NASA TM- 89009

NASA Technical Memorandum 89009

NASA-TM-89009 19860021844

Towards Automatic Markov Reliability Modeling of Computer Architectures

Carlos A. Liceaga and Daniel P. Siewiorek

FOR REFERENCE

NOT TO BE TAKEN FROM THIS ROOM

LIBRARY COPY

SEP 10 1986

LANGLEY RESEARCH CENTER
LIBRARY, NASA
HAMPTON, VIRGINIA

August 1986

NASA
National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23665

Summary

The analysis and evaluation of reliability measures using time-varying Markov models has gained in importance for computer architectures that use standby redundancy or can be repaired. The task of generating these models for arbitrary Processor-Memory-Switch (PMS) interconnection structures, however, is tedious and prone to human error due to the large number of states and transitions involved in any reasonable structure. Existing programs that evaluate these models make the following assumptions:

a) The case analysis of success states of the system has been carried out. Such analysis must be done manually.

b) The input to the program is either an intermediate representation (e.g. Fault Tree), or the state transition matrix (STM).

This is the first attempt to (a) identify and analyze the problems involved in the automatic generation of reliability and availability Markov models for arbitrary interconnection structures at the PMS level, and (b) generate and implement solutions to these problems. This work will automate the task of case analysis and generation of the STM in the computation of the reliability and availability of PMS structures. The advantages of such an approach are (a) utility to a larger class of users, not necessarily expert in reliability analysis, and (b) a lower probability of human error in the computation.

A program named ARM (<u>A</u>utomated <u>R</u>eliability <u>M</u>odeling) will be constructed as a research vehicle. ARM will accept as inputs:

a) The interconnection graph of the PMS structure.

b) The behavior of the PMS structure components in terms of their internal communication structure, and their distributions and corresponding parameters of performance and reliability.

c) The groups of redundant components (e.g. processor triads).

d) A succinct statement of the operational requirements on the PMS structure in the form of a modified Boolean expression.

#
N86-31316

The operational requirements in the case of a redundant multiprocessor may be, for example, "two processor triads and two memory triads". The communication structures in the PMS system (e.g. buses) will be considered in addition to the explicitly stated requirements to determine how the interconnection structure affects the system reliability and availability. The output of the ARM program will be the reliability or availability STM. The STM will be formulated for direct use by evaluation programs.

Acknowledgement

The authors are very grateful for the assistance of Larry D. Lee of NASA-LaRC in defining the various types of time-varying Markov models, calculating the number of repetitions simulations required, and developing a useful state space reduction technique. His numerous and detailed comments have greatly improved the clarity of this document.

Table of Contents

List of Figures

List of Tables

1. Introduction

Computer systems are growing in complexity and sophistication as multiprocessor and distributed computer systems are coming into widespread use to achieve higher performance and reliability. This growth is being assisted by the availability of successively more complex building blocks. This trend has increased the importance of fault-tolerance and system reliability as design parameters. Thus the computation 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 the task of 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. With the exception of the ADVISER program, discussed in Section 1.2, existing software tools usually assume an understanding of reliability analysis techniques and therefore are more in the nature of computational aids once the preliminary system decomposition and analysis has been manually achieved. Although ADVISER does not make this assumption it uses combinatorial techniques and is therefore limited in the complexity of systems and fault types it can analyze.

More advanced techniques are required to analyze computer architectures that use standby redundancy, can be repaired, and are susceptible to transient or intermittent faults. One possibility is a time-varying Markov model. The advantages offered by time-varying Markov models are that they are in widespread use among reliability analysts and several programs, discussed in Section 1.2, have been developed to solve them. However time-varying Markov models can not analyze 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 where two faults are present.

This new state would not take into account the time the system already spent reconfiguring from the first fault.

Another possibility is the extended stochastic Petri net (ESPN) described in [Dugan 84]. The advantages offered by the ESPN is that it can analyze concurrent events and model systems at a lower level of detail than time-varying Markov models. The ESPN can be concurrent because its '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 that can simulate the algorithm of the process being modeled. To solve an ESPN analytically it must be converted to a time-varying Markov model. This conversion is not possible if tokens are moving concurrently at independent transition times that are not exponentially distributed, because this makes the process non-Markovian (i.e. the transition probabilities depend on past states). In general an ESPN must be solved by simulation.

Simulations can include any level of detail, and are thus flexible, but many repetitions of the simulation are needed to ensure accuracy. For example, say the probability of failure P is going to be estimated with a relative error no more than 10% within a confidence interval of 95%. The relative error E is defined as:

$$E = \frac{|\hat{P} - P|}{P} \qquad (1.1)$$

where $\hat{P}$ is the estimate of P. $\hat{P}$ is defined as:

$$\hat{P} = F / N \qquad (1.2)$$

where F is the number of failures observed and N is the sample size. Then an expression for N must be found such that:

$$Pr(E \leqq .1) = .95 \qquad (1.3)$$

Substituting (1.1) into (1.3) and multiplying the inequality by P gives:

$$Pr(|\hat{P} - P| \leqq .1P) = .95 \qquad (1.4)$$

Substituting (1.2) into (1.4) and multiplying the inequality by N gives:

$$Pr(|F - NP| \leq .1NP) = .95 \qquad (1.5)$$

Substituting μ for NP in (1.5) gives:

$$Pr(|F - \mu| \leq .1\mu) = .95 \qquad (1.6)$$

The inequality in (1.6) can be expressed as:

$$Pr(.9\mu \leq F \leq 1.1\mu) = .95 \qquad (1.7)$$

If N is large and P is small F is approximately Poisson distributed with mean $\mu = NP$ and (1.7) can be expressed as:

$$\sum_{i=.9\mu}^{1.1\mu} \frac{\mu^i e^{-\mu}}{i!} = .95 \qquad (1.8)$$

Therefore in life critical applications where a probability of failure in the order 10^{-9} is required, approximately 3.8×10^{11} simulation repetitions are necessary! In general those applications require an analytic approach.

It is the intent of this paper to explore the issues in the automatic generation of reliability and availability Markov models for arbitrary interconnection structures at the Processor-Memory-Switch (PMS) level. The result of this effort will be implemented and experimentally validated in the ARM (Automated Reliability Modeling) program which will accept the PMS interconnection structure and a simple set of operational requirements on the structure. The program will attempt to efficiently analyze, using the divide-and-conquer methodology, the various system states based on the interconnection structure and the operational requirements.

The output of the ARM program will be a file containing the reliability or availability state transition matrix. The output format will vary depending on the program to evaluate the state transition matrix. The evaluation programs whose format the user will be able to specify are: SURE, HARP, and ARIES (described in Section 1.2).

The following sections will present a brief background on reliability calculation at the PMS level using time-varying Markov models. Previous work in the generation and evaluation of reliability models is surveyed. The goals for ARM will be stated and compared with those of previous efforts. The final section will present the organization of this paper.

1.1 Background

Present day computer systems can be viewed at varying levels of detail, and therefore so can the process of designing and analyzing them. Four levels were defined by Siewiorek, Bell, and Newell [Siewiorek 82a]. These range from the circuit level, through the logic and programming levels, to the PMS level. The PMS level view of digital systems is one where the primitives are processors, memories, switches, transducers, etc. as opposed to the logic level where the primitives may be gates, registers, multiplexers, etc.

Hardware components are susceptible to permanent, transient, and intermittent faults as discussed in [Siewiorek 82b]. A fault is an erroneous state of hardware or software resulting from a physical change in the hardware or interference from the environment. Permanent or hard faults are continuous and stable, and result from an irreversible physical change. Transient faults result from temporary environmental conditions. Intermittent faults are occasionally present due to unstable hardware, or varying hardware or software states (for example, as a function of load or activity).

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. Siewiorek and Swarz [Siewiorek 82b] discuss the various redundancy techniques, the more relevant ones are defined below.

STATIC REDUNDANCY - In static redundancy faults are masked through a majority vote involving a fixed group of redundant components. Thus, when the masking redundancy is exhausted by component faults, any further faults will cause errors at the output.

DYNAMIC REDUNDANCY - In dynamic redundancy faults are not masked but the faulty components are detected, isolated, and reconfigured out of the system. The faulty components may be replaced by spares if available.

HYBRID REDUNDANCY - In hybrid redundancy faults are masked through a majority vote involving a group of redundant components that is reconfigured when spares are available. Thus, when the redundancy is exhausted by component faults, any further faults will cause errors at the output.

ADAPTIVE VOTING - In 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 and the voter threshold adjusted to reflect a smaller number of components. Thus, when the redundancy is exhausted by component faults, any further faults that occur will cause errors at the output.

ADAPTIVE HYBRID - In adaptive hybrid faults are masked through a majority vote involving a variable group of redundant components that is reconfigured when spares are available. If spares are not available faulty components are reconfigured out of the system by excluding them from the voting process and adjusting the voter threshold. Thus, when the masking redundancy is exhausted by component faults, 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.

For example, a triad is a group of 3 components that use hybrid redundancy to tolerate at least one fault. If a triad recovers from a fault by replacing the faulty component with a spare it can then tolerate a second fault. Recovery is the process of detecting, isolating, and reconfiguring the faulty component out of the system. The fault coverage of a component is the probability that the system can survive a fault in this component and successfully recover. If the system can always recover it has a "perfect" coverage of 1.

Reliability measures are defined in terms of probabilities because the failure processes in hardware components are non-deterministic. Siewiorek and Swarz [Siewiorek 82b] discuss these various measures, the more relevant ones are defined below.

RELIABILITY - The reliability, R(t), of a system as a function of time t is the conditional probability that the system has survived the interval [0, t] given that it was operational at time zero. It is a non-increasing function whose initial value is one.

MTTF - The MTTF (Mean Time To Failure) is the expected time of the first system failure assuming a new (perfect) system at time zero.

AVAILABILITY - The availability, A(t), of a system as a function of time t is the probability that the system is operational at that instant of time t.

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. 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 computation of system life-cycle costs.

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

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

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

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

where λ is the hazard or failure rate. The failure rate is a constant which reflects the reliability of the component and for highly reliable components is usually expressed in failures per million hours. The exponential reliability function is used when the failure rate is time-independent, such as when components do not age. It is often observed that, after a burn-in period, permanent faults in electronic components follow a relatively constant failure rate. The MTTF for the exponential reliability function has the form:

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

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

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

where λ 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 one. 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 greater than one) and in this case the system is not as good as new when repair takes place. Data presented in [McConnel 81] indicates that transient faults follow a decreasing failure rate (alpha less than one).

The failure processes of different components will be assumed to be independent of each other. This assumption is not strictly true, such as when electrical, mechanical, or thermal conditions in one component affect other components in its proximity. However it 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. Most systems can be classified according to their state space and time parameter as

a) discrete-state and discrete-time

b) discrete-state and continuous-time

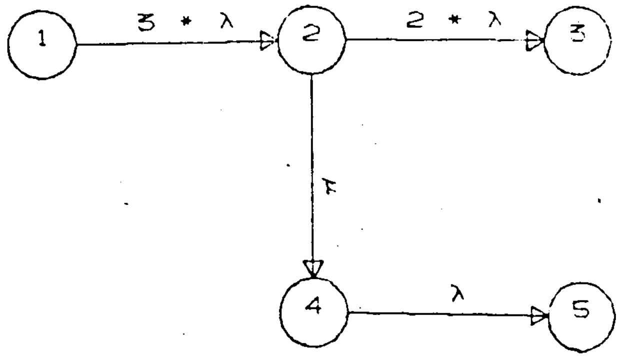
c) continuous-state and discrete-time

d) continuous-state and continuous-time

For a discrete-state system a state transition diagram (STD) may be drawn. The 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 filename of a histogram for more general distributions.

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. 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 1-1. In this model it is assumed that the system has a perfect coverage of 1. The horizontal transitions represent fault arrivals. These follow an exponential distribution and consequently λ represents the constant hazard rate. The coefficients of λ represent the number of working processors are being actively used in the configuration. The vertical transitions represent recovery from a fault. These follow a general distribution and consequently μ represents the filename of the histogram defining the distribution. There is a race between the occurrence of a second fault and the removal of the first. If the second fault wins the race, then system failure occurs. If the removal of the first fault wins the race, then the system reconfigures into a simplex (i.e. 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.


$$\begin{tikzcd} 1 \arrow[r, "3 * \lambda"] & 2 \arrow[r, "2 * \lambda"] \arrow[d, "\mu"] & 3 \\ & 4 \arrow[r, "\lambda"] & 5 \end{tikzcd}$$

Key:

State	Description
1	3 working
2	2 working
3	system failed
4	2 working, uses 1
5	system failed

Figure 1-1: Reliability Graph of a Triad with 1 Spare

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 in the arc from state i to state j.

