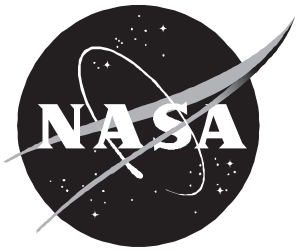


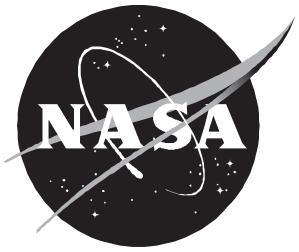
NASA Technical Paper 3452
Volume 1

HiRel: Hybrid Automated Reliability Predictor (HARP) Integrated Reliability Tool System (Version 7.0)

HARP Introduction and User's Guide

Salvatore J. Bavuso, Elizabeth Rothmann, Joanne Bechta Dugan, Kishor S. Trivedi, Nitin Mittal, Mark A. Boyd, Robert M. Geist, and Mark D. Smotherman

November 1994



NASA Technical Paper 3452
Volume 1

HiRel: Hybrid Automated Reliability Predictor (HARP) Integrated Reliability Tool System (Version 7.0)

HARP Introduction and User's Guide

Salvatore J. Bavuso
Langley Research Center • Hampton, Virginia

Elizabeth Rothmann, Joanne Bechta Dugan, Kishor S. Trivedi, Nitin Mittal, and Mark A. Boyd
Duke University • Durham, North Carolina

Robert M. Geist and Mark D. Smotherman
Clemson University • Clemson, South Carolina

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

November 1994

HiRel: Hybrid Automated Reliability Predictor (H
Integrated Reliability Tool System (Version 7.0)

NASA TP-3452, Vol. 1

HARP Introduction and User's Guide

This publication is available from the following sources:

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

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

Contents

Preface

The Hybrid Automated Reliability Predictor (HARP) integrated Reliability (HiRel) tool system for reliability/availability prediction stems from a peer review of the Computer Aided Reliability Estimation third generation (CARE III) computer program that was conducted in 1980 at the Research Triangle Institute in North Carolina. A participating reviewer, Dr. Kishor Trivedi of Duke University, did a mathematical analysis of CARE III and suggested improvements to enhance its modeling capability. Because of the new features and mathematical changes, NASA decided to create a new capability called HARP. I was then the NASA CARE III project engineer and became the HARP project engineer also. The development of CARE III and HARP continued simultaneously.

HARP was developed by researchers at Duke University under NASA grant NAG1-70. Many innovative capabilities were developed for HARP by graduate students and their research advisors. A number of doctoral dissertations contained research that was incorporated into HARP, which became a joint Duke-Langley development project. Langley's contributions to the design included two sequence dependency gates, the redesign and implementation of the textual prompting interface, the integration of all HARP programs with uniform prompts, and other recommendations such as the incorporation of a stiff ordinary differential solver and the state truncation technique. The first working HARP program was sent to requesting beta test sites in 1985.

In 1985 when IBM Corporation announced support for the Graphical Kernel System (GKS), an ANSI standard that promised portability, Langley encouraged the Duke team to investigate the development of a graphical user-interface for HARP. The result of this work is the Graphics Oriented (GO) program. The GO program was completed at Langley with the help of students from Old Dominion University (ODU) working in Langley's Voluntary Services Program and with the help of Sandra Howell Koppen and Pamela J. Haley. The ODU students tested the prototype GO program that was originally written for an IBM-compatible personal computer (PC). Koppen took GO from an alpha to a beta program and implemented many new features, including the sequence dependency gates. Because GKS is not a universally implemented standard, Haley reimplemented the PC GO program on the Sun Microsystems, Inc., and Digital Equipment Corporation VAX workstations.

Tanya R. Arthur and DeAnn E. Junchter, two ODU students working under the Langley's Voluntary Services Program and my direction, implemented the code for the HARP Output (HARPO) program. Although I provided the initial design, Arthur made many refinements to produce the prototype program. Darrell Sproles, from Computer Sciences Corporation, made major modifications to the design and reimplemented the code. We jointly refined the design to bring HARPO to its present state. With the completion of HARPO, the main components for HiRel (HARP, GO, and HARPO) were finished and beta testing commenced in May 1991.

Koppen and I also served as the engineering interface to over 100 beta test site users who carried on the beta test concept that I established at Langley for testing CARE III. Langley also served as an alpha and beta test site. All code was first extensively tested at Duke then again at Langley before being distributed to the user community. The beta test program was a resounding success. The long-term Langley interaction with HARP users (8 yr) and HiRel (2 yr) brought an important element of practicality to HiRel's usage and development. Many changes to HiRel resulted from beta site recommendations. These changes included the discovery of bugs, suggestions for improving the interface, and design modifications. In this regard, I wish to acknowledge Dr. Tilak Sharma at Boeing Commercial Airplane Group. Sharma saw the power of the fault tree sequence dependency gates and encouraged the HARP team to pursue this

work. Sharma, who has become a trusted friend over our long association, extensively tested HARP and "gets the prize" for finding the most bugs for any beta site.

One important point remains to be said regarding the beta testing of reliability/availability programs. Several philosophies exist for the justification of a particular scheme of testing. Based on our experience with CARE III and HiRel, the method we used was extremely effective. We used a wide distribution of users involved in a large diversity of applications ranging from satellites to submarines. We imposed few restrictions on our choice of HiRel users (other than they be U.S. users) and their applications. All distribution was made to unsolicited requesters. Because of this wide exposition, HiRel has become a very flexible and useful capability in many U.S. industries. We also serendipitously found an effective mechanism to transfer NASA developed technology throughout the U.S.

Our experience with CARE III taught us that a useful program eventually becomes modified to suit the specific needs of the user. We had anticipated and fostered this need by distributing source code. Two additional components of HiRel have emerged as a result, phased-mission HARP and Monte Carlo HARP. We also learned that useful code gets absorbed into many company and university computer programs and eventually looses its initial identity. These are excellent examples of NASA technology transfer.

To my colleagues and friends at Duke and Clemson Universities, I wish to say that the most rewarding episode of my professional career has been my association with you. Not only did your efforts produce a product worthy of the 21st century, but you taught me the true meaning of being a dedicated researcher and the friendship and loyalty it brings. It is a glowing tribute to your schools and this country to have such high-quality professionals.

Salvatore J. Bavuso

Contributors

HARP Designers

Kishor S. Trivedi
Department of Computer Science
Duke University

Joanne Bechta Dugan
Department of Computer Science
Duke University

Robert M. Geist
Department of Computer Science
Clemson University

Mark K. Smotherman
Department of Computer Science
Clemson University

Salvatore J. Bavuso
NASA Langley Research Center

Mark A. Boyd
Duke University

HARP Implementors

Elizabeth Rothmann
Duke University

Joanne Bechta Dugan
Duke University

Mark A. Boyd
NASA Ames Research Center

Mark K. Smotherman
Clemson University

Nitin Mittal
Duke University

Salvatore J. Bavuso
NASA Langley Research Center

NASA Project Leader

Salvatore J. Bavuso
NASA Langley Research Center

Chapter 1

Introduction

1.1. HiRel Tool System

The Hybrid Automated Reliability Predictor (HARP) integrated Reliability (HiRel) tool system for reliability/availability prediction (refs. 1 and 2) offers a toolbox of integrated[1] reliability/availability programs that can be used to customize the user's application in a workstation or nonworkstation environment. HiRel consists of interactive graphical input/output programs and four reliability/availability modeling engines that provide analytical and simulative solutions to a wide host of highly reliable fault-tolerant system architectures. Three HiRel programs were developed by researchers at Duke University and Langley Research Center.

