NASA Technical Paper 3301

July 1993

Automatic Specification of Reliability Models for Fault-Tolerant Computers

Carlos A. Liceaga and
Daniel P. Siewiorek

NASA Technical Paper 3301

1993

Automatic Specification of Reliability Models for Fault-Tolerant Computers

Carlos A. Liceaga
Langley Research Center
Hampton, Virginia

Daniel P. Siewiorek
Carnegie Mellon University
Pittsburgh, Pennsylvania

Contents

Tables

Figures

Summary

The calculation of reliability measures using Markov models is required for life-critical processor-memory-switch (PMS) structures that have standby redundancy or that are subject to transient or intermittent faults or repair. The task of specifying these models is tedious and prone to human error because of the large number of states and transitions required in any reasonable system. Therefore, model specification is a major analysis bottleneck, and model verification is a major validation problem. The general unfamiliarity of computer architects with Markov modeling techniques further increases the necessity of automating the model specification.

Automation requires a general system description language (SDL) that can accommodate new fault-tolerant techniques and system designs. For practicality, this SDL should also provide a high level of abstraction and be easy to learn and use.

This paper presents the first attempt to define and implement an SDL with those characteristics. The problems involved in the automatic specification of Markov reliability models for arbitrary interconnection structures at the PMS level are identified and analyzed. Solutions to these problems are generated and implemented.

A program named ARM (Automated Reliability Modeling) has been constructed as a research vehicle. The ARM program uses a graphical user interface (GUI) as its SDL. This GUI is based on a hierarchy of windows. Some windows have graphical editing capabilities for specifying the system's communication structure, hierarchy, reconfiguration capabilities, and requirements. Other windows have text fields, pull-down menus, and buttons for specifying parameters and selecting actions.

The ARM program outputs a Markov reliability model specification formulated for direct use by programs that generate and evaluate the model. The advantages of such an approach include utility to a larger class of users, who are not necessarily expert in reliability analysis, and lower probability of human error in the calculation.

1. Introduction

Computer systems are growing in complexity and sophistication as multiprocessors and distributed computers are coming into widespread use to achieve higher performance and reliability. This growth, which is being assisted by the availability of successively more complex building blocks, has increased the importance of fault tolerance and system reliability as design parameters. Thus, the calculation of system reliability measures has become one of the system design tasks. Several efforts have been reported in the literature and are in progress to make computing system reliability measures easier and more efficient by providing designers with reliability evaluation tools.

The analysis and evaluation of system reliability for complex computer systems is very tedious and prone to error even for experienced reliability analysts. The model of a system with n components can have up to 2^n states if it only has permanent faults and they are not removed. Therefore, the model of a system with just 10 components can have more than 1000 states.

With the exception of the ADVISER (Advanced Interactive Symbolic Evaluator of Reliability) and the RMG (Reliability Model Generator) programs, discussed in subsection 1.2, existing software tools usually assume an understanding of the failure modes and therefore are more in the nature of computational aids once the preliminary system decomposition and analysis have been manually achieved. Although ADVISER does not make this assumption, it uses combinatorial techniques, and it is therefore limited in the complexity of systems and fault types which it can analyze. The RMG program lacks a high-level system description language (SDL) that is easy to learn and use.

More advanced techniques are required to analyze computer architectures that use standby redundancy, can be repaired, or are susceptible to transient or intermittent faults. One possibility is the Markov model, which is discussed in subsection 1.1. The advantages offered by Markov models are that they are widely used among reliability analysts and that several programs, which are discussed in subsection 1.2, have been developed to solve them. However, Markov models cannot be used to analyze nonexponentially distributed concurrent events. For example, a fault that arrives while the system is reconfiguring itself around a previous fault would be represented by a transition to a state in which two faults are present. This new state would not take into account the time that the system has already spent reconfiguring from the first fault.

Another analysis possibility is the extended stochastic Petri net (ESPN) described in Dugan et al. (1984). The advantages offered by the ESPN are that it can analyze concurrent events and can model systems at a lower level of detail than Markov models. The ESPN "tokens" can be simultaneously enabled

to move concurrently at independent transition times. The low-level modeling capability is due to mechanisms, such as queues and counters, which can simulate the algorithm of the process being modeled. To analytically or numerically solve an ESPN, it must be converted to a Markov model. However, if tokens are moving concurrently at independent transition times that are not exponentially distributed, the process becomes non-Markovian (i.e., the transition probabilities depend on past states). This situation makes the conversion impossible. In general, an ESPN must be solved by simulation.

Simulations can include any level of detail, and they are thus flexible; however, for straightforward Monte Carlo simulations, many repetitions are needed to ensure accuracy. For example, in life-critical applications that require a probability of failure of 10^{-9} with a relative error of no more than 10 percent within a confidence interval of 95 percent, approximately 3.8×10^{11} simulation repetitions are necessary (Liceaga 1992). In general, these applications require a Markov model because it can be solved analytically or numerically.

This paper defines a general, high-level SDL that is easy to learn and use, identifies and analyzes the problems involved in the automatic specification of Markov reliability models for arbitrary interconnection structures at the processor-memory-switch (PMS) level,[1] and generates and implements solutions to these problems. The results of this research have been implemented and experimentally validated in the ARM (Automated Reliability Modeling) program.

The ARM program uses a graphical user interface (GUI) as its SDL. This GUI is based on a hierarchy of windows implemented in the C programming language using the Transportable Application Environment Plus (TAE Plus) user interface development tool for building X window-based applications (Szczur 1990). Some windows have graphical editing capabilities for specifying the system's communication structure, hierarchy, reconfiguration capabilities, and requirements. These windows have been implemented using the schematic drawing editor Schem (Vlissides 1990). Other windows have text fields, pull-down menus, and buttons for specifying parameters and selecting actions.

The ARM software outputs a Markov reliability model specification formulated for direct use by programs that generate the model. The advantages of such an approach are utility to a larger class of users, who are not necessarily expert in reliability analysis, and lower probability of human error in the calculation.

A brief background on reliability calculation at the PMS level using Markov models is presented in subsection 1.1. Previous work in the specification, generation, and evaluation of reliability models is surveyed in subsection 1.2. The goals for ARM are stated and compared with those of previous efforts in subsection 1.3. The organization of this paper is presented in subsection 1.4.

1.1. Background

Present-day computer systems and the process of designing and analyzing them can be viewed at various levels of detail. Four levels, which are defined in work by Siewiorek et al. (1982), range from the circuit level, through the logic and programming levels, to the PMS level. The PMS-level view of digital systems is one in which the primitives include processors, memories, switches, and transducers, as opposed to the logic level in which the primitives include gates, registers, and multiplexers.

Hardware components are susceptible to hard and soft faults as discussed by Siewiorek and Swarz (1992). A fault is an incorrect state of hardware or software resulting from a physical change in the hardware, interference from the environment, or design mistakes (Laprie 1985). Hard or permanent faults are continuous and stable, and they result from an irreversible physical change. Soft faults can be transient or intermittent. Transient faults result from temporary environmental conditions. Intermittent faults are occasionally active because of unstable hardware or varying hardware or software states (e.g., as a function of load or activity). Depending on whether the intermittent fault is benign or active, the output of the component will be correct or not, respectively.

Fault-tolerant computer systems can be affected by a limited set of faults without interruptions in their operation. Some computer systems achieve fault tolerance by using redundant groups of components to perform the same operations. The system must determine which is the correct output using diagnostics or majority voting. The various redundancy techniques are discussed in Siewiorek and Swarz (1992), and the more relevant ones are defined below:

Static redundancy: Faults are masked through a majority vote involving a fixed group of redundant components. Thus, when the number

[1] Components are not limited to being a processor, memory, or switch.

of faulty components reaches the maximum that can be tolerated, any further faults will cause errors at the output. Figure 1.1 illustrates a statically redundant processor (P) triad (a group of three components) and its voter (V).

Figure 1.1. Statically redundant processor triad.

Dynamic redundancy: Faults are not masked from causing errors at the output, but the faulty components are detected, isolated, and reconfigured out of the system. The faulty components are replaced when spares are available. Figure 1.2 illustrates a dynamically redundant active processor (AP) with m spares (SP).

Figure 1.2. Dynamically redundant active processor with m spares.

Hybrid redundancy: Faults are masked through a majority vote involving a group of redundant components which are reconfigured when spares are available. Thus, when the number of faulty components reaches the maximum that can be tolerated, any further faults that occur before a faulty component is replaced by a spare will cause errors at the output. Figure 1.3 illustrates a hybrid-redundant triad of active processors with m spares.

Adaptive voting: Faults are masked through a majority vote involving a variable group of redundant components without spares. Faulty components are reconfigured out of the system by excluding them from the voting process. Thus, when the number of faulty components reaches the maximum that can be tolerated, any further faults that occur before a faulty component is reconfigured out of the voting process will cause errors at the output. Figure 1.4 illustrates adaptive voting with n processors and their voter (AV).

Figure 1.3. Hybrid-redundant triad of active processors with m spares.

Figure 1.4. Adaptive voting with n processors.

Adaptive hybrid: Faults are masked through a majority vote involving a variable group of redundant components which are replaced when spares are available. If spares are not available, faulty components are reconfigured out of the system by excluding them from the voting process. Thus, when the number of faulty components reaches the maximum that

can be tolerated, any further faults that occur before a faulty component is replaced by a spare or reconfigured out of the voting process will cause errors at the output. Figure 1.5 illustrates an adaptive hybrid n-tuple of active processors with m spares.

Figure 1.5. Adaptive hybrid n-tuple of active processors with m spares.

For example, if a triad that uses hybrid redundancy "recovers" from a fault by replacing the faulty component with a spare, it can then tolerate a second fault. The following two definitions are those that will be used in this paper, but neither term has a universally accepted definition:

Recovery: The process of detecting, isolating, and reconfiguring a faulty component out of the system.

Coverage: The probability that the system can survive a fault in a component and successfully recover. (If the system can always recover, it has a "perfect" coverage of 1.)

Spares are sometimes left unpowered until they become part of the active configuration to reduce their failure rates (Avižienis et al. 1971). They are sometimes said to be cold if their failure rates are assumed to be 0, warm if their failure rates are reduced but not 0, or hot if their failure rates are not reduced (Butler and Johnson 1990).

Reliability measures are defined in terms of probabilities because the failure processes in hardware components are nondeterministic. These various measures are discussed in Siewiorek and Swarz (1992). The more relevant ones are defined below:

Reliability: The conditional probability $R(t)$ that the system is operational throughout the interval $[0,t]$ given that it was operational at time 0. (This measure is a nonincreasing function whose initial value is 1.)

Availability: The probability $A(t)$ that the system is operational at time t.

Mean time to failure (MTTF): The expected time of the first system failure assuming a new (perfect) system at time 0.

Mean time to repair (MTTR): The expected time for repair of a failed system.

Mean time between failures (MTBF): The expected time between failures in systems with repair.

Availability is typically used as a figure of merit in systems in which service can be delayed or denied for short periods to perform preventive maintenance or repair without serious consequences. The availability is important in the calculation of system life-cycle costs. If the limit of $A(t)$ exists as t goes to infinity, it expresses the expected fraction of time that the system is available to perform useful computations and has the following form:

$$\lim_{t\to\infty} A(t) = \frac{\text{MTTF}}{\text{MTBF}}$$

The MTBF is given by:

$$\text{MTBF} = \text{MTTF} + \text{MTTR}$$

Reliability is used to describe systems in which repair is typically infeasible, such as space applications. The MTTF can be derived from $R(t)$ as follows:

$$\text{MTTF} = \int_0^\infty R(t)\, dt$$

The most commonly used reliability function for a single component, which is based on a Poisson process with an exponential distribution, is called the exponential reliability function, and it has the form

$$R(t) = e^{-\lambda t}$$

where λ is the hazard or failure rate. The failure rate is a constant that reflects the quality of the component and is usually expressed in failures per million

hours for high-quality components. The exponential reliability function is used when the failure rate is time independent, such as when components do not age. After a burn-in period, permanent faults in electronic components often follow a relatively constant failure rate. The MTTF for the exponential reliability function has the following form:

$$\text{MTTF} = \frac{1}{\lambda}$$

Many other reliability functions have been formulated. The second most common reliability function, which is based on the Weibull distribution, is called the Weibull reliability function, and it has the form

$$R(t) = e^{-(\lambda t)^{\alpha}}$$

where λ in this case is the scale parameter and α is the shape parameter. (Other reparameterized forms are also common.) It is equivalent to the exponential function when α is 1. The Weibull reliability function is used when the failure rate is time dependent. Permanent faults for components that age can be described using an increasing failure rate ($\alpha > 1$). In that case, the system is not as good as new when repair takes place. Data presented in McConnel (1981) and Castillo, McConnel, and Siewiorek (1982) indicate that transient faults follow a decreasing failure rate ($\alpha < 1$).

The failure processes of different components are assumed to be independent of one another. This assumption is not strictly true, such as when electrical, mechanical, or thermal conditions in one component affect other components in its proximity. However, the assumption is close enough in practice to be used to simplify the analysis.

The state of a system represents all that must be known to describe the system at any instant. As the system changes, such as when components fail or are repaired, so does its state. These changes of state are called state transitions. If all possible states are assumed to be known, a discrete-state system model is used; if this assumption is not made, a continuous-state system model is used. If the state transition times are assumed to be restricted to some multiple of a given time interval, a discrete-time system model is used. If it is assumed that state transitions can occur at any time, a continuous-time system model is used. Systems can be classified according to their state space and time parameter as the following:

discrete state and discrete time

discrete state and continuous time

continuous state and discrete time

continuous state and continuous time

For a discrete-state system, a state transition diagram (STD) may be drawn. This transition diagram is a directed graph. The nodes correspond to system states, and the directed arcs indicate allowable state transitions. Each arc has a label that identifies the distribution of the conditional probability that the system will go from the originating node to the destination node of that directed arc given the previous history of the system and that the system was initially at the originating node. The label used depends on the distribution. For example, the label could be the hazard rate for the exponential distribution, the scale and shape parameters for the Weibull distribution, or the mean and standard deviation for a general distribution.

If transitions are allowed from failed states to operational states, then the STD is an availability graph and $A(t)$ may be obtained from it. The term $R(t)$ may be obtained by specifically disallowing failed to working state transitions from the STD, thus making it a reliability graph.

A reliability graph of a triad is given in figure 1.6. In this model, it is assumed that the components have a perfect coverage of 1. The horizontal transitions represent fault arrivals. These transitions follow an exponential distribution. Consequently, λ represents the constant hazard rate. The coefficients of λ represent the number of working processors being actively used in the configuration. The vertical transition represents recovery from a fault. This recovery follows a general distribution. Consequently, μ and σ represent its mean and standard deviation. A competition exists between the two transitions that are leaving state 2. If the second fault wins the competition, then system failure occurs; however, if the removal of the first fault wins the competition, then the system reconfigures into a simplex (i.e., it only uses one of the two working components). Unless otherwise noted in the state descriptions, all working processors are being actively used in the configuration.

The information conveyed by the STD is often summarized in a square matrix called the state transition matrix (STM). The STM element in row i and column j is the label on the arc from state i to state j.

The terminology used in this paper to denote the various types of Markov models and the assumptions they are based on are defined below. The hierarchy of Markov models is illustrated in figure 1.7.

Figure 1.6. Reliability graph (state transition diagram) of a triad.

Figure 1.7. Hierarchy of Markov models.

Markov model: A stochastic process model whose future state depends only upon the present state and not upon the state history that led to its present state.

Homogeneous Markov model: A Markov model whose state transition probabilities are time independent. For the continuous-time homogeneous Markov model, this implies that the state transition times follow an exponential distribution. This type of model is discussed in Chung (1967) and Romanovsky (1970) and applied to computer systems in Makam and Avižienis (1982).

Semi-Markov model: A Markov model whose state transition probabilities depend upon the time spent in the present state, which is called the local time. For the continuous-time semi-Markov model, this implies that the state transition times do not follow an exponential distribution; they might follow a Weibull distribution or any other distribution. This type of model is discussed and applied to computer systems in White (1986).

Nonhomogeneous Markov model: A Markov model whose state transition probabilities depend upon the time since the system was first put into operation, which is called the global time. For the continuous-time nonhomogeneous Markov model, this implies that the state transition times do not follow an exponential distribution. Often these times are assumed to follow a Weibull distribution, but they can follow any other distribution. This type of model is discussed and applied to computer systems in Trivedi and Geist (1981).

The probability of being in a particular state for a discrete-state, continuous-time Markov model can be expressed with a differential equation. The set of simultaneous differential equations which describe these models are called the continuous-time Chapman-Kolmogorov equations. For homogeneous

Markov models, these equations can be solved using matrix or Laplace transformations.

If the state transition probabilities are time dependent, it may be quite difficult to obtain explicit solutions to the continuous-time Chapman-Kolmogorov equations. Obtaining the exact probability of reaching a state through a particular path of transitions requires the solution of a multiple integral, in which each integral represents the probability of making one of the transitions in the path. Often the integrals are approximated using numerical integration techniques (Stiffler, Bryant, and Guccione 1979). An alternative method is to approximate the continuous-time model with discrete-time equivalents (Siewiorek and Swarz 1992). The major difficulty with the second method is that many transition rates that are effectively zero in the continuous-time model assume small, but nonzero, probabilities in a discrete-time model.

1.2. Previous Work

Several programs exist, such as ARIES, SURF, CARE III, HARP, SURE, PAWS, STEM, and ASSURE, which use Markov models to evaluate the reliability and/or availability of systems that use standby redundancy or can be repaired and that are susceptible to hard, transient, or intermittent faults. All these programs can evaluate reliability. The ARIES, SURF, and HARP programs can also evaluate availability. Except for CARE III and ASSURE, they all have the state transition matrix as one of the system specification methods.

The ADVISER (Advanced Interactive Symbolic Evaluator of Reliability) program, described in Kini and Siewiorek (1982), automatically generates symbolic reliability functions for PMS structures. The program assumptions are that all the faults are permanent and stochastically independent, the PMS system has a perfect coverage, and the failed components are not repaired and returned to a nonfaulty state. The program's primary input is the interconnection graph of the PMS structure. Other program inputs describe the components of the PMS structure by their types, reliability functions, internal port connections, and ability to communicate with components of the same type. The program also takes as input the requirements for the system and its subsystems or clusters in the form of modified Boolean expressions.

The ARIES (Automated Reliability Interactive Estimation System) program, described in Makam, Avižienis, and Grusas (1982), is restricted to homogeneous Markov models. The system can be specified using a state transition matrix or as a series of independent subsystems each containing identical modules that either are active or serve as spares. The program uses a matrix transformation solution technique that assumes distinct eigenvalues for the state transition matrix.

Described in Landrault and Laprie (1978), the SURF program can solve semi-Markov models that use exponential distributions or nonexponential distributions that are related to the exponential (e.g., Gamma, Erlang, and others). The method of stages (Cox and Miller 1965) is used to produce a homogeneous Markov model. Matrix transformations are used to obtain time-independent values, such as MTTF and the limiting availability. The Laplace transform is used to obtain time-dependent values, such as availability and reliability.

The CARE III (Computer-Aided Reliability Estimation) program, described in Bavuso, Petersen, and Rose (1984), can evaluate the reliability of systems that use reconfiguration to tolerate component faults but that do not repair the faulty components. The program uses a behavioral decomposition/aggregation solution technique described in Trivedi and Geist (1981). This technique assumes the fault-occurrence behavior is composed of relatively infrequent (slow) events, and the fault-handling behavior is composed of relatively frequent (fast) events. The fault-handling behavior is separately analyzed using a fixed semi-Markov model that can use exponential and uniform distributions. The fault-occurrence behavior is analyzed using an aggregate nonhomogeneous Markov model that can use exponential and Weibull distributions. The fault-handling behavior is reflected by parameters in the aggregate nonhomogeneous Markov model. Numerical integration techniques are used to solve these Markov models. The fault-occurrence behavior is specified using extended fault trees, which are automatically converted to the nonhomogeneous Markov model. The fault-handling behavior is specified by providing the transition parameters of the fixed semi-Markov model.

For HARP (Hybrid Automated Reliability Predictor), described in Dugan et al. (1986) and Howell et al. (1990), the state transition probabilities can have exponential, uniform, Weibull, or general distributions. (A histogram must be provided for general distributions.) If the state transition matrix is given by the user, HARP can only evaluate the availability of systems with constant repair rates. The HARP program has several additional methods of specifying the fault-occurrence behavior (e.g., fault trees), all of which are automatically converted to a

nonhomogeneous Markov model. The fault-handling behavior can also be specified by providing the transition parameters of one of several models. The program uses the same behavioral decomposition/aggregation solution technique as CARE III, but the various models are solved in a hybrid fashion. Markov models are solved using numerical integration techniques, and extended stochastic Petri nets are solved by simulation.

The SURE (Semi-Markov Unreliability Range Evaluator) program, described in Butler (1992), evaluates the unreliability upper and lower bounds of semi-Markov models. It uses new mathematical theorems proved in White (1986) and Lee (1985). These theorems provide a technique for bounding the probability of traversing a specific path in the model within a specified time. By applying the theorems to every path of the model, the probability that the system reaches any death state can be determined within usually very close bounds. These theorems assume that slow (with respect to the mission time) exponential transitions describe the occurrence of faults, and fast transitions that follow a general distribution specified by its mean and standard deviation describe the recovery process. The program provides the option of pruning the model during its evaluation by conservatively assuming system failure once the probability of reaching a state falls below a specified or automatically selected prune level. Faults can be modeled as permanent, transient, or intermittent as long as there are no loops in the model which only have fast transitions. The only input method of the program is the state transition matrix.

Described in Butler and Stevenson (1988), the PAWS (Padé Approximation With Scaling) and STEM (Scaled Taylor Exponential Matrix) programs evaluate the unreliability of homogeneous Markov models. The input language for these two programs is essentially the same as for the SURE program. Although the numerical techniques used in these programs are not as fast as the SURE technique, they are suitable for loops with only fast transitions.

The ASSIST (Abstract Semi-Markov Specification Interface to the SURE Tool) program, which uses an abstract language for specifying Markov reliability models, is described in Butler (1986). The language has statements to specify the state space, by defining the state variables and their range; the start state, by the initial values of the state variables; the death states, by a Boolean expression of the state variables; and the state transitions, by a set of if-then rules that define, in terms of the state variables, the possible transitions, their rates, and their destination states. This language has been implemented in the ASSIST program to generate Markov reliability models in the SURE input language (Johnson 1986). The implementation provides three optional state space reduction techniques. The first technique is pruning the model during its generation by conservatively assuming system failure once a state satisfies a prune condition specified as a Boolean expression of the state variables (Johnson 1988). The second technique is trimming the model by conservatively altering states with outgoing recovery transitions (White and Palumbo 1990). The outgoing failure transitions of the altered states that do not go to death states are changed to go to a single trim state. The third technique combines pruning and trimming by changing all states that meet a prune condition to trim states. Each trim state has a single transition to a death state at some trim rate. The trim rate must be the sum of the failure rates of all remaining components.

The ASSURE program, described in Palumbo and Nicol (1990), translates an extension of the ASSIST language into C code, which is linked with SURE solution procedures and executed to generate and solve the model. This reduces the storage required because completely expanded states are discarded since the only state of consequence at any time is the state being expanded. The extended ASSIST language allows the use of user-defined C functions to specify the death states and the state transitions. This specification increases the size and complexity of the systems that can be practically modeled because it makes the model specification more compact.