The terminology used in this paper to denote the various types of time-varying Markov models, and the assumptions they are based on are defined below. The hierarchy of time-varying Markov models is illustrated in Figure 1-2.

TIME-VARYING MARKOV PROCESS - A stochastic process whose future state depends only upon the present state, and not upon the history that led to its present state.

HOMOGENEOUS MARKOV MODEL - A model that uses a pure Markov process whose state transition probabilities are time-independent. For the continuous-time homogeneous Markov process this implies that the state transition times follow an exponential distribution. This model is discussed in [Chung 67] and [Romanovsky 70].

SEMI-MARKOV MODEL - A model that uses a generalization of the pure Markov process whose state transition probabilities depend upon the local time spent in the present state. For the continuous-time semi-Markov process this implies that the state transition times do not follow an exponential distribution, they might follow a Weibull distribution or any other distribution. This model is discussed and applied to computer systems in [White 84].

NON-HOMOGENEOUS MARKOV MODEL - A model that uses a generalization of the pure Markov process whose state transition probabilities depend upon the global time. For the continuous-time non-homogeneous Markov process this implies that the state transition times do not follow an exponential distribution. Often they are assumed to follow a Weibull distribution, but they can follow any other distribution. This model is discussed and applied to computer systems in [Trivedi 81].

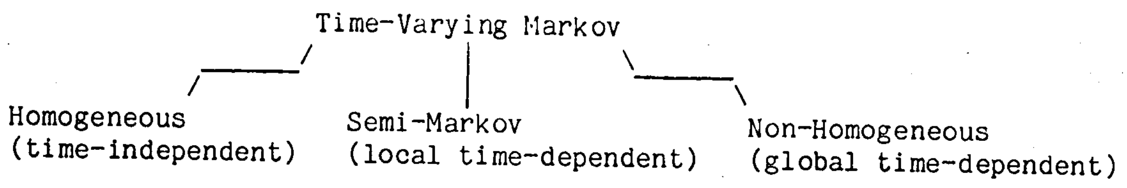


Figure 1-2: Hierarchy of Time-Varying Markov Models

The probability of being in a particular state for a discrete-state and continuous-time Markov model can be expressed with a differential equation. The set of simultaneous differential equations that 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. To obtain the exact probability of reaching a state through a particular path of transitions requires the solution of a multiple integral, where each integral represents the probability of making one of the transitions in the path. Often the integrals are approximated using numerical integration techniques [Stiffler 79]. An alternative method is to approximate the continuous-time process with discrete-time equivalents [Siewiorek 82b]. The major difficulty with the second method is that many transition rates that are effectively zero in the continuous-time process assume small but nonzero probabilities in a discrete-time process.

1.2 Previous Work

There are several programs that use time-varying Markov models to evaluate the reliability and/or availability of systems that use standby redundancy or can be repaired, and are susceptible to hard, transient, and intermittent faults, such as CARE III, ARIES, SURE, SURF, and HARP. All these programs can evaluate both the reliability and availability of a system, except for CARE III which can only evaluate the reliability. Except for CARE III, they all have as one of the system specification methods the state transition matrix.

CARE III (Computer-Aided Reliability Estimation), described in [Bavuso 84], can evaluate the reliability of systems that use reconfiguration to tolerate component faults but do not repair the faulty components. It uses a behavioral decomposition/aggregation solution technique described in [Trivedi 81]. This technique assumes that the fault-occurrence behavior is composed of relatively infrequent events while the fault-handling behavior is composed of relatively frequent 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 non-homogeneous Markov model that can use exponential and Weibull distributions. The fault handling behavior is reflected by parameters in the aggregate non-homogeneous Markov model. Numerical integration techniques are used to solve these time-varying Markov models. The fault-occurrence behavior is specified using extended fault trees, which are automatically converted to the non-homogeneous Markov model. The fault-handling behavior is specified by providing the transition parameters of the fixed semi-Markov model. Therefore state transition matrices can not be accepted as input directly. CARE III was developed at Raytheon, it is written in FORTRAN 77, and runs on a Cyber or a VAX.

ARIES (Automated Reliability Interactive Estimation System), described in [Makam 82], 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 are either active or serve as spares. It uses a matrix transformation solution technique that assumes distinct eigenvalues for the state transition matrix. It was developed at UCLA and runs on a VAX.

Butler [Butler 84] describes a program named SURE (Semi-Markov Unreliability Range Evaluator) which evaluates the unreliability upper and lower bounds of semi-Markov models. It uses new mathematical theorems proven in [White 84] and [Lee 85]. These theorems provide a means of 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 of the system reaching 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 general transitions describe the recovery process. Faults can be modeled as permanent, transient, or intermittent. Its only input method is the state transition matrix. SURE was developed at NASA's Langley Research Center, it is written in VAX-11 Pascal, and runs under VAX/VMS.

SURF, described in [Landrault 78], can solve semi-Markov models that use exponential distributions or non-exponential distributions that are related to the exponential (e.g. Gamma, Erlang, etc.). The method of stages [Cox 68] 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. Written in PL/I, it runs on a IBM System/370 at the IBM research facility in Yorktown Heights, New York. SURF was developed in Toulouse, France.

For HARP (Hybrid Automated Reliability Predictor), described in [Trivedi 85], the state transition probabilities can have exponential, uniform, Weibull, or general (i.e. histogram list must be provided) distributions. If the state transition matrix is given by the user HARP can only evaluate the availability of systems with constant repair rates. HARP has several additional methods of specifying the fault-occurrence behavior (e.g. fault trees), all of which are automatically converted to a non-homogeneous Markov model. The fault-handling behavior can also be specified by providing the transition parameters of one of several models. The fault-handling models available are: an extended stochastic Petri net, the CARE III model, the ARIES model, and the SURE model. It uses the same behavioral decomposition/aggregation solution technique as CARE III, but the various models are solved in a hybrid fashion. Time-varying Markov models are solved analytically using numerical integration techniques, and extended stochastic Petri nets are solved by simulation. It is written in FORTRAN 77 and runs on a VAX. It is still under development at Duke University and Clemson University.

An abstract specification language for Markov reliability models was described by Butler [Butler 85]. The language has statements to specify (a) the state space by defining the state variables and their range, (b) the start state by the initial values of the state variables, (c) the death states by a Boolean expression of the state variables, and (d) the state transitions by a set of if-then rules that define the possible transitions, their rates, and their destination states all in terms of the state variables. This language has been implemented in the ASSIST program to generate Markov reliability models in the SURE input language. The algorithm used in ASSIST to generate the model will be applicable with modifications to ARM as described in Section 3. ASSIST was developed at NASA's Langley Research Center, it is written in VAX-11 Pascal, and runs under VAX/VMS.

Kini [Kini 81] describes a program named ADVISER (Advanced Interactive Symbolic Evaluator of Reliability) which automatically generates symbolic reliability functions for PMS structures. Its assumptions are: (a) all faults are permanent and stochastically independent, (b) the PMS system has a perfect coverage, and (c) failed components are not repaired and returned to a non-faulty state. Its 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 methods used in ADVISER for detecting PMS graph symmetries and tree structures will also be applicable with modifications to ARM as described in Section 3. ADVISER was developed at CMU, it is written in BLISS, and runs on a PDP-10.

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 (a) it is the highest level view of digital systems and therefore the easiest to specify, and (b) it is well known to computer architects. The time-varying Markov model technique of reliability and availability analysis was selected because (a) it is powerful enough to accurately analyze most situations except for concurrent events, and (b) it is in widespread use among reliability analysts and several evaluation programs have been developed.

Previous efforts have been limited in one of two ways. Most 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.

1.4 Organization

The system description required to generate a reliability or availability Markov model is described in Section 2. The problems involved in the automatic generation of reliability and availability Markov models are discussed in Section 3. Examples of automatically generated Markov reliability models are presented in Section 4. A summary of the research and a plan for its accomplishment are presented in Section 5. The algorithms used by the ARM program are described in Appendix A.

2. System Description

It is important to have a general system description method that will accommodate new fault-tolerant techniques and system designs. This section presents the system description method currently envisioned for the ARM program. The generality of this method needs to be investigated to correct any deficiencies.

When calculating a reliability measure for an arbitrary system of components, four items of information are necessary, namely:

a) The reliability behavior of the system components (Section 2.1).

b) The fault tolerant function of individual components or groups of components in the system (Sections 2.2 and 2.3).

c) The communication paths that components in the system may use, and which are the components that need to exchange information (Sections 2.4 to 2.6).

d) The operational requirements placed on the system and its subsystems (Sections 2.7 and 2.8).

Item (b) is the only one that is not necessary for some systems. The ARM program will use eight input categories to obtain these items of information for any arbitrary system. For some systems only three ARM input categories are required to convey the information in (a), (c), and (d). These minimum ARM input categories are: a reliability description of the component types (Section 2.1), the interconnection structure (Section 2.4), and the system requirements (Section 2.8).

The following sections will discuss the purpose and necessity of the input categories that provide the ARM program with these items of information. The final section will give an example of how a multiprocessor system can be specified using all the input categories of the ARM program.

2.1 Component Types

The first input category is a list describing the types of components in the PMS structure. Components of the same type are assumed to be identical in function and reliability. The concept of component types is natural and reduces the system specification burden. The other alternative would be to specify the characteristics of each particular component.

Each type declaration will specify the coverage probability, and the rates of the various failure, recovery, and repair processes for components of that type. Rates will be specified by a probability distribution and the parameters of that distribution. A rate may follow more than one distribution as a function of the system state. The function of the system state that determines the distribution of a rate will be in the form of a modified Boolean expression as defined in Section 2.8. A distribution can be exponential, Weibull, or general (i.e. a histogram must be provided).

The nine classes of information a type declaration can contain are defined below. Each type declaration must contain at least the first two classes. Figure 2-1 illustrates how the first seven classes are used in reliability models.

TYPE - The first class is the name of the component type.

HARD - The second class is the hard failure rate λ. Hard faults are assumed to be caused by permanently damaged components that continuously produce errors when exercised.

TRANSIENTS - The third class consists of two rates. One is the transient failure rate τ. Another is the transient duration rate δ, that is the rate at which the transient stops producing errors. It is assumed transients are not caused by or produce any permanent damage to the components.