The tool system was designed to be compatible with most computing platforms and operating systems, and some programs have been beta tested within the aerospace community for over 8 years. Many examples of HiRel's use have been reported in the literature and at the HARP workshop conducted at Duke University, July 10–11, 1990.

The wide range of applications of interest has caused HiRel to evolve into a family of independent programs that communicate with each other through files that each program generates. In this sense, HiRel offers a toolbox of integrated programs that can be executed to customize the user's application. Figure 1 illustrates the HiRel tool system. The core of this capability consists of the reliability/availability modeling engines, which are collectively called the Hybrid Automated Reliability Predictor suite (HARP/S).

The modeling engines are comprised of four self-contained executable software components: the original HARP program also called textual HARP (described in vols. 1 and 2 of this TP), Monte Carlo integrated HARP (MCI-HARP) (ref. 3), Phased Mission HARP (PM-HARP) (ref. 4), and X Window system HARP (XHARP) (ref. 5). In conjunction with the engine suite, are two interactive graphical input/output programs that provide a workstation environment for HiRel. These programs are called the Graphics Oriented (GO) program (described in vol. 3 of this TP) and the HARP Output (HARPO) program (described in vol. 4 of this TP). The base components of HiRel (GO, HARP, MCI-HARP, and HARPO) are available through NASA's software distribution facility, COSMIC.[2] PM-HARP[3] and XHARP[4] may be available from their respective developers.

1.2. HARP Suite and Its Applicability

HARP/S is comprised of four computer programs that provide a general Markov modeling capability to conveniently model and predict the reliability/availability of a wide variety of systems. The primary input (except XHARP) is a fault tree that can be in tabular or graphical form. (The primary XHARP input is a Markov model in graphic form.) The fault tree is not limited to the traditional combinatorial modeling approach. The addition of four special

[1] HiRel programs can communicate with each other in a common ASCII file format, a necessary capability for computer-aided design (CAD) integration.

[2] COSMIC, The University of Georgia, 382 East Broad St., Athens, GA 30602.

[3] The Boeing Commercial Airplane Group, Seattle, WA 98124 (Tilak Sharma).

[4] Clemson University, Dept. of Computer Science, Clemson, SC 29734 (Robert Geist).

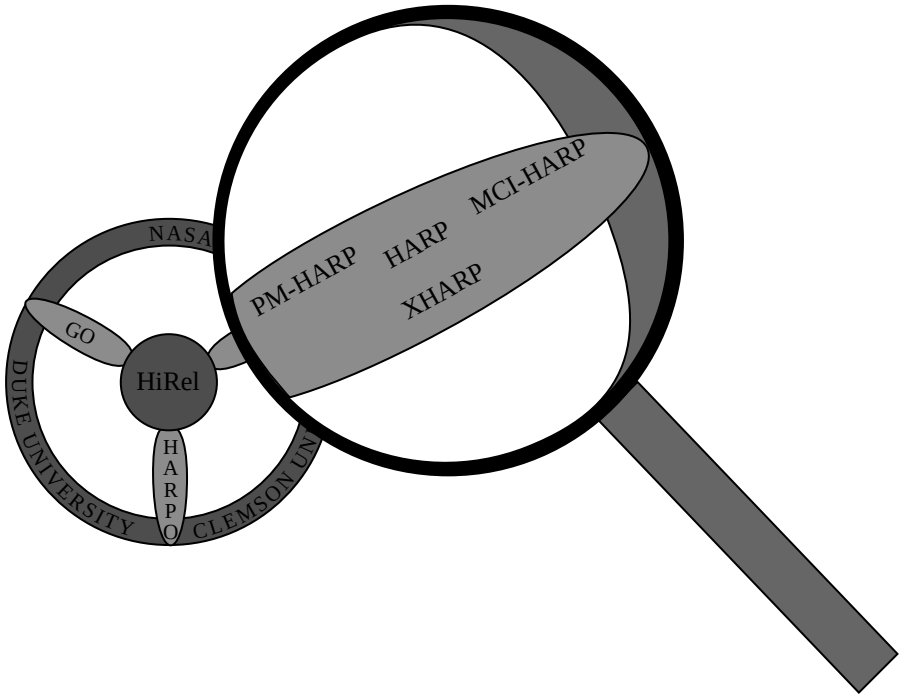


Figure 1. HiRel: GO, HARPO, and HARP suite of reliability engines.

fault tree gates called dependency gates allow the generation of dynamic fault tree models that, until now, were not practical to solve with analytical solution techniques. The fault tree, a familiar and convenient notation for expressing reliability models, is automatically converted by HARP/S into a Markov model that is solved to produce a reliability/availability prediction. The numerical results are expressed in tabular or graphical form with HARPO. HARP/S also accepts system reliability/availability models expressed directly in the form of a Markov model, where the user inputs the model's origin state and destination state together with a state transition rate. Unless HiRel is used in an hierarchical fashion, this form of input can be impractical for many modeling situations where thousands of Markov states must be enumerated.

For fault-tolerant systems that use redundancy and subsystem reconfiguration to achieve ultrahigh predicted reliabilities, even automatic model generation capabilities can be inadequate to cope with the potentially millions of Markov states necessary to model the system reliability. The addition of millions of states in models of fault-tolerant systems results from the need to account for the fault/error handling mechanisms typically used in these systems. For many of these systems, the extremely large state size causes insurmountable computational difficulties that preclude reliability/availability prediction. HARP/S offers an innovative modeling technique that avoids having to generate and solve such large models by implementing a modeling technique called behavioral decomposition (refs. 6 and 7). Behavioral decomposition is a mathematical technique that exploits two specific behaviors of fault-tolerant systems: (1) the failure of hardware parts and subsystems typically occurs after thousands of hours of operation, and (2) the time involved in the handling of faults/errors is usually on the order of milliseconds to seconds. This wide disparity of typically 6 orders of magnitude of the system time constants forms the mathematical basis of behavioral decomposition that guarantees that the reliability computation is conservative (refs. 8 and 9). It is conservative in that it predicts a reliability that is equal to or less than the reliability predicted by the full model.

A key step in arriving at a conservative reliability prediction in any reliability modeling activity is for the modeler to insure that all system failure modes have been accounted for in the model specification; otherwise, a conservative result is not guaranteed. (See section 2.1.) For many applications, this practice is perfectly acceptable provided that the modeler can account for the deviation introduced into the model by dropping improbable failure or recovery events. The effect of this modeling practice is a reliability prediction with less accuracy than is possible with additional modeling effort (when the model is solvable).

The specification of the system model is totally the responsibility of the modeler. Neither HARP/S nor any other computer program can guess the structure of the model that the user has conceptionalized. Many modeling subtleties are associated with the reliability modeling of fault-tolerant systems. This technology is new and many users are unfamiliar with the many concepts and modeling nuances that can make a significant difference in the results.

One such modeling nuance concerns the HARP/S multifault model for computing the near-coincident fault probability. HARP, MCI-HARP, and PM-HARP (collectively called $< HARP >$) do not model all combinations of multiple faults taken 2, 3, ..., N at a time exactly and automatically as XHARP does. Rather, they use another approximation technique (critical-pair multifault model) to further simplify the computational complexity and to ease the user burden of acquiring multifault data that is generally unavailable.

This automatically generated near-coincident multifault model (critical-pair) describes the condition that causes total system failure as a result of two coexisting faults (not simultaneously occurring, see section 2.7). The condition occurs when a system has already experienced one fault and is in the process of recovering from it when a second statistically independent fault occurs in another unit that is critically coupled to the one experiencing the first fault. If a second fault occurs during recovery but is not critically coupled (as specified by the user), the second fault is not accounted for in the coverage computation. It is accounted for in the redundancy exhaustion model.

