NASA Technical Memorandum 102623

THE ART OF FAULT-TOLERANT SYSTEM RELIABILITY MODELING

Ricky W. Butler and Sally C. Johnson

March 1990

(NASA-TM-102623) THE ART OF FAULT-TOLERANT SYSTEM RELIABILITY MODELING (NASA) 132 p CSCL 09B

N90-22292

Unclas

G3/62 0280237

National Aeronautics and Space Administration

Langley Research Center
Hampton, Virginia 23665-5225

Contents

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-hour 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; i.e. fault-tolerant computer systems. 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. A life-testing approach is typically used to determine the reliability or "lifetime" of a diversity of products such as light bulbs, batteries, and electronic devices. The life-testing methodology is clearly impractical, though, for computer systems with reliability goals of the order $1 - 10^{-7}$ 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. Since 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 which enable operation in the presence of failing components. Since current manufacturing techniques cannot produce circuitry with adequate reliability to meet ultra-high reliability requirements, highly reliable systems use redundancy techniques, such as parallel redundant units or dissimilar algorithms for computing the same function, to achieve their 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 that can be done is to carefully develop the reliability model and subject it to scrupulous 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. It is the goal of this paper 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 which are followed by increasingly more complex concepts. Thus, the paper begins with an overview of essential aspects of Markov models. Next, the fundamental techniques used for modeling the non-reconfigurable systems are developed. Then, the basic techniques used in modeling reconfigurable systems are explored. Before pressing on to more complicated models, the computer program SURE (Semi-Markov Unreliability Range Evaluator) [8], which can be used to solve the reliability models numerically, is introduced. Next, the two basic types of reconfiguration—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 for describing reliability models—the ASSIST language. This is necessary since the models presented in the later sections are very large and complex. If they were defined by enumerating the states and transitions of the model exhaustively, the reader would become exhausted. The expressiveness of the ASSIST language allows complex models to be defined in a succint and economical manner. Next, complex systems consisting of multiple triads using various forms of reconfiguration are investigated. Then the techniques used to model transient and intermittent faults are presented. The next section explores the techniques used to model the components of control system architectures including sensors, buses, actuators, etc. Finally, some specialized topics, such as sequence dependencies, phased missions, and non-constant failure rate models, are presented.

2 Introduction to Markov Modeling

Traditionally, the reliability analysis of a complex system consisting of many components has been accomplished using combinatorial mathematics. The standard "fault tree" method of reliability analysis is based on such mathematics. Unfortunately, the fault-tree approach is incapable of analyzing systems where reconfiguration is possible. In reconfigurable systems the critical factor often becomes the effectiveness of the dynamic reconfiguration process. It is necessary to model such systems using the more powerful Markov modeling technique.

The system is thus represented as consisting of a vector of attributes which change over time. A particular set of values of the attributes is called a "state" of the system. These attributes are typically system characteristics such as the number of working processors, the number of spare units, the number of faulty units which have not been removed, etc. The more attributes included in the model, the more complex the model will be. Thus, one typically tries to choose the smallest set of attributes that can accurately describe the fault-related behavior of the system. The next step in the modeling process is to characterize the transition time from one state to another. Since this transition time is rarely deterministic, the transition times are described using 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 make either conservative or non-conservative assumptions about what is system failure. If one wishes to say that the reliability of the system is 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 is conservative since the two faults may not actually corrupt data in such a way as to defeat the voter. This assumption simplifies the model since the probabilities of collusion between the faulty pair does not have to be modeled. If one wishes to say the reliability is no better than some value, then non-conservative assumptions are made. For example, the modeler assumes only certain parts of the system can fail.

It is important that all of the transitions in the reliability model be measurable. This 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 transitions can be obtained from field data and/or MIL-STD 217C calculations. The faster transition rates correspond to system responses to fault arrivals and can be measured experimentally using fault injection. The primary problem is to properly model the system so as to facilitate the determination of these transitions. If the model is too coarse the transitions become experimentally unobservable. If the model is too detailed the number of transitions which must be measured can be exorbitant.

In this introduction some of the issues of reliability modeling have been introduced. The goal of this paper is to explore the methods and assumptions used in the development of reliability models for fault-tolerant computer systems.

3 Modeling Non-Reconfigurable Systems

The simplest types of systems to model are non-reconfigurable systems. This section introduces the basic elements of reliability modeling by describing how to model simple non-reconfigurable systems ranging from a single simplex computer through a majority-voting N-modularly-redundant (NMR) system.

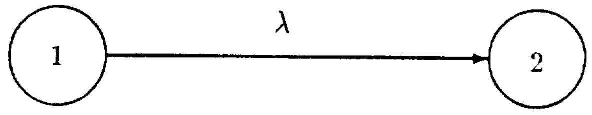


Figure 1: Model of a Simplex Computer

3.1 Simplex Computer

The first example is a system consisting of a single computer. First, we let T be a random variable representing the time to failure of the computer. Next, we must define a distribution for T, say $F(t)$. Typically, it is assumed that electronic components, and consequently computers, fail according to the exponential distribution:

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

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 is the only distribution with a constant hazard rate. The Markov model representing this system is given in figure 1. 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, it is generally assumed that electronic components 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 [1]. The reader is referred to the MIL-STD 217D handbook [2] for a more complete discussion on the problem of estimating the reliability of electronic components. Once the reliability of each chip in a computer is known, the computer's failure rate is simply the sum of the failure rates of the individual chips. To see this, suppose $\lambda_1, \lambda_2, \ldots, \lambda_n$ represent the failure rates of the chips 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 chip fails, the distribution of failure for the computer $F_c(t)$ is determined as follows:

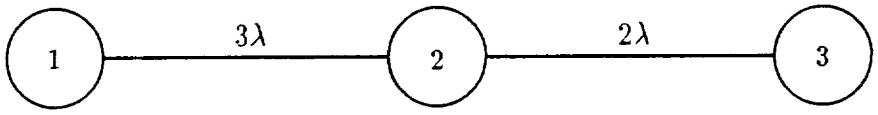


Figure 2: Model of a TMR System

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

If we assume that the chips fail independently, we have

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

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

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

3.2 Static Redundancy

The Triple-Modular Redundant (TMR) 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 such 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 propogate 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. 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 fail. The system is in state 2 when one processor has failed. The transition from state 2 to state 3 has rate 2λ since there are only two working computers that can fail. State 3 represents system failure because a majority of the computers in the system have failed.

In figure 3, the probability of system failure as a function of mission time is given. The failure rate λ is $10^{-4}/hour$. It can be seen that high reliability is strongly dependent on

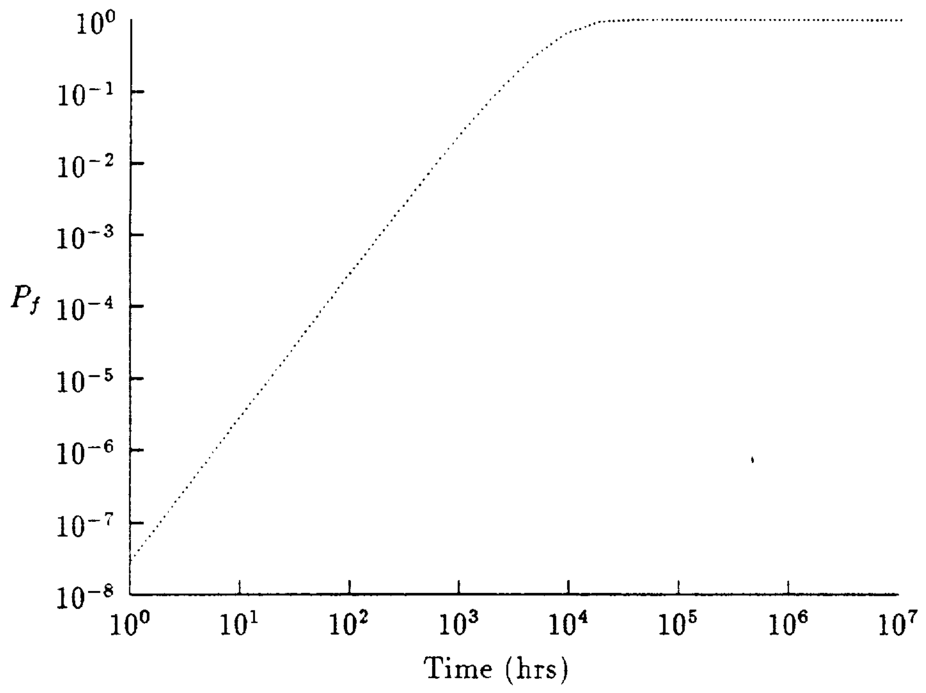


Figure 3: Probability as a Function of Mission Time

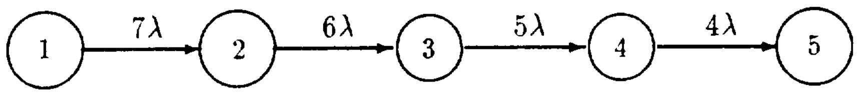


Figure 4: Model of 7MR System

a short mission time. It should be noted that it was implicitly assumed that the system starts with no failed components (i.e. Prob in state 1 at time 0 = 1). This is equivalent to assuming perfect maintenance between missions.

3.3 N-Modularly Redundant System

The assumptions of an N-modularly redundant system are the same as for 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 (figure 4) describes a 7-processor system with a 7-way voter. The probability of system failure as a function of mission time is given in figure 5. Figure 6 shows the unreliability of an NMR system as a function of N. Theoretically, the probability of system failure $\rightarrow 0$ as $N \rightarrow \infty$. 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, seriously increasing the likelihood of a critical task missing a hard deadline [3].

3.4 Fault Tree Analysis: A Few Comments

Fault trees have been used for the reliability analysis of complex system for many years. Nevertheless, it is important to recognize the limitations of the fault-tree method. Basically, a fault-tree can be used to model a system where there is no reconfiguration—the removal of a faulty component from the system. Thus, all of the examples in this section could have been modeled with a fault tree. In reconfigurable systems, the system attempts to remove faulty components before another failure occurs which could overcome the capabilities of the voters. (This will be explored in more detail in the next section.) The probability that the system will not succeed cannot be calculated with the combinatorial fault tree approach. The more powerful Markov state-space modeling approach is needed.

Although fault trees can only be used to solve a limited class of problems, these combinatorial solutions can be calculated quite efficiently using a fault-tree solver such as NASA Langley's Fault Tree Compiler (see [4]). Thus, a reliability engineer who frequently analyzes nonreconfigurable systems that can be solved combinatorically would be wise to use an efficient fault-tree solver whenever possible, because the combinatoric fault-tree computations

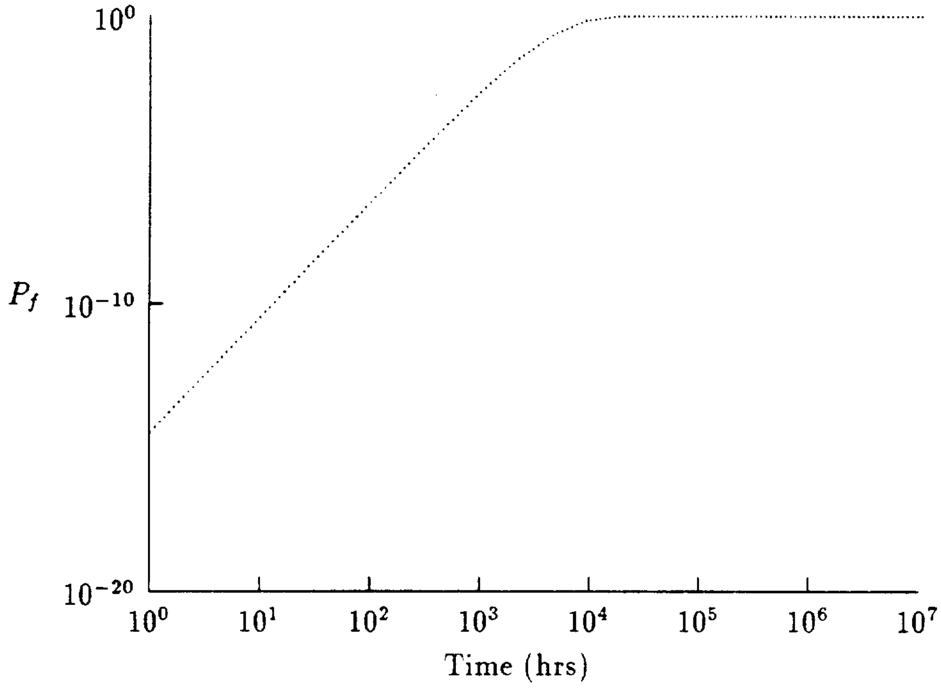


Figure 5: 7MR Unreliability As a Function of Mission Time

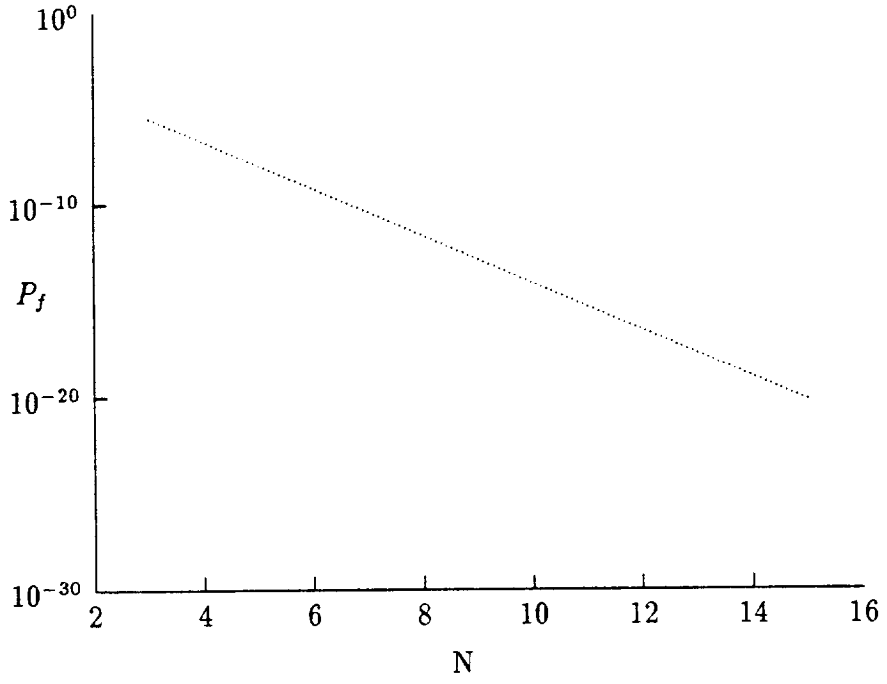


Figure 6: NMR Unreliability As a Function of N

are typically more efficient than are Markov solutions.

4 Modeling Reconfigurable Systems

Fault-tolerant systems are often designed using a strategy of reconfiguration. Reconfiguration strategies come in many varieties but always involve the logical or physical removal of a faulty component. The techniques used to identify the faulty component and the methods used to repair the system vary greatly and can lead to complex reliability models. There are two basic types of reconfiguration strategies—degradation and replacement with spares. The degradation method involves the permanent removal of a faulty component without replacement. The reconfigured system continues with a degraded set of components. The sparing method involves both the removal of faulty components and their replacement with a spare. In this section we will briefly introduce these concepts. They will be explored in greater detail in later sections.

4.1 Degradable Triad

The simplest architecture based upon majority voting is the triplex system. To increase the reliability of the system, triplex systems have been designed which reconfigure by degradation. The model of figure 7 describes the behavior of a simple degradable triplex system.

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. Note that since the processors are identical, we do not have to represent the failure of each processor 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 (i.e. reconfiguration) of the faulty processor. Reconfiguration transitions are labelled with a distribution function (e.g., F(t)) rather than a rate. The reason for this is that experimental measurement of the reconfiguration process has revealed that the distribution of recovery time is not exponential [1]. Consequently, the transition cannot be described by a constant failure rate. The meaning is simple—the probability that the transition time from state 2 to state 4 is less than t is $F(t)$. The presence of a non-exponential 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 a subsequent section, several computer programs will be discussed which can be used to solve Markov and semi-Markov models.

At state 4 the system is operational with two good processors. This transition 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, there is also a second transition

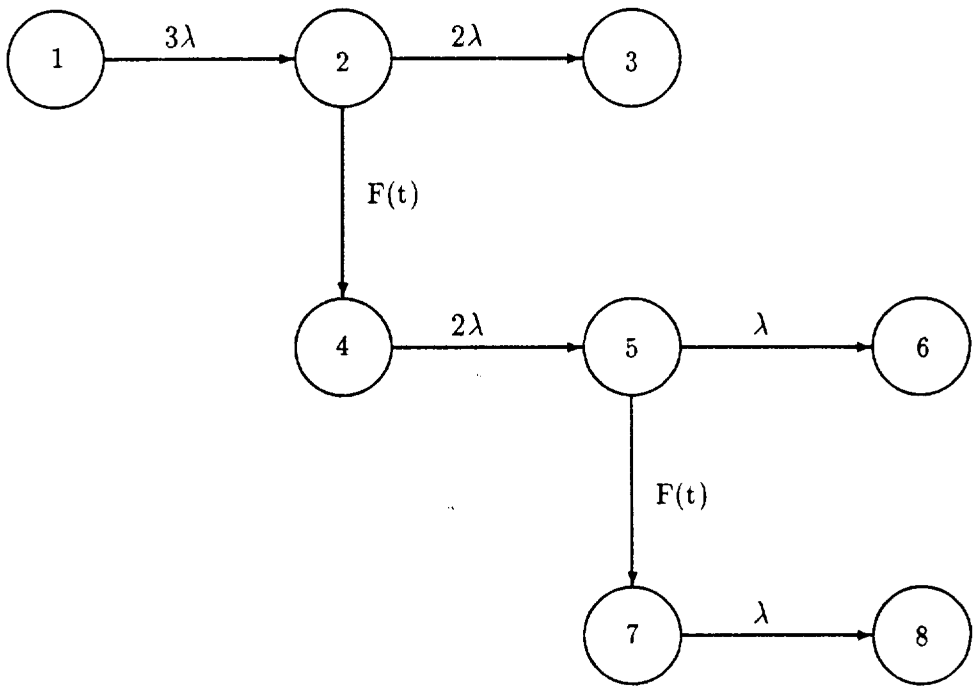


Figure 7: Model of Degradable Triplex

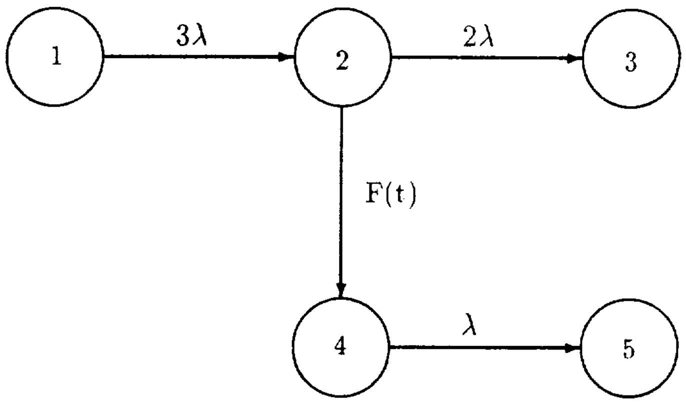


Figure 8: Model of Triplex to Simplex System

from state 2 to state 3 which represents the *coincident* failure of a second processor. The rate of this transition is 2λ, since either of the remaining two processors could fail. State 3 is a death state (i.e. an absorbing state) which represents failure of the system due to *near-coincident* failure.

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 taking the system to state 5. At state 5, once again, a race occurs between the reconfiguration process ending in state 7 and the failure of a second processor ending in state 6. State 6 is thus another death state and state 7 is the operational state where there is one remaining good processor. The transition from state 7 to 8 represents the failure of the last processor. At state 8 there are no good processors remaining, and the probability of reaching this death state is often referred to as *failure by exhaustion of parts.*

4.2 Triad to Simplex

The model presented in the previous section 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 100% accuracy. When using a majority voter on three or more processors, such an assumption is not unrealistic. However, with only two good processors, there is no way for this diagnosis to be perfect. In a later section we will explore the use of self-test programs to diagnose failure in a dual system. In this section, we will explore another option—avoid the dual mode, i.e. degrade directly from a triplex to a simplex system. The model of figure 8 describes this system.