The RMG (Reliability Model Generator) program is specified in Cohen and McCann (1990). As it is now implemented, LISP expressions are required to specify the system failure conditions whose probabilities are to be evaluated and each component's local reliability model (LRM) and function. An LRM is specified in terms of the component modes, the transitions between modes, and the characteristic (good, bad, or none) of the outputs in terms of the modes and the value or characteristic of the inputs. A graphical input is used to specify the interconnection graph of the PMS structure. It aggregates the LRM's to specify a Markov reliability model in the ASSIST language for the system failure conditions given.

Table 1.1 gives the primary inputs and outputs of the programs described in this subsection. None of these programs is able to generate a Markov model or its specification using a high-level SDL that is easy to learn and use.

Table 1.1. Summary of Previous Work

Program name	Primary inputs	Primary outputs
ADVISER	PMS structure	Symbolic reliability function
ARIES	Homogeneous Markov model	Reliability or availability estimate
SURF	Semi-Markov model	Reliability or availability estimate
CARE III	Fault tree and semi-Markov model parameters	Reliability estimate
HARP	Fault tree or nonhomogeneous Markov model	Reliability or availability estimate
SURE	Semi-Markov model	Reliability bounds
PAWS/STEM	Homogeneous Markov model	Reliability estimate
ASSIST	Semi-Markov model specification	Semi-Markov model
ASSURE	Semi-Markov model specification	Reliability bounds
RMG	LRM's, PMS structure, and system failure conditions	Semi-Markov model specification

1.3. Motivation

The goal of this research and development effort is to provide the computer architect a powerful and easy-to-use software tool that will assume the burden of an advanced reliability analysis that considers intermittent, transient, and permanent faults for computer systems of high complexity and sophistication. The PMS level of computer system description was selected because it is the highest level view of digital systems and therefore the easiest to specify and it is well known to computer architects. The Markov model technique was selected because it is powerful enough to accurately model most situations, it is widely used among reliability analysts, and these models can be evaluated by several programs that have been developed.

Previous efforts have been limited in one of three ways. Most efforts provided a computational aid once the preliminary system decomposition and reliability analysis had been manually achieved. Alternatively, computer systems of less complexity and sophistication were considered without transient and intermittent faults, or they did not provide a high-level SDL that is easy to learn and use.

1.4. Organization

The GUI is defined and illustrated in section 2. The problems involved in the automatic specification of Markov reliability models are identified and analyzed in section 3. Examples of GUI applications and their results are given in section 4. An analysis of this approach is presented in section 5. Conclusions are drawn in section 6. The algorithms used by the ARM program are shown in the appendix.

2. Graphical User Interface (GUI) Definition

The GUI is the first of four steps in the automated reliability modeling process proposed in this paper. The second step is the automated specification of the model in the ASSIST language. This step was implemented in the ARM program. The last two steps, the automated generation and evaluation of the model, have already been implemented. The third step has been implemented in the ASSIST program, and the fourth step has been implemented in the SURE, PAWS, and STEM programs.

In order of importance, the major goals of the GUI are defined below:

General: To allow current and future fault-tolerant techniques and system designs to be accommodated

Hierarchical: To allow systems and subsystems to be defined in terms of their subsystems and components, respectively

Compact: To allow subsystem classes to only be defined once with their component types as formal parameters

Subsystems are in the same *class* if they have the same hierarchy and requirements (e.g., triads that require two of their three components). *Subsystems* are of the same *type* if they are in the same class, are

Figure 2.1. System description hierarchy. (The asterisk (*) denotes parts that are always required.)

composed of the same component types, and have the same recovery parameters, if any (e.g., processor triads). *Components* are of the same *type* if they have the same function and parameters (e.g., processors). These categories of subsystems and components are summarized in table 2.1. For the sake of generality, the GUI does not predefine any category.

Table 2.1. Categories of Subsystems and Components

Category	Common attributes
Subsystem class	Hierarchy Requirements
Subsystem type	Subsystem class Component types Recovery parameters
Component type	Function Parameters

Each category is represented by an identifier that starts with a letter and can contain letters, underscores (_), and digits (e.g., a component type could be represented by p). A subsystem identifier can also end with a set of parentheses that enclose a list of parameters separated by commas. Formal parameters, which are identifiers that are not used to represent a category or anything else, are used in the identifier of a subsystem class (e.g., T(x)). Component types are used instead of the formal parameters in the identifier of a subsystem type (e.g., T(p)).

Type identifiers can be either (a) preceded by an integer greater than 1 to represent multiple elements of the same type (e.g., 2T(p)) or (b) followed by a period and a list of subranges and/or integer numbers in the range from 1 to the number of elements of that type, which are separated by commas to represent specific elements of the same type (e.g., T(p).1, 2), but not both (a) and (b). A subrange would be specified by two positive integer numbers separated by a dash, with the larger one on the right (e.g., p.1–3). Unless elements are assigned specific numbers, they are given the lowest positive numbers available (e.g., the components represented by 2p could be assigned the numbers 1 and 2).

The system's description is divided into requirements, architecture, and parameters. The requirements depend on the application of the system. How the system was designed determines the architecture. The technology used to implement the system components determines the parameter values (e.g., failure rates). The sources of the major GUI input categories are summarized in table 2.2. Figure 2.1 shows the hierarchy of the system description. The actual GUI inputs are the leaves of the tree shown in figure 2.1.

Table 2.2. Sources of Major GUI Input Categories

Major GUI input category	Source
Requirements	Application
Architecture	Design
Parameters	Implementation technology

The GUI starts by displaying the main window shown in figure 2.2. It contains text fields for entering the system name and the name of the current selection; the graphs, parameters, and model pull-down

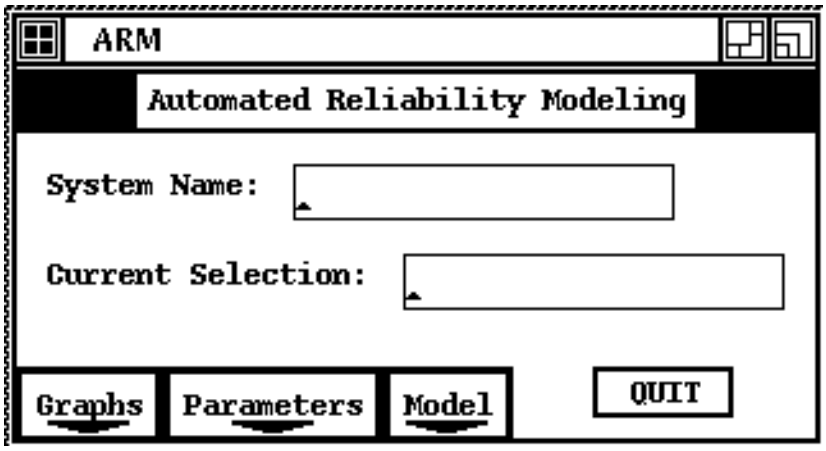

Figure 2.2. Main window.

menus; and a button to quit the GUI. The current selection, which is the initial name used by windows that describe a component type, subsystem type, or reconfiguration, changes automatically to the last name entered in the first text field of any such window, but it can also be changed manually.

The graphs menu, shown in figure 2.3, displays a window for editing the graphs described in subsection 2.1. The parameters menu, shown in figure 2.4, displays windows, with text fields and buttons for parameter specification, which are described in subsection 2.2. The model menu, shown in figure 2.5, executes the programs that specify, generate, and evaluate the Markov model, based on the system description given through the GUI. The ARM program will notify the user if the system description is incomplete (e.g., if the external structure has not been given) and not specify the model. Subsection 2.3 presents a summary of the GUI and recommendations on how to reduce the number of errors in the system description.

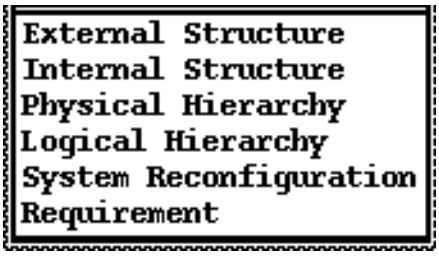

Figure 2.3. Graphs menu.

Figure 2.4. Parameters menu.

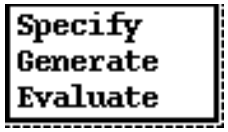

Figure 2.5. Model menu.

2.1. Graphs

The following subsections describe the graphs used for specifying the system's communication structure, hierarchy, reconfiguration capabilities, and requirements.

2.1.1. Structure

Graphs with unidirectional and bidirectional edges describe the system's external and internal communication structures. It is assumed that components which communicate and are critical (i.e., required for the system to be operational) must be able to continue communicating. If this assumption is not true, the result will be conservative. The main purpose of the communication structure description is to analyze which component failures will prevent communication between critical components and therefore cause system failure.

2.1.1.1. External. A system's external structure is defined as the communication interconnection of all its components. The external structure graph is required for all systems because it is also used to identify the system components, their types, and their connectivity equivalence classes (defined in subsection 3.2.1). In the external structure graph, the nodes represent one or more components of the same type. Unless specific numbers are assigned, the components represented by the same node are assigned a continuous range of numbers (e.g., the components represented by a node labeled $3p$ could be assigned the numbers 1 through 3). A unidirectional edge between two nodes indicates that all the components of the source node can communicate with all the components of the target node. A bidirectional edge between two nodes indicates that all the components of one node can communicate with all the components of the other node and vice versa.

A plus sign (+) at the end of a component identifier indicates that this is a self-talking component. A majority of components of the same type are passive, and they do not need to communicate. Examples of passive components are memories, buses, and input/output transducers. Self-talking components need to exchange information amongst one another. Examples of self-talking components are processors, direct-memory-access device controllers, and other "smart" controllers. If not specified, the default is for components to be passive and not communicate with their own type. This information is needed to prevent ARM from requiring communication paths between components of the same type that never exchange information. Not taking this behavior into account would lead to a pessimistic evaluation of the system reliability.

An asterisk (*) at the end of a component identifier indicates that every input port of this component is internally connected to all output ports of the component. Most buses have this internal structure. If not indicated in this way or as described in subsection 2.1.1.2, the default is for every port of a component to be disconnected from the other ports of the component.

The graph in figure 2.6 describes the external structure of a multiprocessor composed of six processors p, six memories m, six watchdog timers w, four transmit buses tb, four receive buses rb, and four watchdog buses wb. The processors and watchdog timers need to communicate with components of their own type. The processors communicate through the memory as described in subsection 2.1.1.2. The watchdog timers communicate through the watchdog bus. All the buses have the typical internal structure described above. This multiprocessor will be used as a running example throughout this section.

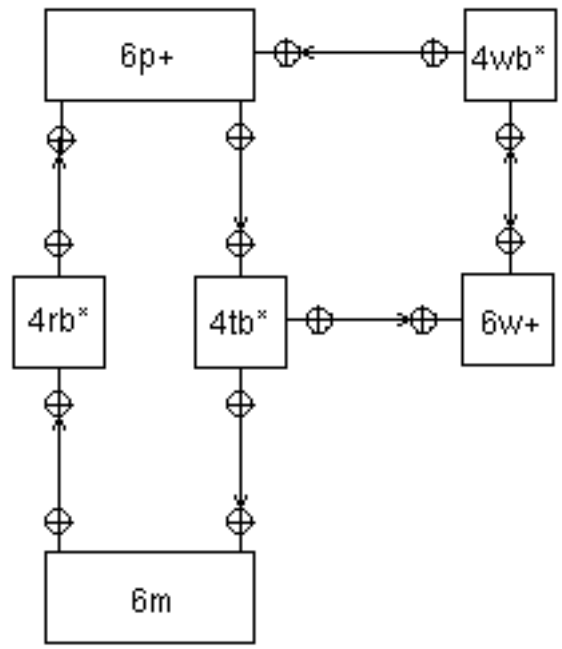

Figure 2.6. External structure of a multiprocessor.

2.1.1.2. Internal. A component's internal structure is defined as the communication interconnection of its ports. This internal structure of one or more components can be described by a graph inside a component with its external port connections labeled on the outside of the component. The absence of an edge between two ports indicates that they cannot communicate through this component.

The internal structure graph of a component is used to determine which of its neighbors can communicate through it. Two components are neighbors if they are interconnected in the external structure graph. If none of a component's neighbors can communicate through it, no internal structure needs to be specified because by default a component cannot be used for communication by its neighbors.

The internal structure of each of the six memories is described by the graph in figure 2.7. This structure indicates that the processors can communicate through the memory.

2.1.2. Hierarchy

A system can have physical and/or logical hierarchies that contain physical and logical subsystems, respectively. These hierarchies are different partial

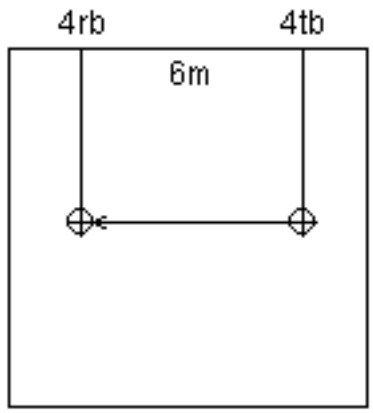

Figure 2.7. Internal structure of each of the six memory components.

views of the same system; therefore, a component of a physical subsystem may also be a component of a logical subsystem. The difference between a physical and a logical subsystem is in their ability to be reconfigured and in how their failure affects the system's operation, as explained in the next two subsections. If present, the system hierarchies show what subsystems are in the initial system configuration and define the composition of the subsystems that may be part of those hierarchies.

A group of components with its own set of requirements constitutes a subsystem. If a subsystem does not meet its requirements, then none of its components are able to perform their function. If a subset of the system components, but not all of them, depends on one or more components in the subset, the subset needs to be defined as a subsystem by giving its hierarchy and requirement graphs. The subsystem defined for the subset must be placed in either the appropriate system hierarchy graph (if it is part of the initial system configuration) or the destination node of a system reconfiguration graph (if it can be part of a future system configuration). The system physical or logical hierarchy graphs can only be given if there are physical or logical subsystems, respectively.

Redundant subsystems are composed of multiple components with the same function to increase their reliability or availability. Some of these redundant subsystems may be part of the initial system configuration, while others serve as alternatives for system reconfiguration (e.g., a quad subsystem that reconfigures into a triad).

A system hierarchy is described by nondirectional tree graphs. Root nodes (identified by a circle) represent the system or one of its subsystems. Other nodes (identified by a rectangle) represent one or more identical subsystems or components.

Unless they are assigned specific components, subsystems are assigned components with the lowest numbers available. For example, if there were six processors, numbered 1 through 6, and two processor triads, one triad would be assigned processors 1 through 3 and the other triad would be assigned processors 4 through 6.

2.1.2.1. Physical. Physical subsystems cannot be reconfigured. However, the failure of a physical subsystem does not preclude the system from operating, as long as the system requirements are met.

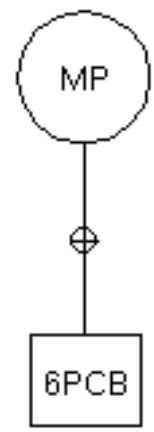

Figure 2.8. Physical hierarchy of the multiprocessor.

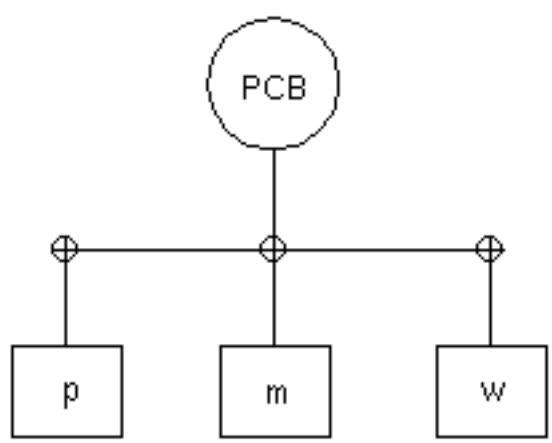

Figure 2.9. Physical hierarchy of the printed circuit board subsystem type.

Figures 2.8 and 2.9 describe the physical hierarchy of the multiprocessor (MP). Initially, the multiprocessor contains six printed circuit boards (PCB's), which belong to the same physical subsystem type.

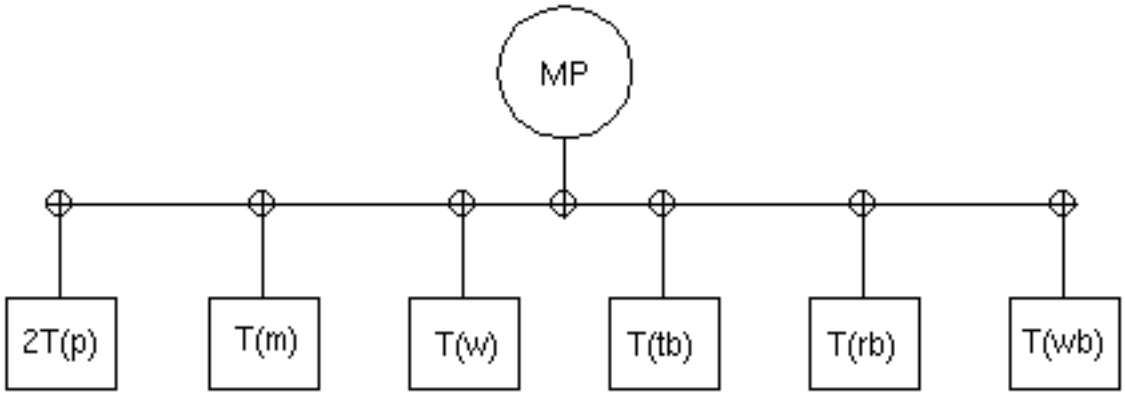

Figure 2.10. Logical hierarchy of the multiprocessor.

Each board contains a processor, memory, and a watchdog timer.

2.1.2.2. Logical. Logical subsystems can be reconfigured. Before component failures cause them to fail, they can recover by replacing the failed components with spares. If not enough spares are available, the system can degrade to a lesser number of subsystems or a less redundant subsystem. When a logical subsystem fails, the system also fails unless it can be reinitialized by a separate subsystem or component.

Figures 2.10 and 2.11 describe the logical hierarchy of the multiprocessor. Initially, the multiprocessor contains two processor triads, one memory triad, one watchdog triad, one transmit bus triad, one receive bus triad, and one watchdog bus triad. These triads are each composed of three components of the same type.

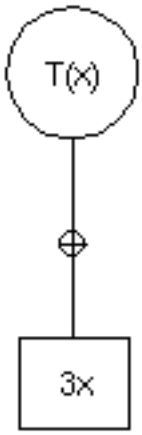

Figure 2.11. Logical hierarchy of the triad (T) subsystem class.

The ARM program will automatically determine what components are spares by comparing the external structure with the logical hierarchy; any extra instances of components in the external structure, beyond what is included in the logical hierarchy, will be assumed to be spares. Therefore, from figures 2.6 and 2.10, the spare components are assumed to be three memories, three watchdog timers, one transmit bus, one receive bus, and one watchdog bus.

2.1.3. System Reconfiguration

The future system configurations are described in terms of the reconfigurations allowed. A change in the system's configuration in response to some triggering event is defined as a reconfiguration. A reconfiguration occurs when the system is reinitialized because of a logical subsystem failure or when the system degrades to a lesser number of subsystems or a less redundant subsystem because no spares exist to replace a failed component. Also, the mission phase may change, thus causing the system to reconfigure. If the system is to be reinitialized because of a logical subsystem failure, only one reconfiguration must do so. To simplify the model specification, a single reconfiguration will only be allowed to degrade a subsystem to, at most, two less components. For example, one reconfiguration could take a subsystem from a quintuple to a triad, and a subsequent reconfiguration could take it to a simplex.

A reconfiguration is described in part by one or more unidirectional graphs. A source node represents one or more of the components or subsystems (physical or logical) which must be active before the reconfiguration. A destination node represents either the reinitialized system or one or more of the logical subsystems that will be active after the reconfiguration in place of the logical subsystems identified by its source node. Each edge is labeled with the name of a specification that will provide the triggering event and the rest of the reconfiguration parameters, described in subsection 2.2.5, which will complete the

description of the reconfiguration. The specification name can contain letters, underscores, and digits in any order.

Figure 2.12 describes the reinitialization of the multiprocessor by the watchdog triad. Figure 2.13 describes the degradation of the multiprocessor from two processor triads (PT's) to one. If the recovery rate of the remaining triad is specified as being greater than 0, the working processors in the deactivated triad are assumed to become spares.

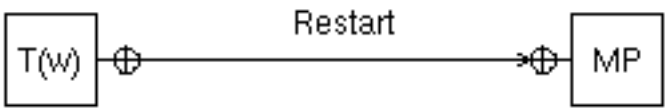

Figure 2.12. Reinitialization of the multiprocessor.

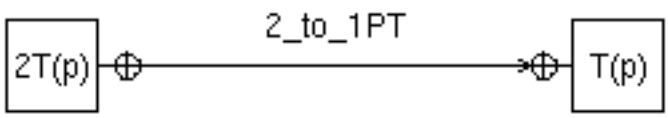

Figure 2.13. Degradation of the multiprocessor.

Currently, only the reconfigurations that degrade the system have been implemented. Therefore, at the present time, reconfiguration graphs are needed only for systems that have logical subsystems and can degrade to a lesser number of logical subsystems and/or to less redundant logical subsystems.

2.1.4. Requirement

The requirement of a system or subsystem is defined as the minimum set of subsystems and components needed. Performance levels can be used to identify the nondegraded mode and the various degraded modes of operation a system might have.

This requirement is described by one or more success trees. Root nodes (identified by a circle) represent the system, one of its subsystems, or a performance level. Other nodes (identified by a rectangle) represent one or more identical subsystems, a performance level, or one or more identical components. It is assumed that components in the system success tree are not in any logical subsystem. A success tree is required for all systems, subsystems, and performance levels.

Success trees and fault trees use the same notation, but they define the combination of events that will cause the system to succeed or fail, respectively. The advantages of success trees over fault trees are that (1) they are more intuitive for a computer engineer who is concerned with making the system work and not with how it can fail and that (2) a conservative reliability estimate is produced if some modes of operation are left out of the success tree, because system failure is assumed for those modes of operation, whereas an optimistic reliability estimate is produced if a failure mode is left out of a fault tree.

The graphs in figures 2.14 to 2.16 describe the system and subsystem requirements of the multiprocessor. This multiprocessor can operate at one of two performance levels. To achieve full performance (FP), both processor triads, the watchdog triad, and the memory triad must be operational. The requirements for degraded performance (DP) are the same except that only one processor triad is needed. Each printed circuit board requires that its memory be working for it to be operational. Subsystems of the triad class require two of their three components to operate.

2.2. Parameters

The following subsections describe the parameter specification windows. Any time unit may be used for the parameter values as long as it is the same one for all of them. The time unit used for ARM parameters throughout this paper is hours. The OK and CANCEL buttons in each window save and discard the parameter changes made, respectively. Selecting either button makes the window disappear.

The ARM program will assume that a transition which reconfigures components and/or subsystems in or out of the system describes sequential processes. For example, if n faults exist in one or more subsystems of the same type with recovery rate ρ, the rate at which one of the faulty components is replaced by a spare is assumed to be ρ not $n\rho$. If this assumption is not true, the result will be conservative. Typically, these transitions are fast, in which case this assumption being false would have little effect.