An example of critically coupled units is two units in a voting triad that is performing a computation required for survival of the system as in a flight control system in a fly-by-wire aircraft.[5] The probability of a near-coincident fault is significant for highly reliable systems with system failure probabilities of less than (10^{-8}) for the mission time of interest. The near-coincident fault involving two faults is called a critical-pair fault. Most commercial and military aircraft flight control systems and most existing systems in commercial use today can be effectively modeled with the critical-pair fault model when the near-coincident fault is a mission critical factor. Systems using computers can have up to four active reconfigurable processing units where a majority vote can be effected until two coexisting faults occur.

A system with five processing units can survive two coexisting faults, but a third fault causes system failure. This system requires a critical-triple fault model to effectively predict the near-coincident fault probability. Because the HARP developers believed that the interest in modeling such systems is small, they did not implement a more complex multifault model; instead, they suggest the automatically generated critical-pair model as a conservative approximation to critical-triple or higher order models. The degree of conservativeness depends on the system architecture and can be unacceptably high for some systems.

The rationale to support the critical-pair modeling decision was based on the belief that the reliability of computers would continue to increase, making it less probable to have critical-triple faults. Consequently, predictions of ultrahigh reliabilities would be achieved with four or fewer

[5] A fly-by-wire aircraft uses an electronic computerized flight control system whose function is required for aircraft survival, and as such, the systems are usually fault-tolerant with redundant hardware units and possibly redundant software modules.

processors. This trend is in fact occurring which justifies the HARP developer's decision to only exactly model critical-pair faults (refs. 10 to 13).

For those applications requiring higher order fault models, the HARP developers suggest that the user modify the $< HARP >$ ASCII files that specify the multifault model exactly. (See section 2.7.4.) The modification can be accomplished with a common text editor before or after HARP has generated the appropriate files. XHARP is another alternative that provides automatic higher order multifault model generation. The $< HARP >$ multifault approximation model is further discussed in section 2.7, and an example is given in chapter 7. The user is cautioned to study these multifault models carefully before application. An incorrect selection of the multifault model options can produce a nonconservative result because a particular option can drop important failure modes from the reliability computation. However, $< HARP >$ cannot warn the user of this modeling specification error.

1.2.1. XHARP

More recently, the user has another modeling alternative. An extended behavioral decomposition model has been developed by researchers at Clemson University (ref. 5) and is implemented in XHARP. XHARP was designed to expand the modeling capability of the original HARP behavioral decomposition technique to include exact multifault modeling, multiple entry/exit fault model transitions, and automatic behavioral decomposition modeling. This capability is demonstrated in chapter 7. XHARP calls HARP, MCI-HARP, and PM-HARP as executable software programs; thus, the entire power of both XHARP and the $< HARP >$ programs are available to the user.

XHARP provides an X Window system environment for graphically specifying a semi-Markov chain that is automatically translated into the HARP structure for the fault-occurrence/repair model (FORM) and the fault/error handling model (FEHM).

1.2.2. PM-HARP

Phased-mission HARP was developed to facilitate the analysis of phased missions (refs. 4 and 14). A mission is phased when the structure of the system (configuration) or component failure distributions change after each epoch (phase) in the mission (refs. 15 and 16). Multiple phases of fixed and random durations are allowed. Also, the system can be specified to be imperfect at the beginning of a mission. The GO and HARPO programs are compatible with PM-HARP; however, the phased-mission specifications may not be specifiable to GO directly. HARPO may not graph all the phased-mission output data; however, the output listings are complete.

1.2.3. MCI-HARP

MCI-HARP is comprised of HARP with a Monte Carlo simulation engine and is fully integrated with HARP. MCI-HARP can solve all types of models that HARP can when the input is specified as a dynamic fault tree (the extended fault tree with sequence dependency gates that HARP accepts). At present, this capability excludes cyclic Markov models that can be specified to HARP in the Markov chain format that HARP accepts. However, MCI-HARP can solve certain model types that HARP cannot, such as non-Markovian models that arise when warm or cold Weibull spares are added to a Weibull fault/occurrence model. An important feature of MCI-HARP is the use of a variance reduction technique called importance sampling (ref. 17). Importance sampling makes it feasible to solve large models that contain widely separated time constants. Such models are called stiff and are common to highly reliable fault-tolerant systems. Although importance sampling is not a new technology, it has become more useful with the

recent application to preexisting Markov chain models (refs. 3, 18, and 19). Another important feature is MCI-HARP's ability to solve very large models with or without model truncation. This capability is possible because MCI-HARP does not store the entire Markovian state space.

1.2.4. Textual HARP

Three versions of textual HARP (PC-DOS HARP 16-bit version, PC-DOS HARP 32-bit version, and PC-OS/2 HARP) are available for operation on a personal computer. Textual HARP executes on Sun and DEC workstations with the same limits as the PC-DOS HARP 32-bit version.

1.2.4.1. PC-DOS HARP 16-Bit Version

We developed and tested the PC-DOS HARP 16-bit version on an IBM PC AT with 512K of memory and have successfully executed it on PC 286, 386, and 486 class machines. Because of memory constraints imposed by MS DOS's 640K memory limit, PC-DOS HARP 16-bit version cannot model large models. The limits on the various parameters defined in PC-DOS HARP are given in table 1. Expanded state size is possible with PC's that have more than 512K of memory by changing the limit sizes of the HARP package. (See section 5.3.)

Table 1. HARP Parameters

Parameter	Limit—	
	Textual HARP	
	32-bit version	16-bit version
Max. no. of states in Markov chain	Sorted: 10000	Sorted: 500
(may be larger if truncation is used)	Unsorted: 500	Unsorted: 500
Max. no. of transitions in Markov chain	Sorted: 90000	Sorted: 4500
	Unsorted: 2050	Unsorted: 2050
Max. no. of symbols in model	15000	500
Max. no. of factors in model	15000	1500
Max. no. of terms in model	15000	750
Line length in input file	80 char	80 char
Max. no. of characters in a parameter name	32 char	32 char
Max. length of rates and state names	12 char	12 char
Max. no. of nodes in fault tree	256	256
Max. no. of component types in fault tree	96	16
Max. no. of basic events in fault tree	96	16
Coverage value precision	No. depends on FEHM	No. depends on FEHM
Max. no. of incoming arcs per fault tree gate	70	16

Note that PC-DOS HARP 16-bit version allows the same number of nodes in the fault tree as the full model (32-bit) version of HARP. This flexibility takes advantage of the feature of truncation in HARP. However, since the maximum number of states allowed in *fiface* and *harpeng* of PC-DOS HARP 16-bit version is only 500, the user is advised to use a truncation level that restricts the model state size to less than 500.

In addition to being unable to solve large models, PC-DOS HARP 16-bit version has some other restrictions. Evaluation of the simple bounds is not possible. Also, Weibull failure rates are not allowed. State-dependent coverage factors cannot be used; hence, no near-coincident-fault calculations are performed. This restriction occurs because the failure probabilities due to near-coincident faults are comparable with the precision allowed by the PC 16-bit version.

A graphical interface to HARP has been developed with Graphics Software System's implementation of the Graphical Kernel System (GKS) ANSI standard. An IBM PC AT with EGA controller and display was used for the development, and VGA is now supported. For more information, refer to the *HARP Graphics Oriented (GO) Input User's Guide* (vol. 3 of this TP) and the *HARP Output (HARPO) Graphics Display User's Guide* (vol. 4 of this TP).