As before the horizontal transitions represent fault arrivals. The vertical transition repre-

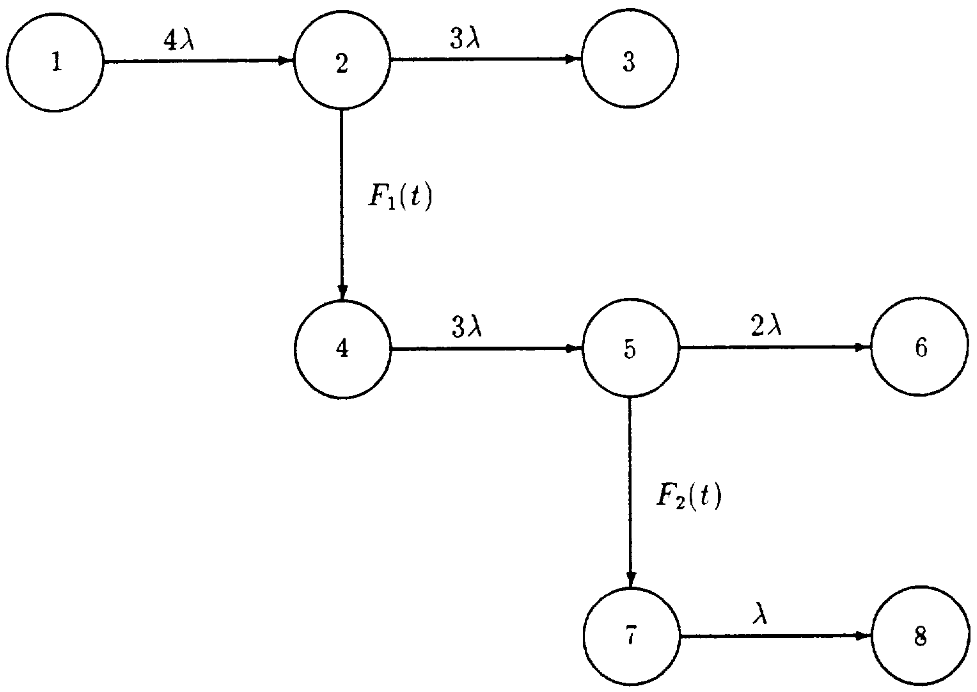


Figure 9: Model of a Degradable Quad

sents system recovery. The recovery transition is labelled 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 there are three active processors that can fail. When one of those processors fails, the system is in state two. Before reconfiguration occurs, there are still two active processors that 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 plus 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 Quad

The model in figure 9 describes a degradable quad. 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, leaving a triad of processors (state 4). When one of the three remaining processors fails (state 5), the reconfiguration process entails removal of the faulty processor plus one of the working processors, leaving a simplex (state 7). Note that a different function is used for the transition from state 5 to state 7 than from state 2

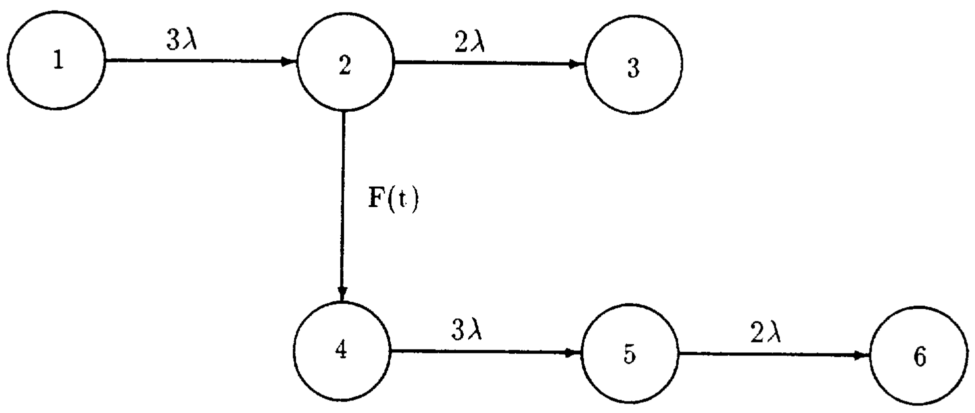


Figure 10: Model of Triplex with One Spare

to state 4. This is necessary if the reconfiguration process, and hence its 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 operation with degraded levels of redundancy. This section provides a brief introduction to the technique of sparing, i.e. replacing a faulty processor with a spare processor. This technique will be explored in detail in section 7.

Suppose we have a triplex system which has one spare that does not fail when it is inactive. The model of figure 10 describes this system.

As before, state 2 represents the situation where there are two good processors and one faulty processor. The transition from state 2 to state 4 represents the detection and isolation of the faulty processor and its replacement with a spare processor. While the system is in state 2, there are 2 active working processors that can fail; thus, the rate of the transition to death state 3 is 2λ. After reconfiguration occurs, there are once again three active processors that 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 Some Observations About Multi-Step Fault-Error Handling Models

Fault-injection experiments have been performed at NASA Langley demonstrating the feasibility of measuring the distribution of the overall recovery process [5]. Recovery processes have been shown to be nonexponential [5, 6], and a given system may even exhibit different distributions for recoveries from different types of faults. Fault injection experiments may be used to estimate the mean and standard deviation conditional upon recovery, and this conditional mean and standard deviation are used in the SURE program to define the behavior of a recovery transition. For more information, see [7, 8] and section 5.

Before White's solution technique was developed, the solution of semi-Markov models with nonexponential recovery transitions was extremely difficult. Some modelers have been willing to make the assumption that the recovery process behaves according to an exponential distribution in order to facilitate the solution of the model. However, it should be recognized that such an assumption introduces a modeling error of unknown quantity.

Another approach utilized to avoid the necessity of solving semi-Markov models was to decompose the reliability model into a fault-occurrence model and a set of fault-error handling models [9, 10]. Coverage parameters derived from the solution of the fault-error handling models are inserted into the fault-occurrence model in order to compute the system reliability. Some of these fault-error handling models, such as the CARE III single-fault model, represent the individual steps that a system performs to accomplish the overall reconfiguration process. This multi-step process was designed to enable a more accurate representation of the reconfiguration process than could a single exponential process. However, these multi-step models use some 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. However, now that an efficient semi-Markov solution technique is available that requires only directly observable parameters, such an approach is unnecessary.

5 Reliability Analysis Programs

Before we proceed further in the art of modeling, the input language for the SURE reliability analysis program will be presented. The same input language is used for the STEM and 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 as well as graphically. This is desirable for two reasons: (1) As the models increase in complexity, it soon becomes impractical to present them graphically and (2) these programs can be used to solve the models as functions of any model parameter. This will provide insight into the nature of the systems being modeled.

5.1 Overview of SURE

Probably the easiest way to learn the SURE input language is by way of example. The input to the SURE program for the degradable quad model in figure 9 is:

```
LAMBDA = 1E-4;
MU1 = 2.7E-4;
SIGMA1 = 1.3E-3;
MU2 = 2.7E-4;
SIGMA2 = 1.3E-3;

1,2 = 4*LAMBDA;
2,3 = 3*LAMBDA;
2,4 = <MU1,SIGMA1>;
4,5 = 3*LAMBDA;
5,6 = 2*LAMBDA;
5,7 = <MU2,SIGMA2>;
7,8 = LAMBDA;
```

The first five statements equate values to identifiers. The first identifier LAMBDA represents the processor failure rate. The next two identifiers MU1 and SIGMA1 are the mean and standard deviation of the recovery time from state 2 to state 4. The identifiers MU2 and SIGMA2 are the mean and standard deviation of the recovery time from state 5 to state 7. The final seven 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 7 to state 8 with rate λ (or 1×10^{-4}/hour). If the transition is a fast recovery process then the mean and standard deviation of the recovery time must be given. For example, the statement `2,4 = <MU1,SIGMA1>` above defines a transition from state 2 to state 4 with mean recovery time MU1 and standard deviation SIGMA1. The following is an illustrative interactive session using SURE to process the above model. The model description has been stored in a file named `TRIADP1`. The user input follows the ? prompt.

```
$ sure

 SURE V7.4   NASA Langley Research Center

 1? read triadp1

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

      0.05 SECS. TO READ MODEL FILE
13? run

MODEL FILE = triadp1.mod                      SURE V7.4 24 Jan 90   10:16:21


   LAMBDA        LOWERBOUND    UPPERBOUND    COMMENTS                 RUN #1
 -----------    -----------   -----------   ----------------------------------
 1.00000e-06    1.68485e-14   1.77002e-14
 1.00000e-05    3.00387e-12   3.12024e-12
 1.00000e-04    1.61714e-09   1.66224e-09
 1.00000e-03    1.45575e-06   1.51644e-06
 1.00000e-02    1.23551e-03   1.26292e-03   <ExpMat>

3 PATH(S) TO DEATH STATES
Q(T) ACCURACY >= 14 DIGITS
0.633 SECS. CPU TIME UTILIZED
14?
15? plot xylog
16? 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 directs the SURE program to automatically perform a sensitivity analysis as a function of this parameter over the specified range. Statement 12 defines the mission time to be 10 hours. Statement 15 directs the program to plot the output on the graphics device. Figure 11 shows the graph generated by this command. The XYLOG argument causes SURE to plot the X and Y axes using logarithmic scales.

In the next subsections, more detail is presented. These subsections can probably be skipped on 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.

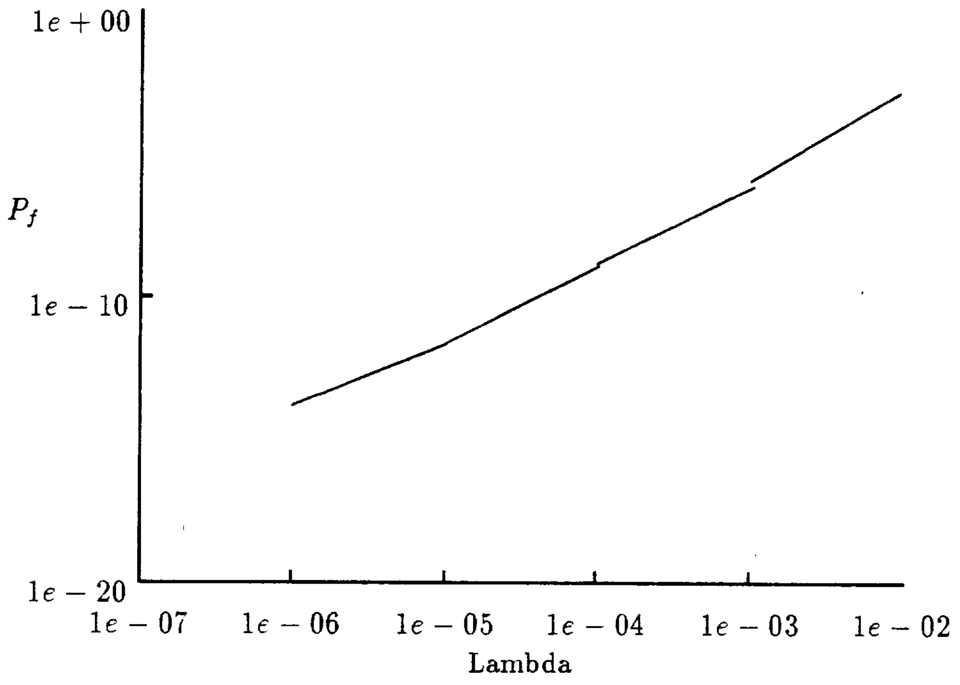


Figure 11: SURE program's plot of output

2. Lowercase words which are in italics indicate items which are to be replaced by something defined elsewhere.

3. Items enclosed in square brackets [ ] can be omitted.

4. Items enclosed in braces { } can be omitted or repeated as many times as desired.

5.2 Model-Definition Syntax

Models are defined in SURE by enumerating all of the transitions of the model. The SURE program distinguishes between fast and slow transitions. If the mean transition time, say μ is small with respect to the mission time, i.e. $\mu < T$, 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 standard deviation of the transition time. If there are multiple competing transitions from a state, the user must supply the respective transition probabilities along with the *conditional* means and standard deviations.

5.2.1 Lexical Details

The state numbers must be positive integers between 0 and the MAXSTATE implementation limit, usually 25,000. This limit can be increased simply by changing the MAXSTATE constant in the program and recompiling. The transition rates, conditional means and standard deviations, etc., 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 notation "(*" indicates the beginning of a comment and "*)" indicates the termination of a comment. 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 (_) beginning with a letter, and *expression* is an arbitrary mathematical expression as described in a subsequent section entitled "Expressions".

5.2.3 Variable Definition

In order 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. If the system is run in graphics mode (to be described later), then a plot of this function will be made. 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 over this range to be computed. 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.

One additional option is available—the BY option. By following the above syntax with BY "*increment*", the value of POINTS is automatically set such that the value is varied by adding or multiplying the specified amount. For example,

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

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

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

sets POINTS equal to 3, and the values of CX used would be 3, 4, and 5.

In general, the syntax is

var = *expression* TO[*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 as described in the next section 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:

+	addition
-	subtraction
*	multiplication
/	division
**	exponentiation

The following standard Pascal functions may be used: EXP(X), LN(X), SIN(X), COS(X), ARCSIN(X), ARCCOS(X), ARCTAN(X), 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	=	an expression defining the conditional mean transition time
sig	=	an expression defining the conditional standard deviation of transition time
frac	=	an expression defining the transition probability

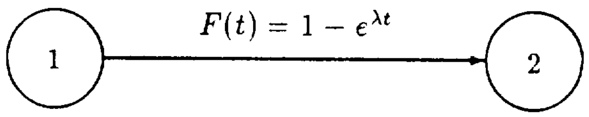
and *source* and *dest* define the source and destination states, respectively. The third parameter *frac* is optional. If omitted, the transition probability is assumed to be 1.0, i.e. only one fast transition. All of the following are valid:

```
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 is unavailable for the fast processes of a system. In this case, one must assume some properties of the underlying processes. 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 since they are fast transitions. If there is only one fast transition from a state, then these parameters are easy to determine. Suppose we have a fast exponential recovery 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 using the following syntax:

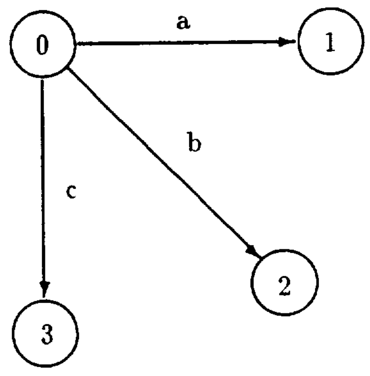


Figure 12: Model of three competing fast transitions

```
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. Suppose we have the model in figure 12 where the unconditional distributions are:

$$F_1(t) = 1 - e^{-at}$$
$$F_2(t) = 1 - e^{-bt}$$
$$F_3(t) = 1 - e^{-ct}$$

The SURE input describing the above model section 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), g/(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 above model:

```
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 non-exponential fast recoveries in order to avoid inconsistencies. For more details see [8].

5.3 SURE Commands

In this section a brief summary of some of the SURE commands is given. For more details about the SURE program, see [8].

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 extent is given, the default extent .mod is assumed. A user can build a model description file using a text editor and use this 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 as follows:

- LIST = 0; No output is sent to the terminal, but the results can still be displayed using the PLOT command.
- LIST = 1; Only the upper and lower bounds on the probability of total system failure are listed. This is the default.
- LIST = 2; The probability bounds for each death state in the model are reported along with the totals.
- LIST = 3; Every path in the model is listed and its probability of traversal. The probability bounds for each death state in the model is 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, if 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 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 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 the pruned states' occupancy probabilities is reported in following format.

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

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

5.4 Overview of the STEM and PAWS Programs

The STEM (Scaled Taylor Exponential Matrix) and PAWS (Padé Approximation With Scaling) programs were developed at the NASA 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*] >;

where

source	=	is the source state
dest	=	is the destination state
mu	=	an expression defining the conditional mean transition time
sig	=	an expression defining the conditional standard deviation of transition time
frac	=	an expression defining the transition probability

is interpreted as

source , *dest* = *frac/mu* ;

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 [11].

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-testing processors.

6.1 Degradable 6-plex

Reconfiguration can be utilized in conjunction with levels of redundancy greater than three. The Software Implemented Fault Tolerance (SIFT) computer system is an example of such an architecture [12]. The SIFT computer initially contains 6 processors. At this level of redundancy, 2 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 = the number of processors currently in the configuration and NF = the number of faulty processors in the configuration. The semi-Markov model for the SIFT system is shown in figure 13. There are three main concepts that 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 in order for the system to be "safe".

There are a few subtle points which 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),

$$\begin{tikzcd}
(6,0) \arrow[r, "6\lambda"] & (6,1) \arrow[r, "5\lambda"] \arrow[d, "{< m,s >}"] & (6,2) \arrow[r, "4\lambda"] \arrow[d, "{< m\_2,s\_2 >}"] & (6,3) & & \\
 & (5,0) \arrow[r, "5\lambda"] & (5,1) \arrow[r, "4\lambda"] \arrow[d, "{< m,s >}"] & (5,2) \arrow[r, "3\lambda"] \arrow[d, "{< m\_2,s\_2 >}"] & (5,3) & \\
 & & (4,0) \arrow[r, "4\lambda"] & (4,1) \arrow[r, "3\lambda"] \arrow[d, "{< m,s >}"] & (4,2) & \\
 & & & (3,0) \arrow[r, "3\lambda"] & (3,1) \arrow[r, "2\lambda"] \arrow[d, "{< m,s >}"] & (3,2) \\
 & & & & (1,0) \arrow[r, "\lambda"] & (1,1)
\end{tikzcd}$$

Figure 13: Semi-Markov Model of SIFT Computer System

(4,1) and (3,1). It is possible that a system in a degraded configuration may recover slower (because less processing power is available) or faster (because there are fewer processors to examine in order to find the faulty one). To determine this would require extensive fault injection in numerous configurations—a very expensive process. If this were done, however, one could easily modify the above model to contain this information. Second, the mean and standard deviation of the recovery time from states with two active faults is probably different from the states with only 1 active fault. Note that in the model 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 the detection/isolation and reconfiguration of the two faults behaves like two independent exponential processes, this assumption is reasonable. The following SURE run reveals the sensitivity of the failure probability to the mean reconfiguration time.

```
% 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
```

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

6.2 Single-Point Failures

All of the previous models assumed that there were no single-point failures in the system, i.e. one fault arrival causes system failure. If a system is not designed properly and is vulnerable to single-point failures, the reliability can be seriously degraded. To see the effects of a single-point failure, consider the following model in figure 14 of a TMR 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.

```
$ sure
```

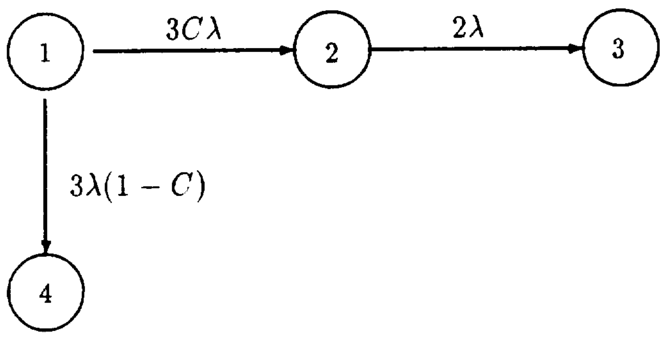


Figure 14: Model of a TMR System with a single point failure

```
 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 15.
From this run it can be seen that C must be greater than .97 in order for the TMR system

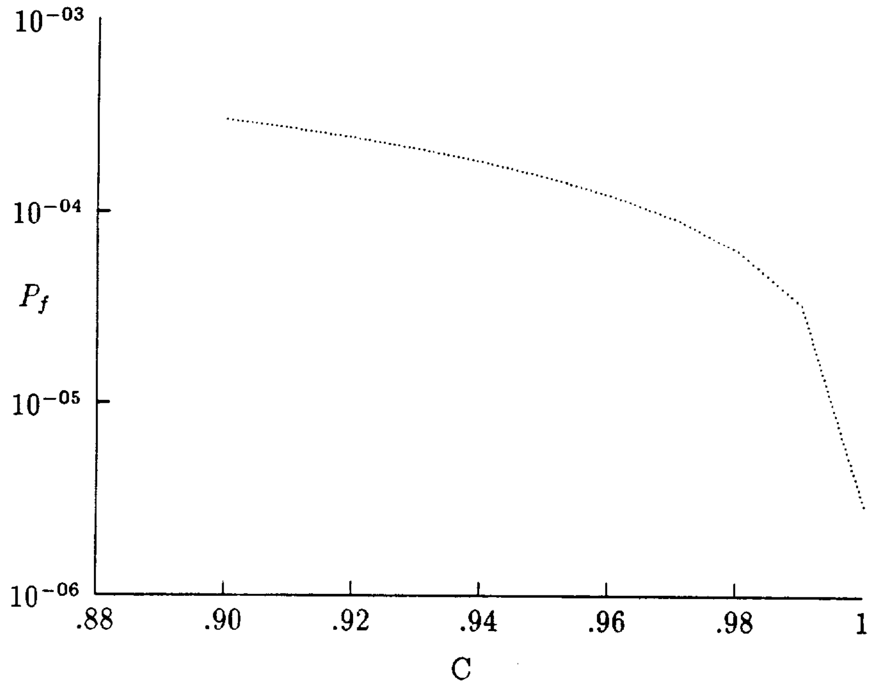


Figure 15: Failure Probability as a Function of C

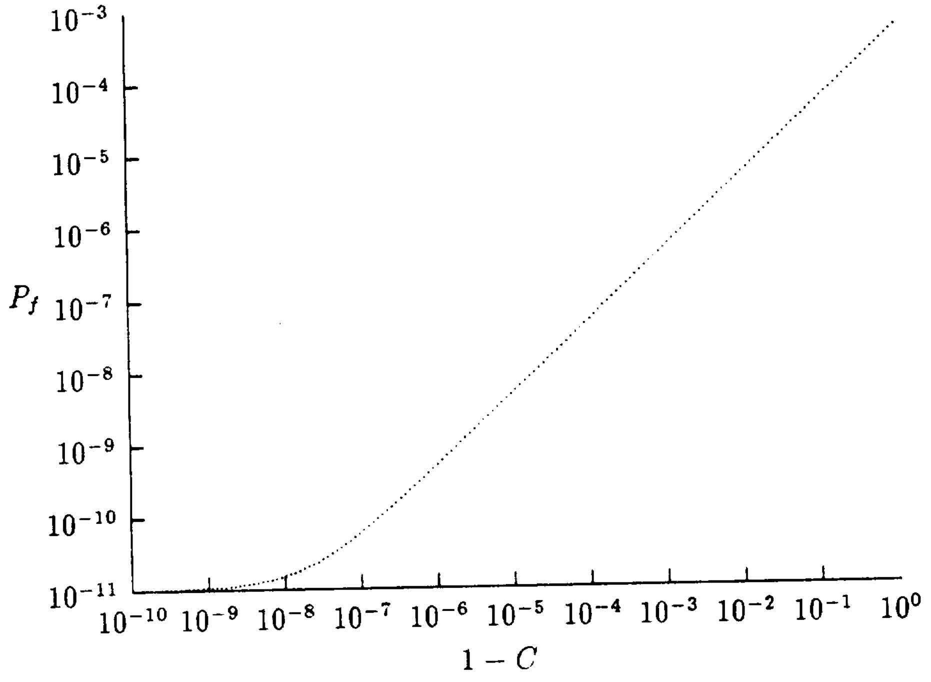


Figure 16: Failure Prob. of 5MR with $\lambda = 10^{-5}$ As a Function Of C

to be more reliable than a simplex computer. Even more distressing is the realization that in order to have a probability of failure less than 10^{-9} in a 5MR system composed of processors whose failure rate is very low, i.e. $10^{-5}/hour$, C must be greater than .999998. A 5MR system that is not subject to single point failure has a probability of failure of 1×10^{-11}. See figure 16 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;
```

Experimental measurement of a parameter to such accuracy is easily shown to be impractical. Alternatively, one can design a system with no single-point failures, and thus remove this transition from the model. If this is done, it is essential that systems be designed in a manner that enables a proof of correctness of this property.

$$\begin{tikzcd}
1 \arrow[r, "2 * PFS * \lambda"] \arrow[d, "2 * (1 - PFS) * \lambda"] & 2 \arrow[r, "\lambda"] & 3 \\
4 & &
\end{tikzcd}$$

```
PFS = 0 TO 1 BY 0.01;
1,2 = 2*PFS*LAMBDA;
2,3 = LAMBDA;
1,4 = 2*(1-PFS)*LAMBDA;
```

Figure 17: Model of Fail-Stop Dual

6.3 Fail-stop Dual

When processors fail, they often either abort the current computation or produce an incorrect answer 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 and to halt its processing. A processor with this capability is often referred to as a "fail-stop" processor. It is simple to build electronic circuitry which can recognize that one fail-stop processor has halted (e.g., since no data arrives) and automatically switch to an alternate fail-stop processor. A system consisting of two fail-stop processors and this kind of "selection" circuitry is called a "dual" system. Reliability engineers often make the mistake of assuming that this process will work correctly 100% of the time. However, most of these so-called fail-stop processors cannot be guaranteed to always halt upon failure. For example, the failure can cause an erroneous answer that is not detectably unreasonable, or the failure can affect the ability of the processor to detect the failure or to halt its processing or its output.

