is a transition from state (1) to state (2), the entry at a_{12} is nonzero. The value of a_{12} is the transition rate 3λ. The transition from state (2) to state (3) leads to the only other nonzero (nondiagonal) entry, a_{23}. Its value is 2λ. The diagonal entries are obtained by summing all nondiagonal entries on the same row and negating it. The solution can be written as

$$P(t) = P(0)\,e^{\mathbf{A}t}$$

The solution requires knowledge of the initial state probabilities $P(0)$. If the system begins in a fault-free state, this is given by

$$P(0) = [1, 0, 0]$$

If the model is changed as shown in figure 6, the matrix **A** becomes

$$\mathbf{A} = \begin{bmatrix} -3\lambda & 3\lambda & 0 \\ \alpha & -2\lambda - \alpha & 2\lambda \\ 0 & 0 & 0 \end{bmatrix}$$

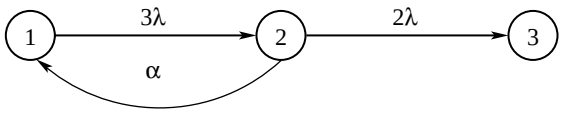


Figure 6. Altered model.

The three equations become

$$P_1'(t) = -3\lambda P_1(t) + \alpha P_2(t)$$

$$P_2'(t) = 3\lambda P_1(t) - (2\lambda + \alpha) P_2(t)$$

$$P_3'(t) = 2\lambda P_2(t)$$

The flow-in and flow-out relationship is clearly seen in these equations. The total flow out of state (1), $P_1'(t)$, is given by $-3\lambda\ P_1(t) + \alpha\ P_2(t)$. There is $3\lambda P_1(t)$ out of state (1) and $\alpha P_2(t)$ into state (1). Thus, the signs of the terms are – and +, respectively.

For more detailed information about the solution of Markov models, see reference 7.

3.4. N-Modular-Redundant System

The assumptions of an *N*-modular-redundant system are the same as a TMR system. The voter used in such a system is usually a majority voter. As long as a majority of processors have not failed, the system is still operational. The following model shown in figure 7 describes a seven-processor system with seven-way voting. The probability of system failure as a function of mission time is given in figure 8. Figure 9 shows the unreliability of an NMR system as a function of *N*.

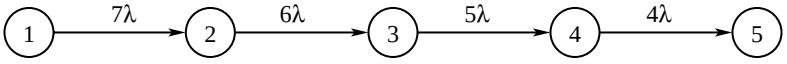


Figure 7. Model of 7MR system.

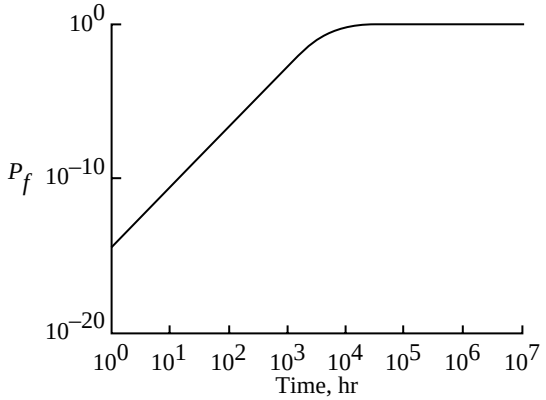


Figure 8. The 7MR system unreliability as function of mission time.

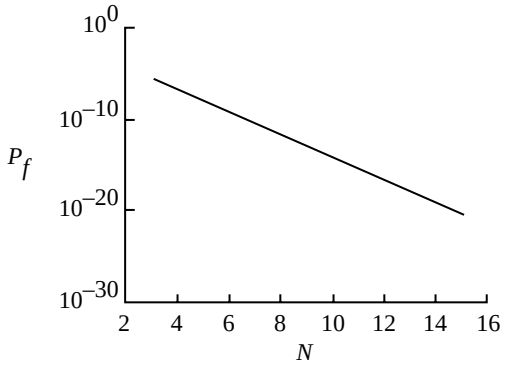


Figure 9. The NMR system unreliability as function *N*.

Theoretically, the probability of system failure approaches zero as *N* approaches infinity. Of course, this model ignores the practical problem of building an arbitrarily large *N*-way voter. If implemented in hardware, the additional hardware would significantly increase the processor failure rate λ. If implemented in software, the CPU overhead could be enormous, which would seriously increase the likelihood of a critical task missing a hard deadline (ref. 8).

4. Modeling Reconfigurable Systems

Fault-tolerant systems are often designed by using a strategy of reconfiguration. Reconfiguration strategies come in many varieties, but always involve the logical or the physical removal of a faulty

component. The techniques that are used to identify the faulty component and the methods that are used to repair the system vary greatly and can lead to complex reliability models. Two basic reconfiguration strategies occur—degradation and replacement with spares. The degradation method involves the permanent removal of a faulty component without replacement. The reconfigured system continues with a reduced set of components. The replacement with spares method involves both the removal of faulty components and their replacement with a spare. In this section, these concepts will be introduced briefly and explored in greater detail in later sections.

4.1. Degradable Triad

The simplest architecture based upon majority voting is the triplex system. To increase reliability, triplex systems have been designed that reconfigure by degradation. The model shown in figure 10 describes the behavior of a simple degradable triplex system.

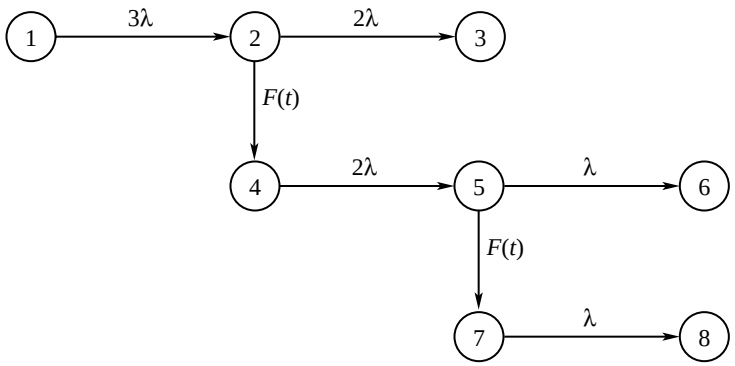


Figure 10. Model of degradable triplex.

The degradable triplex system begins in state (1) with all three processors operational. The transition from state (1) to state (2) represents the failure of any of the three processors. Because the processors are identical, the failure of each processor is not represented with a separate state. At state (2), the system has one failed processor. The system analyzes the errors from the voter and diagnoses the problem. The transition from state (2) to state (4) represents the removal (reconfiguration) of the faulty processor. Reconfiguration transitions are labeled with a distribution function ($F(t)$) rather than a rate. The reason for this labeling is that experimental measurement of the reconfiguration process has revealed that the distribution of recovery time is not exponential (ref. 6). Consequently, the transition cannot be described by a constant rate. This label is interpreted as the probability that the transition time from state (2) to state (4) is less than t is $F(t)$. The presence of a nonexponential transition generalizes the mathematical model to the class of semi-Markov models. Such models are far more difficult to solve than pure Markov models. In sections 5 and 8, several computer programs will be discussed that can be used to solve Markov and semi-Markov models.

At state (4), the system is operational with two good processors. The recovery transition from state (2) to state (4) occurs as long as a second processor does not fail before the diagnosis is complete. Otherwise, the voter could not distinguish the good results from the bad. Thus, a second transition exists from state (2) to state (3), which represents the coincident failure of a second processor. The rate of this transition is 2λ, because either of the remaining two processors could fail. State (3) is a death state (an absorbing state) that represents failure of the system due to near-coincident failure. Of course, this is a conservative assumption. Although two out of the three processors have failed, the failed processors may not produce errors at the same time nor in the same place in memory. In this case, the voting mechanism may effectively mask both faults and the reliability of the system would be better than the model predicts.

Perhaps a less conservative model could be developed, but this development would require the estimation of the probability that two faults would defeat the voter. This probability would likely depend upon the particular software workload on the system and many other design factors. Consequently, it would be difficult to obtain. Also, the reliability analysis would not be independent of the workload. If this probability is underestimated, the model would no longer be conservative. The reliability analyst is often faced with many trade-offs similar to these. For life-critical systems, the trade-off should always be made in the conservative direction.

At state (4), the system is operational with two good processors and no faulty processors in the active configuration. Either one of these processors may fail and take the system to state (5). At state (5), once again, a race occurs between the reconfiguration process that ends in state (7) and the failure of a second processor that ends in state (6). The recovery distribution from state (5) could easily be different from the recovery distribution from state (2) to state (4). However, for simplicity it is assumed to be the same. State (6) is thus another death state and state (7) is the operational state where one good processor remains. The transition from state (7) to state (8) represents the failure of the last processor. At state (8) no good processors remains, and the probability of reaching this death state is often referred to as failure by exhaustion of parts.

To solve the model shown in figure 10, $F(t)$ must be known. Suppose $F(t) = 1 - e^{-\delta t}$. Then, given a value for δ of 10^4/hr and a mission time of 10 hr, the model can be solved as a function of λ as shown in figure 11. Fixing λ at 10^{-4}/hr and T at 10 hr, the model can be solved as a function of δ as shown in figure 12.

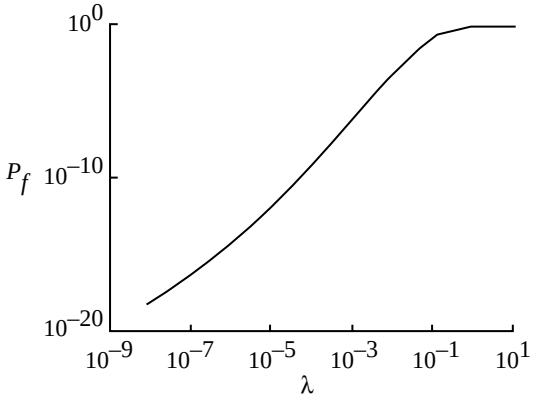


Figure 11. Degradable triplex as function of λ.

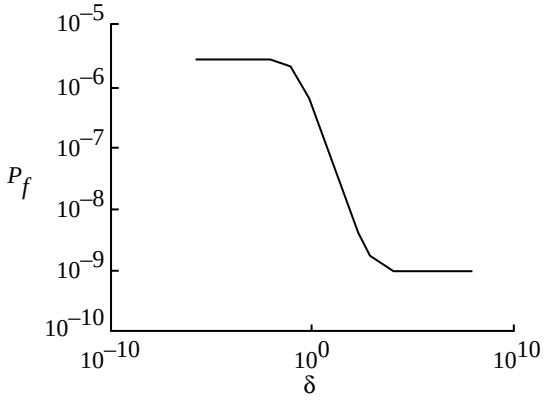


Figure 12. Degradable triplex as funtion of δ.

From these graphs it can be seen that the system unreliability P_f is much more sensitive to λ than to δ. The P_f over much of the graph is proportional to λ^2 and inversely proportional to δ. It can also be seen that a point of diminishing return exists for δ. However, typically this situation occurs at reconfiguration rates much higher than those which can be realized in a physical architecture.

4.2. Triad to Simplex

The model presented in section 4.1 was unrealistic in one major respect—the reconfiguration process from the dual to the simplex was assumed to be perfect. In other words, when either of the two processors failed, the system diagnosed which of the two processors was the faulty one with complete accuracy. When using a majority voter on three or more processors, such an assumption is not unrealistic. However, with only two good processors, this diagnosis cannot be perfect. In a later section, the use of self-test programs to diagnose failure in a dual system will be explored. In this section, the option of

degrading directly from a triplex to a simplex (i.e., avoiding the dual mode) will be examined. The model shown in figure 13 describes this system.

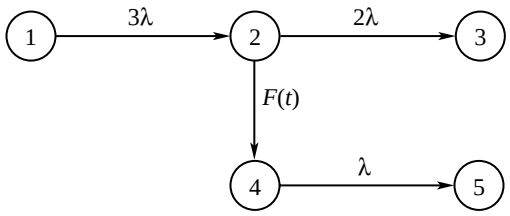


Figure 13. Model of triplex to simplex system.

The horizontal transitions represent fault arrivals. The vertical transition represents system recovery. The recovery transition is labeled with a distribution function rather than a rate to indicate that the transition is not exponential. The transition rate from state (1) to state (2) is 3λ because three active processors can fail. When one of those processors fails, the system is in state (2). Before reconfiguration occurs, two active processors can fail; thus, the transition from state (2) to death state (3) with rate 2λ competes with the recovery transition. Reconfiguration consists of discarding both the faulty processor and one of the working processors. Thus, the transition rate from state (4) to state (5) is λ because only one processor remains in the active configuration.

4.3. Degradable Quadraplex System

The model in figure 14 describes a degradable quadraplex system.

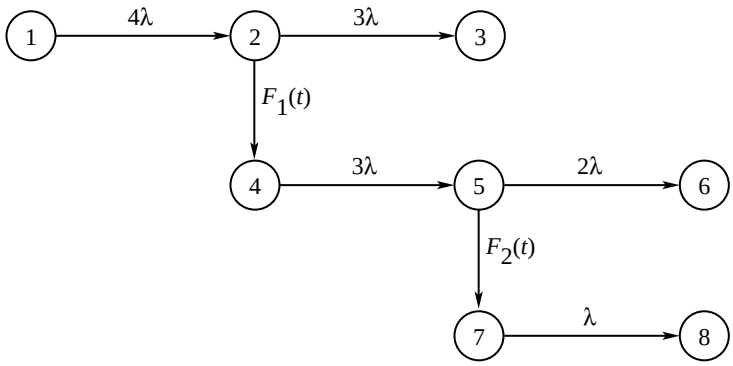


Figure 14. Model of degradable quadraplex.

This system starts with four working processors. When one of those four processors fails (state (2)), the reconfiguration process consists of removing the faulty processor, thereby leaving a triad of processors (state (4)). When one of the three remaining processors fails (state (5)), the reconfiguration process removes the faulty processor plus one of the working processors, which results in a simplex system (state (7)). Note that different functions are used for the transition from state (5) to state (7) and from state (2) to state (4). These different functions are necessary if the reconfiguration process, and hence rate, varies as a function of the state.

4.4. Triad With One Spare

In the previous models, the reconfiguration process removed a faulty processor and the system continued to operate with degraded levels of redundancy. This section provides a brief introduction to the technique of sparing. That is, replacing a faulty processor with a spare processor. This technique will be explored in detail in section 7.

Suppose a triplex system has one spare that does not fail when it is inactive. The model shown in figure 15 shows this system.

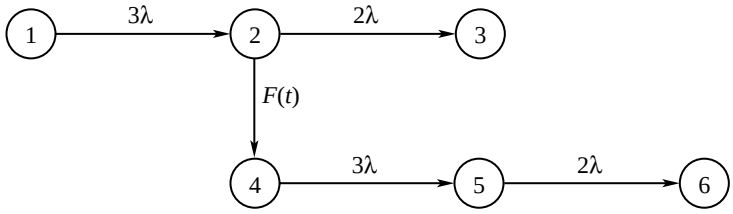


Figure 15. Model of triplex with one spare.

State (2) represents two good processors and one faulty processor. The transition from state (2) to state (4) represents the detection and the isolation of the faulty processor and its replacement with a spare processor. While the system is in state (2), two active working processors can fail; thus, the rate of the transition to death state (3) is 2λ. After reconfiguration occurs, once again three active processors can fail; thus, the transition rate from state (4) to state (5) is 3λ. This model assumes the system does not immediately degrade to simplex upon the next failure, but rather operates in duplex until the next failure brings system failure.

4.5. Reliability Analysis During Design and Validation

In this paper, two major categories of application of the reliability analysis techniques are discussed: design and validation. During the design phase of system development, it is often necessary to perform trade-off analyses in order to make appropriate design decisions. Critical parameters, such as system recovery times, must be estimated with little, if any, data. After the system has been developed, these critical parameters must be measured to validate the reliability of the system implementation. Experiments are performed to measure the actual system recovery times. The reliability models can then be solved by using these accurate values for the parameters of the model. During the design phase, the recovery processes are often modeled with exponential distributions because the actual distribution is unknown. During the validation phase, the observed distribution is used for the reliability analysis.

5. Reliability Analysis Programs

Before the techniques of modeling fault-tolerant systems are explored further, the input language for the SURE reliability analysis program will be presented. The same input language is used for the STEM and the PAWS reliability analysis programs. These programs are described in section 5.4.

In the remainder of this paper, the models will be presented in the SURE input language. This approach is desirable for two reasons. First, as the models increase in complexity, it soon becomes impractical to present them graphically. Second, these programs can be used to solve the models as functions of any model parameter. This presentation style will provide insight into the nature of the systems being modeled.

5.1. Overview of SURE

The SURE program is a reliability analysis tool for ultrareliable computer system architectures (refs. 1 and 9) and is based upon computational methods developed at Langley Research Center (ref. 10). These methods provide an efficient means for computing accurate upper and lower bounds for the state probabilities of a large class of semi-Markov models.

Models are defined in SURE by enumerating all transitions of the model. The SURE program distinguishes between fast and slow transitions. If the mean transition time μ is small with respect to the

mission time, that is, $\mu < T$, then the transition is fast. Otherwise, it is slow. Slow transitions are assumed to be exponentially distributed by the SURE program. Fast transitions can have an arbitrary distribution. The SURE user must supply the mean and the standard deviation of the transition time. If multiple competing fast transitions from a state occur, the user must supply the respective transition probabilities along with the conditional means and standard deviations. Probably the easiest way to learn the SURE input language is by example.

The input to the SURE program for the triad plus one spare in figure 15 is

```
LAMBDA = 1E-6 TO* 1E-2 BY 10;
MU = 2.7E-3;
SIGMA = 1.3E-2;

1,2 = 3*LAMBDA;
2,3 = 2*LAMBDA;
2,4 = <MU,SIGMA>;
4,5 = 3*LAMBDA;
5,6 = 2*LAMBDA;
TIME = 10;
```

The first three statements equate values to identifiers. The first identifier `LAMBDA` represents the processor failure rate. The next two identifiers `MU` and `SIGMA` are the mean and the standard deviation of the recovery time distribution from state (2) to state (4). The next five statements define the transitions of the model. If the transition is a slow fault-arrival process, then only the exponential rate must be provided. For example, the last statement defines a transition from state (5) to state (6) with rate 2λ. If the transition is a fast recovery process, then the mean and the standard deviation of the recovery time must be given. For example, the statement `2,4 = <MU,SIGMA>` defines a transition from state (2) to state (4) with mean recovery time `MU1` and standard deviation `SIGMA1`. The last statement sets the mission time to 10 hr.

The following illustrative interactive session uses SURE to process the preceding model. The model description has been stored in a file named `TP1S`. The user input follows the `?` prompt.

```
air58% sure

 SURE V7.9.8 NASA Langley Research Center

 1? read TP1S

 2:   LAMBDA = 1E-6 TO* 1E-2 BY 10;
 3:   MU = 2.7E-3;
 4:   SIGMA = 1.3E-2;
 5:
 6:   1,2 = 3*LAMBDA;
 7:   2,3 = 2*LAMBDA;
 8:   2,4 = <MU,SIGMA>;
 9:   4,5 = 3*LAMBDA;
10:   5,6 = 2*LAMBDA;
11:   TIME = 10;

      0.02 SECS. TO READ MODEL FILE
12? run
MODEL FILE = TP1S.mod          (SURE V7.9.8 22 Jun 94 12:01:38
```

```
  LAMBDA      LOWERBOUND   UPPERBOUND  COMMENTS                RUN #1
-----------  -----------  -----------  ----------------------------
1.00000e-06  1.39719e-13  1.65000e-13
1.00000e-05  1.65286e-11  1.92000e-11
1.00000e-04  4.20456e-09  4.62000e-09
1.00000e-03  2.92225e-06  3.16200e-06
1.00000e-02  2.35025e-03  2.47391e-03 <ExpMat>

2 PATH(S) TO DEATH STATES
Q(T) ACCURACY >= 14 DIGITS
0.000 SECS. CPU TIME UTILIZED
13? plot XYLOG
14? exit
```

The first statement uses the READ command to input the model description file. It should be noted that λ is defined as a variable over a range of values in this file. This definition directs the SURE program to automatically perform a sensitivity analysis as a function of λ over the specified range. Statement 11 defines the mission time to be 10 hr. The SURE program computes an upper and a lower bound on the probability of system failure. Usually these bounds are within 5 percent of each other, and thus they usually provide an accurate estimate of system failure. Statement 13 directs the program to plot the output on the graphics device. Figure 16 shows the graph generated by this command. The XYLOG argument causes SURE to plot the *X*- and the *Y*-axes with logarithmic scales.

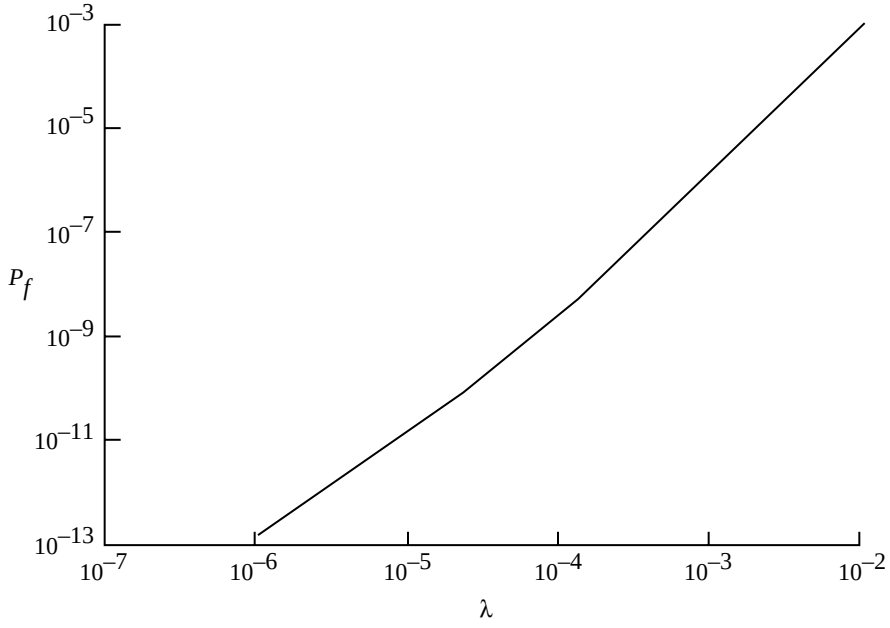


Figure 16. Plot of SURE program output.

Because the upper and the lower bounds are very close, the bounds appear as one line in the plot. The <ExpMat> statement in the COMMENTS field indicates that a slower numerical method was required for this particular parameter value.

5.2. Model Definition Syntax

In these sections, more detail is presented. These sections can be omitted during the first reading and used as a reference when something is encountered that is not clear. The following conventions will be used to facilitate the description of the SURE input language:

1. All reserved words will be capitalized in typewriter-style print.
2. Lowercase words that are in italics indicate items that are to be replaced by something defined elsewhere.
3. Items enclosed in double square brackets ⟦ ⟧ can be omitted.
4. Items enclosed in braces { } can be omitted or repeated as many times as desired.

5.2.1. Lexical details. The state numbers must be positive integers between zero and the MAXSTATE implementation limit, usually 25 000 or larger. This limit can be increased simply by changing the MAXSTATE constant in the program and recompiling. The transition rates and the conditional means and standard deviations are floating point numbers. The Pascal REAL syntax is used for these numbers. The semicolon is used for statement termination. Therefore, more than one statement may be entered on a line. Comments may be included any place that blanks are allowed. The beginning of a comment is indicated with (* and the termination of a comment is indicated with *). The SURE program prompts the user for input by a line number followed by a question mark.

5.2.2. Constant definitions. The user may equate numbers to identifiers. Thereafter, these constant identifiers may be used instead of the numbers. For example,

```
LAMBDA = 0.001;
RECOVER = 1E-4;
```

Constants may also be defined in terms of previously defined constants

```
GAMMA = 10*LAMBDA;
```

In general, the syntax is

name = expression;

where *name* is a string of up to eight letters, digits, and underscores that begin with a letter and *expression* is an arbitrary mathematical expression as described in section 5.2.4.

5.2.3. Variable definition. To facilitate parametric analyses, a single variable may be defined. A range is given for this variable. The SURE system will compute the system reliability as a function of this variable. The following statement defines LAMBDA as a variable with range 0.001 to 0.009:

```
LAMBDA = 0.001 TO 0.009;
```

Only one such variable may be defined. A special constant, POINTS, defines the number of points to be computed over this range. The method used to vary the variable over this range can be either geometric or arithmetic and is best explained by example. Suppose POINTS = 4; then the geometric range

```
XV = 1 TO* 1000;
```

would use XV values of 1, 10, 100, and 1000, while the arithmetic range

```
XV = 1 TO+ 1000;
```

would use XV values of 1, 333, 667, and 1000. An asterisk following the TO implies a geometric range, while TO+ or simply TO implies an arithmetic range. In addition, the BY option is available. With the above syntax and BY *increment*, the value of POINTS is automatically set so that the value is varied by adding or multiplying the specified increment. For example,

```
LAMBDA = 1E-5 TO* 1E-2 BY 10;
```

sets POINTS equal to 4 and uses the values of 1E-5, 1E-4, 1E-3, and 1E-2 for LAMBDA. The statement

```
CX = 3 TO+ 5 BY 1;
```

sets POINTS equal to 3 and uses the values of 3, 4, and 5 for CX . In general, the syntax is

var = *expression* TO s ⟦ *c* ⟧ *expression* ⟦ BY *increment* ⟧;

where *var* is a string of up to eight letters and digits beginning with a letter, *expression* is an arbitrary mathematical expression, which is described in the section 5.2.4, and the optional *c* is a + or *. The BY clause is optional; if it is used, then *increment* is any arbitrary expression.

5.2.4. Expressions. When specifying transition or holding time parameters in a statement, arbitrary functions of the constants and the variable may be used. The following operators may be used: + for addition, – for subtraction, * for multiplication, / for division, and ** for exponentiation.

The following standard Pascal functions may be used: EXP(X), LN(X), SIN(X), COS(X), ARCSIN(X), ARCCOS(X), ARCTAN(X), and SQRT(X). Both ( ) and [ ] may be used for grouping in the expressions. The following are permissible expressions:

```
2E-4
1.2*EXP(-3*ALPHA);
7*ALPHA + 12*L;
ALPHA*(1+L) + ALPHA**2;
2*L + (1/ALPHA)*[L + (1/ALPHA)];
```

5.2.5. Slow transition description. A slow transition is completely specified by citing the source state, the destination state, and the transition rate. The syntax is as follows:

source, dest = *rate*;

where *source* is the source state, *dest* is the destination state, and *rate* is any valid expression defining the exponential rate of the transition. The following are valid SURE statements:

```
PERM = 1E-4;
TRANSIENT = 10*PERM;

1,2 = 5*PERM;
1,9 = 5*(TRANSIENT + PERM);
2,3 = 1E-6;
```

5.2.6. Fast transition description. To enter a fast transition, the following syntax is used:

source, dest = < *mu, sig* ⟦, *frac* ⟧ >;

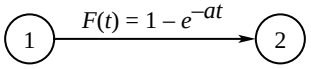
where *mu* defines the conditional mean transition time, *sig* defines the conditional standard deviation of transition time, *frac* defines the transition probability, *source* defines the source state, and *dest* defines the destination state.

The parameter *frac* is optional. If omitted, the transition probability is assumed to be 1.0, that is, only one fast transition. All the following are valid:

```
2,5 = <1E-5, 1E-6, 0.9>;

THETA = 1E-4;
5,7 = <THETA, THETA*THETA, 0.5>;
7,9 = <0.0001,THETA/25>;
```

5.2.7. FAST exponential transition description. Often when performing design studies, experimental data are unavailable for the fast processes of a system. In this case, some properties of the underlying processes must be assumed. For simplicity, these fast transitions are often assumed to be exponentially distributed. However, it is still necessary to supply the conditional mean and standard deviation to the SURE program because these transitions are fast. If only one fast transition from a state occurs, then these parameters are easy to determine. Suppose a fast exponential recovery occurs from state (1) to state (2) with unconditional rate *a*



The SURE input is simply

```
1,2 = < 1/a, 1/a, 1 >;
```

In this case, the conditional mean and standard deviation are equivalent to the unconditional mean and standard deviation. The above transition can be specified by using the following syntax:

```
1,2 = FAST a;
```

When multiple recoveries are present from a single state, then care must be exercised to properly specify the conditional means and standard deviations required by the SURE program. Consider the model in figure 17 where the unconditional distributions are

$$F_1(t) = 1 - e^{-at}$$

$$F_2(t) = 1 - e^{-bt}$$

$$F_3(t) = 1 - e^{-ct}$$

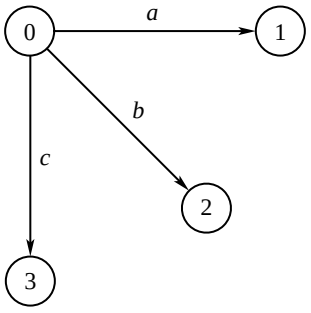


Figure 17. Model of three competing fast transitions.

The SURE input describing the previous model is

```
0,1 = < 1/(a+b+c), 1/(a+b+c), a/(a+b+c) >;
0,2 = < 1/(a+b+c), 1/(a+b+c), b/(a+b+c) >;
0,3 = < 1/(a+b+c), 1/(a+b+c), c/(a+b+c) >;
```

Note that the conditional means and standard deviations are not equal to the unconditional means and standard deviations (e.g., the conditional mean transition time from state (0) to state (1) is not equal to `1/a`.) The following can be used to define the model in figure 17:

```
0,1 = FAST a;
0,2 = FAST b;
0,3 = FAST c;
```

The SURE program automatically calculates the conditional parameters from the unconditional rates `a`, `b`, and `c`. The user may mix `FAST` exponential transitions with other general transitions. However, care must be exercised in specifying the conditional parameters of the nonexponential fast recoveries to avoid inconsistencies. For more details see reference 1.

5.3. SURE Commands

In this section a brief summary of some of the SURE commands is given.

5.3.1. READ command. A sequence of SURE statements may be read from a disk file. The following interactive command reads SURE statements from a disk file named sift.mod:

```
READ sift.mod;
```

If no file name extension is given, the default extension `.mod` is assumed. A user can build a model description file by using a text editor and then use the `READ` command to read it into the SURE program.

5.3.2. RUN command. After a semi-Markov model has been fully described to the SURE program, the `RUN` command is used to initiate the computation

`RUN` *outname;*

The output is written to file *outname*. If *outname* is omitted the output is written to the user terminal.

5.3.3. LIST constant. The amount of information output by the program is controlled by this command. Four list modes are available. For `LIST = 0;`, no output is sent to the terminal, but the results can still be displayed by using the `PLOT` command. For `LIST = 1;`, only the upper and the lower bounds on the probability of total system failure are listed. This is the default. For `LIST = 2;`, the probability bounds for each death state in the model are reported along with the totals. For `LIST = 3;`, every path in the model and its probability of traversal is listed. The probability bounds for each death state in the model are reported along with the totals.

5.3.4. START constant. The `START` constant is used to specify the start state of the model. If the `START` constant is not used, the program will use the source state (i.e., the state with no transitions into it) of the model, if one exists.

5.3.5. TIME constant. The `TIME` constant specifies the mission time. For example, when the user sets `TIME = 1.3`, the program computes the probability of entering the death states of the model within time 1.3. The default value of `TIME` is 10. All parameter values must be in the same units as the `TIME` constant.

5.3.6. PRUNE and WARNDIG constants. The time required to analyze a large model can often be greatly reduced by model pruning. The SURE program automatically selects a pruning level upon detection of the first death state. This feature can be disabled by setting the AUTOPRUNE constant to zero, AUTOPRUNE = 0. The default value of AUTOPRUNE is 1. Alternatively, the SURE user can specify the level of pruning by using the PRUNE constant. A path is traversed by the SURE program until the probability of reaching the current point on the path falls below the pruning level. For example, if PRUNE = 1E-14 and the upper bound falls below 1E-14 at any point on the path, the analysis of the path is terminated and its probability is added to the upper bound. The sum of all occupancy probabilities of the pruned states is reported in the following format:

```
<prune x.xxx>
```

The SURE program will warn the user when the pruning process is too severe, that is, when the pruning produces a result with less than WARNDIG digits of accuracy. In this case, the upper bound is still an upper bound, but it is not close to the lower bound. The default value of WARNDIG is 2.

These commands are explained in more detail in section 11.

5.4. Overview of STEM and PAWS Programs

The STEM (Scaled Taylor Exponential Matrix) and the PAWS (Padé Approximation With Scaling) programs were developed at Langley Research Center for solving pure Markov models (i.e., all transitions are exponentially distributed). The input language for these two programs is the same as for the SURE program. The only major difference is that the fast recovery transition statement is interpreted differently. The following statement

source, dest = < *mu, sig* [[, *frac*]] >;

is interpreted as

source, dest = frac/mu ;

where *source* is the source state, *dest* is the destination state, *mu* is an expression that defines the conditional mean transition time, *sig* is an expression that defines the conditional standard deviation of transition time, and *frac* is an expression that defines the transition probability. If the third parameter *frac* is omitted, a value of 1 is used.

For more information on the solution techniques used by these two reliability analysis programs, see reference 11.

5.5. Introduction to SURE Mathematics

In this section, the bounding theorem upon which SURE is based is presented. First, some notation is developed; then the details of the theorem are presented. This section can be omitted on first reading because later sections do not depend upon the content of this section.

5.5.1. Path-step classification and notation. The presentation of the SURE bounding theorem is simplified by first developing some notation. The theorem provides bounds on the death state probabilities at a specified time. It is assumed that the system is initially in a single state. That is, the probability that the system is in a single state at time 0 = 1.0. The SURE program uses some additional techniques not presented here that enables assignment of initial probability to multiple states. These techniques are discussed in section 16 and in the appendix. This initial single state is called the start state. The SURE program finds every path from the start state to a death state. The contribution of each path to system failure is calculated separately by using the semi-Markov bounding theorem of White, which is described in section 5.5.

Let each state along the path be classified into one of three classes that are distinguished by the type of transitions leaving the state. A state and all transitions leaving it will be referred to as a "path step." The transition on the path that is currently being analyzed will be referred to as the "on-path transition." The remaining transitions will be referred to as the "off-path transitions." The classification is made on the basis of whether on-path and off-path transitions are slow or fast. If no off-path transitions exist, the path step is classified as if it contained a slow off-path transition.

5.5.2. Class 1 path step; slow on path, slow off path. If all transitions leaving the state are slow, then the path step is class 1. The rate of the on-path exponential transition is λ_i. (See fig. 18.) An arbitrary number of slow off-path transitions can occur. The sum of their exponential transition rates is γ_i. If any off-path transitions are not slow, then the path step is in class 3. The path steps $1 \rightarrow 2$, $4 \rightarrow 5$, and $5 \rightarrow 6$ in the model of the triad plus one spare shown in figure 15 are examples of this class.

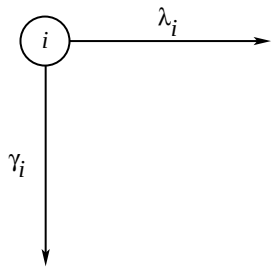


Figure 18. Class 1 path step. Slow on path; slow off path.

5.5.3. Class 2 path step; fast on path, arbitrary off path. If the on-path transition is fast, then the path step is class 2. (See fig. 19.) An arbitrary number of slow or fast off-path transitions may exist. As before, the slow off-path, exponential transitions can be represented as a single transition with a rate ε_i equal to the sum of all the slow off-path transition rates. The path step $2 \rightarrow 4$ in the model of the triad plus one spare shown in figure 15 are examples. The distribution of the fast on-path transition is $F_{i,1}$. The distribution of time for the *k*th fast transition from state (*i*) is referred to as $F_{i,k}$ (i.e., the probability that the next transition out of state (*i*) goes into state (*k*) and occurs within time *t* is $F_{i,k}$). Three measurable parameters must be specified for each fast transition: the transition probability $\rho(F^*_{i,k})$, the conditional mean $\mu(F^*_{i,k})$, and the variance $\sigma^2(F^*_{i,k})$, given that this transition occurs. The asterisk is used to note that the parameters are defined in terms of the conditional distributions combined with definition. Mathematically, these parameters are defined as follows:

$$\rho\,(F^*_{i,k}) = \int_0^\infty \prod_{j \neq k} [\,1 - F_{i,j}\,(t)\,]\, dF_{i,k}\,(t)$$

$$\mu\,(F^*_{i,k}) = \frac{1}{\rho\,(F^*_{i,k})} \int_0^\infty t \prod_{j \neq k} [\,1 - F_{i,j}\,(t)\,]\, dF_{i,k}\,(t)$$

$$\sigma^2\,(F^*_{i,k}) = \frac{1}{\rho\,(F^*_{i,k})} \int_0^\infty t^2 \prod_{j \neq k} [\,1 - F_{i,j}\,(t)\,]\, dF_{i,k}\,(t) - \mu^2\,(F_{i,k})$$

Experimentally, these parameters correspond to the fraction of times that a fast transition is successful and the mean and the variance of the conditional distribution, given that the transition occurs.

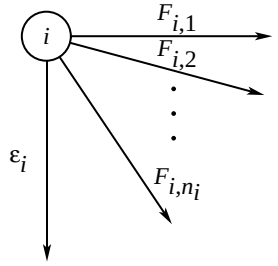


Figure 19. Class 2 path step. Fast on path; arbitrary off path.

Note, in any experiment where competing processes in a system are studied, the observed empirical distributions will be conditional. The time it takes a system to transition to the next state will only be observed when that transition occurs. These expressions are defined independently of the exponential transitions ε_j. Consequently, the sum of the fast transition probabilities $\sum\rho(F^*_{i,k})$ must be 1. In particular, if only one fast transition occurs, its probability is 1 and the conditional mean is equivalent to the unconditional mean. (The SURE user does not have to deal explicitly with the unconditional distributions $F_{i,k}$. However, to develop the mathematical theory, the distributions must be used.)

5.5.4. Class 3 path step; slow on path, fast off path. The on-path transition must be slow for a path step to be categorized as class 3. Both slow and fast off-path transitions can exist; however, at least one off-path transition must be fast. (See fig. 20.) The path step 2 → 3 in the model of the triad plus one spare shown in figure 15 are in this class. The slow on-path transition rate is α_j. The sum of the slow off-path transition rates is β_j. As in class 2, the transition probability $\rho(G^*_{j,k})$, the conditional mean $\mu(G^*_{j,k})$, and the conditional variance $\sigma^2(G^*_{j,k})$ must be given for each fast off-path transition with distribution $G_{j,k}$. Two letters are used to help track whether the transition is a class 2 (labeled *F*) or class 3 (labeled *G*) in the current path.

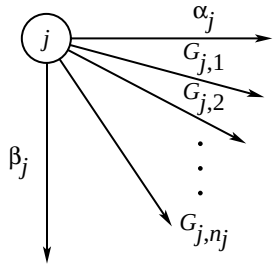


Figure 20. Class 3 path step. Slow on path; fast off path.

In either case, the SURE user supplies the conditional mean, the conditional standard deviation, and the transition probability. Although, the parameters described above suffice to specify a class 3 path step to SURE, the mathematical theory is more easily expressed in terms of the holding time in the state. The holding time in a state is the time the system remains in the state before it transitions to some other state. The bounding theorem is expressed by using a slightly different holding time, which will be referred to as "recovery holding time" to prevent confusion. The recovery holding time is the holding time in the state with the slow exponential distributions removed. Because the slow exponential transitions occur at a rate many orders of magnitude less than the fast transitions, the recovery holding time is

approximately equal to the traditional holding time. Let H_j represent the distribution of the recovery holding time in state *(j)*

$$H_j(t) = 1 - \prod_{k=1}^{n_j} [1 - G_{j,k}(t)]$$

Then the following parameters are used in the theorem:

$$\mu(H_j) = \int_0^\infty \prod_{k=1}^{n_j} [1 - G_{j,k}(t)]\, dt$$

$$\sigma^2(H_j) = 2\int_0^\infty t \prod_{k=1}^{n_j} [1 - G_{j,k}(t)]\, dt - \mu^2(H_j)$$

These parameters are the mean and the variance of the holding time in state (*j)* without consideration for the slow exponential transitions (i.e., with the slow exponential transitions removed). These parameters do not have to be supplied to the SURE program. The SURE program derives these parameters from the other available inputs, such as $\rho(G^*_{j,k})$, $\mu(G^*_{j,k})$, and $\sigma^2(G^*_{j,k})$, as follows:

$$\mu(H_j) = \sum_{k=1}^{n_j} \rho(G^*_{j,k})\ \mu(G^*_{j,k})$$

$$\sigma^2(H_j) = \left\{ \sum_{k=1}^{n_j} \rho(G^*_{j,k}) \left[\sigma^2(G^*_{j,k}) + (\mu^2 G^*_{j,k}) \right] \right\} - \mu^2(H_j)$$

The parameters $\rho(G^*_{j,k})$, $\mu(G^*_{j,k})$, and $\sigma^2(G^*_{j,k})$ are defined exactly as the class 2 path step parameters.

Although the fast distributions are specified without consideration of the competing slow exponential transitions, the theorem gives bounds that are correct in the presence of such exponential transitions. The parameters were defined in this manner to simplify the process of specifying a model. Throughout the paper, the holding time in a state in which the slow transitions have been removed will be referred to as "recovery holding time." For convenience, when referring to a specific path in the model, the distribution of a fast on-path transition will be indicated by a single subscript that specifies the source state. For example, if the transition with distribution $F_{j,k}$ is the on-path transition from state (*j*), then it can be referred to as F_j, where $F_{j,k}$ is the *k*th fast transition from state (*j*) and F_j is the on-path fast transition from state (*j*).

5.5.5. SURE bounding theorem. With the classification and notation from the previous sections, the bounding theorem can now be presented. The proof of this theorem has been published in references 1 and 12 and is not given in this paper.

Theorem: The theorem of White states that the probability *D(T)* of entering a particular death state within the mission time *T*, following a path with *k* class 1 path steps, *m* class 2 path steps, and *n* class 3 path steps is bounded as follows (refs. 1 and 12):

$$LB < D(T) < UB$$

where

$$UB = Q(T)\prod_{i=1}^{m}\rho(F_i^*)\prod_{j=1}^{n}\alpha_j\mu(H_j)$$

$$LB = Q(T-\Delta)\prod_{i=1}^{m}\rho(F_i^*)\left[1-\varepsilon_i\mu(F_i^*)-\frac{\mu^2(F_i^*)+\sigma^2(F_i^*)}{r_i^2}\right]$$

$$\times\prod_{j=1}^{n}\alpha_j\left\{\mu(H_j)-\frac{(\alpha_j+\beta_j)\left[\mu^2(H_j)+\sigma^2(H_j)\right]}{2}-\frac{\mu^2(H_j)+\sigma^2(H_j)}{s_j}\right\}$$

for all $r_i, s_j > 0$ and $\Delta = r_1 + r_2 + ... + r_m + s_1 + s_2 ... + s_n$ and

$Q(T)$ = the probability of traversing a path consisting of only the class 1 path steps within time T

The theorem is true for any $r_i > 0$ and $s_j > 0$ provided that $\Delta < T$. Different choices of these parameters will lead to different bounds. The SURE program uses the following values of r_i and s_j:

$$r_i = \left\{2T\left[\mu^2(F_i^*)+\sigma^2(F_i^*)\right]\right\}^{1/3}$$

$$s_j = \left\{T\left[\frac{\mu^2(H_j)+\sigma^2(H_j)}{\mu(H_j)}\right]\right\}^{1/2}$$

These values have been found to give very close bounds in practice and are usually very near the optimal choice (ref.1).