1.2.4.2. PC-DOS HARP 32-Bit Version

The full HARP capability has been ported to a PC under DOS with Microsoft FORTRAN Powerstation, and it behaves identically to the UNIX and VAX HARP versions. This extended DOS version requires a 386 or higher class machine.

1.2.4.3. PC-OS/2 HARP

Other operating systems such as UNIX or OS/2 executing on a PC remove the 640K memory restriction and hence the restriction on model size. The full HARP capability has been ported to a PC under OS/2 and behaves identically to the UNIX and VAX HARP versions. Because of the unique relationship between DOS and OS/2, ASCII files are totally interchangeable. Thus, it is possible to execute the GO program under DOS, to execute the full model HARP capability under OS/2 using the files created with GO, and to graphically display the HARP results executing HARPO under DOS again. The advantage of this arrangement is that a DOS-compatible GKS program need not be upgraded to OS/2 GKS. Also, small models can be worked entirely on a 286 PC, or just the graphics can be displayed on the PC when an OS/2, UNIX, or VAX computer is necessary for large models.

1.3. HARP/S Key Features and Overview

The HARP/S key features are summarized as follows:

- Very large system modeling (using MCI-HARP or behavioral decomposition and bounds with truncation with HARP)
- Flexible method of modeling dynamic behavior (homogeneous/nonhomogeneous Markov chains)
- Automatic Markov chain generation from a fault tree description (particularly useful for large systems) or direct user input of the Markov chain
- User choice of seven fault/error handling models ranging in complexity from a simple laboratory parameter estimation model to a complex Petri net model for detailed fault/error handling analysis
- Automatic insertion of fault/error handling models into Markov chains
- Automatic parametric analysis
- Phased-mission analysis
- Non-Markovian models with Weibull cold and warm spares
- Written in ANSI standard FORTRAN and successfully ported to many different host computers, including IBM-compatible 286, 386, and 486 PC's (including AT&T 6300 with 640K), DEC VAX, Sun, CRAY Y-MP, Alliant, Convex, Encore, Gould, Pyramid, and Apollo
- Runs under MS/PC-DOS and Microsoft Windows NT, OS/2, DEC VMS and Ultrix, Berkeley UNIX 4.3, and AT&T UNIX 5.2

- Interactive graphical input/output workstation capability for DEC VAX under VMS, Sun, and IBM-compatible PC
- X Window system graphical model generation
- Extensively and independently tested and applied to practical systems (over 8 years of industry beta testing and over 100 copies distributed)
- Independently tested and evaluated within NASA

HARP provides the user with a language to input a model and solves it for the system reliability/availability for user-specified mission times. It uses *behavioral decomposition* to avoid the problems of model largeness and model stiffness (refs. 6 and 7).

The reliability model is decomposed along temporal lines into a FORM and a FEHM. The FORM contains information about the structure of the hardware redundancy, about the fault arrival processes, and about manual (off-line) repair. The user specifies the FORM either as a fault tree or a Markov chain. The FEHM (often called the coverage model) allows for permanent, intermittent, and transient faults (ref. 20), and models the (on-line) recovery procedure necessary for each type. The FORM/FEHM models are merged according to a user-specified multifault model. The resulting system reliability/availability model is a simplification of the originally specified model. The correct specification of the multifault model is crucial for HARP to produce a conservative result.

HARP also accepts as input a nominal value and a variation on all FORM input parameters. The nominal value is used for the reliability prediction, and the variation about the nominal value is used in an approximate (simpler) model to generate bounds about the predicted reliability. Additionally, HARP supports the modeling of time-dependent failure rates by allowing a symbolic failure rate to be associated with a Weibull failure distribution. We caution the user that the use of Weibull distributions leads to a long solution time because the symbols must be reevaluated at each time step, but it can also lead to a more accurate model of the system under study (ref. 21). MCI-HARP's simulation has been shown to be more efficient in solving Weibull models than a numerical integrator (ref. 3). A new feature in HARP is the use of *state truncation* to further avoid the problem of large models.

Input data to reliability models can be inaccurate by as much as hundreds to thousands of a percent (ref. 22). Because of these large errors and the recognition that reliability modeling is more often an art than a science, the user of HARP must view the results with a healthy dose of caution. When trade-off studies are performed with comparable input data, the computed results are meaningful relative to the models being compared. If the user is interested in arriving at absolute reliability predictions, then much caution should be used in the interpretation of the computed results. HARP outputs eight digits plus an exponent. The eight digits are displayed for the purposes of user calibration, that is, to determine whether the user's computer is computing the results that the developers intended. The eight digits do not imply reliability prediction precision to eight digits.

Experience has shown that next to tedious hand calculations, the most practical method of developing confidence in the results computed by any reliability predictor is to compare the computed results of one program with those of a different reliability program. This recommendation is based on the authors' interaction with beta test site users over 8 years where on several occasions, coding bugs were discovered as a result of the user's comparison of HARP results to an in-house company reliability program including some obtained from NASA (such as CARE III)[6] and other institutions. The HARP developers also used this technique

[6] Computer Aided Reliability Estimation third generation computer program.

extensively (refs. 23 to 26). Some examples of these test cases are sent to beta test site users with the HARP code. Over 700 test cases comprising over 2000 files were used to test HARP at Langley during development. A greater number was used at Duke University.

Textual HARP executes on DEC VAX workstations under VMS, Sun workstations under UNIX, and IBM-compatible 286, 386, and 486 PC's under MS DOS. Under OS/2 on 386/486 PC's, the full HARP capability can be executed. Recompilation only requires an ANSI standard Fortran 77 compiler, and textual HARP has been compiled with Lahey and Microsoft Fortran on the PC. It is compatible with a wide range of computing platforms because it was written in ANSI standard Fortran 77 for wide portability. HARP creates ASCII files that are compatible with most computing platforms. For example, files created under the PC environment can be executed by a VAX workstation. In this way, a PC can be used as a workstation for input and output processing, and a VAX workstation can be used for large system number computations. Textual HARP has an interactive prompting input capability and is composed of three stand-alone programs: *tdrive*, *fiface*, and *harpeng*. (See fig. 2.) As the user successively executes the programs in this order, the programs create files that are required by other programs.