In this example, we will investigate the impact on system reliability when a processor is not 100% fail-stop. Suppose PFS = the probability that a processor stops when it fails. Furthermore, we will assume the probability of failure of the comparator is 0. The model in figure 17 describes such a system. The plot of the SURE solution of this model is given in figure 18. 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. As can be seen in figure 18, 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 .9 in order to have a significant improvement in reliability over a simplex computer.

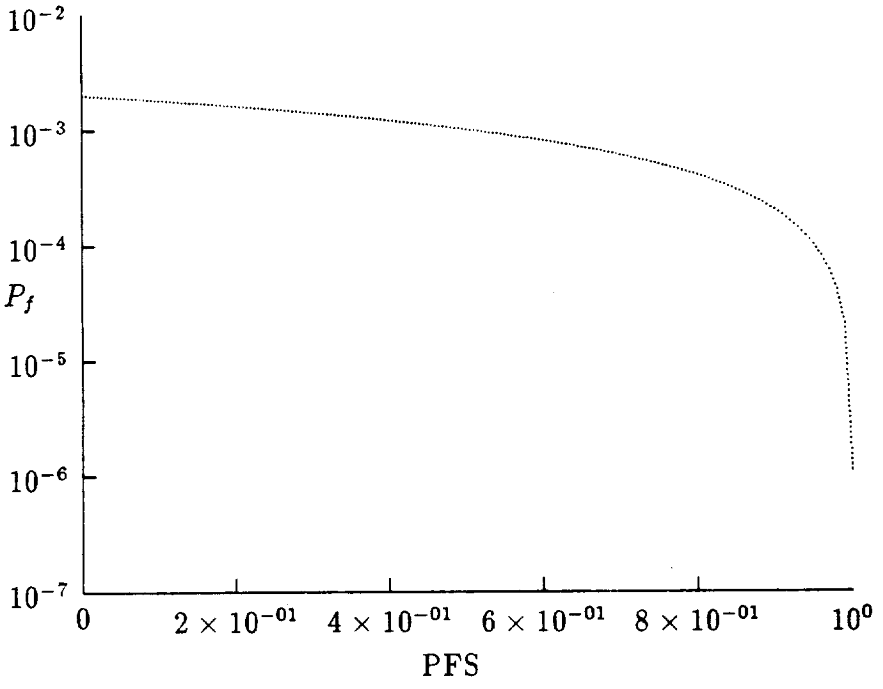
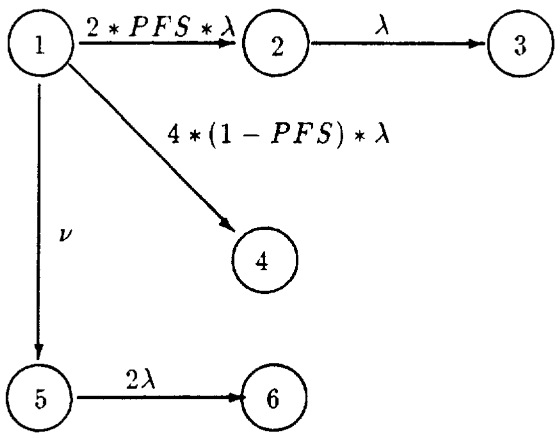


Figure 18: Plot of Fail-Stop Dual Unreliability vs. PFS



```
1,2 = 4*PFS*LAMBDA;
2,3 = 2*LAMBDA;
1,4 = 4*(1-PFS)*LAMBDA;
1,5 = NU;
5,6 = 4*LAMBDA;
```

Figure 19: Model of Dual-Dual

6.4 Dual-Dual

The previous section illustrated the sensitivity of reliability to the assumption of fail-stop. Consequently approaches have been sought to make the fail stop assumption virtually 100 percent. One such approach is the dual-dual architecture. In this system 4 computers are configured into two self-checking pairs. The self-checking pairs run in lock step mode. Upon any "miss-compare" on 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 which can make such circuitry "fail-safe", i.e. 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 as used in the previous model. The self-checking pair serves as the fail-stop processor of the previous model. In such a system it is not unreasonable to assume that the probability PFS that the self-checking pair does not stop, even though 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 0. This probability is intimately connected with fault latency and failure correlation which will be further investigated in later models. In the following model, PFS is assumed to be 0 as is commonly done. The reader, however, is cautioned to remember the previous section. The failure rate of the switch selector ν is included. The parameter λ represents the probability of failure of one of the processors in the self-checking pairs. Also, the selection of the good processor pair after the shut-down of the failed pair is assumed to be instantaneous. The transitions from state 1 to state 2 and from state 1 to state 4 in figure 19 collectively represent the failure of a processor before the selection switch has failed. If the selection switch fails first, then the transition from state 1 to state 5 occurs. The transition from state 1 to state 2 covers the case where the failed pair shuts down properly. The transition state 1 to state 4 covers the situation where the failed processor pair does not fail stop. All bets are off about the behavior of the system in this case, so state 4 is a death state. After the selection switch has failed (i.e. system in state 5), the system is assumed to be permanently switched to one self-checking pair, and the failure of either processor in that pair is assumed to cause system failure.

6.5 Degradable Quad 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 one degrades to a dual, one must address the problem of dealing with a failure in the dual. How can the voter know which processor's answer is correct? If the best that can be done is guess with probability of 0.5 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 naturally the system reliability can be improved. One method of obtaining improvement is to take advantage of the fact that many failures cause a processor to halt, as was done in the previous model. Studies by Bendix [13] have shown that typically 90% of CPU faults result in a processor halting. Although this is far from fail-stop, this aspect of system failure can be utilized in the system design to increase the reliability of a quad system. The majority voting system must be designed so as to recognize the non-arrival 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 20, PFS = the probability that a fault causes the processor to halt. The SURE input file is:

```
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;
```

Since 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 = 1000 hours, is shown in figure 21.

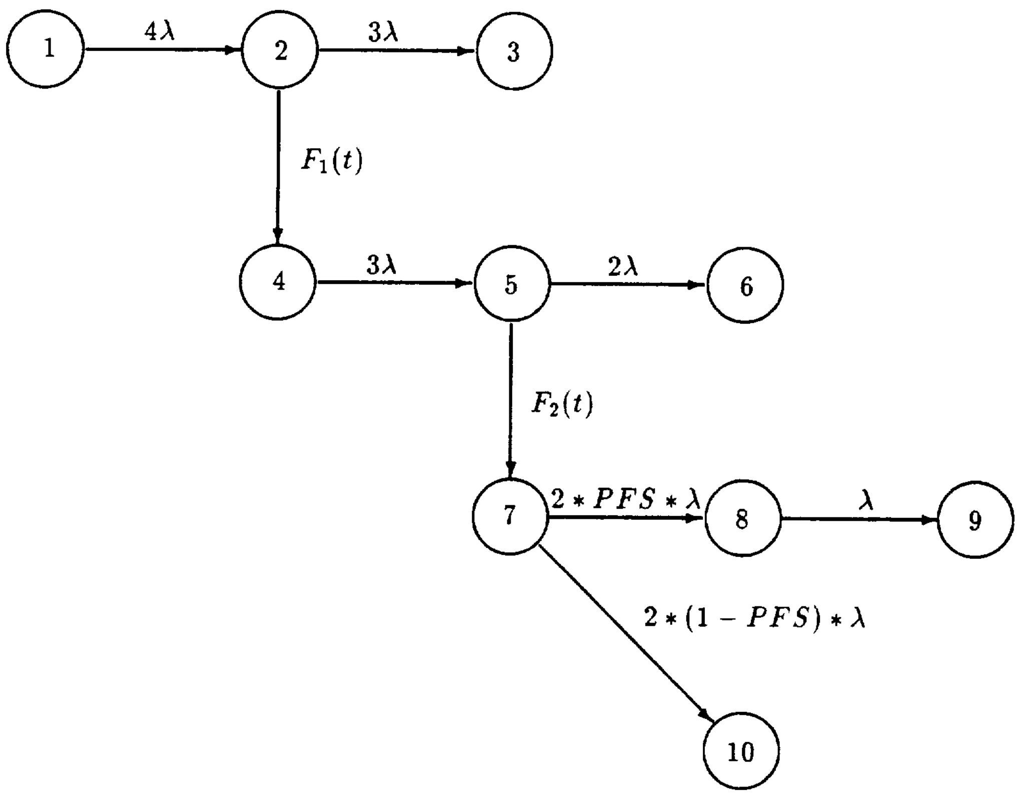


Figure 20: Degradable Quad with Partial Fail-stop

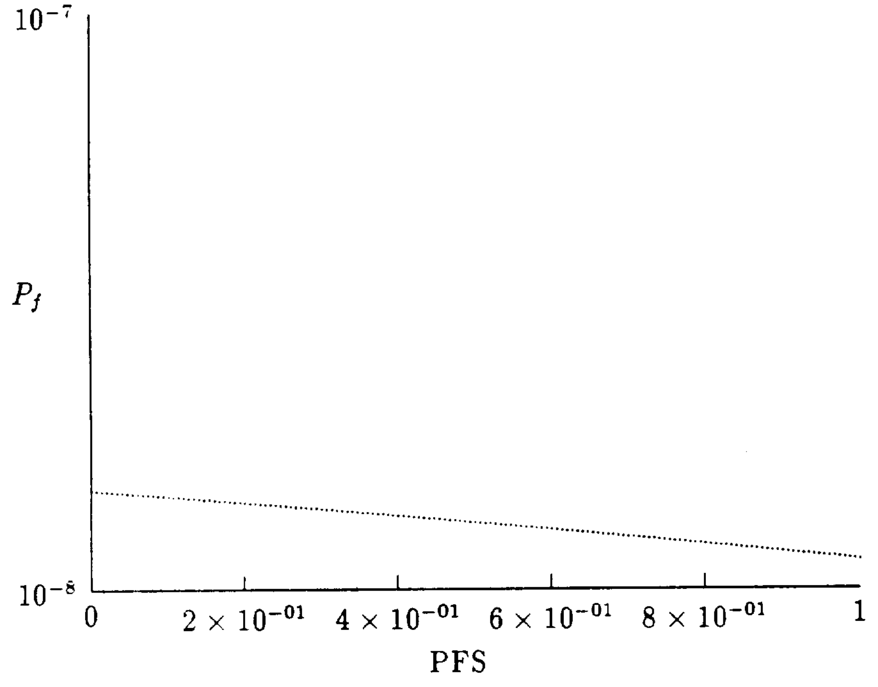


Figure 21: Failure Prob. of Degradable Quad with Partial Fail-Stop

Another approach is to use a self-test program to diagnose the faulty processor in the dual. This 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.

7 Reconfiguration By Sparing

Three categories of spares are possible—cold spares, warm spares and hot spares. Sometimes systems are designed using spares which are unpowered until brought into the active configuration. This is done because unpowered 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 called a hot spare. If the failure rate of an inactive spare is zero, then it is called a cold spare. If the failure rate is somewhere in between 0 and the active processor rate it is called a warm spare. If

$$\begin{aligned} \lambda_s &= \text{failure rate of an inactive spare} \\ \lambda_p &= \text{failure rate of an active processor} \end{aligned}$$

then

cold spare:	$\lambda_s = 0$
warm spare:	$0 < \lambda_s < \lambda_p$
hot spare:	$\lambda_s = \lambda_p$

The disadvantage of an unpowered spare (i.e. cold or warm) is that it must be initialized during reconfiguration whereas a hot spare can be maintained with memory already loaded. This can lead to a longer reconfiguration time. Thus, depending upon the strategy used, the model parameter values will be different. Some reliability programs, such as CARE III [9], explicitly assume that the spares are hot.

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 (i.e. cold) while not in the active configuration. The issues associated with failing spares will be considered in subsequent examples. In the model of figure 22 it is assumed that the reconfiguration process is 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 *)
```

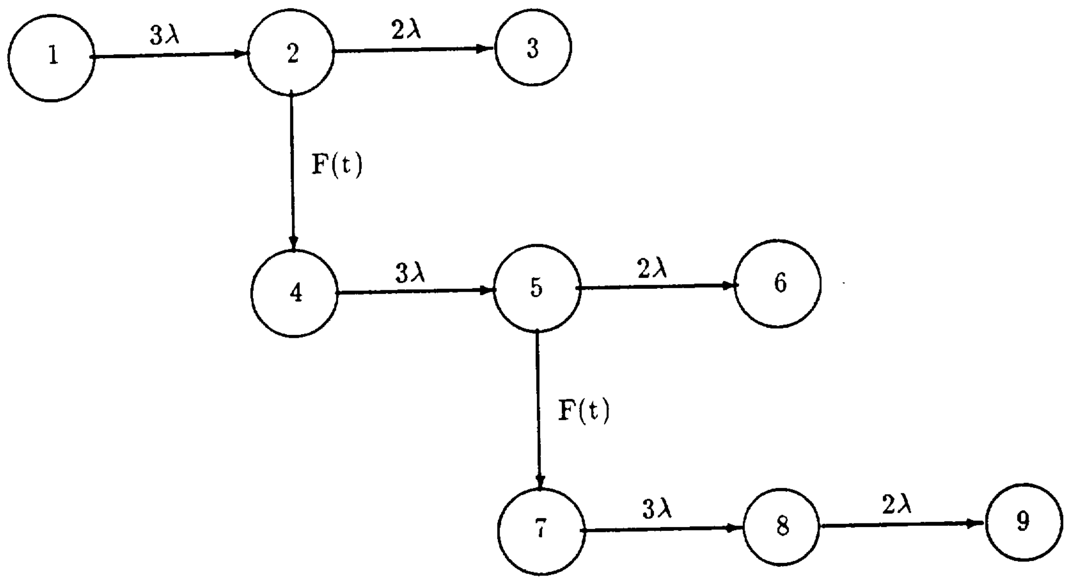


Figure 22: Model of Triplex with 2 Cold Spares

```
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;
```

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. There are two transitions leaving state 2: near-coincident failure of one of the two remaining active processors and 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 $3^*\lambda$. State 5 has the same transitions leaving it as state 2. Once the system reaches state 7, there are no remaining cold spare processors.

7.2 Triad with Two Warm Spares

If we assume the system has perfect detection of failed spare processors, the model developed above can be easily modified to include spare failures. As shown in figure 23, this simply

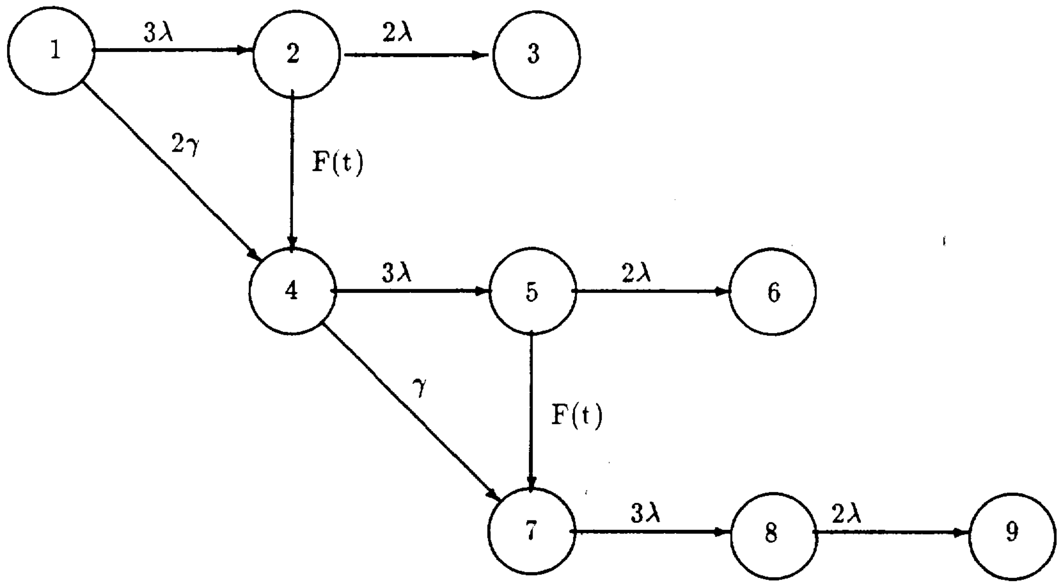


Figure 23: Model of Triplex with 2 Warm Spares

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

The same model can be used to model a system with hot spares by assigning the spare failure rate γ to the same value as the active processor failure rate λ. The probability of failure as a

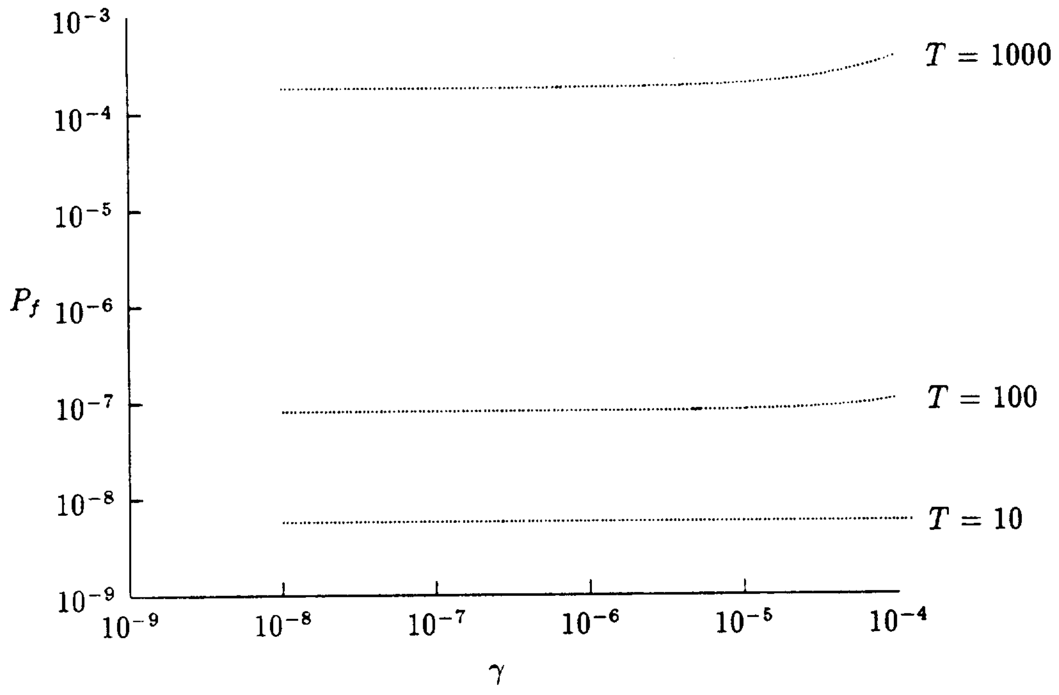


Figure 24: Failure Prob. of Triplex with 2 Warm Spares

function of the spare failure rate is plotted in figure 24 for three mission times—10,100 and 1000 hours.

In this section we made several modeling assumptions, such as perfect detection of failed spare processors and no state-dependent recovery rates. These assumptions significantly simplified the reliability models in this section. In section 9 we will model systems without making these simplifying assumptions and will investigate more complex systems which use several different kinds of reconfiguration and which consist of several subsystems. Models of complex systems are often very large. As complexity is added to a system it quickly becomes impractical to enumerate all of the states and transitions of a model by hand. In the following section 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 which automatically generates the states and 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 the 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 Pascal and runs on the VMS and Unix operating systems [14, 15]. The ASSIST program generates a file containing the generated semi-Markov model in the format needed for input to a number of Langley-developed Markov or semi-Markov reliability analysis programs, 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 more detailed information about ASSIST, the reader is referred to [14]. The process of describing a system in this abstract language also forces the reliability engineer to clearly understand the fault tolerance strategies of the system, and the abstract description is useful for communicating and validating the system model.

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 of the states are either death states or have already been processed. The expressiveness of the ASSIST language is derived from the use of a "recursive" semantics for its constructs.

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 of the states and transitions of the model by enumeration.

8.1 Abstract Language Syntax

A formal description of the language is not presented. Nevertheless, it is necessary to define a few conventions to facilitate description of the language:

1. All reserved words will be capitalized in typewriter-style print.
2. Lowercase words which are in italics, such as *const*, indicate items which are to be replaced by something defined elsewhere.
3. Items enclosed in square brackets [ ] can be omitted.
4. Items enclosed in braces { } can be omitted or repeated as many times as desired.

The language consists of 6 types of statements:

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

Each of these statements 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 commands contain legal expressions:

```
ALPHA = 1E-4;
RECV = 1.2*EXP(-3*ALPHA);
DELTA = 1.2*[(ALPHA + 2.3E-5)*RECV + 1/ALPHA];
```

All of the constant definitions are printed in the SURE model file so that they 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 he wished to see the effects of varying the value of λ exponentially, he could include the following statements in his ASSIST input file:

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

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 where each component of the vector defines an attribute of the system being modelled. In the SIFT-like architecture example of figure 13, the state space is (NW,NF). This would be defined in the abstract language as

```
SPACE = (NW: 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 0, and 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*, used in the SPACE statement are referred to as the "state space variables".