Two simple algebraic approximations for $Q(T)$ were given by White in reference 13. One approximation overestimates and one approximation underestimates, and are given respectively as

$$Q(T) < Q_u(T) = \frac{\lambda_1\lambda_2\lambda_3\ldots\lambda_k T^k}{k!}$$

$$Q(T) > Q_l(T) = Q_u(T)\left[1-\frac{T}{k+1}\sum_{i=1}^{k}(\lambda_i+\gamma_i)\right]$$

Both $Q_u(T)$ and $Q_l(T)$ are close to $Q(T)$ as long as $\Sigma_{i=1}^{k}(\lambda_i+\gamma_i)\,T$ is small. That is, as long as the mission time is short compared with the average lifetime of the components. The SURE program uses the following slightly improved upper bound on $Q(T)$:

$$Q(T) < Q_u^*(T) = \frac{1}{|S|!}\prod_{i\in S}(\lambda_i T)$$

where

$$S = \{i \mid \lambda_i T < 1\}$$

This bound is obtained by removing all the fast exponential transitions from the $Q(T)$ model. Because the path is shorter, the probability of reaching the death state is larger than the original $Q(T)$ model. These algebraic bounds on $Q(T)$ are used when the `QTCALC` option is set equal to zero. When the `QTCALC = 1` option is used, a differential equation solver is used to calculate $Q(T)$ and $Q(T - \Delta)$. If `QTCALC = 2` (the default), then the SURE program automatically selects the most appropriate method.

5.5.6. Algebraic analysis with SURE upper bound. The SURE upper bound can be used symbolically, as well as numerically, to gain insight in the reliability properties of a system. This technique will be illustrated by application to the model shown in figure 21.

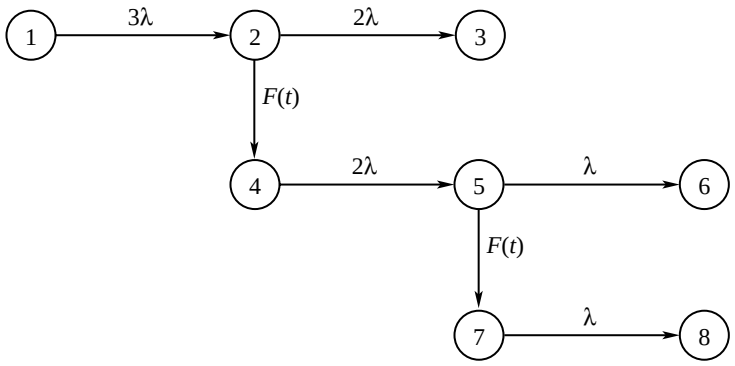


Figure 21. Model of degradable triplex.

In this model there are three death states ((3), (6), (8)) and three paths to death states:

$1 \rightarrow 2 \rightarrow 3$

$1 \rightarrow 2 \rightarrow 4 \rightarrow 5 \rightarrow 6$

$1 \rightarrow 2 \rightarrow 4 \rightarrow 5 \rightarrow 7 \rightarrow 8$

The SURE upper bound can be applied to each path to obtain an algebraic formula for the probability of entering the death states within the mission time. The SURE algebraic upper bound is

$$UB = \prod_{i=1}^{k} \lambda_i T/i \prod_{i=1}^{m} \rho(F_i^*) \prod_{j=1}^{n} \alpha_j \mu(H_j)$$

The first path, $1 \rightarrow 2 \rightarrow 3$ has two steps. The first step is a class 1 path step, and thus contributes $\lambda_i T/i = 3\lambda T$. The second path step is a class 3, and thus, contributes $\alpha_j \mu(H_j) = 2\lambda\mu$. Hence,

$$P_3(t) \approx (3\lambda T)(2\lambda\mu) = 6\lambda^2 T\mu$$

The second path, $1 \rightarrow 2 \rightarrow 4 \rightarrow 5 \rightarrow 6$ has four steps. The first step, $1 \rightarrow 2$, is a class 1 path step, and thus contributes $\lambda_i T/i = 3\lambda T$ (cf. $i = 1$). The second step, $2 \rightarrow 4$, is a class 2 path step, and thus contributes $\rho(F_i^*)$. Because only one recovery exists, $\rho(F_i^*) = 1$. The third step, $4 \rightarrow 5$, is a class 1 path step, and thus contributes $\lambda_i T/i = 2\lambda T/2$ (cf. $i = 2$). The fourth path step is a class 3, and thus contributes $\alpha_j \mu(H_j) = \lambda\mu$. Hence,

$$P_6(t) \approx (3\lambda T)(1)(2\lambda T/2)(\lambda\mu) = 3\lambda^3 T^2\mu$$

In the third path, 1 → 2 → 4 → 5 → 7 → 8, the first three steps are the same as path 2. The fourth step, 5 → 7, is a class 2 path step, and thus contributes $\rho(F_i^*)$. Because only one recovery transition occurs from state (5), $\rho(F_i^*) = 1$. The fifth step is a class 1 path step, and thus contributes $\lambda_i T/i = \lambda T/3$. Hence,

$$P_8(t) \approx (3\lambda T)\ (1)\ (2\lambda T/2)\ (1)\ (\lambda T/3) = \lambda^3 T^3$$

From these formulas, the sensitivity of the system unreliability to the model parameters is immediately seen. For example, partial derivatives, if desired, can easily be taken because the formulas are algebraic.

6. Reconfiguration by Degradation

In this section, the technique of reconfiguration by degradation will be explored. The first example is a simple degradable *n*-plex. Later sections introduce more complicated aspects, such as fail-stop dual processors and self-test processors.

6.1. Degradable 6-Plex

Reconfiguration can be utilized with levels of redundancy greater than three. The Software Implemented Fault Tolerance (SIFT) computer system is an example of such an architecture (ref. 14). The SIFT computer initially contains six processors. At this level of redundancy, two simultaneously faulty computers can be tolerated. As processors fail, the system degrades into lower levels of redundancy. Thus, SIFT is a degradable 6-plex. It is convenient to identify the states of the system by an ordered pair `(NC,NF)`, where `NC` is the number of processors currently in the configuration and `NF` is the number of faulty processors in the configuration. The semi-Markov model for the SIFT system is shown in figure 22. Three main concepts dictate the structure of this model

1. Every processor in the current configuration fails at rate λ.

2. The system removes faulty processors with mean recovery time *m*.

3. A majority of processors in the configuration must not have failed for the system to be safe.

A few subtle points must also be considered. First, this model implicitly assumes that the reconfiguration process is independent of the configuration of the system. For example, the mean recovery time from state (6,1) is the same as from states (5,1), (4,1), and (3,1). A system in a degraded configuration may recover slower (because less processing power is available) or faster (because fewer processors remain to be examined to find the faulty one). To determine the speed of recovery in a degraded configuration would require extensive fault injection in numerous configurations and would be a very expensive process. If these experiments were done, however, one could easily modify the model in figure 22 to contain this information. Second, the mean and the standard deviation of the recovery time from states with two active faults is probably different from that of the states with only one active fault. Note that in the model in figure 22, these transitions are labeled with `<m_2,s_2>`. These parameters would have to be measured with double fault injections. In the absence of experimental data, it is convenient to let `m_2 = m/2` and `s_2 = s/2`. If the detection and isolation and reconfiguration of the two faults behave like two independent exponential processes, this assumption is reasonable. Actually the situation is quite complicated from a strict theoretical viewpoint. A semi-Markov model has a memoryless property, and thus, cannot capture the concept that the first reconfiguration process is already in progress when the second fault arrives. The second fault carries the system into a new state where the progress in reconfiguring the first fault is forgotten. The semi-Markov model thus overestimates the time to reconfigure the first fault in the presence of a second fault. This assumption is usually conservative. The following SURE run reveals the sensitivity of the failure probability to the mean

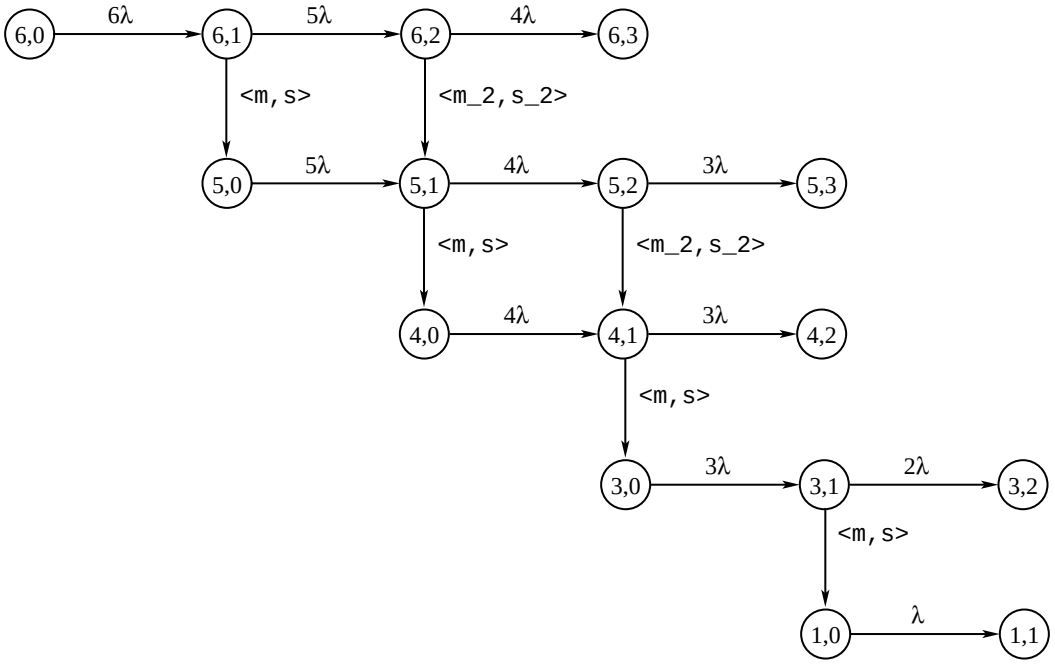


Figure 22. Semi-Markov model of SIFT computer system.

reconfiguration time. The SURE program requires that the states be defined by a single number. Therefore, the state vectors must be mapped onto a set of integers.

```
% sure
  SURE V7.5   NASA Langley Research Center
  1? read sift
  2: LAMBDA = 5.0E-4;
  3: m = 1E-4 TO* 1E-1 BY 10;
  4: s = 6E-4;
  5: m_2 = m/2;
  6: s_2 = s/2;
  7: 1,2 = 6*LAMBDA;
  8: 2,3 = 5*LAMBDA;
  9: 3,4 = 4*LAMBDA;
 10: 2,5 = <m,s>;
 11: 5,6 = 5*LAMBDA;
 12: 3,6 = <m_2,s_2>;
 13: 6,7 = 4*LAMBDA;
 14: 7,8 = 3*LAMBDA;
 15: 6,9 = <m,s>;
 16: 9,10 = 4*LAMBDA;
 17: 7,10 =<m_2,s_2>;
 18: 10,11 = 3*LAMBDA;
 19: 10,12 = <m,s>;
 20: 12,13 = 3*LAMBDA;
 21: 13,14 = 2*LAMBDA;
 22: 13,15 = <m,s>;
 23: 15,16 = 1*LAMBDA;
```

```
      0.15 SECS. TO READ MODEL FILE
24? run

MODEL FILE = sift.mod                    SURE V7.5 26 Feb 90 14:23:14

   M          LOWERBOUND   UPPERBOUND   COMMENTS              RUN #1
-----------  -----------  -----------  --------------------------
1.00000e-04  9.17736e-12  9.75265e-12
1.00000e-03  1.21626e-11  1.32216e-11
1.00000e-02  4.34597e-11  5.48450e-11
1.00000e-01  6.77898e-10  1.16481e-09

15 PATH(S) TO DEATH STATES
1.233 SECS. CPU TIME UTILIZED
26? exit
```

Finally, it should be noted that the SIFT computer degrades from a triplex to a simplex. Thus, the reconfiguration transition out of state (3,1) carries the system into state (1,0).

6.2. Single-Point Failures

All previous models assumed that no single-point failures existed in the system; that is, one fault arrival causes system failure. When a system is not designed properly and is vulnerable to single-point failures, the reliability can be seriously degraded. To understand the effects of a single-point failure, consider the model in figure 23 of a TMR system with a single-point failure. The parameter C represents the fraction of faults that do not cause system failure alone. The sensitivity of the system reliability to C can be seen in the following SURE run.

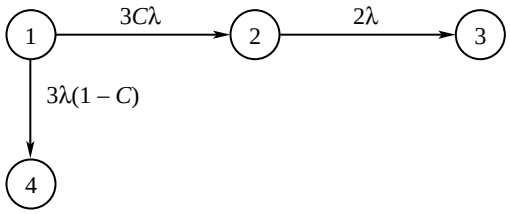


Figure 23. Model of TMR system with single-point failure.

```
$ sure
  SURE V7.4 NASA Langley Research Center
  1? read spf
  2: LAMBDA = 1E-4;
  3: C = .9 TO 1 BY 0.01;
  4: 1,2 = 3*LAMBDA*C;
  5: 2,3 = 2*LAMBDA;
  6: 1,4 = 3*(1-C)*LAMBDA;
       0.05 SECS. TO READ MODEL FILE
  7? run
```

```
MODEL FILE = spf.mod                    SURE V7.4 24 Jan 90 10:20:43
     C          LOWERBOUND    UPPERBOUND    COMMENTS              RUN #1
-----------  -----------  -----------  --------------------------
9.00000e-01  3.02245e-04  3.02700e-04
9.10000e-01  2.72320e-04  2.72730e-04
9.20000e-01  2.42395e-04  2.42760e-04
9.30000e-01  2.12470e-04  2.12790e-04
9.40000e-01  1.82545e-04  1.82820e-04
9.50000e-01  1.52620e-04  1.52850e-04
9.60000e-01  1.22695e-04  1.22880e-04
9.70000e-01  9.27702e-05  9.29100e-05
9.80000e-01  6.28451e-05  6.29400e-05
9.90000e-01  3.29200e-05  3.29700e-05
1.00000e+00  2.99500e-06  3.00000e-06

2 PATH(S) TO DEATH STATES
0.667 SECS. CPU TIME UTILIZED
 8? exit
```

The results of this run are plotted in figure 24. The plot reveals that probability of system failure is very sensitive to C. The sensitivity is even greater as the number of processors is increased in the NMR system. To have a probability of failure less than 10^{-9} in a 5MR system composed of processors whose failure rate is very low (10^{-5}/hr), C must be greater than 0.999998. A 5MR system that is not subject to single-point failure has a probability of failure of 1×10^{-11}. See figure 25, which was produced by solving the following model:

```
LAMBDA = 1E-5;
N = 5;
X = 1E-10 TO* 1 BY 2;
C = 1-X;
1,2 = 5*C*LAMBDA;
1,7 = 5*(1-C)*LAMBDA;
2,3 = 4*C*LAMBDA;
2,8 = 4*(1-C)*LAMBDA;
3,4 = 3*LAMBDA;
```

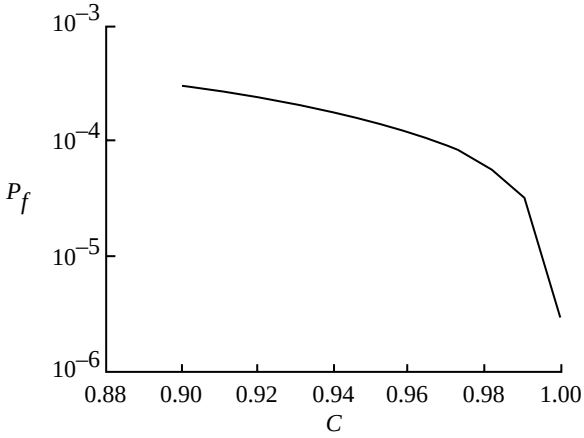


Figure 24. Failure probability as function of C.

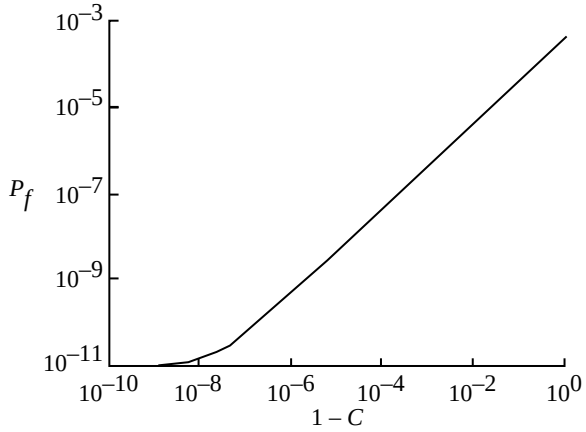


Figure 25. Failure probability of 5MR with $\lambda = 10^{-5}$ as function of C.

Experimental measurement of a parameter to such accuracy is easily shown to be impractical. Alternatively, a system with no single-point failures can be designed, and thus, remove this transition from the model.

6.3. Fail-Stop Dual System

When processors fail, they often either fail to produce an output (i.e., apparently halt) or produce an incorrect answer that is so far from the correct answer that it would fail a simple reasonableness check. In both cases, it is simple for a processor to detect its own failure with special circuitry and to halt its processing. A processor with this capability is often referred to as a fail-stop processor. Electronic circuitry that can recognize that one fail-stop processor has halted (e.g., no data arrives) and automatically switch to an alternate fail-stop processor is simple to build. A system consisting of two fail-stop processors and this selection circuitry is called a dual system. This system is illustrated in figure 26.

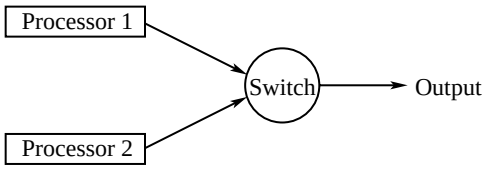


Figure 26. Fail-stop dual system.

The switch determines which output will be used. Reliability engineers sometimes make the mistake of assuming that this process will work correctly 100 percent of the time. However, most fail-stop processors cannot be guaranteed to always halt upon failure. For example, the failure can cause an erroneous answer that passes the reasonableness checks, or the failure can affect the ability of the processor to detect the failure or to halt its processing or its output. In this section, the impact on system reliability when a processor is not 100-percent fail stop will be investigated. Suppose PFS is the probability that a processor stops when it fails. The model in figure 27 describes such a system.

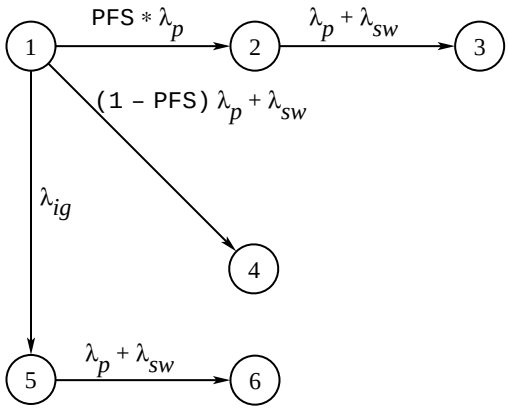


Figure 27. Model of fail-stop dual system.

The system begins in state (1) where all components are operational. Either of two processors or the switch itself could fail. Use λ_p for the failure rate of the currently selected processor, λ_{ig} for the processor that is currently being ignored, and λ_{sw} for the failure rate of the switch. Usually λ_{ig} will be equal to λ_p; however, λ_{ig} is used here to make the model clearer. The transition from state (1) to state (2)

represents the failure of the processor that causes it to stop. Because PFS is the fraction of times this occurs, the rate of this transition is (PFS)λ_p. The cases where it does not fail stop lead to the failure state (4). The rate of such failures is (1 - PFS)λ_p. The failure of the switch could also cause the system to fail, so the total rate from state (1) to state (4) is (1 - PFS)λ_p + λ_{sw}. The failure of the currently unselected processor carries the system to state (5) with a rate of λ_{ig}.

Note that the sum of the rates of all failure transitions from state (1) add up to the sum of the failure rates of all nonfailed components (λ_p + λ_{ig} + λ_{sw}). This property should always be true for all operational states of a reliability model. State (2) represents the situation where the selected processor has halted. Because the switch has not failed in this state, the system is able to make the switch to the alternate. The transition from state (2) to state (3) represents the failure of the switch or the last processor. State (3) is thus a death state. State (5) represents the state where the selected processor is still operating correctly, but the unselected processor has failed. Thus, when the active processor fails (state (5) to state (6)), the system has no working spare. Note also that the failure of the switch always leads to a death state. Furthermore, every operational state is subject to this failure. Thus, every operational state should be inspected to verify that a transition representing the failure of the switch leaving it occurs.

The plot of the SURE solution of this model is shown in figure 28. The statement PFS = 0 TO 1 BY 0.01 directs the SURE program to compute the probability of system failure as a function of PFS. The program solves the model for values of PFS over the range from 0 to 1 in increments of 0.01. The reliability of the system is very sensitive to the probability of the fail-stop processor halting upon failure and PFS must be much greater than 0.9 to have a significant improvement in reliability over a simplex computer.

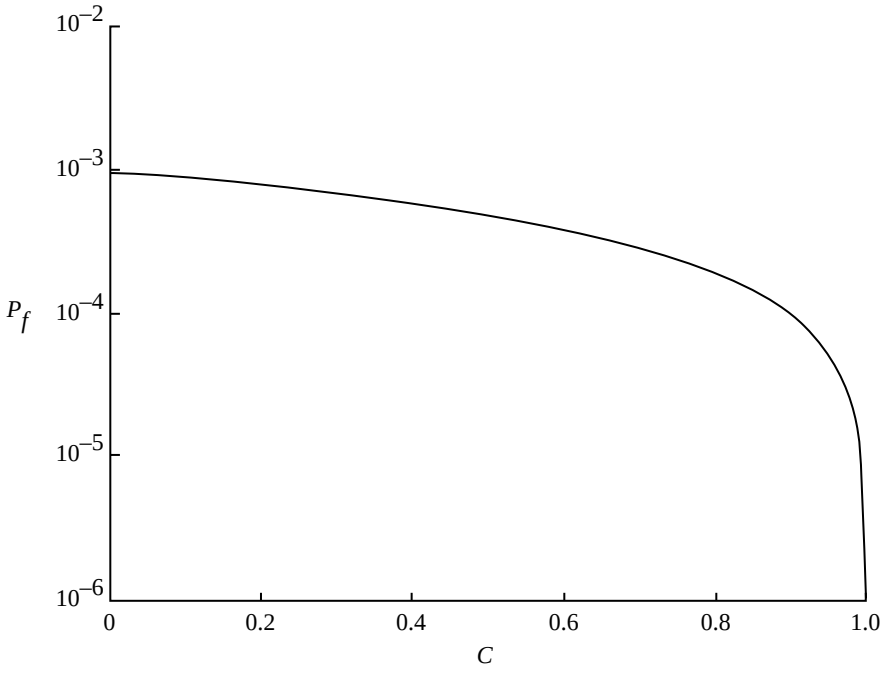


Figure 28. Plot of fail-stop dual system unreliability and PFS.

6.4. Self-Checking Pair Architecture

The previous section illustrated the sensitivity of reliability to the assumption of fail stop. Consequently, approaches have been sought to achieve the fail-stop assumption with almost 100-percent certainty. One such approach is a self-checking pair architecture. In this system, four computers are

configured into two self-checking pairs. The self-checking pairs run in lock step mode. Upon any disagreement of the outputs, the self-checking pair shuts itself down. Of course, special circuitry must be used to perform the self-checking function, but techniques exist that can make such circuitry fail-safe. That is, if the self checker fails, then the pair is shut down. The outputs of the two self-checking pairs are sent to a selection switch. The self-checking pair serves as the fail-stop processor of the model in figure 27. In such a system, it is not unreasonable to assume that the probability PFS, which the self-checking pair does not stop, although it has failed, is the probability that both processors fail concurrently before the selection switch disconnects the pair. Clearly, such a probability is small but not zero. This probability is intimately connected with fault latency and failure correlation, which will be further investigated in later models. The model of section 6.3 applies to the system with two simple modifications

1. The PFS probability is very small.

2. The failure rate of the self-checking pair would likely be $2\lambda_p$.

6.5. Degradable Quadraplex With Partial Fail Stop or Self Test

The question is often asked whether it is preferable to degrade a triad into a simplex or into a dual. If a system degrades to a dual, the problem of determining which processor has failed must be addressed. If the best that can be done is guess with probability of 0.5 percent of success, the probability of system failure is exactly the same as degrading to a simplex at the first failure. However, if the probability of success in detecting the failed processor in the dual can be improved, then the system reliability can be improved. One method of improvement is to take advantage of the fact that many failures cause a processor to halt, which was done in the model in figure 27. Studies by McGough and Swern have shown that typically 90 percent of CPU faults result in a processor halting (ref. 15). Although this method is far from fail stop, this aspect of system failure can be utilized in the system design to increase the reliability of a quadraplex system. The majority-voting system must be designed to recognize the absence of data. The details of such a voter will not be discussed here, but such a design is easily accomplished. In the model shown in figure 29, PFS is the probability that a fault causes the processor to halt. The SURE input file is

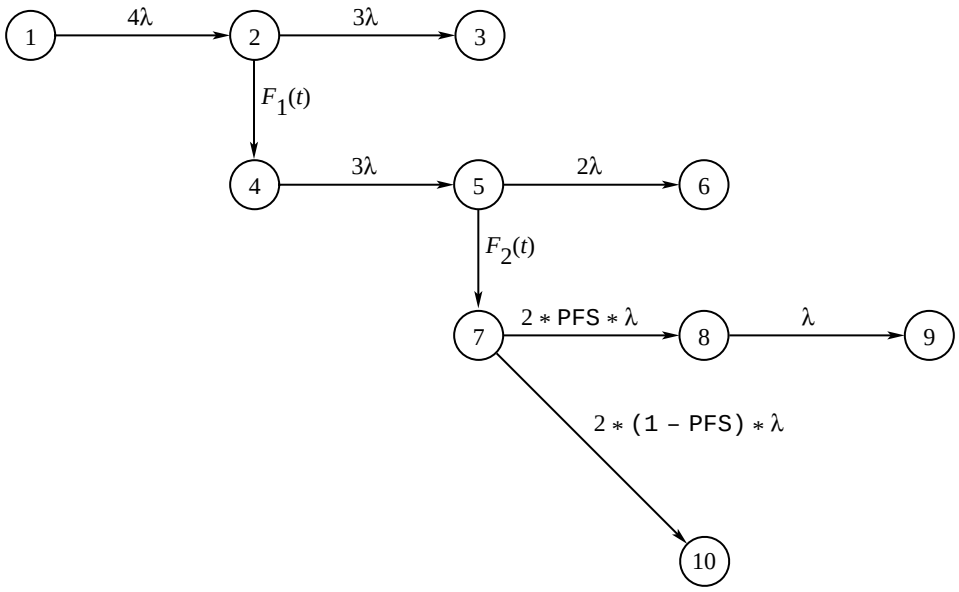


Figure 29. Degradable quadraplex with partial fail-stop.

```
LAMBDA = 1e-4;           (* Failure rate of processor *)
MEANREC = 1e-2;          (* Mean reconfiguration time *)
STDREC = 1e-3;           (* Standard deviation of " " *)
MEANREC2 = 1.2e-2;       (* Mean reconfiguration time *)
STDREC2 = 1.4e-3;        (* Standard deviation of " " *)
PFS = 0 to 1 by .1;      (* Prob. fault halts processor *)

1,2 = 4*LAMBDA;
2,3 = 3*LAMBDA;
2,4 = <MEANREC,STDREC>;
4,5 = 3*LAMBDA;
5,6 = 2*LAMBDA;
5,7 = <MEANREC2,STDREC2>;
7,8 = 2*PFS*LAMBDA;
8,9 = LAMBDA;
7,10 = 2*(1-PFS)*LAMBDA;
```

Because the fail-stop capability is not used until the configuration has been reduced to two processors, it is most effective for long mission times. The result of a SURE run with mission time of 1000 hr is shown in figure 30.

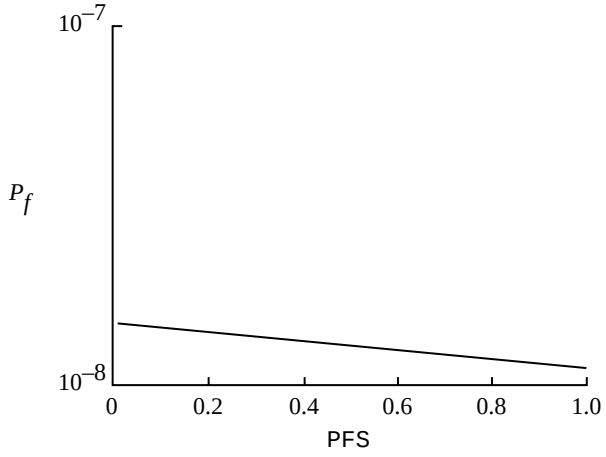


Figure 30. Failure probability of degradable quadraplex with partial fail stop.

Another approach is to use a self-test program to diagnose the faulty processor in the dual. This system is modeled in the same manner. In this case, PFS is the probability that the self-test program correctly diagnoses the faulty processor and the system successfully reconfigures.

6.6. Incomplete Reconfiguration

Suppose a system is designed with incomplete detection of faults. In other words, some faults cannot be detected by the system. Of course, this does not lead to immediate system failure. The good processors will still out vote the bad processor. However, the reconfigurable system will behave like a nonreconfigurable system in the presence of these faults. Assume that the fraction of faults that are detectable D is known. This situation is sometimes referred to as coverage. Because many other definitions of this term exist, it will not be used in this paper. The SURE model is

```
LAMBDA = 1e-4;                (* Failure rate of processor *)
MEAN = 1e-2;                  (* Mean reconfiguration time *)
STD = 1e-3;                   (* Standard deviation of " " *)
D = 0 TO+ 1 by 0.05;

1,2 = 4*LAMBDA*D;
1,12 = 4*LAMBDA*(1-D);
2,3 = 3*LAMBDA;
2,4 = <MEAN,STD>;
4,5 = 3*LAMBDA*D;
4,15 = 3*LAMBDA*(1-D);
5,6 = 2*LAMBDA;
5,7 = <MEAN,STD>;
7,8 = LAMBDA;

12,13 = 3*LAMBDA;
15,16 = 2*LAMBDA;
```

The technique decomposes the fault-arrival process into two transitions. For example, if the total failure rate out of state is 4λ, then two transitions are produced. One transition has a rate of $4D\lambda$ and the other has $4(1 - D)\lambda$. This model is shown in figure 31. The probability of failure as a function of D is shown in figure 32.

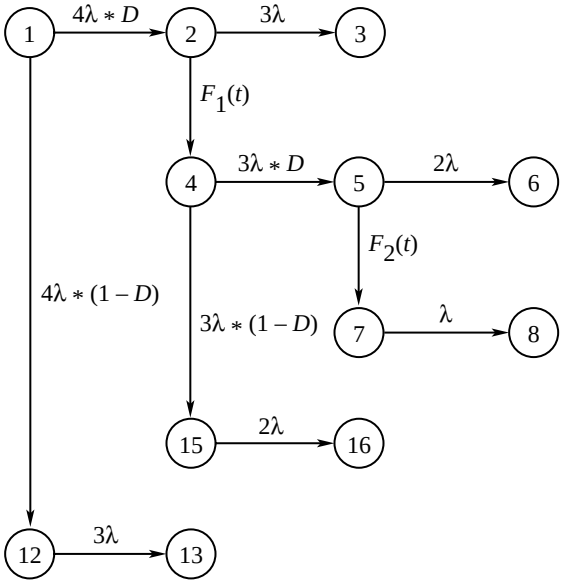


Figure 31. Degradable quadraplex with incomplete reconfiguration.

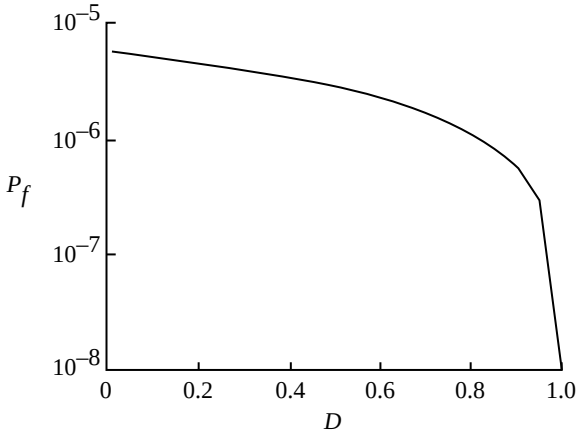


Figure 32. Plot of degradable quadraplex with incomplete reconfiguration.

7. Reconfiguration By Sparing

Three categories of spares will be defined: cold spares, warm spares, and hot spares. Some systems are designed with spares that are unpowered until brought into the active configuration. This approach is used because unpowered (cold) spares usually have a lower failure rate than powered (hot) spares. If the failure rate of the inactive spare is the same as an active processor, it is a hot spare. If the failure rate of an inactive spare is zero, then it is a cold spare. If the failure rate is somewhere in between zero and the active processor rate, it is a warm spare. If λ_s is the failure rate of an inactive spare and λ_p is the failure

rate of an active processor, then the cold spare is $\lambda_s = 0$, the warm spare is $0 < \lambda_s < \lambda_p$, and the hot spare is $\lambda_s = \lambda_p$.

The disadvantage of an unpowered spare (whether cold or warm) is that it must be initialized during reconfiguration, whereas a hot spare can be maintained with memory already loaded. This situation can lead to a longer reconfiguration time. Thus, the model parameter values will differ with the strategy used. Some reliability programs, such as CARE III, explicitly assume that the spares are hot (ref. 16).

7.1. Triad With Two Cold Spares

In this model, a new form of reconfiguration is investigated. Instead of degrading the configuration upon detection of a faulty processor, a spare processor is brought into the configuration to replace the faulty one. For simplicity, in this model it is assumed that the spares do not fail while not in the active configuration. The issues associated with failing spares will be considered in sections 7.2, 9.2, 9.3, and 9.4.

In the model shown in figure 33, the reconfiguration process is assumed to be described by distribution $F(t)$, which is assumed to be independent of the system state. The SURE input is

```
LAMBDA = 1e-4;          (* Failure rate of processor *)
MEANREC = 1e-2;         (* Mean reconfiguration time *)
STDREC = 1e-3;          (* Standard deviation of reconfig. time *)

1,2 = 3*LAMBDA;
2,3 = 2*LAMBDA;
2,4 = <MEANREC,STDREC>;
4,5 = 3*LAMBDA;
5,6 = 2*LAMBDA;
5,7 = <MEANREC,STDREC>;
7,8 = 3*LAMBDA;
8,9 = 2*LAMBDA;
```

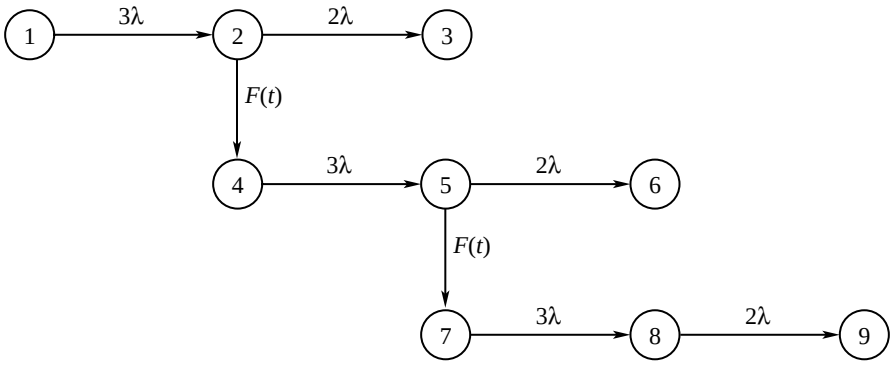


Figure 33. Model of triplex with two cold spares.

State (1) of this model represents the initial system with three active processors and two spare processors. The system is in state (2) when one of the three active processors has failed. Two transitions leave state (2). One transition is near-coincident failure of one of the two remaining active processors. The second transition is the replacement of the failed active processor with a spare. In state (4), the system consists of three active processors plus one remaining cold spare. Once a cold spare processor is brought into the active configuration, it has the same failure rate as the other active processors. Thus,

the transition from state (4) to state (5) has rate 3*λ. State (5) has the same transitions leaving it as state (2). Once the system reaches state (7), no cold spare processors remain.

7.2. Triad With Two Warm Spares

If the system is assumed to have perfect detection of failed spare processors, the model developed in section 7.1 can be easily modified to include spare failures. As shown in figure 34, this modification consists of the addition of two transitions. The transition from state (1) to state (4) represents the failure of one of the two spare processors before either of them is brought into the active configuration. The rate for this transition is 2*γ, where γ is the failure rate for a warm spare. The transition from state (4) to state (7) represents the failure of the remaining spare processor after the first spare processor has either failed or been brought into the active configuration to replace a failed active processor. The SURE input is

```
LAMBDA = 1e-4;          (* Failure rate of active processor *)
GAMMA = 1e-5;           (* Failure rate of warm spare processor *)
MEANREC = 1e-2;         (* Mean reconfiguration time *)
STDREC = 1e-3;          (* Standard deviation of reconfig. time *)

1,2 = 3*LAMBDA;
1,4 = 2*GAMMA;
2,3 = 2*LAMBDA;
2,4 = <MEANREC,STDREC>;
2,5 = 2*GAMMA;
4,5 = 3*LAMBDA;
4,7 = GAMMA;
5,6 = 2*LAMBDA;
5,7 = <MEANREC,STDREC>;
5,8 = GAMMA;
7,8 = 3*LAMBDA;
8,9 = 2*LAMBDA;
```

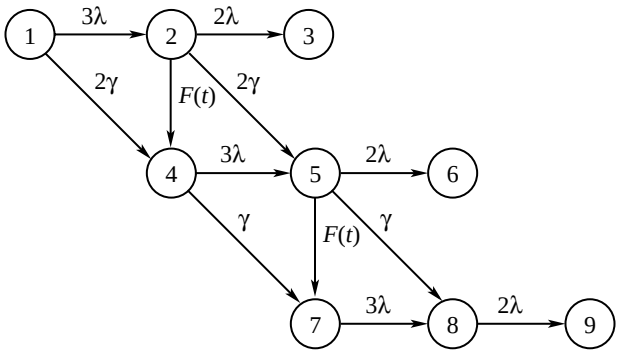


Figure 34. Model of triplex with two warm spares.

The probability of failure as a function of the spare failure rate is plotted in figure 35 for three mission times: 10, 100, and 1000 hr. The same model can be used to analyze a system with hot spares by changing the spare failure rate to equal the active processor failure rate.

In this section, it was assumed that all failed spare processors are detected by the system and no state-dependent recovery rates exist in the model. These assumptions significantly simplified the

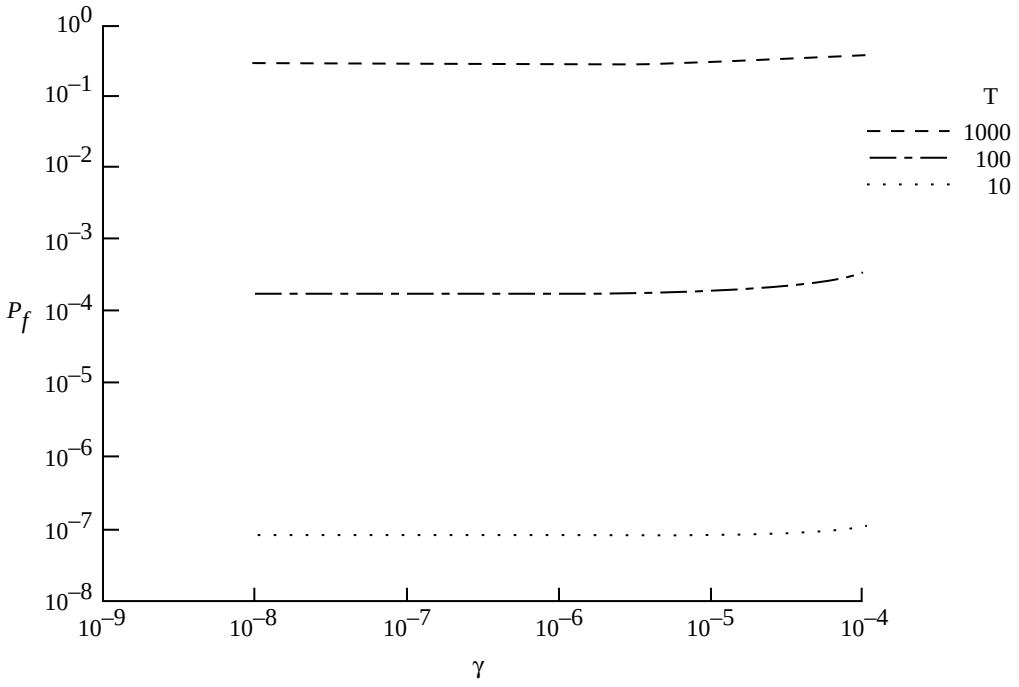


Figure 35. Failure probability of triplex with two warm spares.

reliability models. In section 9, systems will be modeled without making these simplifying assumptions and more complex systems will be investigated. These systems will use different reconfiguration, and consist of several subsystems. As complexity is added to a system, it quickly becomes impractical to enumerate all states and transitions of a model by hand. In section 8, a simple but expressive language is introduced for specifying Markov or semi-Markov models. This language serves as the input language for the ASSIST computer program that automatically generates the states and the transitions of the model. The output of the ASSIST program can be directly processed by the SURE program.

8. The ASSIST Model Specification Language

A computer program was developed at Langley Research Center to automatically generate semi-Markov models from an abstract, high-level language. This program, named the Abstract Semi-Markov Specification Interface to the SURE Tool (ASSIST), is written in the C programming language and runs on the VMS and the Unix operating systems (refs. 17 and 18). The ASSIST program generates a file containing the generated semi-Markov model in the format needed for input to a number of Markov or semi-Markov reliability analysis programs developed at Langley, such as SURE or PAWS.

The abstract language used for input to ASSIST is described in this section. Only the features of the language necessary for understanding the models in this paper are presented. For a description of the complete input language or for more detailed information about ASSIST, see reference 17. Readers already familiar with the ASSIST language can proceed to section 9.

The ASSIST program is based on concepts used in the design of compilers. The ASSIST input language is used to define rules for generating a model. These rules are first applied to a start state. The rules create transitions from the start state to new states. The program then applies the rules to the newly created states. This process is continued until all states are either death states or have already been processed. The rules in the ASSIST language describe the failure and the recovery processes of the system. Often even the most complex characteristics of a system can be described by relatively simple rules. The models only become complex when these few rules combine many times to form models with large

numbers of states and transitions between them. The abstract description is useful for communicating and validating the system model, as well as for model generation. Also, the process of describing a system in this abstract language forces the reliability engineer to clearly understand the fault tolerance strategies of the system.

The ASSIST input language can be used to describe any state-space model. Its full generality makes it useful for specifying Markov and semi-Markov models, even when it is not necessary to generate the model. The ASSIST language can serve as a convenient vehicle for discussing and analyzing complex state-space models without having to specify all states and transitions of the model by enumeration.

8.1. Abstract Language Syntax

A formal description of this language is not presented. Nevertheless, a few conventions must be defined to facilitate description of the language:

1. All reserved words will be capitalized in typewriter print.
2. Lowercase words that are in italics indicate items that are to be replaced by something defined elsewhere.
3. Items enclosed in double square brackets ⟦ ⟧ can be omitted.
4. Items enclosed in braces { } can be omitted or repeated as many times as desired.

The basic language consists of seven statements:

1. The constant-definition statement
2. The `SPACE` statement
3. The `IMPLICIT` statement
4. The `START` statement
5. The `DEATHIF` statement
6. The `PRUNEIF` statement
7. The `TRANTO` statement

Each statement is discussed in the following sections.