The SURE program requires slow transitions to follow an exponential distribution, but it allows fast transitions to follow a general distribution. Because transitions that reconfigure components and/or subsystems in or out of the system are typically fast, ARM allows them to follow a general distribution. However, in SURE, the transition probability must be given for each general transition competing with other fast transitions (Butler and White 1988). These probabilities must be given for each combination of one or more general transitions competing

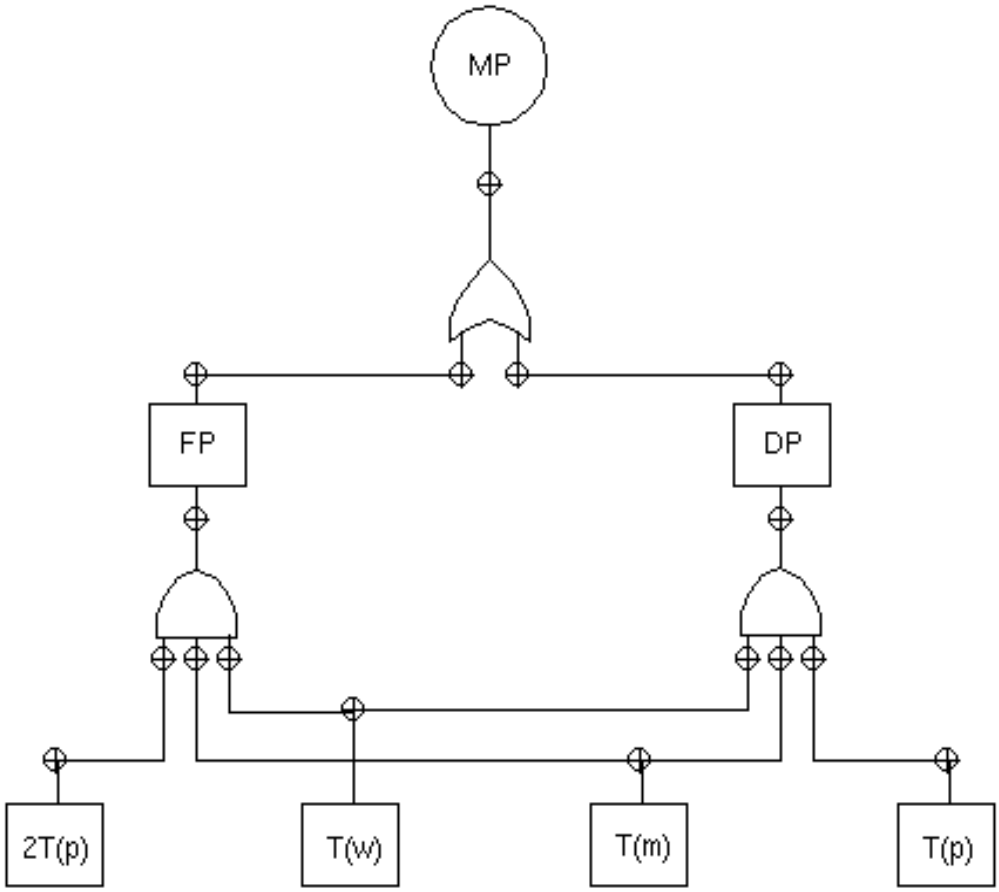

Figure 2.14. Requirements of the multiprocessor.

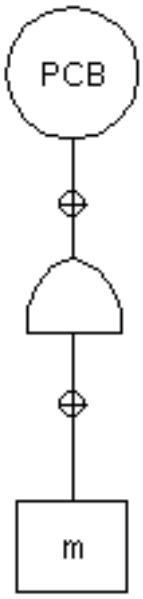

Figure 2.15. Requirement of the printed circuit board subsystem type.

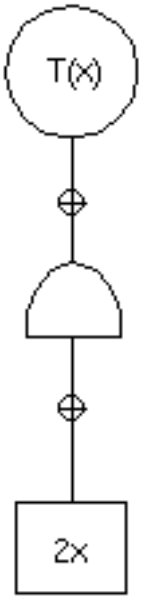

Figure 2.16. Requirements of the triad subsystem class.

with other fast transitions. The number of combinations of n competing general transitions taken two or more at a time is as follows:

$$\sum_{j=2}^{n} \frac{n!}{j!\,(n-j)!}$$

To simplify the system description and the model specification, ARM requests only one probability for each general transition. This is the occurrence probability when it is competing with any of the fast exponential rates at which transient faults disappear or intermittent faults become benign (subsection 2.2.1). Although these rates are fast, ARM does not allow them to follow a general distribution so that only one transition probability is needed for each general transition. This transition probability will be assumed to be the same for all competing fast exponential transitions. This assumption is not strictly true; however, it is often close enough in practice to be used to simplify the analysis.

Because ARM only asks for the probability of a general transition for the case when it is competing with the fast exponential rates at which transients disappear or intermittent faults become benign, these general transitions cannot compete with potentially general transitions. A potentially general transition is one that ARM allows to follow a general distribution. The only transitions that ARM allows to follow a general distribution are those that reconfigure components and/or subsystems in or out of the system.

However, all fast exponential transitions can compete. To determine which potentially general transitions should take precedence over others and which ones have the same precedence and therefore should compete, ARM requires that the user assign a positive integer priority to each potentially general transition. A value of 1 will be interpreted as the highest priority. Transitions that are assigned the same priority can compete if they follow exponential distributions, their triggering conditions are met, and the triggering conditions of higher priority transitions are not met.

Initially, numeric and selection parameters are assigned an appropriate default value. Probabilities default to 1 or 0. Priorities and coverage probabilities default to 1. Rates, means, standard deviations, and transition probabilities default to 0.

For each component, one of the failure rates described in subsection 2.2.1 must not be 0. Otherwise, ARM will notify the user and not specify the model. All other parameters may be left at their default values. Therefore, ARM does not prompt the user for any values.

Instead of values, all numeric parameters except priorities can also be given variable identifiers that start with a letter and can contain letters, underscores (_), and numbers. One of these variables can be given a range as described in subsection 2.2.7 if it is not used for the ASSIST trim rate described in subsection 1.2. If a variable is used for the trim rate, ASSIST will prompt for its value. The SURE, PAWS, or STEM programs will prompt for the value of all other variables without a range.

Numeric parameters are assumed to be independent of the system state. This assumption is not strictly true; however, it is often close enough in practice to be used to simplify the analysis.

2.2.1. Active Component

The active component parameters with example values are shown in figure 2.17. First is the name of the component type. Second is the arrival rate of permanent faults (0.00005 per hour or 2×10^4 hours between permanent failures). The next two are the arrival (0.0005 per hour) and disappearance rates (4000 per hour or 0.9 seconds to removal) of transient faults. If the arrival rate of transient faults is not 0, then the disappearance rate must have a value other than 0. The next three are the rates at which intermittent faults arrive, become benign, and become active again. If the arrival rate of intermittent faults is not 0, then the benign and active rates must have values other than 0. All six rates are assumed to describe concurrent processes. For example, if there are n working components of the same type with a permanent failure rate of λ, the rate at which one of them fails permanently is assumed to be $n\lambda$. If this assumption is not true, the result will be conservative.

The disappearance and benign rates are assumed to describe fast transitions if they are not 0. This is not a severe restriction because the behavior of a transient fault with a slow disappearance rate approximates that of a permanent fault and so does the behavior of an intermittent fault with a slow benign rate. These are the only fast exponential transitions that may compete with general transitions.

2.2.2. Spare Component

The spare component parameters with example values are shown in figure 2.18. First is the name of the component type. Second is the failure rate factor used to indicate which type of spare this is. It

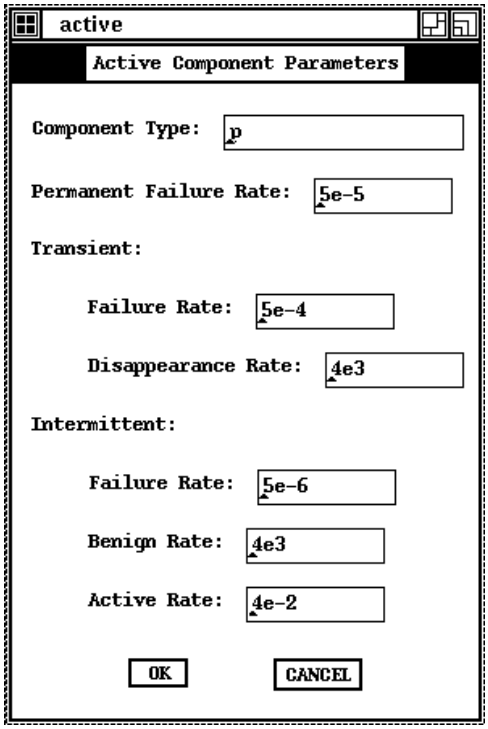

Figure 2.17. Active component parameters with example values.

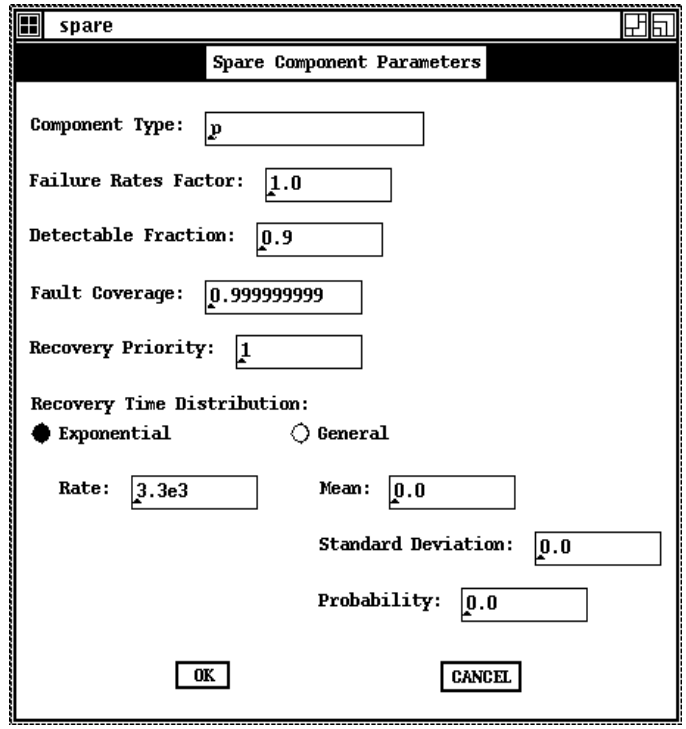

Figure 2.18. Spare component parameters with example values.

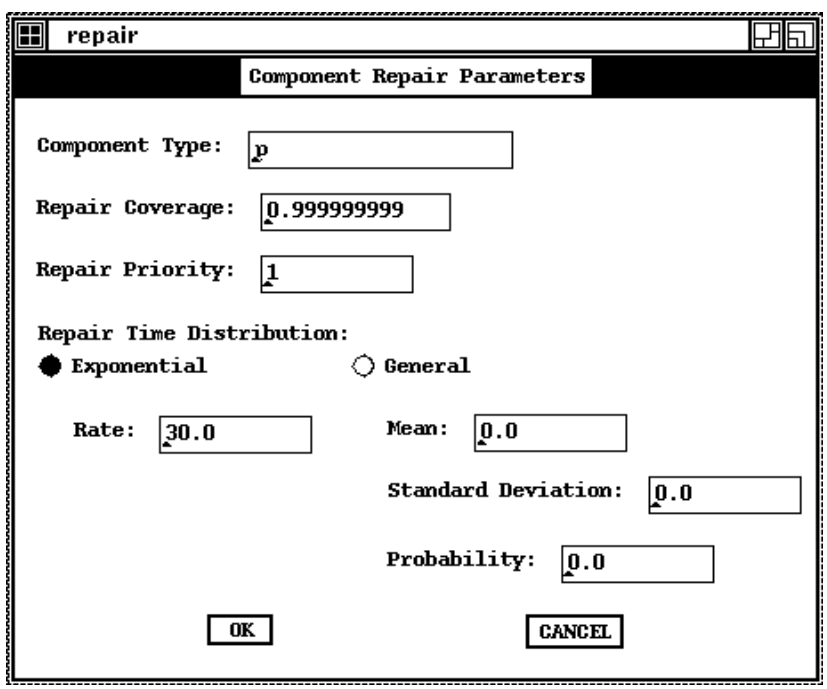

Figure 2.19. Component repair parameters with example values.

is 0 for cold, in the exclusive range of 0 through 1 for warm, and 1 for hot. This factor, which is the spare's fraction of the active component's failure rates, defaults to 1. Third is the fraction of faults that can be detected in a component of this type while it is a spare. This fraction defaults to 0. Fourth is the fault coverage of a spare component of this type. Fifth is the recovery priority. Sixth is the parameter that indicates whether the recovery time of detectable faults follows an exponential or general distribution. The next three parameters for the recovery time are (1) the rate, (2) the conditional mean (μ), and (3) the conditional standard deviation (σ). Parameter 1 is for an exponential distribution, and parameters 2 and 3 are for a general distribution given that the transition takes place.

The last parameter is the probability (P) that this transition will take place if it is competing with fast exponential transitions (whose rates add up to λ). This parameter defaults to 0. If the specification of the competing transitions is not consistent, SURE will not evaluate the model. To be consistent, these transitions must meet the following condition:

$$\mathrm{P} \leq \frac{2}{2\left(1+\lambda\mu\right)+\lambda^2\left(\mu^2+\sigma^2\right)}$$

This expression was derived from the conditions given in Butler and White (1988).

2.2.3. Component Repair

The component repair parameters with example values are shown in figure 2.19. First is the name of the component type. Second is the probability that the system can survive the reintegration of this type of component once it has been repaired. Third is the repair and reintegration priority. The remaining parameters specify the repair and reintegration time distribution.

2.2.4. Subsystem Recovery

The subsystem recovery parameters with example values are shown in figure 2.20. First is the name of the subsystem type. Second is the fault coverage for components in this type of subsystem. Third is the recovery priority. The remaining parameters specify the recovery time distribution.

Figure 2.21 illustrates the meaning of the parameters of active components and subsystem recoveries using a partial Markov model of a processor dual with m cold spares and repair. Except for states 0 and 6, all the states have additional transitions to additional states, none of which are shown. If the spares were warm or hot, state 0 would also have transitions representing the failure of the spares. Permanent, transient, or intermittent failures can take the system into states where a faulty component actively produces errors. From these states, either the system will detect these errors and succeed or fail in reconfiguring out the faulty component, the fault will disappear if it is a transient, or the fault will become benign if it is intermittent. If the faulty component is reconfigured out, it can be repaired and the system can succeed or fail in bringing it back into the configuration. The following notation applies to figure 2.21:

Parameter	Description
F	fault coverage
R	repair coverage
α	intermittent active rate
β	intermittent benign rate
δ	transient disappearance rate
λ	permanent failure rate
μ	repair rate
ρ	recovery rate
τ	transient failure rate
ω	intermittent failure rate

State	Description
0	no faults; m spares
1	1 permanent fault; m spares
2	1 transient fault; m spares
3	1 active intermittent fault; m spares
4	1 benign intermittent fault; m spares
5	no faults; $m-1$ spares
6	system failed

2.2.5. System Reconfiguration

The system reconfiguration parameters with example values are shown in figure 2.22. First is the specification name. The second parameter indicates whether this reconfiguration is triggered by a logical subsystem failure (default), a mission phase change, or a component in a logical subsystem failing without a spare. Third is the name of the component type

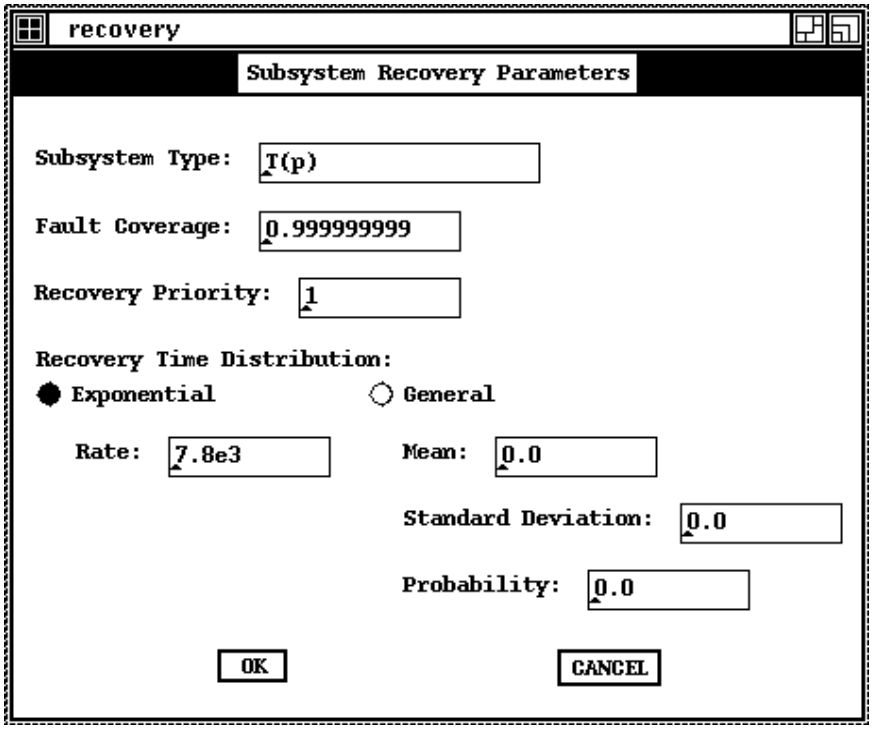

Figure 2.20. Subsystem recovery parameters with example values.

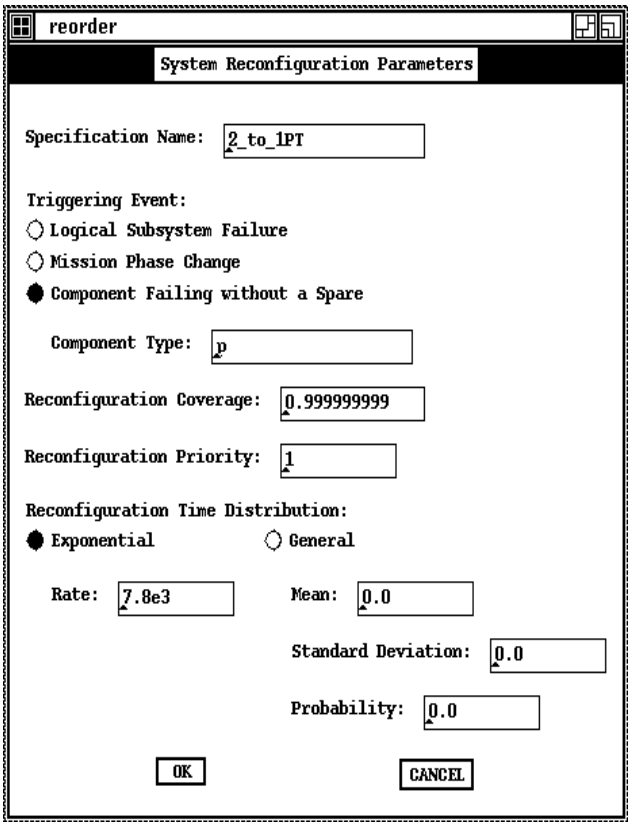

Figure 2.21. Partial Markov model of a processor dual with m cold spares and repair.

Figure 2.22. System reconfiguration parameters with example values.

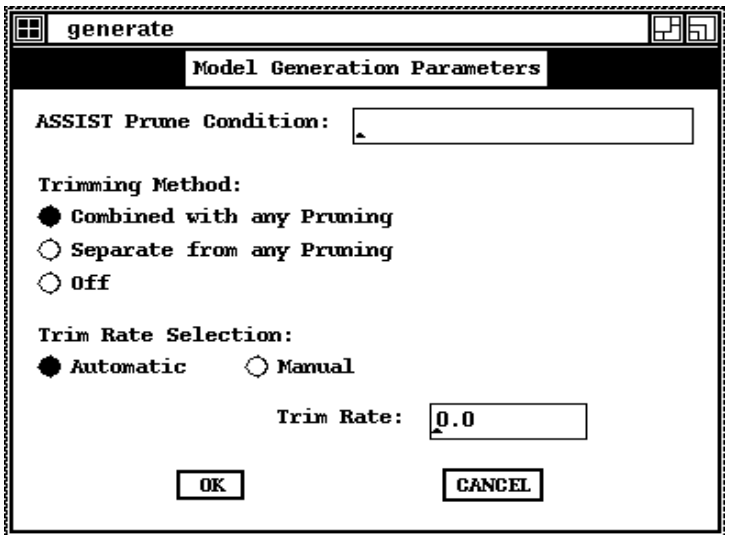

Figure 2.23. Model generation parameters with default values.

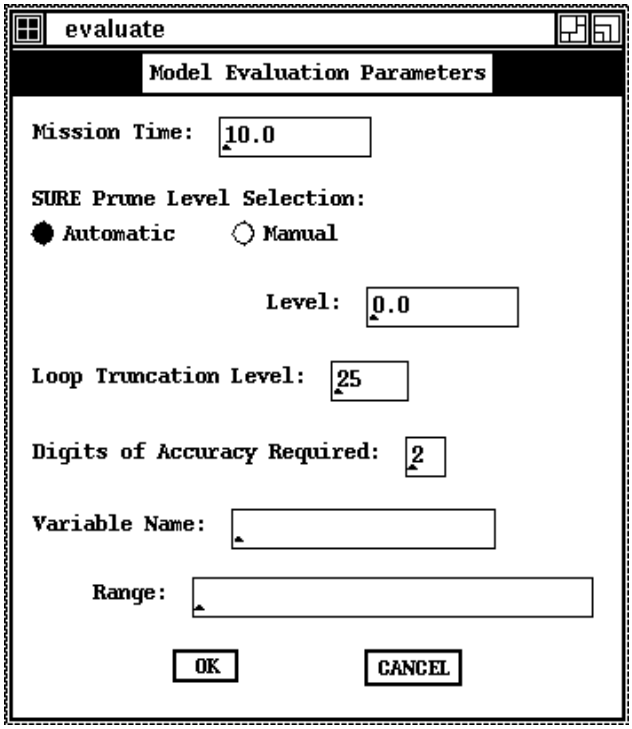

Figure 2.24. Model evaluation parameters with default values.

whose failure without a spare will trigger the reconfiguration. Fourth is the probability the system can survive the triggering event and successfully reconfigure. Fifth is the priority of the mission phase change, if any, and the consequent reconfiguration. This priority can be used to order multiple phase changes into a sequence. Because mission phase changes can occur at any moment, in most cases they should be given a priority equal to or less than other transitions; otherwise, lower priority transitions will have to wait for the higher priority phase changes to occur. If the reconfiguration is reinitializing the system because of a logical subsystem failure, it must have a priority of 1, and it must be the only transition to

have that priority. The remaining parameters specify the combined time distribution of the reconfiguration and the mission phase change, if any.