8.1.3 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 being modeled, i.e., the probability the system is in this state at time 0 is 1. In the SIFT-like architecture example in figure 13, the initial state is (6,0). This 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.

8.1.4 DEATHIF Statement

The `DEATHIF` statement specifies which states are death states, i.e., 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 (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.'

8.1.5 PRUNEIF Statement

A model of a system with a large number of components tends to have many long paths consisting of one or two failures of each type of 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 consisting of failures of "like" components. Model pruning can be used to eliminate the long paths to system failure by conservatively assuming that system failure occurs earlier on those paths.

The `PRUNEIF` statement specifies which states are prune states, i.e., conservative absorbing states in the model. The syntax for the `PRUNEIF` statement is the same as for the `DEATHIF` statement:

`PRUNEIF` *expression*;

The expression in this statement must be a Boolean expression. The use of the `PRUNEIF` statement to reduce the size of a model is discussed and demonstrated in section 12.2.

8.1.6 TRANTO Statement

This 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* of the fault-arrival transitions in the figure 1 model:

```
IF NW > 0 TRANTO (NW-1, NF+1) BY NW*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 `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 using positional or assigned values.

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

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

where the expressions listed 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, including 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 there are 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 expression cannot be within parentheses when assigned values are to be used.

The expression following the **BY** indicates the rate of the transition to be added to the model. This expression must evaluate to a real number. The user may include constants 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 using the '^' character. The use of this feature is demonstrated in section 10.5.

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

```
 IF expression THEN
    { multiple TRANTO statements or TRANTO clauses }
[ELSE
    { multiple TRANTO statements or TRANTO clauses } ]
 ENDIF;
```

where a TRANTO *clause* is of the form:

```
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 according to 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 of the death conditions to determine if the current state is a death state. Since a death state denotes system failure, no transitions can leave a death state. Each of the TRANTO rules 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 of the 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: SIFT-Like Architecture

Now we can specify the model of figure 13 in the language:

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

SPACE = (NW: 0..NP,    (* Number working processors *)
         NF: 0..NP);   (* Number faulty processors *)

START = (NP,0);

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

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

The first three lines equate the identifiers NP, LAMBDA, and DELTA to specific values. The next 2 lines define the state space using the SPACE command. For this system two attributes suffice to define the state of the system:

NW = the number of working processors in the configuration
NF = the number of faulty processors in the configuration

The SPACE statement declares that the state space is 2-dimensional, that the first dimension is named NW and has domain 0 to NP and that 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.

The informal rule is easily converted into an ASSIST statement. The phrase "Every working processor in the current configuration" becomes:

```
IF NW > 0
```

Note that if NW is greater than zero there is a working processor, so a transition should be created. The phrase "fails" is captured by

```
TRANTO NW = NW + 1
```

This says that the destination state is obtained from the current state by incrementing the NW component by 1. The phrase "at rate LAMBDA" is captured by

```
BY NW*LAMBDA
```

This declares that the rate of the generated transition is NW*LAMBDA. The identifier LAMBDA which represents the failure rate is mulipIied by NW because any of the working processors can fail. Each processor fails at rate LAMBDA. Therefore the rate that "any" processor fails is NW*LAMBDA.

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

```
IF NF > 0 TRANTO (NW,  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 DELTA and automatically calculates the mean and standard deviation. Alternatively, the user could specify this TRANTO rule as follows:

```
IF NF > 0 TRANTO (NW,  NF-1) BY <MU,SIG>;      (* system recovery *)
```

The advantage of this form is that the corectness of the solution does not depend upon an assumption that the recovery distribution is exponential.

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

```
DEATHIF NF >= NW
```

9 Reconfigurable Triad Systems

In this section systems which use both sparing and degradation to accomplish reconfiguration will be explored. Even in the first example, a triad with cold spares, the reconfiguration process changes when the supply of spares is exhausted. The later examples add more detail to 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 (i.e. they do not fail while inactive) will be explored. The number of initial spares is defined using a constant, NSI. This is done so that the initial number of spares can be changed by altering only one line of the ASSIST input. (Although the change involves a change of only one line 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 are made active. The system replaces failed processors with spares until they are all 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 5 lines define contants which 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 3-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: the number of spares available. The initial configuration is defined with the **START** statement, i.e. (3,0,NSI) which indicates that NW=3, NF=0 and NS=NSI initially. The next three statements define the rules which are used to build the model. The first of these statements defines processor failure. As long as there are working processors (i.e. 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 is short-hand 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 $(NS > 0)$, i.e. 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), i.e. the number working NW is increased by 1, the number faulty is decremented and the number of spares is decremented. This of course corresponds to the replacement of a faulty processor with a spare. The last TRANTO rule describes how the system degrades to a simplex. This occurs when no spares are available and a processor has failed, i.e. (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 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 which are death states, i.e. whenever the number of faulty processors are greater than or equal to the number of good processors. The following session was performed on this model stored in file TPNFS.AST:

```
$ ASSIST TPNFS

ASSIST VERSION 6.0
The Front End Routine FER SURE

 PROCESSING TIME =  0.52
 NUMBER OF STATES IN MODEL      = 10
 NUMBER OF TRANSITIONS IN MODEL = 13
 5 DEATH STATES AGGREGATED INTO STATES 1 - 1
 Thank you for using ASSIST, FER SURE

$ SURE

  SURE V7.1    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_DE = 1.74E-5;
  8:
  9:
 10:
 11:     2(* 3,0,3 *),    3(* 2,1,3 *) = 3*LAMBDA;
 12:     3(* 2,1,3 *),    1(* 1,2,3 *) = 2*LAMBDA;
 13:     3(* 2,1,3 *),    4(* 3,0,2 *) = <MU,SIGMA>;
 14:     4(* 3,0,2 *),    5(* 2,1,2 *) = 3*LAMBDA;
 15:     5(* 2,1,2 *),    1(* 1,2,2 *) = 2*LAMBDA;
 16:     5(* 2,1,2 *),    6(* 3,0,1 *) = <MU,SIGMA>;
 17:     6(* 3,0,1 *),    7(* 2,1,1 *) = 3*LAMBDA;
 18:     7(* 2,1,1 *),    1(* 1,2,1 *) = 2*LAMBDA;
 19:     7(* 2,1,1 *),    8(* 3,0,0 *) = <MU,SIGMA>;
```

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

Table 1: Model Sizes for Triad of Processors with Spares

```
20:      8(* 3,0,0 *),     9(* 2,1,0 *) = 3*LAMBDA;
21:      9(* 2,1,0 *),     1(* 1,2,0 *) = 2*LAMBDA;
22:      9(* 2,1,0 *),    10(* 1,0,0 *) = <MU_DEG,SIGMA_DEG>;
23:     10(* 1,0,0 *),     1(* 0,1,0 *) = 1*LAMBDA;
24:
25: (* NUMBER OF STATES IN MODEL      = 10 *)
26: (* NUMBER OF TRANSITIONS IN MODEL = 13 *)
27: (* 5 DEATH STATES AGGREGATED STATES 1 - 1 *)

      0.83 SECS. TO READ MODEL FILE
28? RUN

 MODEL FILE = TPNFS.MOD                         SURE V7.1    7-JUN-1989 13:39:09



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

5 PATH(S) PROCESSED
0.060 SECS. CPU TIME UTILIZED
29? EXIT
```

The value of NSI can be changed to model systems with different numbers of spare processors initially. As shown in table 9.1, changing this single value can have a significant effect on the size of the model generated.

9.2 Triad with Instantaneous Detection of Warm Spare Failure

This section builds on the previous model 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 procssors *)
         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;
```

Since failed spares can never be brought into the active configuration, there is no reason to keep track of these spares once they fail. Thus, no state space variable was defined to keep track of 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 Non-Detectable Spare Failure

In the previous models we assumed that the spare does not fail while inactive or that its failure was immediately detected. These are clearly non-conservative assumptions. In this example we will investigate the other extreme—not only can the spares fail (i.e. warm) but the spare's fault remains undetectable until brought into the active configuration. The model in this example utilizes a different failure rate for the spares than for the active processors. This failure rate is varied over a range (up to the active processor rate) to see the advantage of cold spares. This comparison would be more realistic if the increase in recovery time due to having to initialize the warm spare had been modeled.

```
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);
PRG = NWS/(NWS+NFS);    (* probability of switching in a 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 NS*GAMMA;

DEATHIF NF >= NW;
```

When reconfiguration occurs, the probability of switching in a good spare versus a failed spare is equal to the current proportion of good spares to failed spares in the system. The variable PRG is used to calculate this probability. When all of the spares are good, the probability of switching in a good spare is one, and the probability of switching in a bad spare is zero. Conversely, when all of the spares have failed, the probability of switching in a good spare is zero, and the probability of switching in a bad spare is one. 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: (1) a diagnostic usually cannot detect all possible faults and (2) a diagnostic requires time to execute. The first aspect is sometimes referred to as the "coverage" of the diagnostic. We will avoid the term "coverage" since it is used in so many different ways by different people and is thus confusing. Instead, we will just call it the "fraction of detectable faults" and assign it an identifer, K. It is necessary to expand the state space to keep track of whether a fault in a spare is detectable or undetectable:

```
SPACE = (NW: 0..3,              (* number of working processors *)
         NF: 0..3,              (* number of failed active procssors *)
         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 which decrement the NDFS state-space variable according to some 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 such transition is generated for "NUFS" faults.

The active processor failure **TRANTO** rule is the same as in the previous example, except that the state space is larger. The spare failure **TRANTO** rule must be altered to include whether the failure is detectable or not:

```
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 multipled by K and (1-K).

The reconfiguration rule is now more complicated than in the previous example. 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 and (3) the faulty processor is replaced with a spare containing an undetectable fault. The probability of each of these cases are PRW, PRD, and PRU, respectively, defined as follows:

```
PRW = NWS/(NWS+NDFS+NUFS);     (* prob. working spare is used *)
PRD = NDFS/(NWS+NDFS+NUFS);    (* prob. spare w/ detectable fault is used *)
PRU = NUFS/(NWS+NDFS+NUFS);    (* prob. spare w/ undetectable fault is 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 procssors *)
         NWS: 0..NSI,             (* number of working spares *)
         NDFS: 0..NSI,            (* number of detectable failed spares *)
         NUFS: 0..NSI);           (* number of undetectable failed spares *)

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*NS*GAMMA;     (* detectable fault *)
   TRANTO (NW,NF,NWS-1,NDFS,NUFS+1) BY (1-K)*NS*GAMMA; (* undetectable fault *)
ENDIF;

PRW = NWS/(NWS+NDFS+NUFS);       (* prob. a working spare is reconfigured *)
PRD= NDFS/(NWS+NDFS+NUFS);       (* prob. a spare w/ det. f. is reconfigured *)
PRU = NUFS/(NWS+NDFS+NUFS);      (* prob. a spare w/ undet f. is reconfigured *)
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;
```

In this section, systems consisting of a single reconfigurable triad were modeled. In the following section, systems consisting of multiple sets of these reconfigurable triads are discussed.

10 Multiple Triads

This section starts with a simple model of two triads sharing a pool of cold spares. In later models, 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.

10.1 Two Triads with Pooled Cold Spares

In this section we will model a system which consists of two triads which operate independently but replace faulty processors from a common pool of spares. When the pool of spares runs out, the triads continuing operating with faulty processors and do not degrade to simplex. The system fails when either triad has two faulty processors. This 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, we will use the ASSIST **INPUT** statement to define a constant 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.

Since the triads do not degrade to a simplex configuration, there is no need to keep track of the current number of processors in a triad. Thus, the state of each triad can be represented by a single variable—NW, number working. Similarly, the spares do not fail while they are inactive, so their state can be represented by a single variable—NS, 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 is accomplished 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. Note that this is conditioned upon NS > 0—if there are no spares recovery cannot take place. The result of reconfiguration is replacement of the faulty processor with a working processor (i.e. NWx $=$ NWx $+ 1$ for triad x) and depletion of 1 spare from the pool (i.e. NS $=$ NS $- 1$). The system fails whenever either triad experiences two or more coincident faults (i.e. (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. Since this model was developed assuming that the completion times for both recovery processes are exponentially distributed, the SURE keyword **FAST** was used, and 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 is discussed in section 10.8. How to model systems in which the competing recoveries are not independent are also discussed in that section.

10.2 Two Triads with Pooled Cold Spares Reducable to One Triad

The model given above can be modified easily to describe a system that can survive with only 1 triad. This strategy was used in the FTMP system [16]. 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 there is only one triad left, the system maintains the faulty triad until it can no longer out-vote the faulty processor, i.e. until the second fault arrival. As before, this model assumes that the spares are cold—they do not fail while inactive. The state space must be modified to include the notion of whether a triad is active or not. This 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 is redundant—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);` would be wrong. This 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. This follows because the last triad is never collapsed into spares. Thus, this condition can never be satisfied.

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

```
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 above 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, i.e. 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);
```

10.3 Two Degradable Triads with Pooled Cold Spares

The system modeled in this section consists of two triads which 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 more spares are available, the system degrades the faulty triad into a simplex. The extra non-faulty processor is added to the spares pool.

In this system each of the 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 which satisfies the condition NWx = 1 for triad x is a failed state (i.e. 1 good out of three) or an operational state (i.e. 1 good out of 1). 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 is only done when no spares are available, i.e. NS = 0. Also, the extra non-faulty processor is returned to the spares pool, i.e. NS = NS + 1. The DEATHIF conditions are:

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

for each triad x. This 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 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;            (* 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 of the 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.

10.4 Two Degradable Triads with Pooled Warm Spares

The models presented above can be generalized by allowing the spares to fail while inactive. This is easily done if we make a simplifying assumption that the system can instantly detect the failure of a warm spare. This can be accomplished by adding the following statements to the model descriptions:

```
LAMBDA_S = 1E-5;              (* Failure rate of inactive warm spare *)

IF NS > 0 TRANTO NS = NS - 1 BY NS*LAMBDA_S;
```

If the failure of a warm spare is not detectable while it is inactive, the state space must be enlarged by adding a new variable NFS to count the number of failed warm spares. The above rules are modified to be:

```
LAMBDA_S = 1E-5;              (* Failure rate of inactive warm spare *)

IF NS > NFS TRANTO NS = NS - 1 BY (NS-NFS)*LAMBDA;
```

Also the reconfiguration process must be generalized to include two distinct results: (1) a faulty warm spare is brought into the active configuration, and (2) a working warm spare is brought into the active configuration.

Thus, the model presented in section 10.3 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 active processors *)
LAMBDA2 = 1E-4;          (* Failure rate of active processors *)
LAMBDA_S = 1E-5;         (* Failure rate of active 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 (NS > 0) AND (NWS > 0) THEN         (* Replace with working spare *)
   IF (NW1 < 3) AND (NC1 = 3)
      TRANTO NW1 = NW1+1, NS = NS-1, NWS = NWS - 1
         BY FAST (NWS/NS)*DELTA1;
    IF (NW2 < 3) AND (NC2 = 3) AND (NS > 0)
       TRANTO NW2 = NW2+1, NS = NS-1, NWS = NWS - 1
          BY FAST (NWS/NS)*DELTA2;
ENDIF;

IF (NS > 0) AND (NS > NWS) THEN       (* Replace with failed spare *)
  IF (NW1 < 3) AND (NC1 = 3)
     TRANTO NS = NS-1 BY FAST [(NS-NWS)/NS]*DELTA1;
  IF (NW2 < 3) AND (NC2 = 3)
     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 BY FAST OMEGA1;
IF (NW2 < 3) AND (NS = 0) AND (NC2=3)          (* Degrade to simplex *)
     TRANTO NC2 = 1, NW2 = 1, NS = NS+1 BY FAST OMEGA2;

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

10.5 Multiple Non-degradable 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 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 in order to generate a specific model.

For simplicity, we will first investigate a system which is incapable of collapsing a triad into either a simplex or into spares. Thus, this system fails when any triad fails. We will also simplify this first model by assuming we have cold spares that 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 *)

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, and (3) the conditions for that triad that result in system failure.

To accommodate systems with differing reconfiguration rates for different triads, the concatenation feature was used in the rate expression of the reconfiguration TRANTO statement. The expression BY FAST DELTA^J within the FOR loop results in a reconfiguration rate of "FAST DELTA1," for triad 1, "FAST DELTA2," for triad 2, etc. Unfortunately, since the number of triads is unknown until run time (i.e. N_TRIADS is specified using the INPUT statement), there is no way 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.

10.6 Multiple Degradable Triads with Pooled Cold Spares

In this section, the simple model given above will be modified to allow degradation of triads. When no more spares are available, each faulty triad is broken up and the non-faulty 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 mutiple triads, the system can still maintain its vital functions with only 1 triad remaining.