8.1.1. Constant-definition statement. A constant-definition statement equates an identifier consisting of letters and digits to a number. For example

```
LAMBDA = 0.0052;
RECOVER = 0.005;
```

Once defined, an identifier can be used instead of the number it represents. In the following sections, the phrase "*const*" is used to represent a constant, which can be either a number or a constant identifier. Constants can also be defined in terms of previously defined constants

```
LAMBDA = 1E-4;
GAMMA = 10*LAMBDA;
```

In general the syntax is

ident = *expression*;

where *expression* is a legal FORTRAN/Pascal expression. Both ( ) and [ ] can be used for grouping in the expressions. The following statements contain legal expressions

```
ALPHA = 1E-4;
```

```
RECV = 1.2*EXP(-3*ALPHA);
DELTA = 1.2*[(ALPHA + 2.3E-5)*RECV + 1/ALPHA];
```

All constant definitions are printed in the SURE model file so that the definitions may be used by the SURE program. In addition, any statements in the ASSIST input file that are enclosed within double quotes are copied directly into the SURE model file and are not otherwise processed by the ASSIST program. For example, if a user wished to be prompted for the value of γ by the SURE program instead of by the ASSIST program and to see the effects of varying the value of λ exponentially, the following statements could be included in the ASSIST input file:

```
"INPUT GAMMA;"
"LAMBDA = 1E-4 TO* 1E-9;"
```

State-space variables may not be used in constant-definition expressions because the variables do not remain constant throughout model generation. An expression containing state-space variables can be equated to an identifier by using the `IMPLICIT` statement, which are described in section 8.1.3.

8.1.2. SPACE statement. This statement is used to specify the state space on which the Markov model is defined. Essentially, the state space is defined by an *n*-dimensional vector with each component of the vector defined as an attribute of the system being modeled. In the SIFT-like architecture example shown in figure 22, the state space is `(NC,NF)`. This would be defined in the abstract language as

```
SPACE = (NC: 0..6, NF: 0..6);
```

The 0 . . 6 represents the range of values over which the components can vary. The lower bound of the range must be greater than or equal to zero. The upper bound must be greater than the lower bound and less than or equal to 255. This maximum upper bound value can be easily changed by modifying a constant and recompiling the ASSIST program. The number of components (i.e., the dimension of the vector space) can be as large as desired. In general the syntax is

`SPACE = (` *ident*⟦`:` *const* . . *const*⟧ , *ident*⟦`:` *const* . . *const*⟧ `);`

The range specification is optional and defaults to a range from 0 to 255. The identifiers *ident* that are used in the `SPACE` statement are referred to as the state-space variables.

8.1.3. IMPLICIT statement. The `IMPLICIT` statement is used to define a quantity that is not in the state space itself, but is a function of the state space. Earlier versions of ASSIST did not have the `IMPLICIT` statement and allowed the user to equate an identifier to an expression containing state-space variables by using the same syntax as the constant-definition statement. The value of the implicit function is based upon constants and state-space variables.

For example, if `NWP` is a state-space variable representing the number of working processors and `NI` is a constant denoting the number of processors initially, then the declaration

```
IMPLICIT NFP[NWP] = NI - NWP;
```

defines `NFP`, which denotes the number of failed processors. The number of failed processors is defined to be the difference between the initial number and the current number of working processors. The implicit function can be referenced as illustrated in the following `DEATHIF` statement:

```
DEATHIF NWP <= NFP;
```

The `IMPLICIT` statement equates an identifier to an expression containing variables. Every variable used in the expression must be spelled out in either the state-space variable list or the optional parameter list. In general the syntax is

`IMPLICIT` *ident*[*state-space-variable-list*] ⟦(*parameter-list*)⟧ `=` *expression* `;`

The statement *state-space-variable-list* is made up of one or more state-space-variable identifiers separated by commas. The identifiers must already have been defined in a SPACE statement. All state-space variables that are referenced in the expression of an IMPLICIT statement must be listed in the state-space-variable list.

The optional *parameter-list* is used to declare an implicit function that is also a function of specified parameters. Any variables, such as FOR index variables, that are referenced in the expression of an IMPLICIT and are not state-space variables must be listed in the optional parameter list.

The *expression* of the IMPLICIT definition is the expression defining the value as a function of the specified parameters and the state-space variables.

The implicit function may be invoked in TRANTO, DEATHIF, PRUNEIF, or FOR statements or in later IMPLICIT definitions by giving its name followed by the values for each parameter in parentheses. A passed value can be a number, a named constant, a variable, or an expression. If the IMPLICIT definition does not include a parameter list, the IMPLICIT is invoked by its name alone.

8.1.4. START statement. This statement indicates the state from which the ASSIST program will initiate the recursive model generation. This state usually corresponds to the initial state of the system that is being modeled. That is, the probability the system is in this state at time 0 = 1. In the SIFT-like architecture example shown in figure 22, the initial state is (6,0). This initial state is specified in the abstract language by

```
START = (6,0);
```

In general the syntax is

START = (*const* {, *const* });

The dimension of the vector must be the same as in the SPACE statement.

8.1.5. DEATHIF statement. The DEATHIF statement specifies which states are death states, that is, absorbing states in the model. The following is an example in the space (DIM1: 2..4, DIM2: 3..5)

```
DEATHIF (DIM1 = 4) OR (DIM2 = 3);
```

This statement defines states (4,3), (4,4), (4,5), (2,3), and (3,3) as death states. In general, the syntax is

DEATHIF *expression*;

The expression in this statement must be a Boolean expression. A Boolean expression may use the logical operators AND, OR, and NOT.

8.1.6. TRANTO statement. This statement is the most important statement in the language. It is used to describe, and consequently, generate the model in a recursive manner. The following statement generates all fault-arrival transitions in the model shown in figure 22:

```
IF NC > NF TRANTO (NC, NF+1) BY (NC-NF)*LAMBDA;
```

The simplest syntax for a TRANTO statement is

IF *expression* TRANTO *destination* BY *expression*;

The first expression following the IF must be Boolean. Conceptually, this expression determines whether this rule applies to a particular state. For example, in the state-space expression SPACE = (A1: 1..5, A2: 0..1), the expression (A1 > 3) AND (A2 = 0) is true for states (4,0) and (5,0) only.

The *destination* vector following the TRANTO reserved word defines the destination state of the transition to be added to the model. The destination state can be specified by using positional or assigned values.

The syntax for specification of the destination by positional values is as follows:

(*expression*, {, *expression*})

where the listed expressions define each state-space variable value for the destination state. An expression must be included for every state-space variable defined in the SPACE statement, which includes every array element. Each expression within the parentheses must evaluate to an integer. For example, if the state space is (X1, X2) and the source state is (5,3), then the vector (X1+1, X2-1) refers to (6,2).

The syntax for specification of the destination by assigned values is

ident = *expression* {, *ident* = *expression* }

where *ident* is a state-space variable and *expression* is an integer expression. The assignments define the destination state of a transition by specifying the change in one or more state-space variable values from the source state to the destination state. There can be as many assignments as state-space variables. State-space variables that do not change need not be specified. The two syntaxes cannot be mixed in the same statement and the *destination* cannot be within parentheses when assigned values are to be used.

The expression following BY indicates the rate of the transition to be added to the model. This expression must define a real number. The user may include constant names in the rate statement that are not defined in the ASSIST file. These names are simply copied into the rate expressions in the model file to be defined during execution of the SURE program. The ASSIST program also allows the user to concatenate identifiers or values in the rate expression by using the ^ character. The use of this feature is demonstrated in section 13.5.

The condition expression of the TRANTO statement can be nested as follows:

IF *expression* THEN

trantos

⟦ ELSE

trantos ⟧

ENDIF;

where *trantos* is one or more TRANTO statements or TRANTO clauses and where a TRANTO statement is

IF *expression* TRANTO *destination* BY *expression*;

and a TRANTO clause is TRANTO *destination* BY *expression*;

A TRANTO clause may not appear by itself without a condition expression. If the IF is not followed by a THEN, then only one TRANTO clause may be included and no ELSE clause or ENDIF may be used. If the IF is followed by a THEN, then an optional ELSE clause may be included and the IF statement must be terminated with an ENDIF. The THEN clause and the optional ELSE clause may contain multiple TRANTO statements. Every rate expression must be followed by a semicolon and the end of the entire nested statement must be followed with a semicolon.

State-space variables may be used in any of the expressions of the TRANTO statement. The value of a state-space variable is the corresponding value in the source state to which the TRANTO statement is being applied. For example, if the TRANTO statement is being applied to state (4,5) and the state space was defined by SPACE = (A: 0..10, Z: 2..15), then A = 4 and Z = 5.

8.1.7. Model generation algorithm. The ASSIST program generates the model in accordance with the following algorithm:

Initialize READY-SET to contain the start state only.

WHILE READY-SET is not empty DO

Select and remove a state from READY-SET

IF the selected state does not satisfy a DEATHIF or PRUNEIF statement THEN

Apply each TRANTO rule to the selected state as follows:

IF the TRANTO if expression evaluates to TRUE THEN

Add the transition to the model.

IF the destination state is new, add it to the READY-SET

ENDIF

ENDIF

ENDWHILE

The ASSIST program builds the model from the start state by recursively applying the transition rules. A list of states to be processed, the READY-SET, begins with only the start state. Before application of a rule, ASSIST checks all death conditions to determine whether the current state is a death state. Because a death state denotes system failure, no transitions can leave a death state. Each TRANTO rule is then evaluated for the nondeath state. If the condition expression of the TRANTO rule evaluates to true for the current state, then the destination expression is used to determine the state-space variable values of the destination state. If the destination state has not already been defined in the model, then the new state is added to the READY-SET of states to be processed. The rate of the transition is determined from the rate expression and the transition description is printed to the model file. When all TRANTO rules have been applied to it, the state is removed from the READY-SET. When the READY-SET is empty, then all possible paths terminate in death states and model building is complete.

8.2. Illustrative Example of SIFT-Like Architecture

Now the model shown in figure 22 can be specified in the ASSIST language

```
NP = 6;                (* Number of processors initially *)
LAMBDA = 1E-4;         (* Fault arrival rate *)
DELTA = 3.6E3;         (* Recovery rate *)

SPACE = (NC: 0..NP,    (* Number processors in configuration *)
         NF: 0..NP);   (* Number faulty processors *)

START = (NP,0);

IF NC > NF TRANTO (NC,NF+1) BY (NC-NF)*LAMBDA; (* Fault arrivals *)
IF NF > 0 TRANTO (NC-1,NF-1) BY FAST NF*DELTA; (* System recovery *)

DEATHIF 2*NF >= NC;    (* System failure if majority not working *)
```

The first three lines equate the identifiers NP, LAMBDA, and DELTA to specific values. The next two lines define the state space with the SPACE statement. For this system, two attributes suffice to define the state of the system. The first attribute NC is the number of processors currently in the configuration. The second attribute NF is the number of faulty processors in the configuration.

The SPACE statement declares that the state space is two-dimensional. The first dimension is named NC and has domain 0 to NP. The second dimension is NF and has domain 0 to NP. The START

statement declares that the construction of the model will begin with the state (NP,0). The next two TRANTO statements define the rules for building the model. Informally these rules are

1. Every working processor in the current configuration fails at rate LAMBDA.

2. The system removes faulty processors at rate DELTA.

These informal rules are easily converted into ASSIST statements. The phrase "Every working processor in the current configuration" becomes

```
IF NC > NF
```

Note that if NC is greater than NF, at least one working processor remains, so a transition should be created. The word "fails" is captured by

```
TRANTO NF = NF + 1
```

This statement says that the destination state is obtained from the current state by incrementing the NF component by one. The phrase "at rate LAMBDA" is captured by BY (NC-NF)*LAMBDA.

This statement declares that the rate of the generated transition is (NC-NF)*LAMBDA. The identifier LAMBDA, which represents the failure rate, is multiplied by (NC-NF) because any of the working processors can fail. Each processor fails at rate LAMBDA. Therefore, the rate that any processor fails is

```
(NC-NF)*LAMBDA
```

The second rule is translated into ASSIST syntax in a similar manner. A faulty processor NF > 0 is removed (i.e., NC = NC - 1 and NF = NF - 1) at rate DELTA (total rate is NF*DELTA). Therefore, the recovery rule is

```
IF NF > 0 TRANTO (NC-1,NF-1) BY FAST NF*DELTA; (* system recovery *)
```

The keyword FAST alerts the SURE program that this transition is a fast recovery and not a failure. The SURE program assumes that the transition is exponentially distributed with rate NF*DELTA and automatically calculates the mean and the standard deviation. When experimental data are not available, it is usually convenient to assume that the recoveries are exponential. Later, after the recovery time distribution has been measured, the actual means and standard deviations may be used. When multiple simultaneous recoveries occur, that is NF > 1, it is also convenient to assume that the recoveries are independent. The rate of recovery of the first of two or more competing exponential recoveries can be obtained by adding their rates. A brief derivation of this result is given in section 3.1 For a more detailed treatment of competing recoveries, consult a standard text that includes a discussion of order statistics, such as reference 19. In this case, because all recoveries have the same rate, the rate is NF*DELTA.

The DEATHIF statement defines the system failure states. Informally, if two times the number of faulty processors is greater than or equal to the number of processors in the configuration, the system fails. This is translated into

```
DEATHIF 2*NF >= NC;
```

9. Reconfigurable Triad Systems

In this section, systems that use both sparing and degradation to accomplish reconfiguration will be explored. In the first example, a triad with cold spares, the reconfiguration process changes when the supply of spares is exhausted. The later examples add detail and more closely capture the behavior of the spares. The models in this section demonstrate the flexibility of the semi-Markov modeling approach.

9.1. Triad With Cold Spares

A system consisting of a triad with a set of cold spares (spares that do not fail while inactive) will be explored. The number of initial spares is defined by using a constant, NSI. This definition allows the initial number of spares to be changed by altering only one line of the ASSIST input. (Although only one line changes in the input file, the size of the model generated varies significantly as a function of this parameter.) For simplicity, it is assumed in this section that spares do not fail until they become active. The system replaces failed processors with spares until all spares are depleted. Then the system degrades to a simplex.

```
NSI = 3;              (* Number of spares initially *)
LAMBDA = 1E-4;       (* Failure rate of active processors *)
MU = 7.9E-5;          (* Mean time to replace with spare *)
SIGMA = 2.56E-5;     (* Stan. dev. of time to replace with spare *)

MU_DEG = 6.3E-5;      (* Mean time to degrade to simplex *)
SIGMA_DEG = 1.74E-5; (* Stan. dev. of time to degrade to simplex *)

SPACE = (NW: 0..3,    (* Number of working processors *)
NF: 0..3,             (* Number of failed active processors *)
NS: 0..NSI);          (* Number of spares *)

START = (3,0,NSI);

IF NW > 0                    (* Processor failure *)
   TRANTO (NW-1,NF+1,NS) BY NW*LAMBDA;

IF (NF > 0) AND (NS > 0)   (* Non-failed spare becomes active *)
   TRANTO (NW+1,NF-1,NS-1) BY <MU,SIGMA>;

IF (NF > 0) AND (NS = 0)   (* No more spares, degrade to simplex *)
TRANTO (1,0,0) BY  <MU_DEG,SIGMA_DEG>;

DEATHIF NF >= NW;
```

The first statement defines a constant NSI, which represents the number of initial spares. The value of this constant can be changed to generate models for systems with various numbers of initial spares.

The next five statements define constants that are not used directly by ASSIST, but are passed along verbatim to SURE for computation purposes. The SPACE statement defines the domain of the state space. For this model, a three-dimensional space is needed. The components of the space are NW (number of working processors in the active configuration), NF (number of failed processors in the active configuration), and NS (number of spares available). The initial configuration is defined with the START statement (3,0,NSI), which indicates that NW = 3, NF = 0, and NS = NSI initially. The next three statements define the rules that are used to build the model. The first of these statements defines processor failure.

As long as working processors remain (NW > 0), the rule adds a transition. The destination state is derived from the source state according to the formula (NW-1, NF+1, NS). This statement is shorthand notation for NW = NW-1, NF = NF+1, NS = NS. The rate of the resulting transition is NW*LAMBDA. For example, if the current state were (2,1,3), this rule would generate a transition to (1,2,3) with rate 2*LAMBDA. The next rule only applies to states where (NF > 0) AND (NS > 0). That is, states with a failed processor and with available spares. The destination state is derived from the current state by the formula (NW+1, NF-1, NS-1). That is, the number of working processors NW is increased by one, the number of faulty processors NF is decremented and the number of spares NS is decremented. This rule corresponds to the replacement of a faulty processor with a spare.

The last TRANTO rule describes how the system degrades to a simplex. This degradation occurs when no spares are available and a processor has failed, that is, (NF > 0) AND (NS = 0). The transition is to the state (1,0,0). The transition occurs according to a distribution with mean MU_DEG and standard deviation SIGMA_DEG. This term is given in SURE notation

```
<MU_DEG, SIGMA_DEG>
```

Finally, the conditions defining the death states are given. The formula NF >= NW defines the states that are death states, that is, whenever the number of faulty processors are greater than or equal to the number of working processors.

Note that in this model, the method of counting is different from the model in section 8.2. Rather than counting the number of processors in the configuration and the number of faulty processors (NC and NF), the number of working processors and the number of faulty processors (NW and NF) are counted. The number of active processors can be obtained by adding the number of faulty processors to the number of working ones (NC = NF + NW). Thus, these methods are essentially the same and the choice between the two is merely a matter of preference. In this paper, both methods of counting will be used.

The following session was performed on this model and stored in file tpnfs.ast:

```
$ assist tpnfs
ASSIST VERSION 7.1                        NASA Langley Research Center
PARSING TIME = 0.17 sec.
generating SURE model file...
RULE GENERATION TIME = 0.01 sec.
NUMBER OF STATES IN MODEL = 10
NUMBER OF TRANSITIONS IN MODEL = 13
5 DEATH STATES AGGREGATED INTO STATE 1

$ sure

 SURE V7.9.8 NASA Langley Research Center

 1? read tpnfs

 2: NSI = 3;
 3: LAMBDA = 1E-4;
 4: MU = 7.9E-5;
 5: SIGMA = 2.56E-5;
 6: MU_DEG = 6.3E-5;
 7: SIGMA_DEG = 1.74E-5;

 8:
 9:
10:     2(* 3,0,3 *),     3(* 2,1,3 *)       = 3*LAMBDA;
11:     3(* 2,1,3 *),     4(* 3,0,2 *)       = <MU,SIGMA>;
12:     3(* 2,1,3 *),     1(* 1,2,3 DEATH *) = 2*LAMBDA;
13:     4(* 3,0,2 *),     5(* 2,1,2 *)       = 3*LAMBDA;
14:     5(* 2,1,2 *),     6(* 3,0,1 *)       = <MU,SIGMA>;
15:     5(* 2,1,2 *),     1(* 1,2,2 DEATH *) = 2*LAMBDA;
16:     6(* 3,0,1 *),     7(* 2,1,1 *)       = 3*LAMBDA;
17:     7(* 2,1,1 *),     8(* 3,0,0 *)       = <MU,SIGMA>;
18:     7(* 2,1,1 *),     1(* 1,2,1 DEATH *) = 2*LAMBDA;
19:     8(* 3,0,0 *),     9(* 2,1,0 *)       = 3*LAMBDA;
20:     9(* 2,1,0 *),    10(* 1,0,0 *)      = <MU_DEG,SIGMA_DEG>;
```

```
21:     9(* 2,1,0 *),     1(* 1,2,0 DEATH *) = 2*LAMBDA;
22:    10(* 1,0,0 *),     1(* 0,1,0 DEATH *) = 1*LAMBDA;
23:
24: (* NUMBER OF STATES IN MODEL = 10 *)
25: (* NUMBER OF TRANSITIONS IN MODEL = 13 *)
26: (* 5 DEATH STATES AGGREGATED INTO STATE 1 *)

      0.02 SECS. TO READ MODEL FILE

27? run

MODEL FILE = tpnfs.mod              SURE V7.9.8 11 Apr 94 10:26:41


             LOWERBOUND  UPPERBOUND  COMMENTS                RUN #1
----------- ----------- ----------- ------------------------------
            4.71208e-11 4.74718e-11

5 PATH(S) TO DEATH STATES
0.000 SECS. CPU TIME UTILIZED
28? exit
```

The value of `NSI` can be changed to model systems with different numbers of spare processors initially. As shown in table 1, changing this single value can have a significant effect on the size of the model generated.

Table 1. Model Sizes for Triad of Processors With Spares

Number of spares	Number of states	Number of transitions
0	6	4
1	9	7
2	12	10
3	15	13
10	36	34
100	306	304

9.2. Triad With Instantaneous Detection of Warm Spare Failure

This section builds on the model in section 9.1 by allowing the spare to fail. However, the model is still simplistic in that it assumes that the system always detects a failed spare. Thus, a failed spare is never brought into the active configuration:

```
NSI = 3;              (* number of spares initially *)

LAMBDA = 1E-4;        (* failure rate of active processors *)
GAMMA = 1E-6;         (* failure rate of spares *)
MU = 7.9E-5;          (* mean time to replace with spare *)
SIGMA = 2.56E-5;      (* stan. dev. of time to replace with spare *)
MU_DEG = 6.3E-5;      (* mean time to degrade to simplex *)
SIGMA_DEG = 1.74E-5; (* stan. dev. of time to degrade to simplex *)

SPACE = (NW: 0..3,    (* number of working processors *)
         NF: 0..3,    (* number of failed active processors *)
         NS: 0..NSI);(* number of spares *)
```

```
START = (3,0,NSI);

IF NW > 0                (* a processor can fail *)
   TRANTO (NW-1,NF+1,NS) BY NW*LAMBDA;

IF (NF > 0) AND (NS > 0) (* a spare becomes active *)
   TRANTO (NW+1,NF-1,NS-1) BY <MU,SIGMA>;

IF (NF > 0) AND (NS = 0) (* no more spares, degrade to simplex *)
   TRANTO (1,0,0) BY <MU_DEG,SIGMA_DEG>;

IF NS > 0                (* a spare fails and is detected *)
   TRANTO (NW,NF,NS-1) BY NS*GAMMA;

DEATHIF NF >= NW;
```

Because failed spares can never be brought into the active configuration, tracking of these spares once they fail is unnecessary. Thus, no state-space variable was defined to track the number of failed spares and the transition depicting a spare failing simply decrements the number of spare processors by one.

9.3. Degradable Triad With Nondetectable Spare Failure

In the previous models, it was assumed that spares do not fail while inactive or the spare failure was assumed to be immediately detected. These assumptions are clearly nonconservative. In this example, the other extreme will be investigated–not only can the spares fail, but the fault remains undetectable until brought into the active configuration. The model in this example utilizes a failure rate for the spares that is different from the rate for active processors.

```
NSI = 3;              (* number of spares initially *)
LAMBDA = 1E-4;        (* failure rate of active processors *)
GAMMA = 1E-6;         (* failure rate of spares *)
MU = 7.9E-5;          (* mean time to replace with spare *)
SIGMA = 2.56E-5;      (* stan. dev. of time to replace with spare *)

MU_DEG = 6.3E-5;      (* mean time to degrade to simplex *)
SIGMA_DEG = 1.74E-5;  (* stan. dev. of time to degrade to simplex *)
SPACE = (NW: 0..3,    (* number of working processors *)
         NF: 0..3,    (* number of failed active processors *)
        NWS: 0..NSI,  (* number of working spares *)
        NFS: 0..NSI); (* number of failed spares *)

START = (3,0,NSI,0);
IMPLICIT PRG[NWS,NFS]=NWS/(NWS+NFS);(*prob.of switchingin good spare *)
   (* processor failure *)
IF NW > 0 TRANTO (NW-1,NF+1,NWS,NFS) BY NW*LAMBDA;

IF (NF > 0) AND (NWS+NFS > 0) THEN (* reconfigure using a spare *)
      (* a good spare becomes active *)

   IF NWS > 0 TRANTO (NW+1,NF-1,NWS-1,NFS) BY <MU,SIGMA,PRG>;
      (* a failed spare becomes active *)
   IF NFS > 0 TRANTO (NW,NF,NWS,NFS-1) BY <MU,SIGMA,1-PRG>;
ENDIF;

IF (NF > 0) AND (NWS+NFS = 0) (* no more spares, degrade to simplex *)
   TRANTO (1,0,0,0) BY <MU_DEG,SIGMA_DEG>;
```

```
IF NWS > 0                          (* a spare fails *)
   TRANTO (NW,NF,NWS-1,NFS+1) BY NWS*GAMMA;
DEATHIF NF >= NW;
```

When reconfiguration occurs, the probability of switching in a good spare instead of a failed spare is equal to the current proportion of good spares to failed spares in the system. The implicit variable PRG is used to calculate this probability. When all spares are good, the probability of switching in a good spare is 1 and the probability of switching in a bad spare is 0. Conversely, when all spares have failed, the probability of switching in a good spare is 0 and the probability of switching in a bad spare is 1. The tests NWS > 0 and NFS > 0 check for these two cases and prevent the generation of a transition when it is inappropriate.

9.4. Degradable Triad With Partial Detection of Spare Failure

If the system is designed with off-line diagnostics for the spares, this must be included in the model. Two aspects of an off-line diagnostic must be considered. First, a diagnostic usually cannot detect all possible faults and second, a diagnostic requires time to execute. The first aspect is sometimes referred to as the coverage of the diagnostic. The term "coverage" will be avoided because it is used in many different ways, and thus, is confusing. Instead, the first aspect will be termed "fraction of detectable faults" and will be assigned an identifier K. The state space must be expanded to track whether a fault in a spare is detectable:

```
SPACE = (NW: 0..3,          (* number of working processors *)
         NF: 0..3,          (* number of failed active processors *)
         NWS: 0..NSI,       (* number of working spares *)
         NDFS: 0..NSI,      (* number of detectable failed spares *)
         NUFS: 0..NSI);     (* number of undetectable failed spares *)
```

The second aspect requires that a rule be added to generate transitions that decrement the NDFS state-space variable with a fast general recovery distribution:

```
IF NDFS > 0 (* "detectable" spare-failure is detected *)
   TRANTO (NW,NF,NWS,NDFS-1,NUFS) BY <MU_SPD,SIGMA_SPD>;
```

No transition is generated for NUFS faults.

The active processor failure TRANTO rule is the same as in the example in section 9.3, except that the state space is larger. The spare failure TRANTO rule must be altered to include whether the failure is detectable:

```
IF NWS > 0 THEN                 (* a spare fails *)
   TRANTO (NW,NF,NWS-1,NDFS+1,NUFS) BY K*NS*GAMMA; (* detectable fault *)
   TRANTO (NW,NF,NWS-1,NDFS,NUFS+1) BY (1-K)*NS*GAMMA; (* undetectable fault *)
ENDIF;
```

Note that the rates are multiplied by K and (1-K).

The reconfiguration rule is now more complicated than in the example in section 9.3. Three possibilities exist:

1. The faulty active processor is replaced with a working spare.
2. The faulty processor is replaced with a spare containing a detectable fault.
3. The faulty processor is replaced with a spare containing an undetectable fault.

The probability of each case is PRW, PRD, and PRU, respectively, and is defined as follows:

```
IMPLICIT PRW[NWS,NDFS,NUFS] = NWS/(NWS+NDFS+NUFS); (* working spare used *)
IMPLICIT PRD[NWS,NDFS,NUFS] = NDFS/(NWS+NDFS+NUFS);(* spare w/det. fault used *)
```

```
IMPLICIT PRU[NWS,NDFS,NUFS] = NUFS/(NWS+NDFS+NUFS);
   (* spare w/ undet. fault used *)
```

The reconfiguration rule is

```
IF (NF > 0) AND (NWS+NDFS+NUFS > 0) THEN (* a spare becomes active *)
   IF NWS > 0 TRANTO (NW+1,NF-1,NWS-1,NDFS,NUFS) BY <MU,SIGMA,PRW>;
   IF NDFS > 0 TRANTO (NW,NF,NWS,NDFS-1,NUFS) BY <MU,SIGMA,PRD>;
   IF NUFS > 0 TRANTO (NW,NF,NWS,NDFS,NUFS-1) BY <MU,SIGMA,PRU>;
ENDIF;
```

The complete model is

```
NSI = 3;                (* number of spares initially *)
LAMBDA = 1E-4;          (* failure rate of active processors *)
GAMMA = 1E-6;           (* failure rate of spares *)
MU = 7.9E-5;            (* mean time to replace with spare *)
SIGMA = 2.56E-5;        (* stan. dev. of time to replace with spare *)

MU_DEG = 6.3E-5;        (* mean time to degrade to simplex *)
SIGMA_DEG = 1.74E-5;    (* stan. dev. of time to degrade to simplex *)

K = 0.9;                (* fraction of faults that the
                           spare off-line diagnostic can detect *)

MU_SPD = 2.6E-3;        (* mean time to diagnose a failed spare *)
SIGMA_SPD = 1.2E-3;     (* standard deviation of time to diagnose *)

SPACE = (NW: 0..3,      (* number of working processors *)
   NF: 0..3,            (* number of failed active processors *)
   NWS: 0..NSI,         (* number of working spares *)
   NDFS: 0..NSI,        (* number of detectable failed spares *)
   NUFS: 0..NSI);       (* number of undetectable failed spares *)

IMPLICIT PRW[NWS,NDFS,NUFS] = NWS/(NWS+NDFS+NUFS); (* working spare is used *)
IMPLICIT PRD[NWS,NDFS,NUFS] = NDFS/(NWS+NDFS+NUFS); (* spare w/ det. f. used *)
IMPLICIT PRU[NWS,NDFS,NUFS] = NUFS/(NWS+NDFS+NUFS); (* spare w/ undet f. used *)

START = (3,0,NSI,0,0);

IF NW > 0               (* a processor can fail *)
   TRANTO (NW-1,NF+1,NWS,NDFS,NUFS) BY NW*LAMBDA;

IF NWS > 0 THEN           (* a spare fails *)
   TRANTO (NW,NF,NWS-1,NDFS+1,NUFS) BY K*NWS*GAMMA;     (* detectable fault *)
   TRANTO (NW,NF,NWS-1,NDFS,NUFS+1) BY (1-K)*NWS*GAMMA; (* undetectable fault *)
ENDIF;

IF (NF > 0) AND (NWS+NDFS+NUFS > 0) THEN (* a spare becomes active *)
   IF NWS > 0 TRANTO (NW+1,NF-1,NWS-1,NDFS,NUFS) BY <MU,SIGMA,PRW>;
   IF NDFS > 0 TRANTO (NW,NF,NWS,NDFS-1,NUFS) BY <MU,SIGMA,PRD>;
   IF NUFS > 0 TRANTO (NW,NF,NWS,NDFS,NUFS-1) BY <MU,SIGMA,PRU>;
ENDIF;

IF (NF > 0) AND (NWS+NDFS+NUFS = 0) (* no more spares, degrade to simplex *)
   TRANTO (1,0,0,0,0) BY <MU_DEG,SIGMA_DEG>;

IF NDFS > 0 (* “detectable” spare-failure is detected *)
   TRANTO (NW,NF,NWS,NDFS-1,NUFS) BY <MU_SPD,SIGMA_SPD>;

DEATHIF NF >= NW;
```

9.5. Byzantine Faults

In this section, the concept of Byzantine faults and Byzantine-resilient algorithms will be introduced (refs. 20 and 21). Byzantine faults arise from the need to distribute single-source data, such as

sensor data, to the replicated computational sites. Data values from sensors are unreplicated. Although, redundant sensors may exist, they do not produce exactly the same result. Thus, if each processor were connected to one of the redundant sensors, the processors would get different results. This difference is unacceptable in a system that uses exact-match voting algorithms for fault detection. For example, if the system uses voting for fault detection as well as fault masking, Byzantine faults can cause the system to reconfigure a working processor.

Furthermore, the problem is not solved by having each processor read all redundant sensors. Because the redundant processors run off of different clocks, the processors would access the sensors at slightly different times and receive different results. Consequently, a signal-processing algorithm is run on each processor to derive a trustworthy value from the set of redundant sensors. This algorithm necessitates that each sensor be distributed to all redundant processing sites in a consistent manner. Suppose the sensor value is read and stored. If a failure in the transmission medium between this value and the redundant sites occurs, different values may be received by the good processors.

For each processing site to be guaranteed to receive the same set of raw values, special Byzantine-resilient algorithms must be used to distribute the single-source value. The algorithm depends fundamentally upon the availability of four separate fault-isolation regions. If processors are used for the rebroadcasting, then a minimum of four processors must be used. Consequently, a triplex system cannot be Byzantine resilient without the addition of special additional hardware. The model illustrated in figure 36 represents the effect of a Byzantine fault on a triplex system with one spare that does not contain extra hardware.

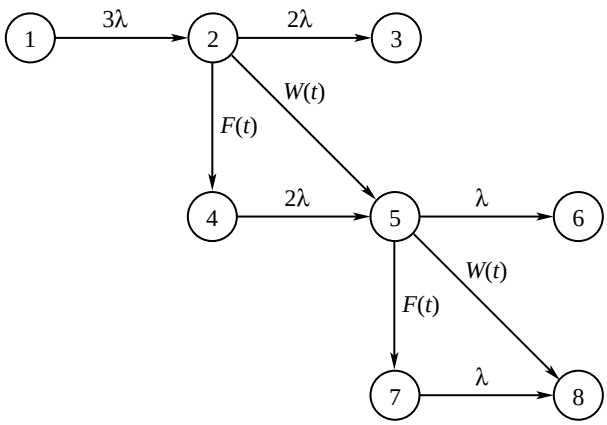


Figure 36. Simple triplex system with one spare subject to Byzantine faults.

This model is the same as the traditional triplex model except that it contains two extra transitions—from state (2) to state (5) and from state (5) to state (8). These transitions represent the situations where a Byzantine fault has confused the operating system into reconfiguring the wrong processor. In the first case, a good processor, not the faulty one, has been replaced by the spare. In the second case, the system incorrectly diagnoses the faulty processor and degrades to a faulty simplex. The competing transitions at state (2) would be

```
2,3 = 2*LAMBDA;
2,4 = <MU_F,STD_F,1-P_W>;
2,5 = <MU_W,STD_W,P_W>;
```

where `P_W` is the probability that the system incorrectly removes a good processor. The competing transitions at state (5)

```
5,6 = 2*LAMBDA;
5,7 = <MU_F,STD_F,1-P_W>;
5,8 = <MU_W,STD_W,P_W>;
```

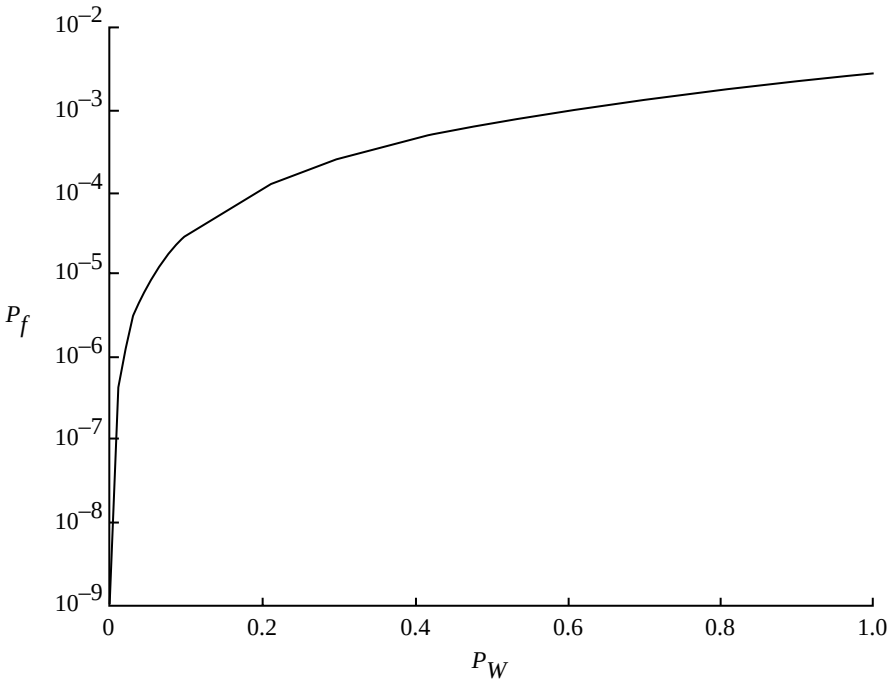


Figure 37. Failure probability as function of P_W.

The parameter P_W is the most critical parameter in this model. This can be seen in figure 37, which shows a plot of the results of executing SURE on the full model:

```
LAMBDA = 1E-4;
MU_F = 1E-4; STD_F = 1E-4;
MU_W = 1E-4; STD_W = 1E-4;
P_W = 0 TO 1 BY 0.1;
1,2 = 3*LAMBDA;
2,3 = 2*LAMBDA;
2,4 = <MU_F,STD_F,1-P_W>;
2,5 = <MU_W,STD_W,P_W>;
4,5 = 3*LAMBDA;
5,6 = 2*LAMBDA;
5,7 = <MU_F,STD_F,1-P_W>;
5,8 = <MU_W,STD_W,P_W>;
7,8 = LAMBDA;
TIME = 10;
```

Unfortunately, very little experimental data are available to aid in the estimation of P_W. For this reason, many conservative system designers have elected to add the additional hardware and software to make the architecture Byzantine resilient, and thus, eliminate this failure mode from the system. However, the failure of any additional hardware must be modeled. Alternatively, the system can be designed in a manner that enables the calculation of the fraction P_W (ref. 22).

10. Systems With Multiple Independent Subsystems

In this section, techniques for modeling systems that contain multiple independent subsystems are discussed. For the subsystems to be independent, it is necessary that no failure dependencies between the subsystems exist. Mathematically, two events *A* and *B* are independent if

$$\text{Prob}\,[A \text{ and } B] \;=\; \text{Prob}\,[A]\;\text{Prob}\,[B]$$

This definition leads to the following:

Property: if *A* and *B* are independent, then

$$\text{Prob}\,[A|B] \;=\; \frac{\text{Prob}\,[A \text{ and } B]}{\text{Prob}\,[B]} \;=\; \text{Prob}\,[A]$$

If two subsystems are located in separate chassis, powered by separate power supplies, electrically isolated from each other, and sufficiently shielded from the environment, it is not unreasonable to assume failure independence.

10.1. System With Two Independent Triad-to-Simplex Subsystems

Consider a system with two triplexes, each of which degrades to a simplex. The system requires both subsystems to be operating for the system to work. Although each processor within a subsystem is identical, the processors in one subsystem can be given a different failure rate from the processors in the other subsystem. Let λ_1 represent the failure rate of the processors in subsystem 1, and λ_2 the failure rate of the processors in subsystem 2. This system is illustrated in figure 38. In this figure, each state has been labeled with four numbers representing four attributes of the system (`NW1`, `NF1`, `NW2`, and `NF2`):

1. `NW1` is the number of working processors in subsystem 1.
2. `NF1` is the number of faulty processors in subsystem 1.
3. `NW2` is the number of working processors in subsystem 2.
4. `NF2` is the number of faulty processors in subsystem 2.

The system starts in state (3030). This notation means that each subsystem has three working processors and no faulty processors. If a processor in subsystem 1 fails, the system transitions to state (2130). If a processor in subsystem 2 fails, the system transitions to state (3021). While in state (2130), the system is trying to reconfigure. If it reconfigures before a second processor in subsystem 1 fails, the system transitions to state (1030). That is, the first subsystem is a simplex and the second subsystem is still a triplex. If a second processor in subsystem 1 fails before it reconfigures, then the system fails in death state (1230). Note that two simultaneous faults have occured in subsystem 1 in this situation. If a second processor in the other subsystem fails before reconfiguration is completed, then the system goes to state (2121). In state (2121) both triads have a single faulty processor. Because the triads are independent, this state does not represent system failure. From this state, four possible events can happen next

1. A second processor in subsystem 1 fails, which causes system failure.
2. A second processor in subsystem 2 fails, which causes system failure.
3. Subsystem 1 reconfigures by degrading to a simplex.
4. Subsystem 2 reconfigures by degrading to a simplex.

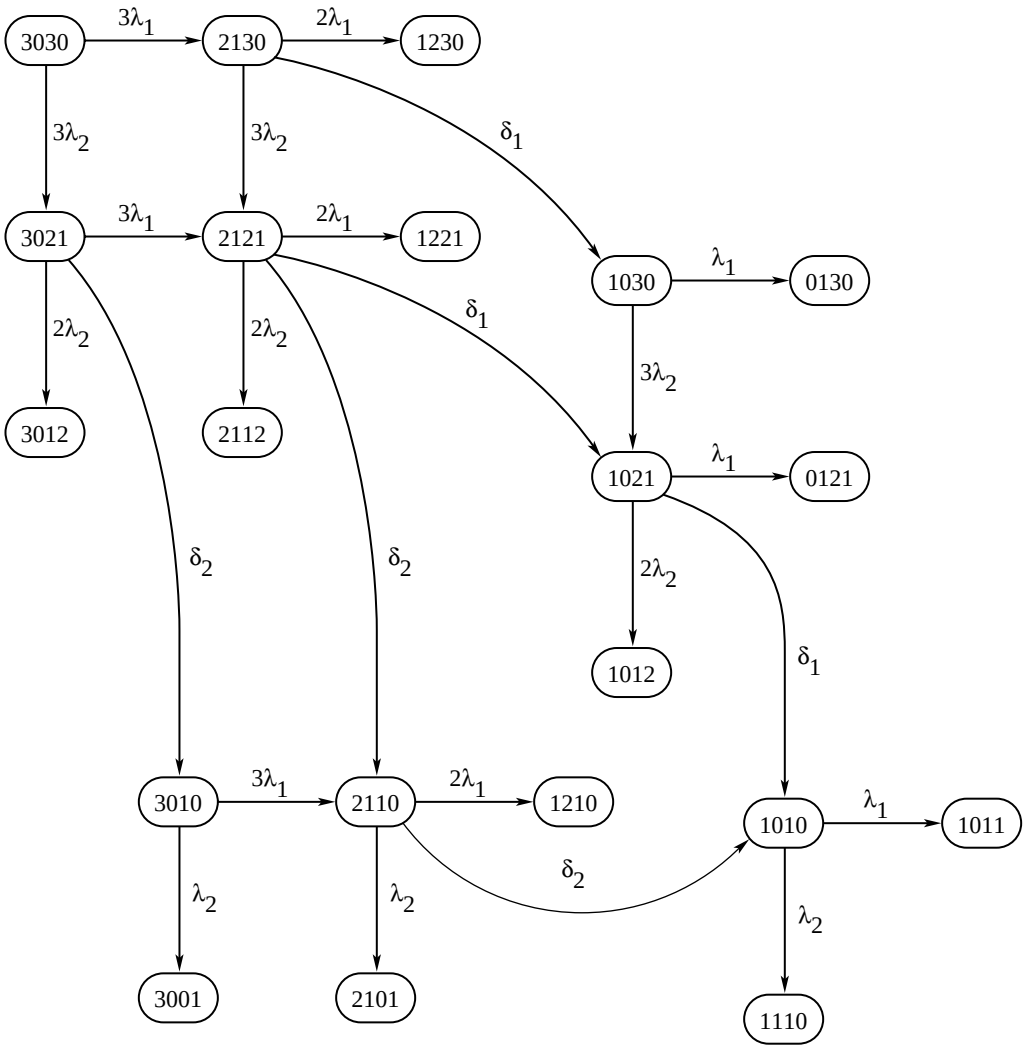


Figure 38. Model of system consisting of two triad to simplex subsystems.

The two reconfiguration transitions go to states (1021) and (2110), respectively. Note that in state (2121) two competing recoveries occur. Because the subsystems are independent, this situation is modeled with two competing transitions.