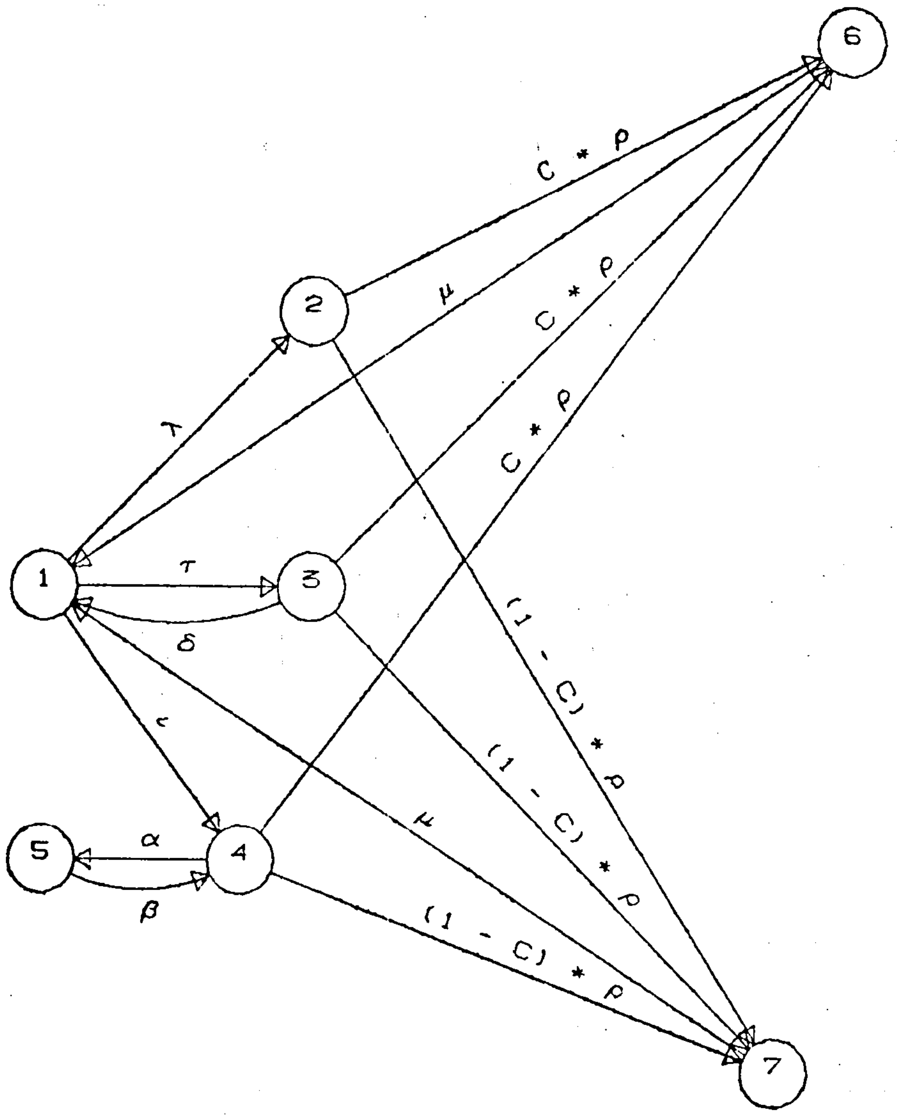
INTERMITTENTS - The fourth class consists of three rates. One is the intermittent failure rate ι. Another is the intermittent benign rate β, that is the rate at which an intermittent becomes benign or stops producing errors. Last is the active rate α, that is the rate at which an intermittent fault that had stopped producing errors becomes active and starts producing errors once more. It is assumed intermittents are caused by permanently damaged components.

COVERAGE - The fifth class is the fault coverage C expressed as the probability that the system can survive a fault in this type of component and successfully recover. This probability defaults to 1. Coverage has a great impact on the reliability of a system. Therefore it must be estimated very accurately using one or more of the following: simulation, analytic methods, or fault injection experiments.

REPAIR - The sixth class is the repair rate μ, that is the rate at which components of this type are repaired and returned to service. Only if the repair rate is specified can the availability of the system be modeled.

RECOVERY - The seventh class is the recovery rate ρ, that is the rate at which the system can detect, isolate, and reconfigure from faults in components of this type by using a shadow (a hot or powered up spare that is imitating the active component).

SHADOW - The eighth class is the shadow activation rate σ, that is the rate at which the system can provide a shadow. A shadow is a spare component that is performing all the functions of a redundant group of active components with the exception that its output is not being used. The purpose of shadows is to increase the recovery rate. An example of this is the rate at which a memory module can be reloaded to shadow a different memory triad. A shadow can be provided by changing the redundant group that a hot or powered up spare is imitating, or by powering up and activating a cold or unpowered spare.



Key: State	Description
1	no faults
2	hard fault
3	transient fault
4	active intermittent fault
5	benign intermittent fault
6	correct fault detection, isolation, and reconfiguration
7	incorrect fault detection, isolation, and reconfiguration

Figure 2-1: Use of Component Type Information in Reliability Models

DEGRADATION - The ninth class is the degradation rate θ. That is the rate at which the system can gracefully degrade by eliminating one redundant group of components which are all of this type. A group is a set of components performing the same operations such that the correct output can be selected using diagnostics or majority vote. Degradation is necessary when a group component fails, there are no spares to replace it, and the number of these groups is above the minimum requirements for the system. This is done because a group with a failed component has a greater probability of failure (fewer group components need to fail for the group not to meet its minimum requirements), and if a group fails and there is no watchdog timer (defined in Section 2.3) the system fails.

2.2 Redundant Groups

The second input category is a list that specifies any redundant group of components in the system. A group is a set of components performing the same operations such that the correct output can be selected using diagnostics or majority vote. Each group declaration will contain the maximum number of groups of this type, the group name, the requirements, the type of components in the group, and if using adaptive voting the name of the adapted group and the adaptive rate. The adapted group is the group with the adjusted voter threshold. The adaptive rate corresponds to the time involved in changing the voting threshold.

Currently the redundancy technique used for a component is specified by three things. One is whether it is part of a redundant group or not. The other two are whether its recovery and adaptive rates are zero or not. Table 2-1 shows how each redundancy technique is specified. This method of redundancy technique specification must be extended so systems with new redundancy techniques can be described.

The semantics for this input category are the following. When a component in a group using hybrid or adaptive hybrid redundancy fails, there are no spares to replace it, and the number of these groups is above the minimum requirements for the system, then the system gracefully degrades by eliminating the group. If the number of these groups is not above the minimum requirements for the system and the group uses adaptive hybrid redundancy, then the system reconfigures the faulty component out of the voting process.

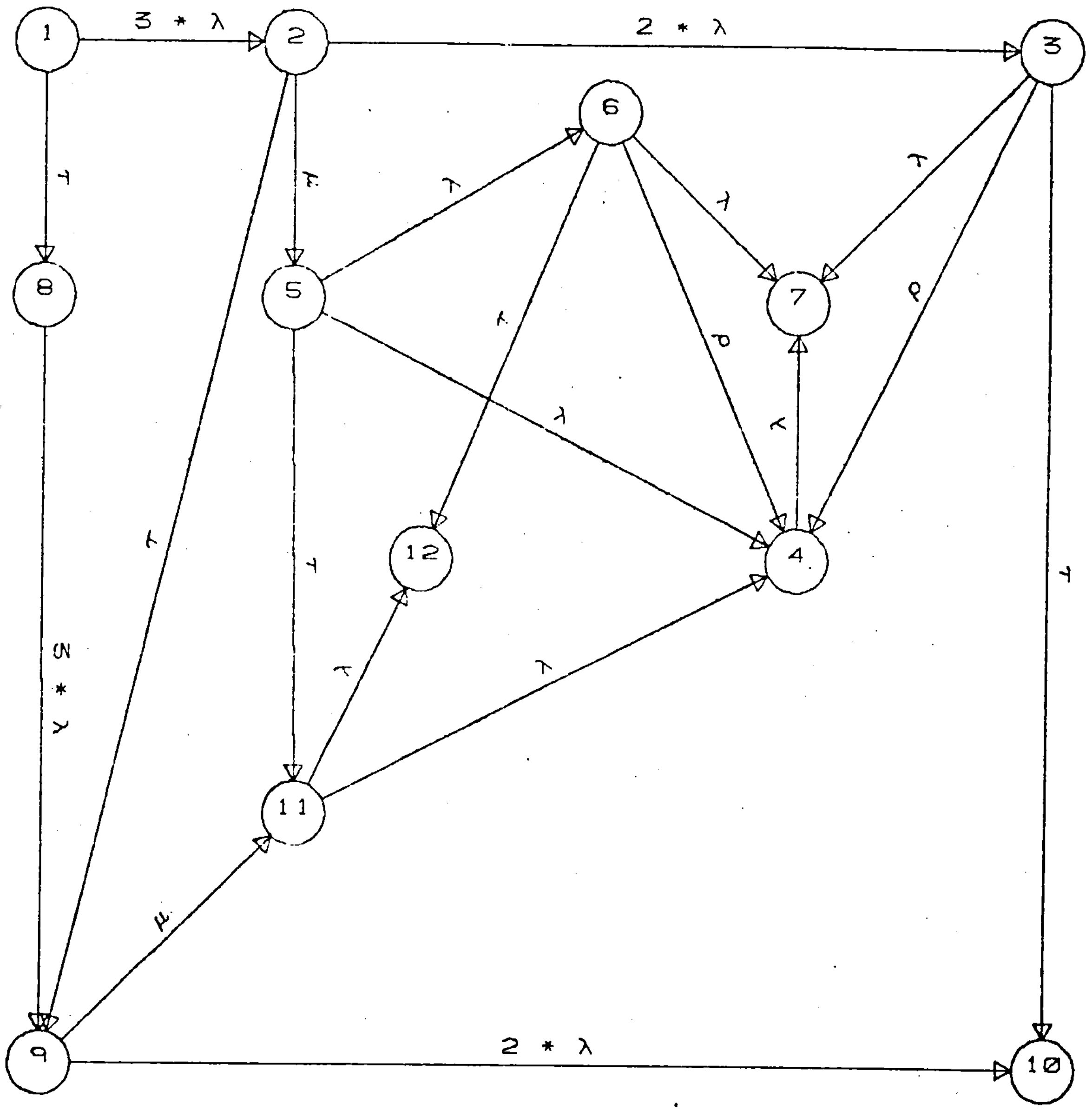
If there are shadows, then they are assumed to be evenly distributed among the groups. If a group has to be able to transmit to another group, then each component of the transmitting group has to be able to transmit to all the components of the receiving group. The reason for the latter is so each component of the receiving group can do an independent majority vote on the information from the transmitting group.

	REDUNDANT GROUP	RECOVERY RATE	ADAPTIVE RATE
STATIC REDUNDANCY	yes	zero	zero
DYNAMIC REDUNDANCY	no	nonzero	zero
HYBRID REDUNDANCY	yes	nonzero	zero
ADAPTIVE VOTING	yes	zero	nonzero
ADAPTIVE HYBRID	yes	nonzero	nonzero

Table 2-1: Redundancy Technique Specification

2.3 System Watchdog Timers

The third input category is a list that specifies which (if any) component type or group of components acts as a watchdog timer for the system, and the rate at which the watchdog can restart the system. A watchdog is assumed to have a timer that must be reset before it runs out or the watchdog will restart the system. A watchdog decreases the probability that multiple faults in a redundant group of components will cause system failure.



Key: State	Description	State	Description
1	3 working	7	system failed
2	2 working	8	watchdog failed
3	system crashed	9	2 working, no watchdog
4	1 working	10	system failed
5	2 working, uses 1	11	2 working, uses 1, no watchdog
6	system crashed	12	system failed

Figure 2-2: Reliability Graph of a Triad with a Watchdog

The semantics for this input category are the following. If there is no watchdog, and any group fails, then the system fails. If the watchdog fails, and any group fails, then the system fails. In other words, there has to be a watchdog for the system to survive a group failure.

Adding a watchdog timer modifies the system model by preventing some states from being failure states and by creating new states. For example, if a watchdog with failure rate τ and system restart rate ρ is added to a triad the system model changes from the one in Figure 1-1 to the one in Figure 2-2. In this new model it is assumed that the system has a perfect coverage of 1 and the failure of the watchdog will not cause system failure. The watchdog prevents a system crash caused by the failure of two processing elements from causing system failure by restarting the system as a simplex without spares.

2.4 PMS Structure

The fourth input category is an interconnection list of the PMS structure. It is assumed that critical components which are required for the system to be operational must be able to communicate. The main purpose of the interconnection list is to analyze which component failures will prevent communication between critical components and therefore cause system failure.

The interconnection list can also be used to detect which substructures in the PMS graph are symmetrical in their component types and neighboring components. Symmetrical substructures are assumed to be identical in function and reliability. Therefore the reliability models of symmetrical substructures are identical and only have to be generated once. These models can then be duplicated and merged to obtain the reliability model of the system.

Each component will have an interconnection declaration that specifies its type and neighboring components. Since the PMS graph is non-directed it is possible to completely specify an arc by its occurrence in one interconnection declaration. However, it will be noted that each arc must occur on two interconnection declarations. The purpose of this redundancy is twofold. Firstly, inconsistencies can be detected thus making the system specification less likely to contain errors. Secondly, a reader of a system specification can more easily comprehend the structure if the connection is made quite explicit with two-way links.

Although not within the scope of the current work, the system specification could be further eased by a graphics based user friendly interface. The interconnection list would then be provided by an input interface that would accept a graphic description of the PMS structure. Since this is not part of the current research and ARM could easily accept its input from an interface program, this interface could be generated independently by support personnel using tools such as the Future Net program for the IBM PC. Such a graphics interface already exists for the PERQ personal work stations at CMU.