Since we are going to be breaking up triads, we need a way of deciding if a triad is active. This will be done by adding an array state-space variable NP to keep track of the number of *active* processors in a triad. This will have the value of three for each triad initially and will be set to zero for each triad when it is broken up. The array state space variable NFP keeps a count of the number of failed processors active in each triad. The state space variable NT is used to keep track of 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:

```
(*  MULTIPLE TRIADS WITH POOL OF WARM 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 NFP[J]*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 of the 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 there is an active failed processor in a triad, the second TRANTO statement replaces that failed processor with one from the pool of spares. If there are no spares available, then the faulty triad is broken up and its working processors are put into the spares pool.

(This assumes that the system can determine which processor has failed with 100last triad in the system (i.e., NT <= 1) then the triad is not broken up. The last triad is allowed to continue operation with one faulty processor until another of its processors fails, defeating 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) -> (0,3,3,1,0,0,1,2) -> (0,3,3,0,0,0,0,2) ->
(0,3,3,0,1,0,0,2) -> (0,0,3,0,0,0,2,2) -> ...
```

10.7 Multiple Degradable Triads with Pooled Warm Spares

The model given above can be easily modified to include spare failures:

```
(*  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))*NFP[J]*DELTA1;
            (* Replace failed processor with working spare *)

         IF NFS > 0 TRANTO NS = NS-1, NFS = NFS-1
            BY FAST (NFS/NS)*NFP[J]*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 keep track of 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 defines replacement of the failed processor with a working spare, and it is conditioned on the existence of non-failed spares. The second defines replacement of a failed processor with a failed spare, conditioned on the existence of failed spares.

10.8 Multiple Competing Recoveries

In the preceding examples (i.e. multiple triads with pooled spares), we encountered the first example of a model containing states with multiple recovery processes leaving a single state. This occurs when multiple faults accumulate in different parts of the system which together do not cause system failure. The diagram in figure 25 illustrates this concept: Here we have two triads which are accumulating faults—the first at rate $\lambda 1$ and the second at rate $\lambda 2$. In state (2,2,7) they both have a faulty processor active at the same time. This is not system failure since the two failures are in separately voted triads. There are two possible recoveries from this state—triad 1 recovers first then triad 2, or vice versa. Which case occurs depends on how long each recovery takes.

In some systems, the recovery process may take longer when there is another recovery also ongoing in the system. But even when the two recovery processes have no effect on each other in the system, the presence of competing recoveries still impacts the transition specification, since the SURE program requires *conditional* means and *conditional* standard deviations for the competing recovery processes. Consider the simple case where the two recovery distributions are identical. 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 recoveries times. The SURE program also requires the specification of a third parameter—the transition probability. This is the probability

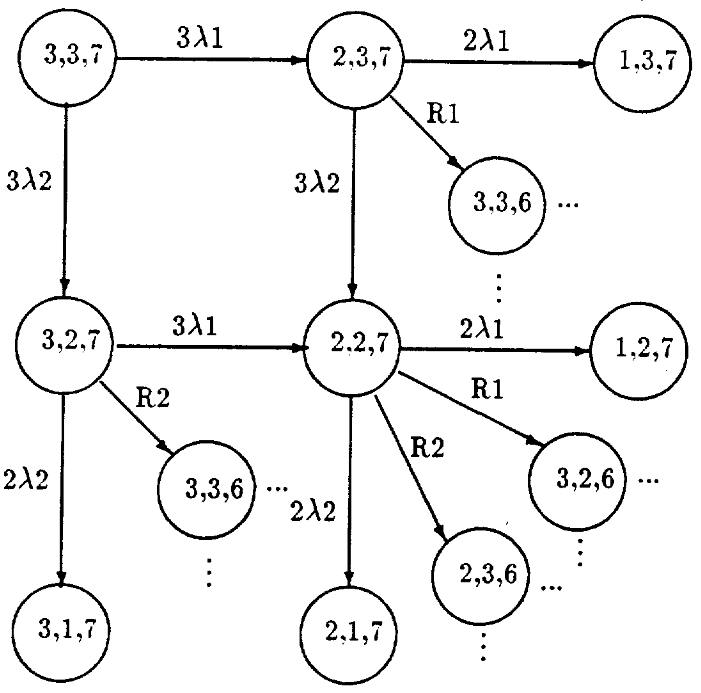


Figure 25: Model With Multiple Competing Recoveries

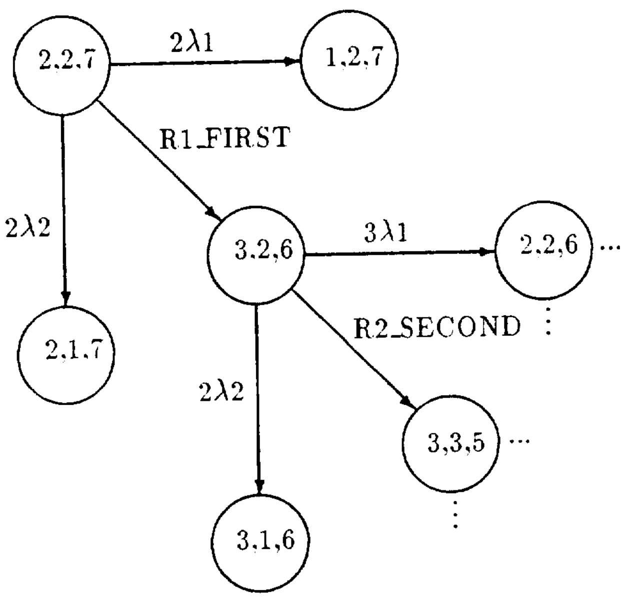


Figure 26: Triad 1 is Always Repaired First

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 of the fast transitions leaving a state must equal one.

In the examples above, 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, for this special case the SURE program will automatically calculate the conditional rates from the unconditional fast exponential rates given.

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, and (3) two independent repair processes take place.

The model for case (1), triad 1 always repaired first, is shown in figure 26. 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 rates, R1 and R2. This depends on the system, and may be determined by analysis or experimentation.

The case (2) model of both triads being repaired at the same time is shown in figure 27.

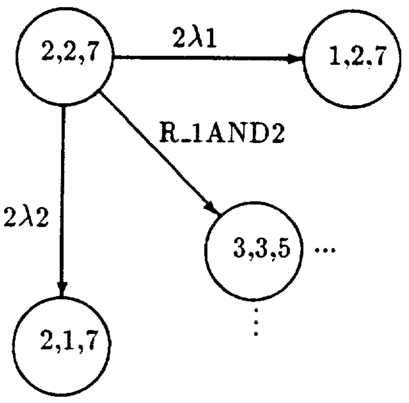


Figure 27: Both Triads are Repaired at the Same Time

The mean and standard deviation of the multiple recovery transition must be determined experimentally. This can be accomplished by injecting two simultaneous faults and measuring the time to recovery completion.

The third case, two independent repair processes, is shown in figure 28. Even if the two recoveries are truly independent and not competing for resources, the transition rates will still be different from the noncompeting rates because they are conditioned on "winning" the competition. There is no way to analytically determine the conditional means and standard deviations from the unconditional recovery distributions in general; therefore, these four distributions must be measured experimentally.

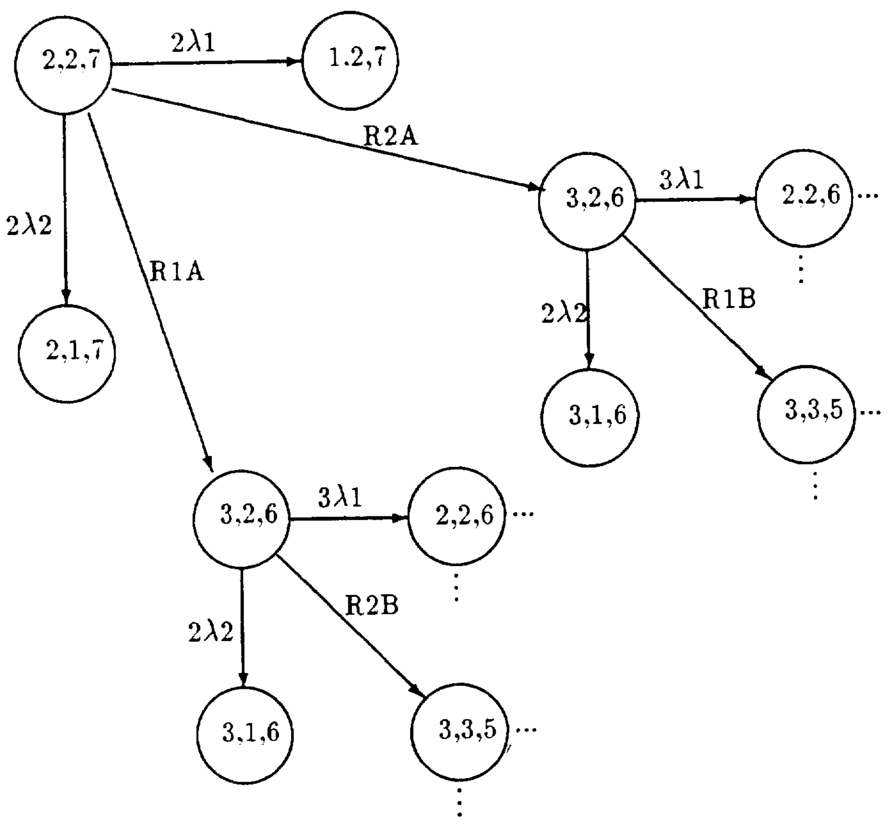


Figure 28: Two Independent Repair Processes

11 Transient and Intermittent Faults

Computer systems are susceptable to transient and intermittent faults as well as solid permanent faults. Transient faults are faults which cause erroneous behavior for a short period of time and then disappear. Intermittent faults are permanent faults which periodically exhibit erroneous behavior then correct behavior. The problem with transient faults is that they can 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. Since they tend to occur more frequently than permanent faults, this can have a significant impact on the probability of system failure. The problem with intermittent faults is that they can deceive the operating system into diagnosing that the fault is transient rather than permanent. Therefore, a processor experiencing an intermittent fault may be left in operation much longer than a solid permanent fault or may be repeatedly removed, restarted, and returned to operation. This makes the system vulnerable to near-coincident faults for a much longer time than would a solid permanent fault, and also may increasing the fault management overhead enough to degrade performance. Clearly a properly designed system must deal effectively with these types of faults. Furthermore, the assessment of such systems depends upon careful modeling of these faults.

11.1 Transient Fault Behavior

A transient fault may or may not generate errors which are detectable by the operating system's voters. The following two timing graphs illustrate the two possible effects of a transient fault:

Case 1: Reconfiguration does not occur

```
------------------------------------------ ... ----------------> t
      |           |  |  |    |    |  |         |
      s           e1 e2 e3   e4   e5 e6 ...    en

      |<-------------------- Z -------------------->|
```

Case 2: Reconfiguration occurs

```
------------------------------------------ ... ----------------> t
      |           |  |  |    |    |  |         |    |
s                 e1 e2 e3     e4       e5 e6  ...       en        r

|<------------------------- R ----------------------->|
```

where

$$\begin{aligned} s &= \text{time of fault arrival} \\ e_i &= \text{time of detection of the } i^{th} \text{ error } (1 < i < n) \\ r &= \text{time operating system reconfigures} \\ Z &= e_n - s \\ R &= r - s \end{aligned}$$

These two cases represent the outcome of two competing processes—the disappearance of the transient fault 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).

$$F_R(r) = Prob[R < r] \tag{1}$$

Let $F_Z(z)$ represent the distribution of the time for the disappearance of the transient fault given that reconfiguration does not occur.

$$F_Z(z) = Prob[Z < z] \tag{2}$$

The first distribution F_R can be directly observed. The second distribution F_Z is more troublesome to determine. The problem is that a fault produces errors which may persist long after the fault has actually disappeared. Sometimes the errors disappear quickly, sometimes they don't. The problem is that 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 some type of "state-restoration" process periodically, a transient fault is as damaging as a permanent fault. For example, an alpha particle may flip a bit in memory. If this memory is not re-written, the error will persist indefinitely. Therefore it is essential that the fault-tolerant operating system periodically rewrite volatile memory with "voted" versions of the state.

11.2 Modeling Transient Faults

In this section we will investigate the problem of modeling a triplex system that is subject to transient faults. First, a failure rate γ must be determined for the transient class of

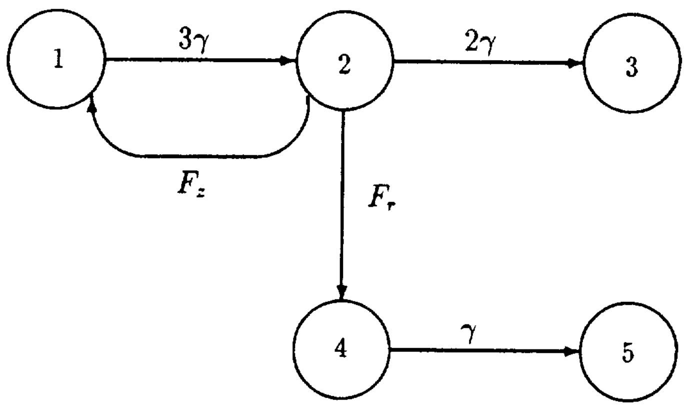


Figure 29: Degradable Triad Subject to Transient Faults

faults (i.e. the rate of transient fault arrivals) . Often the transient fault rate γ is assumed to be 10 times [1] the permanent fault rate λ. We will assume that this system has been designed such that it can recover from transient faults. (Otherwise, transient faults are as deadly as permanent faults and should be modeled as such.) This recovery is accomplished by periodically voting all of the *volatile* internal state of the processor. Each (non-faulty) processor rewrites each data value of its internal state with a voted value. Let ISVP = the period with which the operating system replaces the entire volatile state with voted values. We also will assume that the active duration of a transient fault is small in comparison to ISVP. Assuming that the time from the fault arrival to the operating system update is uniformally distributed, the mean is ISVP/2 and the standard deviation is $\text{ISVP}/2\sqrt{3}$. 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 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 3-way voter can no longer mask the faults. Such a state is a system failure state. In figure 29, a model of a degradable triad system subject to only transient faults is given. 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;
```

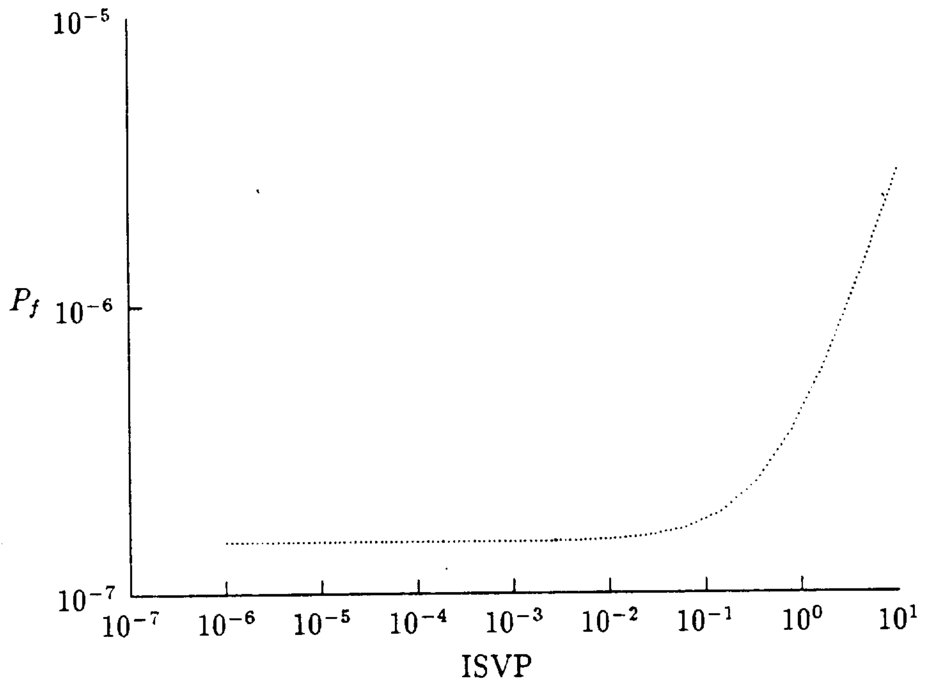


Figure 30: Failure Probability as a Function of IVSP

```
2,3 = 2*GAMMA;
2,4 = <MU1,SIGMA1,PROB_RECONF>;
2,1 = <ISVP/2,ISVP/(2*SQRT(3)),1-PROB_RECONF>;
4,5 = GAMMA;
```

In this model there are two recovery transitions from state 2. Therefore, it is necessary that SURE's three-parameter form of recovery be used. The first two parameters are the conditional mean and standard deviation. The third parameter is the probability that the recovery transition succeeds over all of the other competing recovery transitions. An experimental procedure for measuring these parameters is decribed in [17]. The probability of failure of the system as a function of the voting period, IVSP, is shown in figure 30

11.3 Model of Quad Subject to Transient and Permanent Faults

Since transient faults tend to occur at a faster rate than permanent faults, many systems are designed to tolerate transients that disappear after a short amount of time. Because fewer processors are needlessly reconfigured out, this 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. Since this algorithm is not foolproof, it is necessary to include a transition in the model representing the operating system incorrectly reconfiguring in the presence of a transient fault.

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 if 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 non-volatile read-only memory (ROM). This is also true of the program code.

The ASSIST input file for a SIFT-like system starting 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 *)
```

11.4 Degradable NMR with Transients

In this section, some problems associated with modelling degradable NMR systems subject to permanent and transient faults is explored. The major problem is that many different situations arise where there are competing recoveries. Each of these situations involves different parameters which must be experimentally measured. To illustrate the problem, we will first consider a degradable 6-plex. If we modify the model of the previous section 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 we examine the generated model, we find that at state 12, we have five transitions:

```
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. The reason for this is that there are two active faults 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 and (3) the transient fault disappears. The ASSIST model was originally constructed for a quad system where any state with two active faults would be a death state. However, with higher levels of redundancy comes more complexity. There are several ways around this problem. Unfortunately, the more satisfactory models are more complex. We will begin will the simplest.

The easiest way around the problem, is to make all such states death states. This is the approach used by programs based on the "critical-pair" approach such as CARE and HARP [9, 10]. This can be done with ASSIST by changing the **DEATHIF** statement to

```
DEATHIF NFT + NFP >= 2;
```

Although this results in a conservative answer, it is not a satisfactory solution since the model simply ignores all of the additional redundancy. Overly conservative results can be obtained using this technique.

A second way around the problem is to model all of the recovery transitions with exponential distributions. The SURE program automatically determines all of the conditional parameters when this is done. The model would be:

```
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 of the recovery distributions are exponentially distributed.

The most accurate way to model such systems is to use general recovery distributions. This necessitates analysis of each of the situations where multiple competing recoveries occur. For a 5-plex or a 6-plex, there are operational states with two active faults. The following cases exist: (1) two permanents, (2) two transients and (3) a transient and a permanent. The conditional moments for each of these cases must be measured experimentally. These parameters are:

Case 1: two permanents

- MU_2 = conditional mean recovery time of the first of two competing recoveries
- STD_2 = conditional standard deviation of the recovery time of the first of two competing recoveries

Case 2: two transients

- MU_DISAPPEAR_2 = conditional mean time of disappearance of one of the two transients
- STD_DISAPPEAR_2 = conditional standard deviation of the time of disappearance of one of the two transients
- P_DISAPPEAR_2 = probability one of the transients disappears before the system reconfigures one of the transients.

- MU_REC_2 = conditional mean time to reconfigure one of the transients before either disappears
- STD_REC_2 = conditional standard deviation of time to reconfigure one of the transients before either disappears.

Case 3: a transient and a permanent

- P_DIS_BEF2 = probability the transient disappears before the system reconfigures either fault.
- P_REC_TRAN = probability the system reconfigures the transient before it disappears or the permanent is reconfigured.
- P_REC_PERM = probability the system reconfigures the permanent before the transient disappears or is reconfigured.
- MU_DIS_3 = conditional mean time of disappearance of the transient given that it wins the 3-way race.
- STD_DIS_3 = conditional standard deviation of the time of disappearance of the transient given that it wins the 3-way race.
- MU_REC_3 = conditional mean time to reconfigure the transient given that it wins the 3-way race.
- STD_REC_3 = conditional standard deviation of time to reconfigure the transient given that it wins the 3-way race.
- MU_3 = conditional mean time to reconfigure the permanent given that it wins the 3-way race.
- STD_3 = conditional standard deviation of time to reconfigure the permanent 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 one would want to perform a rough sensitivity analysis to determine how sensitive a system is to transient faults before developing such a complex model and measuring so many parameters.

11.5 FTP

The strategy used in the Charles Stark Draper Laboratory's Fault-Tolerant Processor (FTP) for dealing with transient faults is different than that used in earlier fault-tolerant systems such as SIFT [19]. In the earlier 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 31. In this model each state is described by a triple:

(NW,NFA,NFT)

where

$$\begin{tikzcd}
(4,0,0) \arrow[r, "4\lambda"] & (3,1,0) \arrow[r, "3\lambda"] \arrow[d, "F_s"] & (3,2,0) & \\
& (3,0,1) \arrow[ul, "F_{pass}"'] \arrow[r, dash, "3\lambda"] \arrow[d, "F_{fail}"] & (3,1,1) \arrow[ul, "F_{pass}"'] \arrow[r, "2\lambda"] \arrow[d, "F_{fail}"] & (3,2,1) \\
& (3,0,0) \arrow[r, "3\lambda"] & (3,1,0) \arrow[ul, "F_{pass}"'] \arrow[r, dotted] \arrow[d, dotted] & {} \\
& & {} &
\end{tikzcd}$$

Figure 31: Partial Model of FTP

NW	=	number of working processors
NFA	=	number of faulty processors (both transient and permanent)
NFT	=	number of processors undergoing self test

The transition from (4,0,0) to (3,1,0) represents the failure of any processor in the configuration. The transition from (3,1,0) to (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, as represented by the transition from (3,0,1) back to (4,0,0). If the processor fails the self-test, the processor is permanently removed from the configuration. This occurs in the model in the transition (3,0,1) to (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 is true because the outputs from the removed processor are not considered in the voting, thus the majority of the outputs being voted are nonfaulty. Thus, state (3,1,1) 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;
```

$$\begin{tikzcd}
(3,0,0) \arrow[r, "3\lambda"] & (2,0,1) \arrow[r, "2\lambda"] \arrow[d, bend left, "a"] & (1,0,2) \arrow[d, bend left, "a"] \\
& (2,1,0) \arrow[u, bend left, "b"] \arrow[r, "2\lambda"] \arrow[dr] & (1,1,1) \arrow[u, bend left, "b"] \\
& & (1,0,0)
\end{tikzcd}$$

Figure 32: Detailed Intermittent Fault Submodel

In this model it is assumed that the FTP does not allow a second processor to undergo self test, while another processor is undergoing self-test. Note that the "IF-expression" which governs the generation of transitions which 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. There is virtually no data available on what are reasonable values for this parameter because this data cannot be measured on operational equipment or through fault injection experiments.

This parameter was not used explicitly in the above model of the FTP system. The disappearance rate of short transients 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 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 disappearance rate affects what the ratio of transient to permanent faults will be. And that ratio, which is unknown, can play an important part in assessing whether the FTP strategy of reinstating processors is a good strategy.

11.6 Modeling Intermittent Faults

A remnant of the multi-step fault error-handling model methodology (see section 4.5) is the notion that separate states must be used to represent the active and inactive states of an intermittent fault. Models are constructed that resemble the partial model in figure 32.