10.2. ASSIST Model for N Independent Triads

As the number of subsystems is increased, the size of the model can increase rapidly. It very quickly becomes impractical to produce graphical pictures. Fortunately, the ASSIST language can define these models economically. Array state-space variables will be used to model the multiple triads. The syntax for specifying an array state-space variable is

ident: ARRAY[*const .. const*] ⟦ OF *const .. const* ⟧

The range specified within the single square brackets denotes the array range, that is, the range of values over which an array index can vary. The optional range specified after the OF denotes the range of values that the array state-space variable can hold. Individual array state-space variable elements are referenced in other ASSIST statements by specifying the index in square brackets, for example NP[3] denotes the third element of array NP. When using array state-space variables, the repetition feature in

the START statement is convenient to use. Repetition allows the user to specify a sequence such as 3,3,3,3,3 by a shorthand notation, 5 of 3.

The following model describes a system of N independent triads:

```
INPUT N_TRIADS;        (* Number of triads initially *)
LAMBDA = 1E-4;         (* Failure rate of active processors *)
DELTA = 3.6E3;         (* Reconfiguration rate *)

SPACE = (NP: ARRAY[1..N_TRIADS] OF 0..3, (* Num. active processors in triad *)
         NFP: ARRAY[1..N_TRIADS] Of 0..3); (* Num. failed active procs *)

START = (N_TRIADS OF 3, N_TRIADS OF 0);

FOR J = 1, N_TRIADS;
   IF NP[J] > NFP[J] TRANTO NFP[J] = NFP[J]+1
        BY (NP[J]-NFP[J])*LAMBDA; (* Active processor failure *)

   IF NFP[J] > 0 TRANTO
        NP[J]=1, NFP[J]=0 BY FAST DELTA;

   DEATHIF 2 * NFP[J] >= NP[J];
     (* Two faults in an active triad or simplex with a fault *)
ENDFOR;
```

The user is prompted during ASSIST execution for the number of independent triads to be modeled by N_TRIADS. Two array state-space variables are used:

1. NP[J] is the array containing number of active processors in each subsystem J.

2. NFP[J] is the array containing number of faulty processors in each triad J.

Because reconfiguration collapses a triad to a simplex, the value of each NP[J] must always be either 3 or 1. Processor failure in triad J results in the increment of NFP[J]. Note that the DEATHIF condition covers both the triplex and the simplex situation.

10.3. The Additive Law of Probability

As shown in section 10.2, larger systems can be constructed by combining several independent subsystems. If the probability of failure of a single subsystem is $P(E_i)$, then the probability of failure of a system consisting of two of these subsystems that fails when either subsystem fails is

$$P_{sys} = P(E_1 \text{ or } E_2) = P(E_1) + P(E_2) - P(E_1)P(E_2)$$

This formula follows from the independence of the subsystems and the additive law of probability and can be used to greatly simplify the analysis of many systems. The formula can be illustrated with the models presented in section 10.2. ASSIST was used to generate a model of a system consisting of two independent triads. This system could be solved by submitting the model to a Markov solver:

```
air58% paws

 PAWS V7.9.3   NASA Langley Research Center

 1? read twotriads

 2: N_TRIADS = 2;
 3: LAMBDA = 1E-4;
 4: DELTA = 3.6E3;
 5:
 6:
 7:      2(* 3,3,0,0 *),     3(* 3,3,1,0 *)       = (3-0)*LAMBDA;
 8:      2(* 3,3,0,0 *),     4(* 3,3,0,1 *)       = (3-0)*LAMBDA;
 9:      3(* 3,3,1,0 *),     5(* 1,3,0,0 *)       = FAST DELTA;
```

```
10:      3(* 3,3,1,0 *),     1(* 3,3,2,0 DEATH *) = (3-1)*LAMBDA;
11:      3(* 3,3,1,0 *),     6(* 3,3,1,1 *)       = (3-0)*LAMBDA;
12:      4(* 3,3,0,1 *),     7(* 3,1,0,0 *)       = FAST DELTA;
13:      4(* 3,3,0,1 *),     6(* 3,3,1,1 *)       = (3-0)*LAMBDA;
14:      4(* 3,3,0,1 *),     1(* 3,3,0,2 DEATH *) = (3-1)*LAMBDA;
15:      5(* 1,3,0,0 *),     1(* 1,3,1,0 DEATH *) = (1-0)*LAMBDA;
16:      5(* 1,3,0,0 *),     8(* 1,3,0,1 *)       = (3-0)*LAMBDA;
17:      6(* 3,3,1,1 *),     8(* 1,3,0,1 *)       = FAST DELTA;
18:      6(* 3,3,1,1 *),     9(* 3,1,1,0 *)       = FAST DELTA;
19:      6(* 3,3,1,1 *),     1(* 3,3,2,1 DEATH *) = (3-1)*LAMBDA;
20:      6(* 3,3,1,1 *),     1(* 3,3,1,2 DEATH *) = (3-1)*LAMBDA;
21:      7(* 3,1,0,0 *),     9(* 3,1,1,0 *)       = (3-0)*LAMBDA;
22:      7(* 3,1,0,0 *),     1(* 3,1,0,1 DEATH *) = (1-0)*LAMBDA;
23:      8(* 1,3,0,1 *),    10(* 1,1,0,0 *)       = FAST DELTA;
24:      8(* 1,3,0,1 *),     1(* 1,3,1,1 DEATH *) = (1-0)*LAMBDA;
25:      8(* 1,3,0,1 *),     1(* 1,3,0,2 DEATH *) = (3-1)*LAMBDA;
26:      9(* 3,1,1,0 *),    10(* 1,1,0,0 *)       = FAST DELTA;
27:      9(* 3,1,1,0 *),     1(* 3,1,2,0 DEATH *) = (3-1)*LAMBDA;
28:      9(* 3,1,1,0 *),     1(* 3,1,1,1 DEATH *) = (1-0)*LAMBDA;
29:     10(* 1,1,0,0 *),     1(* 1,1,1,0 DEATH *) = (1-0)*LAMBDA;
30:     10(* 1,1,0,0 *),     1(* 1,1,0,1 DEATH *) = (1-0)*LAMBDA;
31:
32: (* NUMBER OF STATES IN MODEL = 10 *)
33: (* NUMBER OF TRANSITIONS IN MODEL = 24 *)
34: (* 12 DEATH STATES AGGREGATED INTO STATE 1 *)

      0.07 SECS. TO READ MODEL FILE
35? run

MODEL FILE = twotriads.mod               PAWS V7.9.3 7 Feb 92 14:43:49

                  PROBABILITY        ACCURACY     ----- RUN #1
-----------   -------------------   --------
              2.9961673328249e-06   7 DIGITS

0.050 SECS. CPU TIME UTILIZED

36? exit
```

Another way to solve this system is to apply the additive law of probability. First solve a single triad subsystem:

```
air58% paws
  PAWS V7.9.3 NASA Langley Research Center

  1? read onetriad

  2: N_TRIADS = 1;
  3: LAMBDA = 1E-4;
  4: DELTA = 3.6E3;
  5:
  6:
  7:      2(* 3,0 *),      3(* 3,1 *)       = (3-0)*LAMBDA;
  8:      3(* 3,1 *),      4(* 1,0 *)       = FAST DELTA;
  9:      3(* 3,1 *),      1(* 3,2 DEATH *) = (3-1)*LAMBDA;
 10:      4(* 1,0 *),      1(* 1,1 DEATH *) = (1-0)*LAMBDA;
 11:
 12: (* NUMBER OF STATES IN MODEL = 4 *)
 13: (* NUMBER OF TRANSITIONS IN MODEL = 4 *)
```

```
 14: (* 2 DEATH STATES AGGREGATED INTO STATE 1 *)
      0.00 SECS. TO READ MODEL FILE
 15? run
MODEL FILE = onetriad.mod                 PAWS V7.9.3 7 Feb 92 14:44:01
                 PROBABILITY       ACCURACY       ----- RUN #1
-----------   -------------------  --------
             1.4980847885419e-06  9 DIGITS
 0.017 SECS. CPU TIME UTILIZED
```

Then, apply the additive law of probability:

$$\begin{aligned} P_{sys} &= P(E_1 \text{ or } E_2) = P(E_1) + P(E_2) - P(E_1)\,P(E_2) \\ &= 1.498 \times 10^{-6} + 1.498 \times 10^{-6} - (1.498 \times 10^{-6})\,(1.498 \times 10^{-6}) \\ &= 2.996 \times 10^{-6} \end{aligned}$$

Notice that the same answer is obtained as in the N-triad model. This calculation is automated in the ORPROB command found in SURE, PAWS, and STEM as illustrated below:

```
air58% paws
  PAWS V7.9.3   NASA Langley Research Center
  1? read0 onetriad
 15? run
                   PROBABILITY        ACCURACY     ----- RUN #1
 -----------   -------------------   --------
               1.4980847885419e-06   9 DIGITS
 16? read0 onetriad
 30? run
                   PROBABILITY        ACCURACY     ----- RUN #2
 -----------   -------------------   --------
               1.4980847885419e-06   9 DIGITS
 31? orprob
    RUN #    PROBABILITY
 ---------- -----------
      1     1.49808e-06
      2     1.49808e-06
 ---------- -----------
 OR PROB = 2.99617e-06
```

The SURE command `read0` prevents the echoing of the file as it is read in. It is equivalent to `ECHO = 0; READ onetriad`. The `orprob` command at line 31 first reports the results of the previous runs, then gives the result of applying the additive law of probability to independent events.

11. Model Pruning

A model of a system with a large number of components tends to have many long paths that consist of one or two failures of each component before a condition of system failure is reached. Because the occurrence of so many failures is unlikely during a short mission, these long paths typically contribute insignificant amounts to the probability of system failure. The dominant failure modes of the system are typically the short paths to system failure that consist of failures of critically coupled components. The

model can be pruned to eliminate the long paths by conservatively assuming that system failure occurs earlier on these paths. Model pruning is supported in ASSIST by the PRUNEIF statement. The PRUNEIF statement has the same syntax as the DEATHIF statements:

PRUNEIF *condition*;

where *condition* is a Boolean expression describing the states to be pruned. The following syntax:

PRUNIF *condition*;

is also acceptable. The pruned states generated are grouped by the PRUNEIF statement they satisfy, just as the death states are. In a model generated from an input file with three DEATHIF statements and two PRUNEIF statements, states (1) through (3) will be death states corresponding to the three DEATHIF statements and states (4) and (5) will be pruned states corresponding to the two PRUNEIF statements. ASSIST generates a statement in the model file that identifies the pruned states in the model. For example, the model with four death states and two pruned states would contain the statement

```
PRUNESTATES = (5,6);
```

which will be used by SURE to separately list the prune state probabilities from the death state probabilities. Versions of SURE earlier than 6.0 will simply list the pruned states as death states. The SURE program reports the ASSIST pruned states separately from the death states as follows:

```
DEATHSTATE     LOWERBOUND     UPPERBOUND    COMMENTS                  RUN #4
----------    -----------    -----------    --------------------------------
     1        9.99500e-12    1.00000e-11
     2        1.66542e-10    1.66667e-10
     3        9.99500e-14    1.00000e-13
sure prune    0.00000e+00    1.22666e-14
              -----------    -----------
  SUBTOTAL    1.76637e-10    1.76779e-10

PRUNESTATE     LOWERBOUND     UPPERBOUND
----------    -----------    -----------
prune   5     9.99500e-15    1.00000e-14
prune   6     9.99500e-18    1.00000e-17
              -----------    -----------
  SUBTOTAL    1.00050e-14    1.00100e-14

TOTAL         1.76637E-10    1.76789e-10
```

In the TOTAL line, the upper bound includes the contribution of the pruned states, whereas the lower bound does not. Thus, the TOTAL lines are valid bounds on the system failure probability. If the prune state upper bound is significant with respect to the TOTAL upper bound, then the model has probably been pruned too severely. The upper and lower bounds can be made significantly closer by relaxing the amount of pruning. The ASSIST program wrote the following into the SURE input file to inform the SURE program which states are ASSIST-level pruned states:

```
PRUNESTATES = (5,6);
```

Two types of pruning are supported by SURE and ASSIST. One is SURE-level pruning, which was described in section 5.3.6, and the other is ASSIST-level pruning, which is described in this section. ASSIST-level pruning is done at model generation time. After model building is completed, the amount of processing time can be reduced by using SURE-level pruning. This is invoked by the SURE command PRUNE = <rate>. SURE-level pruning will still be effective in conjunction with ASSIST-level pruning. ASSIST pruning will be demonstrated for many systems in subsequent sections.

12. Multiple Subsystems With Failure Dependencies

Flight control systems are composed of many different components, including processors. Previous sections have only considered processors. In this section, how to include the fault behavior of all components in a typical flight control system architecture in the reliability model will be discussed.

12.1. Simple Flight Control System

The system shown in figure 39 consists of five subsystems:

1. Triplicated sensors
2. Triplicated bus from sensor to processor SP_BUS
3. Degradable quadraplex of processors
4. Triplicated bus from processor to actuator PA_BUS
5. Force-sum voting actuator

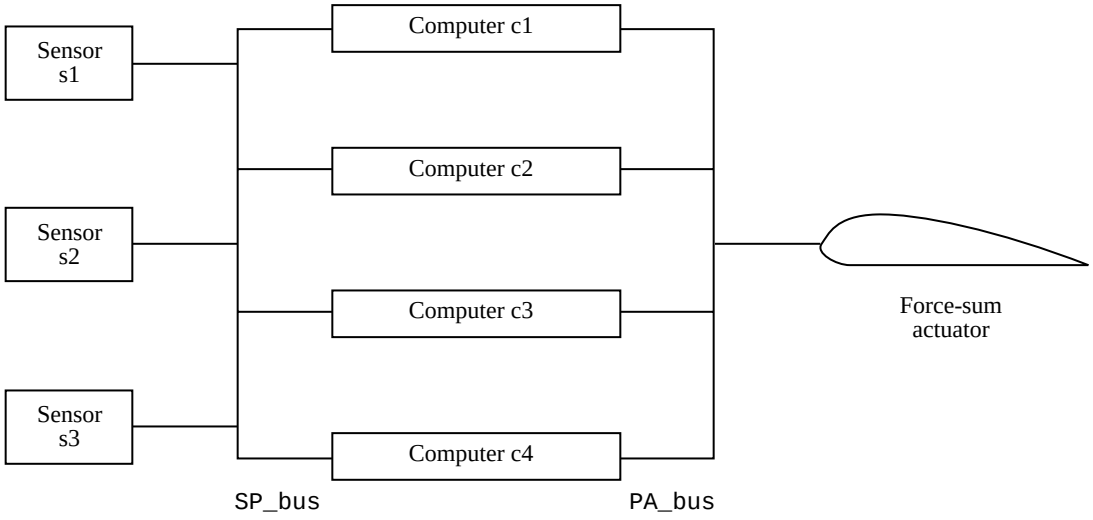


Figure 39. System with five subsystems.

All sensor values are sent to all processors over the SP_BUS. The system fails when two sensors have failed. No dependencies exist between the sensors and the processors. Thus, as long as the fault-tolerant SP_BUS is working, each sensor value can reach all processors. Of course, the system fails if two channels of the SP_BUS fail. The processors form a degradable quadraplex. Outputs of the processors are sent over all channels of the triplicated PA_BUS. This transmission is accomplished by time multiplexing the bus. As long as at least two PA_BUS channels are working, all working-processor outputs reach the force-sum actuation system. Thus, no dependencies exist between the processors, the force-sum actuator, and the PA_BUS. For simplicity, the force-sum actuation system will be treated as a black box. Force-sum actuation systems are complex mechanical and electronic systems that utilize multiple actuators, and sometimes, internal fault-tolerant electronics, and thus, would require a sophisticated model to properly analyze. The force-sum actuator will be assumed to ignore the outputs of all processors that have been removed from the quadraplex. Thus, the force-sum actuator performs a mechanical vote on all outputs from processors that are currently in the configuration. The failure rate of the force-sum actuator is 10^{-8}/hr.

Because no failure dependencies exist, the separate subsystems can be represented by separate reliability models. Each model is solved in isolation. Finally, the results are added together probabilistically, that is, the probability of the union as described in section 10.3. The SURE command ORPROB performs the probabilistic addition automatically. The SURE input file is

```
LAMBDA_SENSORS = 3.8E-6;
1,2 = 3*LAMBDA_SENSORS;
2,3 = 2*LAMBDA_SENSORS;
RUN;

LAMBDA_SP_BUS = 3.8E-6;
1,2 = 3*LAMBDA_SP_BUS;
2,3 = 2*LAMBDA_SP_BUS;
RUN;

LAMBDA_PA_BUS = 3.8E-6;
1,2 = 3*LAMBDA_PA_BUS;
2,3 = 2*LAMBDA_PA_BUS;
RUN;

LAMBDA_ACT = 1E-8;
1,2 = LAMBDA_ACT;
RUN;

LAMBDA = 1E-4;                  (* Failure rate of processor *)
MEANREC = 1E-5;                 (* Mean reconfiguration time *)
STDREC = 1E-5;                  (* Standard deviation of " " *)

1,2 = 4*LAMBDA;
2,3 = 3*LAMBDA;
2,4 = <MEANREC,STDREC>;
4,5 = 3*LAMBDA;
5,6 = 2*LAMBDA;
5,7 = <MEANREC,STDREC>;
7,8 = LAMBDA;
RUN;

ORPROB;
```

The interactive session uses a single input file with three models, three RUN statements, and an ORPROB command:

```
$ sure

  SURE V7.4    NASA Langley Research Center

  1? read sa

  2: LAMBDA_SENSORS = 3.8E-6;
  3: 1,2 = 3*LAMBDA_SENSORS;
  4: 2,3 = 2*LAMBDA_SENSORS;
  5: RUN;

MODEL FILE = sa.mod                      SURE V7.4 24 Jan 90   10:28:46

               LOWERBOUND   UPPERBOUND    COMMENTS                RUN #1
 -----------   -----------   -----------    --------------------------------
              4.33173e-09  4.33200e-09

1 PATH(S) TO DEATH STATES
0.034 SECS. CPU TIME UTILIZED
```

```
 6:
 7: LAMBDA_SP_BUS = 3.8E-6;
 8: 1,2 = 3*LAMBDA_SP_BUS;
 9: 2,3 = 2*LAMBDA_SP_BUS;
10: RUN;
```

MODEL FILE = sa.mod SURE V7.4 24 Jan 90 10:28:46

LOWERBOUND	UPPERBOUND	COMMENTS	RUN #2
4.33173e-09	4.33200e-09		

```
1 PATH(S) TO DEATH STATES
0.034 SECS. CPU TIME UTILIZED
11:
12: LAMBDA_PA_BUS = 3.8E-6;
13: 1,2 = 3*LAMBDA_PA_BUS;
14: 2,3 = 2*LAMBDA_PA_BUS;
15: RUN;
```

MODEL FILE = sa.mod SURE V7.4 24 Jan 90 10:28:47

LOWERBOUND	UPPERBOUND	COMMENTS	RUN #3
4.33173e-09	4.33200e-09		

```
1 PATH(S) TO DEATH STATES
0.050 SECS. CPU TIME UTILIZED
16:
17: LAMBDA_ACT = 1E8;
18: 1,2 = LAMBDA_ACT;
19: RUN
```

MODEL FILE = sa.mod SURE V7.4 24 Jan 90 10:28:47

LOWERBOUND	UPPERBOUND	COMMENTS	RUN #4
1.00000e-07	1.00000e-07		

```
1 PATH(S) TO DEATH STATES
0.050 SECS. CPU TIME UTILIZED
20:
21: LAMBDA = 1E-4;                  (* Failure rate of processor *)
22: MEANREC = 1E-5;                 (* Mean reconfiguration time *)
23: STDREC =  1E-5;                 (* Standard deviation of " " *)
24:
25: 1,2 = 4*LAMBDA;
26: 2,3 = 3*LAMBDA;
27: 2,4 = <MEANREC,STDREC>;
28: 4,5 = 3*LAMBDA;
29: 5,6 = 2*LAMBDA;
30: 5,7 = <MEANREC,STDREC>;
31: 7,8 = LAMBDA;
32: RUN;
```

MODEL FILE = sa.mod SURE V7.4 24 Jan 90 10:28:47

```
                LOWERBOUND    UPPERBOUND    COMMENTS                RUN #5
 -----------    -----------   -----------    --------------------------------
                2.00574e-09  2.01201e-09

3 PATH(S) TO DEATH STATES
0.134 SECS. CPU TIME UTILIZED
33:
34: ORPROB;

 MODEL FILE = sa.mod                       SURE V7.4 24 Jan 90   10:28:48

   RUN #      LOWERBOUND    UPPERBOUND
----------   -----------   -----------
     1       4.33173e-09   4.33200e-09
     2       4.33173e-09   4.33200e-09
     3       4.33173e-09   4.33200e-09
     4       1.00000e-07   1.00000e-07
     5       2.00574e-09   2.01201e-09
----------   -----------   -----------
 OR PROB =   1.15001e-07   1.15008e-07

        0.70 SECS. TO READ MODEL FILE
35? exit
```

The sensor subsystem for this example was very simple to model. Section 12.2 shows a more complex sensor subsystem.

12.2. Flight Control System With Failure Dependency

The flight control system modeled in this section contains components that are dependent upon each other. The system consists of four sensors (s1, s2, s3, and s4), four computers (c1, c2, c3, and c4) and a quadraplex PA_BUS. The system is illustrated in figure 40.

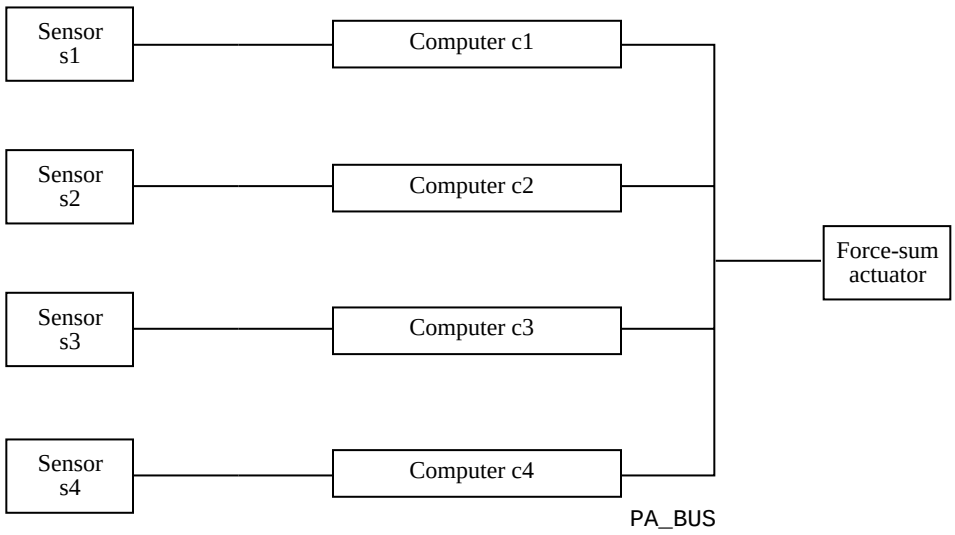


Figure 40. Flight control system with dependent components.

Each sensor is connected to only one processor; therefore, the failure of a processor makes the attached sensor useless. However, the reverse is not true because the processors are cross-strapped and exchange sensor values.

Because the sensors and the computers are dependent, they can no longer be analyzed with separate models. Furthermore, counting the number of sensor and computer failures is no longer sufficient. The specific sensor or computer that has failed must be recorded in the state information. Consider the two sequences of failures:

1. Sensor 1 fails, then processor 1 fails.

2. Sensor 1 fails, then processor 2 fails.

Note that in both cases one sensor and one processor fail. However, the second sequence of failures is more serious, because it results in the loss of two sensors.

```
LS    = 6.5E-5;   (* Failure rate of sensors *)
LC    = 3.5E-4;   (* Failure rate of computers *)
DELTA = 1E4;      (* Rate of removing faulty computer from configuration *)
ONEDEATH OFF;
SPACE = (WS : ARRAY[1..4] OF 0..1,  (* Status of the 3 sensors *)
         AC : ARRAY[1..4] OF 0..1,   (* Computers in configuration *)
         WC : ARRAY[1..4] OF 0..1    (* Working computers in configuration *)
        );
START = (4 OF 1, 4 OF 1, 4 OF 1);
FOR I = 1,4
   IF WS[I] = 1 TRANTO WS[I] = 0            BY LS; (* Sensor   fails *)
   IF WC[I] = 1 TRANTO WC[I] = 0, WS[I] = 0 BY LC; (* Computer fails *)
   IF AC[I] = 1 AND WC[I] = 0 TRANTO AC[I] = 0 BY DELTA;
ENDFOR;
DEATHIF WS[1] + WS[2] + WS[3] + WS[4] = 0;
DEATHIF AC[1] + AC[2] + AC[3] + AC[4] >= 2 * (WC[1] + WC[2] + WC[3] + WC[4]);
```

The first three statements define the failure rates of the sensors, the failure rates of the computers, and the recovery rate of the computers. The `ONEDEATH OFF` statement turns off aggregation of the death states. By default, ASSIST collapses all death states into one state. This command causes ASSIST to give each death state a unique state number. The `FOR` loop effectively creates 12 `TRANTO` rules. These rules create failure transitions corresponding to failures of the eight individual components and recovery transitions corresponding to the removal of faulty processors. The two `DEATHIF` statements define two failure conditions:

1. No sensors are left.

2. A majority of the processors are no longer working.

The ASSIST output is

```
ASSIST VERSION 7.1                          NASA Langley Research Center
PARSING TIME = 0.23 sec.
generating SURE model file...
 100 transitions processed.
 200 transitions processed.
 300 transitions processed.
 400 transitions processed.
 500 transitions processed.
 00 transitions processed.
RULE GENERATION TIME = 1.09 sec.
```

```
NUMBER OF STATES IN MODEL = 397
NUMBER OF TRANSITIONS IN MODEL = 600
```

The transcript of the SURE run follows:

```
% sure
  SURE V7.9.8   NASA Langley Research Center
  1? read0 fcs-dep

       0.60 SECS. TO READ MODEL FILE
1810? run
 MODEL FILE = fcs-dep.mod               SURE V7.9.8 27 Jun 94  08:56:16

             LOWERBOUND  UPPERBOUND   COMMENTS           RUN #1
-----------  ----------  ----------   --------------------------
             1.71645e-07  1.71645e-07  <prune 1.3e-14> <ExpMat>

6171 PATH(S) TO DEATH STATES 666 PATH(S) PRUNED
HIGHEST PRUNE LEVEL =  1.59354e-15
Q(T) ACCURACY >= 9 DIGITS
30.500 SECS. CPU TIME UTILIZED

% sure
  SURE V7.9.8   NASA Langley Research Center
  1? read0 fcs-dep

       0.60 SECS. TO READ MODEL FILE
1810? run
 MODEL FILE = fcs-dep.mod               SURE V7.9.8 27 Jun 94  09:25:56

              LOWERBOUND  UPPERBOUND  COMMENTS          RUN #1
 -----------  ----------  ----------   -------------------------
              1.71645e-07  1.71645e-0   <prune 1.3e-14> <ExpMat>

6171 PATH(S) TO DEATH STATES 666 PATH(S) PRUNED
HIGHEST PRUNE LEVEL =  1.59354e-15
Q(T) ACCURACY >= 9 DIGITS
29.883 SECS. CPU TIME UTILIZED
```

The SURE program found 6171 paths in the model that lead to a death state. The SURE pruning facility pruned 666 paths that contributed a total of 1.3×10^{-14} to the upper bound. Because this number is small when compared with the final value of 1.7×10^{-7}, the pruning is not too severe. The SURE execution time was 29.9 sec. Although, the default pruning level was used in this case, the user can specify a probability level for model pruning, for example 10^{-15}. Each time the probability of encountering a state in the model falls below the specified value, that path is pruned. The SURE program sums the probabilities of all pruned paths and reports that value to the user as the estimated error due to pruning. SURE always adds the pruned value to the upper bound.

This run did not account for the failure of the PA_BUS or the actuator. Because these components are not dependent upon the sensors or the computers, they can be analyzed by separate models:

```
1811? read fcs-other
1812: LAMBDA_PA_BUS = 3.8E-6;
1813: 1,2 = 3*LAMBDA_PA_BUS;
```

```
1814: 2,3 = 2*LAMBDA_PA_BUS;
1815: RUN;
 MODEL FILE = fcs-other.mod          SURE V7.9.8 27 Jun 94  09:28:29

               LOWERBOUND    UPPERBOUND    COMMENTS            RUN #2
-----------   -----------   -----------   --------------------------
              4.33173e-09   4.33200e-09
 1 PATH(S) TO DEATH STATES
 0.000 SECS. CPU TIME UTILIZED
1816:
1817: LAMBDA_ACT = 1E-8;
1818: 1,2 = LAMBDA_ACT;
1819: RUN;
 MODEL FILE = fcs-other.mod             SURE V7.9.8 27 Jun 94  09:28:29

              LOWERBOUND   UPPERBOUND  COMMENTS            RUN #3
-----------   -----------  -----------   ------------------------
              1.00000e-0  1.00000e-07

 1 PATH(S) TO DEATH STATES
 0.016 SECS. CPU TIME UTILIZED

       0.03 SECS. TO READ MODEL FILE
1820? ORPROB
 MODEL FILE = fcs-other.mod             SURE V7.9.8 27 Jun 94  09:28:44

    RUN #      LOWERBOUND    UPPERBOUND
 ----------   -----------   -----------
     1        1.71645e-07   1.71645e-07
     2        4.33173e-09   4.33200e-09
     3        1.00000e-07   1.00000e-07
 ----------   -----------   -----------
  OR PROB =   2.75976e-07   2.75976e-07

1821?
```

The final ORPROB command calculates the final system reliability from the three separate SURE runs.

Suppose a new X component is added to the architecture. This component is dependent upon the computer to which it is attached in the same way that the sensors are. The new ASSIST model is

```
LS = 6.5E-5;   (* Failure rate of sensors *)
LC = 3.5E-4;   (* Failure rate of computers *)
LX = 5.0E-5;   (* Failure rate of X component *)
DELTA = 1E4;   (* Rate of removing faulty computer from configuration *)
ONEDEATH OFF;
SPACE = (WS : ARRAY[1..4] OF 0..1,  (* Status of the 3 sensors *)
         AC : ARRAY[1..4] OF 0..1,  (* Computers in configuration *)
         WC : ARRAY[1..4] OF 0..1,  (* Working computers in configuration *)
         WX : ARRAY[1..4] OF 0..1  (* Status of the 4 X-components *)
        );
START = (4 OF 1, 4 OF 1, 4 OF 1, 4 OF 1);
FOR I = 1,4
   IF WS[I] = 1 TRANTO WS[I] = 0                        BY LS;
   IF WC[I] = 1 TRANTO WC[I] = 0, WS[I] = 0, WX[I] = 0 BY LC;
   IF AC[I] = 1 AND WC[I] = 0 TRANTO AC[I] = 0         BY FAST DELTA;
```

```
   IF WX[I] = 1 TRANTO WX[I] = 0,                        BY LX;
ENDFOR;

DEATHIF WS[1] + WS[2] + WS[3] + WS[4] = 0;
DEATHIF WX[1] + WX[2] + WX[3] + WX[4] = 0;
DEATHIF AC[1] + AC[2] + AC[3] + AC[4] >= 2 * (WC[1] + WC[2] + WC[3] + WC[4]);
```

Notice that the second TRANTO statement in the FOR loop has been modified to include the dependency of the X component on the computer to which it is connected. The ASSIST generation transcript follows:

```
% assist frcs-dep
ASSIST VERSION 7.1                              NASA Langley Research Center
PARSING TIME = 0.22 sec.
generating SURE model file...
 100 transitions processed.
 200 transitions processed.
 300 transitions processed.
 400 transitions processed.
 500 transitions processed.
 600 transitions processed.
 700 transitions processed.
 800 transitions processed.
 900 transitions processed.
1000 transitions processed.
2000 transitions processed.
3000 transitions processed.
4000 transitions processed.
5000 transitions processed.
6000 transitions processed.
7000 transitions processed.
8000 transitions processed.
9000 transitions processed.
10000 transitions processed.
RULE GENERATION TIME = 28.12 sec.
NUMBER OF STATES IN MODEL = 8387
NUMBER OF TRANSITIONS IN MODEL = 14328
```

The addition of the four X components to the system has increased the number of states from 397 to 8387, a significant increase. The system is allowed to aggregate the death states. The ASSIST program is rerun with the death state aggregation and the following is obtained:

```
% assist fcs-dep
ASSIST VERSION 7.1                              NASA Langley Research Center
PARSING TIME = 0.21 sec.
generating SURE model file...
 100 transitions processed.
  .
  .
10000 transitions processed.
RULE GENERATION TIME = 27.60 sec.
NUMBER OF STATES IN MODEL = 2214
NUMBER OF TRANSITIONS IN MODEL = 14328
6176 DEATH STATES AGGREGATED INTO STATES 1 - 3
```

Death state aggregation has significantly reduced the number of states, but a large number of transitions remain in the model. This large number of transitions is one of the major difficulties encountered when modeling real systems. The number of states and transitions in the model grows exponentially with the number of components. The reason that the models become large is that many combinations of components in the system can fail before system failure occurs. In this model, combinations of up to four or more failures that involve different components can occur before a condition of system failure is reached. Because the occurrence of so many failures is unlikely during a short mission, these long paths typically contribute insignificant amounts to the probability of system failure. The dominant failure modes of the system are typically the short paths to system failure that consist of failures of only two or three components. Although the long paths contribute insignificantly to the probability of system failure, the majority of the execution time needed to generate and solve this model is spent handling long paths.

Fortunately, model pruning can be used to eliminate the long paths to system failure by conservatively assuming that system failure occurs earlier on these paths. However, a more effective strategy is to prune long paths in ASSIST to prevent the generation of all these paths. Although SURE will prune them, a large amount of time can be consumed by SURE in reading the generated file.

Pruning in the ASSIST program can reduce the model generation time and memory requirements, as well as the solution time. However, because pruning in ASSIST must be based on state-space variable values rather than probability calculations, it must be done more crudely. The ASSIST user typically specifies pruning at a certain level of component failures in the system. For example, if the user knows or suspects that the dominant failures occur after only three or four component failures, then ASSIST can be commanded to prune the model at the fourth component failure. This pruning is usually done by introducing a new state space, NF, that counts the total number of component failures in the system. This state-space variable must be incremented in every TRANTO statement that defines a component failure. The input file for flight control system with the X component modified to perform pruning at the fourth component failure level is

```
LS = 6.5E-5;   (* Failure rate of sensors *)
LC = 3.5E-4;   (* Failure rate of computers *)
LX = 5.0E-5;   (* Failure rate of X component *)
DELTA = 1E4;   (* Rate of removing faulty computer from configuration *)

ONEDEATH OFF;

SPACE = (WS : ARRAY[1..4] OF 0..1,  (* Status of the 3 sensors *)
         AC : ARRAY[1..4] OF 0..1,  (* Computers in configuration *)
         WC : ARRAY[1..4] OF 0..1,  (* Working computers in configuration *)
         WX : ARRAY[1..4] OF 0..1,  (* Status of the 4 X-components *)
         NF : 0..16
        );

START = (4 OF 1, 4 OF 1, 4 OF 1, 4 OF 1, 0);

FOR I = 1,4
   IF WS[I] = 1 TRANTO WS[I] = 0, NF = NF + 1    BY LS;
   IF WC[I] = 1 TRANTO WC[I] = 0, WS[I] = 0, WX[I] = 0 NF = NF + 1 BY LC;
   IF AC[I] = 1 AND WC[I] = 0 TRANTO AC[I] = 0   BY FAST DELTA;
   IF WX[I] = 1 TRANTO WX[I] = 0, NF = NF + 1    BY LX;
ENDFOR;

DEATHIF WS[1] + WS[2] + WS[3] + WS[4] = 0;
DEATHIF WX[1] + WX[2] + WX[3] + WX[4] = 0;
DEATHIF AC[1] + AC[2] + AC[3] + AC[4] >= 2 * (WC[1] + WC[2] + WC[3] + WC[4]);

PRUNEIF NF >= 4;
```

Notice that all failure TRANTO statements now increment the NF variable. The PRUNEIF statement informs ASSIST to prune all states where NF >= 4. The ASSIST transcript follows:

```
% assist fcs-dep-prune
ASSIST VERSION 7.1                          NASA Langley Research Center
PARSING TIME = 0.26 sec.
generating SURE model file...
 100 transitions processed.
  .
  .
3000 transitions processed.
RULE GENERATION TIME = 7.45 sec.
NUMBER OF STATES IN MODEL = 445
NUMBER OF TRANSITIONS IN MODEL = 3476
736 DEATH STATES AGGREGATED INTO STATES 1 - 3
1664 PRUNE STATES AGGREGATED INTO STATE 4
```

Thus, pruning at the fourth component level reduced the model from 2214 states and 14 328 transitions to only 445 states and 3476 transitions. All pruned states are grouped into state (4). If more than one PRUNEIF statement had existed, the pruned states would have been grouped according to which PRUNEIF statement was satisfied.

Whenever the ASSIST input file includes one or more PRUNEIF statements, the program automatically includes a statement in the SURE input file that indicates which states are pruned states generated by ASSIST. For this model, the following is generated:

```
PRUNESTATES = 4;
```

The output from the SURE run is

```
% sure
  SURE V7.9.8   NASA Langley Research Center
  1? read0 fcs-dep-prune
    PRUNE STATES ARE: 4

        4.33 SECS. TO READ MODEL FILE
10442? list = 2;
10443? run
 MODEL FILE = fcs-dep-prune.mod        SURE V7.9.8 27 Jun 94  10:19:42
 DEATHSTATE    LOWERBOUND    UPPERBOUND    COMMENTS            RUN #1
 ----------   -----------   -----------   --------------------------
      1       1.16655e-10   1.18689e-10
      2       8.30445e-11   8.45037e-11
      3       1.70577e-07   1.73274e-07
 sure prune   0.00000e+00   1.55244e-12
               -----------   -----------
   SUBTOTAL   1.70777e-07   1.73478e-07
```

```
 PRUNESTATE     LOWERBOUND     UPPERBOUND
 ----------   -----------   -----------
 prune   4    4.22619e-10   4.29605e-10
              -----------   -----------
   SUBTOTAL   4.22619e-10   4.29605e-10

 TOTAL        1.70777e-07   1.73909e-07

  13836 PATH(S) TO DEATH STATES, 243 PATH(S) PRUNED
  HIGHEST PRUNE LEVEL =  5.85986e-12

 1.800 SECS. CPU TIME UTILIZED
```

The states pruned by ASSIST are reported separately from the death states. When reporting the total bounds on probability of system failure (in the line labeled "TOTAL"), the upper bound includes the contribution of the prune states, whereas the lower bound does not. Thus, the TOTAL line reports valid bounds on the system failure probability. If the PRUNESTATE upper bound is significant with respect to the TOTAL upper bound, then the model has probably been pruned too severely in ASSIST. For this example, the upper bound on the error due to the pruning done in ASSIST is 4.29605×10^{-10}. The SURE program performed some additional pruning which added 1.55×10^{-12}. Both numbers are insignificant in comparison with the total upper bound of 1.7×10^{-7}. The user often wants to find the best level of pruning. This level can be found by rerunning the model with different levels of pruning. The model is rerun with level 3 pruning. That is, the PRUNEIF statement is changed to PRUNEIF NF >= 3;

The following transcript shows the result of running both SURE and ASSIST:

```
%assist fcs-dep-prune
ASSIST VERSION 7.1                          NASA Langley Research Center
PARSING TIME = 0.22 sec.
generating SURE model file...
 100 transitions processed.
  .
  .
1000 transitions processed.
RULE GENERATION TIME = 2.41 sec.
NUMBER OF STATES IN MODEL = 123
NUMBER OF TRANSITIONS IN MODEL = 1060
132 DEATH STATES AGGREGATED INTO STATES 1 - 3
724 PRUNE STATES AGGREGATED INTO STATE 4

% sure

  SURE V7.9.8   NASA Langley Research Center

  1? read0 fcs-dep-prune

    PRUNE STATES ARE: 4

       1.33 SECS. TO READ MODEL FILE
3194? run

 MODEL FILE = fcs-dep-prune.mod        SURE V7.9.8 27 Jun 94  10:27:34
```

```
                LOWERBOUND   UPPERBOUND     COMMENT           RUN #1
  -----------  -----------  -----------    -------------------------
               1.70286e-07   5.35732e-07    <prune 1.2e-10>

1724 PATH(S) TO DEATH STATES 6 PATH(S) PRUNED
HIGHEST PRUNE LEVEL =  5.54876e-11
ASSIST PRUNE STATE PROBABILITY IS IN [ 3.57338e-07,  3.62633e-07]
0.233 SECS. CPU TIME UTILIZED
```

A significant reduction in the size of the model was obtained; however, the reported pruning by ASSIST of 3.6×10^{-7} is significant in comparison with the upper bound. Although this amount has been added to the upper bound, and thus, the bound is indeed an upper bound, it is not close to the lower bound of 1.7×10^{-7}. If this level of accuracy is not sufficient, one must return to level 4 pruning.

12.3. Monitored Sensors

In this example, a set of monitored sensors is modeled. The system consists of five sensors with five monitors to detect the failure of each sensor. Sensors fail at rate λ_S, and monitors fail at rate λ_M. The monitors are assumed to fail in a fail-stop manner, that is, a good sensor is not falsely accused. If a sensor fails and the monitor is working, the monitor will detect with 90-percent probability that the sensor failed and will remove that sensor from the active configuration. If the monitor has failed or does not detect that the sensor has failed, then the faulty sensor remains in the active configuration and contributes faulty answers to the voting. There is no other means of reconfiguration.

Because values from all sensors in the active configuration are voted, system failure occurs when half or more of the active sensors are faulty.

The ASSIST input file to describe this system is as follows:

```
LAMBDA_S = 1E-4;    (* Failure rate of sensor *)
LAMBDA_M = 1E-5;    (* Failure rate of monitor *)
COV = .90;          (* Detection coverage of monitor *)

SPACE = (NW: 0..5,      (* Number of working sensors *)
         NW_MON: 0..5,  (* Number of working sensors with monitors *)
         NF: 0..5);     (* Number of failed active sensors *)

START = (5,5,0);   (* Start with 5 working sensors with monitors *)

   (* Failure of monitored sensor *)
IF (NW_MON > 0) THEN
   TRANTO NW=NW-1,NW_MON=NW_MON-1 BY COV*LAMBDA_S;
   TRANTO NW=NW-1,NW_MON=NW_MON-1,NF=NF+1 BY (1-COV)*LAMBDA_S;
ENDIF;

   (* Failure of unmonitored sensor *)
IF (NW > NW_MON) TRANTO NW = NW-1, NF=NF+1 BY (NW-NW_MON)*LAMBDA_S;

   (* Failure of monitor *)
IF NW_MON > 0 TRANTO NW_MON = NW_MON-1 BY LAMBDA_M;

DEATHIF NF >= NW;
```

The state space consists of three variables: `NW`, `NW_MON`, and `NF`. The state-space variable `NW` represents the number of working sensors in the active configuration and is decremented whenever a monitor detects that its sensor has failed. The variable `NW_MON` represents how many of the `NW` sensors

have functioning monitors. This variable is decremented whenever a monitor fails or a monitored sensor fails. The variable NF represents the number of failed sensors in the active configuration and is incremented whenever a sensor fails and its monitor is either faulty or fails to detect that the sensor has failed.