2.2.6. Model Generation

The window with default values shown in figure 2.23 allows a user familiar with the ASSIST program, which is described in subsection 1.2, to select which, if any, state space reduction techniques are to be used in generating the model and to specify any associated parameters. The first parameter is the optional ASSIST prune condition, which is specified as a Boolean expression of the total number of component failures (TNF) and/or the number of failures for a type of component (e.g., NF(p)). For example, the following expression would prune the model when there were two processor failures or three component failures of any type:

```
NF(p) >= 2 or TNF >= 3
```

The second parameter indicates the optional trimming method to be used. The last two parameters indicate whether the trim rate should be selected automatically or manually and the trim rate to be used when it is selected manually. If the trim rate is to be selected automatically, variables cannot be used for the arrival rates of faults.

2.2.7. Model Evaluation

The window with default values shown in figure 2.24 allows a user familiar with the SURE, PAWS, or STEM programs, which are described in subsection 1.2, to specify parameters used in the evaluation of the model. The first parameter is the mission time used for calculating the failure probability.

The next two parameters indicate whether the SURE prune level should be selected automatically or manually and the prune level to be used when it is selected manually. The ASSIST pruning affects which states are generated, whereas the SURE pruning affects which of the generated states are evaluated. If no SURE pruning is desired, the SURE prune level selection should be manual, and the prune level should be left at its default value of 0.

The fourth parameter is the maximum number of times the SURE program will go around a loop in the model before truncating its traversal. The fifth parameter is the number of digits of accuracy required. The SURE program will issue a warning if SURE pruning and truncation result in an upper bound on the failure probability that does not meet this accuracy requirement.

The last two parameters are used when the failure probability is to be calculated as a function of a previously defined variable. In that case, the name of the variable must be given along with its range. The range can be specified as follows:

$$l \text{ to } h \text{ add } i$$

where l and h are the low and high ends of the range and i is the increment added to vary the variable's value over that range. The range can also be specified as follows:

$$l \text{ to } h \text{ by } f$$

where f is the multiplication factor used to vary the variable's value over the range.

2.3. Summary and Recommendations

Although the graphs and parameters described in the previous two subsections can be given in any order, the number of errors in the system description may be reduced by following the same order of steps each time. The order suggested by the GUI pull-down menus is recommended because it should be easy to remember since the user sees it every time the pull-down menus are used and it has been designed to be natural and intuitive. The order and steps recommended for describing a system are as follows:

1. Identify all the system components and their interconnections in the external structure graph.
2. If the neighbors of a component use it to communicate, indicate so by ascribing to the component either the fully connected internal structure typical of buses by following its identifier in the external structure graph with an asterisk (*) or a specific internal structure graph.
3. If a subset of the system components, but not all of them, depends on one or more components in the subset, define the subset as a subsystem by giving its hierarchy and requirement graphs. If the subsystem defined for the subset is part of the initial system configuration, include it in the appropriate system hierarchy graph or else it is part of a future system configuration and must be included in a system reconfiguration graph as a destination node.
4. If physical and/or logical subsystems exist, give the system physical and/or logical hierarchy graphs.

5. If the system can degrade to a lesser number of logical subsystems and/or to less redundant logical subsystems, give the system reconfiguration graphs that describe these degradations.
6. Give the success tree of the system and each subsystem and performance level.
7. Use the active component parameters window to assign at least one nonzero failure rate to each component type.
8. If any of the system components at some point become potential spares (components that are not part of a logical subsystem) and the default values do not apply to some of them, use the spare component parameters window to assign the applicable values.
9. If some of the system component types can be repaired, use the component repair parameters window to define the process.
10. Use the subsystem recovery parameters window to define the process if there are logical subsystems and their faulty components can be replaced by spares.
11. If any system reconfiguration graphs have been given, use the system reconfiguration parameters window to specify each reconfiguration.
12. Use the model generation parameters window to provide the applicable values if the default values do not apply.
13. If the default values do not apply, use the model evaluation parameters window to provide the applicable values.

3. Automated Reliability Modeling (ARM) Implementation

The use of fixed-size arrays in the ARM program is limited to the storage of values or variable identifiers for parameters whose lengths are determined by the GUI implementation. (These lengths could be easily changed in later versions of ARM.) All other data are stored in dynamically allocated structures or arrays; therefore, the size of the problems which ARM can handle is limited only by the computer on which it is running. Dynamic allocation also permits ARM to more efficiently use storage by allocating only what is needed by the current problem.

The ARM program has been implemented using a C program with more than 8000 source lines of which more than 1000 were automatically generated, as described in subsection 3.1.2. The following subsections identify the problems involved in doing so and describe the steps taken to solve them.

3.1. Graphical User Interface

The GUI, which is based on a hierarchy of windows, has two types of windows. The implementation of the graphical editing windows is described in subsection 3.1.1. The implementation of the windows used to specify parameters and select actions is described in subsection 3.1.2.

3.1.1. Graphical Editing Windows

Graphical editing has been implemented using the schematic drawing editor Schem (Vlissides 1990). This editor provides three windows with which to create, view, and edit graphs.

The editor window, shown in figure 3.1, provides arrows for selecting what part of the graph is being viewed and at what scale. This window also provides five pull-down menus. The File menu can be used for reading and saving files; creating, adding, or removing tools for the creation of graphical components called elements; and creating a textual representation of a graph that identifies its elements and their interconnections called a netlist. The Edit menu can be used to copy, paste, or delete graphical components and to assign the graph a name that identifies it in the netlist (by using the command `Info`). The Structure menu can be used to group components and then copy or move them as a single entity. The Align menu can be used to center graphical components or align them in relation to one another. The View menu can be used to display the drawing tools window and to center the whole graph. The two other windows are displayed automatically when Schem is executed.

The tools window, shown in figure 3.2, provides commands for selecting and moving graphical components, connecting the nodes of graphical elements, and assigning to the elements and their nodes the names that identify them in the netlist. This window also displays the tools currently available for creating graphical elements. The node, wire, bulb, and switch tools (the top four shown in the window) are the only ones provided by Schem. All other elements have to be created by the Schem user. Figure 3.2 shows 7 of the 28 tools created for the ARM user.

The drawing tools window, shown in figure 3.3, provides five pull-down menus, eight drawing tools, and four commands. The pull-down menus can be used to select the current font, brush, pattern, and foreground and background colors. The drawing

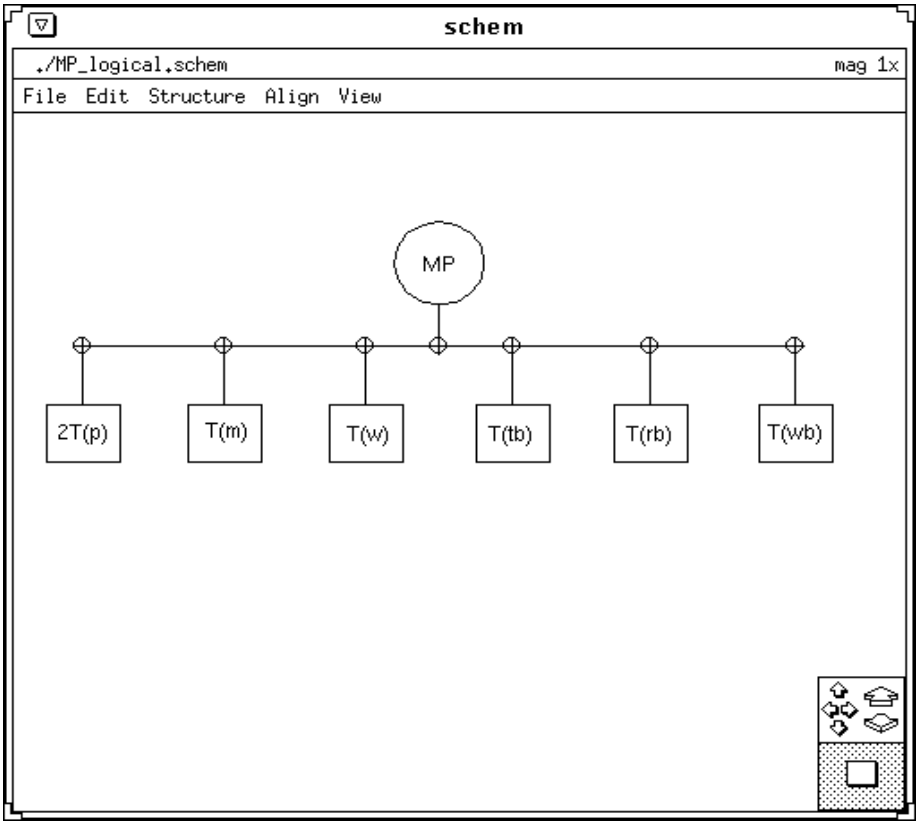

Figure 3.1. Graphical editor window.

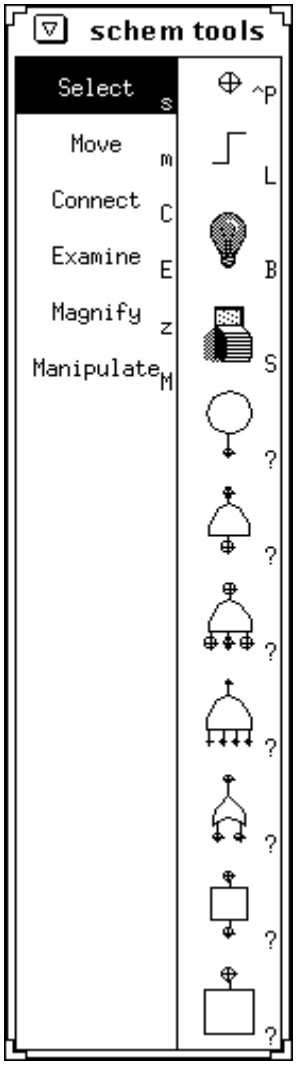

Figure 3.2. Tools window.

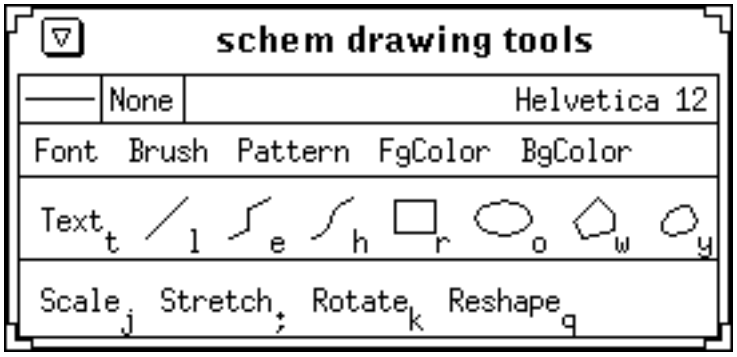

Figure 3.3. Drawing tools window.

tools can be used to create a graphical component that can be text or a line, rectangle, or circle. The commands can be used to scale, stretch, rotate, or reshape a graphical component.

The netlist file of each graph must be generated for ARM to process the graph. These files must be of the *net* type. Their file names must be composed of an identifier, an underscore (_), and a suffix. The identifier must correspond to the name of the system, subsystem, component(s), or performance level being described, except that underscores are substituted for parentheses and periods. The suffix describes the class of graph in the file. The suffixes that must be used for each class of graph are shown in table 3.1. For example, the file specifications for the hierarchies of logical subsystem class T(x) and physical subsystem CR_A.1 are T_x_logical.net and CR_A_1_physical.net.

Table 3.1. File Name Suffixes for Each Class of Graph

Graph class	File name suffix
External structure	External
Internal structure	Internal
Physical hierarchy	Physical
Logical hierarchy	Logical
System reconfiguration	Reorder
Requirement	Require

The names that identify the graph and its elements in the netlist must be exactly the same as the name of the system, subsystem, component(s), or performance level being described. The only exception to this requirement is that logical gate elements and the root elements (identified by a circle) of tree graphs must retain their original netlist names of *and*, *or*, plus *root*. The names that identify graph element nodes in the netlist file must be composed of a function identifier and a tag. The function identifier has to be *input*, *output*, or *bidirectional*. For internal structure graphs, the tag is composed of an underscore followed by the name of the element that the node is connected to in the external structure graph. For system reconfiguration graphs, the tag is composed of an underscore followed by the name of the specification.

For all other graphs, a tag is needed only if an element has more than one node with the same function. If present, this tag must be composed of an underscore followed by an identifier that is unique for nodes with the same function and in the same element. For example, two nodes in the same element could be named *input_1* and *input_2*.

3.1.2. Parameter Specification and Action Selection Windows

The windows for entering parameters and selecting actions have been implemented using the TAE Plus user interface development tool for building X window-based applications (Szczur 1990). These windows were defined using the TAE Plus workbench. The workbench generates the more than 1000 source lines that display these windows, including an event handler for each item in a window. Function calls have been added to each event handler to check the validity of the input and to store it if so indicated by the user. The inputs from each window are stored in a separate file.

These parameter files are of the *par* type. Their file names are composed of an identifier, an underscore (_), and a suffix. The identifier corresponds to the name of the system, subsystem type, component type, or reconfiguration specification being described, except that underscores are substituted for parentheses and periods. The suffix describes the class of parameters in the file. The suffixes that are used for each class of parameters are shown in table 3.2.

Table 3.2. File Name Suffixes for Each Class of Parameter

Parameter class	File name suffix
Active component	Active
Spare component	Spare
Component repair	Repair
Subsystem recovery	Recovery
System reconfiguration	Reorder
Model generation	Generate
Model evaluation	Evaluate

3.2. Reading and Processing the System Description

When indicated by the user, ARM uses the current system name to read and process the system description files and check them for completeness and consistency. If at any point these files are found to be incomplete or inconsistent, the user is notified, and

the reading and processing is aborted. The ARM program takes the following steps to read, process, and check the system description files:

1. Reads the external structure
2. Reads the component parameters
3. Determines the internal structure of the components
4. Detects symmetries in the external structure
5. Reads the system physical hierarchy
6. Determines the hierarchy of physical subsystems
7. Reads the system logical hierarchy
8. Determines the hierarchy of logical subsystems in the initial configuration
9. Reads the reconfiguration graphs and parameters
10. Determines the hierarchy of logical subsystems not in the initial configuration
11. Reads the logical subsystem parameters
12. Reads the requirements
13. Reads the modeling parameters

The program identifies the system component types in the external structure in step 2 and reads their active parameter files. It also reads the spare and repair parameter files if they are present; otherwise, it uses default values.

In step 3, the program reads the internal structure files of each vertex in the external structure if they are present; otherwise, the default internal structure indicated in the external structure is used. It also records any numbers assigned to specific components in the external structure.

The program divides each component type into classes in step 4; these classes are equivalent in terms of their connections in the external structure, as described in subsection 3.2.1. The program also assigns numbers to those components that did not have any in the external structure.

In step 5, the program reads the system physical hierarchy file if it is present; if so, it then identifies the physical subsystems, their types, and their classes.

In step 6, the program determines the hierarchy of any physical subsystems as described in subsection 3.2.2. It then assigns specific components to the physical subsystems where necessary. Based on the components assigned, the program then identifies the component equivalence classes in each physical subsystem.

In step 7, the program reads the system logical hierarchy file if it is present; if so, it then identifies the logical subsystems in the initial configuration, their types, and their classes.

Step 8 is the same as step 6 except for the logical subsystems in the initial configuration.

In step 9, the program reads the system reconfiguration file if it is present; if so, it then finds a reconfiguration graph whose source vertex represents a previously identified component or subsystem, stores the reconfiguration information, and identifies any new logical subsystem type and class in the destination vertex. This step assumes that the new subsystem type can contain any component equivalence classes in the old subsystem type. When there are no more reconfiguration graphs to process, it reads the parameter files that specify the reconfigurations.

In step 10, the program reads the hierarchy file of any logical subsystem class that is not in the initial configuration. It then determines the hierarchy of any logical subsystem that is not in the initial configuration from the hierarchy of its subsystem class and the component type arguments, if any, of its subsystem type.

The program, in step 11, reads the recovery parameter files of the logical subsystem types if present; otherwise, it uses default values.

In step 12, the program reads the system requirements file. If there are any subsystem classes or performance levels whose requirements are not defined in the system success tree, it reads their requirements files. From the requirements of its subsystem class and the component type arguments, if any, of its subsystem type, the program then determines the conditions under which each subsystem will fail and any component dependencies that exist.

In step 13, the program reads the model generation and evaluation parameter files if present; otherwise, default values are used.

3.2.1. Detection of Symmetry in the External Structure Graph

Each component type is divided into equivalence classes because ARM assumes that when a component in a logical subsystem fails, it can only be replaced by an equivalent component. Substructures in the external structure graph G are considered symmetric if they are isomorphic and the corresponding vertices of the two graphs have identical component-type labels. Symmetrical substructures

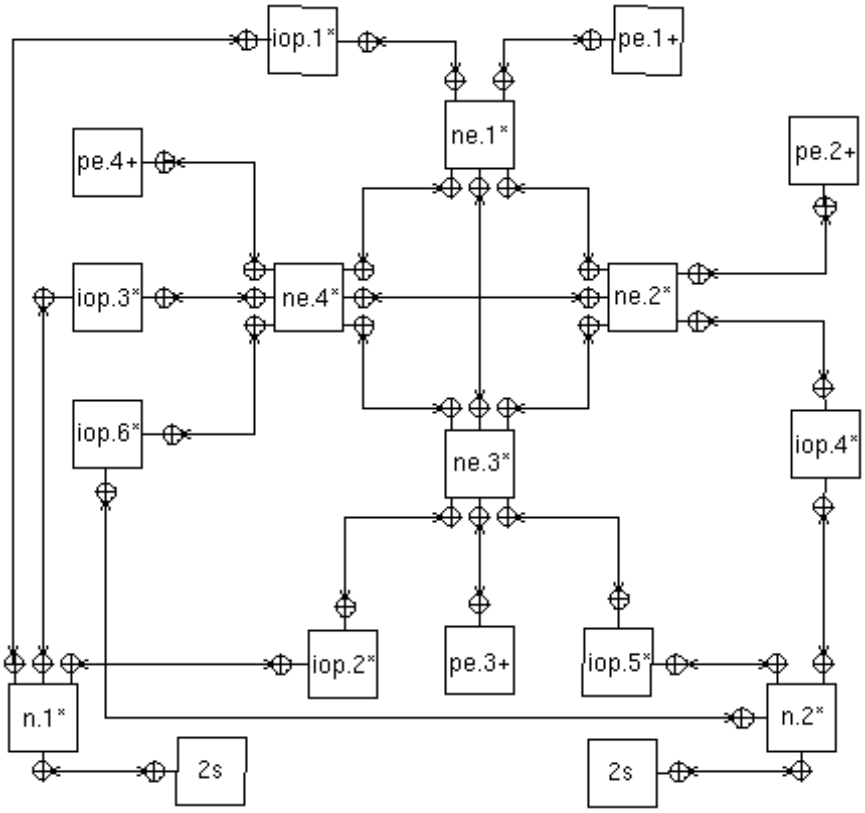

Figure 3.4. External structure with symmetry.

are assumed to be identical in function and reliability.

Subsection A1 shows the algorithm used by ARM to detect symmetries in the external structure graph G. This algorithm has been derived from the one used in ADVISER (Kini and Siewiorek 1982) for nondirected graphs with a single component per vertex because the ADVISER algorithm is mature, well documented, and simple. However, the ARM algorithm applies to directed graphs that can have multiple components per vertex. This algorithm is based on the component-type labels and the degree of the vertices in the graph. The degree of a vertex is the number of neighbor vertices which it has of each type. Two vertices are neighbors if they are interconnected. Neighbors can be of the input, output, or bidirectional types.

The ARM algorithm requires three steps to partition the vertex set of a labeled graph into equivalence classes whose vertices are symmetrical. In the first step, the partition is based on the component-type label of each vertex. For the second step, the partition is based on the degree of each vertex. In the third step, partitioning is attempted based on the number of neighbor types each vertex has in each equivalence class.

The last step must be repeated until there are no more changes in the equivalence classes. The reason for this is that each partition changes the number of neighbors in each equivalence class; therefore, other partitions may become necessary. In the worst case, this repetition will stop when each equivalence class has a single element.

In the first step, the example external structure in figure 3.4 is divided into five equivalence classes—one for each component type. Components of the *ne* type are split into two equivalence classes—one of degree 5 and another of degree 6. The first time the third step is taken, the program splits the *pe* and the *iop* component types into two equivalence classes—one connected to the *ne* equivalence class of degree 5 and the other connected to the *ne* equivalence class of degree 6. The second time that the third step is taken, the eight equivalence classes are left unchanged.

Each class is related to other classes in a connectivity sense because the vertices in the class are symmetrically connected to the vertices in other

Figure 3.5. Equivalence class graph.

classes. These equivalence classes and their connectivity relationships may be viewed as defining another graph G′. The vertices of G′ correspond uniquely to the equivalence classes in G. Unlike the basic directed graph without self-loops, which was taken to be the model for G, G′ may have vertices that have self-loops. A self-loop occurs when the vertices in the same equivalence class are connected to each other in some symmetric fashion, thus making the equivalence class its own neighbor. Also, the number of links or connection density between two vertices of G′ can be greater than 1. This would be the result of a case in which multiple vertices in the same equivalence class are connected to one or more vertices in another equivalence class.

Figure 3.5 shows the equivalence class graph corresponding to the external structure in figure 3.4. Vertex $2ne[1]$ corresponds to equivalence class 1 of component type ne, which has two elements in that class. The 2/1 on the edge between vertex $2n[1]$ and vertex $4s[1]$ indicates that each element of equivalence class $2n[1]$ is connected to two elements of equivalence class $4s[1]$ and each element of equivalence class $4s[1]$ is connected to one element of equivalence class $2n[1]$.

3.2.2. Determining the Subsystem Hierarchies

The hierarchies of any logical subsystems in the initial configuration and of any physical subsystems are determined using the algorithm in subsection A2. This algorithm tries to obtain the hierarchy of each subsystem class from the system hierarchy or a separate file. For those subsystem classes whose hierarchy it cannot obtain, the algorithm goes through each subsystem in the class and reads its individual subsystem hierarchy file that assigns specific components to the subsystem. For those subsystem classes whose hierarchy it did obtain, this algorithm then goes through each subsystem in the class and determines its hierarchy based on the hierarchy of its subsystem class and the component-type arguments, if any, of its subsystem type.

3.3. Specifying the System Reliability Model

If the system description files are complete and consistent, ARM then specifies the system reliability model in the ASSIST language. A reliability model specification in the ASSIST language must define the following:

1. Any constants
2. State space variables
3. Start state
4. Any functions
5. Death conditions
6. Any pruning conditions
7. State transitions

The following subsections describe the steps that ARM takes to make these definitions.

3.3.1. Constants

The ARM parameters that were given numeric values will be defined as ASSIST constants. An ASSIST input statement will be specified if a variable identifier is given for the trim rate because ASSIST needs that value to generate the model. The SURE input statements will be specified for any other ARM parameters that were given variable identifiers without a range. If one of the variable identifiers was given a range, a SURE variable definition statement will be specified for it. Only one input or variable definition statement will be specified for each variable identifier even if it is used for more than one ARM parameter.

The ARM parameters will not be placed in ASSIST constant arrays because if a variable identifier is given for one of the parameters, then one of the array elements would be undefined. Although the undefined array element could be given some initial value like 0 and then redefined when SURE evaluates the model, each redefinition would cause a SURE warning message. Longer model specification files are a consequence of not using arrays for constants, but longer files are justified to avoid these warnings and to provide the convenience of using variable identifiers.

In addition to the ARM parameters, scalar and array constants used in later definitions have to be specified. These constants define the system components and the logical subsystems.