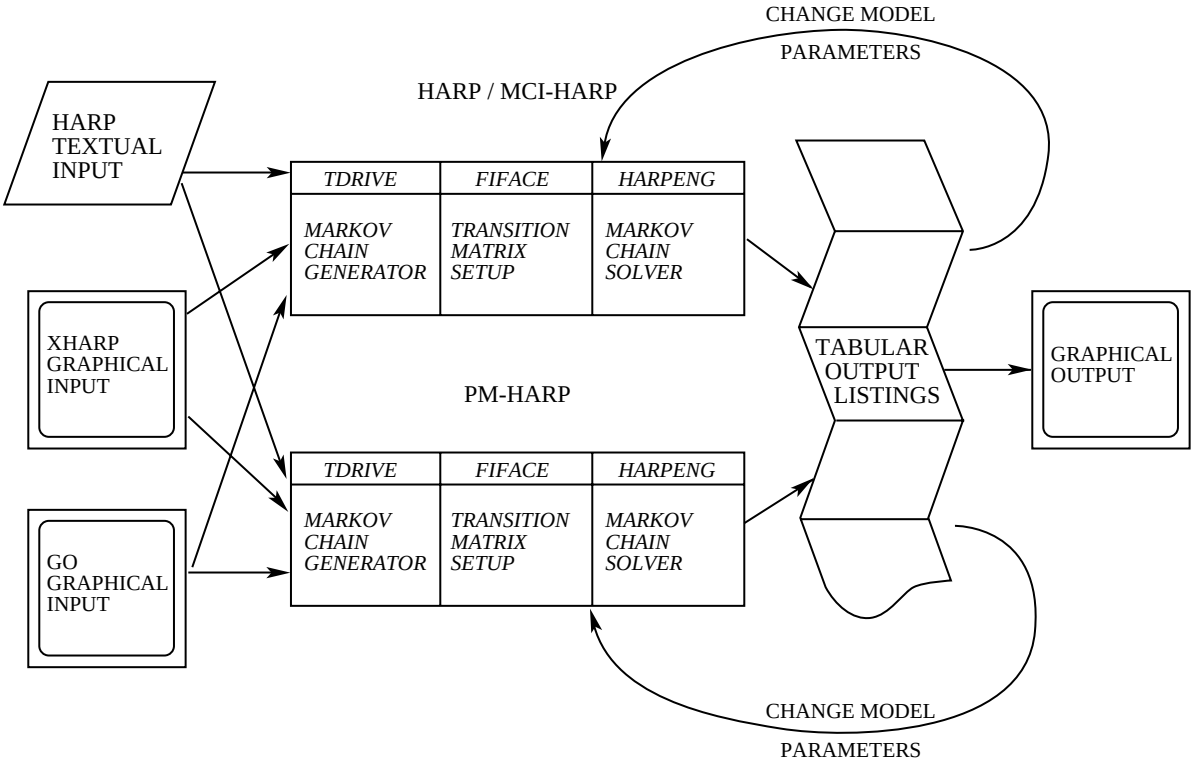


Figure 2. HARP execution flow and relationship to GO and HARPO.

The programs also accept files created with a text editor. Thus, the user can use the interactive input capability or simply input text files. The input to *tdrive* can also come from files generated by the GO program. The output of textual HARP are tabular structured files. These files can be used as input to HARPO, which allows the user to graphically display the HARP tabular data in a wide variety of forms in an interactive mode. Thus, as an overview, textual HARP is by analogy the central processing unit, the GO program is a graphical input to textual HARP that bypasses textual HARP's interactive input-prompting capability, and HARPO is the graphical output processor that reads textual HARP's tabular output files. Separate users

guides for GO and HARPO (vols. 3 and 4 of this TP) and a tutorial (vol. 2 of this TP) are also available.

Table 1 gives the limit on various parameters defined in HARP. It gives the user an idea of the model sizes that can be effectively handled by HARP; however, many of these limits can be altered. (See chapter 5.)

The number of symbols in the model refers to the sum of the number of distinct failure rate symbols, number of distinct coverage factors, and the number of states having symbolic names. The number of factors and terms in the model can be explained as follows. If a transition in the MODELNAME.INT file is $3 * \lambda + 2 * \mu * \mathrm{C1}$, then there are two terms ($3 * \lambda$ and $2 * \mu * \mathrm{C1}$), five factors (3, λ, 2, μ, C1), and three symbols (λ, μ, C1) associated with the transition. The arrays containing the terms and factors are used dynamically. In other words, the arrays, which are filled symbolically, are converted to the numeric representation when they are full. The space is then reused.

1.4. HARP Version 6.0

HARP version 6.0 (April 1989) has the following additional features. Four novel dynamic gates have been included in the description of the fault tree. They are the functional dependency gate, the priority and gate, the cold spare gate, and the sequence-enforcing gate. (See sections 2.5 and 4.4.) These gates can be used to model those features of a system frequently characterized as sequence dependencies, which cannot be modeled by the standard and, k/n, and or gates. Hence, they greatly enhance the modeling capabilities of HARP.

Another feature added to HARP is the capability to solve stiff models with a special stiff solver. A stiff solver is automatically invoked in *harpeng* if GERK (nonstiff ordinary differential equation solver) is taking too many steps to solve the model. (See section 3.3.2.)

The format of the MODELNAME.INP file created by *harpeng* during run time has been altered to make it easier for the user to read and modify. The new format makes it much more robust. (See section 4.5.) However, *harpeng* still reads in the old format of the MODELNAME.INP file. A small change has been made in the questions asked by *harpeng* in that the last two questions have been combined into one. A simple run through *harpeng* clearly demonstrates these changes.

In *fiface*, the user can now choose to create only the relevant near-coincident fault rate files. In the older versions, all near-coincident fault rate files were created by default. A few warning and error messages have been altered to clarify their meaning.

1.5. HARP Version 6.1

HARP version 6.1 (November 1989) has the following modifications. The dynamic sequence-enforcing gate (see sections 2.5 and 4.4) now requires that all inputs to the gate except the first must be basic events (possibly replicated, i.e., multiple basic events with identical failure distributions). The first input remains unrestricted and thus unchanged from version 6.0. The cold spare gate now allows its inputs to be (possibly replicated) basic events, where earlier only unreplicated basic events were accepted.

The output MODELNAME.RES file from program *harpeng* has been renamed MODELNAME.RS*, where * is an integer from 1 to 9 for compatibility with HARPO. For each run of the same input file, the integer is incremented, beginning with 1.

In addition, the method for determining the near-coincident fault rates is changed. The new method is conservative in that it regards outgoing arcs of the source and target states. In

previous versions, only the rates of the target state were utilized. (See section 2.7 for details.) Carriage return selected defaults are fully implemented, more extensive warning messages have been added to the code, and some coding bugs have been fixed.

1.6. HARP Version 7.0

HARP version 7.0 (February 1993) has added two new fault tree gates: the invert (inv) and exclusive or (xor) gates. The results file now reports the number of transitions in the Markov chain. This information is useful if the maximum number of covered failures attained by the Markov chain states exceeds 90 000. Changes to max# are made in *fiface* and *harpeng*, and are useful if *tdrive* runs out of memory in the POOL(*) array (i.e., increase the value of PLEN or rerun *tdrive* with a larger truncation value). (See section 5.3.) When *tdrive* is invoked and it senses the presence of existing files, *tdrive* queries the user to select one of several choices. One choice is AT. AT allows the user to append additional fault tree (FT) nodes to an existing tree in MODELNAME.TXT, but only if the last existing node is not FBOX. If the last node in the MODELNAME.TXT file is an FBOX node, the MODELNAME.TXT file should be edited to delete the line containing the FBOX node text prior to executing *tdrive*. This feature is useful for huge fault tree files when during entry something goes wrong after many nodes have been entered. This feature also precludes the need for the user to recreate the entire file. If the user specifies AT, *tdrive* reads the contents of the existing MODELNAME.TXT file, prints the last FT node in the file, and then allows the user to enter the next FT node as usual. A number of bugs were fixed and are delineated in appendix A.

1.7. HARP Quick Reference

1.7.1. Summary—HARP Capabilities and Limitations

HARP is intended for reliability analysis of reliable fault-tolerant systems with complex recovery management techniques, particularly those used in flight control systems. The following sections provide a list of the capabilities and limitations of HARP that were determined during beta testing. Certain listed capabilities cannot be accomplished during the same execution of HARP; thus, a fault tree and a Markov chain input specification of the same system would not both be processed in the same HARP run. However, if a fault tree is specified, HARP creates the equivalent Markov chain and solves the Markov chain. The converse is not true. The limits on the size of the problem that HARP can solve depends on the system.