12.4. Two Triads With Three Power Supplies

This example consists of two triads of computers with one triad of power supplies connected so that one computer in each triad is connected to each power supply. Thus, if a power supply fails, then one computer in each triad fails. Because of the complex failure dependencies, this system is not easy to model. The usual method of using state-space variables to represent the number of failed computers in each triad is insufficient because which computers have failed is also important state information. One method to model this system is to use the state-space variables as flags to indicate the failure of each computer and power supply. This method uses a large number of state-space variables, but the system can be described with only a few simple TRANTO statements. The ASSIST input file is as follows:

```
LAM_PS = 1E-6;             (* Failure rate of power supplies *)
LAM_C = 1E-5;              (* Failure rate of computers *)

SPACE = (CAF: ARRAY[1..3] OF 0..1,     (* Failed computers in Triad A *)
         CBF: ARRAY[1..3] OF 0..1,     (* Failed computers in Triad B *)
         PSF: ARRAY[1..3] OF 0..1);    (* Failed power supplies *)
START = (9 OF 0);

DEATHIF CAF[1] + CAF[2] + CAF[3] > 1;  (* 2/3 computers in Triad A failed *)
DEATHIF CBF[1] + CBF[2] + CBF[3] > 1;  (* 2/3 computers in Triad B failed *)

FOR I = 1,3
   IF CAF[I]=0 TRANTO CAF[I] = 1 BY LAM_C;
      (* Failure of computer in Triad A *)
   IF CBF[I]=0 TRANTO CBF[I] = 1 BY LAM_C;
      (* Failure of computer in Triad B *)
   IF PSF[I]=0 TRANTO CAF[I] = 1, CBF[I] = 1, PSF[I] = 1 BY LAM_PS;
      (* Power supply failure *)
ENDFOR;
```

This method of modeling each component individually can lead to a combinatorial explosion in the number of states in the generated model when the number of components becomes large. The semi-Markov model generated for this relatively simple example contains 21 states and 138 transitions. A different ASSIST description could be developed for this system by taking advantage of the symmetry that exists in the system to reduce the model size. However, that description would be more difficult to understand and verify. It is not unusual to encounter a trade-off between the size of the model and the simplicity of the rules for generating the model.

12.5. Failure Rate Dependencies

Consider a triad of processors in which the processors are protected from voltage surges by voltage regulators. The processors fail at rate LAMBDA_P. The system initially contains two voltage regulators. These voltage regulators fail at rate LAMBDA_R. Once both voltage regulators fail, the processors are also subject to failure due to voltage surges, which arrive at an exponential rate LAMBDA_V.

```
(* FAILURE RATE DEPENDENCIES *)

LAMBDA_P = 1E-5;          (* Permanent failure rate of processors *)
LAMBDA_V = 1E-2;          (* Arrival rate of damaging voltage surges *)
LAMBDA_R = 1E-3;          (* Failure rate of voltage regulators *)
```

```
SPACE = (NP: 0..3,        (* Number of active processors *)
         NFP: 0..3,       (* Number of failed active processors *)
         NR: 0..2);       (* Number of working voltage regulators *)
START = (3,0,2);          (* Start with 3 working processors, 2 regs. *)

DEATHIF 2 * NFP >= NP;    (* Voter defeated *)

  (* Processor failures *)
IF NP > NFP TRANTO NFP = NFP+1 BY (NP-NFP)*LAMBDA_P;

  (* Failure of processor due to voltage surge *)
IF (NR = 0) AND (NP > NFP) TRANTO NFP = NFP+1 BY (NP-NFP)*LAMBDA_V;

  (* Voltage regulator failures *)
IF NR > 0 TRANTO NR = NR-1 BY NR*LAMBDA_R;

  (* Reconfiguration *)
IF NFP > 0 TRANTO (NP-1,NFP-1,NR) BY <8.0E-5,3.0E-5>;
```

Because this model contains only one DEATHIF statement, all system failure probabilities will be grouped together. The addition of another DEATHIF statement placed in front of the existing one could capture the probability of both voltage regulators failing before the system failure condition is reached:

```
DEATHIF (NR=0) AND (2 * NFP >= NP);
```

This modification allows the user to better understand the failure modes of the system, but does not affect the total system failure probability.

12.6. Recovery Rate Dependencies

In this system, the speed of the recovery process is significantly affected by the number of active components. This recovery process is modeled by making the recovery rate a function of the state-space variable NP, which is the number of active processors.

```
(* RECOVERY RATE DEPENDENCIES *)

N_PROCS = 10;           (* Initial number of processors *)
LAMBDA_P = 8E-3;        (* Permanent failure rate of processors *)

SPACE = (NP: 0..N_PROCS,        (* Number of active processors *)
         NFP: 0..N_PROCS);      (* Number of failed active processors *)

START = (N_PROCS,0);

DEATHIF 2 * NFP >= NP;  (* Voter defeated *)

  (* Processor failures *)
IF NP > NFP TRANTO NFP = NFP+1 BY (NP-NFP)*LAMBDA_P;

  (* Reconfiguration where rate is a function of NP *)
IF NFP > 0 TRANTO (NP-1,NFP-1) BY <NP*1.0E-5 + 3.0E-5,NP*2.0E-6 + 1.0E-5>;
```

The ASSIST language gives the user great flexibility in describing the reconfiguration behavior of a system.

13. Multiple Triads With Pooled Spares

This section starts with a simple model of two triads that share a pool of cold spares. Next, models of systems with various reconfiguration concepts are introduced and spare failures are included. These models are then generalized to model more than two triads. Finally, the section concludes with a general discussion of how to model multiple competing recoveries.

13.1. Two Triads With Pooled Cold Spares

In this section, a system that consists of two triads, which operate independently but replace faulty processors from a common pool of spares will be modeled. When the pool of spares runs out, the triads continue operating with faulty processors and do not degrade to simplex. The system fails when either triad has two faulty processors. This failure can happen because a second fault occurs in a triad before the first faulty processor can be replaced by an available spare or because the supply of spares to replace the faulty processors has been exhausted. For this model, it is assumed that the spares do not fail while they are inactive.

To facilitate performing trade-off studies, the ASSIST `INPUT` statement will be used to define a constant `N_SPARES` to represent the initial number of spares in the system. The ASSIST program will query the user interactively for the value of this constant before generating the model.

Because the triads do not degrade to a simplex configuration, there is no need to track the current number of processors in a triad. Thus, the state of each triad can be represented by a single variable `NW`, which is the number of working processors. Similarly, the spares do not fail while they are inactive, so their state can be represented by a single variable `NS`, which is the number of spares available. Thus, the state space is

```
SPACE = (NW1, NW2, N_SPARES);
```

The full model description is

```
(*  TWO TRIADS WITH POOL OF SPARES *)

INPUT N_SPARES;          (* Number of spares *)
LAMBDA_P = 1E-4;         (* Failure rate of active processors *)
DELTA1 = 3.6E3;          (* Reconfiguration rate of triad 1 *)
DELTA2 = 6.3E3;          (* Reconfiguration rate of triad 2 *)

SPACE = (NW1,            (* Number of working processors in triad 1 *)
         NW2,            (* Number of working processors in triad 2 *)
         NS);            (* Number of spare processors *)

START = (3, 3, N_SPARES);

   (* Active processor failure *)
IF NW1 > 0 TRANTO NW1 = NW1-1 BY NW1*LAMBDA_P;
IF NW2 > 0 TRANTO NW2 = NW2-1 BY NW2*LAMBDA_P;

   (* Replace failed processor with working spare *)
IF (NW1 < 3) AND (NS > 0) TRANTO NW1 = NW1+1, NS = NS-1 BY FAST DELTA1;
IF (NW2 < 3) AND (NS > 0) TRANTO NW2 = NW2+1, NS = NS-1 BY FAST DELTA2;

DEATHIF NW1 < 2;        (* Two faults in triad 1 is system failure *)
DEATHIF NW2 < 2;        (* Two faults in triad 2 is system failure *)
```

The start state is `(3, 3, N_SPARES)`, which indicates that both triads have a full complement of working processors and the number of initial spares is `N_SPARES`. The first two `TRANTO` rules define the fault-arrival process in each triad. This process is modeled by decrementing either `NW1` or `NW2` depending upon which triad experiences the failure. The next two `TRANTO` rules define recovery by replacing the faulty processor with a spare. These rules are conditioned upon `NS > 0`, that is, if there are no spares, recovery cannot take place. The result of reconfiguration is the replacement of the faulty processor with a working processor (`NWx = NWx + 1` for triad `x`) and depletion of one spare from the pool (`NS = NS - 1`). The system fails whenever either triad experiences two or more coincident faults (`NW1 < 2 or NW2 < 2`).

This system has two different recovery processes—recovery in triad 1 and recovery in triad 2—that can potentially occur at the same time. Because this model was developed with the assumption that the

completion times for both recovery processes are exponentially distributed, the SURE keyword FAST was used. Thus, the SURE program will automatically calculate the conditional recovery rates wherever these two recovery processes compete. This feature was described in section 5.2.7. Modeling of multiple competing recovery processes that are not exponentially distributed and systems in which the competing recoveries are not independent are discussed in section 13.8.

13.2. Two Triads With Pooled Cold Spares Reducible to One Triad

The model given in section 13.2 can be modified easily to describe a system that can survive with only one triad. This strategy was used in the FTMP system (ref. 23). If spares are available, the system reconfigures by replacing a faulty processor with a spare. If no spares are available, the faulty triad is removed and its good processors are added to the spares pool. There is one exception, however. When only one triad is left, the system maintains the faulty triad until it can no longer out vote the faulty processor, that is, until the second fault arrives. This system is said to be reducible to one triad. This situation is very different from degradation. Note that it requires that the system be capable of surviving with only one triad of processing power, although it initially begins with significantly more parallel computational power in the form of multiple triads.

As before, this model assumes that the spares are cold, that is, they do not fail while inactive. The state space must be modified to include whether a triad is active or not. This modification is accomplished by setting NWx = 0 when triad x is inactive. The number of triads is maintained in a state-space variable NT. Although this variable is redundant because the number of active triads can be determined by looking at NW1 and NW2, the inclusion of this extra state-space variable greatly simplifies the ASSIST input description. Thus, the state space is

```
SPACE = (NW1,      (* Number of working processors in triad 1 *)
         NW2,      (* Number of working processors in triad 2 *)
         NT,       (* Number of active triads *)
         NS);      (* Number of spare processors *)
```

The initial state is (3, 3, 2, N_SPARES). The DEATHIF statement becomes

```
DEATHIF (NW1 = 1) OR (NW2 = 1);
```

Note that the statement DEATHIF (NW1 < 2) OR (NW2 < 2); is wrong. This statement would conflict with the strategy of setting NWx equal to 0 when triad x becomes inactive. Note also that the condition (NW1 = 0) AND (NW2 = 0) is also not included. This clause could be added, but it would not change the model because the last triad is never collapsed into spares. Thus, this condition can never be satisfied.

Next, two new constants OMEGA1 and OMEGA2 that define the rate at which triads are collapsed when no spares are available are defined:

```
OMEGA1 = 5.6E3;              (* Collapsing rate of triad 1 *)
OMEGA2 = 8.3E3;              (* Collapsing rate of triad 2 *)
```

The fault-arrival rules are the same as in the previous model. However, the reconfiguration specification must be altered. The rules for each triad x are

```
IF (NWx = 2) AND (NS > 0) TRANTO NWx = NWx+1, NS = NS-1
BY FAST DELTAx;
IF (NWx = 2) AND (NS = 0) AND (NT > 1) TRANTO NWx = 0, NS = NS+2, NT = NT-1
BY FAST OMEGAx;
```

The first rule defines reconfiguration by replacement with a spare. Thus, this rule is conditioned by (NS > 0). The second rule defines the collapsing of a triad when no spares are available, that is,

when `NS = 0`. Note that the condition `(NT > 1)` prevents the collapse of the last triad. The complete model is

```
(*  TWO TRIADS WITH POOL OF SPARES --> 1 TRIAD *)
INPUT N_SPARES;          (* Number of spares *)
LAMBDA1 = 1E-4;          (* Failure rate of active processors *)
LAMBDA2 = 1E-4;          (* Failure rate of active processors *)
DELTA1 = 3.6E3;          (* Reconfiguration rate of triad 1 *)
DELTA2 = 6.3E3;          (* Reconfiguration rate of triad 2 *)
OMEGA1 = 5.6E3;          (* Collapsing rate of triad 1 *)
OMEGA2 = 8.3E3;          (* Collapsing rate of triad 2 *)

SPACE = (NW1,            (* Number of working processors in triad 1 *)
         NW2,            (* Number of working processors in triad 2 *)
         NT,             (* Number of active triads *)
         NS);            (* Number of spare processors *)

START = (3, 3, 2, N_SPARES);

   (* Active processor failure *)
IF (NW1 > 0) TRANTO NW1 = NW1-1 BY NW1*LAMBDA1;
IF (NW2 > 0) TRANTO NW2 = NW2-1 BY NW2*LAMBDA2;

   (* Replace failed processor with working spare *)
IF (NW1 = 2) AND (NS > 0) TRANTO NW1 = NW1+1, NS = NS-1 BY FAST DELTA1;
IF (NW2 = 2) AND (NS > 0) TRANTO NW2 = NW2+1, NS = NS-1 BY FAST DELTA2;
IF (NW1 = 2) AND (NS = 0) AND (NT > 1) TRANTO NW1 = 0, NS = NS+2, NT = NT-1
     BY FAST OMEGA1;     (* Degrade to one triad only -- triad 2 *)
IF (NW2 = 2) AND (NS = 0) AND (NT > 1) TRANTO NW2 = 0, NS = NS+2, NT = NT-1
     BY FAST OMEGA2;     (* Degrade to one triad only -- triad 1*)

DEATHIF (NW1 = 1) OR (NW2 = 1);
```

13.3. Two Degradable Triads With Pooled Cold Spares

The system modeled in this section consists of two triads that can degrade to a simplex. However, unlike the previous example, this system requires the throughput of two processors. Therefore, the system does not degrade to one triad. Instead, when no spares are available, the system degrades the faulty triad into a simplex. The extra nonfaulty processor is added to the spares pool.

In this system, each of two triads can be degraded into a simplex. Therefore, it is necessary to add state-space variables that indicate whether the active configuration is a triad or simplex. Otherwise it is impossible to determine whether each state that satisfies the condition `NWx = 1` for triad `x` is a failed state (one good out of three) or an operational state (one good out of one). Thus, two state-space variables, `NC1` and `NC2`, are added to the model to indicate the total number of processors in the current configuration of each triad. If triad `x` still has three active processors, then `NCx = 3;` if triad `x` has already degraded to a simplex, then `NCx = 1`. The complete state space is

```
SPACE = (NC1,            (* Number of active processors in triad 1 *)
         NW1,            (* Number of working processors in triad 1 *)
         NC2,            (* Number of active processors in triad 2 *)
         NW2,            (* Number of working processors in triad 2 *)
         NS);            (* Number of spare processors *)
```

The initial state is `(3, 3, 3, 3, N_SPARES)` where `N_SPARES` represents the number of processors in the spares pool initially. The processor failure rules are the same as in previous models. As expected, the reconfiguration rules must be altered. These rules for each triad are

```
   IF (NWx < 3) AND (NCx = 3) AND (NS > 0) TRANTO NWx = NWx+1, NS = NS-1
        BY FAST DELTAx;    (* Replace failed processor with working spare *)

   IF (NWx < 3) AND (NS = 0) AND (NCx=3) TRANTO NCx = 1, NWx = 1, NS = NS+1
        BY FAST OMEGAx;    (* Degrade to simplex *)
```

where x represents triad 1 or triad 2. The first rule describes the replacement of a faulty processor in a triad with a spare. Note that the condition NCx = 3 has been added. Otherwise states with NWx = 1 and NCx = 1 (i.e., a good simplex processor) would erroneously have a recovery transition leaving them. The second rule describes the process of degrading a triad to a simplex. Note that this process is only done when no spares are available, that is, NS = 0. Also, the extra nonfaulty processor is returned to the spares pool, that is, NS = NS + 1. The DEATHIF conditions are

```
   DEATHIF 2*NWx <= NCx;
```

for each triad x. This condition restricts the operational states to only those where a majority of the processors are working. The complete specification is

```
(*  TWO DEGRADABLE TRIADS WITH A POOL OF SPARES *)

INPUT N_SPARES;          (* Number of spares *)
LAMBDA1 = 1E-4;          (* Failure rate of processor in triad 1 *)
LAMBDA2 = 1E-4;          (* Failure rate of processor in triad 2 *)
DELTA1 = 3.6E3;          (* Reconfiguration rate of triad 1 *)
DELTA2 = 6.3E3;          (* Reconfiguration rate of triad 2 *)
OMEGA1 = 5.6E3;          (* Reconfiguration rate of triad 1 *)
OMEGA2 = 8.3E3;          (* Reconfiguration rate of triad 2 *)

SPACE = (NC1,            (* Number of active processors in triad 1 *)
         NW1,            (* Number of working processors in triad 1 *)
         NC2,            (* Number of active processors in triad 2 *)
         NW2,            (* Number of working processors in triad 2 *)
         NS);            (* Number of spare processors *)

START = (3, 3, 3, 3, N_SPARES);

   (* Active processor failure *)
IF NW1 > 0 TRANTO NW1 = NW1-1 BY NW1*LAMBDA1;
IF NW2 > 0 TRANTO NW2 = NW2-1 BY NW2*LAMBDA2;

   (* Replace failed processor with working spare *)
IF (NW1 < 3) AND (NC1 = 3) AND (NS > 0) TRANTO NW1 = NW1+1, NS = NS-1
     BY FAST DELTA1;
IF (NW2 < 3) AND (NC2 = 3) AND (NS > 0) TRANTO NW2 = NW2+1, NS = NS-1
     BY FAST DELTA2;

   (* Degrade to simplex *)
IF (NW1 < 3) AND (NS = 0) AND (NC1=3) TRANTO NC1 = 1, NW1 = 1, NS = NS+1
     BY FAST OMEGA1;
IF (NW2 < 3) AND (NS = 0) AND (NC2=3) TRANTO NC2 = 1, NW2 = 1, NS = NS+1
     BY FAST OMEGA2;

DEATHIF 2*NW1 <= NC1;
DEATHIF 2*NW2 <= NC2;
```

All previous models have used the simplifying assumption that spare processors cannot fail until they are brought into the active configuration. While this assumption significantly simplifies the modeling, it is too optimistic an assumption for many systems, especially those with long mission times.

13.4. Two Degradable Triads With Pooled Warm Spares

The models presented in section 13.3 can be generalized by allowing the spares to fail while inactive. This generalization can be accomplished by adding a variable NWS to track the number of working spares. A rule for generating the transitions corresponding to the failure of the spares must be added

```
LAMBDA_S = 1E-5;        (* Failure rate of inactive warm spare *)
IF NWS > 0 TRANTO NWS = NWS - 1 BY NWS*LAMBDA_S;
```

In this system, the failure of a warm spare is not detectable. Thus, faulty spares can be brought into the active configuration by a recovery process. The recovery TRANTO rules must cover two cases:

1. Recovery with a working spare

2. Recovery with a faulty spare

Because NWS working spares and NS-NWS faulty spares remain, the probability of selecting a working spare is NWS/NS and the corresponding recovery rate is (NWS/NS)*DELTA1. Similarly, the rate of recovery with a faulty spare is ((NS-NWS)/NS)*DELTA1. Notice that the sum of these two rates are DELTA1, as one would expect.

Also the reconfiguration process must be generalized to include two distinct results:

1. A faulty warm spare is brought into the active configuration.

2. A working warm spare is brought into the active configuration.

Thus, the model presented in section 13.3 can be modified to describe a system of two degradable triads with a pool of warm spares:

```
(*  TWO DEGRADABLE TRIADS WITH A POOL OF WARM SPARES *)

INPUT N_SPARES;         (* Number of spares *)
LAMBDA1 = 1E-4;         (* Failure rate of processor in triad 1 *)
LAMBDA2 = 1E-4;         (* Failure rate of processor in triad 2 *)
LAMBDA_S = 1E-5;        (* Failure rate of spare processors *)
DELTA1 = 3.6E3;         (* Reconfiguration rate of triad 1 *)
DELTA2 = 6.3E3;         (* Reconfiguration rate of triad 2 *)
OMEGA1 = 5.6E3;         (* Reconfiguration rate of triad 1 *)
OMEGA2 = 8.3E3; (* Reconfiguration rate of triad 2 *)

SPACE = (NC1,           (* Number of active processors in triad 1 *)
         NW1,           (* Number of working processors in triad 1 *)
         NC2,           (* Number of active processors in triad 2 *)
         NW2,           (* Number of working processors in triad 2 *)
         NS,            (* Total number of warm spares *)
         NWS);          (* Number of working warm spares*)

START = (3, 3, 3, 3, N_SPARES, N_SPARES);

IF NW1 > 0 TRANTO NW1 = NW1-1 BY NW1*LAMBDA1;  (* Active processor failure *)
IF NW2 > 0 TRANTO NW2 = NW2-1 BY NW2*LAMBDA2;  (* Active processor failure *)
IF NWS > 0 TRANTO NWS = NWS-1 BY NWS*LAMBDA_S;  (* Warm spare failure *)

IF (NW1 < 3) AND (NC1 = 3) THEN        (* Triad 1 has a fault *)
   IF (NWS > 0)                        (* Replace with working spare *)
      TRANTO NW1 = NW1+1, NS = NS-1, NWS = NWS - 1
         BY FAST (NWS/NS)*DELTA1;
   IF (NS > NWS)                       (* Replace with failed spare *)
      TRANTO NS = NS-1 BY FAST ((NS-NWS)/NS)*DELTA1;
ENDIF;
```

```
IF (NW2 < 3) AND (NC2 = 3) THEN        (* Triad 2 has a fault *)
   IF (NWS > 0)                        (* Replace with working spare *)
      TRANTO NW2 = NW2+1, NS = NS-1, NWS = NWS - 1
          BY FAST (NWS/NS)*DELTA2;
   IF (NS > NWS)                       (* Replace with failed spare *)
      TRANTO NS = NS-1 BY FAST ((NS-NWS)/NS)*DELTA2;

ENDIF;

IF (NW1 < 3) AND (NS = 0) AND (NC1=3)           (* Degrade to simplex *)
      TRANTO NC1 = 1, NW1 = 1, NS = NS+1, NWS = NWS + 1 BY FAST OMEGA1;
IF (NW2 < 3) AND (NS = 0) AND (NC2=3)           (* Degrade to simplex *)
   TRANTO NC2 = 1, NW2 = 1, NS = NS+1, NWS = NWS + 1 BY FAST OMEGA2;

DEATHIF 2*NW1 <= NC1;
DEATHIF 2*NW2 <= NC2;
```

Four sections are devoted to recovery in this model. The first two sections cover the recovery of the two triads by using a spare. The last two sections deal with the situation where no spares are available and the triad is degraded to a simplex. Notice that the first two sections begin with an `IF` test of the form

```
IF (NWx < 3) AND (NCx = 3) THEN           (* Triad x has a fault *)
```

The second term of the `IF` test `(NCx = 3)` insures that the triad is still intact. The first term `(NWx < 3)` indicates that a failed processor exists in the triad. The last two recovery sections begin with an `IF` test of the form

```
IF (NWx < 3) AND (NS = 0) AND (NCx=3)     (* Degrade to simplex *)
```

The last term of the `IF` test `(NCx = 3)` insures that the triad is still intact. The first term `(NWx < 3)` indicates that a failed processor exists in the triad. The second term `(NS=0)` indicates that no spares are available.

If the system were designed so that the extra working processor of a collapsed triad were not added to the spares pool, the last two recovery `TRANTO` sections would be as follows:

```
IF (NW1 < 3) AND (NS = 0) AND (NC1=3)          (* Degrade to simplex *)
   TRANTO NC1 = 1, NW1 = 1 BY FAST OMEGA1;
IF (NW2 < 3) AND (NS = 0) AND (NC2=3)          (* Degrade to simplex *)
   TRANTO NC2 = 1, NW2 = 1 BY FAST OMEGA2;
```

with the `NS = NS+1, NWS = NWS + 1` phrases removed.

13.5. Multiple Nondegradable Triads With Pooled Cold Spares

This section demonstrates development of a generalized description that can be used to model an arbitrary number of triads. This modeling will be accomplished by creating a general specification that will work for any number of initial triads and having the ASSIST program prompt for a specific value to generate a specific model.

For simplicity, a system that is incapable of collapsing a triad into either a simplex or into spares will be investigated. Thus, this system fails when any triad fails. This first model will be simplified by assuming that cold spares cannot fail until they are brought into the active configuration. The complete generalized specification is

```
(*  MULTIPLE TRIADS WITH POOL OF COLD SPARES *)

INPUT N_TRIADS;              (* Number of triads initially *)
INPUT N_SPARES;              (* Number of spares *)
```

```
LAMBDA = 1E-4;              (* Failure rate of active processors *)
DELTA = 5.1E3;              (* Reconfiguration rate to switch in spare *)

SPACE = (NW: ARRAY[1..N_TRIADS] OF 0..3, (* Number working procs per triad *)
         NS);                             (* Number of spare processors *)

START = (N_TRIADS OF 3, N_SPARES);

FOR J = 1, N_TRIADS;

      (* Active processor failure *)
   IF NW[J] > 0 TRANTO NW[J] = NW[J]-1 BY NW[J]*LAMBDA;

      (* Replace failed processor with working spare *)
   IF (NW[J] < 3) AND (NS > 0) TRANTO NW[J] = NW[J]+1, NS = NS-1
      BY FAST DELTA^J;

   DEATHIF NW[J] < 2;        (* Two faults in a triad is system failure *)

ENDFOR;
```

The array state-space variable NW contains a value for each triad representing a count of the number of working processors in that triad. Similarly, the FOR loop, which terminates at the ENDFOR statement, defines for each triad

1. The active processor failures in that triad
2. The replacement of failed processors in that triad with spares from the pool
3. The conditions for that triad that result in system failure

To accommodate systems with different reconfiguration rates for different triads, the concatenation feature was used in the rate expression of the reconfiguration TRANTO statement. The rate expression BY FAST DELTA^J within the FOR loop results in a reconfiguration rate of FAST DELTA1 for triad 1 and FAST DELTA2 for triad 2. Unfortunately, because the number of triads is unknown until run time (N_TRIADS is specified with the INPUT statement), no way exists to assign values to these identifiers. This must be done by editing the output file or entering them at SURE run time. For simplicity, the rest of the models in this section will assume that all triads have the same reconfiguration rates.

13.6. Multiple Triads With Pooled Cold Spares Reducible to One

In this section, the simple model given in section 13.5 will be modified to allow degradation of triads. When no spares are available, each faulty triad is broken up and the nonfaulty processors are added to the spares pool. It is assumed that the system can operate with degraded performance with the throughput of only one processor. In other words, although the initial configuration consists of multiple triads, the system can still maintain its vital functions with only one triad remaining. Because triads will be broken up, a method of deciding whether a triad is active is needed. This method adds an array state-space variable NP to track the number of active processors in a triad. This variable will have the value of 3 for each triad initially and will be set to 0 for each triad when it is broken up. The array state-space variable NFP counts the number of failed processors active in each triad. The state-space variable NT is used to track how many triads are still in operation. This variable will always equal the number of nonzero entries in array NP. Thus, it is in some sense redundant. However, the specification of the TRANTO is simplified by including it in the SPACE statement.

The specification is

```
(*  REDUCEABLE MULTIPLE TRIADS WITH POOL OF COLD SPARES *)

INPUT N_TRIADS;             (* Number of triads initially *)
INPUT N_SPARES;             (* Number of spares *)
```

```
LAMBDA = 1E-4;              (* Failure rate of active processors *)
DELTA1 = 3.6E3;             (* Reconfiguration rate to switch in spare *)
DELTA2 = 5.1E3;             (* Reconfiguration rate to break up a triad *)

SPACE = (NP: ARRAY[1..N_TRIADS] OF 0..3,  (* Number of processors per triad *)
        NFP: ARRAY[1..N_TRIADS] OF 0..3, (* Num. failed active procs/triad *)
         NS,                (* Number of spare processors *)
         NT: 0..N_TRIADS);  (* Number of non-failed triads *)

START = (N_TRIADS OF 3, N_TRIADS OF 0, N_SPARES, N_TRIADS);

FOR J = 1, N_TRIADS;

   IF NP[J] > NFP[J] TRANTO NFP[J] = NFP[J]+1
        BY (NP[J]-NFP[J])*LAMBDA;  (* Active processor failure *)

   IF NFP[J] > 0 THEN
      IF NS > 0 THEN TRANTO NFP[J] = NFP[J]-1, NS = NS-1
            BY FAST DELTA1;
            (* Replace failed processor with working spare *)

      ELSE
       IF NT > 1 TRANTO NP[J]=0, NFP[J]=0, NS = NS + (NP[J]-NFP[J]), NT = NT-1
            BY FAST DELTA2;
            (* Break up a failed triad when no spares available *)
      ENDIF;
   ENDIF;

   DEATHIF 2 * NFP[J] >= NP[J] AND NP[J] > 0;
     (* Two faults in an active triad is system failure *)

ENDFOR;
```

As before, all TRANTO and DEATHIF statements are set inside of a FOR loop so that they are repeated for each triad. The first TRANTO statement defines failure of an active processor. When a triad has an active failed processor, the second TRANTO statement replaces that failed processor with one from the pool of spares. If no spares are available, then the faulty triad is broken up and its working processors are put into the spares pool. (This procedure assumes that the system can determine which processor has failed with 100-percent accuracy.) However, if the faulty triad is the last triad in the system (i.e., NT <= 1) then the triad is not broken up. This test makes the statement DEATHIF NT = 0 unnecessary. The last triad is allowed to continue operation with one faulty processor until another of its processors fails and defeats the voter. The single DEATHIF statement captures the occurrence of the second fault in the last triad as well as near-coincident faults in the other triads.

The following sequence of states represents a typical path through the model:

```
(3,3,3,0,0,0,1,3) -> (3,3,3,0,0,1,1,3) -> (3,3,3,0,0,0,0,3) ->
(3,3,3,1,0,0,0,3) -> (0,3,3,0,0,0,2,2) -> (0,3,3,0,1,0,2,2) ->
(0,3,3,0,0,0,1,2) -> (0,3,3,0,1,0,1,2) -> (0,3,3,0,0,0,0,2) ->
(0,3,3,0,1,0,0,2) -> (0,0,3,0,0,0,2,1) -> ...
```

13.7. Multiple Reducible Triads With Pooled Warm Spares

The model given in section 13.7 can be easily modified to include spare failures:

```
(*  REDUCIBLE MULTIPLE TRIADS WITH POOL OF WARM SPARES *)

INPUT N_TRIADS;             (* Number of triads initially *)
INPUT N_SPARES;             (* Number of spares *)
LAMBDA_P = 1E-4;            (* Failure rate of active processors *)
```

```
LAMBDA_S = 1E-5;             (* Failure rate of warm spare processors *)
DELTA1 = 3.6E3;              (* Reconfiguration rate to switch in spare *)
DELTA2 = 5.1E3;              (* Reconfiguration rate to break up a triad *)

SPACE = (NP: ARRAY[1..N_TRIADS] OF 0..3,  (* Number of processors per triad *)
        NFP: ARRAY[1..N_TRIADS] Of 0..3, (* Num. failed active procs/triad *)
         NS,                  (* Number of spare processors *)
         NFS,                (* Number of failed spare processors *)
         NT: 0..N_TRIADS);   (* Number of non-failed triads *)

START = (N_TRIADS OF 3, N_TRIADS OF 0, N_SPARES, 0, N_TRIADS);

IF NS > NFS TRANTO NFS = NFS+1 BY (NS-NFS)*LAMBDA_S;  (* Spare failure *)

FOR J = 1, N_TRIADS;

   IF NP[J] > NFP[J] TRANTO NFP[J] = NFP[J]+1
        BY (NP[J]-NFP[J])*LAMBDA_P;  (* Active processor failure *)

   IF NFP[J] > 0 THEN
      IF NS > 0 THEN
         IF NS > NFS TRANTO NFP[J] = NFP[J]-1, NS = NS-1
            BY FAST (1-(NFS/NS))*DELTA1;
            (* Replace failed processor with working spare *)

         IF NFS > 0 TRANTO NS = NS-1, NFS = NFS-1
            BY FAST (NFS/NS)*DELTA1;
            (* Replace failed processor with failed spare *)

      ELSE
         IF NT > 1 TRANTO NP[J]=0, NFP[J]=0, NS = NP[J]-NFP[J], NT = NT-1
            BY FAST DELTA2;
            (* Break up a failed triad when no spares available *)
      ENDIF;
   ENDIF;

   DEATHIF 2 * NFP[J] >= NP[J] AND NP[J] > 0;
     (* Two faults in an active triad is system failure *)

ENDFOR;
```

The additional state-space variable NFS is needed to track how many failed spares are in the spares pool. The failure of spares is defined by the first TRANTO statement. Note the placement of this statement outside of the FOR loop. If this statement were incorrectly placed inside of the FOR loop, it would be equivalent to having the spare failure rate multiplied by N_TRIADS. This model includes two TRANTO statements to define replacement of a failed processor with a spare. The first statement defines replacement of the failed processor with a working spare that is conditional on the existence of non-failed spares. The second statement defines replacement of a failed processor with a failed spare that is conditional upon the existence of failed spares.

13.8. Multiple Competing Recoveries

In the examples of multiple triads with pooled spares, the first example of a model containing states with multiple recovery processes that leave a single state was encountered. Multiple recovery processes occurs when multiple faults accumulate in different parts of the system that together do not cause system failure. The diagram in figure 41 illustrates this concept.

In this model, two triads are accumulating faults—the first at rate λ_1 and the second at rate λ_2. In state (2,2,7), both triads have a faulty processor active at the same time. This situation is not system failure because the two failures are in separately voted triads. Two recoveries are possible from this

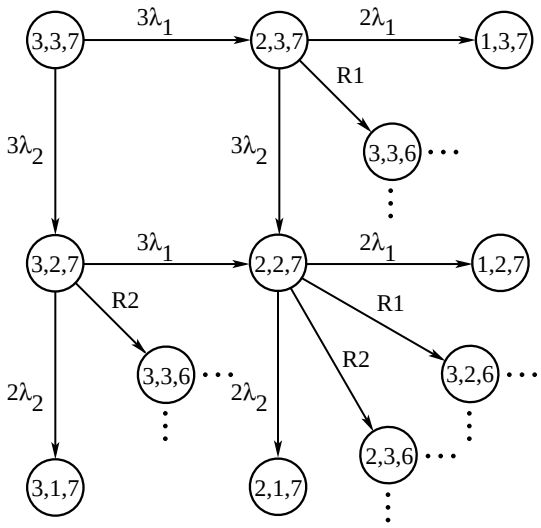


Figure 41. Model with multiple competing recoveries.

state—triad 1 recovers first then triad 2, or triad 2 recovers first then triad 1. Which case occurs depends upon how long each recovery takes.

In some systems, the recovery process may take longer when another recovery is also ongoing in the system. When the two recovery processes have no effect on each other in the system, the presence of competing recoveries still impacts the transition specification because the SURE program requires conditional means and conditional standard deviations for the competing recovery processes. Consider the simple case of two identical recovery distributions. On average, half of the time triad 1 will recover first, and half of the time triad 2 will recover first. The conditional mean recovery time is the mean of the minimum of the two competing recovery times. The SURE program also requires the specification of a third parameter—the transition probability. The transition probability is the probability that this transition will be traversed rather than one of the other fast transitions leaving this state. The sum of the transition probabilities given for all fast transitions leaving a state must equal one.

In all the preceeding examples of section 13, the recovery processes were assumed to be exponentially distributed, and the SURE FAST keyword was used to specify these transitions. As discussed in section 5.2.7, the FAST keyword causes the SURE program to automatically calculate the conditional rates from the unconditional fast exponential rates that are provided.

For systems with nonexponential recovery times or in which recovery times are affected by the presence of other competing recoveries, the problem of competing recoveries can be difficult to model accurately. How the system actually behaves in state (2,2,7) of the two-triad example depends upon the design of the redundancy management system. Many possibilities exist. To illustrate, three possible systems are discussed:

1. The system always repairs triad 1 first

2. The system repairs both triads at the same time

3. Two independent repair processes take place

The model for case 1 in which triad 1 is always repaired first, is shown in figure 42. Although the two recovery transitions no longer occur simultaneously, the recovery transition rates, R1_FIRST and R2_SECOND, may or may not have the same distribution as the noncompeting recovery transition rates, R1 and R2. This distribution depends on the system, and may be determined by analysis or experimentation.

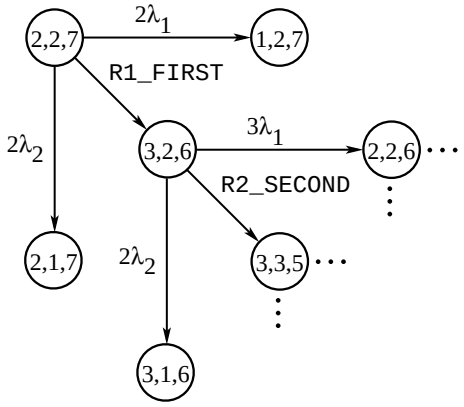


Figure 42. Triad 1 always repaired first.

The case 2 model of both triads being repaired at the same time is shown in figure 43. The mean and the standard deviation of the multiple recovery transition must be measured experimentally. This measurement can be accomplished by injecting two simultaneous faults and measuring the time to recovery completion.

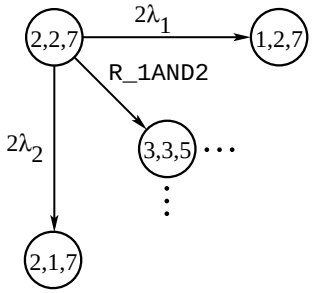


Figure 43. Both triads repaired simultaneously.

The third case, two independent repair processes, is shown in figure 44. If the two recoveries are truly independent and not competing for resources, the transition rates will be different from the non-competing rates because they are conditional upon winning the competition. The conditional means and standard deviations cannot be analytically determined from the unconditional recovery distributions in general; therefore, these four distributions must be measured experimentally.

14. Multiple Triads With Other Dependencies

In this section, the analysis of systems that consist of multiple triads is continued. However, other dependencies will be analyzed here.

14.1. Modeling Multiple Identical Triads

In the previous sections, specific triads that were faulty in the models were tracked. This tracking was necessary because different triads had different failure rates and different recovery rates. However, if all triads are identical, then a substantially simpler model can be developed that takes advantage of the extra symmetry. Because of the symmetry, the specific triad that was faulty need not be tracked. Instead, the number of triads with faults must be counted. The following set of attributes are sufficient to characterize this system:

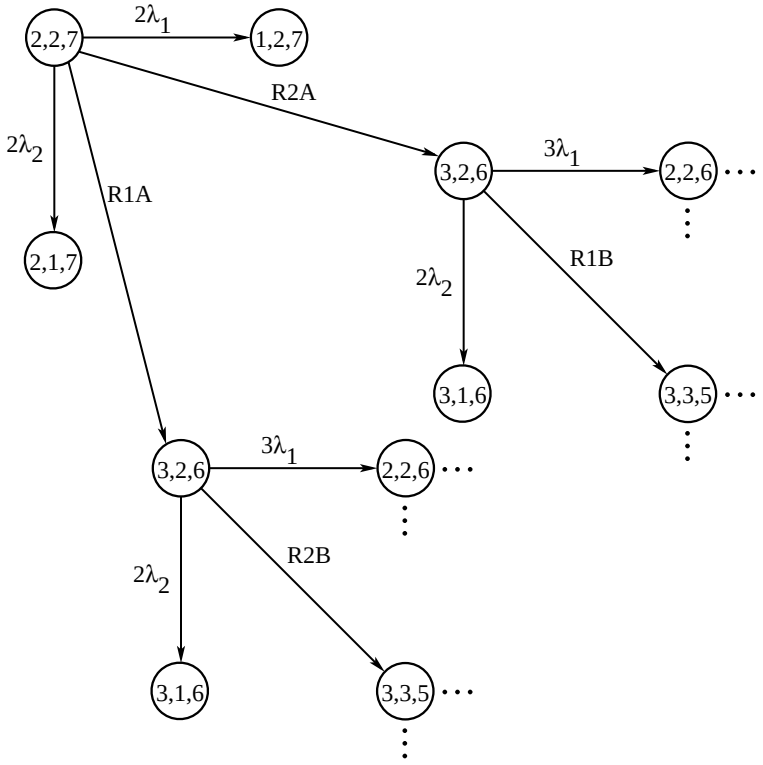


Figure 44. Two independent repair processes.

1. NAT, number of active triads
2. NTWF, number of triads with one fault
3. NS, number of spares available
4. NCF, one if and only if near-coincident failure
5. NI, number of triads initially
6. NSI, number of initial spares
7. MNT, minimum number of triads needed

If a processor in a triad fails, the NTWF variable is incremented. If a second processor in a triad fails, then the NCF variable is set to 1. Any state with NCF = 1 is a death state. The system replaces faulty processors by a spare. This replacement is represented by decrementing the NTWF variable by 1 and the number of spares variable NS by 1. If no spares are available, the system removes the faulty triad from the system. The remaining good processors are added to the spares pool. This addition is accomplished by decrementing the NAT variable by 1, decrementing the NTWF variable by 1, and incrementing the number of spares variable by 2. The complete ASSIST model is

```
LAMBDA = 1e-4;
DELTA1 = 1e4;
DELTA2 = 1e4;
INPUT NI;       (* Number of triads initially *)
INPUT NSI;      (* Number of spares initially *)
INPUT MNT;      (* Minimum number triads needed *)
```

```
SPACE = (NAT: 0..NI,    (* Number of Active Triads *)
         NTWF: 0..NI,   (* Number of Triads with 1 fault *)
         NS: 0..NSI,    (* Number of Available Spares *)
         NCF: 0..1);    (* 1 iff near-coincident failure *)

START = (NI,0,NSI,0);

IF NAT > 0 THEN
   IF NAT > NTWF TRANTO NTWF = NTWF + 1 BY 3*(NAT-NTWF)*LAMBDA;
   IF NTWF > 0 TRANTO NCF = 1 BY 2*(NTWF)*LAMBDA;
   IF (NTWF > 0) THEN
      IF NS > 0 THEN
         TRANTO NTWF = NTWF - 1, NS = NS - 1 BY FAST NTWF*DELTA2;
      ELSE
         IF NAT > MNT TRANTO NAT = NAT - 1, NTWF = NTWF - 1, NS = NS + 2
                                   BY FAST NTWF*DELTA1;
      ENDIF;
   ENDIF;
ENDIF;
DEATHIF (NAT=MNT) AND NCF=1;
DEATHIF NCF = 1;
```

14.2. Multiple Triad Systems With Limited Ability to Handle Multiple Faults

The example considered in this section is similar to (and motivated by) a system analyzed by Alan White of the Assessment Technology Branch at Langley Research Center. The system consists of N triads. If a processor in a triad fails and spares are available, the system repairs the triad with a spare. If no spares are available, the system removes the faulty triad from the configuration and adds the good processors to the spares pool. System failure occurs when a triad has two faulty processors (i.e., a second processor fails before it can be repaired or removed from the system) or when not enough triads remain to run the workload (exhaustion of parts). The system has difficulty diagnosing which processors are faulty when more than one triad has a faulty processor. Therefore, in this situation, the system does not reconfigure. For simplicity, the system is assumed to never misdiagnose a faulty processor and know when it has more than one triad with a faulty processor.

14.2.1. Two triads. In this section, a two-triad configuration will be modeled, that is, a system with two initial triads that can still operate with only one triad. The complete model is shown in figure 45.

Initially the system has two good triads and no spares. Thus, the first transition is from state (3) $\rightarrow$ state (4) with rate 6λ. After the system is in state (4) several events can happen. Another processor in the same triad could fail and cause system failure (i.e., enter state (1)). This failure is a near-coincident failure that occurs at rate 2λ. Alternatively, the system could recover from the first fault and go to state (6). One option remains; a processor could fail in the other triad. This causes the system to enter state (5) and occurs at rate 3λ. The rest of the model is clear if one keeps in mind that the two good processors of the removed triad are made spares. Thus, the recovery transitions from state (7) $\rightarrow$ state (8) and from state (9) $\rightarrow$ state (10) replace a faulty processor with a spare. While not active, the failure rate of the spares is assumed to be 0 to simplify the model.