2.5 Intracomponent Port Connections

The fifth input category is a list specifying the internal port connectivity of some components and/or component types. The purpose of the internal port connectivity is to analyze which component failures will prevent communication between critical components and therefore cause system failure. This information is needed to prevent the reliability modeling program from assuming incorrect communication paths through intermediate components to other components. Not taking this behavior into account would lead to an optimistic evaluation of the system reliability.

This input category is needed because it is impossible for a reliability modeling program to have this knowledge for all the component types that will be designed. Even if the PMS graph where modified to be a directed graph, the program would still need to know if information passed from A to B can be passed from B to C.

If not specified, the default is for every port of a component to be connected bidirectionally to all other ports of the component. If the internal port connectivity is specified, then for that component or component type all port connections and their direction must be made explicit. Each connection declaration contains the following parameters:

VERTEX	The specific components or component type whose port connections are being specified.
TRANSMITTER	A transmitter port of the VERTEX. It is specified by the component or component type connected to it.
RECEIVER	A port that receives from the previous transmitter port. It is specified by the component or component type connected to it.

2.6 Intra Component-Type Communication

The majority of components of like type are passive and do not need to communicate. Examples of passive components are memories, buses, and input/output transducers. Active or self-talking components need to exchange information amongst each other. Examples of active 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.

The sixth input category is a list specifying the component types for which communication between components of like type is necessary. The purpose of the intra component type communication list is to analyze which component failures will prevent communication between critical components that need to exchange information and therefore cause system failure. This information is needed to prevent the reliability modeling program 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.

2.7 Component Clustering

The seventh input category is a list specifying which (if any) components form clusters, that is subsystems with their own separate requirements. If the cluster requirements are not met all the cluster components fail but the system may continue to operate depending on the system requirements. The purpose of clusters is to represent the dependencies that sometimes exist between components. Each cluster declaration will contain the name of the cluster, its components, and its requirements in the form of a modified Boolean expression as defined in Section 2.8.

2.8 System Requirements

The eighth input category is a succinct statement of the minimum set of critical component types and/or component groups which are required for the system to be operational. Together they constitute a minimum critical resource set (MCRS). The set is minimum in the sense that the system may only function if a MCRS of components are functional (depending on the status of other components in the structure). In other words, the success of an MCRS is a necessary, though not sufficient, condition for system success.

The MCRS will be defined using a modified Boolean expression. The simple grammar of requirements is shown in the traditional Backus-Naur form in Figure 2-3.

```
<requirements> ::= <conjunction> | <conjunction> OR <requirements>

<conjunction>  ::= <atom> | <atom> AND <conjunction> | (<requirements>)

<atom>         ::= <integer> OF <type> | <integer> OF <group>
```

Figure 2-3: Grammar of Requirements

2.9 Example

In this example a multiprocessor system is described using ARM's tabular format in Table 2-2. Failure rates are assumed to be specified in failures per million hours, all other rates are assumed to be on a per hour basis. All rates default to zero and are assumed to follow a single exponential distribution unless otherwise indicated. A multiple distribution rate is specified with a 'M' followed by the name of the file containing the necessary discriminating function and distribution specifications. For the exponential distribution only its constant rate is given. The Weibull distribution is specified with a 'W' followed by the scale and shape parameters. A general distribution is specified with a 'G' followed by the name of the file containing the necessary histogram.

The first component type described in Table 2-2 is a processor P with the following characteristics:

hard failure rate:	λ = 200 failures per million hours
transient failure rate:	τ = 10000 failures per million hours
transient benign rate:	δ = 3600 per hour
intermittent failure rate:	ι = 10000 failures per million hours
intermittent benign rate:	β = 3600 per hour
intermittent active rate:	α = 360 per hour
coverage probability:	C = 1
repair rate:	μ = Weibull distribution of scale=1 and shape=1.1
recovery rate:	ρ = multiple rates defined in the file RECP
shadow rate:	σ = general distribution defined in the file SHADP
degradation rate:	θ = general distribution defined in the file DEGP

The PMS diagram of the multiprocessor is shown if Figure 2-4. The multiprocessor has 10 LRU (Line Replaceable Units) clusters, LRU.1 to LRU.10. LRU.i has a processor P.i, a memory M.i, and a watch dog timer

Component Types (Section 2.1):

TYPE	HARD λ	TRANSIENT (τ, δ)	INTERMITTENT (ι, β, α)	COVERAGE C	REPAIR μ	RECOVERY ρ	SHADOW σ	DEGRADATION θ
P	200	(10000, 3600)	(20, 3600, 360)	1	W 1 1.1	M RECP	G SHADP	G DEGP
M	210	(10500, 3600)	(21, 3600, 360)	1	W 1 1.1	M RECM	G SHADM	G DEGM
WT	50	(2500, 3600)	(5, 3600, 360)	1	W 1 1.1	M RECW	G SHADW	
B	10	(500, 3600)	(1, 3600, 360)	1	W 1 1.1	M RECB	G SHADB	
WB	10	(500, 3600)	(1, 3600, 360)	1	W 1 1.1	M RECWB	G SHADWB	

Redundant Groups (Section 2.2):

SIZE	GROUPNAME	REQUIREMENTS	TYPE	ADOPTS	ADAPTATION
3	PTriad	2 OF 3	P	PSimplex	G ADAPTP
1	PSimplex	1 OF 1	P		
2	MTriad	2 OF 3	M	MSimplex	G ADAPTM
1	MSimplex	1 OF 1	M		
1	WTriad	2 OF 3	WT	WSimplex	G ADAPTW
1	WSimplex	1 OF 1	WT		
1	BTriad	2 OF 3	B		
1	WBTriad	2 OF 3	WB		

System Watchdog Timers (Section 2.3): WTriad

PMS Structure (Section 2.4):

COMPONENT	TYPE	NEIGHBOR-COMPONENTS
P.1-10	P	B.1-5, WB.1-5
M.1-10	M	B.1-5, WB.1-5
WT.1-10	WT	B.1-5, WB.1-5
B.1-5	B	P.1-10, M.1-10, WT.1-5
WB.1-5	WB	P.1-10, M.1-10, WT.1-5

Intracomponent Port Connections (Section 2.5):

VERTEX	TRANSMITER	RECEIVER
B	P	M
B	P	WT
B	M	P
WB	WT	P
WB	WT	M

Intra Component-Type Communicators: (Section 2.6): P

Component Clusters (Section 2.7):

CLUSTERNAME	COMPONENTS	REQUIREMENTS
LRU.1-10	P.i, M.i, WT.i	1 OF M.i

System Requirements (Section 2.8):
(1 OF PTriad OR 1 OF PSimplex) AND (1 OF MTriad OR 1 OF MSimplex)

Table 2-2: Multiprocessor System Description Example

WT.i, and for any of its components to be available M.i must be working properly. Components of the same type are grouped into 2 out of 3 triad subsystems.

The system uses adaptive hybrid redundancy so that if a component other than a bus fails and there is only one triad without spares of that component type, then it reconfigures the two remaining components into a simplex (a single component emulating a triad) with a spare. The system must have a minimum of 1 processor triad or simplex, and 1 memory triad or simplex to be operational.

Processor and memory triads transmit on a bus triad formed out of 5 buses, B.1 to B.5. A processor triad can transmit to any kind of triad including another processor triad. A memory triad can only transmit to processor triads. The watchdog triad transmits on another bus triad formed out of 5 buses, WB.1 to WB.5. The watchdog triad can only transmit to processor and memory triads.

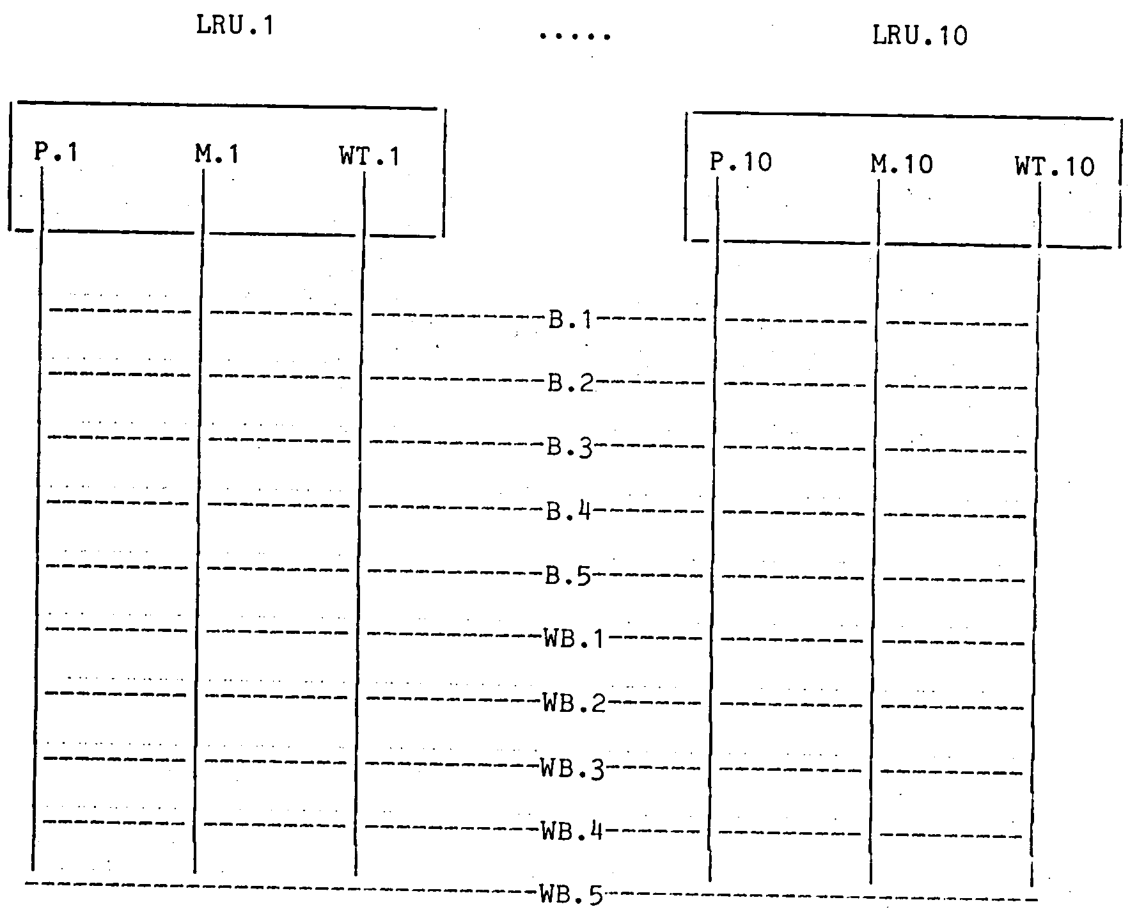


Figure 2-4: PMS Diagram of Multiprocessor Described in Table 2-2

3. Automated Reliability Modeling Considerations

The ARM program will attempt to efficiently generate the system reliability model based on the interconnection structure and the operational requirements. The divide-and-conquer methodology was selected to increase the computational efficiency and reduce the program development complexity. The steps the ARM program is going to follow in generating reliability models are shown in Table 3-1.

1) Interface with user and obtain system description.
2) Detect symmetries in the PMS graph.
3) Segment the PMS graph.
4) Identify the PMS system success and failure states based on the operational requirements.
5) Generate the models for the PMS graph segments.
6) Merge the models for the PMS graph segments.
7) Reduce the state space of the resulting model.
8) Format and output the state transition matrix of the model.

Table 3-1: Automated Reliability Modeling Steps

Steps 2 and 3 of Table 3-1 have been implemented using algorithms derived from those presented in Kini's dissertation [Kini 81] because they are mature, well documented, and simple. The major research effort will be the identification, analysis, and solution of the fundamental problems in each of steps 1, and 4 through 7 of Table 3-1. The research will also include the development of efficient algorithms, and methods to theoretically and experimentally validate the algorithms.

The feasibility of the algorithms developed depends on their efficiency due to the large number of states and transitions involved in any reasonable structure. The validity of these algorithms is particularly important for life critical applications where a probability of failure in the order 10^{-9} is required.

The following sections will discuss the purpose and necessity of steps 2 through 7. Progress already made in identifying and analyzing the problems involved, and developing and implementing algorithms to solve them is also presented.