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	=	number of working processors
NFA	=	number of processors with active faults
NFB	=	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 which would enable the system to detect the fault. The question that the modeler must address, is whether "benign" faults cause near-coincident failure. One conservative approach is to assume that they do. In this case, intermittent faults behave identically to permanent faults except that they are reconfigured at a different rate than permanents. If faults in the benign state are assumed to not cause near-coincident failure, then there are many "additional" states in the model which contain benign faults. For example, states (1,0,2), (1,0,3), (2,0,2) contain more 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);
```

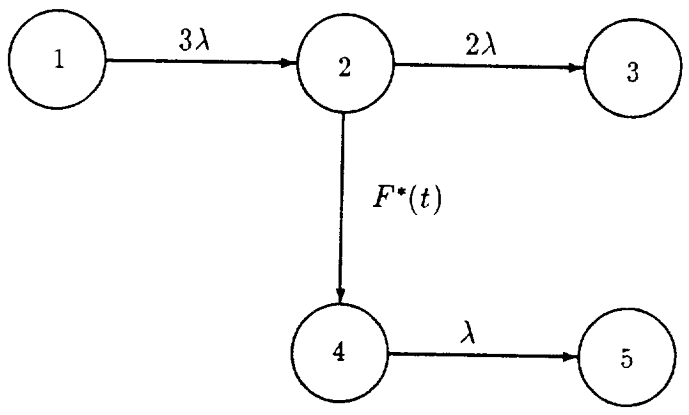

$$\begin{tikzcd}
1 \arrow[r, "3\lambda"] & 2 \arrow[r, "2\lambda"] \arrow[d, "F^*(t)"] & 3 \\
 & 4 \arrow[r, "\lambda"] & 5
\end{tikzcd}$$

Figure 33: Model of Triplex to Simplex System Subject to Intermittent

The recovery rule generates two competing recoveries. This is necessary because the operating system makes an arbitrary choice among the processors which 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 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 we could accurately measure these parameters, a semi-Markov model may not have enough generality to accurately represent the behavior of the active-benign oscillations. We believe it is preferable to inject intermittent faults and observe the impact on the system. The system recovery time will probably be longer for intermittents than for transients. The resulting model is shown in figure 33. Even though this model is considerably simpler, it can be much more accurate than the detailed model given above in some cases because it relies only on directly observable parameters. Note that this method uses the conservative approach of assuming that benign faults can cause near-coincident failure.

The SURE program has difficulty solving models with "fast loops", i.e, loops containing no slow transitions. The SURE program can solve the model generated by the ASSIST input above. The output is

```
air51% 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 secs. to solve the model. In fact as $B \rightarrow \infty$, the execution time $\rightarrow \infty$ 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.

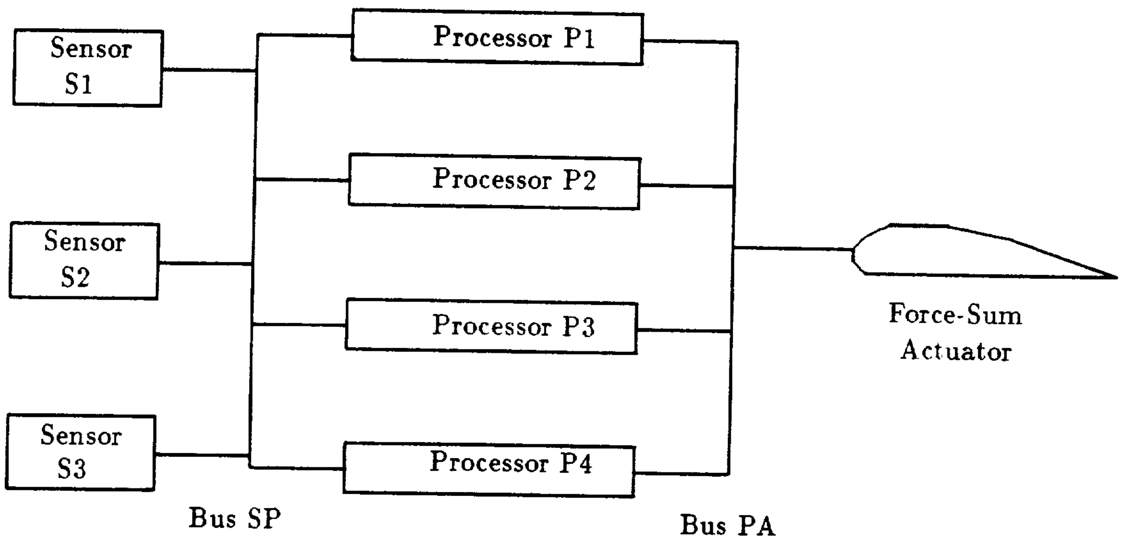


Figure 34: System with 5 Subsystems

12 Modeling Control System Architectures

All of the models in the previous sections contained only processors in various configurations. In this section we will discuss how to include the fault behavior of all of the components in a typical flight control system architecture in the reliability model. Although some of the previous models were somewhat complex, they typically dealt with only a few components. As more components are added to the models, the possible combinations and sequences of component failures, and thus the size of the reliability model, increases exponentially. Therefore, model pruning techniques needed to reduce the size of these models are introduced in this section.

The system shown in figure 34 consists of five subsystems: (1) the triplicated sensors, (2) the triplicated sensor-to-processor bus SP_BUS, (3) the degradable quad of processors, (4) the triplicated processor-to-actuator bus PA_BUS, and (5) the forced-sum voting actuator. As long as there are no failure dependencies, the separate subsystems can be represented by separate reliability models. Each of these are solved in isolation. Finally, the results are added together probabilistically, i.e. the probability of the union. The SURE command ORPROB performs the probabilistic add 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 follows:

```
$ 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 = 1E-8;
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. The following section shows a more complex sensor subsystem.

12.1 Monitored Sensors

In this example, a set of monitored sensors is modeled. Initially, the system consists of five sensors, and each sensor has a monitor to detect failure of that sensor. Sensors fail at rate λ_S, and monitors fail at rate λ_M. If a sensor fails and the monitor is working, the monitor will detect with 90% 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, contributing its faulty answers to the voting. There is no other means of reconfiguration.

Since values from all of the sensors in the active configuration are voted, system failure occurs when one-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 2*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 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.

In the above examples, subsystems could be modeled separately because the functioning and failures in each subsystem were not dependent on the current state of the other subsystems. Unfortunately, this is rarely the case. In the following sections, modeling of systems with various failure dependencies between subsystems will be discussed.

12.2 Failure Dependency

The following architecture description for the ARCS flight control system was taken from [18]. The system consists of 3 sensors (s1, s2, s3), 3 hydraulics units (h1, h2, h3), 3 computers (c1, c2, c3) and 3 servos (v1, v2, v3). The system is assumed to have perfect failure detection. Therefore, as long as there is one computer working, the system still has computational capability. Also, the reconfiguration is assumed to be instantaneous, which means that the probability of system failure due to near-coincident failures is zero. This assumption eliminates the need for recovery transitions. The three sensors are redundant, but are linked to specific computers (e.g., s1 to c1), and thus there is failure dependency between them. If s1, s2 and c3 all fail, system failure occurs. However, if s1, s2 and c1 fail, the system is still operational. The sensor/computer combinations producing system failure are:

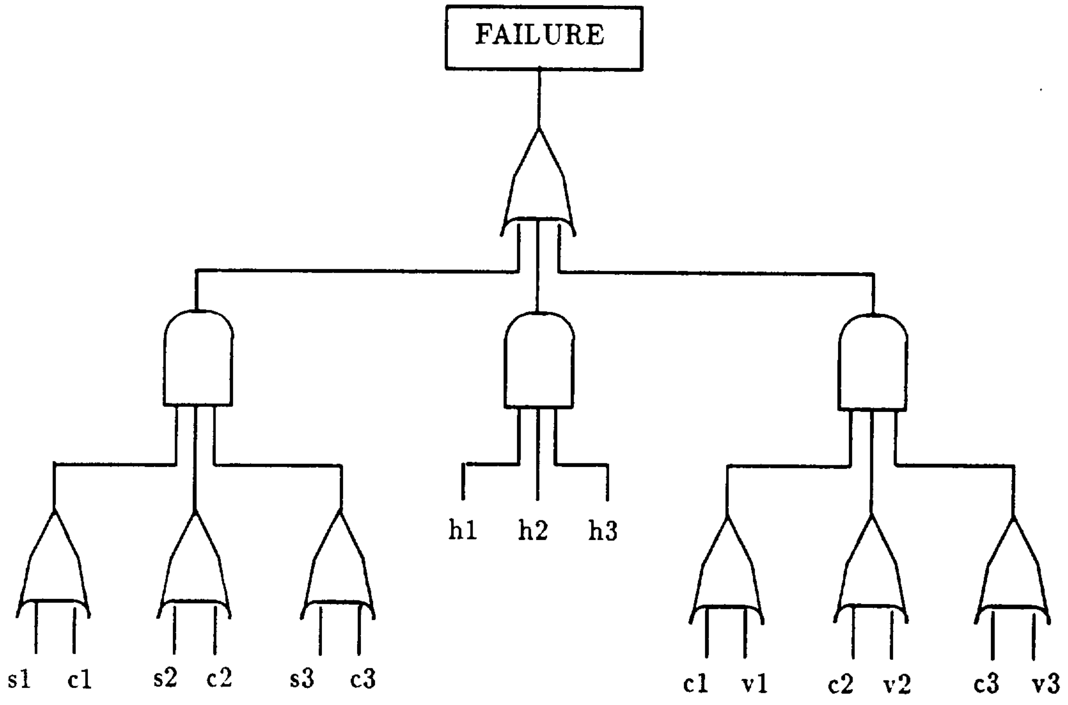


Figure 35: Fault Tree of Failure Modes

```
    AND( OR(s1,c1), OR(s2,c2), OR(s3,c3) )
```

Similarly, the servos are linked with the computers. System failure occurs for the following combinations of servo/computer failures:

```
    AND( OR(v1,c1), OR(v2,c2), OR(v3,c3) )
```

System failure also occurs when no hydraulic units are working:

```
    AND( h1, h2, h3 )
```

The fault-tree of figure 35 defines all of these failure modes together.

Since certain failure combinations of specific processors and specific sensors cause system failure, it is not possible to classify the states simply by a count of the number of working processors. It is necessary to keep track of each component separately. Thus, the state space is:

```
SPACE = (WS : ARRAY[1..3] OF 0..1,
         WC : ARRAY[1..3] OF 0..1,
         WV : ARRAY[1..3] OF 0..1,
         NH : 0..3;
```

Each of the ARRAY constructs defines three boolean variables, thus, the state space consists of 9 boolean variables (i.e {0,1} domain) and one integer variable. The complete ASSIST model description is:

```
 SPACE = (WS : ARRAY[1..3] OF 0..1,  (* Status of the 3 sensors *)
          WC : ARRAY[1..3] OF 0..1,  (* Status of the 3 computers *)
          WV : ARRAY[1..3] OF 0..1,  (* Status of the 3 servos *)
          NH : 0..3);                (* Number of working hydraulics units *)

START = (3 OF 1, 3 OF 1, 3 OF 1, 3); (* All components working *)

LS = 7.62E-4;                        (* Failure rate of sensor *)
LC = 3.50E-4;                        (* Failure rate of computer *)
LV = 3.90E-4;                        (* Failure rate of servo *)
LH = 6.00E-5;                        (* Failure rate of hydraulics *)

FOR I = 1,3
   IF WS[I] > 0 TRANTO WS[I] = WS[I] - 1 BY LS;   (* Sensor failure *)
   IF WC[I] > 0 TRANTO WC[I] = WC[I] - 1 BY LC;   (* Computer failure *)
   IF WV[I] > 0 TRANTO WV[I] = WV[I] - 1 BY LV;   (* Servo failure *)
ENDFOR;

IF NH > 0 TRANTO NH = NH - 1 BY NH*LH; (* Hydraulics unit failure *)

DEATHIF (WS[1]=0 OR WC[1]=0) AND       (* Enumeration of sensor/computer  *)
        (WS[2]=0 OR WC[2]=0) AND       (* combinations leading to failure *)
        (WS[3]=0 OR WC[3]=0);

DEATHIF NH = 0;                        (* Loss of all hydraulics units *)

DEATHIF (WV[1]=0 OR WC[1]=0) AND       (* Enumeration of servo/computer   *)
        (WV[2]=0 OR WC[2]=0) AND       (* combinations leading to failure *)
        (WV[3]=0 OR WC[3]=0);
```

The FOR loop effectively creates 9 TRANTO rules. These create failure transitions corresponding to failures of the 9 individual components. Since there are no dependencies between the hydraulics and the other parts of the system, the individual units do not have to be separately accounted for. Thus, the last TRANTO rule merely decrements the count of working hydraulic units. The DEATHIF statements define all of the failure combinations. The ASSIST output is:

```
ASSIST VERSION 6.0
The Front End Routine FER SURE

 PROCESSING TIME = 145.62
 NUMBER OF STATES IN MODEL      = 618
 NUMBER OF TRANSITIONS IN MODEL = 4116
 1672 DEATH STATES AGGREGATED INTO STATES 1 - 3
```

C.2

This model is large enough to require significant time to generate and to solve. The reason the model is so large is because there are many combinations of components in the system that can fail before system failure occurs. In fact, there are many combinations of up to 5 or 6 component failures consisting of one or two failures of each type of 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 consisting of failures of only 3 or 4 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 those 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 those paths. Both the ASSIST and SURE programs provide the capability to prune paths in the model automatically.

With the SURE program, the user specifies a probability level for model pruning, for example 10^{-15}, and each time the probability of encountering a state in the model falls below the specified value that path is pruned. The program sums the probabilities of all of the pruned paths and reports that value to the user as the estimated error due to pruning. Pruning in SURE is very effective in reducing the execution time required to solve most large models. However, this does nothing to reduce the execution time required to generate the large model or the amount of memory required to store a large model.

Pruning in the ASSIST program can reduce the model generation time and memory requirements as well as the solution time. However, since 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 3 or 4 component failures, then he can command ASSIST to prune the model at the 4th component failure. Two ways to specify this are shown below. The easiest method is to write a PRUNEIF statement that calculates the component failure level directly from the state space variables:

```
PRUNEIF (9 - WS[1]+WS[2]+WS[3]+WC[1]+WC[2]+WC[3]+WV[1]+WV[2]+WV[3]) + 3-NH >= 4;
```

This command is added to the ASSIST input file. However, this type of PRUNEIF statement can be quite complicated, and calculation of component failure level directly from the other state space variable values is not possible for some models. The second method is to introduce a new state space variable, say "NF", specifically for the purpose of keeping a count of the number of component failures in the system. This state space variable must be incremented in every TRANTO statement that defines a component failure. The complete

ASSIST input file for the ARCS architecture with pruning at the fourth component failure level is thus:

```
LS = 7.62E-4;     (* Failure rate of sensors *)
LV = 3.90E-4;     (* Failure rate of servos *)
LC = 3.50E-4;     (* Failure rate of computers *)
LH = 6E-5;        (* Failure rate of hydraulics units *)

SPACE = (WS : ARRAY[1..3] OF 0..1,  (* Status of the 3 sensors *)
         WC : ARRAY[1..3] OF 0..1,  (* Status of the 3 computers *)
         WV : ARRAY[1..3] OF 0..1,  (* Status of the 3 servos *)
         NH : 0..3,                 (* Number of working hydraulics units *)
         NF: 0..12);                (* Number of component failures *)

START = (3 OF 1, 3 OF 1, 3 OF 1, 3, 0); (* All components working *)

FOR I = 1,3
   IF WS[I] > 0 TRANTO WS[I] = WS[I] - 1, NF = NF + 1 BY LS; (* Sensor fails *)
   IF WC[I] > 0 TRANTO WC[I] = WC[I] - 1, NF = NF + 1 BY LC; (* Compu. fails *)
   IF WV[I] > 0 TRANTO WV[I] = WV[I] - 1, NF = NF + 1 BY LV; (* Servo fails *)
ENDFOR;

IF NH > 0 TRANTO NH = NH - 1, NF = NF + 1 BY NH*LH;  (* Hydraulics unit fails *)

DEATHIF (WS[1]=0 OR WC[1]=0) AND     (* Enumeration of sensor/computer  *)
        (WS[2]=0 OR WC[2]=0) AND     (* combinations leading to failure *)
        (WS[3]=0 OR WC[3]=0);

DEATHIF NH = 0;                      (* Loss of all hydraulics units *)

DEATHIF (WV[1]=0 OR WC[1]=0) AND     (* Enumeration of servo/computer   *)
        (WV[2]=0 OR WC[2]=0) AND     (* combinations leading to failure *)
        (WV[3]=0 OR WC[3]=0);

PRUNEIF NF >= 4;           (* Pruning at fourth component failure level *)
```

The ASSIST program reports the number of states at which the model was pruned:

```
$ assist arcs
ASSIST VERSION 6.0
The Front End Routine FER SURE

 PROCESSING TIME = 15.60
 NUMBER OF STATES IN MODEL      = 175
 NUMBER OF TRANSITIONS IN MODEL = 1332
 262 DEATH STATES AGGREGATED INTO STATES 1 - 3
 636 PRUNED STATES AGGREGATED INTO STATES 4 - 4
 THANK YOU FOR USING ASSIST, FER SURE
```

Thus, pruning at the fourth component level reduced the model from 618 states and 4116 transitions to only 175 states and 1332 transitions. All of the pruned states are lumped together into one state. If there had been more than one PRUNEIF statement, the pruned states would have been lumped according to which PRUNEIF statement they satisfied. In this example, all of the paths pruned by the ASSIST program end in state number four.

Whenever the ASSIST input file includes one or more PRUNEIF statements, the program automatically includes a statement in the SURE input file indicating which states are pruned states generated by ASSIST. For example, if an ASSIST input file contained two PRUNEIF statements and the model generated has states four and five as prune states, then the statement

```
    PRUNESTATES = (4,5);
```

would be included in the ASSIST output file, i.e., the ".mod" file.

The output from the SURE run is:

```
$ sure

  SURE V7.4   NASA Langley Research Center

  1? read0 arcs

      11.10 SECS. TO READ MODEL FILE

4010? list=2
4011? run

 MODEL FILE = arcs.mod                          SURE V7.4 24 Jan 90   12:12:20



 DEATHSTATE     LOWERBOUND    UPPERBOUND     COMMENTS                 RUN #1
 ----------    -----------   -----------     -------------------------------
       1       1.35113e-06   1.39351e-06
       2       2.15552e-10   2.23300e-10
       3       3.57167e-07   3.69102e-07
               -----------   -----------
   SUBTOTAL    1.70852e-06   1.76283e-06

 PRUNESTATE     LOWERBOUND    UPPERBOUND
 ----------    -----------   -----------
 prune    4    5.07787e-08   5.24308e-08
               -----------   -----------
   SUBTOTAL    5.07787e-08   5.24308e-08


 TOTAL         1.70852e-06   1.81526e-06

  4960 PATH(S) TO DEATH STATES
 51.467 SECS. CPU TIME UTILIZED
```

```
4012? exit
```

The ASSIST prune states are reported separately from the death states as follows. 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 user has probably pruned his model too severely in ASSIST. The upper and lower bounds can be made significantly closer by relaxing the amount of pruning.

For this example, the upper bound on the error due to the pruning done in ASSIST is 5.24308×10^{-8}.

Next, we will consider the effect of imperfect coverage of single-point failures. This will be done by using "coverage parameters". The system is known to fail a certain fraction of the time in the presence of a single fault. In the ARCS architecture this is assumed to only occur after one of the units in a triad has failed and been removed. In other words, there is perfect coverage when the triad is working. Once it has degraded to a duplex, then it is subject to single point failures. The probability that a single fault causes system failure (in duplex mode) are COV_S, COV_V, COV_C, and COV_H for sensors, servos, computers and hydraulics, respectively. To simplify the `DEATHIF` statements, a new variable `SPF` is added to the state space. The SPF variable is originally set to 0. If a single fault leads to failure then SPF is set to 1.