The system component constants and their definitions are as follows:

NCT: Number of component types

LCTS: Largest component-type size

NEC: Number of equivalence classes

LECS: Largest equivalence class size

CT: Component type of each equivalence class

The size of a component type or equivalence class is the number of components in the type or class. All of the above are integer scalars except CT, which is an integer array indexed by the equivalence class number.

The logical subsystem constants and their definitions are as follows:

NLT: Number of logical subsystem types

LLTS: Largest logical subsystem-type size

NL: Number of logical subsystems

LNEL: Largest number of equivalence classes in a logical subsystem

LESL: Largest number of equivalent components in a logical subsystem

LT: Type of each logical subsystem

NEL: Number of equivalence classes in each logical subsystem

EC: Equivalence class number of a subset of equivalent components in a logical subsystem

The size of a logical subsystem type is the number of subsystems in the type. All of the above are integer scalars except LT, NEL, and EC. The constants LT and NEL are integer arrays indexed by the logical subsystem number. The constant EC is an integer array whose first index is the logical subsystem number and whose second index is the equivalent subset number within the subsystem.

All the parameters involved in a potentially general transition are defined as constants except the priority that is used by ARM in specifying these transitions as described in subsection 3.3.5. The order in which all of the constants are defined is as follows:

1. Trimming model generation parameters
2. Model evaluation parameters
3. System component constants
4. Active component parameters
5. Spare component parameters
6. Component repair parameters
7. Logical subsystem recovery parameters
8. Logical subsystem constants
9. Reconfiguration parameters

3.3.2. State Space Variables and the Start State

The system components will be divided into those that belong to a logical subsystem and those that do not. Although those that do not will be referred to as spares, because they often are, they are not necessarily spares. The components of a logical subsystem will be divided into subsets of components in the same equivalence class.

The state space variables and their definitions are as follows:

CF: Coverage failure

LTC: Logical subsystem-type count

Table 3.3. State Space Variable Functions

Function	Definition
NB(i,j) = P[i,j] + T[i,j] + A[i,j]	Number of nonbenign failures in subset
WB(i,j) = C[i,j] − NB(i,j)	Number of working or benign components in subset
W(i,j) = WB(i,j) − B[i,j]	Number of working components in subset
PL(i) = sum(P[i,*])	Number of permanent failures in logical subsystem
TL(i) = sum(T[i,*])	Number of transient failures in logical subsystem
AL(i) = sum(A[i,*])	Number of active failures in logical subsystem
NBL(i) = PL(i) + TL(i) + AL(i)	Number of nonbenign failures in logical subsystem
TNF = sum(NF)	Total number of failure transitions
NS(k) = WS[k] + PS[k] + TS[k] + AS[k] + BS[k]	Number of equivalent spares
NBS(k) = PS[k] + TS[k] + AS[k]	Number of nonbenign failures in equivalent spares
FCE = sum(T, A, B, TS, AS, BS)	Number of fast exponentials competing with any general transitions

LO: Logical subsystem operational

C: Components in a logical subsystem subset

P: Permanent failures in a logical subsystem subset

T: Transient failures in a logical subsystem subset

A: Active intermittent failures in a logical subsystem subset

B: Benign intermittent failures in a logical subsystem subset

NF: Number of failure transitions per component type

NR: Number of components reconfigured out per equivalence class

WS: Working equivalent spares

PS: Permanent failures in equivalent spares

TS: Transient failures in equivalent spares

AS: Active intermittent failures in equivalent spares

BS: Benign intermittent failures in equivalent spares

Except for CF, which is a Boolean scalar, all others are integer arrays. The LTC variable is indexed by the logical subsystem type number, and it indicates the number of operational logical subsystems for each type. The LO variable is indexed by the logical subsystem number. The first index of C, P, T, A, and B is the logical subsystem number, and the second index is the equivalent subset number within the subsystem. The NF variable is indexed by the component type number. The variables NR, WS, PS, TS, AS, and BS are indexed by the equivalence class number.

The ARM program assumes that the only logical subsystems that are operational initially are those present in the system logical hierarchy; it also assumes a start state where no failures have yet occurred.

3.3.3. Functions and Final State Conditions

The state space variable functions and their definitions are given in table 3.3. The ASSIST function sum adds all the elements in one or more dimensions of one or more arrays.

If a pruning condition was given, it is implemented using the state variable NF and the function TNF. For example, the pruning condition

```
NF(p) >= 2 or TNF >= 3
```

can be implemented by the following statement:

```
pruneif NF[1] >= 2 or TNF >= 3;
```

The Boolean expression represented by the system requirements success tree is logically negated and used as a death state condition. For example,

the MP system requirements can be implemented by the following statement:

```
deathif not ((LTC[1] >= 2 and LTC[2] >= 1
  and LTC[3] >= 1) or (LTC[1] >= 1
  and LTC[2] >= 1 and LTC[3] >= 1));
```

The conditions under which a logical subsystem fails are used to trigger a restart reconfiguration, if there is one; otherwise, they are used as death state conditions. For example, the requirements of the logical subsystem T(m) can be implemented by the following statement:

```
deathif WB(2,1) < 2;
```

Other death state conditions include situations in which the components in the system requirements success tree can no longer communicate. The condition for being in the death state that corresponds to a coverage failure is for CF to be true.

3.3.4. Failure Transitions

The ARM program analyzes the system description to specify the failure transitions. For each failure transition, the following must be specified: the conditions under which the transition could take place, its destination state, and its transition rate. Failure transitions are only specified for those fault types with nonzero rates.

Several options exist for specifying failure transitions for components that depend on a component that has a soft fault:

1. The components could be put in the same state as the component they depend on; however, they could become benign at their own rate even before the component they depend on.
2. They could be declared to have a permanent fault to avoid the problem described in option 1. To implement option 2, these components would have to be (a) tracked to determine if they were reconfigured out so they could be declared to be working when the component they depend on becomes benign or (b) left failed.

The ARM program implements option 2(b) because it is conservative and it avoids the inconsistency problems with option 1 and the implementation difficulties with option 2(a).

3.3.4.1. Logical subsystem component failures. The condition for fault arrival in a logical subsystem is that working components exist. The destination state is one in which the number of failures in the subsystem and in the component's type (NF) has been increased by 1. Also any components that depended on the failed component are marked as failed. The fault arrival rate is obtained by multiplying the number of working equivalent components in the subsystem by their failure rate. For example, the arrival of a permanent fault in equivalent subset 1 of logical subsystem 2 can be described by the following statement:

```
if W(2,1) > 0 tranto NF[CT[EC[2,1]]]++,
  P[2,1]++ by W(2,1) * PFR_^CT[EC[2,1]];
```

where PFR stands for the permanent failure rate. A caret (^) is used to concatenate a string and a value to form a previously defined identifier.

The condition for the disappearance of a transient fault exists when there are components with transient faults. The destination state is one in which the number of transient failures in the subsystem (T) has been decreased by 1. The transition rate is obtained by multiplying the number of equivalent components with transient faults in the subsystem (T) by their disappearance rate. For example, the disappearance of a transient fault in equivalent subset 1 of logical subsystem 2 can be described by the following statement:

```
if T[2,1] > 0 tranto T[2,1]--
  by fast T[2,1] * TDR_^CT[EC[2,1]];
```

where TDR stands for the transient disappearance rate.

The condition for an intermittent fault to go from active to benign is that components with active intermittent faults exist. The destination state is one in which the number of benign components (B) has been increased by 1, and the number of components with active intermittent faults (A) has been decreased by 1. The transition rate is obtained by multiplying the number of equivalent components with active intermittent faults in the subsystem (A) by their benign rate. For example, an intermittent fault that goes from active to benign in equivalent subset 1 of logical subsystem 2 can be described by the following statement:

```
if A[2,1] > 0 tranto B[2,1]++, A[2,1]--
  by fast A[2,1] * IBR_^CT[EC[2,1]];
```

where IBR stands for the intermittent benign rate.

The condition for an intermittent fault to go from benign to active is that components with benign intermittent faults exist. The destination state is one in which the number of components with active intermittent faults (A) has been increased by 1 and the number of components with benign intermittent faults (B) has been decreased by 1. The transition

rate is obtained by multiplying the number of equivalent components with benign intermittent faults in the subsystem (B) by their active rate. For example, an intermittent fault going from benign to active in equivalent subset 1 of logical subsystem 2 can be described by the following statement:

```
if B[2,1] > 0 tranto A[2,1]++, B[2,1]--
  by B[2,1] * IAR_^CT[EC[2,1]];
```

where IAR stands for the intermittent active rate.

3.3.4.2. Spare failures. Spare failure transitions are specified only for those component types in which the failure rates factor is nonzero. The condition for fault arrival is that working spares exist. The destination state is one in which the number of working spares (WS) has been decreased by 1, both the number of failed spares and the number of failed components of the spare's type (NF) have been increased by 1, and any components that depended on the failed component are marked as failed. The fault arrival rate is obtained by multiplying the number of working spares in the equivalence class (WS) by their failure rate and failure rates factor. For example, the arrival of a permanent fault in a spare in equivalence class 1 can be described by the following statement:

```
if WS[1] > 0
  tranto WS[1]--, NF[CT[1]]++, PS[1]++
  by WS[1] * PFR_^CT[1] * FRF_^CT[1];
```

where FRF stands for the failure rates factor.

The condition for transient disappearance is that spares with transient faults exist. The destination state is one in which the number of working spares (WS) has been increased by 1 and the number of failed spares has been decreased by 1. The transition rate is obtained by multiplying the number of spares with transient faults in the equivalence class (TS) by their disappearance rate and failure rates factor. For example, the disappearance of a transient fault in a spare in equivalence class 1 can be described by the following statement:

```
if TS[1] > 0 tranto WS[1]++, TS[1]--
  by fast TS[1] * TDR_^CT[1] * FRF_^CT[1];
```

The condition for an intermittent fault to go from active to benign exists when there are spares with active intermittent faults. The destination state is one in which the number of benign spares (BS) has been increased by 1, and the number of spares with active intermittent faults (AS) has been decreased by 1. The transition rate is obtained by multiplying the number of spares with active intermittent faults in the equivalence class (AS) by their benign rate and failure rates factor. For example, an intermittent fault going from active to benign in a spare in equivalence class 1 can be described by the following statement:

```
if AS[1] > 0 tranto BS[1]++, AS[1]--
  by fast AS[1] * IBR_^CT[1] * FRF_^CT[1];
```

The condition for an intermittent fault to go from benign to active is that spares with benign intermittent faults exist. The destination state is one in which AS has been increased by 1, and BS has been decreased by 1. The transition rate is obtained by multiplying the number of spares with benign intermittent faults in the equivalence class (BS) by their active rate and failure rates factor. For example, an intermittent fault going from benign to active in a spare in equivalence class 1 can be described by the following statement:

```
if BS[1] > 0 tranto AS[1]++, BS[1]--
  by BS[1] * IAR_^CT[1] * FRF_^CT[1];
```

3.3.4.3. Dependents in logical subsystems. If each component of type A depends on one component of type B and some components of type A can be in one or more logical subsystems, then multiple transitions are specified for the failure of a component of type B. One transition is for the case in which the dependent component is a spare, and the other transitions are for each of the logical subsystems containing a component of type A. Therefore, if the dependent component can be in n logical subsystems, $n+1$ transitions are generated. If there are two dependent components that can be in n and m logical subsystems, then $(n+1)(m+1)$ transitions are generated and so on.

In addition to the conditions mentioned in subsections 3.3.4.1 and 3.3.4.2, these multiple transitions are conditioned on the dependent components being operational as a spare or in a logical subsystem, as the case may be. The fault arrival rates mentioned in subsections 3.3.4.1 and 3.3.4.2 are multiplied by the probability of each dependent component being in a particular logical subsystem or being a spare. For example, if a component in subset 1 of logical subsystem 2 depends on a component in equivalence class 3, the arrival of a permanent fault in a spare in equivalence class 3 can be described by the following statement:

```
if WS[3] > 0 & W(2,1) > 0 tranto WS[3]--,
  PS[3]++, NF[CT[3]]++, P[2,1]++
  by WS[3] * PFR_^CT[3] * FRF_^CT[3]
  * W(2,1) / (WS[EC[2,1]] + W(2,1));
```

3.3.5. Potentially General Transitions

The ARM program analyzes the system description to derive the potentially general transitions.

These transitions are only derived for those fault types with a nonzero failure rate. They must also have a nonzero transition rate, if they are exponential, or mean, if they are general. For each potentially general transition, the following must be derived: the conditions under which the transition would take place, its destination state, and its transition rate expression.

Each condition can lead to two types of transitions which include a successful transition, if the coverage is not 0, and a system failure transition, if the coverage is not 1. The destination state for such system failure transitions is a death state in which CF is true. The transition rate expression of a system failure transition is the same as that for a successful transition except that the coverage c is replaced by $(1 - c)$. Therefore, only the destination states and transition rate expressions of successful transitions are given in the subsections that follow.

The rate expression of a general transition is composed of the mean time, standard deviation, and probability. Except for spare recoveries, the transition time mean and standard deviation are the repair, recovery, or reconfiguration mean and standard deviation. The transition probabilities given in the subsections that follow are for the case in which there are no competing fast exponential transitions. These probabilities are multiplied by the repair, recovery, or reconfiguration probability when there is such competition.

After all the potentially general transitions have been derived, they are specified based on their priority using the algorithm in subsection A3. This algorithm only allows competition between transitions with the same priorities.

3.3.5.1. Spare recoveries. Spare recovery transitions are derived only for those component types whose failure rates factor and detectable fraction are nonzero. The transition condition is that a spare has failed. The destination state is one in which the number of components reconfigured out (NR) has been increased by 1 and the number of failed spares has been decreased by 1.

The exponential transition rate expression is the product of multiplying the recovery rate, the coverage, the detectable fraction, and the probability that the system recovers from this fault type and not some other fault types that it may have in spares of this component type. The general transition probability is the product of multiplying the coverage by the previous probability. The general transition time mean and standard deviation are the quotients of dividing the detectable fraction into the recovery mean and standard deviation. For example, the recovery from a permanent fault in a spare in equivalence class 2 and of type 1, which has two equivalence classes, can be described by the following statement:

```
if PS[2] > 0 tranto NR[2]++, PS[2]--
  by DF_1 * SRR_1 * SRC_1 * PS[2]
  / (NBS(1) + NBS(2));
```

where SRR, SRC, and DF stand for the spare recovery rate, the spare recovery coverage, and the detectable fraction, respectively.

3.3.5.2. Component repairs. The transition condition for a component repair is that NR is nonzero. The destination state is one in which the number of working spares (WS) in the equivalence class has been increased by 1 and NR has been decreased by 1. An alternative to this destination state is to restore a subsystem that was retired or whose redundancy had been diminished instead of increasing the number of working spares. Because this alternative requires tracking all subsystem retirements and degradations to decide which one to reverse, it has not yet been implemented.

The exponential transition rate expression is the product of multiplying the repair rate, the coverage, and the probability that a component from this equivalence class is repaired and not some other component of this type. The general transition probability is the product of multiplying the coverage by the previous probability. For example, the repair of a component in equivalence class 2 and of type 1, which has two equivalence classes, can be described by the following statement:

```
if NR[2] > 0 tranto WS[2]++, NR[2]-- by
  CRR_1 * CRC_1 * NR[2] / (NR[1] + NR[2]);
```

where CRR and CRC stand for the component repair rate and coverage.

3.3.5.3. Logical subsystem recoveries. The transition conditions for a logical subsystem recovery are that the subsystem is operational, one of its components has failed, and there is an equivalent spare to replace it. If the spare is working properly, the destination state is one in which NR has been increased by 1 and both the number of failed components in that subsystem and the number of spares have been decreased by 1. If the spare is not working properly, the destination state is one in which NR has been increased by 1 and the number of spares has been decreased by 1.

The exponential transition rate expression is the product of multiplying the recovery rate, the coverage, and two probabilities. One probability is that of the system recovering from this fault and not some other faults that it may have in subsystems of this

type. The other probability is that of getting a spare in the working state that is indicated in the destination state. The general transition probability is the product of multiplying the coverage by the previous two probabilities. For example, the recovery from a permanent fault in equivalent subset 1 of logical subsystem 2 of type 3, which has two subsystems, can be described by the following statement:

```
if LO[2] & P[2,1] > 0 & WS[EC[2,1]] > 0
  tranto NR[EC[2,1]]++, P[2,1]--,
  WS[EC[2,1]]-- by LRR_3 * LRC_3 * PL(2)
  / (NBL(2) + NBL(4)) * WS[EC[2,1]]
  / NS(EC[2,1]);
```

where LRR and LRC stand for the logical subsystem recovery rate and coverage.

3.3.5.4. Reconfigurations that retire a subsystem. The transition conditions for a reconfiguration that retires a subsystem are that the subsystem is operational, one of its components has failed, there is no equivalent spare to replace it, and the number of operational subsystems of this type is the same as specified in the source vertex of the reconfiguration graph. The destination state is one in which the subsystem is no longer operational, the number of operational subsystems of this type (LTC) has been decreased by 1, NR has been increased by 1, the number of failed components in that subsystem is 0, and the number of spares has been incremented by the number of components in that subsystem minus 1.

The exponential transition rate expression is the product of multiplying the reconfiguration rate, the coverage, and the probability that the system reconfigures because of this fault and not some other faults it may have in subsystems of this type. The general transition probability is the product of multiplying the coverage by the previous probability. For example, the reconfiguration from a permanent fault in the only equivalent subset of logical subsystem 2, of a type which has two subsystems, can be described by the following statement:

```
if LO[2] & P[2,1] > 0 & WS[EC[2,1]] = 0
  & LTC[LT[2]] = 2 tranto NR[EC[2,1]]++,
  WS[EC[2,1]] = WS[EC[2,1]] + W(2,1),
  PS[EC[2,1]] = PS[EC[2,1]] + P[2,1] - 1,
  TS[EC[2,1]] = TS[EC[2,1]] + T[2,1],
  AS[EC[2,1]] = AS[EC[2,1]] + A[2,1],
  BS[EC[2,1]] = BS[EC[2,1]] + B[2,1],
  LTC[LT[2]]--, LO[2] = false, P[2,1] = 0,
  T[2,1] = 0, A[2,1] = 0, B[2,1] = 0 by
  RR_3 * RC_3 * PL_(2) / (NBL(2) + NBL(4));
```

where RR and RC stand for the system reconfiguration rate and coverage.

3.3.5.5. Reconfigurations that degrade a subsystem. If a subsystem is going to be degraded by m components and m is greater than 1, two alternative ways exist to deal with the $m-1$ components that could be working but are not going to be part of the new subsystem. These components could be reconfigured out of the system, or they could be declared to be spares.

The implementation of either alternative must consider all the possible ways to chose $m-1$ components out of the n components in the old subsystem and the probability of each selection. The reason for these alternatives is that more than one of the n components might have a fault. In that case, it makes a difference which ones are chosen.

The first alternative is always conservative, but for some systems, it could be overly conservative. The ARM program implements the second alternative in a way that does not lead to optimistic results. For subsystems that do not recover from faults by using spares, their recovery rate or mean, whichever applies, would be 0, and therefore the $m-1$ components would not be used as spares for them.

The transition conditions for a reconfiguration that degrades a subsystem by m components are that the old subsystem is operational, one of its components has failed, no equivalent spare exists to replace it, and the number of operational subsystems of this type (LTC) is greater than or equal to the number specified in the source vertex of the reconfiguration graph. The destination state is one in which the old subsystem is no longer operational, NR has been increased by 1, the number of spares has been incremented by $m-1$, the number of failed components in the old subsystem is 0, a new subsystem is operational with the components of the old subsystem minus m, and the number of operational subsystems is decreased for the old subsystem's type and increased for the new subsystem's type. To simplify the model specification, ARM currently limits m to 2.

The exponential transition rate expression is the product of multiplying the reconfiguration rate, the coverage, and the probability that the system reconfigures because of this fault and not because of some other faults that it may have in subsystems of this type. The general transition probability is the product of multiplying the coverage by the previous probability. For example, the degradation to a subsystem that has two less components is caused by a permanent fault in the only equivalent subset of logical subsystem 2, of a type which has two subsystems.

This degradation can be described by the following statement:

```
if LO[2] & P[2,1] > 0 & WS[EC[2,1]] = 0
  & LTC[LT[2]] = 2 tranto NR[EC[2,1]]++,
  WS[EC[2,1]]++, LTC[LT[2]]--,
  LO[2] = false, C[2,1] = 0, P[2,1] = 0,
  T[2,1] = 0, A[2,1] = 0, B[2,1] = 0,
  LTC[LT[3]]++, LO[3] = true,
  C[3,1] = C[2,1] - 2, P[3,1] = P[2,1] - 1,
  T[3,1] = T[2,1], A[3,1] = A[2,1],
  B[3,1] = B[2,1] by RR_5 * RC_5 * PL(2)
  / (NBL(2) + NBL(4));
```

3.4. Advanced Features Not Yet Implemented

Two of the more advanced features of the ARM system description language have not yet been implemented due to the complexities they involve. If their use is attempted, ARM warns the user that these features have not yet been implemented. The following subsections outline how they might be implemented in the future.

3.4.1. Reinitializing Reconfigurations

The transition conditions for a reinitializing reconfiguration are that (1) a logical subsystem has failed; (2) excluding the components whose failure caused the logical subsystem to fail, there are still enough components to meet the system requirements; and (3) the components and/or subsystems required for the reinitializing reconfiguration have not failed. The destination state is one in which the components whose failure caused the logical subsystem to fail have been reconfigured out of the system. The destination state is also one in which either those components reconfigured out would be replaced by spares, if available, or the failed subsystem would be retired or degraded.

The exponential transition rate expression is the product of multiplying the reconfiguration rate by the coverage. The general transition probability is the coverage of the reinitializing reconfiguration.

3.4.2. Mission Phase Change Reconfigurations

No transition conditions exist for mission phase change reconfigurations because they can occur at any moment. The destination state is determined by the reconfiguration graph. This graph is one in which the logical subsystems in the source vertex are no longer operational and the logical subsystems in the destination vertex are now operational.

The exponential transition rate expression is the product of multiplying the reconfiguration rate by the coverage. The general transition probability is the coverage of the mission phase change reconfiguration.

4. Application Examples and Results

The following subsections illustrate the type of systems that can be described with the GUI and give evidence that the results ARM generates are correct. This is done by comparing the results ARM generates for four systems with manually generated results. To make this comparison possible, each pair of results is based on the same architecture, requirements, and parameters. For all the system reconfigurations in the examples of this section, the triggering event used was a component that failed without a spare. All of the models were evaluated by SURE using a mission time of 10, a loop truncation level of 25, and a requirement of 2 digits of accuracy. The manually and ARM-generated Markov reliability model specifications for these four systems are presented in Liceaga (1992).

4.1. Comparison With Example Multiprocessor Results

The first system is the multiprocessor used as an example throughout section 2 but without repair. The architecture and requirements of this system remain the same except that no reinitializing reconfiguration is used because that feature of ARM is not yet implemented.