3.1 Detection of Symmetry in the PMS Graph

Substructures in the PMS graph G will be considered symmetric if they are isomorphic and the corresponding vertices of the two graphs have identical component type labels. Symmetrical substructures will be assumed to be identical in function and reliability. Therefore the reliability models of symmetrical substructures are identical. The purpose of detecting symmetrical substructures is to avoid needless duplication of effort by generating their reliability model only once. These models will then be duplicated and merged to obtain the reliability model of the system.

The symmetry detection algorithm is shown in Appendix A.1. It 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 it has. Two vertices are neighbors if they are interconnected.

The algorithm requires three steps to partition the vertex set of a labelled 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. The third step attempts to partition based on the number of neighbors 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, and therefore other partitions may become necessary. In the worst case this repetition will stop when each equivalence class has a single element.

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 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 non-directed graph without self-loops, which was taken to be the model for G, G' may have vertices which have self-loops. This would be the result of a case in which 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 one. 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.

3.2 Segmentation of the PMS Graph

The purpose of segmenting the PMS graph is to follow the divide-and-conquer methodology. The segmenting proceeds by searching for what are termed Pendant Tree Subgraphs (PTS). These are maximal trees, that is they are not part of another tree. In these tree subgraphs the simple path between any pair of vertices is the only path between those vertices in the overall graph, in other words there are no cycles. It is common to find PTS's in most PMS structures. In particular input/output subsystems typically assume this character.

If the PMS interconnection graph G is not a PTS and all its PTS's, excluding their roots, are removed then the remaining vertices and arcs form a subgraph of G that is not tree-connected. This will be referred to as the Kernel. The root of each PTS has dual status as member of the PTS as well as the Kernel. The PTS's along with the Kernel form a natural set of segments of G on the basis of which the reliability computation task may be divided.

The segmentation algorithm is shown in Appendix A.2. It discovers the PTS's in a given PMS structure by collecting those leaf vertices of G' which represent classes of leaf vertices of G (step 1). These "germinal trees" are then "grown" upward towards the root by adding on neighboring vertices of these leaves (step 2), and merging the germinal trees that overlap at their roots (step 3). Steps 2 and 3 continue until no more adding of vertices or merging of trees is possible. At this point a set of tree subgraphs of G' have been generated. Depending on the number of vertices of G represented by the root, each of these trees in G' may represent one PTS of G or a set of PTS's. In the latter instance all PTS's in the set will be symmetric.

There are three "stopping conditions" under which a tree is not capable of further growth, due to the fact that cycles would be formed and it would no longer be a tree. The first condition is when the root of the tree is a neighbor to itself. The second condition is when the root of the tree has a single neighbor, which is not already in that tree, with a connection density greater than one. The third condition is when the tree has been merged with another tree that meets one of the previous conditions.

3.3 Identification of Success and Failure States

Depending on whether the system is operational or not the states in the reliability model are termed success states or failure states respectively. The identification of success and failure states is essential during the generation of the reliability model because they assume a very different form. Success states must have transitions to other states, because the system must be able to reach a failure state through some sequence of transitions. Failure states are trapping states and therefore can not have any transitions to other states.

The identification of failure states may also prevent some unnecessary generation of failure states. The reason some failure states are not needed is that the only way the system can arrive at them is by being in another failure state.

For example, consider a system that requires 2 out of 3 processors to be operational. For that system the failure state where all three processors have failed does not have to be generated. The reason for this is that the only way the system can arrive at that state is by being in another failure state where two processors have failed.

An algorithm to identify success and failure states has already been developed and implemented. This algorithm is shown in Appendix A.3. It traverses the system requirements parse tree searching for some way in which a system state can satisfy the requirements. The system state is assumed to include those components that are not operational because they do not have communication paths, due to the failure of other components. The Boolean expression of requirements is assumed to have been transformed into a sum-of-products form so that it does not contain any parenthesis.

The parse tree of a sum-of-products Boolean expression only has three levels. The bottom level represents atomic requirements such as "2 of processors". The intermediate level represents pure conjunctive requirements, that is an AND expression of atomic requirements. The top level represents the sum-of-products expression of the system requirements, that is an OR expression of pure conjunctive requirements.

For example, consider a system that can operate with either one processor and two memories, or with one processor, one disk, and one memory. For readability the symbol $\psi(N,X)$ will represent the atomic requirement "N of X". The sum-of-products expression of the system requirements is

$$\psi(1,P) \text{ AND } \psi(2,M) \text{ OR } \psi(1,P) \text{ AND } \psi(1,D) \text{ AND } \psi(1,M) \quad (3.1)$$

The parse tree of such an expression is shown in Figure 3-1.

The algorithm is a Boolean function that takes a state as an argument and returns true if it is a success state. The algorithm works at three levels that correspond to the levels of the parse tree. At the first level it will return true if any conjunctive requirement, in the sum-of-products expression of system requirements, is meet. At the second level it will return true if all atomic requirements in a conjunction are meet. The third level determines which atomic requirements are meet.

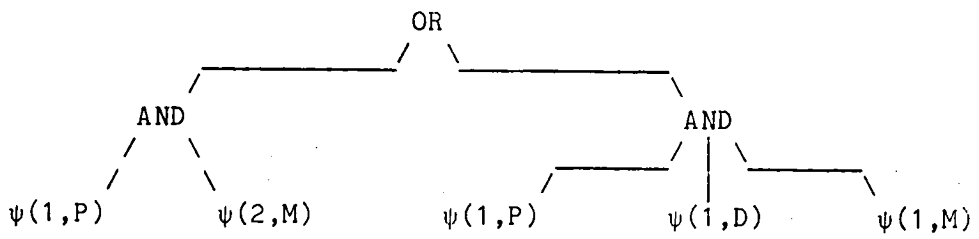


Figure 3-1: Parse tree of requirement expression (3.1)

3.4 Generation of Models for PMS Graph Segments

The generation of the system reliability model will also follow the divide-and-conquer methodology. For that purpose, the states and transitions corresponding to the different segments of the equivalence class graph G', will be separately generated and then merged to produce the system reliability model. The generation of states and transitions in the model will be implemented using algorithms derived from the model generation algorithm presented in [Butler 85].

An algorithm for what are termed minimal subtrees of PTS's has already been developed and implemented. This algorithm is shown in Appendix A.4. Minimal subtrees of PTS's are those that are below the minimum system requirements or meet them exactly. When the root of a minimal subtree of a PTS fails all the nodes in that minimal subtree fail because none of the subtrees within it, which become isolated from other nodes in the graph, can meet the system requirements by itself.

This algorithm is limited to hard faults in non-redundant and non-repairable minimal subtrees. The steps the algorithm follows in generating the minimal subtree models are shown in Table 3-2.

The minimal subtree model generation algorithm must be extended to redundant and repairable minimal subtrees which are susceptible to transient and intermittent faults. Two more algorithms must be developed to generate the system reliability model. The first algorithm will generate a model for those nodes of a PTS that are not in a minimal subtree, and merge it with the minimal subtree models to produce the PTS model. The second algorithm will generate a model for the kernel and merge it with the PTS models to produce the system reliability model.

1) Initialize the set of new states New_Set to the start state.

2) While the New_Set is not empty, get a state out of the New_Set until a success state is found.

3) For every equivalence class node in the minimal subtree, if more components of this class can fail then generate the transitions out of the success state.

4) For every transition generated:

    a) If the destination state is new then add it to the New_Set.

    b) Add the transition to the model by obtaining the two factors whose product is the transition's rate: the number of working components in the class whose failure is described by the transition, and their failure rate.

Table 3-2: Minimal Subtree Modeling Steps

3.5 Merging of Models for PMS Graph Segments

For the purpose of following the divide-and-conquer methodology, the models of the segments of the equivalence class graph G' will be merged to generate the PTS models and also the system reliability model. When two models with N and M states are merged the resulting model has at most NM states. All the states in the original models and their incoming transitions appear in the resulting model along with new ones. Table 3-3 shows the steps that must be followed to merge two models. Algorithms that follow these steps must be developed and implemented to generate the PTS models and also the system reliability model.

1) Retain only one of the two identical start states.
2) Retain all the other original states, which amount to N + M - 2 states.
3) Produce at most NM - N - M + 1 new states by combining each original state in one model with all the original states in the other model, except for the start states.

Table 3-3: Two Model Merging Steps

3.6 Reduction of the State Space

The use of time-varying Markov models to analyze complex systems runs into three problems when the state space becomes extremely large. First, the models become intractable for any human, but this can be alleviated by the use of computer aided modeling and evaluation tools such as the ARM program and others already discussed. Second, the computational cost of evaluating the model may become prohibitive. Third, the evaluation of the model using certain computer systems may become impossible due to their memory space limitations. For the purpose of alleviating the last two problems the user of the ARM program will have the option of applying a state space reduction technique to the system reliability model.

The number of states can be reduced by merging them into subsets and computing the equivalent transition rates between the subsets [Singh 72]. The equivalent transition rate between subsets A and B is

$$\lambda_{AB} = \sum_{j \varepsilon B} \lambda_{ij} \quad \text{for any } i \varepsilon A \tag{3.2}$$

where λ_{ij} is the transition rate from state i in subset A to state j in subset B. State merging must follow two conditions. First, the equivalent transition rate given by equation (3.2) must be the same for any state i in subset A. Second, the probabilities of all the states merged into a subset must be equal.

The number of states can also be reduced by deleting states with a relatively low probability. Two techniques that can be used for this purpose are called state space truncation [Singh 72] and sequential truncation [Singh 75].

State space truncation must follow two conditions. First, the biggest probability in the truncated state space should be less that the smallest probability in the remaining state space. Second, after the states have been truncated, the states transition diagram should be examined to see if the process of truncation has generated any new absorbing states. Either the new absorbing states should be deleted or the states whose truncation has generated this new absorbing states should be retained. In systems consisting of N identical components with two states these conditions are not hard to achieve. The state space may be divided into N + 1 subsets, each subset having states of a certain level of coincident failures. At first an arbitrary level of truncation should be selected. The computation can then be repeated by including the next subset. If the new values are not significantly different from the previous ones, the computation can be stopped, otherwise one more subset should be included and the computations repeated. This should be extensible to systems with different types of components which are susceptible to transient and intermittent faults.

In sequential truncation the state probabilities are calculated every time a new state is generated and states with probabilities less than a reference value are deleted. This method consumes more computation time than state space truncation but does not have to be repeated to insure the accuracy of the approximation.

The state space truncation technique can be extended to produce a conservative estimate. The states with a certain level of coincident failures can be made failure states. This eliminates the out going transitions of the new failure states, and truncates those states that could only be reached through the new failure states. This will produce a conservative estimate because the truncated states will also be analyzed as though they were failure states.

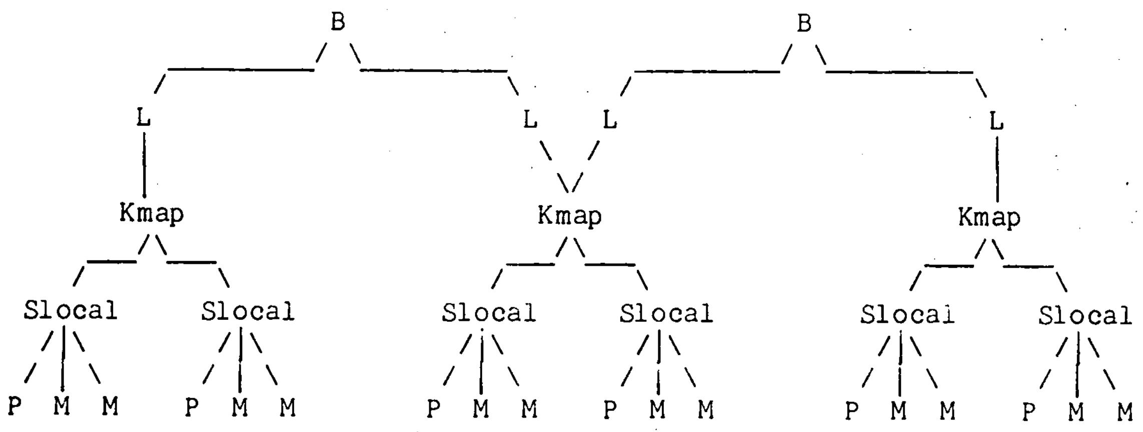
Only the extended state space truncation technique is applicable to the ARM program. The reason for this is that the ARM program will not be evaluating the reliability model. State space truncation will be attempted during the generation of the models for the PMS graph segments. Algorithms that follow this technique must be developed and implemented.