```
LS = 7.62E-4;      (* Failure rate of sensors *)
LV = 3.90E-4;      (* Failure rate of servos *)
LC = 3.50E-4;      (* Failure rate of computers *)
LH = 6E-5;         (* Failure rate of hydraulics units *)

COV_S = .7231;     (* Sensor single-point failure coverage *)
COV_V = .95;       (* Servo single-point failure coverage *)
COV_C = .95;       (* Computer single-point failure coverage *)
COV_H = .95;       (* Hydraulics single-point failure coverage *)

SPACE = (WS : ARRAY[1..3] OF 0..1,  (* Status of the 3 sensors *)
         WC : ARRAY[1..3] OF 0..1,  (* Status of the 3 computers *)
         WV : ARRAY[1..3] OF 0..1,  (* Status of the 3 servos *)
         WH : 0..3,                 (* Number of working hydraulics units *)
         SPF: 0..1,                 (* Single-point failure flag *)
         NF : 0..12);               (* Number of component failures *)

START = (3 OF 1, 3 OF 1, 3 OF 1, 3, 0, 0);

  (* Sensor failures *)
IF WS[1] + WS[2] + WS[3] = 3 THEN   (* Triplex *)
   TRANTO WS[1] = 0, NF = NF + 1 BY LS;
   TRANTO WS[2] = 0, NF = NF + 1 BY LS;
   TRANTO WS[3] = 0, NF = NF + 1 BY LS;
```

```
ELSE   (* Vulnerable to SPF *)
   TRANTO WS[1] = 0, NF = NF + 1 BY COV_S*LS;
   TRANTO WS[2] = 0, NF = NF + 1 BY COV_S*LS;
   TRANTO WS[3] = 0, NF = NF + 1 BY COV_S*LS;
   TRANTO SPF = 1, NF = NF + 1   BY (1-COV_S)*(WS[1] + WS[2] + WS[3])*LS;
ENDIF;

  (* Computer failures *)
IF WC[1] + WC[2] + WC[3] = 3 THEN  (* Triplex *)
   TRANTO WC[1] = 0, NF = NF + 1 BY LC;
   TRANTO WC[2] = 0, NF = NF + 1 BY LC;
   TRANTO WC[3] = 0, NF = NF + 1 BY LC;
ELSE   (* Vulnerable to SPF *)
   TRANTO WC[1] = 0, NF = NF + 1 BY COV_C*LC;
   TRANTO WC[2] = 0, NF = NF + 1 BY COV_C*LC;
   TRANTO WC[3] = 0, NF = NF + 1 BY COV_C*LC;
   TRANTO SPF = 1, NF = NF + 1   BY (1-COV_C)*(WC[1] + WC[2] + WC[3])*LC;
ENDIF;

  (* Servo failures *)
IF WV[1] + WV[2] + WV[3] = 3 THEN  (* Triplex *)
   TRANTO WV[1] = 0, NF = NF + 1 BY LV;
   TRANTO WV[2] = 0, NF = NF + 1 BY LV;
   TRANTO WV[3] = 0, NF = NF + 1 BY LV;
ELSE  (* Vulnerable to SPF *)
   TRANTO WV[1] = 0, NF = NF + 1 BY COV_V*LV;
   TRANTO WV[2] = 0, NF = NF + 1 BY COV_V*LV;
   TRANTO WV[3] = 0, NF = NF + 1 BY COV_V*LV;
   TRANTO SPF = 1   BY (1-COV_V)*(WV[1] + WV[2] + WV[3])*LV;
ENDIF;

  (* Hydraulics failures *)
IF WH = 3 THEN   (* Triplex *)
   TRANTO WH = WH - 1, NF = NF + 1 BY WH*LH;
ELSE   (* Vulnerable to SPF *)
   TRANTO WH = WH - 1, NF = NF + 1 BY COV_H*WH*LH;
   TRANTO SPF = 1   BY (1-COV_H)*WH*LH;
ENDIF;

DEATHIF SPF = 1;                     (* Single-point failure *)

DEATHIF (WS[1]=0 OR WC[1]=0) AND     (* Enumeration of sensor/computer  *)
        (WS[2]=0 OR WC[2]=0) AND     (* combinations leading to failure *)
        (WS[3]=0 OR WC[3]=0);

DEATHIF WH = 0;                      (* Loss of all hydraulics units *)

DEATHIF (WV[1]=0 OR WC[1]=0) AND     (* Enumeration of servo/computer   *)
        (WV[2]=0 OR WC[2]=0) AND     (* combinations leading to failure *)
        (WV[3]=0 OR WC[3]=0);

PRUNEIF (NF >= 4);  (* Pruning at fourth component failure level *)

COMMENT=0;    (* Tells ASSIST not to print the state space variable *)
              (*   values for each state in comments, to decrease   *)
              (*   the memory needed to hold the .MOD file          *)
LIST=2;
```

This input file generates a model with 239 states and 2813 transitions:

```
$ assist arcs2
ASSIST VERSION 6.0
The Front End Routine FER SURE

 PROCESSING TIME = 26.50
 NUMBER OF STATES IN MODEL      = 239
 NUMBER OF TRANSITIONS IN MODEL = 2813
 780 DEATH STATES AGGREGATED INTO STATES 1 - 4
 1419 PRUNED STATES AGGREGATED INTO STATES 5 - 5
 THANK YOU FOR USING ASSIST, FER SURE
```

The output from the SURE program for this model is:

```
$sure

  SURE V7.4   NASA Langley Research Center

  1? read0 arcs2

      19.30 SECS. TO READ MODEL FILE
2833? list=2
2834? run

 MODEL FILE = arcs2.mod                        SURE V7.4 24 Jan 90   12:41:21


 DEATHSTATE    LOWERBOUND    UPPERBOUND    COMMENTS                  RUN #1
 ----------    -----------   -----------   ----------------------------------
      1        5.18371e-05   5.34767e-05
      2        9.59118e-07   9.93014e-07
      3        1.94166e-10   2.01157e-10
      4        3.36427e-07   3.48644e-07
 sure prune    0.00000e+00   2.60604e-09
               -----------   -----------
   SUBTOTAL    5.31328e-05   5.48186e-05

 PRUNESTATE    LOWERBOUND    UPPERBOUND
 ----------    -----------   -----------
 prune    5    1.03318e-07   1.07218e-07
               -----------   -----------
   SUBTOTAL    1.03318e-07   1.07218e-07


 TOTAL         5.31328e-05   5.49284e-05

  11430 PATH(S) TO DEATH STATES, 1 PATH(S) PRUNED
  HIGHEST PRUNE LEVEL =  3.99916e-09
 116.700 SECS. CPU TIME UTILIZED
2835? exit
```

The SURE program pruned one additional path. The SURE program automatically sets a pruning level based on the value of the first death state encountered in the model. The SURE program reports a bound on the error due to SURE-level pruning in a separate row:

```
 sure prune    0.00000e+00   2.60604e-09
```

The SURE-level pruning can be disabled by issuing the **AUTOPRUNE** command

```
     AUTOPRUNE = 0;
```

in the SURE input file. BY default **AUTOPRUNE** = 1. Once the model has been solved, subsequent runs can be accelerated by specifying a manual **PRUNE** level, e.g.

```
     PRUNE = 5E-9;
```

The result is:

```
$ sure
  SURE V7.4   NASA Langley Research Center

  1? read0 arcs2

      19.20 SECS. TO READ MODEL FILE
2833? prune=5e-9
2834? run

 MODEL FILE = arcs2.mod                      SURE V7.4 24 Jan 90   13:07:15


 DEATHSTATE    LOWERBOUND   UPPERBOUND   COMMENTS                  RUN #1
 ----------    -----------  -----------  ---------------------------------
      1        5.18370e-05  5.34766e-05
      2        9.58952e-07  9.92843e-07
      3        1.88177e-10  1.94940e-10
      4        3.35984e-07  3.48183e-07
 sure prune    0.00000e+00  4.78666e-07
               -----------  -----------
   SUBTOTAL    5.31321e-05  5.48178e-05

 PRUNESTATE    LOWERBOUND   UPPERBOUND
 ----------    -----------  -----------
 prune    5    9.87666e-08  1.02492e-07
               -----------  -----------
   SUBTOTAL    9.87666e-08  1.02492e-07


 TOTAL         5.31321e-05  5.53990e-05
```

```
  9768 PATH(S) TO DEATH STATES, 135 PATH(S) PRUNED AT LEVEL  5.00000e-09

  99.416 SECS. CPU TIME UTILIZED
 2835? exit
```

Finally, we will remove the assumption of instantaneous reconfiguration. We have four types of components—sensors, servos, computers, and hydraulics—that all have recovery processes associated with them. Since all of these components can fail while others are recovering, this system could potentially experience up to four simultaneous competing recoveries. Rather than try to model all of these competing recoveries, we will use a bounding technique similar to one developed by Dr. Allan White and Daniel Palumbo [20], and conservatively assume that any second component failure during a recovery leads to system failure. This is accomplished by generating a transition to a death state from each state in which the system is experiencing a recovery. This transition to death will have a rate that is the sum of the failure rates for all other components in the system. We can conveniently reuse the SPF state space variable to flag that this is a death state, and we will distinguish the near-coincident failure deaths from the single-point failure deaths by checking for the presence of an active component failure:

```
DEATHIF (SPF = 1) AND (SA+VA+HA+CA > 0);  (* Near-coincident failures *)
DEATHIF SPF = 1;                          (* Single-point failures *)
```

Note that since the death states are lumped according to the first DEATHIF statement they satisfy, the more specific DEATHIF statement that uses the SPF flag to signal near-coincident failures as death states must precede the DEATHIF statement that defines true single-point failures as death states. If these two statements were reversed, all of these failures would be lumped as single-point failures.

The bounding near-coincident failure transitions must be defined for each component type—sensors, servos, computers, and hydraulics. For example, the transitions from each state with an active sensor failure are:

```
   TRANTO SA = 0 BY FAST REC_S;                 (* Recovery *)
   TRANTO SPF = 1 BY 2*LS + 3*LV + 3*LC + 3*LH; (* Conservative bound on    *)
                                                (* near-coincident failures *)
```

The complete ASSIST description of the model is:

```
LS = 7.62E-4;    (* Failure rate of sensors *)
LV = 3.90E-4;    (* Failure rate of servos *)
LC = 3.50E-4;    (* Failure rate of computers *)
LH = 6E-5;       (* Failure rate of hydraulics units *)
```

```
COV_S = .7231;  (* Sensor single-point failure coverage *)
COV_V = .95;    (* Servo single-point failure coverage *)
COV_C = .95;    (* Computer single-point failure coverage *)
COV_H = .95;    (* Hydraulics single-point failure coverage *)

REC_S = 1E4;   (* Exponential time to recovery from sensor failure *)
REC_V = 1E4;   (* Exponential time to recovery from servo failure *)
REC_C = 1E4;   (* Exponential time to recovery from computer failure *)
REC_H = 1E4;   (* Exponential time to recovery from hydraulics failure *)

SPACE = (WS : ARRAY[1..3] OF 0..1,  (* Status of the 3 sensors *)
         WC : ARRAY[1..3] OF 0..1,  (* Status of the 3 computers *)
         WV : ARRAY[1..3] OF 0..1,  (* Status of the 3 servos *)
         WH : 0..3,                 (* Number of working hydraulics units *)
         CA: 0..1,                  (* Fault in active computer *)
         VA: 0..1,                  (* Fault in active servo *)
         SA: 0..1,                  (* Fault in active sensor *)
         HA: 0..1,                  (* Fault in active hydraulics unit *)
         SPF: 0..1,                 (* Single-point failure flag *)
         NF : 0.. 12);              (* Number of component failures *)

START = (3 OF 1, 3 OF 1, 3 OF 1, 3, 4 OF 0, 0, 0);

  (* Sensors *)
IF SA = 1 THEN  (* Active sensor failure *)
   TRANTO SA = 0 BY FAST REC_S;                   (* Recovery *)
   TRANTO SPF = 1 BY 2*LS + 3*LV + 3*LC + 3*LH;  (* Conservative bound on    *)
                                                  (* near-coincident failures *)
ELSE   (* No active sensor failures *)
   IF WS[1] + WS[2] + WS[3] = 3 THEN  (* Triplex *)
      TRANTO WS[1] = 0, NF = NF + 1, SA = 1 BY LS;
      TRANTO WS[2] = 0, NF = NF + 1, SA = 1 BY LS;
      TRANTO WS[3] = 0, NF = NF + 1, SA = 1 BY LS;
   ELSE   (* Vulnerable to SPF *)
      TRANTO WS[1] = 0, NF = NF + 1, SA = 1 BY COV_S*LS;
      TRANTO WS[2] = 0, NF = NF + 1, SA = 1 BY COV_S*LS;
      TRANTO WS[3] = 0, NF = NF + 1, SA = 1 BY COV_S*LS;
      TRANTO SPF = 1,   NF = NF + 1 BY (1-COV_S)*LS;
   ENDIF;
ENDIF;

  (* Computers *)
IF CA = 1 THEN   (* Active computer failure *)
   TRANTO CA = 0 BY FAST REC_C;                   (* Recovery *)
   TRANTO SPF = 1 BY 3*LS + 3*LV + 2*LC + 3*LH;  (* Conservative bound on    *)
                                                  (* near-coincident failures *)
ELSE   (* No active computer failures *)
   IF WC[1] + WC[2] + WC[3] = 3 THEN   (* Triplex *)
      TRANTO WC[1] = 0, NF = NF + 1, CA = 1 BY LC;
      TRANTO WC[2] = 0, NF = NF + 1, CA = 1 BY LC;
      TRANTO WC[3] = 0, NF = NF + 1, CA = 1 BY LC;
   ELSE   (* Vulnerable to SPF *)
      TRANTO WC[1] = 0, NF = NF + 1, CA = 1 BY COV_C*LC;
      TRANTO WC[2] = 0, NF = NF + 1, CA = 1 BY COV_C*LC;
      TRANTO WC[3] = 0, NF = NF + 1, CA = 1 BY COV_C*LC;
      TRANTO SPF = 1, NF = NF + 1, CA = 1 BY (1-COV_C)*LC;
   ENDIF;
ENDIF;

  (* Servos *)
IF VA = 1 THEN   (* Active servo failure *)
   TRANTO VA = 0 BY FAST REC_V;                       (* Recovery *)
   TRANTO SPF = 1 BY 3*LS + 2*LV + 3*LC + 3*LH;       (* Conservative bound on    *)
                                                      (* near-coincident failures *)
ELSE   (* No active servo failures *)
   IF WV[1] + WV[2] + WV[3] = 3 THEN (* Triplex *)
      TRANTO WV[1] = 0, NF = NF + 1, VA = 1 BY LV;
      TRANTO WV[2] = 0, NF = NF + 1, VA = 1 BY LV;
      TRANTO WV[3] = 0, NF = NF + 1, VA = 1 BY LV;
   ELSE   (* Vulnerable to SPF *)
      TRANTO WV[1] = 0, NF = NF + 1, VA = 1 BY COV_V*LV;
      TRANTO WV[2] = 0, NF = NF + 1, VA = 1 BY COV_V*LV;
      TRANTO WV[3] = 0, NF = NF + 1, VA = 1 BY COV_V*LV;
      TRANTO SPF = 1 BY (1-COV_V)*LV;
   ENDIF;
ENDIF;

  (* Hydraulics *)
IF HA = 1 THEN   (* Active hydraulics failure *)
   TRANTO HA = 0 BY FAST REC_H;                       (* Recovery *)
   TRANTO SPF = 1 BY 3*LS + 3*LV + 3*LC + 2*LH;       (* Conservative bound on    *)
                                                      (* near-coincident failures *)
ELSE   (* No active hydraulics failure *)
   IF WH = 3 THEN   (* Triplex *)
      TRANTO WH = WH - 1, HA = 1 BY WH*LH;
   ELSE   (* Vulnerable to SPF *)
      TRANTO WH = WH - 1, HA = 1 BY COV_H*WH*LH;
      TRANTO SPF = 1 BY (1-COV_H)*WH*LH;
   ENDIF;
ENDIF;

DEATHIF (SPF = 1) AND (SA+VA+HA+CA > 0);  (* Near-coincident failures *)
DEATHIF SPF = 1;                          (* Single-point failures *)

DEATHIF (WS[1]=0 OR WC[1]=0) AND        (* All combinations of sensor/computer  *)
        (WS[2]=0 OR WC[2]=0) AND        (* failures that lead to system failure *)
        (WS[3]=0 OR WC[3]=0);

DEATHIF WH = 0;                         (* Loss of all hydraulics *)

DEATHIF (WV[1]=0 OR WC[1]=0) AND        (* All combinations of servo/computer   *)
        (WV[2]=0 OR WC[2]=0) AND        (* failures that lead to system failure *)
        (WV[3]=0 OR WC[3]=0);

PRUNEIF (NF = 3);                       (* Pruning at 3rd component failure level *)
```

The ASSIST output is:

```
$ assist arcs3
ASSIST VERSION 6.0
The Front End Routine FER SURE
```

```
 PROCESSING TIME = 181.97
 NUMBER OF STATES IN MODEL      = 821
 NUMBER OF TRANSITIONS IN MODEL = 8802
 2570 DEATH STATES AGGREGATED INTO STATES 1 - 5
 4185 PRUNED STATES AGGREGATED INTO STATES 6 - 6
```

The SURE output using the AUTOPRUNE feature is:

```
$ sure

  SURE V7.4   NASA Langley Research Center

  1? read0 arcs3

     119.17 SECS. TO READ MODEL FILE
26429? run

 MODEL FILE = arcs3.mod                      SURE V7.4 24 Jan 90   13:24:06

 DEATHSTATE    LOWERBOUND    UPPERBOUND    COMMENTS                  RUN #1
 ----------    -----------   -----------   ---------------------------------
      1        9.32078e-07   9.63921e-07
      2        2.50414e-05   2.58639e-05
      3        9.40684e-07   9.78264e-07
      4        1.93553e-10   2.01578e-10
      5        3.28266e-07   3.41884e-07
 sure prune    0.00000e+00   9.42664e-09
               -----------   -----------
   SUBTOTAL    2.72427e-05   2.81482e-05

 PRUNESTATE    LOWERBOUND    UPPERBOUND
 ----------    -----------   -----------
 prune    6    1.07040e-05   1.11349e-05
               -----------   -----------
   SUBTOTAL    1.07040e-05   1.11349e-05


 TOTAL         2.72427e-05   3.92925e-05

  12018 PATH(S) TO DEATH STATES, 1673 PATH(S) PRUNED
  HIGHEST PRUNE LEVEL =  1.54070e-10

 133.817 SECS. CPU TIME UTILIZED
26430? exit
```

The five death states correspond to the five DEATHIF statements in the ASSIST input file. The dominant death state is the second death state, which represents the probability of system failure due to single-point failures. The second largest failure mode is death state three, combinations of sensor/computer failures that lead to system failure. The conservative approximation of the probability of failure due to near-coincident failures, which is death state two, is almost two orders of magnitude smaller than the dominant failure mode.

The ASSIST pruning on this model, at the third component failure level, is very severe, and the calculated error due to pruning is of the same order of magnitude as the dominant failure mode probability. However, the size of the model will become quite large if the ASSIST pruning is moved back to the fourth component failure level. We can use a technique similar to the bounding technique used on the near-coincident failures to reduce the conservativeness of the calculation of error due to pruning. The state into which all of the pruned paths terminate, state six, is actually an operational state of the system, not a death state. Therefore, some other component in the system would have to fail before system failure is reached. This can be represented by adding an additional transition to the model from state six to some new death state not already defined in the model, say state 0. The rate for this new transition can be conservatively set as the sum of the failure rates of all of the components in the system:

```
    6,0 = 3*LS + 3*LV + 3*LC + 3*LH;
```

This rate is conservative because some of three of these components will already have failed at this point. In fact, the three lowest component failure rates can even be deleted from this calculation and the rate will still be conservative. (Note. One does not have to delete the statement PRUNESTATES=(6) from the generated model file. Since an operational state cannot be a prune state, SURE will ignore this statement.) The SURE output below shows that this modification, while still conservative, significantly reduced the calculated error due to ASSIST pruning:

```
$ sure

  SURE V7.4   NASA Langley Research Center

  1? read0 arcs3

     119.53 SECS. TO READ MODEL FILE

26432? 6,0 = 3*LS + 3*LV + 3*LC + 3*LH;
26433? run

 MODEL FILE = arcs3.mod                              SURE V7.4 24 Jan 90   13:37:00

 DEATHSTATE    LOWERBOUND    UPPERBOUND     COMMENTS                    RUN #1
 ----------    -----------   -----------    ---------------------------------
      0        1.24779e-07   1.30345e-07
      1        9.32079e-07   9.63921e-07
      2        2.50414e-05   2.58639e-05
      3        9.40697e-07   9.78278e-07
      4        1.93553e-10   2.01578e-10
     5         3.28268e-07   3.41886e-07
 sure prune    0.00000e+00   7.68769e-09

 TOTAL         2.73675e-05   2.82862e-05

  6464 PATH(S) TO DEATH STATES, 8760 PATH(S) PRUNED
  HIGHEST PRUNE LEVEL =  4.60607e-11

 87.917 SECS. CPU TIME UTILIZED
26433? exit
```

12.3 Two Triads with Three Power Supplies

This example consists of two triads of computers with one triad of power supplies connected such 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 is not an easy system 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 way to model this system is to use the state space variables as flags to indicate the failure of each computer and power supply in the system. This uses a large number of state space variables, but the system can be described using only a few simple TRANTO statements. The large number of state space variables, however, leads to an unnecessarily complex semi-Markov model. 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 rather brute-force method of modeling the system leads to a semi-Markov model with 70 states and 138 transitions to model this relatively simple system.

Using state space variables to represent the number of failed computers in each triad and adding a flag to signal the dependencies between failed computers, the system may be modeled with a much smaller state space. Combining the resulting complex transition rules by logical reasoning, the system described above can be modeled by the following input file:

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

SPACE = (NFP: ARRAY[1..2] OF 0..3, (* Number of failed            *)
                                    (*    computers in each triad *)
         NFS: 0..3,                 (* Number of failed power supplies *)
         SAME: 0..1);        (* Set to 0 if 2 failed computers are on    *)
                             (*    different power supplies, 1 otherwise *)
START = (0, 0, 0, 1);

DEATHIF NFP[1]>1 OR NFP[2]>1;
   (* The system fails if 2/3 computers in either triad fail *)

FOR I=1,2
   IF NFP[I]<3 THEN
      IF NFP[3-I]=1 THEN   (* Other triad has a failed computer *)
         TRANTO NFP[I] = NFP[I]+1 BY LAM_C;
            (* Failure of computer on same power supply as other failed one *)
         TRANTO NFP[I] = NFP[I]+1, SAME = 0 BY (2-NFP[I])*LAM_C;
            (* Failure of computer on different   *)
            (* power supply than other failed one *)
      ELSE
         TRANTO NFP[I] = NFP[I]+1 BY (3-NFP[I])*LAM_C;
            (* Failure of computer when other triad has no failures yet *)
      ENDIF;
   ENDIF;
ENDFOR;

IF (NFP[1]=0 AND NFP[2]=0) THEN
   TRANTO (NFP[1]+1, NFP[2]+1, NFS+1, 1) BY 3*LAM_PS;
      (* Power supply failures when no previous *)
      (* computer failures have occurred.       *)
ELSE
   TRANTO (2, 2, 2, 0) BY (3-SAME)*LAM_PS;
      (* Failure of a power supply not connected to another  *)
      (* previously failed computer.  NOTE: State (2,2,2,1)  *)
      (* is an aggregation of several death states.          *)
   IF SAME = 1 TRANTO (1, 1, 1, 1) BY *LAM_PS;
      (* Failed power supply connected to *)
      (* a previously failed computer.    *)
ENDIF;
```

This second ASSIST input file leads to a semi-Markov model with only 17 states and 30 transitions to model the same system that using the first strategy required 70 states and 138 transitions. However, this input file is much 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.