1.7.2. HARP and MCI-HARP Capabilities

- Dynamic fault trees with repeated nodes (i.e., shared basic events)
- Repairable systems (to determine instantaneous availability), which are specified with a Markovian FORM
- Systems with sequence-dependent failures as dynamic fault trees of Markov chains
- Weibull failure distribution including hot spare repairable systems
- Weibull failure distribution with cold Weibull spares (MCI-HARP)
- Provide guaranteed automatically generated parametric bounds on system reliability (for a large number of applications of practical interest and all Markov models with ASCII file editing)
- Provide detailed coverage modeling with a choice of FEHM's

- Automatically solve arbitrary Markov chains for most practical systems and, with file editing, all Markov chains
- Solve truncated models for large systems given as a fault tree and large models with or without truncation using MCI-HARP
- Systems with cold and warm spares (refs. 27 to 30)

1.7.3. HARP Limitations

- Mean time to failure (MTTF) or mean time between failures (MTBF)
- Steady-state evaluation
- Weibull failure distribution mixed with constant failure rates in repairable systems
- Bounds analysis for systems with Weibull rates or with no absorbing states
- Automatic generation of Markov chains for repairable systems
- Phased missions; use PM-HARP (ref. 2)
- Weibull failure rate for stiff systems (use MCI-HARP)
- Weibull failure rate for models containing the cold spare gate and warm spares (see section 2.9.1); use MCI-HARP (refs. 18 and 19)
- Slow recovery with behavioral decomposition
- Model systems whose unreliability is less than (10^{-15}) when FEHM models are included (unless the epsilon variable parameter EPX is changed, see section 3.3.3)

1.7.4. About Volume 1

Volume 1 of this Technical Paper is a user's guide for the textual HARP program, which textually and interactively prompts the user for keyboard entered input data. This volume is divided into the following chapters:

Chapter 2 discusses the various steps needed to completely specify a system in HARP. It also discusses the different FEHM types available. Chapter 3 presents the solution techniques used in HARP; and chapter 4 presents an overview of the HARP program and the files it generates along with the user input. Chapter 5 provides practical information about the HARP program. Chapter 6 gives a mathematical description of the nonstandard fault tree dependency gates and chapter 7 illustrates some advanced modeling techniques. Appendix A lists known bugs in versions 6.1 and 6.2, and appendix B lists warning and error messages. The tutorial (vol. 2 of this TP) steps the user through several examples and further explains many of the HARP concepts. Additional applications can be found in references 26 and 29 to 32.

Chapter 2

Model Specification

2.1. System Mathematical Model Overview

The reliability model that HARP actually solves is always a Markov chain even though it can be input as a combinatorial or sequence-dependent fault tree. Depending on the user's choice, the model can be a homogeneous (only exponential failure distributions, i.e., constant failure rates) Markov chain or a nonhomogeneous (at least one Weibull failure distribution, i.e., nonconstant failure rate) Markov chain. Although these general stochastic models cover a wide range of systems, some systems require even more general and computationally more difficult stochastic models. These systems are the highly reliable fault-tolerant variety that use redundant subsystems for increasing system reliability. These systems often use computers for real-time-processing control and system management of failed redundant components.

Because failure recovery requires either a random or possibly a deterministic time, a second component failure can occur while the first is being properly dealt with. The second fault is called a near-coincident fault. The reliability/availability of highly reliable fault-tolerant systems is sensitive to the near-coincident fault and is typically the dominant unreliability contributor. To capture the effects of this important parameter, a semi-Markov chain model is required to account for the system holding time during system recovery (recovery time) when fault occurrences are exponentially distributed. If fault occurrences are Weibull distributed, the stochastic model becomes a mixed-Markov chain, an even more complex mathematical model. These stochastic models are computationally costly to solve in the traditional manner and thus severely limits the size of the model. Thus, HARP was designed to model these systems efficiently.

2.2. Fault/Error Handling Mathematical Model

A mathematical technique that significantly simplifies the solution of both these models is called behavioral decomposition. The technique makes use of the fact that fault occurrence times are typically on the order of thousands to tens of thousands of hours, while fault recovery times are on the order of fractions of a second to seconds. This disparity of event times makes it possible to solve a FEHM in isolation with respect to the FORM. The solution of the FEHM model determines the internal (to the FEHM) race condition times of exit from the FEHM and are expressed as exit probabilities and holding times. Timing considerations are carefully modeled within the FEHM. Once the exit probabilities and holding times are thus determined, the behavioral decomposition model assumes that the recovery outcome (FEHM exits) happened in zero time. The exit probabilities behave as exit path switches with infinitely fast switching speed. These switch probabilities are often called coverage probabilities in the literature. The coverage probabilities are automatically incorporated into the Markov chain (i.e., homogeneous or nonhomogeneous model) and solved with a straightforward ordinary differential equation solver (GERK).

HARP and XHARP offer two classes of FEHM's: single-fault and multifault models. The single-fault model capability ranges from simple to complex, while the multifault model capability is relatively simple. It uses a near-coincident model that causes system failure resulting from user-specified synergistic critical-pair faults. In contrast, XHARP has a near-coincident multifault model that is general and removes the critical-pair faults restriction of HARP. The more general model in XHARP is especially useful for systems where many fault containment regions are modeled and more than two near-coincident faults can be tolerated.

2.3. Implementation of System Mathematical Model

The stochastic model that is actually solved, called an instantaneous jump model, is a computationally efficient approximation of a more complex mathematical model. HARP's fundamental mathematical model (behavioral decomposition) guarantees that the instantaneous jump model approximation is conservative (ref. 8). When the user specifies the correct model, the unreliability prediction is always greater than the exact result from the full model. The user must specify the correct stochastic model to HARP to guarantee a conservative computation. An incorrectly specified model may not guarantee a conservative result or may produce an overly conservative result.

Figures 3 and 4 show the relationship between the different models involved in the HARP modeling process. Figure 3 shows the use of behavioral decomposition, and figure 4 shows the explicit specification of Markov chains.