4. Automated Reliability Modeling Examples

Currently the ARM program is limited to systems that are non-redundant and non-repairable minimal subtrees, and are not susceptible to transient or intermittent faults. Only the three minimum ARM input categories have been implemented. These input categories are: the hard failure rate of the component types, the interconnection structure, and the system requirements. Only the output format for the SURE program has been implemented. The Cm* multiprocessor architecture described in [Swan 77] will be used to illustrate the current capabilities of the ARM program.

The Cm* multiprocessor architecture is based on the LSI-11 microcomputer. Figure 4-1 shows one possible version of the architecture. Each computer module (Cm) is composed of one processor module connected via an interface (Slocal) to one or more memory modules. The memories in the structure collectively realize the virtual address space shared by the processors. Each cluster is composed of a cluster controller (Kmap) and two or more Cms. The Slocal controls local memory access and passes external references (i.e. to memory elsewhere in the cluster or in a different cluster) to the Kmap. The Kmaps are mapping controllers which allow processors in Cms to access memory elsewhere in the cluster or in other clusters via the Intercluster Buses (B in the figure). The components marked L in the figure are interfaces from the Kmaps to the Intercluster Buses.

The following sections will illustrate the automatic generation of reliability models by modeling several versions of the Cm* architecture. The sensitivity of the models generated to the system requirements and the PMS interconnection graph is demonstrated. To validate the models generated they will be evaluated using the SURE program, and compared with the results of manually derived probability of failure equations. The exponential failure rates used to evaluate the models where obtained from [Siewiorek 78] and are reproduced in Table 4-1.



Key:

B	Intercluster Bus	Slocal	Local Switch
L	Intercluster Bus Interface	P	Processor
Kmap	Mapping Controller	M	Memory

Figure 4-1: Cm* Architecture

Processor	29.893E-6
Memory	46.278E-6
Local Switch	24.059E-6
Mapping Controller	130.935E-6
Intercluster Bus Interface	34.836E-6
Intercluster Bus	0.000E-6

Table 4-1: Failure Rates of Cm* Modules

4.1 Cm* Computer Module

The Cm* computer module to be modeled is composed of one processor module connected via an interface (Slocal) to three memory modules. The PMS diagram of the Cm* computer module is shown in Figure 4-2.

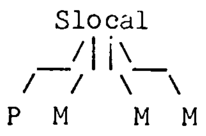
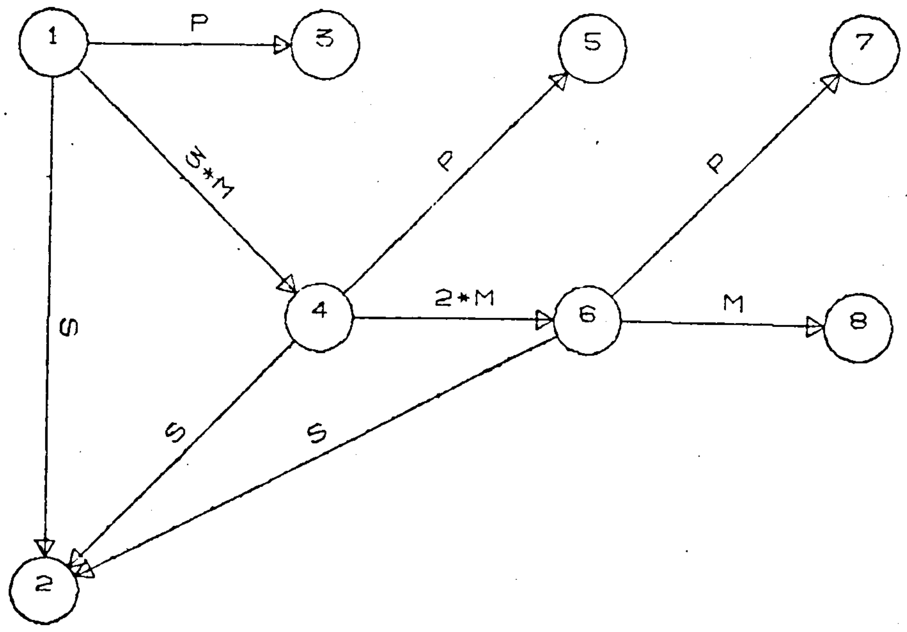


Figure 4-2: Cm* Computer Module

The model ARM automatically generated when the Cm* computer module in Figure 4-2 requires one processor and one memory to perform its function is shown in Figure 4-3. Only states 1, 4, and 6 are not failure states. The computer module will fail if three memories fail, or if any single component other than a memory fails. The system starts in state 1 with all its components working. The probability of failure, during the first ten hours of operation, obtained from this model is 5.39375E-4.



Key:	State	Failed components
	1	None
	2	1 S
	3	1 P
	4	1 M
	5	1 M & 1 P
	6	2 M
	7	2 M & 1 P
	8	3 M

Figure 4-3: Model of Figure 4-2 Cm* Requiring 1 P & 1 M

The equation for the probability of failure P_f is

$$P_f = 1 - R_s R_p (R_m^3 + 3R_m^2(1 - R_m) + 3R_m(1 - R_m)^2) \qquad (4.1)$$

where R_s, R_p, and R_m are the reliability functions of a local switch, a processor, and a memory. The R_m^3 term corresponds to the state in which all three memories function. The $3R_m^2(1 - R_m)$ term corresponds to the state in which one memory failed and two are functional. The $3R_m(1 - R_m)^2$ term corresponds to the state in which two memories failed and one is functional. The probability of failure, during the first ten hours of operation, obtained from this equation is 5.39375E-4. This is the same result obtained from the model in Figure 4-3.

4.2 Effect of the System Requirements

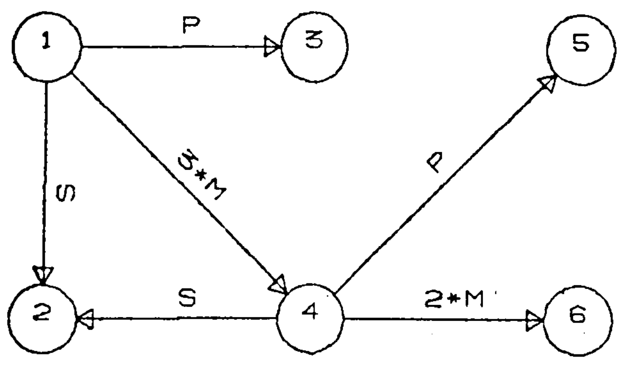
The number of components N required for a system to perform its function affects both the number of states and the probability of failure. The number of states is a non-increasing function of N. The probability of failure is a non-decreasing function of N.

For example, the model ARM automatically generated when the requirements of the Cm* computer module in Figure 4-2 are increased to 1 processor and 2 memories is shown in Figure 4-4. Only states 1 and 4 are not failure states. The computer module will fail if two memories fail, or if any single component other than a memory fails. The system starts in state 1 with all its components working. Comparing this model to the one shown in Figure 4-3, the number of states decreased to six and the probability of failure, during the first ten hours of operation, increased to 5.40016E-4.

The equation for the probability of failure P_f is

$$P_f = 1 - R_s R_p (R_m^3 + 3R_m^2(1 - R_m)) \qquad (4.2)$$

where R_s, R_p, and R_m are the reliability functions of a local switch, a processor, and a memory. The R_m^3 term corresponds to the state in which all three memories function. The $3R_m^2(1 - R_m)$ term corresponds to the state in which one memory failed and two are functional. The probability of failure, during the first ten hours of operation, obtained from this equation is 5.40016E-4. This is the same result obtained from the model in Figure 4-4.



Key:	State	Failed components
	1	None
	2	1 S
	3	1 P
	4	1 M
	5	1 M & 1 P
	6	2 M

Figure 4-4: Model of Figure 4-2 Cm* Requiring 1 P & 2 M

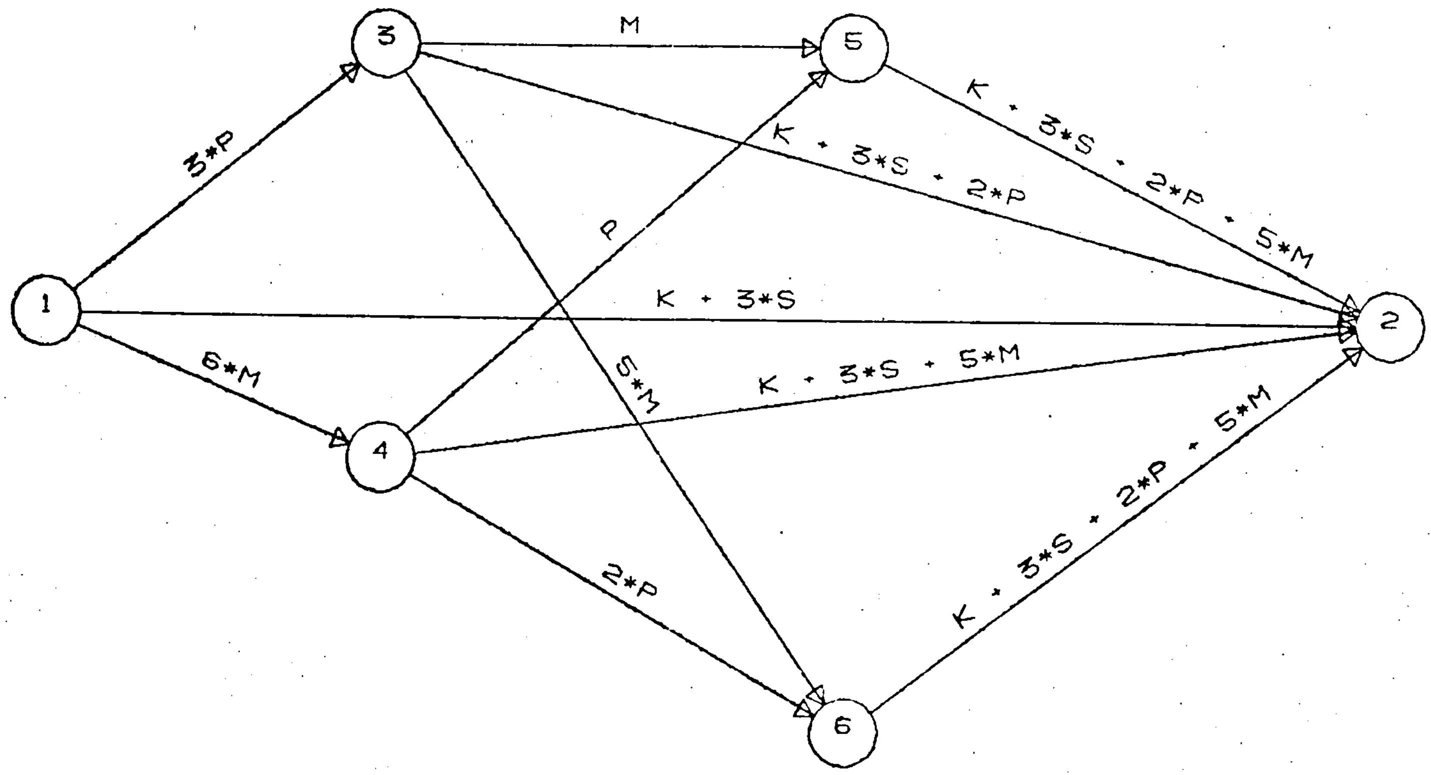
4.3 Cm* Cluster

The Cm* cluster to be modeled is composed of three computer modules connected via a cluster controller (Kmap). Each computer module is composed of one processor module connected via an interface (Slocal) to two memory modules. The PMS diagram of the Cm* cluster is shown in Figure 4-5.

```
                      Kmap
                      /|\
        _____________/ | \_____________
       /               |               \
    Slocal          Slocal          Slocal
     /|\             /|\             /|\
    / | \           / | \           / | \
   P  M  M         P  M  M         P  M  M
```

Figure 4-5: Cm* Cluster

The model ARM automatically generated when the Cm* cluster in Figure 4-5 requires 2 processors and 5 memories to perform its function is shown in Figure 4-6. All the failure states have been collapsed into state 2. The cluster will fail if two memories or two processors fail. The system starts in state 1 with all its components working. In state 5 one processor and one memory in the same computer module have failed, therefore if their local switch fails no other components will be affected. In state 6 one processor and one memory in a different computer module have failed, therefore if a local switch fails other components will also be affected. The probability of failure, during the first ten hours of operation, obtained from this model is 2.03253E-3.