However, some of the parameters used are not the same. Only permanent failure rates were used, and they are shown in table 4.1. A value of 0.999999999 was used for all fault coverage probabilities, and a value of 1 was used for all priorities. A failure rates factor of 1 was used for all the spare components. The other parameters used for the spare components are shown in table 4.2. A rate of 7.8×10^3 was used for all the subsystem recoveries. The values used for the reconfiguration from two processor triads to one were shown in figure 2.22. To generate the model, an ASSIST pruning condition of `TNF >= 5` was used without any trimming. To evaluate the model, a SURE prune level of 1×10^{-18} was used.

Statistics comparing the manually and ARM-generated Markov reliability model specifications are shown in table 4.3. All of the ASSIST and SURE execution times in table 4.3 and throughout this paper were measured on a Sun Microsystems SPARCstation 2 computer. The probability-of-failure results are compared in table 4.4.

Table 4.1. All Permanent Failure Rates of the Multiprocessor

Parameter	Component type					
	p	m	w	tb	rb	wb
Permanent failure rate, hr^{-1}	5×10^{-5}	5×10^{-5}	2×10^{-6}	3×10^{-6}	3×10^{-6}	3×10^{-6}

Table 4.2. Some Spare Component Parameters of the Multiprocessor

Parameter	Component type					
	p	m	w	tb	rb	wb
Detectable fraction	0.9	0	0.9	0	0	0
Recovery rate, hr^{-1}	3.3×10^{3}	0	1.8×10^{4}	0	0	0

Table 4.3. Reliability Model Statistics for the Multiprocessor

Statistic	Manual	ARM
ASSIST file lines	195	631
Model generation time, hr	0.36	2.72
States	33 611	46 338
Transitions	694 980	1 092 898
Model evaluation time, hr	1.76	3.18

Table 4.4. Probability-of-Failure Results for the Multiprocessor

Measure	Manual	ARM	Difference	Percentage
Lower bound	2.01939×10^{-10}	2.01835×10^{-10}	1.04×10^{-13}	5.15×10^{-2}
Upper bound	2.05305×10^{-10}	2.05307×10^{-10}	-2×10^{-15}	-9.74×10^{-4}

4.2. Application to Systems Described in the Literature

The following subsections illustrate the applicability of the GUI to systems that have been described in the technical literature and give evidence that the results ARM generates are correct.

4.2.1. Software Implemented Fault-Tolerance (SIFT) Computer

A SIFT computer, described in Goldberg et al. (1984), can initially be configured as a processor *p* sextuple (ST). In addition to a central processing unit (CPU), each processor contains its own memory, input/output port, and power supply. As processors fail, SIFT first reconfigures into a quintuple, then a quad, then a triad, and finally into a nonreconfigurable dual.

The logical hierarchy and requirements of the triad subsystem class were shown in figures 2.11 and 2.16. The remainder of the architecture and requirements of this SIFT computer are described by figures 4.1 to 4.15. Figures 4.1 to 4.3 describe the initial configuration. Figures 4.4 to 4.6 describe possible future cofigurations. Figures 4.7 to 4.10 describe

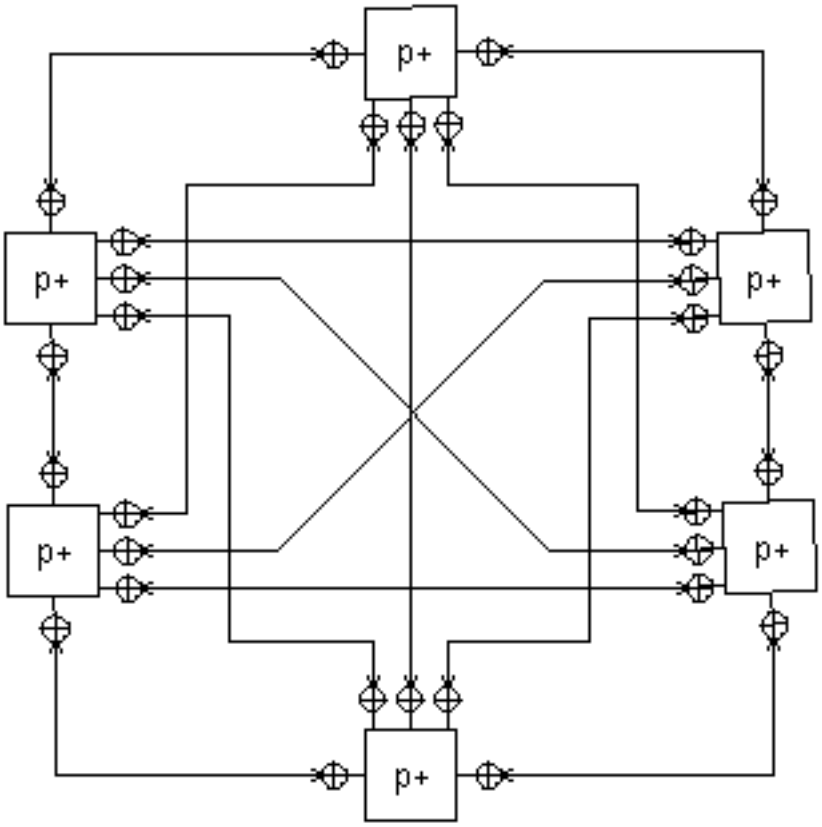

Figure 4.1. External structure of a SIFT computer.

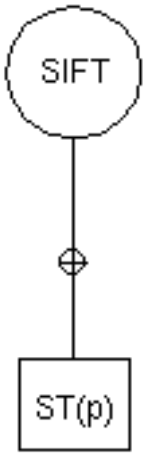

Figure 4.2. Logical hierarchy of a SIFT computer.

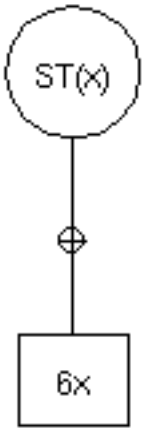

Figure 4.3. Logical hierarchy of the sextuple subsystem class.

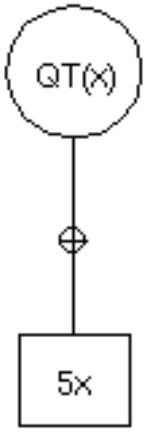

Figure 4.4. Logical hierarchy of the quintuple (QT) subsystem class.

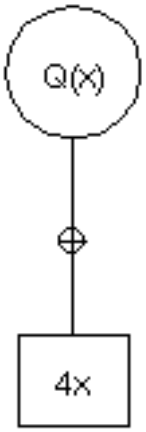

Figure 4.5. Logical hierarchy of the quad (Q) subsystem class.

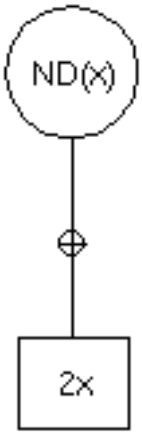

Figure 4.6. Logical hierarchy of the nonreconfigurable dual (ND) subsystem class.

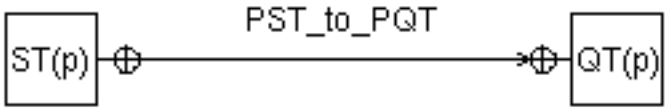

Figure 4.7. Degradation from a processor sextuple (PST) to a quintuple (PQT).

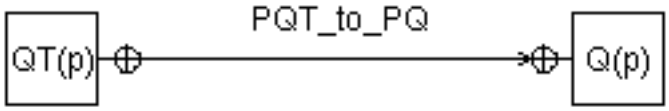

Figure 4.8. Degradation from a processor quintuple to a quad (PQ).

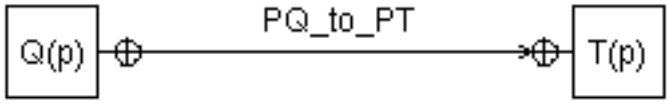

Figure 4.9. Degradation from a processor quad to a triad (PT).

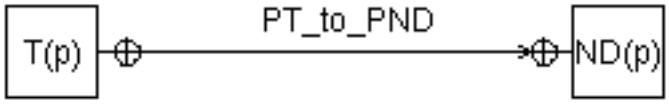

Figure 4.10. Degradation from a processor triad to a nonreconfigurable dual (PND).

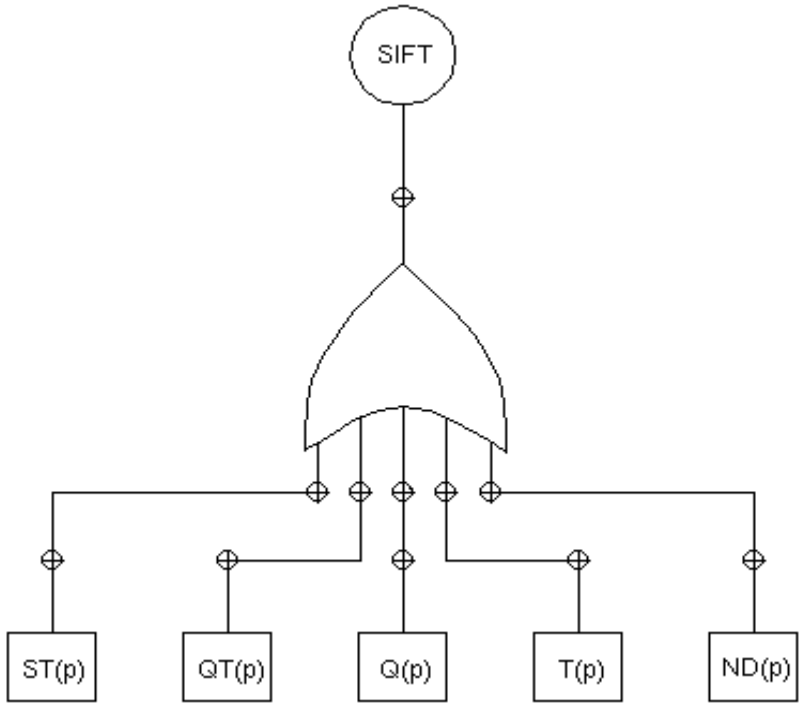

Figure 4.11. Requirements of a SIFT computer.

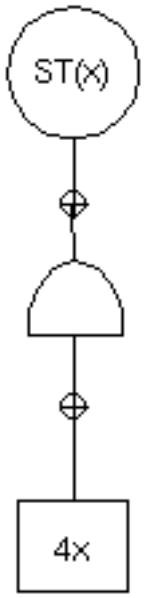

Figure 4.12. Requirements of the sextuple subsystem class.

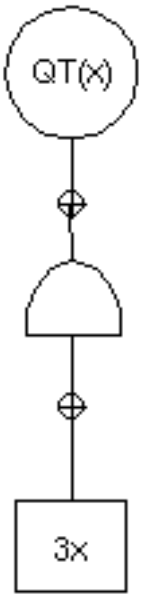

Figure 4.13. Requirements of the quintuple subsystem class.

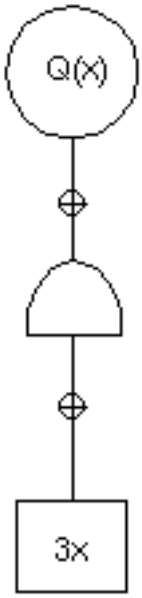

Figure 4.14. Requirements of the quad subsystem class.

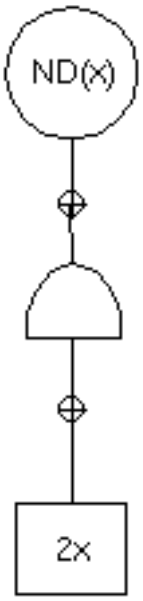

Figure 4.15. Requirements of the nonreconfigurable dual subsystem class.

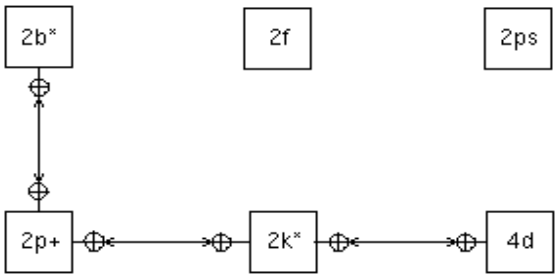

Figure 4.16. External structure of a Tandem computer.

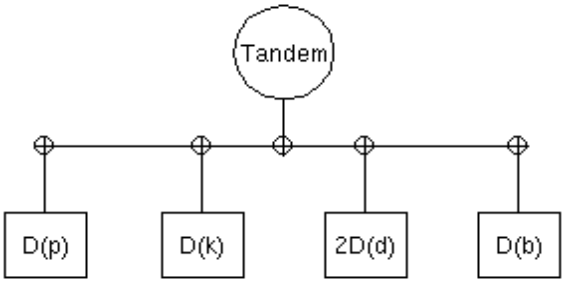

Figure 4.17. Logical hierarchy of a Tandem computer.

Table 4.5. Reliability Model Statistics for a SIFT Computer

Statistic	Manual	ARM
ASSIST file lines	18	330
Model generation time, sec	0.23	1.69
States	12	18
Transitions	17	17
Model evaluation time, sec	0.02	0.063

Table 4.6. Probability-of-Failure Results for a SIFT Computer

Measure	Manual	ARM	Difference	Percentage
Lower bound	7.43383×10^{-15}	7.47849×10^{-15}	-4.466×10^{-17}	-0.601
Upper bound	7.71581×10^{-15}	7.76217×10^{-15}	-4.636×10^{-17}	-0.601

the degradations from six to two processors. Figures 4.11 and 4.12 describe the requirements of the initial configuration, and figures 4.13 to 4.15 describe the requirements of possible future configurations.

A permanent failure rate of 1×10^{-4} hr^{-1} was used for the processors. A coverage of 1, a priority of 1, and a rate of 3.6×10^{3} were used for all the system reconfigurations. The model was generated without any pruning or trimming. Default values were used for the model evaluation parameters, and these were shown in figure 2.24.

A Markov reliability model for this SIFT computer was specified on page 48 of Butler and Johnson (1990). Statistics comparing the manually and ARM-generated Markov reliability model specifications are shown in table 4.5. Their probability-of-failure results are compared in table 4.6.

4.2.2. Comparison With Self-Generated Results

The following subsections give further evidence that the results ARM generates are correct by comparing them with results that were generated manually. A value of 4×10^{3} was used for all transient disappearance rates and intermittent benign rates. A value of 4×10^{-2} was used for all intermittent active rates.

4.2.2.1. Tandem computer. An almost minimal version of a Tandem computer, described in Katzman (1977), is composed of one processor p dual, one disk controller k dual, two disk drive d duals, two fans f, two power supplies ps, and one interprocessor bus b dual. In addition to a CPU, each processor contains its own memory. When a component in a dual (D) fails, the subsystem is reconfigured into a simplex (S).

This Tandem computer requires all subsystems, one fan, and one power supply for it to be operational. Each dual requires that one of its components be working for it to be operational.

The architecture and requirements of this Tandem computer are described by figures 4.16 to 4.30. Figures 4.16 to 4.19 describe the initial and future configurations. Figures 4.20 to 4.23 describe the degradations from dual to simplex subsystems. Figures 4.24 to 4.30 describe the initial and future configuration requirements.

To generate the model, an ASSIST pruning condition of `TNF >= 4` was used without any trimming. To evaluate the model, a SURE prune level of 1×10^{-11} was used. The other parameters used for this Tandem computer are shown in tables 4.7 to 4.9.

Statistics comparing the manually and ARM-generated Markov reliability model specifications are shown in table 4.10. Their probability-of-failure results are compared in table 4.11.

4.2.2.2. Stratus computer. An almost minimal version of a Stratus computer, described in Siewiorek and Swarz (1992), is composed of one computer module and two disk drive d duals. The components in this version of a computer module are grouped into two module regions (MR's). Each MR is composed of a processor board pb, a memory board mb, a disk

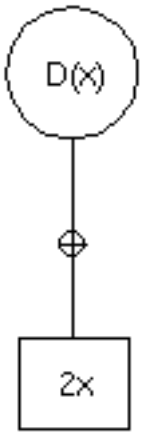

Figure 4.18. Logical hierarchy of the dual subsystem class.

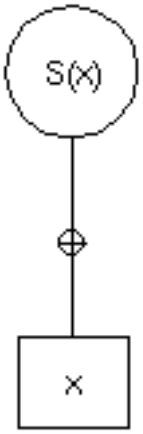

Figure 4.19. Logical hierarchy of the simplex subsystem class.

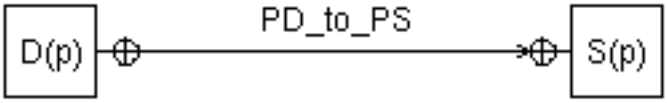

Figure 4.20. Degradation from a processor dual (PD) to a simplex (PS).

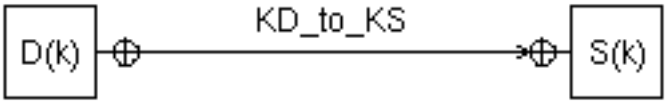

Figure 4.21. Degradation from a disk controller dual (KD) to a simplex (KS).

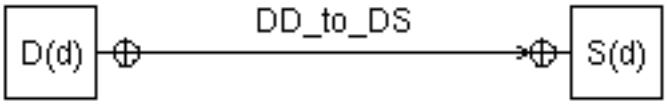

Figure 4.22. Degradation from a disk drive dual (DD) to a simplex (DS).

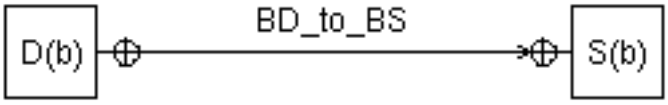

Figure 4.23. Degradation from a bus dual (BD) to a simplex (BS).

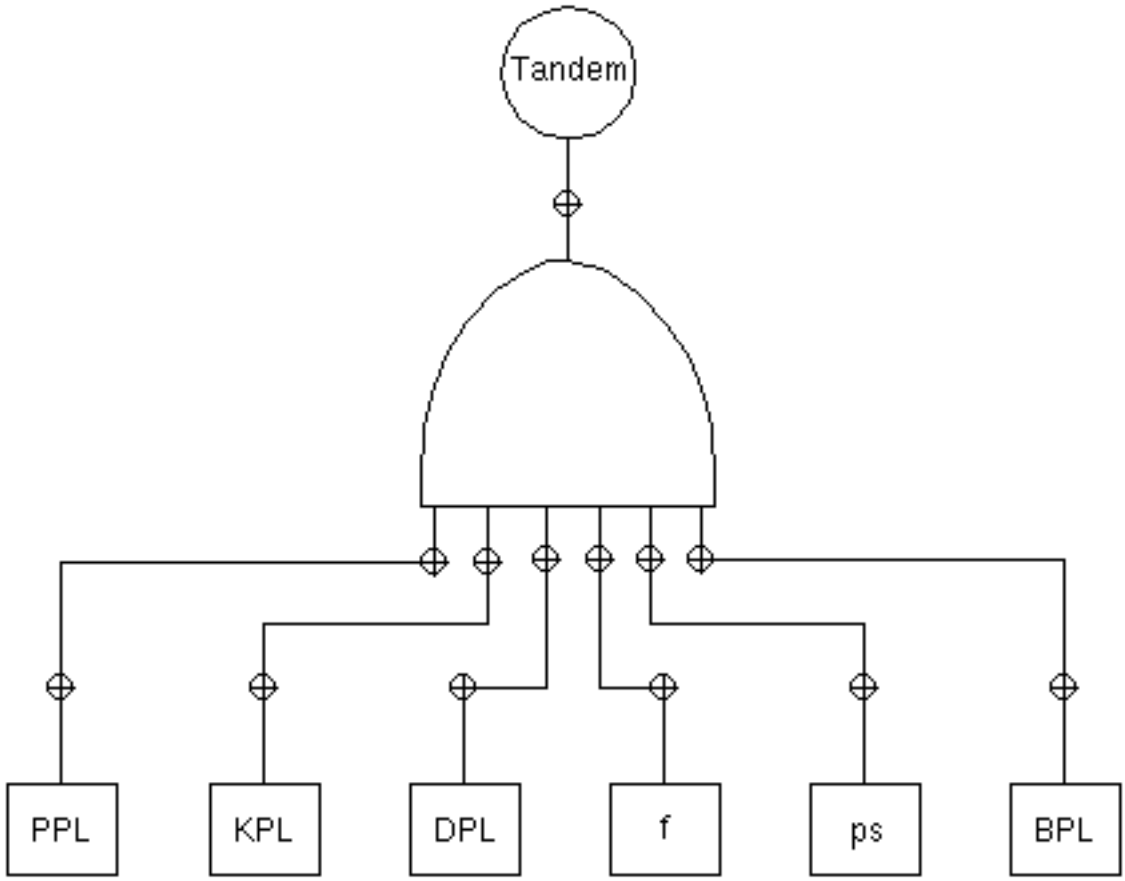

Figure 4.24. Requirements of a Tandem computer.

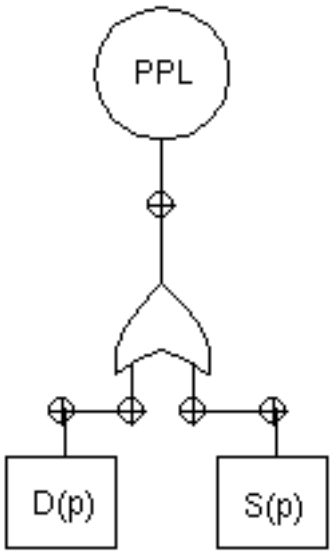

Figure 4.25. Requirements of the PPL performance level.

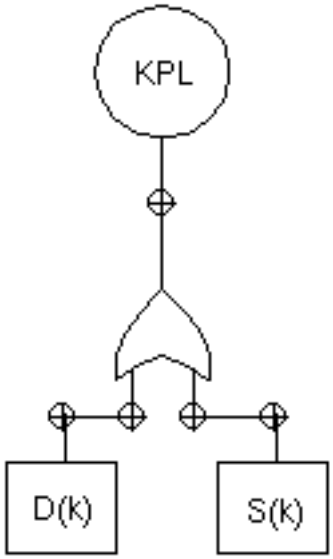

Figure 4.26. Requirements of the KPL performance level.

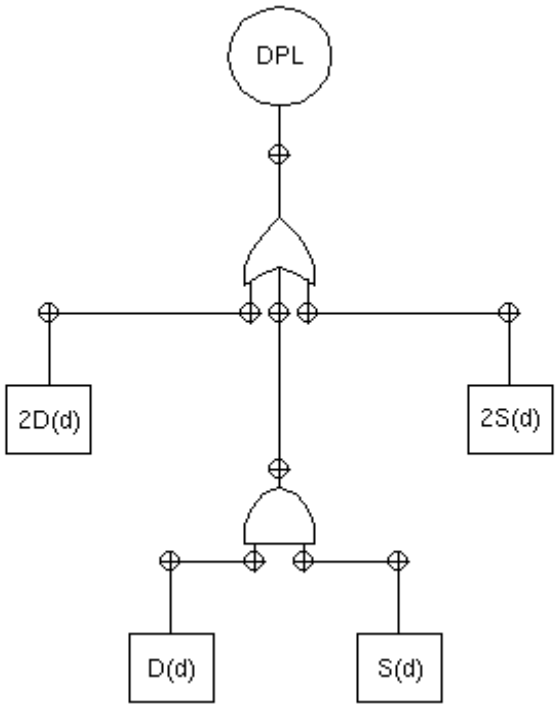

Figure 4.27. Requirements of the DPL performance level.