12.4 Byzantine Faults

In this section we will introduce the concept of "Byzantine" faults and "Byzantine-resilient" algorithms. 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 there may be redundant sensors, these do not produce exactly the same result. Thus, if each processor were connected to one of the redundant sensors, they would get different results. This is unacceptable in a system which 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 the wrong processor. Furthermore, the problem is not solved by having each of the processors read all of the redundant sensors. Since the redundant processors run off of different clocks, they would access the sensors at slightly different times and receive different results. Consequently, a signal-processing algorithm is run on each of the processors to derive a trustworthy value from the set of redundant sensors. This necessitates that each sensor be distributed to all of the redundant processing sites in a consistent manner. Suppose the sensor value is read and stored. If there is a failure in the transmission medium between this value and the redundant sites, different values may be received by the good processors.

In order 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 4 separate fault-isolation regions. If processors are used for the rebroadcasting, then there must be a minimum of four processors. Consequently a simplex triplex system cannot be Byzantine resilient without the addition of special additional hardware. The model in figure 36 models the effect of a Byzantine fault on a triplex system with one spare that does not contain any extra hardware. This model is the same as the traditional triplex model except that it contains two extra transitions—from (2) to (5) and from (5) to (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 has been replaced by the spare and not the faulty one. 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*L;
2,4 = <MU_F,STD_F,1-P_W>;
2,5 = <MU_W,STD_W,P_W>;
```

The competing transitions at state (5) would be:

```
5,6 = 2*L;
```

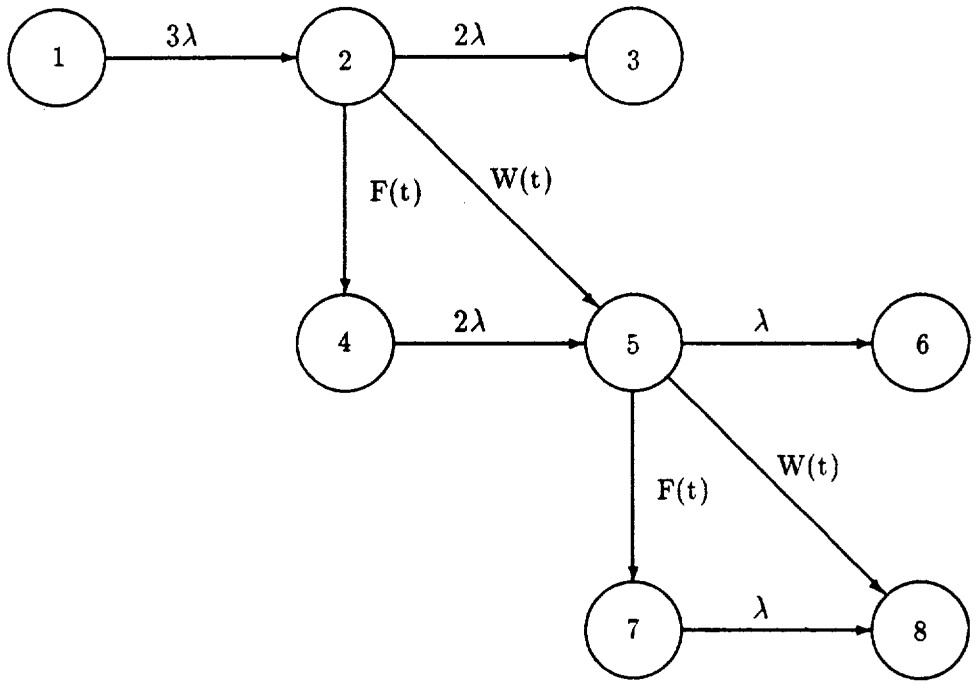


Figure 36: Simple Triplex System with One Spare Subject to Byzantine Faults

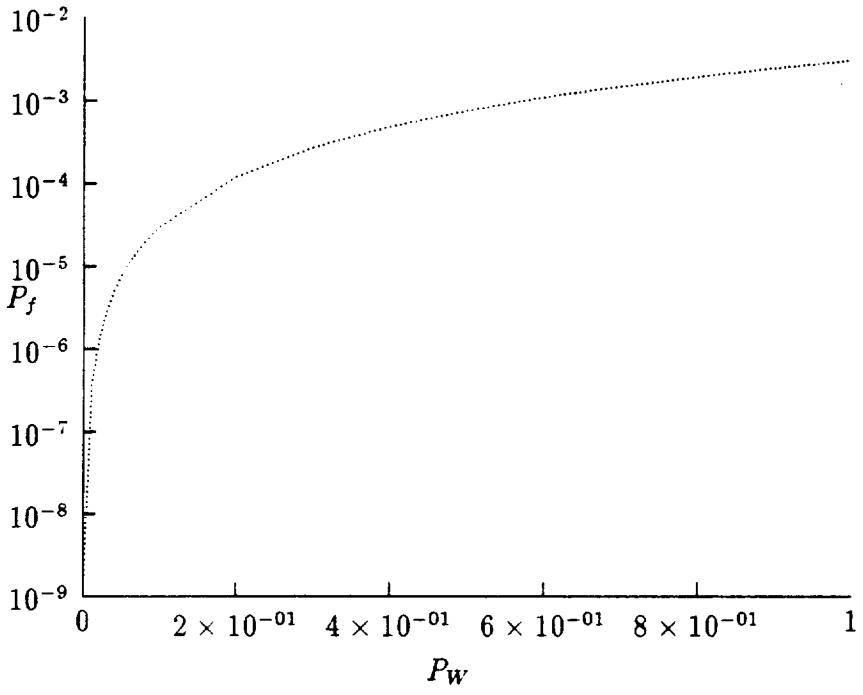


Figure 37: Failure Prob. As a Function of P_W

```
5,7 = <MU_F,STD_F,1-P_W>;
5,8 = <MU_W,STD_W,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, there is very little experimental data or methods available to aid in the estimation of P_W. For this reason, conservative fault-tolerant system designers have elected to add the additional hardware to make the architecture Byzantine-resilient. This eliminates this failure mode from the system. However, the failure of any additional hardware must be modelled.

13 Time and Sequence Dependencies

Many systems experience failure or recovery rates that are dependent on the current state or the failure history of the system. Several example systems exhibiting these types of dependencies are given in this section.

13.1 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 λ_P. The system initially contains two voltage regulators. These voltage regulators fail at rate λ_R. Once both of the voltage regulators fail, the processors are also subject to failure due to voltage surges, which arrive at an exponential rate λ_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>;
```

Since this model contains only one DEATHIF statement, all of the system failure probabilities will be lumped together. The addition of another DEATHIF statement placed in front of the one above could capture the probability of having both voltage regulators fail before the system failure condition is reached:

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

13.2 Recovery Rate Dependencies

In this system, the speed of the recovery process is significantly affected by the number of active components. This is accomplished by making the recovery rate a function of the state space variable NP, 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>;
```

14 Sequences of Reliability Models

The SURE program provides the user with the capability to calculate and store the probability of terminating in each of the operational states of the model as well as the death state probabilities. The program also allows the user to initialize a model 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.

14.1 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 liftoff 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 three-minute 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 as many phases as there are in the mission.

The SURE program reports upper and lower bounds on the operational states just as for the death states. These bounds are not as tight as the death state probabilities, but are usually acceptable. The upper and lower bounds on recovery states (i.e. states with fast transitions leaving them) are usually not very close together. Fortunately, these states usually have operational probabilities which 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, in subsequent phases the crudeness of the bounds for the recovery states in previous phases do 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 which operates in two basic phases—(1) cruise and (2) landing. The system is implemented using a triad of processors and two warm spares. For simplicity, we will assume perfect detection of spare failure. During the cruise phase which lasts for 2 hours, the system reconfigures by sparing and degradation. After the cruise phase, the system goes into a landing phase which lasts 3 minutes. During this 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.

In order to model this two-phased mission, two different models must be created—one 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_DE = 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 *),     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;

(* NUMBER OF STATES IN MODEL      = 8 *)
(* NUMBER OF TRANSITIONS IN MODEL = 14 *)
(* 4 DEATH STATES AGGREGATED STATES 1 - 1 *)
```

The model for the second phase (call it "phaz2.mod") is easily created with an editor by deleting the reconfiguration transitions and changing the mission time to 0.05 hours. The resulting file is:

```
NSI = 2;
LAMBDA = 1E-4;
GAMMA = 1E-6;
TIME = 0.05;
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 *) = 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 *) = 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 *) = 2*LAMBDA;

8(* 1,0,0 *),    1(* 0,1,0 *) = 1*LAMBDA;
```

The SURE program is then executed on the first model (stored in file "phaz.mod"), using the LIST = 3 option. This causes the SURE program to output all of the operational state probabities as well as the death state probabilities. This is illustrated below:

```
 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 probabilities in a format that can be used to initialize the states for the next phase. The SURE program names the file "phaz.ini", i.e. adds ".ini" to the file name. The contents of this file generated by the run above 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 using the SURE `INITIAL_PROBS` command. The second model must number its states in an equivalent manner to the first model. Note that ".ini" file output 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?
```

14.2 Non-Constant Failure Rates

In the previous section, a two-phased system was analyzed which required different models for each of the phases. 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 that experiences different failure rates for each of the phases:

- phase 1 (6 min): $\lambda = 2 \times 10^{-4}$, $\gamma = 10^{-4}$
- phase 2 (2 hours): $\lambda = 10^{-4}$, $\gamma = 10^{-5}$
- phase 3 (3 min): $\lambda = 10^{-3}$, $\gamma = 10^{-4}$

The same SURE model can be used for all of the phases, and the user can be prompted for the parameter values using the SURE **INPUT** command:

```
INPUT LAMBDA, GAMMA, TIME;
```

The full SURE model, 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
```

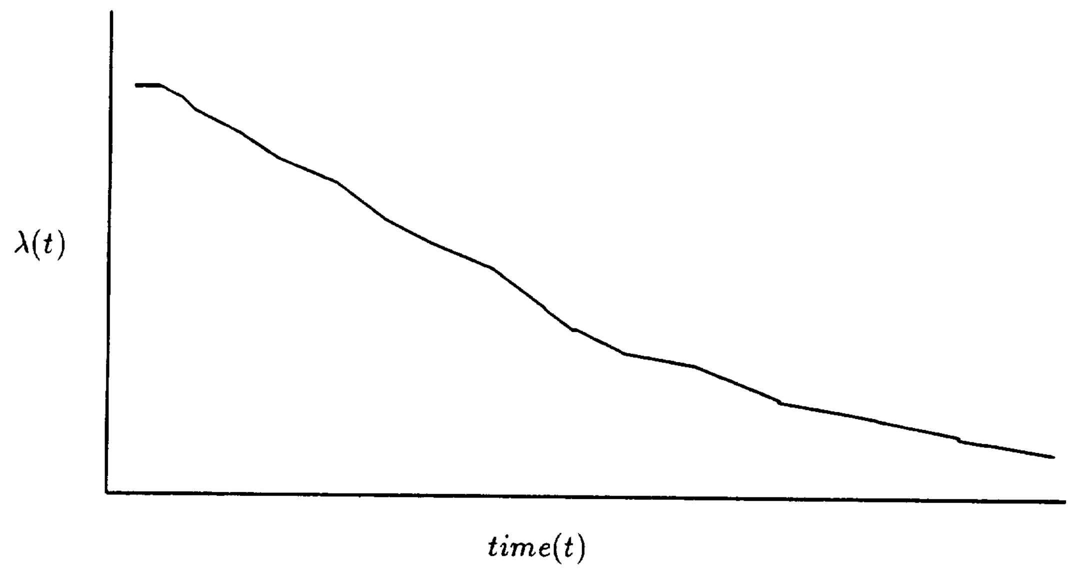


Figure 38: Decreasing Failure Rate Function

```
 13 PATH(S) PROCESSED
 3.350 SECS. CPU TIME UTILIZED
113? exit
```

As in the previous section, 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.

14.3 Continuously Varying Failure Rates

Suppose that the failure rates change continuously in time as shown in figure 38. This type of failure rate is called a "decreasing failure rate". The SURE program cannot handle this type of failure rate directly since it leads to "non-homogeneous" or "non-stationary" Markov models. However, good results can be obtained by using the phased-mission approach on a "linearized" upper bound shown in figure 39.

This problem 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 using a consistently lower bound for the failure rate function to obtain a lower bound on the system failure probability.

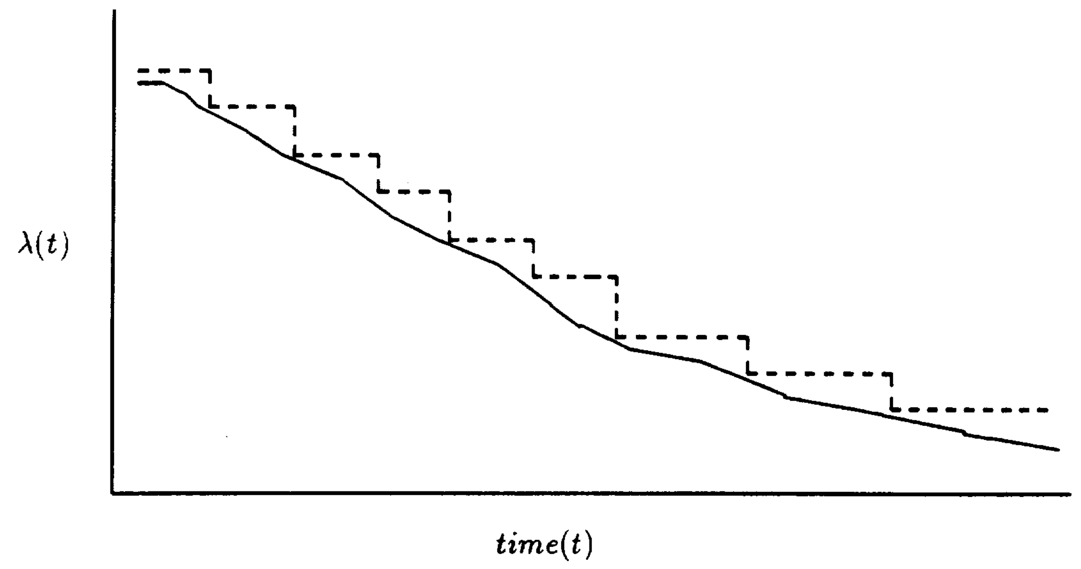


Figure 39: Upper bound On Failure-Rate Function

15 Concluding Remarks

In this paper, we have presented a number of techniques for developing reliability models of fault-tolerant systems. We have tried to present various modeling techniques in a systematic way, building from simple systems to more complicated ones, and introducing techniques for modeling specific aspects, such as single-point failures, near-coincident failures, transient-fault recoveries, cold spares, etc. However, it must be recognized that there is no "right" way to model a system—there are many valid ways to model a given system, and choosing which method will result in an efficient, informative model is more of an art than a science.

It is impossible to include every minute detail in a reliability model of a complex system, because such a model would be exhorbitantly 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. In reality, if two reliability engineers were modelling the same system independently, they would undoubtedly produce two different models because they would make different assumptions about the system.

Markov modeling can be a very powerful reliability analysis tool for three reasons. First, Markov models provide the reliability engineer the flexibility to include whatever assumptions or behaviors he wishes. Second, the reliability engineer is fully aware of the assumptions he is making because he must make them explicitly. And third, he 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, he is likely to forget them if they are not made visible to him. 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 of the flexibility of Markov models. It also requires the same level of modeling expertise.

We hope this paper can serve as a tutorial for reliability engineers learning how to develop Markov models of fault-tolerant systems.

References

[1] Siewiorek, Daniel P.; and Swarz, Robert S.: The Theory and Practice of Reliable System Design, Digital Press, 1982, pp. 31-42.

[2] *Reliability Prediction of Electronic Equipment.* MIL-HDBK-217D, U.S. Department of Defense, January 1982.

[3] Krishna, C. M.; Shin, K. G.; and Butler, R. W.: Synchronization and Fault-Masking in Redundant Real-Time Systems, *Fourteenth International Conference on Fault-Tolerant Computing (FTCS-14)*, Kissimmee, Florida, June 1984.

[4] Butler, Ricky W.; and Martensen, Anna L.: *The Fault-Tree Compiler (FTC).* NASA TP-2915, July 1989.

[5] Finelli, George B.: Characterization of Fault Recovery through Fault Injection on FTMP, *IEEE Transactions on Reliability*, Vol. R-36, No. 2, June 1987.

[6] Lala, Jaynarayan H.; and Smith, T. Basil, III: *Development and Evaluation of a Fault-Tolerant Multiprocessor (FTMP) Computer. Vol. III - FTMP Test and Evaluation.* NASA CR-166073, 1983.

[7] White, Allan L.: *Synthetic Bounds for Semi-Markov Reliability Models.* NASA CR-178008, 1985.

[8] Butler, Ricky W.; and White, Allan L.: *SURE Reliability Analysis: Program and Mathematics*, NASA TP-2764, March 1988.

[9] Bavuso, S. J.; and Petersen, P. L.: *CARE III Model Overview and User's Guide (First Revision).* NASA TM-86404, 1985.

[10] Dugan, J. B.; Trivedi, K. S.; Smotherman, M. K.; and Geist, R. M.: The Hybrid Automated Reliability Predictor, *Journal of Guidance, Control, and Dynamics*, Vol. 9, No. 3, May-June 1986, pp. 319-331.

[11] Butler, Ricky W; and Stevenson, Philip H.: *The PAWS and STEM Reliability Analysis Programs.* NASA TM-100572, March 1988.

[12] Goldberg, Jack, et. al.: *Development and Analysis of the Software Implemented Fault-Tolerance (SIFT) Computer.* NASA CR-172146, 1984.

[13] McGough, J. G. and Swern, F. L.: Measurement of Fault Latency in a Digital Avionic Mini Processor, NASA Contractor Report 3462, Oct. 1981.

[14] Johnson, Sally C.: *ASSIST User's Manual.* NASA TM-87735, August 1986.

[15] Johnson, Sally C.: Reliability Analysis of Large, Complex Systems using ASSIST, *AIAA/IEEE 8th Digital Avionics Systems Conference*, San Jose, California, October 1988.

[16] Hopkins, Albert L. Jr.; Smith, T. Basil III; and Jaynarayan, H. Lala: FTMP—A Highly Reliable Fault-Tolerant Multiprocessor for Aircraft, *Proceedings of the IEEE*, Vol 66, No. 10, Oct. 1978, pp.1221-1239.

[17] Butler, Ricky W; and Elks, Carl R.: *A Preliminary Transient-Fault Experiment on the SIFT Computer.* NASA TM-89058, February 1987.

[18] Bavuso, S. J., et. al.: Analysis of Typical Fault-Tolerant Architectures Using HARP, *IEEE Transactions on Reliability*, Vol. R-36, No. 2., June 1987.

[19] 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.

[20] White, Allan L., and Palumbo Daniel L.: State Reduction for Semi-Markov Reliability Models, *The 36th Annual Reliability and Maintainability Symposium*, Los Angeles, CA, January 1990.

NASA
National Aeronautics and Space Administration

Report Documentation Page

1. Report No. NASA TM-102623	2. Government Accession No.	3. Recipient's Catalog No.
4. Title and Subtitle The Art of Fault-Tolerant System Reliability Modeling		5. Report Date March 1990
		6. Performing Organization Code
7. Author(s) Ricky W. Butler and Sally C. Johnson		8. Performing Organization Report No.
		10. Work Unit No. 505-66-21-01
9. Performing Organization Name and Address NASA Langley Research Center Hampton, VA 23665-5225		11. Contract or Grant No.
		13. Type of Report and Period Covered Technical Memorandum
12. Sponsoring Agency Name and Address National Aeronautics and Space Administration Washington, DC 20546-0001		14. Sponsoring Agency Code

15. Supplementary Notes

16. Abstract

This paper presents a step-by-step tutorial of the methods and tools used for the reliability analysis of fault-tolerant systems. The emphasis is on the representation of architectural features in mathematical models. The paper does not present details of the mathematical solution of complex reliability models. Instead the paper describes the use of several recently developed computer programs--SURE, ASSIST, STEM, PAWS--which automate the generation and solution of these models.

17. Key Words (Suggested by Author(s)) Reliability Modeling, Markov Models, Reliability Analysis, Fault Tolerance		18. Distribution Statement Unclassified - Unlimited; Star Category 62	
19. Security Classif. (of this report) Unclassified	20. Security Classif. (of this page) Unclassified	21. No. of pages 131	22. Price A07