Key:

State	Failed components
1	None
2	1 K or 1 S or 2 P or 2 M
3	1 P
4	1 M
5	1 M & 1 P in the same Cm
6	1 M & 1 P in a different Cm

Figure 4-6: Model of Figure 4-5 Cm* Requiring 2 P & 5 M

The equation for the probability of failure P_f is

$$P_f = 1 - R_k R_s^3 (R_p^3 + 3R_p^2(1 - R_p))(R_m^6 + 6R_m^5(1 - R_m)) \qquad (4.3)$$

where R_s, R_p, and R_m are the reliability functions of a local switch, a processor, and a memory. The R_p^3 term corresponds to the state in which all three processors function. The $3R_p^2(1 - R_p)$ term corresponds to the state in which one processor failed and two are functional. The R_m^6 term corresponds to the state in which all six memories function. The $6R_m^5(1 - R_m)$ term corresponds to the state in which one memory failed and five are functional. The probability of failure, during the first ten hours of operation, obtained from this equation is 2.03253E-3. This is the same result obtained from the model in Figure 4-6.

4.4 Effect of the PMS Interconnection

The PMS interconnection affects both the number of states and the probability of failure. For example, let us change the PMS interconnection of the Cm* cluster in Figure 4-5 so that two computer modules are composed of one processor and one memory, and the third computer module is composed of one processor and four memories. The resulting PMS diagram of the Cm* cluster is shown in Figure 4-7.

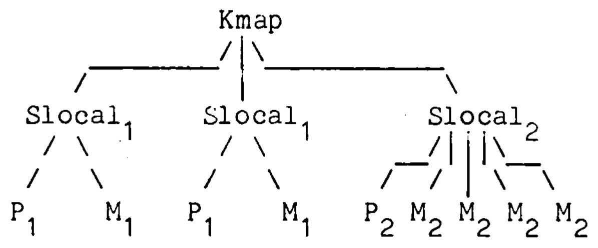


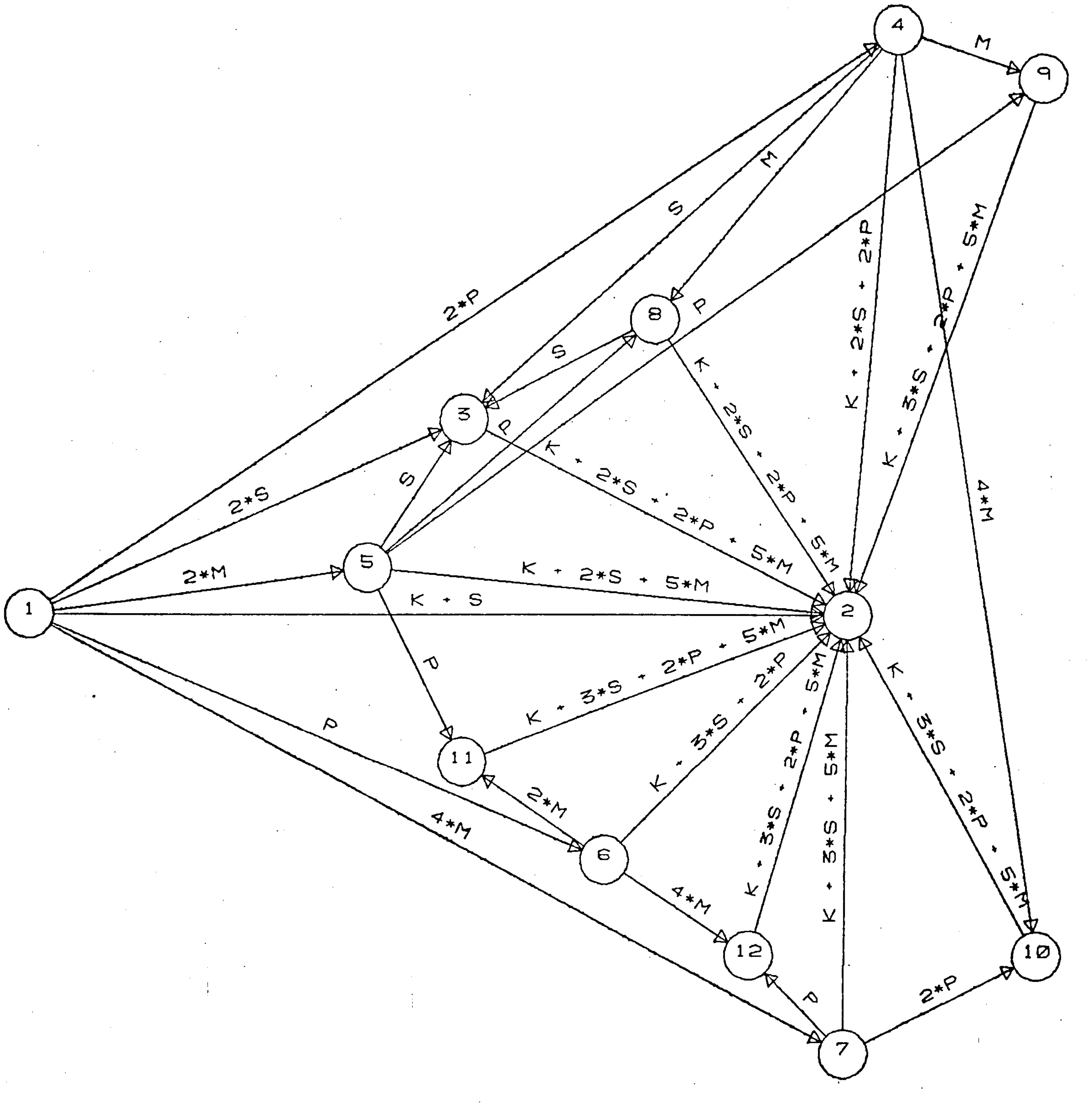
Figure 4-7: Nonsymmetrical connection of Figure 4-5 Cm* Cluster

The model ARM automatically generated when the Cm* cluster in Figure 4-7 requires 2 processor and 5 memories to perform its function is shown in Figure 4-8. All the failure states have been collapsed into state 2. The cluster will fail if two memories or two processors fail. The system starts in state 1 with all its components working. Comparing this model to the one shown in Figure 4-6, the number of states increased to twelve and the probability of failure, during the first ten hours of operation, decreased to 1.55366E-3.

The equation for the probability of failure P_f is

$$P_f = 1 - R_k R_s^3 (R_p^3 + 3R_p^2(1 - R_p))(R_m^6 + 6R_m^5(1 - R_m)) - 2R_k R_s^2(1 - R_s) R_p^2 R_m^5 \quad (4.4)$$

where R_s, R_p, and R_m are the reliability functions of a local switch, a processor, and a memory. The only difference between this equation and equation (4.3) is the addition of the $2R_k R_s^2(1 - R_s) R_p^2 R_m^5$ term. This term corresponds to the state in which one local switch S_1 failed and the other two local switches are functional. The probability of failure, during the first ten hours of operation, obtained from this equation is 1.55366E-3. This is the same result obtained from the model in Figure 4-8.



Key: State	Failed components	State	Failed components
1	None	7	1 M_2
2	1 K, 1 S_2, 2 P, or 2 M	8	1 P_1 & M_1 in the same Cm
3	1 S_1 & 1 P_1 & 1 M_1	9	1 P_1 & M_1 in a different Cm
4	1 P_1	10	1 P_1 & 1 M_2
5	1 M_1	11	1 P_2 & 1 M_1
6	1 P_2	12	1 P_2 & 1 M_2

Figure 4-8: Model of Figure 4-7 Cm* Requiring 2 P & 5 M

5. Plans for Future Work

The architectures and fault types the research will address will increase in complexity in phases as described below. The reason for breaking the research work into phases is to keep the complexity of the problem being addressed at a manageable level. The results of each phase of the research will be theoretically and experimentally validated before proceeding to the next phase. The ARM program will be used as part of the experimental validation of each phase. Next the performance and range of applications of the ARM program must be evaluated. Based on the results of the validation and evaluation the approach will be reformulated as necessary.

The first phase of the research will address hard faults, and non-redundant and non-repairable PMS tree structures that require their root to be operational. This phase will only involve research into steps 1, 4, and 5 of Table 3-1. All subsequent phases will involve research into steps 1, and 4 through 7 of Table 3-1.

Phase two of the research will address hard faults and non-redundant general structures with no repair. The third phase of the research will address hard faults, and dynamically redundant architectures that can have imperfect coverage, and repair. Examples of such architectures are a multiprocessor at CMU, named Cm* [Swan 77], and the Electronic Switching Systems (ESS) used in the Bell System [Toy 78]. Cm* and ESS will be used in the experimental validation of this phase.

Phase four of the research will address hybrid redundant architectures but only for hard faults. The fifth and last phase of the research will address intermittent, transient, and hard faults. For the last two phases the architectures used for experimental validation will be the Fault-Tolerant Multiprocessor (FTMP) at NASA's Langley Research Center [Lala 83], an Intel 432 [Siewiorek 82] based multiprocessor, Cm*, and ESS.

6. Conclusion

The previous sections presented an approach for automatic Markov reliability and availability modeling of computer architectures. This approach consists of eight steps which were summarized in Table 3-1. The Automated Reliability Modeling (ARM) program is being developed to implement these steps. The first step is to obtain a system description consisting of the Processor-Memory-Switch (PMS) interconnection graph, the behavior of the PMS components, the fault-tolerant strategies, and the operational requirements. Section 2 described the eight input categories currently envisioned for the system description method of the ARM program. These input categories are capable of describing most of the current computer architectures.

The other steps generate the Markov model from this system description. Section 3 discussed the purpose and necessity of these steps. Progress already made in identifying and analyzing the problems involved, and developing and implementing algorithms to solve them was also presented.

Section 4 presented examples of the current capabilities of the ARM program. The sensitivity of the models generated to the system requirements and the PMS interconnection graph was also demonstrated. Section 5 presented the current plans for extending the capabilities of the ARM program to include all of the steps in Table 3-1.

A. ARM Program Algorithms

A.1 Symmetry Detection Algorithm

Function definitions:

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 neighbors in each class as the preceding elements of class C.

```
procedure Symmetry;

function Equivalent(Current_Element, Class, Relation);

begin
   if Relation = Degree then
        return Equal_Degree(Current_Element, Class)
   else return Equal_Neighbor_Classes(Current_Element, Class);
end;

function Split_Class(Relation, This_class, Last_class);

begin
   Split := false;
   for I := 2 to Size(This_Class) do
   begin
      Current_Element := Element(I, This_Class);
      if not Equivalent(Current_Element, This_class, Relation) then
      begin
         if not Split then
         begin
            Split := true;
            Last_Class := Last_Class + 1;
            ( Create a new Last_class with the degree and neighbor
              attributes of the Current_element of This_class. );
         end;
         ( Move the Current_Element of This_class to the Last_class. );
      end;
   end;
   return Last_Class;
end; { Split_Class }

begin { Symmetry }

   { Step 1: Split based on equal type. }

   Last_Class := Last_Type;
   for I := 1 to Last_Class do
      ( Add elements of type I to class I. );

   { Step 2: Split based on equal degree. }

   I := 1;
   while I <= Last_Class do
   begin
      Last_Class := Split_Class(Degree, I, Last_Class);
      I := I + 1;
   end;

   { Step 3: Split based on equal neighbor classes. }

   New_Last := Last_Class;
   Done := false;
   while not Done do
   begin
      for I := 1 to Last_Class do
         New_Last := Split_Class(Neighbors, I, New_Last);
      if Last_Class = New_Last then
           Done := true
      else Last_Class := New_Last;
   end;

end; { Symmetry }
```

A.2 Segmentation Algorithm

Function definitions:

Degree(C) - Returns the degree of class C.

Root(T) - Returns the root of tree T.

Neighbors(C) - Returns the set of neighbors of class C.

Up_Neighbors(C) - Returns the number of neighbors of class C that are not already in the tree of which class C is the root.

Up_Degree(C) - Returns the degree of connectivity of class C with neighbors that are not already in the tree of which class C is the root.

Variable definitions:

Dead[T] - True when tree T has been merged into another tree. All the array is initialized as false.