One interesting question to explore is whether the trimming of all failure transitions from a recovery state that does not immediately lead to a death state is a good approximation. Such an approach would lead to the model shown in figure 46. In this model, the transition from state (4) $\rightarrow$ state (5) has been removed.

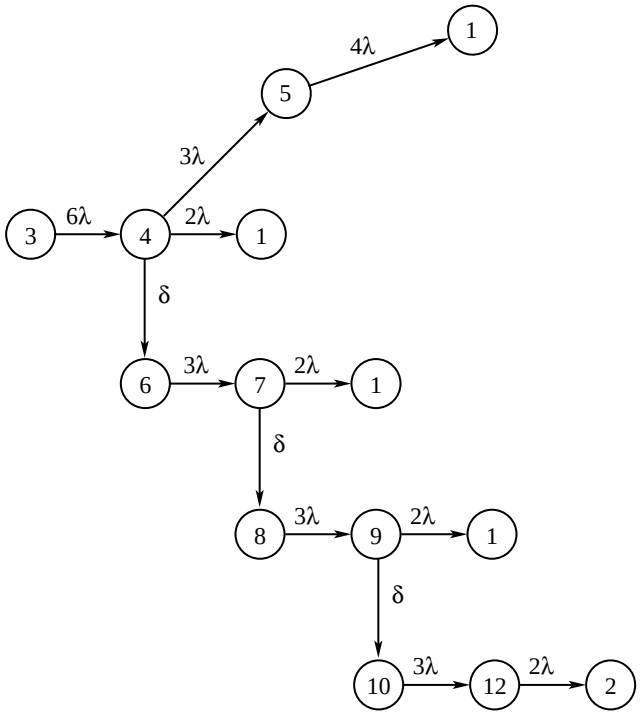


Figure 45. Full model of two triads.

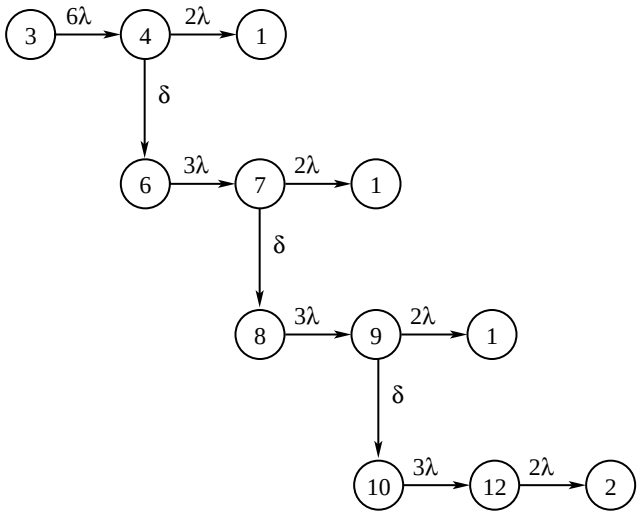


Figure 46. Trimmed model of two triads.

The probability of system failure after 1000 hr is given in table 2. For this system, the difference is clearly negligible. It is important that a bound on the error due to removing these transitions be calculated. In the next section, an example of the contribution of the removed states is shown to be significant.

14.2.2. N triads. In this section, a system consisting of `NI` triads will be analyzed. The larger `NI` becomes, the larger the number of transitions removed by the strategy described in the previous section becomes. Consider a system with 10 triads. A portion of the 175-state model that begins with 10 triads is shown in figure 47.

Table 2. P_f Estimated by PAWS for State (2,1) at 1000 hr, $\lambda = 10^{-4}$/hr, and $\delta = 10^4$/hr

Model	P_f
Full model[a]	2.03968×10^{-5}
Trimmed model[b]	2.03942×10^{-5}

[a]See figure 45.
[b]See figure 46.

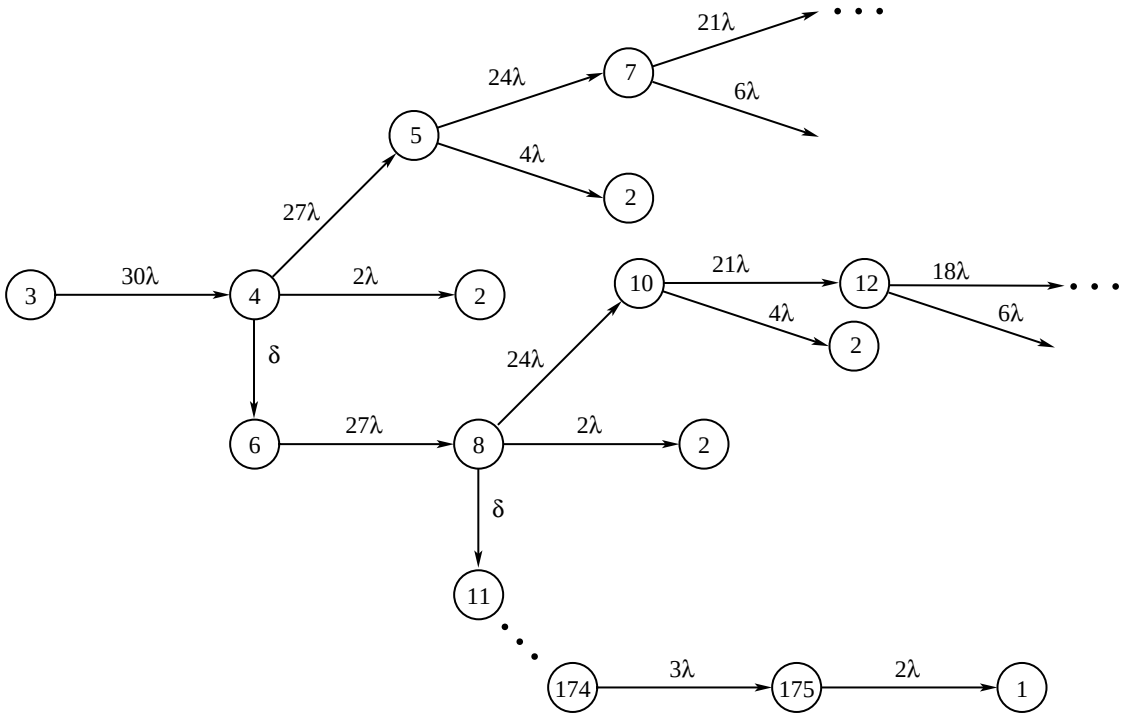


Figure 47. Full model of N triads.

A complete description of the model can be given with the ASSIST language:

```
LAMBDA = 1e-4;  (* processors failure rate *)
DELTA1 = 1e4;   (* reconfiguration by collapsing a triad *)
DELTA2 = 1e4;   (* reconfiguration by replacing with a spare *)
INPUT NI;       (* Number of triads initially *)
INPUT MNT;      (* Minimum number triads needed *)
INPUT MAXNTF;   (* Maximum number of faults handled *)
TIME = 1000;

SPACE = (NAT: 0..NI,    (* Number of Active Triads *)
         NTWF: 0..NI,   (* Number of Triads with 1 fault *)
         NS: 0..NI*3,   (* Number of Available Spares *)
         NCF: 0..1);    (* 1 iff near-coincident failure *)

START = (NI,0,0,0);

IF NAT > 0 THEN
   IF NAT > NTWF TRANTO NTWF = NTWF + 1 BY 3*(NAT-NTWF)*LAMBDA;
```

```
   IF NTWF > 0 TRANTO NCF = 1 BY 2*(NTWF)*LAMBDA;
   IF (NTWF > 0) AND (NTWF <= MAXNTF) THEN
      IF NS > 0 THEN
        TRANTO NTWF = NTWF - 1, NS = NS - 1 BY FAST NTWF*DELTA2;
      ELSE
        IF NAT > MNT TRANTO NAT = NAT - 1, NTWF = NTWF - 1, NS = NS + 2
                                    BY FAST NTWF*DELTA1;
      ENDIF;
   ENDIF;
ENDIF;
DEATHIF (NAT=MNT) AND NCF=1;
DEATHIF NCF = 1;
```

This model differs from the N independent identical triads modeled in section 14.1 by the use of an additional test on the reconfiguration TRANTO rule:

```
   IF (NTWF > 0) AND (NTWF <= MAXNTF) THEN
```

This rule is only enabled when (NTWF <= MAXNTF), that is, when the number of triads with faults is less than the limit that the operating system can handle MAXNTF.

The model generated for NI = 2 follows:

```
LAMBDA = 1E-4;
DELTA1 = 1E4;
DELTA2 = 1E4;
NI = 2;
MNT = 1;
MAXNTF = 99;
TIME = 1000;
     3(* 2,0,0,0 *),        4(* 2,1,0,0 *)       = 3*(2-0)*LAMBDA;
     4(* 2,1,0,0 *),        5(* 1,0,2,0 *)       = FAST 1*DELTA1;
     4(* 2,1,0,0 *),        6(* 2,2,0,0 *)       = 3*(2-1)*LAMBDA;
     4(* 2,1,0,0 *),        2(* 2,1,0,1 DEATH  *)= 2*(1)*LAMBDA;
     5(* 1,0,2,0 *),        7(* 1,1,2,0 *)       = 3*(1-0)*LAMBDA;
     6(* 2,2,0,0 *),        7(* 1,1,2,0 *)       = FAST 2*DELTA1;
     6(* 2,2,0,0 *),        2(* 2,2,0,1 DEATH  *)= 2*(2)*LAMBDA;
     7(* 1,1,2,0 *),        8(* 1,0,1,0 *)       = FAST 1*DELTA2;
     7(* 1,1,2,0 *),        1(* 1,1,2,1 DEATH  *)= 2*(1)*LAMBDA;
     8(* 1,0,1,0 *),        9(* 1,1,1,0 *)       = 3*(1-0)*LAMBDA;
     9(* 1,1,1,0 *),       10(* 1,0,0,0 *)       = FAST 1*DELTA2;
     9(* 1,1,1,0 *),        1(* 1,1,1,1 DEATH  *)= 2*(1)*LAMBDA;
    10(* 1,0,0,0 *),       11(* 1,1,0,0 *)       = 3*(1-0)*LAMBDA;
    11(* 1,1,0,0 *),        1(* 1,1,0,1 DEATH  *)= 2*(1)*LAMBDA;
(* NUMBER OF STATES IN MODEL = 11 *)
(* NUMBER OF TRANSITIONS IN MODEL = 14 *)
(* 5 DEATH STATES AGGREGATED INTO STATES 1 - 2 *)
```

The trimmed model of N triads analyzed can be generated by using the following ASSIST input file:

```
LAMBDA = 1e-4;
DELTA1 = 1e4;
DELTA2 = 1e4;
INPUT NI,MNT;    (* Number triads initially, number needed *)
```

```
TIME = 1000;

SPACE = (NAT: 0..NI,    (* Number of Active Triads *)
         NTWF: 0..NI,   (* Number of Triads with 1 fault *)
         NS: 0..NI*3,   (* Number of Available Spares *)
         NCF: 0..1);    (* 1 iff near-coincident failure *)

START = (NI,0,0,0);

IF NAT> 0 THEN
   IF (NAT > NTWF) AND (NTWF=0) TRANTO NTWF = NTWF + 1 BY 3*
      (NAT-NTWF)*LAMBDA;
   IF NTWF > 0 TRANTO NCF = 1 BY 2*(NTWF)*LAMBDA;
   IF (NTWF > 0) AND (NTWF <= MAXNTF) THEN
      IF NS > 0 THEN
         TRANTO NTWF = NTWF - 1, NS = NS - 1 BY FAST NTWF*DELTA2;
      ELSE
         IF NAT > MNT TRANTO NAT = NAT - 1, NTWF = NTWF - 1, NS = NS + 2
                                 BY FAST NTWF*DELTA1;
      ENDIF;
   ENDIF;
ENDIF;
DEATHIF (NAT=MNT) AND NCF=1;
DEATHIF NCF = 1;
```

This model differs from the previous model in this section by the use of an additional test on the fault-arrival `TRANTO` rule:

```
IF (NAT > NTWF) AND (NTWF=0) TRANTO NTWF = NTWF + 1 BY 3*(NAT-NTWF)*LAMBDA;
```

Note, this rule only applies when `NTWF = 0`, that is, when the number of triads with faults is 0. As before, the trimmed model does not have any failure transitions exiting from a recovery state that does not end in a failure state. The results for a system with `NI = 10` and `MNT = 1` are shown in table 3.

Table 3. P_f Estimated by PAWS for One Triad With Ten Initial Spares at 1000 hr, $\lambda = 10^{-4}$/hr, and $\delta = 10^{4}$/hr

Model	P_f
Full model	2.15×10^{-7}
Trimmed model	5.53×10^{-8}

Thus, for this model, the trimming method yields a result that is significantly nonconservative (the exact value is four times larger than the trimmed value). The nonconservatism grows as the mission time is increased. The results for a 2000 hr mission are given in table 4.

Table 4. P_f Estimated by PAWS for State (10,1) at 2000 hr, $\lambda = 10^{-4}$/hr, and $\delta = 10^{4}$/hr

Model	P_f
Full model	6.62×10^{-7}
Trimmed model	1.05×10^{-7}

The error resulting from the trimming method can be shown to be negligible for many systems. Thus, this technique can be used successfully if the probability contribution from all removed transitions is assured to be insignificant. This insignificance can be determined by performing some additional hand calculations (refs. 24 and 25) or by using the ASSIST implementation of this technique discussed in section 14.3. Clearly, optimistic models such as this one must be used with caution. Fortunately, the ASSIST program performs trimming in a manner that guarantees a conservative result.

14.3. ASSIST Trimming

The trimming method implemented in ASSIST for reducing the size of semi-Markov reliability models was developed by Allan White and Daniel Palumbo of Langley Research Center. The details of this trimming method and the theorem establishing that this method is conservative are given in references 24 and 25. Either of these references can be used to determine the characteristics of systems for which this trimming method is guaranteed to be conservative. Not all systems can be trimmed. However, if used correctly, the ASSIST implementation of trimming guarantees that the result is conservative.

In the implementation of ASSIST, the user has three choices. First, the default is no trimming `(TRIM = 0;)`. The second choice is conservative trimming of noncritically coupled failure transitions `(TRIM = 1;)`. In this mode, the model is altered so that each state in which the system is experiencing a recovery has all failure transitions from it that do not go immediately to a death state go to an intermediate trim state. That intermediate trim state transitions to a final state at rate `TRIMOMEGA`. This mode is invoked by including the statement `TRIM = 1 WITH real` where `real` is some real number. For example,

```
TRIM = 1 WITH 6e-4;
```

The expression `WITH real` is optional. If this expression is not included, the user will be prompted for the value. In that case, the user is requested to type in the value followed by a semicolon. A constant in ASSIST, `TRIMOMEGA`, will then be set to whatever value was specified by the user. Proper setting of `TRIMOMEGA` is discussed below.

The justification for trimming `(TRIM = 1)` is as follows. In a typical reconfigurable system, when a component fails, the system goes into a recovery state in which it is vulnerable to near-coincident failures until that failed component is removed from the active configuration. While in that recovery state, a component that is not critically coupled could fail and the system could go into a state in which it is recovering from two simultaneous, but noncritically coupled, component failures.

Because of the combinatorics inherent in semi-Markov modeling, accurately modeling these highly improbable states and transitions adds significant size and complexity to the model. Trimming allows the user to automatically replace these complex parts of the model with a few simple transitions that are conservative approximations. For each recovery state, all failure (slow) transitions that do not go directly to a death state, go to an intermediate state. Because this intermediate state represents the system experiencing two noncritically coupled component failures, an additional component failure is assumed to cause system failure. For simplicity, any component failure from this intermediate state is conservatively assumed to lead to immediate system failure. Thus, a single transition leaves this state and goes to an absorbing state.The rate for this transition `TRIMOMEGA` must be conservatively set to the sum of all component failure rates in the system. Trimming will not be valid for a system in which some recovery (fast) transitions from the trimmed states might lead to system failure because these behaviors will not be modeled. Also, `TRIMOMEGA` must be set to conservatively cover the sum of all component failure rates, even if these rates change over the model. For systems with unusual behaviors, see reference 25 to determine if this method is valid for a particular system.

The third choice is similar to `TRIM = 1`; however, extra transitions are added. In particular, from each prune state an additional transition at rate `TRIMOMEGA` is generated. This mode, `TRIM = 2` is invoked similarly to the case of `TRIM = 1`, with `TRIMOMEGA` specified in the same manner.

The justification for the method of `TRIM = 2` is that each prune state of the model is not actually a death state of the system, but a conservative approximation. Another failure of some component must occur before system failure can occur. The feature of `TRIM = 2` simply includes this extra failure transition before pruning the model. For `TRIM = 2` to be guaranteed conservative, the same assumptions must hold as for `TRIM = 1`. Namely, no recovery (fast) transitions can lead directly to system failure and `TRIMOMEGA` must be greater than or equal to the sum of the failure rates for all components still in the system when any prune state is reached.

14.4. Trimming Example

Trimming with a model of a system of up to three independent triads, where the user is prompted for the number of triads will be illustrated. For convenience, array state-space variables will be used to model the failure behavior of the triads, and the concatenation feature will be used to allow the triads to have different failure rates. Although the user is prompted for the number of triads, only three or fewer triads can be modeled because failure and recovery rates have only been defined for the first three triads. This model is, of course, easily expandable to accommodate a larger number of triads.

```
INPUT N_TRIADS;       (* Number of triads initially *)

LAMBDA1 = 1E-4;       (* Failure rate of active processors in 1 *)
DELTA1 = 3.6E3;       (* Reconfiguration rate in 1 *)
LAMBDA2 = 1E-4;       (* Failure rate of active processors in 2 *)
DELTA2 = 3.6E3;       (* Reconfiguration rate in 2 *)
LAMBDA3 = 1E-4;       (* Failure rate of active processors in 3 *)
DELTA3 = 3.6E3;       (* Reconfiguration rate in 3 *)

SPACE = (NP: ARRAY[1..N_TRIADS] OF 0..3,  (* Num. active processors in triad *)
         NFP: ARRAY[1..N_TRIADS] Of 0..3); (* Num. failed active procs *)

START = (N_TRIADS OF 3, N_TRIADS OF 0);

FOR J = 1, N_TRIADS;
   IF NP[J] > NFP[J] TRANTO NFP[J] = NFP[J]+1
        BY (NP[J]-NFP[J])*LAMBDA^J;  (* Active processor failure *)

   IF NFP[J] > 0 TRANTO
        NP[J]=1, NFP[J]=0 BY FAST DELTA^J;

   DEATHIF 2 * NFP[J] >= NP[J];
     (* Two faults in an active triad or simplex with a fault *)
ENDFOR;
```

Two state-space variables exist:

1. `NP[i]`, number of active processors in subsystem *i*

2. `NFP[i]`, number of faulty processors in triad *i*

Because reconfiguration collapses a triad to a simplex, `NP[i]` must be either 3 or 1. Processor failure in triad *i* results in the increment of `NFP[i]`. The `DEATHIF` condition covers both the triplex and simplex situation.

The generated model for two triads is

```
N_TRIADS = 2;
LAMBDA1 = 1E-4;
DELTA1 = 3.6E3;
```

```
LAMBDA2 = 1E-4;
DELTA2 = 3.6E3;
LAMBDA3 = 1E-4;
DELTA3 = 3.6E3;
     2(* 3,3,0,0 *),     3(* 3,3,1,0 *)        = (3-0)*LAMBDA1;
     2(* 3,3,0,0 *),     4(* 3,3,0,1 *)        = (3-0)*LAMBDA2;
     3(* 3,3,1,0 *),     5(* 1,3,0,0 *)        = FAST DELTA1;
     3(* 3,3,1,0 *),     1(* 3,3,2,0 DEATH  *) = (3-1)*LAMBDA1;
     3(* 3,3,1,0 *),     6(* 3,3,1,1 *)        = (3-0)*LAMBDA2;
     4(* 3,3,0,1 *),     7(* 3,1,0,0 *)        = FAST DELTA2;
     4(* 3,3,0,1 *),     6(* 3,3,1,1 *)        = (3-0)*LAMBDA1;
     4(* 3,3,0,1 *),     1(* 3,3,0,2 DEATH  *) = (3-1)*LAMBDA2;
     5(* 1,3,0,0 *),     1(* 1,3,1,0 DEATH  *) = (1-0)*LAMBDA1;
     5(* 1,3,0,0 *),     8(* 1,3,0,1 *)        = (3-0)*LAMBDA2;
     6(* 3,3,1,1 *),     8(* 1,3,0,1 *)        = FAST DELTA1;
     6(* 3,3,1,1 *),     9(* 3,1,1,0 *)        = FAST DELTA2;
     6(* 3,3,1,1 *),     1(* 3,3,2,1 DEATH  *) = (3-1)*LAMBDA1;
     6(* 3,3,1,1 *),     1(* 3,3,1,2 DEATH  *) = (3-1)*LAMBDA2;
     7(* 3,1,0,0 *),     9(* 3,1,1,0 *)        = (3-0)*LAMBDA1;
     7(* 3,1,0,0 *),     1(* 3,1,0,1 DEATH  *) = (1-0)*LAMBDA2;
     8(* 1,3,0,1 *),    10(* 1,1,0,0 *)        = FAST DELTA2;
     8(* 1,3,0,1 *),     1(* 1,3,1,1 DEATH  *) = (1-0)*LAMBDA1;
     8(* 1,3,0,1 *),     1(* 1,3,0,2 DEATH  *) = (3-1)*LAMBDA2;
     9(* 3,1,1,0 *),    10(* 1,1,0,0 *)        = FAST DELTA1;
     9(* 3,1,1,0 *),     1(* 3,1,2,0 DEATH  *) = (3-1)*LAMBDA1;
     9(* 3,1,1,0 *),     1(* 3,1,1,1 DEATH  *) = (1-0)*LAMBDA2;
    10(* 1,1,0,0 *),     1(* 1,1,1,0 DEATH  *) = (1-0)*LAMBDA1;
    10(* 1,1,0,0 *),     1(* 1,1,0,1 DEATH  *) = (1-0)*LAMBDA2;

(* NUMBER OF STATES IN MODEL = 10 *)
(* NUMBER OF TRANSITIONS IN MODEL = 24 *)
(* 12 DEATH STATES AGGREGATED INTO STATE 1 *)
```

This model is shown in figure 48.

To use trimming, the following two lines are added:

```
MAX_LAMBDA = 1E-4;
TRIM=1 WITH 3*N_TRIADS*MAX_LAMBDA;
```

The resulting ASSIST model is

```
INPUT N_TRIADS;       (* Number of triads initially *)
LAMBDA1 = 1E-4;       (* Failure rate of active processors in 1 *)
DELTA1 = 3.6E3;       (* Reconfiguration rate in 1 *)
LAMBDA2 = 1E-4;       (* Failure rate of active processors in 2 *)
DELTA2 = 3.6E3;       (* Reconfiguration rate in 2 *)
LAMBDA3 = 1E-4;       (* Failure rate of active processors in 3 *)
DELTA3 = 3.6E3;       (* Reconfiguration rate in 3 *)

MAX_LAMBDA = 1E-4

TRIM=1 WITH 3*N_TRIADS*MAX_LAMBDA;

SPACE = (NP: ARRAY[1..N_TRIADS] OF 0..3,  (* Num. active processors in triad *)
         NFP: ARRAY[1..N_TRIADS] Of 0..3); (* Num. failed active procs *)

START = (N_TRIADS OF 3, N_TRIADS OF 0);

FOR J = 1, N_TRIADS;
   IF NP[J] > NFP[J] TRANTO NFP[J] = NFP[J]+1
     BY (NP[J]-NFP[J])*LAMBDA^J;  (* Active processor failure *)
```

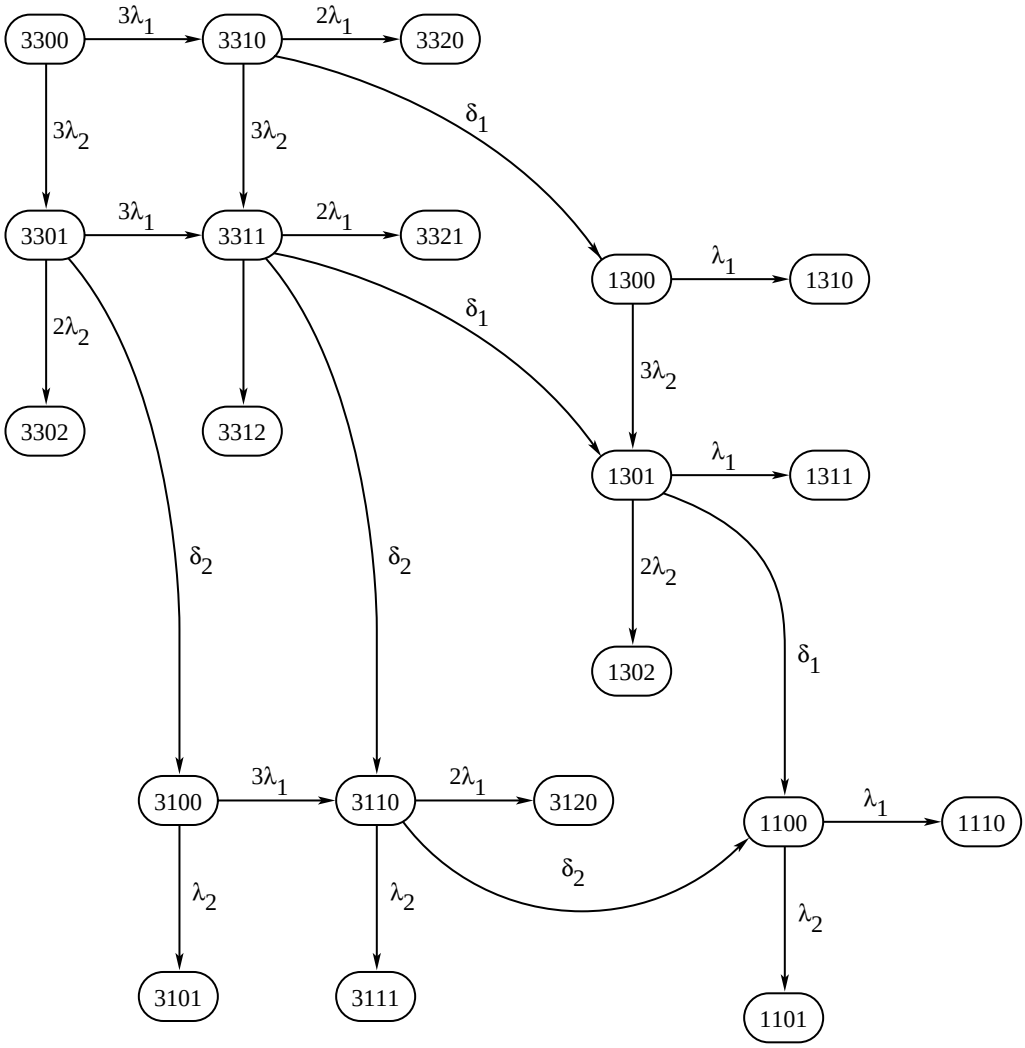


Figure 48. Two triads.

```
  IF NFP[J] > 0 TRANTO
     NP[J]=1, NFP[J]=0 BY FAST DELTA^J;

  DEATHIF 2 * NFP[J] >= NP[J];
    (* Two faults in an active triad or simplex with a fault *)
ENDFOR;
```

The following model is generated for `N_TRIADS = 2`:

```
N_TRIADS = 2;
LAMBDA1 = 1E-4;
DELTA1 = 3.6E3;
LAMBDA2 = 1E-4;
DELTA2 = 3.6E3;
LAMBDA3 = 1E-4;
DELTA3 = 3.6E3;
MAX_LAMBDA = 1E-4;
TRIMOMEGA = 3*N_TRIADS*MAX_LAMBDA;
PRUNESTATES = 2;
```

```
   4(* 3,3,0,0 *),     5(* 3,3,1,0 *)        = (3-0)*LAMBDA1;
   4(* 3,3,0,0 *),     6(* 3,3,0,1 *)        = (3-0)*LAMBDA2;
   5(* 3,3,1,0 *),     7(* 1,3,0,0 *)        = FAST DELTA1;
   5(* 3,3,1,0 *),     1(* 3,3,2,0 DEATH  *) = (3-1)*LAMBDA1;
   5(* 3,3,1,0 *),     3(* 3,3,1,1 TRIM   *) = (3-0)*LAMBDA2;
   6(* 3,3,0,1 *),     8(* 3,1,0,0 *)        = FAST DELTA2;
   6(* 3,3,0,1 *),     3(* 3,3,1,1 TRIM   *) = (3-0)*LAMBDA1;
   6(* 3,3,0,1 *),     1(* 3,3,0,2 DEATH  *) = (3-1)*LAMBDA2;
   7(* 1,3,0,0 *),     1(* 1,3,1,0 DEATH  *) = (1-0)*LAMBDA1;
   7(* 1,3,0,0 *),     9(* 1,3,0,1 *)        = (3-0)*LAMBDA2;
   8(* 3,1,0,0 *),    10(* 3,1,1,0 *)        = (3-0)*LAMBDA1;
   8(* 3,1,0,0 *),     1(* 3,1,0,1 DEATH  *) = (1-0)*LAMBDA2;
   9(* 1,3,0,1 *),    11(* 1,1,0,0 *)        = FAST DELTA2;
   9(* 1,3,0,1 *),     1(* 1,3,1,1 DEATH  *) = (1-0)*LAMBDA1;
   9(* 1,3,0,1 *),     1(* 1,3,0,2 DEATH  *) = (3-1)*LAMBDA2;
  10(* 3,1,1,0 *),    11(* 1,1,0,0 *)        = FAST DELTA1;
  10(* 3,1,1,0 *),     1(* 3,1,2,0 DEATH  *) = (3-1)*LAMBDA1;
  10(* 3,1,1,0 *),     1(* 3,1,1,1 DEATH  *) = (1-0)*LAMBDA2;
  11(* 1,1,0,0 *),     1(* 1,1,1,0 DEATH  *) = (1-0)*LAMBDA1;
  11(* 1,1,0,0 *),     1(* 1,1,0,1 DEATH  *) = (1-0)*LAMBDA2;
   3(*  TRIM   *),     2(*  TRIM.D *) = TRIMOMEGA;
(* NUMBER OF STATES IN MODEL = 11 *)
(* NUMBER OF TRANSITIONS IN MODEL = 21 *)
(* 10 DEATH STATES AGGREGATED INTO STATE 1 *)
(* 2 TRIMMED TRANSITIONS AGGREGATED INTO STATE 3 *)
```

This model is displayed in figure 49.

The result of solving the two-triad model with and without trimming follows:

```
air58% sure

 SURE V7.9.8   NASA Langley Research Center

 1? read0 nt

      0.03 SECS. TO READ MODEL FILE
41? list=2
42? run

MODEL FILE = nt.mod                    SURE V7.9.8 15 Apr 94  10:03:07

DEATHSTATE    LOWERBOUND    UPPERBOUND    COMMENTS              RUN #1
----------   -----------   -----------    --------------------------
     1       2.98539e-06   3.00633e-06

TOTAL        2.98539e-06   3.00633e-06

 32 PATH(S) TO DEATH STATES

0.017 SECS. CPU TIME UTILIZED
43? read0 nt-trim

      0.03 SECS. TO READ MODEL FILE
84? list=2
85? run
```

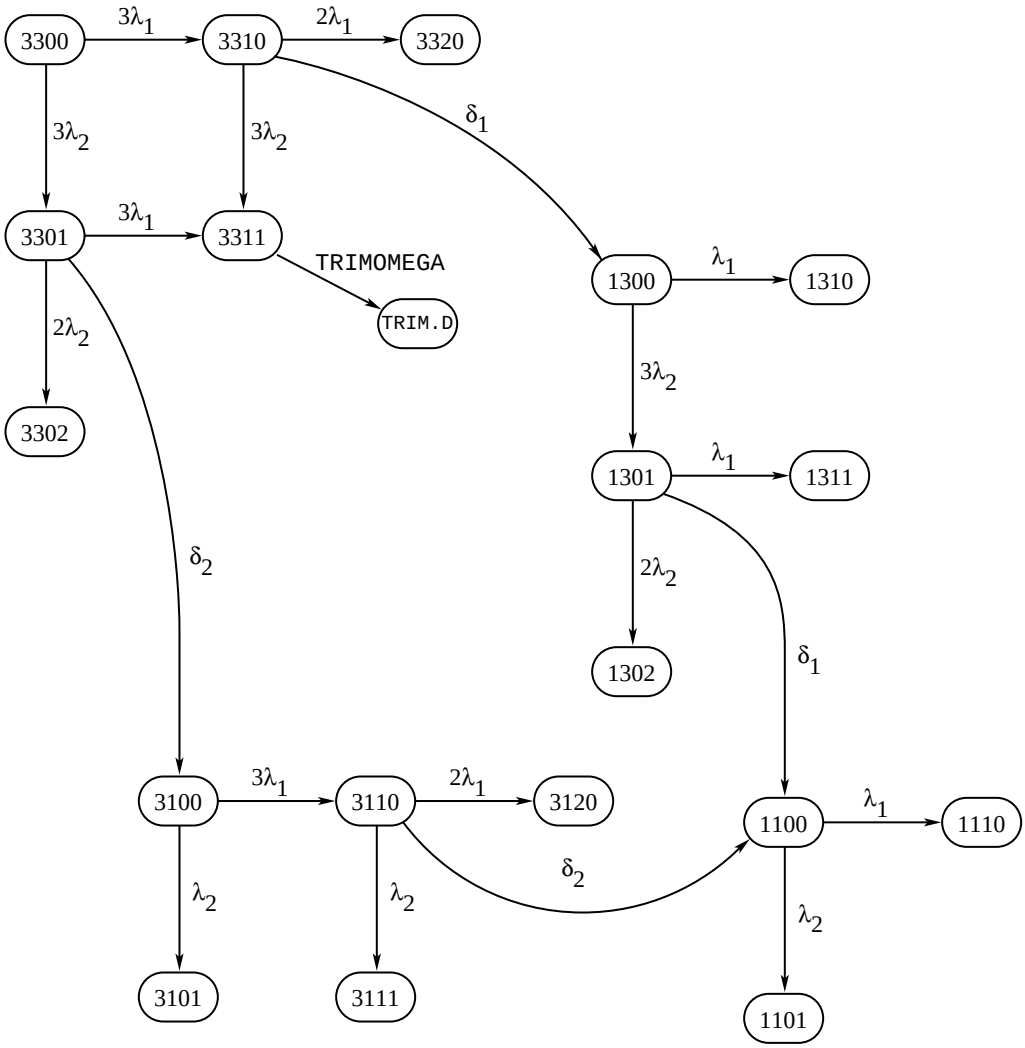


Figure 49. Two triads trimmed by ASSIST.

```
MODEL FILE = nt-trim.mod              SURE V7.9.8 15 Apr 94  10:03:23

DEATHSTATE     LOWERBOUND     UPPERBOUND     COMMENTS             RUN #2
----------   -----------    -----------    --------------------------
     1       2.98539e-06    3.00633e-06
             -----------    -----------
  SUBTOTAL   2.98539e-06    3.00633e-06
PRUNESTATE     LOWERBOUND     UPPERBOUND
----------   -----------    -----------
prune   2    1.46089e-12    1.50000e-12
             -----------    -----------
  SUBTOTAL   1.46089e-12    1.50000e-12
TOTAL        2.98539e-06    3.00633e-06

 14 PATH(S) TO DEATH STATES

0.017 SECS. CPU TIME UTILIZED
86? orprob
```

```
 MODEL FILE = nt-trim.mod              SURE V7.9.8 15 Apr 94  10:03:34

    RUN #      LOWERBOUND    UPPERBOUND
 ----------   -----------   -----------
     1        2.98539e-06   3.00633e-06
     2        2.98539e-06   3.00633e-06
 ----------   -----------   -----------
  OR PROB =   5.97078e-06   6.01266e-06

 87? exit
```

As can be seen, the results are the same. For two triads, the computation times are also very close. However, for larger configurations, the savings due to trimming can be substantial. An ASSIST/SURE session solving a similar model for `N_TRIADS = 10` with and without trimming follows:

```
$ assist nt
ASSIST VERSION 7.1                          NASA Langley Research Center
N_TRIADS? 10

PARSING TIME = 0.33 sec.
generating SURE model file...

787200 transitions processed.
RULE GENERATION TIME = 1941.87 sec.
NUMBER OF STATES IN MODEL = 59050
NUMBER OF TRANSITIONS IN MODEL = 787320
393660 DEATH STATES AGGREGATED INTO STATE 1


$ assist nt-trim
ASSIST VERSION 7.1                          NASA Langley Research Center
N_TRIADS? 10

PARSING TIME = 0.44 sec.
generating SURE model file...

66400 transitions processed.
RULE GENERATION TIME = 379.30 sec.
NUMBER OF STATES IN MODEL = 6147
NUMBER OF TRANSITIONS IN MODEL = 66561
33280 DEATH STATES AGGREGATED INTO STATE 1
23040 TRIMMED TRANSITIONS AGGREGATED INTO STATE 3
```

Substantial savings can be realized by trimming. The model generation time has been reduced from 1941 sec to 379 sec. The SURE solution times are similarly reduced:

```
$ sure

  SURE V7.9.3   NASA Langley Research Center

  1? read0 nt

    2081.90 SECS. TO READ MODEL FILE

2361989? run

 MODEL FILE = nt.mod                  SURE V7.9.3 12 Feb 92  15:34:32
```

```
                LOWERBOUND    UPPERBOUND    COMMENT              RUN #1
 -----------   -----------   -----------    -------------------------
                1.49226e-05   1.52747e-05    <prune 5.5e-10>
 87968 PATH(S) TO DEATH STATES 119932 PATH(S) PRUNED
 HIGHEST PRUNE LEVEL =  5.14253e-13
 39.716 SECS. CPU TIME UTILIZED
2361990? exit
$ sure
  SURE V7.9.3   NASA Langley Research Center
  1? read0 nt-trim
    PRUNE STATES ARE: 2

      178.73 SECS. TO READ MODEL FILE
199715? run
 MODEL FILE = nt-trim.mod             SURE V7.9.3 12 Feb 92  12:12:18

 DEATHSTATE    LOWERBOUND    UPPERBOUND    COMMENTS             RUN #1
 ----------   -----------   -----------    -------------------------
      1       1.49226e-05   1.52741e-05
 sure prune   0.00000e+00   1.72755e-08
              -----------   -----------
   SUBTOTAL   1.49226e-05   1.52741e-05
 PRUNESTATE    LOWERBOUND    UPPERBOUND
 ----------   -----------   -----------
 prune   2    3.23486e-10   3.37530e-10
              -----------   -----------
   SUBTOTAL   3.23486e-10   3.37530e-10

  TOTAL        1.49226e-05   1.52917e-05
  4836 PATH(S) TO DEATH STATES, 10830 PATH(S) PRUNED
  HIGHEST PRUNE LEVEL =  2.75830e-11
 2.683 SECS. CPU TIME UTILIZED
199716? exit
```

As can be seen, most of the time was spent reading in the model. Because the trimmed model is much smaller, the execution time is likewise much smaller—less than 3 min rather than 34 min. A very slight increase is seen in the separation of the bounds. The upper bound is `1.529e-05` rather than `1.527e-05`. However, this difference is clearly a very small price to pay for the dramatic decrease in execution time.

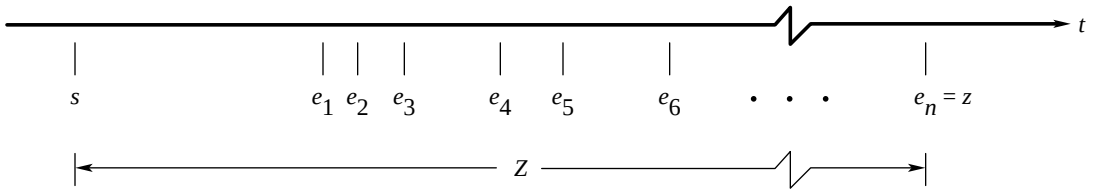
15. Transient and Intermittent Faults

Computer systems are susceptible to transient and intermittent faults, as well as permanent faults. Transient faults are faults that cause erroneous behavior for a short period of time and then disappear. Intermittent faults are permanent faults that periodically exhibit erroneous behavior then correct behavior. Transient faults can corrupt the state (i.e., memory) of an otherwise good processor. Unless the state of the processor is recovered, the impact can be as damaging as a permanent fault. Transient faults can also confuse a reconfigurable system. If the system improperly diagnoses a transient fault as a

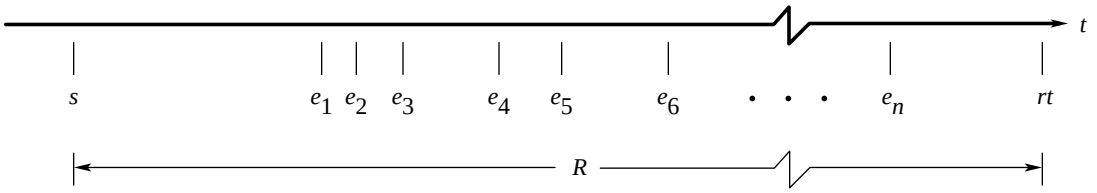
permanent fault, then a good processor is unnecessarily eliminated from the active configuration. Because transient faults tend to occur more frequently than permanent faults, this can have a significant impact on the probability of system failure. Intermittent faults can deceive the operating system into diagnosing the fault as transient rather than permanent. Therefore, a processor experiencing an intermittent fault may be left in operation much longer than a permanent fault or may be repeatedly removed, restarted, and returned to operation. Thus, intermittent faults make the system vulnerable to near-coincident faults for a much longer time than a permanent fault, and also may increase the fault management overhead enough to degrade performance. Clearly a properly designed system must deal effectively with these faults. Furthermore, the assessment of such systems depends upon careful modeling of these faults.

15.1. Transient Fault Behavior

A transient fault may or may not generate errors that are detectable by the voters of the operating system. The following two timing graphs illustrate the two possible effects of a single transient fault:



Case 1; reconfiguration does not occur.



Case 2; reconfiguration occurs.

where s is time of fault arrival, e_i is time of detection of the ith error ($1 < i < n$), rt is time operating system reconfigures, Z is $e_n - s$, and R is $rt - s$.

These two cases represent the outcome of two competing processes—the disappearance of the transient fault and its effects and the reconfiguration process of the operating system. In the first case, Z is a random variable, which represents the duration of transient errors given that reconfiguration does not occur, and R is a random variable, which represents the reconfiguration time given that reconfiguration does occur. Let $F_R(r)$ represent the distribution of the reconfiguration time given that the system reconfigures.

$$F_R(r) = \text{Prob}\ [R < r] \tag{1}$$

Let $F_Z(z)$ represent the distribution of the time for the disappearance of the transient error, given that reconfiguration does not occur.

$$F_Z(z) = \text{Prob}\ [Z < z] \tag{2}$$

The first distribution F_R can be directly observed. The second distribution F_Z is more troublesome to determine because a fault produces errors that may persist long after the fault has actually disappeared. Sometimes the errors disappear quickly, sometimes they do not. The exact time when the last error has disappeared is not directly observable. However, determination of a worst-case result is often possible. This maximum time of disappearance can sometimes be derived from the operating system code. This follows from the fact that the operating system is responsible for the recovery from the transient fault. If the operating system does not perform a state-restoration process periodically, a transient fault can be as damaging as a permanent fault. For example, an alpha particle may flip a bit in memory. If this memory is not rewritten, the error will persist indefinitely. Therefore, the fault-tolerant operating system must periodically rewrite volatile memory with voted versions of the state.