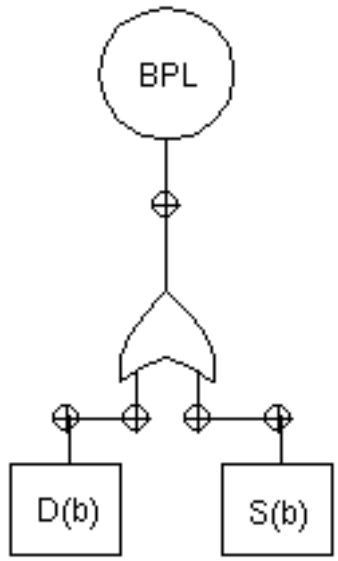

Figure 4.28. Requirements of the BPL performance level.

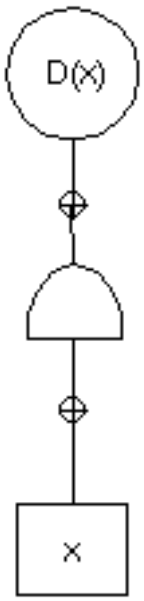

Figure 4.29. Requirements of the dual subsystem class.

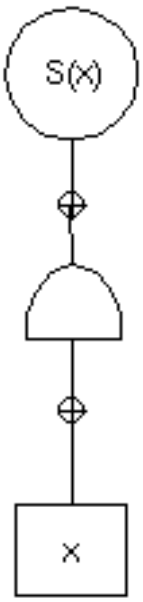

Figure 4.30. Requirements of the simplex subsystem class.

Table 4.7. Some Active Component Parameters of a Tandem Computer

Parameter	Component type					
	p	k	d	f	ps	b
Permanent failure rate, hr^{-1}	5×10^{-5}	2×10^{-5}	4×10^{-5}	1×10^{-6}	3×10^{-5}	3×10^{-6}
Transient failure rate, hr^{-1}	5×10^{-4}	2×10^{-4}	4×10^{-4}	1×10^{-5}	3×10^{-4}	3×10^{-5}
Intermittent failure rate, hr^{-1}	5×10^{-6}	2×10^{-6}	4×10^{-6}	1×10^{-7}	3×10^{-6}	3×10^{-7}

Table 4.8. Some Component Repair Parameters of a Tandem Computer

Parameter	Component type					
	p	k	d	f	ps	b
Repair priority	5	7	9	8	6	10
Repair rate, hr^{-1}	30	30	30	30	30	15

Table 4.9. Some System Reconfiguration Parameters of a Tandem Computer

Parameter	Specification name			
	PD_to_PS	KD_to_KS	DD_to_DS	BD_to_BS
Component type	p	k	d	b
Reconfiguration coverage	0.999999	0.99999975	0.99999975	0.99999975
Reconfiguration priority	1	2	3	4
Reconfiguration rate, hr^{-1}	1.8×10^3	7.2×10^3	25	7.2×10^3

Table 4.10. Reliability Model Statistics for a Tandem Computer

Statistic	Manual	ARM
ASSIST file lines	157	794
Model generation time, min	6.46	75.6
States	11 945	12 920
Transitions	284 121	314 967
Model evaluation time, min	3.94	8.85

Table 4.11. Probability-of-Failure Results for a Tandem Computer

Measure	Manual	ARM	Difference	Percentage
Lower bound	1.84026×10^{-5}	1.7866×10^{-5}	5.366×10^{-7}	2.92
Upper bound	2.10696×10^{-5}	2.05823×10^{-5}	4.873×10^{-7}	2.31

controller board kb, a power supply ps, and a module bus b. Each board contains duplicated logic and an onboard comparator that will stop the board from transmitting on the bus in case of a mismatch. Except for the power supply, each component in one MR forms a dual with the component of the same type in the other MR. The bus is used to perform the OR logical function on the output signals of boards in a dual. Therefore, no reconfiguration takes place when a board fails. When a component fails in a disk drive or bus dual, the subsystem is reconfigured into a simplex.

This Stratus computer requires all subsystems for it to be operational. Each dual requires that one of its components be working for it to be operational. Each MR requires that its power supply be working for it to be operational.

The logical hierarchy and requirements of the dual and simplex subsystem classes are shown in figures 4.18, 4.19, 4.29, and 4.30. The requirements of the DPL and BPL performance levels are shown in figures 4.26 and 4.27, respectively. The system reconfiguration graphs corresponding to the disk drive and bus subsystems are shown in figures 4.22 and 4.23, respectively. The remainder of the architecture and requirements of this Stratus computer are described by figures 4.31 to 4.36. Figures 4.31 to 4.34 complete the description of the initial configuration. Figures 4.35 and 4.36 complete the description of the initial configuration requirements.

To generate the model, an ASSIST pruning condition of `TNF >= 4` was used without any trimming. To evaluate the model, a SURE prune level of 1×10^{-10} was used. The other parameters used for this Stratus computer are shown in tables 4.12 to 4.14.

Statistics comparing the manually and ARM-generated Markov reliability model specifications are shown in table 4.15. Their probability-of-failure results are compared in table 4.16.

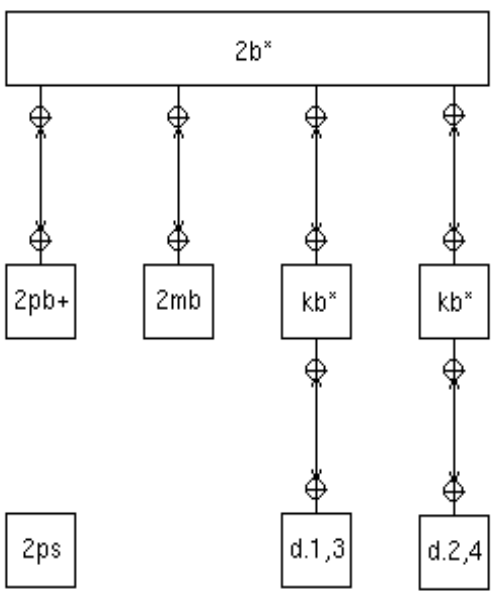

Figure 4.31. External structure of a Stratus computer.

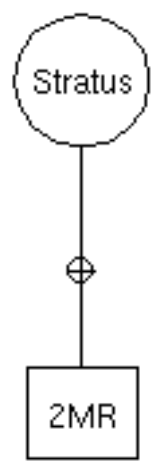

Figure 4.32. Physical hierarchy of a Stratus computer.

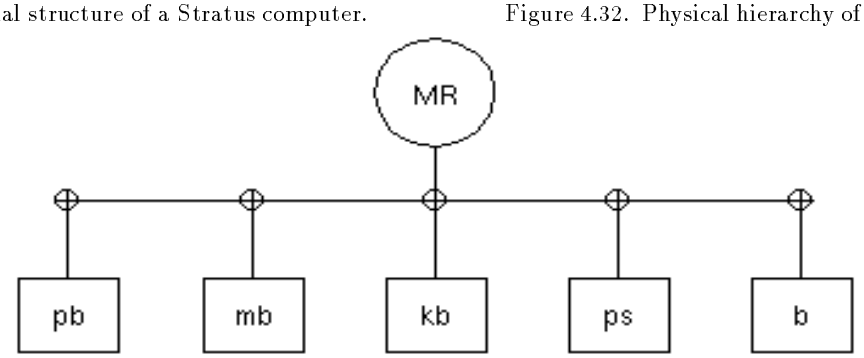

Figure 4.33. Physical hierarchy of the MR subsystem type.

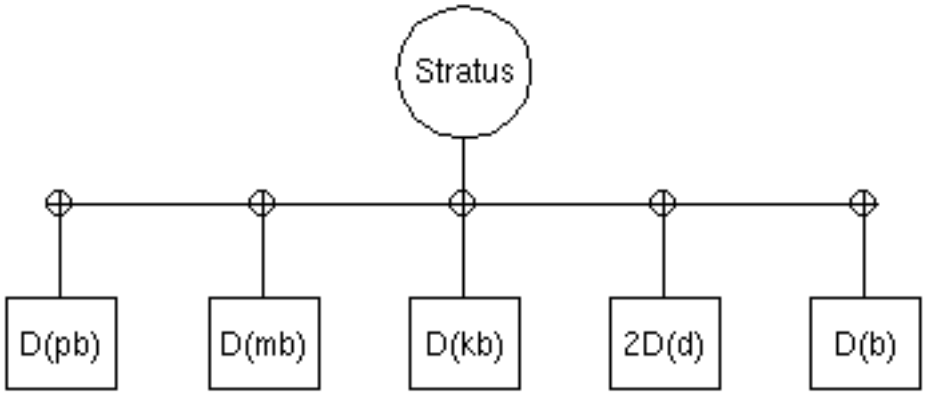

Figure 4.34. Logical hierarchy of a Stratus computer.

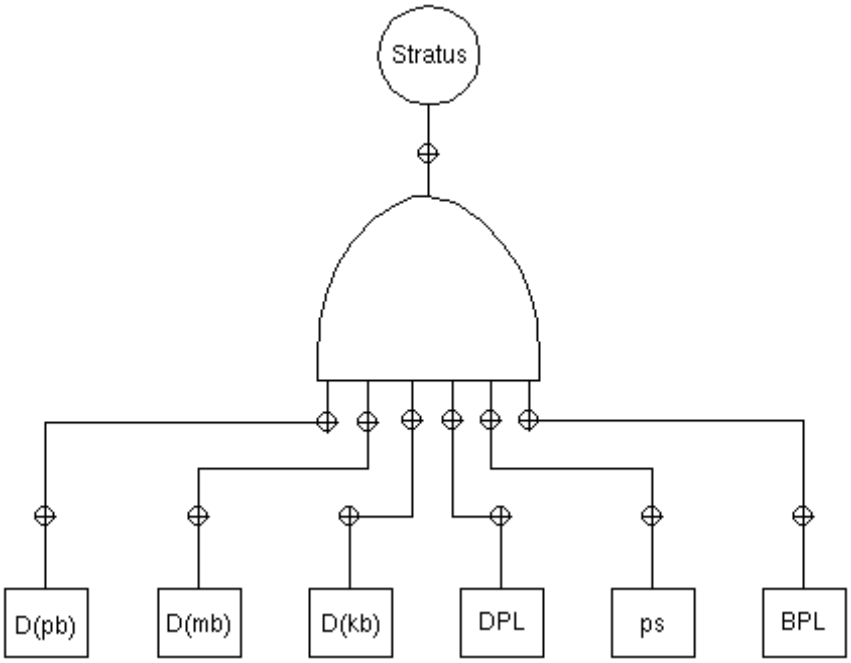

Figure 4.35. Requirements of a Stratus computer.

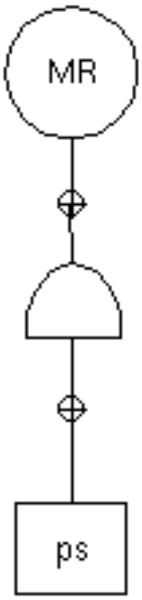

Figure 4.36. Requirements of the MR subsystem type.

Table 4.12. Some Active Component Parameters of a Stratus Computer

Parameter	Component type					
	pb	mb	kb	d	ps	b
Permanent failure rate, hr^{-1}	5×10^{-5}	5×10^{-5}	2×10^{-5}	4×10^{-5}	3×10^{-5}	3×10^{-6}
Transient failure rate, hr^{-1}	5×10^{-4}	5×10^{-4}	2×10^{-4}	4×10^{-4}	3×10^{-4}	3×10^{-5}
Intermittent failure rate, hr^{-1}	5×10^{-6}	5×10^{-6}	2×10^{-6}	4×10^{-6}	3×10^{-6}	3×10^{-7}

Table 4.13. Some Component Repair Parameters of a Stratus Computer

Parameter	Component type					
	pb	mb	kb	d	ps	b
Repair priority	5	4	6	7	3	8
Repair rate, hr^{-1}	30	30	30	30	30	15

Table 4.14. Some System Reconfiguration Parameters of a Stratus Computer

Parameter	Specification name	
	DD_to_DS	BD_to_BS
Component type	d	b
Reconfiguration coverage	0.99999975	0.99999975
Reconfiguration priority	1	2
Reconfiguration rate, hr^{-1}	25	7.2×10^3

Table 4.15. Reliability Model Statistics for a Stratus Computer

Statistic	Manual	ARM
ASSIST file lines	155	1472
Model generation time, min	4.2	58.88
States	8197	10 462
Transitions	193 790	248 109
Model evaluation time, min	3.13	8.37

Table 4.16. Probability of Failure Results for a Stratus Computer

Measure	Manual	ARM	Difference	Percentage
Lower bound	8.0522×10^{-5}	7.53168×10^{-5}	5.2052×10^{-6}	6.46
Upper bound	9.03802×10^{-5}	8.85121×10^{-5}	1.8681×10^{-6}	2.07

5. Analysis

The next three subsections analyze the assumptions, utility, and performance of ARM. Subsection 5.4 discusses how ARM might be validated. Subsection 5.5 gives some lessons learned from using ARM.

5.1. Summary of Assumptions

The purpose of this subsection is to make sure that the potential users of the ARM approach to reliability modeling are aware of the assumptions that are inherent in it. The user is responsible for determining whether those assumptions are applicable to the system whose reliability they want to estimate or the error they introduce is acceptable given the reliability and accuracy the system requires. If any of the following six assumptions are not true, the result will be conservative:

1. It is assumed that components which communicate and are critical (i.e., required for the system to be operational) must be able to continue communicating.
2. A transition that reconfigures components and/or subsystems in or out of the system is assumed to describe sequential processes. For example, if there are n faults in one or more subsystems of the same type with recovery rate ρ, the rate at which one of the faulty components is replaced by a spare is assumed to be ρ not $n\rho$. Typically, these transitions are fast, in which case this assumption being false would have little effect.
3. A rate that characterizes the failure behavior of a component is assumed to describe concurrent processes. For example, if there are n working components of the same type with permanent failure rate λ, the rate at which one of them fails permanently is assumed to be $n\lambda$.
4. It is assumed that when a component in a logical subsystem fails, it can only be replaced by a component that is equivalent in terms of its type and its connections to other components.
5. Components that depend on a component that has a soft fault are assumed to have failed permanently.
6. It is assumed that repaired components become spares and that they are not used to restore a subsystem that was retired or whose redundancy had been diminished.

The following four assumptions are not strictly true. However, they are often close enough in practice to be used to simplify the analysis:

7. The failure processes of different components are assumed to be independent of one another.
8. Each failure process is assumed to follow an exponential distribution.
9. Numeric parameters are assumed to be independent of the system state.
10. The transition probability of a potentially general transition is assumed to be the same against any competing fast exponential transitions.

To evaluate a model specified by ARM using SURE and get close bounds on the probability of failure, the following three assumptions must be met:

11. The rate at which an intermittent fault goes from benign to active is assumed to be slow.
12. The coverage c and rate ρ specified for transitions that could have been general must be such that the successful transition rate ρc and the system failure transition rate $\rho(1-c)$ that they produce are either slow or fast. If the product of a transition rate and the mission time is less than 0.01, it is slow, and if it is greater than 100, it is fast.
13. The following are assumed to be fast:
    a. Recovery transitions
    b. Reconfiguration transitions
    c. Repair transitions

If the system components do not meet assumption 11, the system's probability of failure due to intermittent faults can be calculated by first estimating or measuring, with fault injection experiments, the recovery and reconfiguration times for intermittent faults and then using those times in a model in which the components fail permanently at the intermittent fault arrival rate. A transition that does not meet assumption 12 can be specified as a general transition with a mean and standard deviation of $1/\rho$, but then it cannot compete with other potentially general transitions. Assumptions 12, 13a, and 13b do not severely restrict the systems that can be modeled because most systems meet them to achieve their reliability requirements. However, assumption 13c is not usually true. The PAWS or STEM programs can be used to evaluate models that do meet assumptions 11 through 13.

5.2. Utility

As demonstrated in section 4, the major goals given for the GUI in section 2 were achieved. The GUI is quite general in that all the redundancy techniques defined in subsection 1.1 can be accommodated. It makes use of physical and logical hierarchies. The GUI also uses subsystem classes and types to reduce the number of subsystems that need to be defined.

The input from each GUI window, whether it be a graph or a set of parameters, is stored in a separate file. A single file can be part of the description of more than one system. A file can also be edited with the GUI to produce similar files without having to create them from scratch. The files used to describe the systems used as examples in section 4 will be available to the ARM user so that they may be reused in this manner. Sharing files can be very useful because, as demonstrated in section 4, many systems use the same subsystem classes and types. Even without reusing files, an experienced user could describe each of the systems used as examples in section 4 in approximately 1 hour.

The GUI is an alternative to learning the ASSIST language and using it to manually specify the reliability model. Subsection 5.2.1 illustrates how simple changes in the system description given through the GUI would require changing a large percentage of a manual ASSIST file. Subsection 5.2.2 illustrates how easy and natural it is to make architectural changes for design tradeoff studies using the GUI.

5.2.1. Adding System Characteristics

This subsection compares what it takes to make a simple addition to a system description given through the GUI versus the corresponding percentage of changes and additions to a manual ASSIST file. The example system used for these additions is the Stratus computer described in subsection 4.2.2.2 but without dependencies between components, imperfect coverage, transient and intermittent faults, and repair.

Dependencies between components were added by giving the hierarchy and requirements of the MR physical subsystem type and the system physical hierarchy. Imperfect coverage, transient and intermittent faults, and repair were added by simply changing the corresponding parameter values. The effect of these additions, described in subsection 4.2.2.2, on a manual ASSIST file is shown in table 5.1. The model specifications for these additions were also generated by ARM to validate the manually generated specifications. The probability-of-failure results calculated with the manual and ARM model specifications are compared in table 5.2.

5.2.2. Performing Design Tradeoffs

The following variations of the sextuple SIFT presented in subsection 4.2.1 were considered:

1. One quintuple plus one spare
2. One quad plus two spares
3. One triad plus three spares
4. One dual plus four spares
5. Two duals plus two spares
6. Three duals
7. Two triads

In addition to the parameter values used for the sextuple SIFT, the following parameter values were used for the other variations of SIFT: a coverage of 1, a priority of 1, and a rate of 3.6×10^3 for the spare and subsystem recoveries; and a failure rates factor of 1 and a detectable fraction of 1 for the spare components.

The quintuple SIFT required the same graphs as the sextuple SIFT except for the ST(x) logical subsystem class hierarchy and requirements and the PST_to_PQT system reconfiguration. Of the remaining graphs, only two had to be modified to replace ST by QT in the system logical hierarchy and remove ST(p) from the system requirements.

The quad SIFT required the same graphs as the quintuple SIFT except for the QT(x) logical subsystem class hierarchy and requirements and the PQT_to_PQ system reconfiguration. Of the remaining graphs, only two had to be modified to replace QT by Q in the system logical hierarchy and remove QT(p) from the system requirements.

The one-triad SIFT required the same graphs as the quad SIFT except for the Q(x) logical subsystem class hierarchy and requirements and the PQ_to_PT system reconfiguration. Of the remaining graphs, only two had to be modified to replace Q by T in the system logical hierarchy and remove Q(p) from the system requirements.

The one-dual SIFT required the same graphs as the one triad SIFT except for the T(x) logical subsystem class hierarchy and requirements and the PT_to_PND system reconfiguration. Of the remaining graphs, only two had to be modified to replace T by ND in the system logical hierarchy and remove T(p) from the system requirements.

Table 5.1. Effect of Simple Changes in the System Description on a Manual ASSIST File

Addition	Declaration lines, percent		Control line, percent	
	Changed	Added	Changed	Added
Dependency	0	0	6.67	13.33
Coverage	1	2	20.00	26.67
Transients	1	4	40.00	40.00
Intermittents	1	6	40.00	46.67
Repair	2	2	20.00	126.67
All of above	2	13	46.67	306.67

Table 5.2. Probability of Failure Results for Some Variations of a Stratus Computer

Addition	Bound	Manual	ARM
None	Lower	9.21527×10^{-7}	9.21527×10^{-7}
	Upper	9.6179×10^{-7}	9.6179×10^{-7}
Dependency	Lower	2.0254×10^{-6}	2.0254×10^{-6}
	Upper	2.06689×10^{-6}	2.06689×10^{-6}
Coverage	Lower	9.21873×10^{-7}	9.21873×10^{-7}
	Upper	9.62206×10^{-7}	9.62206×10^{-7}
Transients	Lower	1.04005×10^{-5}	1.04005×10^{-5}
	Upper	1.14293×10^{-5}	1.14293×10^{-5}
Intermittents	Lower	1.11775×10^{-6}	1.03363×10^{-6}
	Upper	1.18582×10^{-6}	1.10316×10^{-6}
Repair	Lower	8.97387×10^{-7}	8.97377×10^{-7}
	Upper	9.64093×10^{-7}	9.65213×10^{-7}
All of above	Lower	8.0522×10^{-5}	7.53168×10^{-5}
	Upper	9.03802×10^{-5}	8.85121×10^{-5}

Table 5.3. Probability-of-Failure Results for Some Variations of a SIFT Computer

Configuration	Lower bound	Upper bound
One sextuple	7.47849×10^{-15}	7.76217×10^{-15}
One quintuple plus one spare	7.43369×10^{-15}	7.71588×10^{-15}
One quad plus two spares	3.28383×10^{-10}	3.34341×10^{-10}
One triad plus three spares	1.64194×10^{-10}	1.67218×10^{-10}
One dual plus four spares	1.99798×10^{-3}	2.00401×10^{-3}
Two duals plus two spares	3.99197×10^{-3}	4.004×10^{-3}
Three duals	5.982×10^{-3}	6×10^{-3}
Two triads	5.96846×10^{-6}	6.01271×10^{-6}

The two-dual SIFT and the three-dual SIFT required the same graphs as the one-dual SIFT except that in the system logical hierarchy ND was replaced by 2ND and 3ND, respectively. The two-triad SIFT required the same graphs as the one-triad SIFT except that in the system logical hierarchy T was replaced by 2T. The probability-of-failure results for all of these variations of SIFT are compared in table 5.3.

5.3. Performance

From the results presented in section 4, it is seen that the ARM approach produces reliability model specification files with about an order of magnitude as many lines. This occurs because the current implementation specifies all possible parameters, state variables, and functions that might be needed to model the system. Consequently, the time to generate the model is increased by approximately an order of magnitude. It may be possible to optimize ARM to specify only what is actually needed.

From the results presented in section 4, it is observed that the ARM approach specifies models with about a factor of 2, at the most, as many states and transitions. Although this increases the time to evaluate the model by about a factor of 2, it does not limit the systems that can be analyzed because the models can be piped directly into the evaluation program without having to store them.

Table 5.4 shows the time and virtual memory required by ARM to read the system description and specify the model of the systems used as examples in section 4. These measurements were made on a Digital Equipment Corporation VAXstation 3100 computer that uses the VMS operating system. The memory utilization is given in blocks of 512 bytes of 8 bits. From these measurements and the generation and evaluation times presented in section 4, it can be concluded that the time and memory to automatically specify the reliability model are not factors limiting the systems that can be analyzed.

Table 5.4. ARM Model Specification Performance

System	Time, CPU sec	Memory
MP	2.60	6668
SIFT	2.02	6540
Tandem	3.24	6924
Stratus	3.82	6924

5.4. Validation

The user of the automated reliability modeling process proposed in this paper should be aware of its several sources of errors. The failure rates calculated with MIL-HDBK-217F (U.S. Department of Defense 1991) can be off by several orders of magnitude, especially for new technologies or environments for which there are little data. This can also be true of the parameters that describe how the system responds to faults if they are not measured in the laboratory.