Complete[T] - True when tree T is not capable of further growth. All the array is initialized as false.

```
procedure Segmentation;

begin

   { Step 1: Collect the germinal trees, that is the leaf vertices of
     G' which represent classes of leaf vertices of G. }

   Last_Tree := 0;
   for I := 1 to Last_class do
      if Degree(I) = 1 then
      begin
         Last_Tree := Last_Tree + 1;
         ( Initialize the Last_Tree with class I as its root and
           single node. );
      end;

   { Continue growing the germinal trees (steps 2 and 3) until no more
     adding of vertices or merging of trees is possible. }

   Changes := true;
   while Changes do
   begin
      Changes := false;

      { Step 2: Grow these germinal trees upward by adding on
        neighboring vertices of these leaves. }

      for I := 1 to Last_Tree do
         if not Dead[I] and not Complete[I] then

            { Stopping condition 1: The root is a neighbor to itself. }

            if Root(I) in Neighbors(I) then
                 Complete[I] := true
            else if Up_Neighbors(I) = 1 then
```

```
                  { Stopping condition 2: The root has a single
                    neighbor outside of the tree with density greater
                    than one. }

                  if Up_Degree(I) = 1 then
                  begin
                       ( Grow the tree by adding the root's unique
                         neighbor, outside of the tree, with density
                         of one as its new root. );
                       Changes :=true;
                  end
                  else Complete[I] := true;

      { Step 3: Merge the germinal trees that overlap at their roots. }

      for I := 1 to Last_Tree do
         if not Dead[I] then
            for J := I + 1 to Last_Tree do
               if not Dead[J] then
                  if Root(I) = Root(J) then
                  begin
                     ( Merge tree J into tree I. );
                     Changes := true;
                     Dead[J] := true;

                     { Stopping condition 3: The tree is merged with
                       another tree that meets conditions 1 or 2. }

                     if Complete[J] then
                        Complete[I] := true;
                  end;
   end; { while Changes }
end; { Segmentation }
```

A.3 Success and Failure State Identification Algorithm

Variable definitions:

Last_Conjunction - The number of conjunctions in the sum-of-products expression of system requirements.

Last_Atom[I] - The number of atomic requirements in conjunction I.

Classes[I][J] - The set of equivalence classes whose components are of the type specified by atomic requirement J of conjunction I.

Max_Dead[I][J] - The maximum number of components, of the type specified by atomic requirement J of conjunction I, that can fail and the system remain operational.

State[K] - The number of components in class K that have failed when the system is in this state.

```
function Success(State); { True if its argument is a success state. }

begin
   I := 1;
   Alive := false;

   { Level 1: If any conjunctive requirement, in the sum-of-products
     expression of system requirements, is meet then this is a success
     state. }

   while not Alive and I <= Last_Conjunction do
   begin
      J := 1;
      Alive := true;

      { Level 2: If all atomic requirements in a conjunction are meet
        then this is a success state. }

      while Alive and J <= Last_Atom[I] do
      begin
         K := 1;
         Dead := 0;

         { Level 3: Count the number of components of the specified
           type that have failed and if it does not exceed the maximum
           value then the state meets the atomic requirement. }

         while Alive and K <= Last_Class do
         begin
            if K in Classes[I][J] then
            begin
               Dead := Dead + State[K];
               if Dead > Max_Dead[I][J] then
                  Alive := false;
            end;
            K := K + 1;
         end;
         J := J + 1;
      end;
      I := I + 1;
   end; { while not Alive }
   Success := Alive;
end; { Success }
```

A.4 Minimal Subtree Model Generation Algorithm

Function definitions:

Get_State(N) - Returns a state from the set of new states N and removes it from the set.

Success(S) - True if state S is a success state.

Class(T) - Returns the equivalence class of node T of the minimal subtree.

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

New_State(S) - True if state S is a new state.

Lambda(C) - Returns the failure rate of components in class C.

Variable definitions:

State[C] - The number of components in class C that have failed when the system is in this state.

Last_Node - The number of nodes in the minimal subtree.

Subtree[I][C] - The number of components in class C that are in subtree I of the minimal subtree.

Next[I][C] - The number of components in class C that have failed when the system reaches its next state I.

Working[I][J] - The number of working components in the class whose failure is described by the transition from state I to state J.

Failure_Rate[I][J] - The failure rate of the class of component whose failure is described by the transition from state I to state J.

```
procedure Model_Subtree(Subtree, Last_Node);

begin

   { Step 1: Initialize New_Set to the start state. }

   for I := 1 to Last_Class do
      State[I] := 0;
   New_Set := [State];

   { Step 2 : While the New_Set is not empty, get a state out of the
     New_Set until a success state is found. }

   while New_Set <> [] do
   begin
      State := Get_State(New_Set);
      if Success(State) then
```

```
        { Step 3: For every node in the minimal subtree, if more
          components of this class can fail then generate the
          transitions out of the success state. }

        for I := 1 to Last_Node do
        begin
           C := Class(Subtree[I]);
           if State[C] < Size(C) then
           begin
              Num_States := 0;
              If (a node subtree has all its components working) then
              begin
                 (generate a transition for one such subtree failure);
                 Num_States := Num_States + 1;
              end;
              For J := 1 to (number of subtrees with a different set
                             of failed components) do
              begin
                 (generate a transition for the failure of subtree J);
                 Num_States := Num_States + J;
              end;

              { Step 4: For every transition generated: }

              for J := 1 to Num_States do
              begin

                 { Substep 4a: If the destination state is new then
                   add it to the New_Set. }

                 if New_State(Next[J]) then
                    New_Set := New_Set + Next[J];

                 { Substep 4b: Add the transition to the model. }

                 if J = 1 then
                      Working[State][Next[J]] :=
                              Size(C) - State[C] - Num_States + 1
                 else Working[State][Next[J]] := 1;
                 Failure_Rate[State][Next[J]] := Lambda(C);
              end;
           end; { if more can fail }
        end; { for all nodes in minimal subtree }
    end; { while the set of new states is not empty }
end; { Model_Subtree }
```

References

[Bavuso 84]	S. J. Bavuso, P. L. Peterson, and D. M. Rose. CARE III Model Overview and User's Guide. Technical Report TM85810, NASA-LaRC, 1984.
[Butler 84]	Ricky W. Butler. The Semi-Markov Unreliability Range Evaluator (SURE) Program. Technical Report TM86261, NASA-LaRC, 1984.
[Butler 85]	Ricky W. Butler. An Abstract Specification Language for Markov Reliability Models. Technical Report TM86423, NASA-LaRC, 1985.
[Cox 68]	R. E. Cox and H. D. Miller. The Theory of Stochastic Processes. Methuen, 1968.
[Chung 67]	Kai Lai Chung. Markov Chains with Stationary Transition Probabilities. Springer-Verlag, 1967.
[Dugan 84]	Joanne B. Dugan. Extended Stochastic Petri Nets: Applications and Analysis. Ph. D. Thesis, Duke University, 1984.
[Kini 81]	Vittal Kini. Automatic Generation of Reliability Functions for Processor-Memory-Switch Structures. Ph.D. Thesis, Carnegie-Mellon University, 1981.
[Lala 83]	Jaynarayan Lala and T. Basil Smith III. Development and Evaluation of a Fault-Tolerant Multiprocessor (FTMP) Computer. Technical Reports CR166071-3, NASA-LaRC, 1983.
[Landrault 78]	C. Landrault and J. C. Laprie. SURF-A Program for Modeling and Reliability Prediction for Fault-Tolerant Computing Systems. In Josef Moneta (editor), Information Technology, North-Holland, 1978.
[Lee 85]	Larry D. Lee. Reliability Bounds for Fault-Tolerant Systems With Competing Responses to Component Failures. Technical Report TP2409, NASA-LaRC, 1985.
[Makam 82]	S. V. Makam, A. Avizienis, and G. Grusas. ARIES 82 User's Guide. Technical Report CSD-82-830, UCLA, 1982.

[McConnel 81]	Stephen R. McConnel. Analysis and Modeling of Transient Errors in Digital Computers. Ph.D. Thesis, Carnegie-Mellon University, 1981.
[Romanovsky 70]	V. I. Romanovsky. Discrete Markov Chains. Wolters-Noordhoff, 1970.
[Siewiorek 78]	D. P. Siewiorek and D. E. Thomas (Eds.). The Analysis of the Performance, Reliability and Life Cycle Cost of Multi-Processor Architectures and their Impact on SENET. Technical Report CMU-CS-78-126, CMU, 1978.
[Siewiorek 82a]	Daniel P. Siewiorek, C. Gordon Bell, and Allen Newell. Computer Structures: Principles and Examples. McGraw Hill, 1982.
[Siewiorek 82b]	D. P. Siewiorek and R. S. Swarz. The Theory and Practice of Reliable System Design. Digital Press, 1982.
[Singh 72]	Chanan Singh. Reliability Modelling and Evaluation in Electric Power Systems. Ph.D. Thesis, University of Saskatchewan, Saskatoon, Canada, 1972.
[Singh 75]	Chanan Singh. Reliability Calculations of Large Systems. In Proceedings of the Reliability and Maintainability Symposium, 1975.
[Stiffler 79]	J. J. Stiffler, L. A. Bryant, and L. Guccione. CARE III Final Report: Phase One. Technical Report CR159122, NASA-LaRC, 1979.
[Swan 77]	R. J. Swan, S. H. Fuller, and D. P. Siewiorek. Cm*-A Modular, Multi-microprocessor. In Proceedings of the National Computer Conference, AFIPS, 1977.
[Toy 78]	W. N. Toy. Fault-Tolerant Design of Local ESS Processors. In Proceedings of the IEEE, 1978.
[Trivedi 81]	K. Trivedi and R. Geist. A Tutorial on the CARE III Approach to Reliability Modeling Technical Report CR3488, NASA-LaRC, 1981.

[Trivedi 85] K. Trivedi, R. Geist, M. Smotherman, and J. B. Dugan.
A Description of the HARP User Interface.
Duke University, 1985.

[White 84] Allan L. White.
Upper and Lower Bounds for Semi-Markov Reliability Models of Reconfigurable Systems.
Technical Report CR172340, NASA-LaRC, 1984.

1. Report No. NASA TM-89009	2. Government Accession No.	3. Recipient's Catalog No.
4. Title and Subtitle Towards Automatic Markov Reliability Modeling of Computer Architectures		5. Report Date August 1986
		6. Performing Organization Code 505-66-21-02
7. Author(s) Carlos A. Liceaga and Daniel P. Siewiorek		8. Performing Organization Report No.
		10. Work Unit No.
9. Performing Organization Name and Address NASA Langley Research Center Hampton, Virginia 23665-5225		11. Contract or Grant No.
		13. Type of Report and Period Covered Technical Memorandum
12. Sponsoring Agency Name and Address National Aeronautics and Space Administration Washington, DC 20546-0001		14. Sponsoring Agency Code

15. Supplementary Notes

Carlos A. Liceaga, NASA Langley Research Center, Hampton, Virginia.
Daniel P. Siewiorek, Carnegie-Mellon University, Pittsburgh, Pennsylvania.

16. Abstract

The analysis and evaluation of reliability measures using time-varying Markov models is required for Processor-Memory-Switch (PMS) structures that have competing processes such as standby redundancy and repair, or renewal processes such as transient or intermittent faults. The task of generating these models is tedious and prone to human error due to the large number of states and transitions involved in any reasonable system. Therefore model formulation 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 formulation.

This paper presents an overview of the Automated Reliability Modeling (ARM) program, under development at NASA Langley Research Center. ARM will accept as input a description of the PMS interconnection graph, the behavior of the PMS components, the fault-tolerant strategies, and the operational requirements. The output of ARM will be the reliability or availability Markov model formulated for direct use by evaluation programs. The advantages of such an approach are (a) utility to a larger class of users, not necessarily expert in reliability analysis, and (b) a lower probability of human error in the computation.

17. Key Words (Suggested by Author(s)) Reliability Analysis; Automated Reliability Modeling; Time-Varying Markov Models; Fault-Tolerance; Processor-Memory-Switch (PMS) Structures	18. Distribution Statement Unclassified--Unlimited Subject Category 66

19. Security Classif. (of this report) Unclassified	20. Security Classif. (of this page) Unclassified	21. No. of Pages 67	22. Price A04

N-305