15.2. Modeling Transient Faults

Techniques for modeling various systems with transient faults will be detailed in this section.

15.2.1. Degradable triad subject to transient faults. In this section, modeling a triplex system that is subject to transient faults will be investigated. First, a failure rate γ must be determined for transient faults (the rate of transient fault arrivals). Often γ is assumed to be 10 times the permanent fault rate λ (ref. 6). This system will be assumed to have been designed so that it can recover from transient faults. Otherwise, transient faults are as damaging as permanent faults and should be modeled as permanent faults.

This recovery is accomplished by periodically voting all volatile internal states of the processor. Each nonfaulty processor rewrites each data value of its internal state with a voted value. Let ISVP equal the period during which the operating system replaces the entire volatile state with voted values. The active duration of a transient fault is assumed to be small in comparison to ISVP. Assuming that the time from the fault arrival to the operating system update is uniformly distributed, the mean is ISVP/2 and the standard deviation is ISVP/$2\sqrt{3}$. Of course, the actual mean and standard deviation should be experimentally measured. The values of these parameters would depend strongly upon the strategy of transient recovery that is used by the operating system.

During the period of time from the arrival of a transient fault until the system can recover, the system is vulnerable to near-coincident failures. If a second processor experiences a transient or permanent fault while transient errors are present, then the three-way voter can no longer mask the faults. Such a state is a system failure state. In figure 50, a model of a degradable triad system subject to only transient faults is shown.

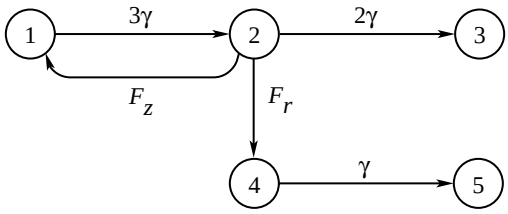


Figure 50. Degradable triad subject to transient faults.

The corresponding SURE model is

```
GAMMA = 1E-4;          (* Arrival rate for transient faults *)
MU1 = 2.7E-4;          (* Mean reconfiguration time *)
SIGMA1 = 1.3E-4;       (* Standard deviation of reconfiguration time *)
ISVP = 1E-3;           (* Mean Internal State Voting Period *)
PROB_RECONF = .1;     (* Probability of reconfiguring out transient fault *)
```

```
1,2 = 3*GAMMA;
2,3 = 2*GAMMA;
2,4 = <MU1,SIGMA1,PROB_RECONF>;
2,1 = <ISVP/2,ISVP/(2*SQRT(3)),1-PROB_RECONF>;
4,5 = GAMMA;
```

The transition labeled F_z represents the disappearance of a transient fault and the removal of all errors produced by it. The transition labeled F_r represents the improper reconfiguration of the system in the presence of a transient fault. Because two recovery transitions occur from state (2), it is necessary with SURE that the three-parameter form of recovery must be used. The first two parameters are the conditional mean and standard deviation. The third parameter is the probability that this transition succeeds over the fast transitions. The parameter `PROB_RECONF` represents the probability that the reconfiguration transition succeeds over the F_z transition. Thus, the third SURE parameter for F_r is `PROB_RECONF` and the third SURE parameter for F_z is `1 - PROB_RECONF`. An experimental procedure for measuring these parameters is described in reference 26. The probability of failure of the system as a function of the voting period ISVP is shown in figure 51.

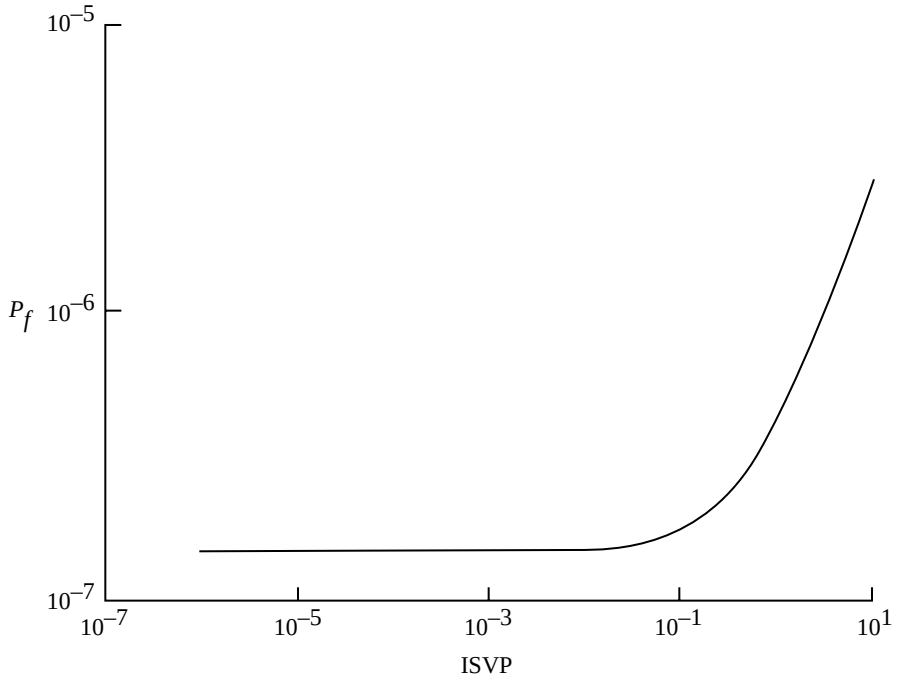


Figure 51. Failure probability as function of ISVP.

15.2.2. The SURE program and loop truncation. The model in figure 50 contains a loop, that is a path that returns to a state. Loops can lead to infinitely long paths. Fortunately in SURE, pruning has been shown to be conservative, even when used on models that include loops (ref. 9).

The error due to SURE-level pruning is reported in the comments field as follows when `LIST=0` or `1`.

```
<prune 1.2e-12>
```

If `LIST` is set to 2 or more, the prune values are listed on a separate row as follows:

```
DEATHSTATE     LOWERBOUND     UPPERBOUND     COMMENTS              RUN #4
----------   -----------   -----------     ---------------------------
     1       9.19500E-12   1.00000E-11
```

```
      4        3.46542E-10    4.77867E-10
 sure prune    0.00000e+00    1.00000E-13
               -----------    -----------
   SUBTOTAL    3.475645-10    4.87797E-10
```

The row which begins "`sure prune`" reports an upper bound on the error because of pruning. As mentioned previously, the pruning error is added to the upper bound. The upper bound is consequently always an upper bound on the probability of system failure, even if pruning is too severe. If the pruning is too severe, then the bounds will be far apart, but valid.

Models that contain fast loops, that is, loops with only fast transitions, can cause the SURE program to run forever unless a safety value is used. Fast loops generate an infinite sequence of paths that do not decrease in probability (as far as the upper bound of SURE is concerned). Thus, the program would run forever when only pruning is invoked. The `TRUNC` command sets the maximum number of times that SURE will expand a loop. The default value is 25, which will not be reached in most models. However, for models containing fast loops, this value will keep the program from running forever. The mathematical basis for loop truncation in SURE is given in the appendix.

15.2.3. Nonreconfigurable system subject to transient faults. In this section, an NMR system subject to both transient as well as permanent faults will be analyzed. The motivation for this example is the RCP (Reliable Computing Platform) architecture developed at Langley Research Center (ref. 27). The RCP utilizes NMR-style redundancy to mask faults and internal majority voting to flush the effects of transient faults. A major goal of this work is to provide the system with significant capability to withstand the transient effects of High Intensity Radiated Fields (HIRF). For simplicity, a quadraplex version of the RCP system shown in figure 52 will be examined first.

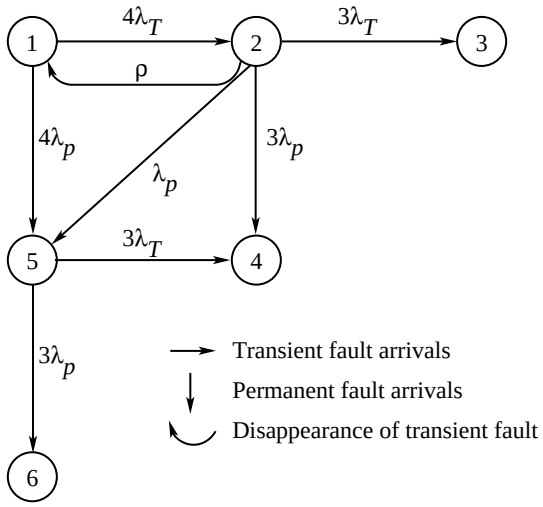


Figure 52. Reliability model of quadraplex RCP.

These faults arrive at rate λ_T for transient faults and λ_p for permanent faults. The transient fault and all errors produced by it can be removed by voting the internal state. This removal is sometimes referred to as transient fault scrubbing. In the RCP system, this scrubbing takes place continuously and does not rely upon the system detecting the transient fault. The presence of the transition from state (2) to state (1) depends upon the proper design of the operating system and is necessary so that a system can recover the state of a processor that has been affected by a transient fault. To simplify this discussion, the arrival of a second transient fault on the same processor before the disappearance of the first

transient fault has not been included in the model. The model has six states of which three are operational. State (1) represents the initial fault-free state of the system. Only two transitions from state (1) result from the arrival of either a transient or permanent fault. These transitions carry the system into states (2) and (4), both of which are not system failure states. This property has been justified by a formal proof that establishes that for a quadraplex RCP, no single fault can cause the system to produce an erroneous majority output. All transitions except one from these states are caused by second failures. These second failures lead to system failure states. The other transition from state (2) back to state (1) models the transient fault scrubbing process. A formal proof justifying the inclusion of this transition has been developed that demonstrates that the RCP system removes the effects of a transient fault within a bounded amount of time (ref. 27).

The probability of system failure as a function of 1/ρ, the rate of recovering the state, is shown in figure 53. The model was solved by using the STEM reliability analysis program for the parameters of $\lambda_p = 10^{-4}$/hr, $\lambda_T = 10^{-3}$/hr, and mission time $T = 10$ hr (ref. 11). Because no recoveries exist in the model, a pure Markov model solver is sufficient, although SURE could have been used if desired.

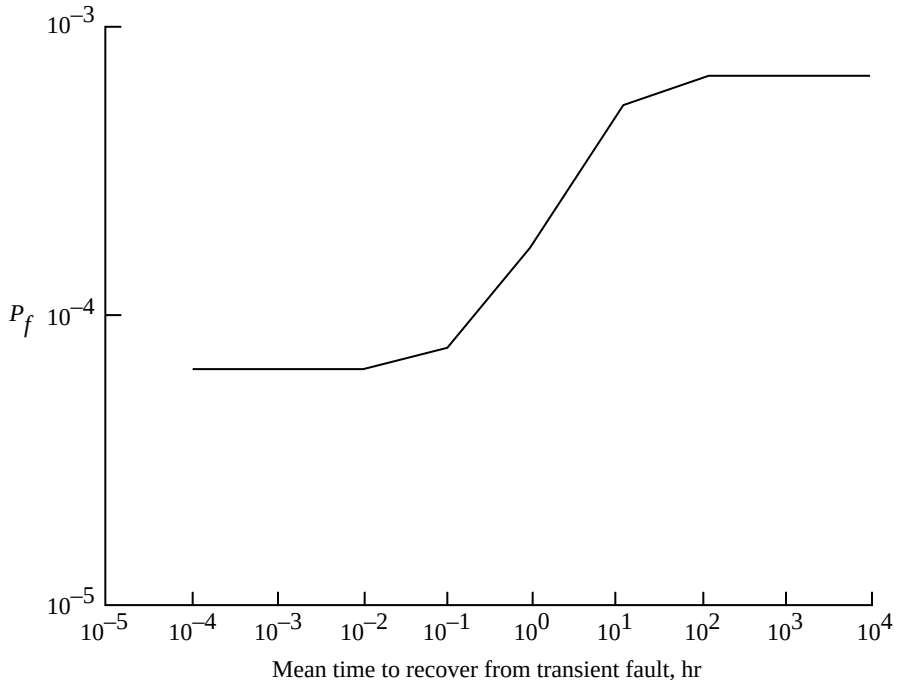


Figure 53. Probability of failure as function of 1/ρ.

Note the validation tasks that have been eliminated by not using reconfiguration in the RCP. First, it is not necessary to perform fault-injection experiments to measure the recovery time distributions. Second, fault latency is of no concern. Fault latency is only a concern when detecting and removing a faulty component because latency merely defers error production, and thus detection. Third, the logical complexity of the system is greatly reduced. No reconfiguration process is necessary and the interface to the sensors and the actuators is static instead of dynamic. Hence, fewer design errors must be corrected during the validation process. The following ASSIST input will generate a model of the RCP for a specified value of N.

```
INPUT N;                  (* Number of processors *)
LAMBDA = 1E-4;            (* Permanent fault arrival rate *)
GAMMA = 10*LAMBDA;        (* Transient fault arrival rate *)
"FLUSHTIME = 1E-1 TO* 1E4 by 10;"
```

```
SPACE = (NW: 0..N,            (* Number of working processors *)
NFP: 0..N,                    (* Active procs. with permanent faults *)
NFT: 0..N);                   (* Active procs. with transient faults *)

START = (N, 0, 0);
DEATHIF NFP+NFT >= NW;  (* Majority of active processors failed *)

IF NW>0 THEN
   TRANTO NW = NW-1, NFP = NFP+1 BY NW*LAMBDA; (* Permanent fault arrival *)
   TRANTO NW = NW-1, NFT= NFT+1 BY NW*GAMMA;   (* Transient fault arrival *)

ENDIF;

IF NFT > 0 THEN
   TRANTO NFP = NFP+1, NFT = NFT-1 BY NFT*LAMBDA;  (* transient -> permanent *)
   TRANTO NW=NW+1, NFT = NFT-1 BY FAST 1/FLUSHTIME;
      (* Transient fault disappearance *)
ENDIF;
```

The model has two TRANTO rules that generate fault-arrival transitions—one TRANTO rule generates permanent fault-arrival transitions and one TRANTO rule generates transient fault-arrival transitions. Another TRANTO rule generates transitions corresponding to flushing of the effects of a transient fault. Note that this rule increments NW, as well as decrementing NFT. Another TRANTO rule covers a transiently faulted processor that fails permanently. A processor that has been upset by a transient phenomena is not immune to a permanent failure. The DEATHIF statement sums the number of transiently and permanently faulted processors and the result is compared with the number of working processors.

The probability of system failure as a function of ρ (1/FLUSHTIME) is given for the N = 3, 5, and 7 processor configurations in figure 54.

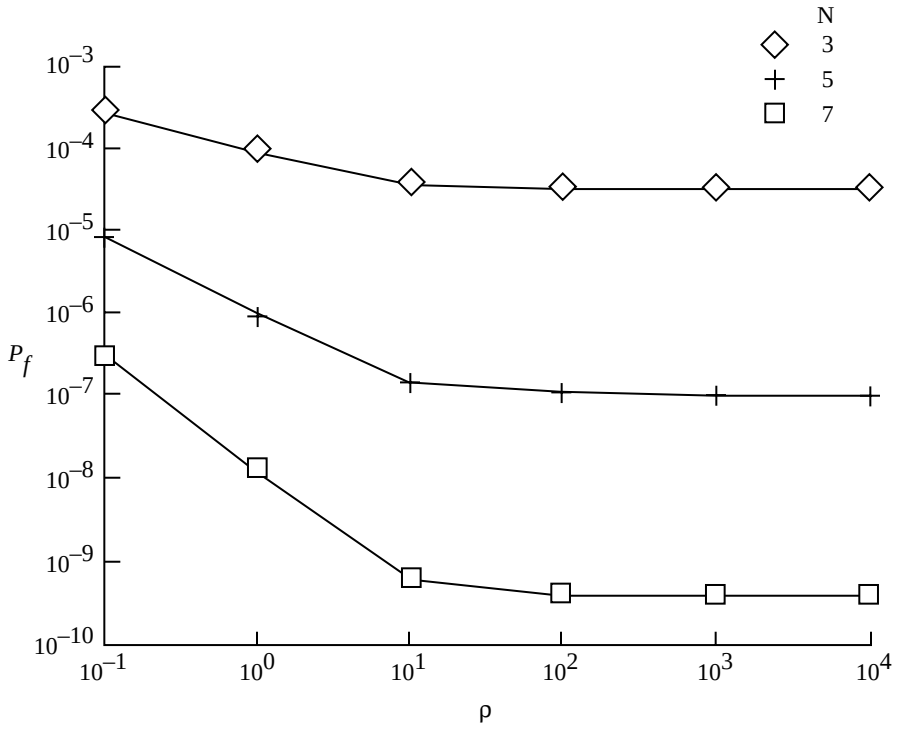


Figure 54. Probability of failure as function of ρ.

Note that the inflection point of the curve does not vary significantly with variation in N.

15.3. Degradable Quadraplex Subject to Transient and Permanent Faults

Because transient faults tend to occur at a faster rate than permanent faults, many systems are designed to tolerate transient faults that disappear after a short amount of time. Because fewer processors are needlessly reconfigured, this design can significantly reduce the number of spare components needed. However, the operating system must be able to distinguish between transient faults and permanent faults. Typically, a simple algorithm is used by the operating system to distinguish the two types of faults. Because this algorithm is not foolproof, a transition in the model that represents the operating system incorrectly reconfiguring in the presence of a transient fault must be included.

In the SIFT system, significant consideration was given to this problem. The operating system is faced with conflicting goals. If the fault is permanent, the system needs to reconfigure as quickly as possible. If the fault is transient, then the system should not reconfigure. Typically, the operating system delays the reconfiguration process temporarily to see whether the fault will disappear. Clearly, the amount of time the operating system delays has a significant impact on system reliability because of the susceptibility to near-coincident faults. Only a minimal amount of information resides in the dynamic (volatile) portions of system memory. The schedule table in SIFT is static, so it could be stored in nonvolatile read-only memory (ROM). The program code can also be stored in ROM.

The ASSIST input file for a SIFT-like system that starts with four processors is

```
NP = 4;                    (* Number of processors *)
LAMBDA = 1E-4;             (* Permanent fault arrival rate *)
GAMMA = 10*LAMBDA;         (* Transient fault arrival rate *)
MU = 1E-4;                 (* Mean permanent fault reconfiguration time *)
STD = 2E-4;                (* Standard dev. of permanent fault reconfig. *)

MU_REC = 7.4E-5;           (* Cond. mean reconfiguration time for transient fault *)
STD_REC = 8.5E-5;         (* Cond. standard deviation of transient reconfiguration *)
P_REC = .10;               (* Probability system reconfigures out a transient *)
"ISVP = 1E-2;"             (* Period of system rewrite of internal state *)
"MU_DISAPPEAR = ISVP/2;"             (* Cond. mean time to transient disappearance *)
"STD_DISAPPEAR = ISVP/(2*SQRT(3));"  (* Cond. stan. dev. of disappearance time *)

SPACE = (NW: 0..NP,              (* Number of working processors *)
         NFP: 0..NP,             (* Active procs. with permanent faults *)
         NFT: 0..NP);            (* Active procs. with transient faults *)
START = (NP, 0, 0);

DEATHIF NFP+NFT >= NW;  (* Majority of active processors failed *)

IF NW>0 THEN
   TRANTO (NW-1, NFP+1, NFT) BY NW*LAMBDA; (* Permanent fault arrival *)
   TRANTO (NW-1, NFP, NFT+1) BY NW*GAMMA;  (* Transient fault arrival *)
ENDIF;

IF NFT > 0 THEN
   TRANTO (NW+1, NFP, NFT-1) BY <MU_DISAPPEAR,STD_DISAPPEAR,1-P_REC> ;
      (* Transient fault disappearance *)
   TRANTO NFT = NFT-1 BY <MU_REC, STD_REC,P_REC>;
      (* Transient fault reconfiguration *)
ENDIF;

IF NFP > 0 TRANTO NFP = NFP-1 BY <MU,STD>;
   (* Permanent fault reconfiguration *)
```

In this model, the system does not collapse a triad to a simplex. Instead, the triad degrades to a dual.

15.4. NMR With Imperfect Recovery From Transient Faults

If a system is not designed with a foolproof capability of removing the effects of a transient fault, the fraction of transient faults R that are recoverable must be measured. The effect of a nonrecoverable

transient fault is the same as a permanent fault. Therefore, this situation can be modeled by increasing the permanent failure rate by (1 – R) times the transient fault rate for each working processor in the system.

An ASSIST model for this situation is

```
INPUT NP;                  (* Number of processors *)
LAMBDA = 1E-4;             (* Permanent fault arrival rate *)
GAMMA = 10*LAMBDA;         (* Transient fault arrival rate *)
RHO = 1E2;
"R = 0 TO+ 1 BY 0.05;"

SPACE = (NW: 0..NP,              (* Number of working processors *)
         NFP: 0..NP,             (* Active procs. with permanent faults *)
         NFT: 0..NP);            (* Active procs. with transient faults *)
START = (NP, 0, 0);

DEATHIF NFP+NFT >= NW;  (* Majority of active processors failed *)

IF NW>0 THEN
    TRANTO (NW-1, NFP+1, NFT) BY NW*LAMBDA + NW*(1-R)*GAMMA;
    TRANTO (NW-1, NFP, NFT+1) BY NW*R*GAMMA;
ENDIF;

IF NFT > 0 THEN
   TRANTO (NW+1, NFP, NFT-1) BY FAST RHO; (* Transient fault disappearance *)
   TRANTO (NW,NFP+1,NFT-1) BY NFT*LAMBDA; (* Transient -> permanent *)
ENDIF;
```

The four-processor case is shown in figure 55. Figure 56 shows the probability of system failure. The effect becomes even more dramatic as the number of processors is increased as shown in figure 57.

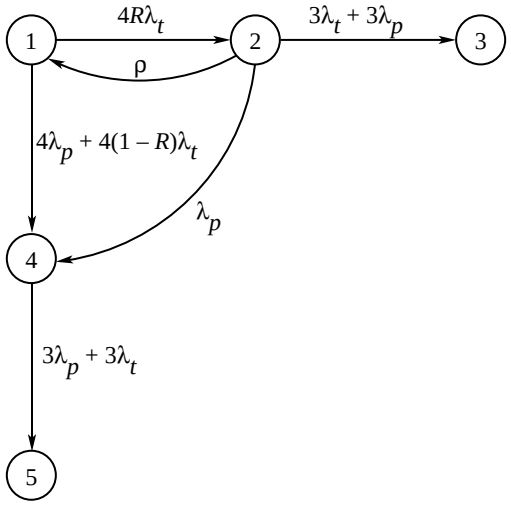


Figure 55. Markov model of imperfect transient recovery.

15.5. Degradable NMR With Perfect Transient Fault Recovery

In this section, some problems with modeling degradable NMR systems that are subject to permanent and transient faults are explored. The major problem is that many different situations with competing recoveries are possible. To simplify the discussion, 100 percent of the errors produced by a single transient fault are assumed to be flushed by the operating system `R = 1`. Each situation involves

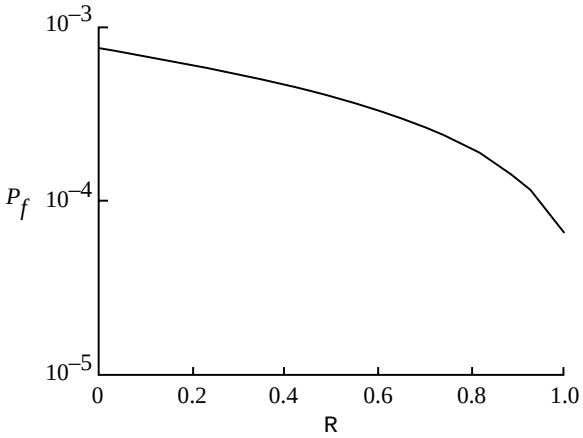


Figure 56. Probability of system failure as function of R.

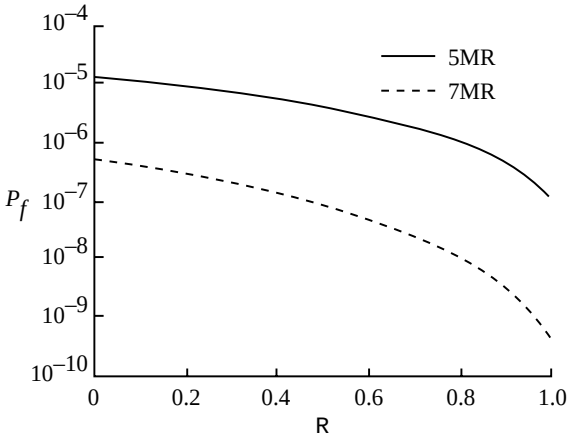


Figure 57. Probability of system failure as function of R for 5MR and 7MR

different parameters, which must be experimentally measured. To illustrate the problem, a degradable 6-plex will first be considered. If the model in section 15.3 is modified by changing the first line to

```
NP = 6;
```

the SURE program will object with the following message:

```
*** ERROR: SUM OF EXITING PROBABILITIES IS NOT 1 AT 12
```

When the generated model is examined, five transitions are found at state (12):

```
48:    12(* 3,1,1 *),    1(* 2,2,1 *) = 3*LAMBDA;
49:    12(* 3,1,1 *),    1(* 2,1,2 *) = 3*GAMMA;
50:    12(* 3,1,1 *),    9(* 4,1,0 *) = <MU_DISAPPEAR,STD_DISAPPEAR,1-P_REC>;
51:    12(* 3,1,1 *),   15(* 3,1,0 *) = <MU_REC,STD_REC,P_REC>;
52:    12(* 3,1,1 *),   16(* 3,0,1 *) = <MU,STD>;
```

Three of the five transitions are competing recoveries. This competition occurs because two active faults exist at state (12)—one transient and one permanent. The three possible outcomes are

1. The permanent fault is reconfigured.

2. The transient fault is reconfigured.

3. The transient fault disappears.

The ASSIST model was originally constructed for a quadraplex system where any state with two active faults would be a death state. However, higher levels of redundancy result in more complexity. There are several solutions to this problem. Unfortunately, the more satisfactory models are more complex. We will begin will the simplest model.

The simplest solution to the problem is to make all states with two active faults death states. This method is used by programs that are based on the critical-pair approach such as found in CARE and HARP (refs. 16 and 28). This method can be used with ASSIST by changing the

```
DEATHIF NFT + NFP >= 2;
```

Although this change results in a conservative answer, the solution is not satisfactory because the model simply ignores all additional redundancy. Overly conservative results can be obtained with this technique.

A second solution to the problem is to model all recovery transitions with exponential distributions. The SURE program automatically determines all conditional parameters when this method is used. The model is

```
NP = 6;                          (* Number of processors *)
LAMBDA = 1E-4;                   (* Permanent fault arrival rate *)
GAMMA = 10*LAMBDA;               (* Transient fault arrival rate *)
W = .5;                          (* Transient fault disappearance rate *)
DELTA = 3.6E3;                   (* Reconfiguration rate *)

SPACE = (NW: 0..NP,              (* Number of working processors *)
         NFP: 0..NP,             (* Active procs. with permanent faults *)
         NFT: 0..NP);            (* Active procs. with transient faults *)
START = (NP, 0, 0);

DEATHIF NFP+NFT >= NW;  (* Majority of active processors failed *)

IF NW>0 THEN
   TRANTO (NW-1, NFP+1, NFT) BY NW*LAMBDA; (* Permanent fault arrival *)
   TRANTO (NW-1, NFP, NFT+1) BY NW*GAMMA;  (* Transient fault arrival *)
ENDIF;

IF NFT > 0 THEN
   TRANTO (NW+1, NFP, NFT-1) BY FAST W;   (* Transient fault disappearance *)
   TRANTO NFT = NFT-1 BY FAST DELTA;      (* Transient fault reconfiguration *)
ENDIF;

IF NFP > 0 TRANTO NFP = NFP-1 BY FAST DELTA; (* Permanent f. reconfiguration *)
```

This model will work for arbitrary values of NP. Unfortunately, this model makes the assumption that all recovery distributions are exponentially distributed.

The most accurate way to model such systems is to use general recovery distributions. This model necessitates analysis of each situation where multiple competing recoveries occur. For a 5-plex or a 6-plex, the following operational states with two active faults exist:

1. Two permanent faults
2. Two transient faults
3. One transient and one permanent fault

The conditional moments for each case must be measured experimentally.

15.5.1. Two permanent faults in one state. Because two active permanent faults exist in a state, the mean and the standard deviation of the time until the first recovery occurs must be measured. Because the two processors are identical, it is not necessary to track which processor recovers first or estimate the moments of two conditional recovery distributions. The parameters are

1. MU_2, mean recovery time of the first of two competing recoveries
2. STD_2 standard deviation of the recovery time of the first of two competing recoveries

15.5.2. Two transient faults in one state. The transient fault situation is complicated because the transient faults may disappear rather than be reconfigured. Because of symmetry, which transient fault disappears or is reconfigured first need not be recorded. However, the mean and the standard deviation of the first one to disappear or reconfigure and the ratio of times between these two outcomes must be obtained. The parameters are

1. P_DISAPPEAR_2, probability one of the transient faults disappears before the system reconfigures one of the transient faults.

2. MU_DISAPPEAR_2, conditional mean time of disappearance of one of two transient faults
3. STD_DISAPPEAR_2, conditional standard deviation of time of disappearance of one of two transient faults
4. MU_REC_2, conditional mean time to reconfigure one of the transient faults before either disappears
5. STD_REC_2, conditional standard deviation of time to reconfigure one of the transient faults before either disappears

15.5.3. One transient and one permanent fault in one state. This situation is the most complicated because it is not symmetrical—one processor has a transient fault and the other processor has a permanent fault. Thus, the conditional means and standard deviations must be estimated for each possible outcome:

1. The transient fault disappears before the system reconfigures either fault.
2. The system reconfigures the transient fault before it disappears or the permanent fault is reconfigured.
3. The system reconfigures the permanent fault before the transient fault disappears or is reconfigured.

In the design phase, these parameters would be difficult to estimate. However, after the system is built, three histograms could be collected for each outcome and the corresponding means and standard deviations could be computed. The parameters are

1. P_DIS_BEF2, probability the transient fault disappears before the system reconfigures either fault
2. P_REC_TRAN, probability the system reconfigures the transient fault before it disappears or the permanent fault is reconfigured
3. P_REC_PERM, probability the system reconfigures the permanent fault before the transient fault disappears or is reconfigured
4. MU_DIS_3, conditional mean time of disappearance of the transient fault given that it wins the 3-way race
5. STD_DIS_3, conditional standard deviation of the time of disappearance of the transient fault given that it wins the 3-way race
6. MU_REC_3, conditional mean time to reconfigure the transient fault given that it wins the 3-way race
7. STD_REC_3, conditional standard deviation of time to reconfigure the transient fault given that it wins the 3-way race
8. MU_3, conditional mean time to reconfigure the permanent fault given that it wins the 3-way race
9. STD_3, conditional standard deviation of time to reconfigure the permanent fault given that it wins the 3-way race

The complete model is

```
NP = 6;                    (* Number of processors *)
LAMBDA = 1E-4;             (* Permanent fault arrival rate *)
GAMMA = 10*LAMBDA;         (* Transient fault arrival rate *)
(* ------------ Constants associated with one permanent ----------- *)
```

```
MU = 1E-4;                (* Mean permanent fault recovery time *)
STD = 2E-4;               (* Standard deviation permanent fault *)

(* ------------ Constants associated with one transient ----------- *)

MU_REC = 7.4E-5;          (* Mean reconfiguration time from transient  *)
STD_REC = 8.5E-5;         (* Standard deviation of transient reconfiguration *)
P_REC = .10;              (* Probability system reconfigures transient *)
"ISVP = 1E-2;"              (* Period of system rewrite of internal state *)
"MU_DISAPPEAR = ISVP/2;"            (* Mean time to transient disappearance *)
"STD_DISAPPEAR = ISVP/(2*SQRT(3));"  (* Stan. dev. of disappearance time *)

(* ------------ Constants associated with two transients ----------- *)

MU_REC_2 = 7.4E-5;        (* Mean reconfiguration time from transient  *)
STD_REC_2 = 8.5E-5;       (* Standard deviation of transient reconfiguration *)
P_DISAPPEAR_2 = .92;      (* Probability system reconfigures transient *)
"MU_DISAPPEAR_2 = 5E-3;"  (* Mean time to transient disappearance *)
"STD_DISAPPEAR_2 = 3E-3;" (* Stan. dev. of disappearance time *)

(* ------------ Constants associated with two permanents ----------- *)

MU_2 = 1E-4;              (* Mean permanent fault recovery time *)
STD_2 = 2E-4;             (* Standard deviation permanent fault *)

(* --- constants associated with states with a permanent and a transient --- *)

"P_DIS_BEF2 = .3;"      (* Probability the transient disappears *)
"P_REC_TRAN = .3;"      (* Probability the transient is reconfigured *)
"P_REC_PERM = 1-(P_DIS_BEF2+P_REC_TRAN);" (* Prob. permanent is reconfigured *)
"MU_DIS_3 = 1E-4;"      (* Conditional mean time of disappearance of
                           transient given that it wins the 3-way race. *)
"STD_DIS_3 = 1E-4;"     (* Conditional standard time of disappearance of
                           the transient given that it wins the 3-way race. *)
"MU_REC_3 = 1E-4;"      (* Conditional mean time to reconfigure the
                           transient given that it wins the 3-way race. *)
"STD_REC_3 = 1E-4;"     (* Conditional standard deviation of time to
                           reconfigure the transient given that it wins *)
"MU_3 = 1E-4;"          (* Conditional mean time to reconfigure the
                           permanent given that it wins. *)
"STD_3 = 1E-4;"         (* Conditional standard deviation of time
                           to reconfigure the permanent given that it wins *)

SPACE = (NW: 0..NP,           (* Number of working processors *)
         NFP: 0..NP,          (* Active procs. with permanent faults *)
         NFT: 0..NP);         (* Active procs. with transient faults *)
START = (NP, 0, 0);

DEATHIF NFP+NFT >= NW;  (* Majority of active processors failed *)

IF NW>0 THEN
   TRANTO (NW-1, NFP+1, NFT) BY NW*LAMBDA; (* Permanent fault arrival *)
   TRANTO (NW-1, NFP, NFT+1) BY NW*GAMMA;  (* Transient fault arrival *)
ENDIF;

IF NFT + NFP = 1 THEN   (* 1 active fault *)
   IF NFT > 0 THEN
      TRANTO (NW+1, NFP, NFT-1) BY <MU_DISAPPEAR,STD_DISAPPEAR,1-P_REC> ;
        (* Transient fault disappearance *)
      TRANTO NFT = NFT-1 BY <MU_REC, STD_REC,P_REC>;
         (* Transient fault reconfiguration *)
   ENDIF;

   IF NFP > 0 TRANTO NFP = NFP-1 BY <MU,STD>; (* Perm. f. reconfiguration *)
ENDIF;
IF NFP = 2      (* Case 1: Two permanents *)
```

```
   TRANTO NFP = NFP-1 BY <MU_2,STD_2>;  (* Permanent fault reconfiguration *)
IF NFT = 2 THEN (* Case 2: Two transients *)
   TRANTO (NW+1, NFP, NFT-1)
     BY <MU_DISAPPEAR_2,STD_DISAPPEAR_2,P_DISAPPEAR_2> ;
         (* Transient fault disappearance *)
    TRANTO NFT = NFT-1 BY <MU_REC_2, STD_REC_2,1-P_DISAPPEAR_2>;
          (* Transient fault reconfiguration *)
ENDIF;

IF (NFT = 1) AND (NFP = 1) THEN   (* 1 transient and 1 permanent *)
   TRANTO (NW+1, NFP, NFT-1) (* Transient fault disappearance *)
      BY <MU_DIS_3,STD_DIS_3,P_DIS_BEF2> ;
   TRANTO NFT = NFT-1        (* Transient fault reconfiguration *)
      BY <MU_REC_3, STD_REC_3,P_REC_TRAN>;
   TRANTO NFP = NFP-1        (* Permanent fault reconfiguration *)
      BY <MU_3,STD_3,P_REC_PERM>;
ENDIF;
```

Obviously, a rough sensitivity analysis should be performed to determine how sensitive a system is to transient faults before developing such a complex model and measuring so many parameters.

15.6. Fault-Tolerant Processor

The strategy used in the fault-tolerant processor (FTP) of the Charles Stark Draper Laboratory for handling transient faults is different from that used in earlier fault-tolerant systems such as SIFT (ref. 29). In these systems, reconfiguration was deferred until the system was reasonably certain that the fault was permanent. Once a processor was removed, it was never reinstated. In FTP, a different strategy is used. Upon the first detection of an error, the faulty processor is removed. The system then executes a self test on the removed processor. If the processor passes the test, the system diagnoses the problem as a transient fault and reinstates the processor. If the processor fails the self-test program, the fault is diagnosed as permanent and the processor is permanently removed. Thus, a transient fault that does not disappear in time will be diagnosed as permanent.

A partial model for the FTP is shown in figure 58. In this model, each state is described by a triple (NW,NFA,NFT) where NW is the number of working processors, NFA is the number of faulty processors (both transient and permanent), and NFT is the number of processors undergoing self test.

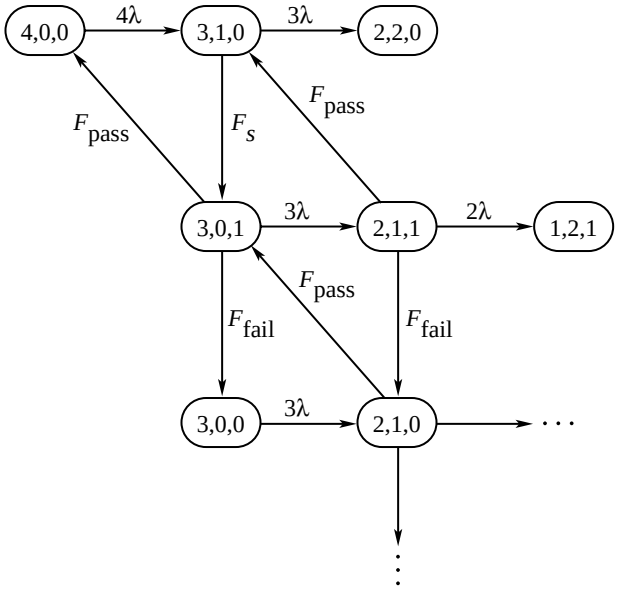


Figure 58. Partial model of FTP.

The transition from state (4,0,0) to state (3,1,0) represents the failure of any processor in the configuration. The transition from state (3,1,0) to state (3,0,1) represents the detection of a fault, the temporary removal of the processor from the active configuration, and the initiation of the self-test program. If the processor passes the self test, the processor is returned to the active configuration by the transition from state (3,0,1) back to state (4,0,0). If the processor fails the self test, the processor is permanently removed from the configuration. This removal occurs in the model in the transition from state (3,0,1) to state (3,0,0). Note that while the self-test program is in progress (i.e., in state (3,0,1)), that a second failure does not lead to system failure. This situation occurs because the outputs from the removed processor are not considered in the voting, thus the majority of the outputs being voted are nonfaulty. Therefore, state (2,1,1) is not a death state. The complete SURE model is

```
F_P = 1E-6 TO* 1 BY 10;
F_T = 1.0-F_P;
LAMBDA = 1E-4;
DET = 1E-7;
SIGDET = 10*DET;
TESTTIME = 1E-3;
SIGTEST = 2*TESTTIME;

     2(* 4,0,0 *),    3(* 3,1,0 *) = 4*LAMBDA;
     3(* 3,1,0 *),    1(* 2,2,0 *) = 3*LAMBDA;
     3(* 3,1,0 *),    4(* 3,0,1 *) = <DET,SIGDET>;
     4(* 3,0,1 *),    5(* 2,1,1 *) = 3*LAMBDA;
     4(* 3,0,1 *),    2(* 4,0,0 *) = <TESTTIME,SIGTEST,F_T>;
     4(* 3,0,1 *),    6(* 3,0,0 *) = <TESTTIME,SIGTEST,1-F_T>;
     5(* 2,1,1 *),    1(* 1,2,1 *) = 2*LAMBDA;
     5(* 2,1,1 *),    3(* 3,1,0 *) = <TESTTIME,SIGTEST,F_T>;
     5(* 2,1,1 *),    7(* 2,1,0 *) = <TESTTIME,SIGTEST,1-F_T>;
     6(* 3,0,0 *),    7(* 2,1,0 *) = 3*LAMBDA;
     7(* 2,1,0 *),    1(* 1,2,0 *) = 2*LAMBDA;
     7(* 2,1,0 *),    8(* 2,0,1 *) = <DET,SIGDET>;
     8(* 2,0,1 *),    1(* 1,1,1 *) = 2*LAMBDA;
     8(* 2,0,1 *),    6(* 3,0,0 *) = <TESTTIME,SIGTEST,F_T>;
     8(* 2,0,1 *),    9(* 2,0,0 *) = <TESTTIME,SIGTEST,1-F_T>;
     9(* 2,0,0 *),    1(* 1,1,0 *) = 2*LAMBDA;
```

This model was generated with the ASSIST input given below.

```
SPACE = (NW: 0..4,        (* number of working processors *)
         NFA: 0..4,       (* number of faulty active processors *)
         NFT: 0..4);      (* number of processors undergoing self test *)

START = (4,0,0);

LAMBDA = 1E-4;          (* Arrival rate of failures -- perm. or transient *)
DET = 1E-7;             (* Mean time to detect and remove proc. with fault *)
SIGDET = 10*DET;        (* Stan. dev. time to detect and remove processor *)
TESTTIME = 1E-3;        (* Mean time to execute self test *)
SIGTEST = 2*TESTTIME;   (* Stan. dev. of time to execute self test *)

"F_P = 1E-6 TO* 1 BY 10;" (* Probability failure was permanent *)
"F_T = 1.0-F_P;"          (* Probability failure was transient *)

   (* Fault arrival *)
IF NW > 0 TRANTO NW=NW-1, NFA = NFA + 1 BY NW*LAMBDA;
   (* Detection of fault and removal of processor for self test *)
IF (NFA > 0) AND (NFT = 0) TRANTO NFT=NFT+1, NFA = NFA - 1 BY <DET,SIGDET>;

IF NFT > 0 THEN
```

```
      (* Reinstatement of processor after transient fault *)
   TRANTO NFT=NFT-1, NW = NW+1 BY <TESTTIME,SIGTEST,F_T>;
      (* Permanent removal of processor with permanent fault *)
   TRANTO NFT=NFT-1 BY <TESTTIME,SIGTEST,1-F_T>;
ENDIF;

  (* System failure occurs if majority of outputs sent to voter are faulty *)
DEATHIF NFA >= NW;
```

In this model, the FTP is assumed to not allow a second processor to undergo self test while a first processor is undergoing self test. Note that the `IF` expression, which governs the generation of transitions that remove a processor from the active configuration for self test, is `IF (NFA > 0) AND (NFT = 0)`. The second term prevents the generation of a self-test transition when a processor is already under self test.

Most models containing transient faults require the estimation of the disappearance rates for transient faults. Virtually no experimental values are available for this parameter because it cannot be directly measured on operational equipment or through fault-injection experiments.

This parameter was not used explicitly in the model of the FTP system in this section. The disappearance rate of short transient faults does not matter because the FTP operating system masks all outputs after the first erroneous output until the self test is complete. However, if a transient fault persists long enough for a processor to fail the self test, then the fault is assumed to be permanent and the processor is permanently removed. Thus, the true transient fault disappearance rate affects the ratio of transient to permanent faults. This ratio, which is unknown, can play an important part in assessing the FTP strategy of reinstating processors.