For highly reliable systems, the coverage parameter values need to be so close to 1 (e.g., 0.9999999) that it becomes impractical to measure them. For that reason, some computer architects have opted to

prove their system designs and then use coverage parameter values of 1 (Moser et al. 1987). Coverage can have a profound effect on the system probability of failure. This effect is illustrated in table 5.5 for the version of the SIFT computer presented in section 4.

Table 5.5. Effect of Coverage on the Probability of Failure of a SIFT Computer

Coverage	Lower Bound	Upper Bound
1.0	7.47849×10^{-15}	7.76217×10^{-15}
0.99999999999	6.65572×10^{-14}	6.79125×10^{-14}
0.9999999999	5.98266×10^{-13}	6.09357×10^{-13}
0.999999999	5.91535×10^{-12}	6.02288×10^{-12}
0.99999999	5.90862×10^{-11}	6.01614×10^{-11}
0.9999999	5.90795×10^{-10}	6.01513×10^{-10}
0.999999	5.90788×10^{-9}	6.01503×10^{-9}
0.99999	5.90787×10^{-8}	6.01508×10^{-8}
0.9999	5.90786×10^{-7}	6.01562×10^{-7}
0.999	5.90776×10^{-6}	6.02104×10^{-6}
0.99	5.98368×10^{-5}	6.01487×10^{-5}
0.9	5.98234×10^{-4}	6.01352×10^{-4}

Other sources of errors are design and implementation faults in the software tools that specify, generate, and evaluate the reliability model. Although the probability of this type of error must be minimized, experience has shown that the probability of committing an error when manually performing these tasks is far greater even for very new tools such as ARM.

If practical, the most desirable method of validation is to formally prove that ARM is correct and that it has a perfect reliability of 1. However, manual proofs are lengthy, tedious, and error-prone (Ramamoorthy and Bastani 1982). Furthermore, automated proving techniques are still impractical for realistic programs (Ramamoorthy and Bastani 1982).

As with most software of significant size and complexity, exhaustive testing is impossible because there are an infinite number of system descriptions that could be given to ARM as input. However, thorough testing of all program features can be achieved to uncover as many software faults as possible and increase user confidence. This testing should combine black-box and white-box techniques (Myers 1979). The four application examples presented in section 4 are black-box test cases because only the output of the program was considered. The goal of white-box test cases is to make certain that all parts of the program have been exercised.

One of the main difficulties with developing a test case is defining what the output is expected to be. As the software being tested grows in complexity, so does the task of defining the expected output. This is especially true for programs that automate a previously manual task, such as ARM, because the usual reason the task was automated is that it could only be done quickly and reliably for simple cases, as is the case for ARM. This difficulty limits the number of complex test cases which is practical to develop. If and when other programs that perform the same function become available, comparison with them would be a possible solution to this problem.

In the case of ARM, a way to increase the benefits provided by the test cases developed is to perform a sensitivity analysis of each one. This analysis would ensure that the two models agree not only for a specific set of parameter values but also for a range of values. The results of doing this for the version of the SIFT computer presented in section 4 are shown in tables 5.6 and 5.7.

Table 5.6. Effect of the Failure Rate on the Probability of Failure of a SIFT Computer

Failure rate, hr^{-1}	Bound	Manual	ARM
1×10^{-3}	Lower	5.77469×10^{-10}	5.77513×10^{-10}
	Upper	6.16966×10^{-10}	6.17013×10^{-10}
1×10^{-4}	Lower	7.43383×10^{-15}	7.47849×10^{-15}
	Upper	7.71581×10^{-15}	7.76217×10^{-15}
1×10^{-5}	Lower	2.63296×10^{-19}	3.08019×10^{-19}
	Upper	2.73023×10^{-19}	3.1934×10^{-19}

Table 5.7. Effect of the Reconfiguration Rate on the Probability of Failure of a SIFT Computer

Reconfiguration rate, hr^{-1}	Bound	Manual	ARM
3.6×10^{2}	Lower	2.38272×10^{-14}	2.79663×10^{-14}
	Upper	2.7356×10^{-14}	3.19926×10^{-14}
3.6×10^{3}	Lower	7.43383×10^{-15}	7.47849×10^{-15}
	Upper	7.71581×10^{-15}	7.76217×10^{-15}
3.6×10^{4}	Lower	6.1037×10^{-15}	6.10416×10^{-15}
	Upper	6.16738×10^{-15}	6.16877×10^{-15}

5.5. Lessons Learned

For most systems, it is essential to specify a reasonable ASSIST prune condition. This is the factor that most directly controls the size of the model generated. It is quicker to start with severe pruning (e.g., `TNF >= 3`) and then reduce it (e.g., `TNF >= 4`) until its effect (the evaluation programs report it as the *prunestate* bounds) on the probability of failure bounds reaches some acceptable level (e.g., less than 1 percent).

When using SURE to evaluate the model, the evaluation time can be reduced by manually selecting a reasonable SURE prune level. For this type of pruning, it is also quicker to start with a severe pruning level (e.g., 1×10^{-8}) and then reduce it (e.g., 1×10^{-9}) until its effect (SURE reports it as the *sure prune* bounds) on the probability of failure bounds reaches some acceptable level (e.g., less than 1 percent). These two types of pruning can and should be used in conjunction with each other.

6. Conclusions

This paper has demonstrated that the tedious and error-prone task of specifying reliability models can be further automated by graphical representations. From the results presented in section 4, it can be concluded that the Automated Reliability Modeling (ARM) approach produces reliability model specification files with approximately an order of magnitude as many lines. Consequently, the time to generate the model is increased by about an order of magnitude. The number of states and transitions, however, increased by a factor of 2 at the most, and the time to evaluate the model only increased by approximately a factor of 2. The probability of failure calculated using ARM specified models was within 7 percent of that calculated using manually specified models. With present computers, the size of the specification file is not a problem. Typical systems have large models whose model generation time is in the range of 10 to 100 minutes and whose model evaluation time is in the range of 1 to 10 hours. Hence, the modest increase in the model generation and evaluation time will be more than offset by the time saved in specifying the model, which in very complicated systems could be months. Therefore, it can be concluded that the ARM approach to automatic reliability model specification is an efficient way to evaluate the reliability of complex fault-tolerant systems.

6.1. Summary of Work and Contributions

The goal of this research and development effort was to provide the computer architect with a powerful and easy to use software tool that assumes the burden of an advanced reliability analysis that considers intermittent, transient, and permanent faults for computer systems of high complexity and sophistication. This paper defined a general, high-level system description language (SDL) that is easy to learn and use, identified and analyzed the problems involved in the automatic specification of Markov reliability models for arbitrary interconnection structures at the processor-memory-switch (PMS) level, and generated and implemented solutions to these problems. The results of this research have been implemented and experimentally validated in the ARM program.

The ARM program uses a graphical user interface (GUI) as its SDL. This GUI is based on a hierarchy of windows. Some windows have graphical editing capabilities for specifying the system's communication structure, hierarchy, reconfiguration capabilities, and requirements. Other windows have text fields, pull-down menus, and buttons for specifying parameters and selecting actions.

The ARM program outputs a Markov reliability model specification formulated for direct use by programs that generate and evaluate the model. The advantages of such an approach are utility to a larger class of users, not necessarily expert in reliability analysis, and lower probability of human error in the calculation.

6.2. Future Work

This work could be extended in several ways. The most obvious one is to implement reinitializing reconfigurations and mission phase change reconfigurations. An outline of how this might be done has been given in subsection 3.4. Another possible extension is to modify the way that repaired components are currently handled such that they can be used to restore subsystems that were retired or whose redundancy had been diminished instead of just becoming spares.

The ARM program could be generalized such that a subsystem can be degraded by more than two components at a time. This program could also be generalized such that components which depend on a component that has a soft fault become benign when the component they depend on becomes benign. Another way to generalize ARM would be to allow a subsystem to be composed of other subsystems.

Further research is needed to describe transition time distributions that depend on the global state of the system. Additional work is also needed

to easily allow competing general transitions. The competing transition probabilities possibly could be automatically estimated.

The ARM program could also be extended to automate the specification of availability models. However, this automation would require modifying the present model generation and evaluation tools or using other ones.

NASA Langley Research Center
Hampton, VA 23681-0001
January 25, 1993

Appendix

ARM Program Algorithms

This appendix shows the algorithms used by the ARM program.

A1. Symmetry Detection

The function definitions are as follows:

Split_Class (R, C, L): If relation R is not satisfied, it then partitions class C and creates a new class after the last class L. Returns the number of equivalence classes.

Size (C): Returns the number of elements in the vertex equivalence class C.

Element (E, C): Returns element E of the vertex equivalence class C.

Equivalent (E, C, R): True if element E of class C is equivalent in terms of relation R to the preceding class elements.

Equal_Degree (E, C): True if element E of class C has the same degree as the preceding elements of class C.

Equal_Neighbor_Classes (E, C): True if element E of class C has the same number of neighbor types in each class as the preceding elements of class C.

```
Equivalent (Current_Element, Class, Relation) {
  if (Relation == Degree)
    return Equal_Degree (Current_Element, Class);
  else
    return Equal_Neighbor_Classes (Current_Element, Class);
}

Split_Class (Relation, This_Class, Last_Class) {
  Split = false;
  for (I = 2; I <= Size (This_Class); I++) {
    Current_Element = Element (I, This_Class);
    if (!Equivalent (Current_Element, This_Class, Relation)) {
      if (!Split) {
        Split = true;
        Last_Class++;
        /* Create a new Last_Class with the degree and
          neighbor attributes of the Current_Element of This_Class.  */
      }
      /* Move the Current_Element of This_Class to the Last_Class.  */
    }
  }
  return Last_Class;
} /* Split_Class */

Symmetry () {

  /* Step 1:  Split based on equal type.  */

  Last_Class = Last_Type;
  for (I = 1; I <= Last_Class; I++)
    /* Add elements of type I to class I. */
```

```
  /* Step 2:  Split based on equal degree.  */

  I = 1;
  while (I <= Last_Class) {
    Last_Class = Split_Class (Degree, I, Last_Class);
    I++;
  }

  /* Step 3:  Split based on equal neighbor classes.  */

  New_Last = Last_Class;
  Done = false;
  while (!Done) {
    for (I = 1; I <= Last_Class; I++)
      New_Last = Split_Class (Neighbors, I, New_Last);
    if (Last_Class == New_Last)
      Done = true;
    else
      Last_Class = New_Last;
  }

} /* Symmetry */
```

A2. Determining the Subsystem Hierarchies

The variable definitions are as follows:

View: Determines whether the algorithm is dealing with logical subsystems in the initial configuration or physical subsystems.

Subsystem_Classes[V]: Number of subsystem classes in view V.

Subsystem_Types[C, V]: Number of subsystem types of class C in view V.

Subsystems[C, T, V]: Number of subsystems of type T and class C in view V.

Class_Hierarchy[C, V]: True if the hierarchy of subsystem class C in view V was obtained from the system hierarchy or a separate file.

The function definitions are as follows:

Get_Class_Hierarchy (C, V): Returns true if it is able to obtain the hierarchy of subsystem class C in view V from the system hierarchy or a separate file.

Read_Subsystem_Hierarchy (C, T, S, V): Reads the hierarchy file of subsystem S of type T and class C in view V which assigns specific components to the subsystem.

Determine_Subsystem_Hierarchy (C, T, S, V): Determines the hierarchy of subsystem S of type T and class C in view V from the hierarchy of its subsystem class and the component type arguments, if any, of its subsystem type.

```
Subsystem_Hierarchies (View) {
  for (C = 0; C < Subsystem_Classes[View]; C++) {
    Class_Hierarchy[C, View] = Get_Class_Hierarchy (C, View);
    if (!Class_Hierarchy[C, View])
      for (T = 0; T < Subsystem_Types[C, View]; T++)
```

```
        for (S = 0; S < Subsystems[C, T, View]; S++)
          Read_Subsystem_Hierarchy (C, T, S, View))
  }
  for (C = 0; C < Subsystem_Classes[View]; C++)
    if (Class_Hierarchy[C, View])
      for (T = 0; T < Subsystem_Types[C, View]; T++)
        for (S = 0; S < Subsystems[C, T, View]; S++)
          Determine_Subsystem_Hierarchy (C, T, S, View))
} /* Subsystem_Hierarchies */
```

A3. Specifying Potentially General Transitions

The variable definitions are as follows:

Transitions[P]: Number of transitions of priority P.

Condition[P, T]: Transition condition T of priority P.

Destination[P, T]: Destination state T of priority P.

Rate[P, T]: Transition rate expression T of priority P.

LP: Lowest priority.

The function definition is as follows:

Condition_Or (I): Returns true if any of the transition conditions of priority I are true.

```
if (Condition_Or(1))
  for (T = 0; T < Transitions[1]; T++)
    if (Condition[1, T])
      tranto Destination[1, T] by Rate[1, T];
else if (Condition_Or(2))
  for (T = 0; T < Transitions[2]; T++)
    if (Condition[2, T])
      tranto Destination[2, T] by Rate[2, T];
⋮
else if (Condition_Or(LP))
  for (T = 0; T < Transitions[LP]; T++)
    if (Condition[LP, T])
      tranto Destination[LP, T] by Rate[LP, T];
```

References

Avižienis, Algirdas; Gilley, George C.; Mathur, Francis P.; Rennels, David A.; Rohr, John A.; and Rubin, David K. 1971: The STAR (Self-Testing and Repairing) Computer: An Investigation of the Theory and Practice of Fault-Tolerant Computer Design. *IEEE Trans. Comput.,* vol. C-20, no. 11, Nov., pp. 1312–1321.

Bavuso, S. J.; Petersen, P. L.; and Rose, D. M. 1984: *CARE III Model Overview and User's Guide.* NASA TM-85810.

Butler, Ricky W. 1986: An Abstract Language for Specifying Markov Reliability Models. *IEEE Trans. Reliab.,* vol. R-35, no. 5, Dec., pp. 595–601.

Butler, Ricky W. 1992: The SURE Approach to Reliability Analysis. *IEEE Trans. Reliab.,* vol. 41, no. 2, June, pp. 210–218.

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

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

Butler, Ricky W.; and Johnson, Sally C. 1990: *The Art of Fault-Tolerant System Reliability Modeling.* NASA TM-102623.

Castillo, Xavier; McConnel, Stephen R.; and Siewiorek, Daniel P. 1982: Derivation and Calibration of a Transient Error Reliability Model. *IEEE Trans. Comput.,* vol. C-31, no. 7, July, pp. 658–671.

Chung, Kai Lai 1967: *Markov Chains—With Stationary Transition Probabilities,* Second ed. Springer-Verlag.

Cohen, Gerald C.; and McCann, Catherine M. 1990: *Reliability Model Generator Specification.* NASA CR-182005.

Cox, D. R.; and Miller, H. D. 1965: *The Theory of Stochastic Processes.* John Wiley & Sons, Inc.

Dugan, Joanne Bechta; Trivedi, Kishor S.; Geist, Robert M.; and Nicola, Victor F. 1984: Extended Stochastic Petri Nets: Applications and Analysis. *Performance '84—Models of Computer System Performance,* E. Gelenbe, ed., Elsevier Science Publ. Co., Inc., pp. 507–519.

Dugan, Joanne Bechta; Trivedi, Kishor S.; Smotherman, Mark K.; and Geist, Robert M. 1986: The Hybrid Automated Reliability Predictor. *AIAA J. Guid., Control & Dyn.,* vol. 9, no. 3, May–June, pp. 319–331.

Goldberg, Jack; Kautz, William H.; Melliar-Smith, P. Michael; Green, Milton W.; Levitt, Karl N.; Schwartz, Richard L.; and Weinstock, Charles B. 1984: *Development and Analysis of the Software Implemented Fault-Tolerance (SIFT) Computer.* NASA CR-172146.

Howell, Sandra V.; Bavuso, Salvatore J.; and Haley, Pamela J. 1990: A Graphical Language for Reliability Model Generation. *Proceedings of 1990 Annual Reliability and Maintainability Symposium,* Inst. of Electrical and Electronics Engineers, Inc., pp. 471–475.

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

Johnson, Sally C. 1988: Reliability Analysis of Large, Complex Systems Using ASSIST. *A Collection of Technical Papers, Part I—AIAA/IEEE 8th Digital Avionics Systems Conference,* Oct., pp. 227–234. (Available as AIAA-88-3898-CP.)

Katzman, James A. 1977: System Architecture for Nonstop Computing. *14th IEEE Computer Society International Conference,* IEEE Catalog No. 77CH1165-OC, IEEE Computer Soc., pp. 77–80.

Kini, Vittal; and Siewiorek, Daniel P. 1982: Automatic Generation of Symbolic Reliability Functions for Processor-Memory-Switch Structures. *IEEE Trans. Comput.,* vol. C-31, no. 8, Aug., pp. 752–771.

Landrault, C.; and Laprie, J.-C. 1978: SURF—A Program for Modeling and Reliability Prediction for Fault-Tolerant Computing Systems. *Information Technology 78: Proceedings of the Third Jerusalem Conference on Information Technology,* Josef Moneta, ed., North-Holland Publ. Co., pp. 17–26.

Laprie, Jean-Claude 1985: Dependable Computing and Fault Tolerance: Concepts and Terminology. *The Fifteenth Annual International Symposium on Fault-Tolerant Computing—Digest of Papers,* IEEE Catalog No. 85CH2143-6, IEEE Computer Soc., pp. 1–11.

Lee, Larry D. 1985: *Reliability Bounds for Fault-Tolerant Systems With Competing Responses to Component Failures.* NASA TP-2409.

Liceaga, Carlos A. 1992: Automatic Specification of Reliability Models for Life-Critical Processor-Memory-Switch Structures. Ph.D. Diss., Carnegie-Mellon Univ., Apr.

Makam, Srinivas V.; and Avižienis, Algirdas 1982: ARIES 81: A Reliability and Life-Cycle Evaluation Tool for Fault-Tolerant Systems. *FTCS 12th Annual International Symposium, Fault-Tolerant Computing—Digest of Papers,* IEEE Catalog No. 82CH1760-8, IEEE Computer Soc., pp. 267–274.

Makam, Srinivas V.; Avižienis, Algirdas; and Grusas, Gintaras: *UCLA ARIES 82 User's Guide.* Rep. No. CSD-82-830 (ONR Contract No. N00014-79C-0866), Univ. of California, Aug. 1982.

McConnel, Stephen Roy 1981: Analysis and Modeling of Transient Errors in Digital Computers. Ph.D. Diss., Carnegie-Mellon Univ.

Moser, Louise; Melliar-Smith, Michael; and Schwartz, Richard 1987: *Design Verification of SIFT.* NASA CR-4097.

Myers, Glenford J. 1979: *The Art of Software Testing.* John Wiley & Sons, Inc.

Palumbo, Daniel L.; and Nicol, David M. 1990: Generation and Analysis of Large Reliability Models. *Proceedings of 9th IEEE/AIAA/NASA Digital Avionics Systems Conference,* Inst. of Electrical and Electronics Engineers, Inc., pp. 350–354.

Ramamoorthy, C. V.; and Bastani, Farokh B. 1982: Software Reliability—Status and Perspectives. *IEEE Trans. Softw. Eng.,* vol. SE-8, no. 4, July, pp. 354–371.

Romanovsky, V. I. (E. Seneta, transl.) 1970: *Discrete Markov Chains.* Walters-Noordhoff Publ. Groningen (Netherlands).

Siewiorek, Daniel P.; Bell, C. Gordon; and Newell, Allen 1982: *Computer Structures: Principles and Examples.* McGraw-Hill, Inc.

Siewiorek, Daniel P.; and Swarz, Robert S. 1992: *Reliable Computer Systems: Design and Evaluation,* 2nd ed. Digital Press.

Stiffler, J. J.; Bryant, L. A.; and Guccione, L. 1979: *CARE III Final Report, Phase I—Volume I.* NASA CR-159122.

Szczur, Martha R. 1990: TAE Plus: Transportable Applications Environment Plus—A User Interface Development Tool for Building X Window-Based Applications. Proceedings of the 4th MIT Annual X Conference.

Trivedi, Kishor S.; and Geist, Robert M. 1981: *A Tutorial on the CARE III Approach to Reliability Modeling.* NASA CR-3488.

U.S. Dep. of Defense 1991: *Reliability Prediction of Electronic Equipment.* MIL-HDBK-217F, Dec. 2. (Supersedes MIL-HDBK-217E, Notice 1, Jan. 2, 1990.)

Vlissides, John M. 1990: *Generalized Graphical Object Editing.* Tech. Rep. CSL-TR-90-427 (NASA Grant NAGW-419), Stanford Univ., June.

White, Allan L. 1986: Reliability Estimation for Reconfigurable Systems With Fast Recovery. *Microelectron. & Reliab.,* vol. 26, no. 6., pp. 1111–1120.

White, Allan L.; and Palumbo, Daniel L. 1990: State Reduction for Semi-Markov Reliability Models. *Annual Reliability and Maintainability Symposium—1990 Proceedings,* IEEE Catalog No. 90CH2804-3, Inst. of Electrical and Electronics Engineers, Inc. pp. 280–285.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*

2. REPORT DATE
July 1993

3. REPORT TYPE AND DATES COVERED
Technical Paper

4. TITLE AND SUBTITLE
Automatic Specification of Reliability Models for Fault-Tolerant Computers

5. FUNDING NUMBERS
WU 505-64-10-07

6. AUTHOR(S)
Carlos A. Liceaga and Daniel P. Siewiorek

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
NASA Langley Research Center
Hampton, VA 23681-0001

8. PERFORMING ORGANIZATION REPORT NUMBER
L-17144

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
National Aeronautics and Space Administration
Washington, DC 20546-0001

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
NASA TP-3301

11. SUPPLEMENTARY NOTES
Liceaga: Langley Research Center, Hampton, VA; Siewiorek: Carnegie Mellon University, Pittsburgh, PA.

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Unclassified–Unlimited

Subject Category 66

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*
The calculation of reliability measures using Markov models is required for life-critical processor-memory-switch structures that have standby redundancy or that are subject to transient or intermittent faults or repair. The task of specifying these models is tedious and prone to human error because of the large number of states and transitions required in any reasonable system. Therefore, model specification is a major analysis bottleneck, and model verification is a major validation problem. The general unfamiliarity of computer architects with Markov modeling techniques further increases the necessity of automating the model specification. Automation requires a general system description language (SDL). For practicality, this SDL should also provide a high level of abstraction and be easy to learn and use. This paper presents the first attempt to define and implement an SDL with those characteristics. A program named Automated Reliability Modeling (ARM) was constructed as a research vehicle. The ARM program uses a graphical interface as its SDL, and it outputs a Markov reliability model specification formulated for direct use by programs that generate and evaluate the model.

14. SUBJECT TERMS
System description language; Computer-aided design; Graphical user interface; Reliability modeling; Markov models; Fault tolerance

15. NUMBER OF PAGES
68

16. PRICE CODE
A04

17. SECURITY CLASSIFICATION OF REPORT
Unclassified

18. SECURITY CLASSIFICATION OF THIS PAGE
Unclassified

19. SECURITY CLASSIFICATION OF ABSTRACT

20. LIMITATION OF ABSTRACT