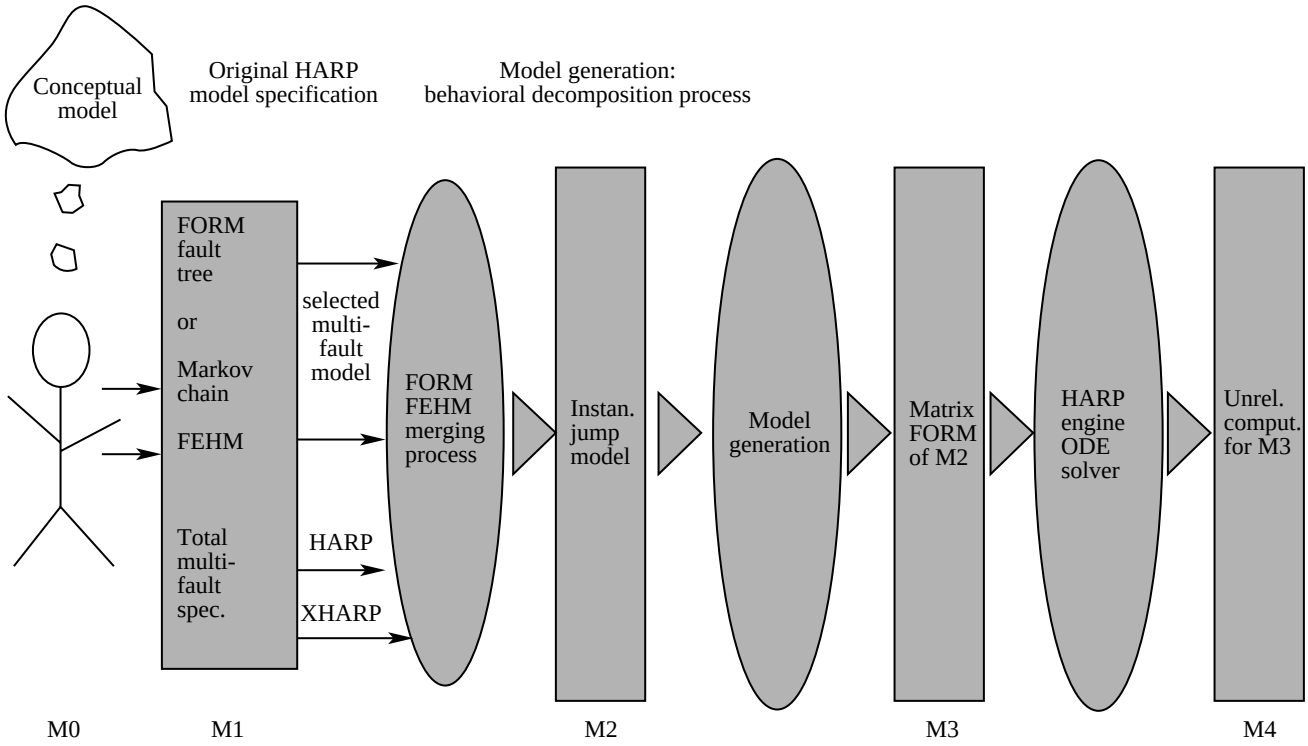


Figure 3. Relationship of modeling considerations with behavioral decomposition.

2.3.1. Behavioral Decomposition Model

The process of reliability/availability modeling begins with the user's conceptual model (also called the original or full model), which the modeler has formulated from the system under consideration. This process is not well understood and further consideration is beyond the scope of this document. The user translates the conceptual model $M0$ into the HARP paradigm $M1$ by using the FORM/FEHM and the near-coincident multifault model specification (fig. 3) or by entering it directly as a Markov chain (fig. 4). The choice of which notation to use is multifaceted and depends on the user's inclination and modeling familiarity, the need to model near-coincident faults, and modeling complexity.

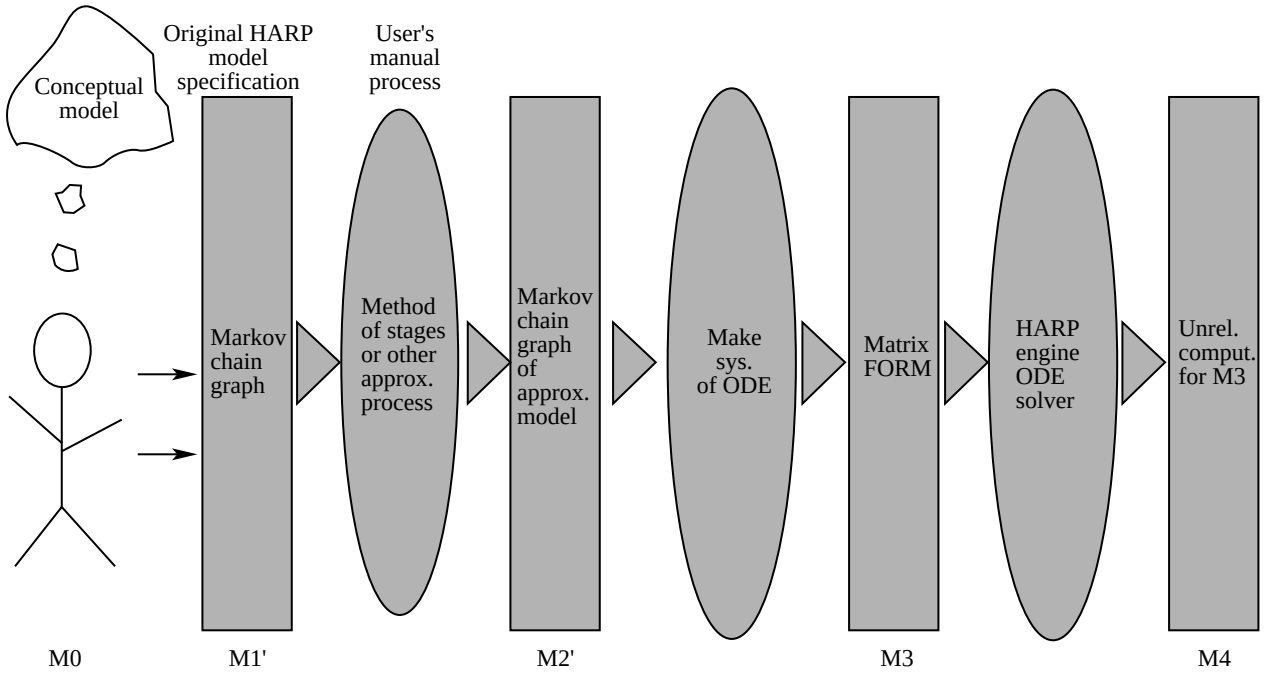


Figure 4. Relationship of modeling considerations with direct Markov chain modeling.

2.3.2. Markov Chain Model

Direct entry of Markov chain for simple models requiring no near-coincident fault modeling is a reasonable choice as is the use of the fault tree notation shown in figure 3 without behavioral decomposition. When near-coincident faults are needed, direct Markov chain entry becomes more complicated because it may be necessary to use the *method of stages* if nonexponential fault holding time distributions are required. Otherwise, using direct Markov chain entry is limited by the complexity of the model. The problem is one of model specification, not solution. The specification of a complex Markov chain is tedious and error prone.

2.4. Important Modeling Considerations

The use of the HARP capability shown in figure 3 is more convenient than direct Markov chain entry. The use of behavioral decomposition eliminates the need for the method of stages, but the modeling specification of complex Markov chains is still present even though the FEHM modeling is simplified. Complex models require the use of the fault tree notation that is automatically converted into a Markov chain, which can include fault/error handling as specified by the user. For this modeling convenience, the user must insure that $M1$ is the correct model that represents the conceptual model.

$M1$ is a notational model. These user-specified inputs result in the creation of a new and simpler model through the process of FORM/FEHM merging to create the instantaneous jump model ($M2$). If the original model is an N-plex (one that degrades by one component per component failure), the instantaneous jump model is automatically generated and is exact; otherwise, it is an approximation. Editing the automatically generated ASCII files also produces an exact model for any Markov chain. Model $M2$ is given to an ordinary differential equation solver that translates $M2$ into model $M3$ for solution. Complex models often require several model translations to make the solution tractable. Errors are introduced in the translation

process, but the objective is to estimate the total error and report the desired result in light of that error.

A useful result has an error that is small with respect to the result. The usefulness of the result with its estimated error also depends on the accuracy of the input data and the user's model $M0$, e.g., rationale, assumptions, and simplifications.

Some observations of the HARP modeling process are worthy of further discussion to galvanize this modeling process in the user's mind and preclude the misapplication of the program. No computer program can provide model $M0$. This process can be aided, but not by HARP directly. In the process of specifying the HARP paradigm, however, the user's concept of $M0$ is often crystallized. The user must insure that model $M1$ is equivalent to model $M0$ or is an acceptable approximation. HARP nor any other program cannot infer that $M1$ and $M0$ are comparable.