15.7. Modeling Intermittent Faults

Before a solution technique was developed by White, the solution of semi-Markov models with nonexponential recovery transitions was extremely difficult. To circumvent the need for a semi-Markov model, the recovery process can be represented with a submodel of states that decompose the recovery into a series of smaller steps. This procedure is often referred to as the method of stages. The CARE III single fault model is an example of the method of stages (ref. 16). This multistep process was designed to provide a more accurate representation of the reconfiguration process than could be obtained with a single exponential process. However, many of these multistep models have used parameters that are not directly observable or measurable. For example, while the overall time of reconfiguration is directly observable, the individual times required to detect, isolate, and recover from a fault can be extremely difficult to measure accurately.

A remnant of the multistep recovery model approach is the concept that separate states must be used to represent the active and the inactive states of an intermittent fault. Therefore, models are frequently constructed that resemble the partial model shown in figure 59. In this partial model of a triplex-simplex system subject to intermittent faults, the states are described with a triple (`NW,NFA,NFB`) where `NW` is the number of working processors, `NFA` is the number of processors with active faults, and `NFB` is the number of processors with benign faults.

When a processor fails, the fault is initially benign. At some rate A the fault becomes active. At some rate B the active intermittent fault returns to the benign state. While the fault is benign, no errors are produced that would enable the system to detect the fault. The question of whether benign faults cause near-coincident failure must be addressed. One conservative approach is to assume that they do cause near-coincident failure. In this case, intermittent faults behave identically to permanent faults except that intermittent faults are reconfigured at a different rate than permanent faults. If faults in the benign state are assumed to not cause near-coincident failure, then many additional states, which contain benign faults, exist in the model. For example, states (1,0,2), (1,0,3), and (2,0,2) contain more

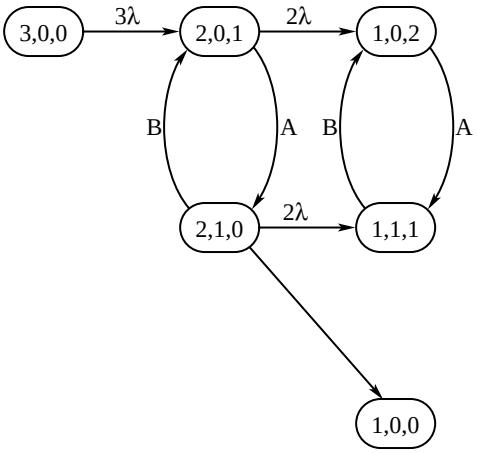


Figure 59. Detailed intermittent fault submodel.

faulty benign processors than good processors, yet these states are operational. The following ASSIST input could be used to generate the complete model for a triplex system

```
SPACE = (NW: 0..3,        (* Number of working processors *)
         NFA: 0..3,       (* Number of processors with active int. faults *)
         NFB: 0..3);      (* Number of processors with benign int. faults *)

START = (3,0,0);

L = 1E-4;                 (* Rate of arrival of intermittent faults *)
REC = 1E4;                (* Mean rate of reconfiguration *)
A = 1E2;                  (* Rate benign intermittent fault goes active *)
B = 1E2;                  (* Rate active intermittent fault goes benign *)

   (* Arrival of intermittent fault -- assumed to start out benign *)
IF NW > 0 TRANTO NW = NW-1, NFB = NFB + 1 BY NW*L;
   (* Benign intermittent fault becomes active *)
IF NFB > 0 TRANTO NFB = NFB - 1, NFA = NFA + 1 BY FAST A;

IF NFA > 0 THEN
      (* Active intermittent fault becomes benign *)
   TRANTO NFB = NFB + 1, NFA = NFA - 1 BY FAST B;
      (* Processor with active intermittent fault reconfigured -- 2 cases: *)
         (* Reconfigure to simplex working processor *)
   IF NW > 0 TRANTO (1,0,0) BY FAST (NW/(NW+NFB))*REC;
         (* Reconfigure to simplex with benign intermittent fault *)
   IF NFB > 0 TRANTO (0,0,1) BY FAST (NFB/(NW+NFB))*REC;
ENDIF;
  (* System failure occurs when majority of processors have active fault *)
DEATHIF NFA >= (NW+NFB);
```

The recovery rule generates two competing recoveries. This rule is necessary because the operating system makes an arbitrary choice among the processors that do not contain active faults when it degrades to a simplex. The probability that a processor with a benign fault becomes the remaining simplex processor is `NFB/(NW+NFB)`.

The problem with this model is that the on-off cycles of the intermittent fault must be modeled and the associated parameters must be measured. Realistic intermittent faults are difficult to create in the laboratory, and the rates at which they become active and benign are difficult to measure. Even if these parameters could be accurately measured, a semi-Markov model may not have enough generality to

accurately represent the behavior of the oscillations between active and benign. It is preferable to inject intermittent faults and observe the impact on the system. The system recovery time will probably be longer for intermittent faults than for transient faults. The resulting model is shown in figure 60. Although this model is considerably simpler, it can be much more accurate than the detailed model given in figure 59 because it relies only on directly observable parameters. Note that this method uses the conservative approach of assuming that intermittent faults can cause near-coincident failure during their benign phase.

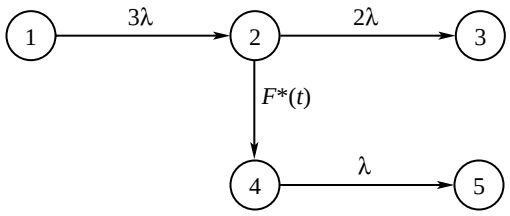


Figure 60. Model of triplex to simplex system subject to intermittent faults.

The SURE program has difficulty solving models with fast loops, that is, loops containing no slow transitions. The SURE program can solve the model generated by the ASSIST input above. The output is

```
$ sure
 SURE V7.4   NASA Langley Research Center

 1? read0 intm

      0.20 SECS. TO READ MODEL FILE
35? run

MODEL FILE = intm.mod                    SURE V7.4 24 Jan 90   14:17:49

                 LOWERBOUND    UPPERBOUND     COMMENTS          RUN #1
 -----------   -----------   -----------    --------------------------
               1.38175e-06   1.50309e-06    <prune 8.9e-13>

64 PATH(S) TO DEATH STATES 54 PATH(S) PRUNED
HIGHEST PRUNE LEVEL =  6.18304e-13
1.650 SECS. CPU TIME UTILIZED
36? exit
```

However for some parameter regions, the program may require large amounts of CPU time. For example, if the value of `B` is changed to `1E5`, the SURE program will require 3458 sec to solve the model. As `B` approaches infinity, the execution time approaches infinity. If the SURE program is unable to solve the model in a reasonable amount of time, the PAWS or STEM programs may be used to solve the model. However, these programs assume that all recoveries are exponentially distributed.

16. Sequences of Reliability Models

The SURE program provides the user with the capability to calculate and store the probability of terminating in each operational state of the model, as well as the death state probabilities. The program also allows the user to initialize a model by using these same operational state probabilities. These features support the use of sequences of reliability models to model systems with phased missions or non-constant failure rates.

16.1. Phased Missions

Many systems exhibit different failure behaviors or operational characteristics during different phases of a mission. For example, a spacecraft may experience considerably higher component failure rates during lift-off than in the weightless, benign environment of space. Also, the failure of a particular component may be catastrophic only during a specific phase, such as the 3-min landing phase of an aircraft. In a phased-mission solution, a model is solved for the first phase of the mission. The final probabilities of the operational states are used to calculate the initial state probabilities for a second model. (The second model usually differs from the first model in some manner.) This process is repeated for all phases in the mission.

The SURE program reports upper and lower bounds on the operational states, just as for the death states. The bounds of the operational states are not as close as the death state bounds, but are usually acceptable. The upper and lower bounds on recovery states (states with fast transitions leaving them) are usually not very close together. Fortunately, recovery states usually have operational bounds that are several orders of magnitude lower than the other states in the model because systems typically spend a very small percentage of their operational time performing recoveries. Thus, the crudeness of the bounds for the recovery states in early phases does not lead to an excessive separation of the final death state bounds. In other words, the crude operational recovery state probabilities will usually result in only a small separation of the final bounds obtained in phased-mission calculations. Although the bounds may sometimes be unacceptably far apart, they will always be mathematically correct.

Suppose we have a system that operates in two basic phases—cruise and landing. The system is implemented with a triad of processors and two warm spares. For simplicity, perfect detection of spare failure is assumed. During the cruise phase, which lasts for 2 hr, the system reconfigures by sparing and degradation. After the cruise phase, the system goes into a landing phase, which lasts 3 min. During the landing phase, the workload on the machines is so high that the additional processing that would be needed to perform reconfiguration cannot be tolerated. Therefore, the system is designed to turn off the reconfiguration processes during this phase.

To model this two-phased mission, a different models must be created for each phase. The following ASSIST input describes a model for the cruise phase:

```
NSI = 2;                      (* Number of spares initially *)
LAMBDA = 1E-4;                (* Failure rate of active processors *)
GAMMA = 1E-6;                 (* Failure rate of spares *)
TIME = 2.0;                   (* Mission time *)

MU = 7.9E-5;                  (* Mean time to replace with spare *)
SIGMA = 2.56E-5;              (* Stan. dev. of time to replace with spare *)

MU_DEG = 6.3E-5;              (* Mean time to degrade to simplex *)
SIGMA_DEG = 1.74E-5;          (* Stan. dev. of time to degrade to simplex *)

SPACE = (NW: 0..3,            (* Number of working processors *)
         NF: 0..3,            (* Number of failed active procssors *)
         NS: 0..NSI);         (* Number of spares *)

START = (3,0,NSI);

LIST=3;

IF NW > 0                                (* A processor can fail *)
   TRANTO (NW-1,NF+1,NS) BY NW*LAMBDA;

IF (NF > 0) AND (NS > 0)                 (* A spare becomes active *)
   TRANTO (NW+1,NF-1,NS-1) BY <MU,SIGMA>;

IF (NF > 0) AND (NS = 0)              (* No more spares, degrade to simplex *)
```

```
   TRANTO (1,0,0) BY <MU_DEG,SIGMA_DEG>;

IF NS > 0                                (* A spare fails and is detected *)
   TRANTO (NW,NF,NS-1) BY NS*GAMMA;

DEATHIF NF >= NW;
```

The ASSIST program generates the following SURE model.

```
NSI = 2;
LAMBDA = 1E-4;
GAMMA = 1E-6;
TIME = 2.0;
MU = 7.9E-5;
SIGMA = 2.56E-5;
MU_DEG = 6.3E-5;
SIGMA_DEG = 1.74E-5;
LIST = 3;

      2(* 3,0,2 *),     3(* 2,1,2 *)        = 3*LAMBDA;
      2(* 3,0,2 *),     4(* 3,0,1 *)        = 2*GAMMA;
      3(* 2,1,2 *),     4(* 3,0,1 *)        = <MU,SIGMA>;
      3(* 2,1,2 *),     1(* 1,2,2 DEATH  *) = 2*LAMBDA;
      3(* 2,1,2 *),     5(* 2,1,1 *)        = 2*GAMMA;
      4(* 3,0,1 *),     5(* 2,1,1 *)        = 3*LAMBDA;
      4(* 3,0,1 *),     6(* 3,0,0 *)        = 1*GAMMA;
      5(* 2,1,1 *),     6(* 3,0,0 *)        = <MU,SIGMA>;
      5(* 2,1,1 *),     1(* 1,2,1 DEATH  *) = 2*LAMBDA;
      5(* 2,1,1 *),     7(* 2,1,0 *)        = 1*GAMMA;
      6(* 3,0,0 *),     7(* 2,1,0 *)        = 3*LAMBDA;
      7(* 2,1,0 *),     8(* 1,0,0 *)        = <MU_DEG,SIGMA_DEG>;
      7(* 2,1,0 *),     1(* 1,2,0 DEATH  *) = 2*LAMBDA;
      8(* 1,0,0 *),     1(* 0,1,0 DEATH  *) = 1*LAMBDA;

(* NUMBER OF STATES IN MODEL = 8 *)
(* NUMBER OF TRANSITIONS IN MODEL = 14 *)
(* 4 DEATH STATES AGGREGATED INTO STATE 1 *)
```

The model for the second phase (`phaz2.mod`) is easily created with a text editor by deleting the reconfiguration transitions and changing the mission time to 0.05 hr. The resulting file is

```
NSI = 2;
LAMBDA = 1E-4;
GAMMA = 1E-6; TIME = 2.0; LIST = 3;
      2(* 3,0,2 *),     3(* 2,1,2 *)        = 3*LAMBDA;
      2(* 3,0,2 *),     4(* 3,0,1 *)        = 2*GAMMA;

      3(* 2,1,2 *),     1(* 1,2,2 DEATH  *) = 2*LAMBDA;
      3(* 2,1,2 *),     5(* 2,1,1 *)        = 2*GAMMA;
      4(* 3,0,1 *),     5(* 2,1,1 *)        = 3*LAMBDA;
      4(* 3,0,1 *),     6(* 3,0,0 *)        = 1*GAMMA;

      5(* 2,1,1 *),     1(* 1,2,1 DEATH  *) = 2*LAMBDA;
      5(* 2,1,1 *),     7(* 2,1,0 *)        = 1*GAMMA;
      6(* 3,0,0 *),     7(* 2,1,0 *)        = 3*LAMBDA;

      7(* 2,1,0 *),     1(* 1,2,0 DEATH  *) = 2*LAMBDA;
      8(* 1,0,0 *),     1(* 0,1,0 DEATH  *) = 1*LAMBDA;

(* NUMBER OF STATES IN MODEL = 8 *)
```

```
(* NUMBER OF TRANSITIONS IN MODEL = 14 *)
(* 4 DEATH STATES AGGREGATED INTO STATE 1 *)
```

The SURE program is then executed on the first model (stored in file `phaz.mod`) by using the `LIST = 3` option. This operation causes the SURE program to output all operational state probabilities, as well as the death state probabilities. This output is

```
  SURE V7.2   NASA Langley Research Center
  1? read0 phaz
 31? run
 MODEL FILE = phaz.mod                      SURE V7.2 11 Jan 90   13:56:49

DEATHSTATE    LOWERBOUND    UPPERBOUND    COMMENTS                  RUN #1
----------   -----------   -----------    ---------------------------------
     1       9.35692e-12   9.48468e-12
TOTAL        9.35692e-12   9.48468e-12

OPER-STATE    LOWERBOUND    UPPERBOUND
----------   -----------   -----------
      2       9.99396e-01   9.99396e-01
      3       0.00000e+00   1.53952e-06
      4       6.02277e-04   6.03819e-04
      5       0.00000e+00   1.43291e-09
      6       1.80332e-07   1.81768e-07
      7       0.00000e+00   5.59545e-13
      8       3.57995e-11   3.63591e-11

20 PATH(S) PROCESSED
0.617 SECS. CPU TIME UTILIZED
32? exit
```

The SURE program also creates a file containing these operational and death state probabilities in a format that can be used to initialize the states for the next phase. The SURE program names the file `phaz.ini`, that is, it adds `.ini` to the file name. The contents of this file generated by the previous run is

```
INITIAL_PROBS(
    1: ( 9.35692e-12, 9.48468e-12),
    2: ( 9.99396e-01, 9.99396e-01),
    3: ( 0.00000e+00, 1.53952e-06),
    4: ( 6.02277e-04, 6.03819e-04),
    5: ( 0.00000e+00, 1.43291e-09),
    6: ( 1.80332e-07, 1.81768e-07),
    7: ( 0.00000e+00, 5.59545e-13),
    8: ( 3.57995e-11, 3.63591e-11)
 );
```

Next, the SURE program is executed on the second model. The state probabilities are initialized with the SURE `INITIAL_PROBS` command. The second model must number its states in a manner equivalent to the first model. Note that the output of the `.ini` file is in the correct format for the SURE program.

```
$ sure
  SURE V7.2   NASA Langley Research Center
  1? read0 phaz2
```

```
 31? read phaz.ini

 32: INITIAL_PROBS(
 33:     1: ( 9.35692e-12, 9.48468e-12),
 34:     2: ( 9.99396e-01, 9.99396e-01),
 35:     3: ( 0.00000e+00, 1.53952e-06),
 36:     4: ( 6.02277e-04, 6.03819e-04),
 37:     5: ( 0.00000e+00, 1.43291e-09),
 38:     6: ( 1.80332e-07, 1.81768e-07),
 39:     7: ( 0.00000e+00, 5.59545e-13),
 40:     8: ( 3.57995e-11, 3.63591e-11)
 41:   );

 42? run

 MODEL FILE = phaz.ini                    SURE V7.2 11 Jan 90   13:58:12

DEATHSTATE    LOWERBOUND    UPPERBOUND    COMMENTS                  RUN #1
----------   -----------   -----------    --------------------------------
     1       8.43564e-11   9.98944e-11
TOTAL        8.43564e-11   9.98944e-11

OPER-STATE    LOWERBOUND    UPPERBOUND
----------   -----------   -----------
     2       9.99381e-01   9.99381e-01
     3       1.49908e-05   1.65304e-05
     4       6.02368e-04   6.03910e-04
     5       9.03554e-09   1.04918e-08
     6       1.80359e-07   1.81795e-07
     7       2.70540e-12   3.28658e-12
     8       3.57993e-11   3.63589e-11

 9 PATH(S) PRUNED AT LEVEL  1.49540e-16
 SUM OF PRUNED STATES PROBABILITY <  5.04017e-18

9 PATH(S) PROCESSED
0.417 SECS. CPU TIME UTILIZED
43?
```

16.2. Nonconstant Failure Rates

In section 16.1, a two-phased mission that required different models for each phase was analyzed. A related situation occurs when the structure of the model remains the same, but some parameters, such as the failure rates, change from one phase to another.

Consider a triad with warm spares (see section 9.2) that experiences different failure rates for each phase.

1. Phase 1: 6 min, $\lambda = 2 \times 10^{-4}$, $\gamma = 10^{-4}$
2. Phase 2: 2 hr, $\lambda = 10^{-4}$, $\gamma = 10^{-5}$
3. Phase 3: 3 min, $\lambda = 10^{-3}$, $\gamma = 10^{-4}$

Immediate detection of spare failure will be assumed for simplicity.

The same SURE model can be used for all phases and the user can be prompted for the parameter values by using the SURE `INPUT` command:

```
INPUT LAMBDA, GAMMA, TIME;
```

The full SURE model, which is stored in file `phase.mod`, is

```
INPUT LAMBDA, GAMMA, TIME;
NSI = 2;
MU = 7.9E-5;
SIGMA = 2.56E-5;
MU_DEG = 6.3E-5;
SIGMA_DE = 1.74E-5;
LIST = 3;
QTCALC = 1;

    2(* 3,0,2 *),    3(* 2,1,2 *) = 3*LAMBDA;
    2(* 3,0,2 *),    4(* 3,0,1 *) = 2*GAMMA;
    3(* 2,1,2 *),    1(* 1,2,2 *) = 2*LAMBDA;
    3(* 2,1,2 *),    4(* 3,0,1 *) = <MU,SIGMA>;
    3(* 2,1,2 *),    5(* 2,1,1 *) = 2*GAMMA;
    4(* 3,0,1 *),    5(* 2,1,1 *) = 3*LAMBDA;
    4(* 3,0,1 *),    6(* 3,0,0 *) = 1*GAMMA;
    5(* 2,1,1 *),    1(* 1,2,1 *) = 2*LAMBDA;
    5(* 2,1,1 *),    6(* 3,0,0 *) = <MU,SIGMA>;
    5(* 2,1,1 *),    7(* 2,1,0 *) = 1*GAMMA;
    6(* 3,0,0 *),    7(* 2,1,0 *) = 3*LAMBDA;
    7(* 2,1,0 *),    1(* 1,2,0 *) = 2*LAMBDA;
    7(* 2,1,0 *),    8(* 1,0,0 *) = <MU_DEG,SIGMA_DEG>;
    8(* 1,0,0 *),    1(* 0,1,0 *) = 1*LAMBDA;
```

The `QTCALC = 1` command causes the SURE program to use more accurate (but slower) numerical routines. This increased accuracy is often necessary when analyzing phased missions. The interactive session follows:

```
SURE V7.2   NASA Langley Research Center

 1? read0 phase

    LAMBDA? 2e-4

    GAMMA? 1e-4

    TIME? .1

30? run

MODEL FILE = phase.mod                    SURE V7.2 12 Jan 90   09:35:50

TIME =  1.000e-01,  GAMMA =  1.000e-04,  LAMBDA =  2.000e-04,

DEATHSTATE    LOWERBOUND    UPPERBOUND    COMMENTS                  RUN #1
----------   -----------   -----------    ---------------------------------
     1       1.78562e-12   1.89600e-12    <ExpMat>

TOTAL        1.78562e-12   1.89600e-12    <ExpMat - 14,14>

OPER-STATE    LOWERBOUND    UPPERBOUND
----------   -----------   -----------
     2       9.99920e-01   9.99920e-01    <ExpMat>
     3       0.00000e+00   9.98043e-07    <ExpMat>
     4       7.89960e-05   7.99941e-05    <ExpMat>
     5       0.00000e+00   1.14966e-10    <ExpMat>
     6       2.67751e-09   2.80076e-09    <ExpMat>
```

```
     7       0.00000e+00   5.17358e-15    <ExpMat>
     8       5.08706e-14   5.60442e-14    <ExpMat>

 10 PATH(S) PRUNED AT LEVEL  4.75740e-20
 SUM OF PRUNED STATES PROBABILITY <  6.11113e-20
 Q(T) ACCURACY >= 14 DIGITS

10 PATH(S) PROCESSED
2.867 SECS. CPU TIME UTILIZED
31? read0 phase

    LAMBDA? 1e-4

    GAMMA? 1e-5

    TIME? 2.0

60? read phase.ini

61: INITIAL_PROBS(
62:     1: ( 1.78562e-12, 1.89600e-12),
63:     2: ( 9.99920e-01, 9.99920e-01),
64:     3: ( 0.00000e+00, 9.98043e-07),
65:     4: ( 7.89960e-05, 7.99941e-05),
66:     5: ( 0.00000e+00, 1.14966e-10),
67:     6: ( 2.67751e-09, 2.80076e-09),
68:     7: ( 0.00000e+00, 5.17358e-15),
69:     8: ( 5.08706e-14, 5.60442e-14)
70:   );

      0.07 SECS. TO READ MODEL FILE  71? run

MODEL FILE = phase.ini                   SURE V7.2 12 Jan 90   09:36:19

TIME =  2.000e+00,  GAMMA =  1.000e-05,  LAMBDA =  1.000e-04,

DEATHSTATE    LOWERBOUND    UPPERBOUND    COMMENTS                  RUN #2
----------   -----------   -----------    ---------------------------------
     1       1.11438e-11   1.13950e-11    <ExpMat>

TOTAL        1.11438e-11   1.13950e-11    <ExpMat - 14,14>

OPER-STATE    LOWERBOUND    UPPERBOUND
----------   -----------   -----------
     2       9.99280e-01   9.99280e-01    <ExpMat>
     3       0.00000e+00   2.35621e-06    <ExpMat>
     4       7.17134e-04   7.20490e-04    <ExpMat>
     5       0.00000e+00   2.82024e-09    <ExpMat>
     6       2.48355e-07   2.51362e-07    <ExpMat>
     7       0.00000e+00   1.19806e-12    <ExpMat>
     8       5.53210e-11   5.65243e-11    <ExpMat>
 30 PATH(S) PRUNED AT LEVEL  4.61326e-19
 SUM OF PRUNED STATES PROBABILITY <  1.15985e-18
 Q(T) ACCURACY >= 14 DIGITS

19 PATH(S) PROCESSED
4.267 SECS. CPU TIME UTILIZED
72? read0 phase

    LAMBDA? 1e-3

    GAMMA? 1e-4
```

```
     TIME? 0.05

101? read phase.ini

102: INITIAL_PROBS(
103:     1: ( 1.11438e-11, 1.13950e-11),
104:     2: ( 9.99280e-01, 9.99280e-01),
105:     3: ( 0.00000e+00, 2.35621e-06),
106:     4: ( 7.17134e-04, 7.20490e-04),
107:     5: ( 0.00000e+00, 2.82024e-09),
108:     6: ( 2.48355e-07, 2.51362e-07),
109:     7: ( 0.00000e+00, 1.19806e-12),
110:     8: ( 5.53210e-11, 5.65243e-11)
111:   );

112? run

 MODEL FILE = phase.ini                    SURE V7.2 12 Jan 90   09:36:57

 TIME =  5.000e-02,  GAMMA =  1.000e-04,  LAMBDA =  1.000e-03,

 DEATHSTATE    LOWERBOUND    UPPERBOUND    COMMENTS                  RUN #3
 ----------   -----------   -----------    ---------------------------------
      1       3.29083e-11   3.54718e-11    <ExpMat>

 TOTAL        3.29083e-11   3.54718e-11    <ExpMat - 14,14>

 OPER-STATE    LOWERBOUND    UPPERBOUND
 ----------   -----------   -----------
      2       9.99120e-01   9.99120e-01    <ExpMat>
      3       0.00000e+00   6.30518e-06    <ExpMat>
      4       8.72933e-04   8.82595e-04    <ExpMat>
      5       0.00000e+00   7.50836e-09    <ExpMat>
      6       3.68000e-07   3.78561e-07    <ExpMat>
      7       0.00000e+00   3.72751e-12    <ExpMat>
      8       9.99350e-11   1.04866e-10    <ExpMat>

  33 PATH(S) PRUNED AT LEVEL  8.23385e-18
  SUM OF PRUNED STATES PROBABILITY <  3.35190e-17
  Q(T) ACCURACY >= 14 DIGITS

 13 PATH(S) PROCESSED
 3.350 SECS. CPU TIME UTILIZED
113? exit
```

As in section 16.2, the results of each previous phase are loaded by reading the `.ini` file created by the previous run. The `<ExpMat>` output in the `COMMENTS` field indicates that the more accurate `QTCALC=1` numerical routines were utilized.

16.3. Continuously Varying Failure Rates

Suppose that the failure rates change continuously with time as shown in figure 61. This type of failure rate is called a decreasing failure rate. The SURE program cannot handle this failure rate directly because it leads to nonhomogenous or nonstationary Markov models. Nonhomogenous Markov models are more general than pure Markov models in that they allow the transition rates to vary as a function of global time. This generalization is different from the semi-Markov model first discussed in section 4.1. However, good results can be obtained by using the phased-mission approach on a linearized upper bound as shown in figure 62.

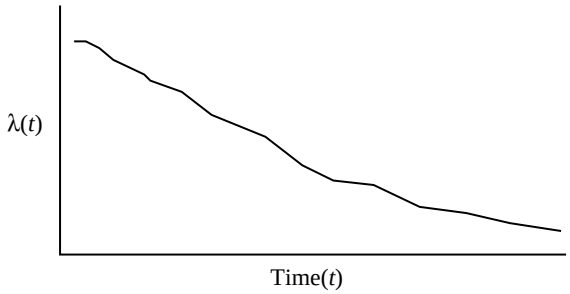


Figure 61. Decreasing failure rate function.

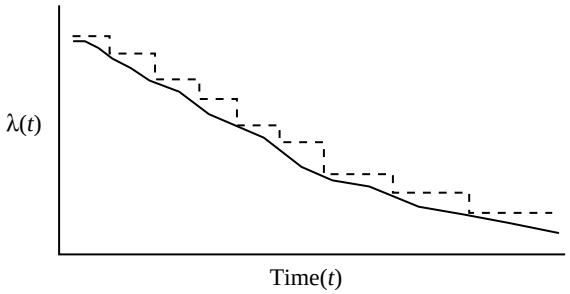


Figure 62. Upper bound on failure rate functions.

The solution of the linearized model requires nine steps, but is quite easy with the use of the `.ini` files. Because an upper bound is used for the failure rate, the result will be conservative. The problem can then be solved again by using a consistently lower bound on the failure rate function to obtain a lower bound on the system failure probability.

17. Concluding Remarks

This paper is intended to serve as a tutorial for reliability engineers who are learning how to develop Markov models of fault-tolerant systems. A number of techniques for developing reliability models of fault-tolerant systems have been presented. Various modeling techniques have been presented in a systematic way, building from simple systems to more complicated ones. Techniques for modeling specific aspects, such as single-point failures, near-coincident failures, transient fault recoveries, and cold spares have been discussed. However, it must be recognized that there is no "right" way to model a system. Many valid ways to model a given system exist, and choosing which method will result in an efficient, informative model is sometimes more of an art than a science.

Including every minute detail in a reliability model of a complex system is impossible, because such a model would be exorbitantly large. It is not even possible to completely understand and measure the reliability behavior of a system in minute detail. Therefore, the reliability engineer must make certain assumptions about the behavior of a system. Some of these assumptions are immediately obvious; while others must be demonstrated or proven correct.

Markov modeling can be a very powerful reliability analysis tool for three reasons. First, Markov models provide the reliability engineer the flexibility to include a variety of assumptions and behaviors. Second, the reliability engineer is fully aware of the assumptions being made because they are made explicitly. And third, the reliability engineer can estimate the effects of those assumptions on the system failure probability calculations.

However, reliability analysis requires a certain level of expertise that cannot be easily automated. The use of an automated tool that makes implicit assumptions can be dangerous. Even if the engineer completely understands what implicit assumptions a tool can make, these assumptions are likely to be forgotten if they are not made visible. For this reason, the ASSIST program is designed to generate exactly the model described in the input language and to not make any implicit assumptions. Thus, ASSIST includes all the flexibility of Markov models. It also requires the same level of modeling expertise.

NASA Langley Research Center
Hampton, VA 23681-0001
April 21, 1995

Appendix

Additional SURE Mathematics

In this appendix, additional important aspects of the SURE mathematical foundation are presented. The methods described in this section were not published in reference 1, but were discussed in reference 9.

A1. Technique for Solving Models With Loops Using SURE

Although the bounding theorem of White is only concerned with paths that do not contain a loop, the SURE program uses a strategy based upon pruning to solve models with loops. A model containing loops (its graph structure contains a circuit) has an infinite number of paths. Consider the model in figure A1. Unfolding the loop produces an infinite sequence of paths as shown in figure A2.

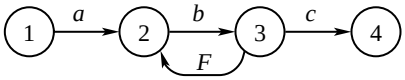


Figure A1. Model with loop.

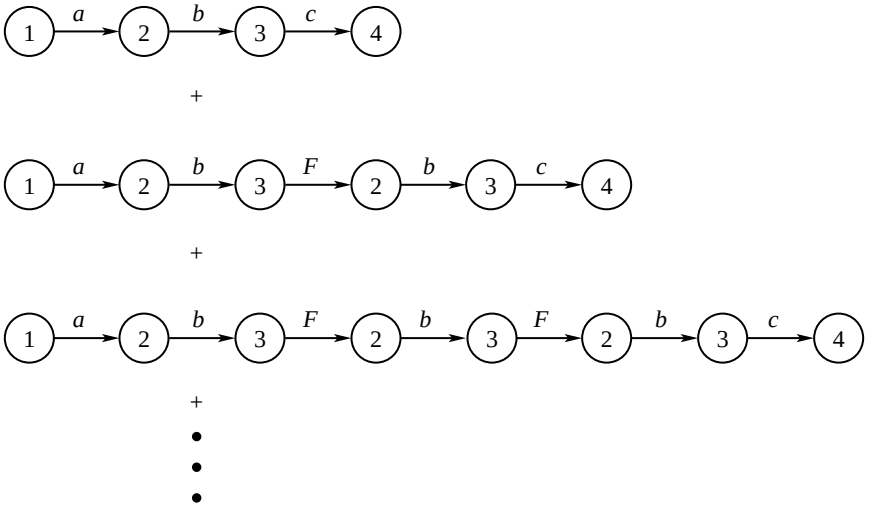


Figure A2. Infinite sequence of paths.

However, the situation is not intractable because the resulting sequence consists of increasingly longer paths. The death state probabilities of the paths decrease rapidly like a Taylor's series. A typical sequence of probabilities would be

$$10^{-6}, 10^{-10}, 10^{-14}, 10^{-18}, 10^{-22}, \ldots$$

The series can be truncated at a point where the sum of all subsequent probabilities becomes negligible. The SURE program accomplishes this truncation in a manner that enables a calculation of an upper and a lower bound on the truncation error. The technique is based on a model with a loop that is equivalent to a finite sequence of paths where only the last path in the sequence contains a loop. For example, the model shown in figure A1 can be reduced to the three paths shown in figure A3. The

probability of entering the death state of the original path is equal to the sum of the probabilities of entering the death state of the three new paths.

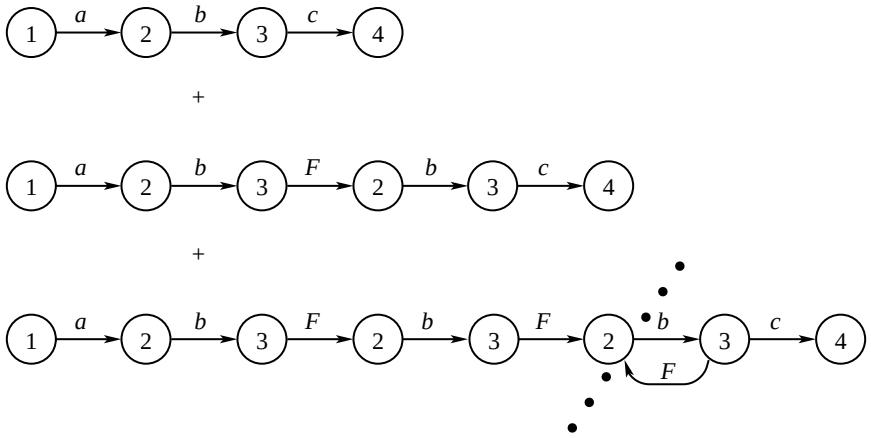


Figure A3. Reduction to finite set of paths.

If the third path is pruned before the loop, then an upper bound can be obtained. Consequently, the sum of the three paths shown in figure A4 provides an upper bound on the original model of figure A1. Thus, a finite unfolding of a loop in conjunction with pruning can be used to solve a model with loops. This is precisely the technique implemented in the SURE program.

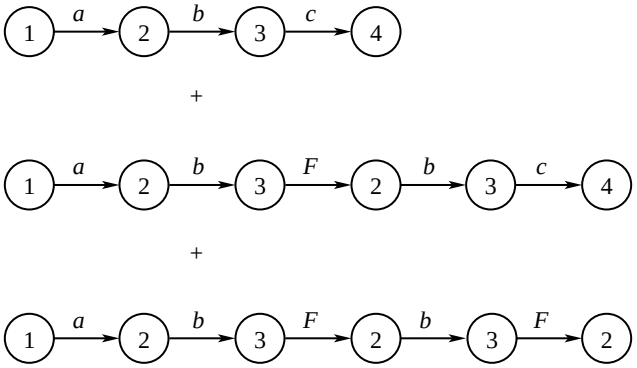


Figure A4. Model after pruning and reduction.

A2. Solving Models With Distributed Initial State Probabilities

Suppose that we need to solve a model, whose initial state is not exactly determined, needed to be solved. Suppose, for example, that the probability that the system is in state (1) at time 0 = 0.5 and that the initial probabilities for states (2) and (3) are 0.3 and 0.2, respectively. The SURE program solves this model with the following strategy. The model is solved three times—first with state (1) as the start state, then with state (2) as the start state, and finally with state (3) as the start state. Each preliminary result is multiplied by the initial probability of its start state, then the resulting values are added together.

The SURE input language contains an `INITIAL_P` statement for initializing states. For example,

```
INITIAL_P(1: 0.3, 2: 0.7);
```

assigns an initial probability of 0.3 to state (1) and an initial probability of 0.7 to state (2). The user may also specify upper and lower bounds on the initial state probabilities:

```
INITIAL_P(1: (0.27,0.31), 2: (0.69,0.71));
```

A3. Operational State Probabilities

The SURE program can also specify the bound of the operational state probabilities. Given the model shown in figure A5, the program first calculates the death state probabilities $p_8(T)$ and $p_9(T)$. Next, the program solves the reduced model shown in figure A6, obtaining $p_{7'}(T)$.

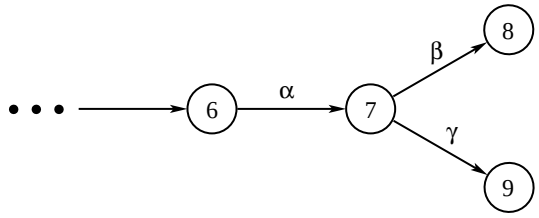


Figure A5. Model of SURE calculations for operational state probabilities.

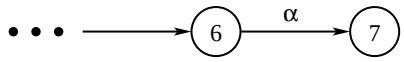


Figure A6. Reduced model.

The probability of being in the operational state (7), $p_7(T)$ in the original model, can be calculated as follows:

$$p_7(T) = p_{7'}(T) - [p_8(T) + p_9(T)]$$

Although this process might seem to be laborious, it is efficiently implemented in SURE. All basic probabilities are calculated while SURE is traversing the graph structure of the model. Because the bounds are algebraic, the calculations are not costly. The subtractions are performed as SURE backs out of the recursions after reaching a death state. Of course, the calculations are performed by using bounds rather than exact probabilities. The bounds on the operational states are not as close as the bounds on the death states, but are usually close enough to be useful for the solution of a sequence of semi-Markov models used for phased missions. The closer a state is to a death state, the closer the bounds are.

References

1. Butler, Ricky W.; and White, Allan L.: *SURE Reliability Analysis—Program and Mathematics*. NASA TP-2764, 1988.

2. Henley, Ernest J.; and Kumamoto, Hiromitsu: *Reliability Engineering and Risk Assessment*. Prentice-Hall, Inc., 1981.

3. Butler, Ricky W.; and Martensen, Anna L.: *The Fault-Tree Compiler (FTC)—Program and Mathematics*. NASA TP-2915, 1989.

4. Ross, Sheldon M.: *Applied Probability Models With Optimization Applications*. Holden-Day, 1970.

5. U.S. Department of Defense: *Reliability Prediction of Electronic Equipment*. MIL-HDBK-217F, Apr. 1979.

6. Siewiorek, Daniel P.; and Swarz, Robert S.: *The Theory and Practice of Reliable System Design*. Digital Press, 1982.

7. Trivedi, Kishor Shridharbhai: *Probability and Statistics With Reliability, Queuing, and Computer Science Applications*. Prentice-Hall, Inc., 1982.

8. Krishna, C. M.; Shin, K. G.; and Butler, R. W.: Synchronization and Fault-Masking in Redundant Real-Time Systems. *Proceedings of the Fourteenth International Conference on Fault-Tolerant Computing*, IEEE, 1984, pp. 152–157.

9. Butler, Ricky W.: The SURE Approach to Reliability Analysis. *IEEE Trans. Reliab.*, vol. 41, no. 2, June 1992, pp. 210–218.

10. White, Allan L.: *Synthetic Bounds for Semi-Markov Reliability Models*. NASA CR-178008, 1985.

11. Butler, Ricky W.; and Stevenson, Philip H.: *The PAWS and STEM Reliability Analysis Programs*. NASA TM-100572, 1988.

12. White, Allan L.: Reliability Estimation for Reconfigurable Systems With Fast Recovery. *Microelectron. & Reliab.*, vol. 26, no. 6, 1986, pp. 1111–1120.

13. White, Allan L.: *Upper and Lower Bounds for Semi-Markov Reliability Models of Reconfigurable Systems*. NASA CR-172340, 1984.

14. Goldberg, Jack; Kautz, William H.; Melliar-Smith, P. Michael; Green, Milton W.; Levitt, Karl N.; Schwartz, Richard L.; and Weinstock, Charles B.: *Development and Analysis of the Software Implemented Fault-Tolerance (SIFT) Computer.* NASA CR-172146, 1984.

15. McGough, John G.; and Swern, Fred L.: *Measurement of Fault Latency in a Digital Avionic Mini Processor.* NASA CR-3462, 1981.

16. Bavuso, S. J.; and Petersen, P. L.: *CARE III Model Overview and User's Guide (First Revision)*. NASA TM-86404, 1985.

17. Johnson, S. C.: *ASSIST User Manual*. NASA TM-4592, 1994.

18. Johnson, Sally C.: Reliability Analysis of Large, Complex Systems Using ASSIST. *A Collection of Technical Papers, AIAA/IEEE 8th Digital Avionics Systems Conference*, Part 1, Oct. 1988, pp. 227–234. (Available as AIAA-88-3898-CP.)

19. Feller, William: *An Introduction to Probability Theory and Its Applications, Volume II*. John Wiley & Sons, Inc., 1966.

20. Pease, M.; Shostak, R.; and Lamport, L.: Reaching Agreement in the Presence of Faults. *J. ACM*, vol. 27, no. 2, Apr. 1980, pp. 228–234.

21. Lamport, Leslie; Shostak, Robert; and Pease, Marshall: The Byzantine Generals Problem. *ACM Trans. Program. Lang. & Syst.*, vol. 4, no. 3, July 1982, pp. 382–401.

22. Azadmanesh, A.; and Kieckhafer, R.: The General Convergence Problem: A Unification of Synchronous and Asynchronous Systems. *Fourth International Working Conference on Dependable Computing for Critical Applications*, IFIP Working Group, Jan. 1994, pp. 168–182.

23. Hopkins, Albert L., Jr.; Smith, T. Basil, III; and Lala, Jaynarayan H.: FTMP—A Highly Reliable Fault-Tolerant Multiprocessor for Aircraft. *Proc. IEEE*, vol. 66, no. 10, Oct. 1978, pp. 1221–1239.

24. White, Allan L.; and Palumbo, Daniel L.: *Model Reduction by Trimming for a Class of Semi-Markov Reliability Models and the Corresponding Error Bound*. NASA TP-3089, 1991.

25. White, Allan L.; and Palumbo, Daniel L.: State Reduction for Semi-Markov Reliability Models. *Annual Reliability and Maintainability Symposium—1990 Proceedings*, IEEE, 1990, pp. 280–285.

26. Butler, Ricky W.; and Elks, Carl R.: *A Preliminary Transient-Fault Experiment on the SIFT Computer System*. NASA TM-89058, 1987.

27. DiVito, Ben L.; Butler, Ricky W.; and Caldwell, James L.: *Formal Design and Verification of a Reliable Computing Platform for Real-Time Controls, Phase 1: Results*. NASA TM-102716, 1990.

28. Dugan, J. B.; Trivedi, K. S.; Smotherman, M. K.; and Geist, R. M.: The Hybrid Automated Reliability Predictor. *J. Guid., Control, & Dyn.*, vol. 9, May–June 1986, pp. 319–331.

29. Lala, J. H.; Alger, L. S.; Gauthier, R. J.; and Dzwonczyk, M. J.: *A Fault Tolerant Processor To Meet Rigorous Failure Requirements*. CSDL-P-2705, Charles Stark Draper Lab., Inc., July 1986.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	September 1995	Reference Publication

4. TITLE AND SUBTITLE
Techniques for Modeling the Reliability of Fault-Tolerant Systems With the Markov State-Space Approach

5. FUNDING NUMBERS
WU 505-64-10-07

6. AUTHOR(S)
Ricky W. Butler and Sally C. Johnson

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
NASA Langley Research Center
Hampton, VA 23681-0001

8. PERFORMING ORGANIZATION REPORT NUMBER
L-17425

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
National Aeronautics and Space Administration
Washington, DC 20546-0001

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
NASA RP-1348

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Unclassified–Unlimited
Subject Category 62
Availability: NASA CASI (301) 621-0390

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This paper presents a step-by-step tutorial of the methods and the tools that were used for the reliability analysis of fault-tolerant systems. The approach used in this paper is the Markov (or semi-Markov) state-space method. The paper is intended for design engineers with a basic understanding of computer architecture and fault tolerance, but little knowledge of reliability modeling. The representation of architectural features in mathematical models is emphasized. This paper does not present details of the mathematical solution of complex reliability models. Instead, it describes the use of several recently developed computer programs—SURE, ASSIST, STEM, and PAWS—that automate the generation and the solution of these models.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Reliability modeling; Markov models; Reliability analysis; Fault tolerance	130
	16. PRICE CODE: A07

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	