Examples of $M1$ models are shown in chapter 3. The specification of the near-coincident multifault model is also a part of $M1$ but is not shown graphically. Because the FEHM models can be non-Markovian, model $M1$ can also be non-Markovian. Behavioral decomposition enables a transformation from a non-Markovian model (which can be extremely difficult to solve with conventional techniques) into a Markovian model with a significant potential for state-size reduction. This transformation is the FORM/FEHM merging process shown in figure 3. The resulting transformed model ($M2$) is always a Markovian model. Model $M2$ is automatically and mechanically produced by HARP and is a mathematical approximation of model $M1$. Examples of transformed $M2$ models are shown in chapters 3 and 5. The McGough-Trivedi theorem (ref. 8) guarantees that $M2$ is mathematically conservative with respect to $M1$. By forcing the FEHM events to occur in zero time, the net effect is that a greater probability of transition to a system failure state occurs than would occur in reality. This property, which is the essence of the McGough-Trivedi proof, guarantees that Markovian model $M2$ produces a conservative system failure prediction with respect to model $M1$, the possible non-Markovian model. For this remarkable achievement, a conservative approximation error is introduced. The magnitude of the error depends on the disparity of the FEHM and component event times. For highly reliable fault-tolerant system models, the error is significantly smaller than the component failure rate data errors.

The most crucial step in the HARP modeling process is to insure that $M0$ and $M1$ are equivalent or acceptable approximations. When using HARP, the specification of $M1$ is a manual process. However, XHARP automatically creates $M1$ from the user's full-Markov chain and insures that the proper fault models are used and that no failure modes are dropped. Because XHARP requires the specification of the full-Markov chain, the size of the model is limited to the user's endurance, which results in models of under 50 noncoverage states.

The HARP model generation capability, in contrast, is capable of modeling an enormous number of states; however, insuring that no significant failure modes are ignored in $M1$ requires special consideration by the modeler when specifying the near-coincident failure model. (See chapter 7.) Three multifault models are offered for the specification of the near-coincident fault rate to cover the majority of applications of practical interest. In section 2.7, these models are called the ALL-inclusive, SAME-type, and USER-defined. The multifault models are exact and automatically generated for N-plex systems when $N \leq 4$ and are approximations to higher order systems. If greater accuracy is required for the higher system models, some manual intervention is required by the user. Using a text editor to edit the ASCII files generated in HARP is necessary. Another alternative is to use XHARP, which provides an automated higher order model generator. The specific models specified by the user depends on the particular system architecture and whether the modeler intends to achieve a conservative reliability/availability result or a nonconservative one. Specifying the wrong multifault model to HARP during the $M0$ to $M1$ specification can produce a nonconservative result because the incorrectly specified

multifault model can drop important failure modes from $M0$. The user must understand how $M0$ is conceptually related to $M1$. (See chapters 3 and 7.)

Another issue of importance when considering the use of these models is the degree of conservatism or nonconservatism produced by the use of the multifault models. The trade-off that the user must grapple within any modeling exercise is one of modeling complexity versus computational accuracy. Because the HARP developers intended HARP to be applied to practical systems, many mathematical techniques were used to reduce model complexity and still provide useful results. Behavioral decomposition and the HARP multifault models were selected to achieve this purpose. Another extremely useful model reduction technique is the Markov model truncation scheme described in section 2.8. This technique is especially useful for solving extremely large Markov models that can easily result from a modest looking fault tree.

The advantage of behavioral decomposition is that fault/error handling modeling, no matter how complex the FEHM's and no matter how many FEHM's are included, contributes at most two additional Markov chain states. The savings in computation for typical systems of interest can be substantial and makes it entirely feasible to model intricate fault/error handling detail even when the FEHM itself is non-Markovian as is the case for the Extended Stochastic Petri Net (ESPN) FEHM model or when deterministic (constant) recovery times are specified in a number of HARP FEHM models.

The disadvantage of this scheme is that the disparity of the FEHM and FORM event times affects the accuracy and hence the degree of conservatism of the HARP predictions. The farther apart the event times, the more accurate the results become. Also as the event times approach each other, the accuracy decreases but the deviation always accumulates on the conservative side. For typical highly reliable systems, the time disparity is six or more orders of magnitude, virtually insuring a result much more accurate than the input data accuracy could ever justify.

As an example of a worst case for disparity, a two triad system with processor failure rates of 10^{-4}/hr showed a conservative deviation of about 80 percent when the event times were on the same order of magnitude. Please note the significance of this model: The recovery time is about the same as the expected time to failure of one component. This system is hardly realistic, but HARP still yields an acceptable result. As the time disparity increases, the accuracy increases. One order of magnitude difference in time disparity produced a deviation of 6 percent and 0.5 percent with two orders difference. On considering that failure rate values can be in error by hundreds to thousands of a percent (ref. 22), the relative deviation resulting from behavioral decomposition even for this pathological example is minuscule.

The advantage of using HARP's multifault models is that for the majority of practical systems (up to four critically coupled units), an effective model is automatically generated by HARP. For systems with more than four critically coupled units, HARP produces less accurate but always conservative results. (See section 2.7 for selection of conservative fault models.) HARP's multifault models are also easy to specify and no further (usually unavailable) data are required.

The user has two alternatives if the conservative deviation is unacceptable. Manual editing of the HARP generated ASCII files allows the specification of detailed multifault models for more accurate predictions. The XHARP program (see section 1.1) contains an automatic model generation capability that includes a detailed multifault model that produces more accurate coverage computations than HARP. The XHARP multifault model places no additional computational load over HARP but requires more input data from the user (ref. 5). The trade-off of using these two extended techniques relates to the model size. XHARP requires a Markov chain input specification, which for large models can be tedious for the user to input. On the other hand, HARP generates a large model from a fault tree that is relatively easy to specify, but some file editing is necessary to accurately model the more complex multifault model. When one needs the modeling power of the HARP FEHM's, no other easier alternative presently exists.

As stated previously, behavioral decomposition can also be applied to mixed-Markov models. Although no theorem has yet been proven that guarantees a conservative result for this model, a nonconservative result of a practical system has not yet been demonstrated, but caution is advised. The following section describes the various steps necessary to completely specify a system to be modeled by HARP.

2.5. Fault-Occurrence/Repair Model

The fault-occurrence/repair model contains information about the structure of the system (how many components of what type and interconnected in what way) and about the fault arrival and repair processes (how often does each component type fail and how long does it take to fix it). This information can be entered either as a Markov chain or as a fault tree (in the case of nonrepairable systems), depending on whether a state-space based for small models (limitation is imposed by user willingness to input the model manually) or a fault tree representation of the system (for large models) is more appropriate.

A Markov chain is entered as a state-transition rate diagram, in which each state represents a particular configuration of the system. Transitions between states represent units failing or being repaired. A fault tree is a model that graphically and logically represents the various combinations of events occurring in a system that can lead to system failure (ref. 33). The fundamental logic gates of fault trees allowed by HARP are the and gate, the or gate, the k/n gate, the inv gate, and the xor gate. A k/n gate is used when the occurrence of k or more of n possible events cause failure. The basic events in the fault tree represent failure of the components that form the system being modeled.

When the FORM is entered by the user for either a fault tree or a Markov chain, the component failure rates are initially specified in symbolic form as symbolic failure rate names. Numerical values are requested later when *harpeng* is executed or in some cases by *fiface*. This scheme allows for the efficient solution of the model for performing sensitivity or trade-off analyses when several sets of numerical data are examined. The specification of symbolic failure rate names should avoid the use of special characters as these can often interfere with the user's operating system. In particular, do not use the symbols \$ or &.

Four dynamic dependency gates are available for modeling sequence dependencies. Because HARP automatically converts a fault tree into an equivalent Markov chain for solution (see chapter 6), the addition of the dependency gates is a natural extension to the more common combinatorial fault tree gates. Many applications have demonstrated their modeling power (refs. 18, 19, 32, 26 to 28, and 34 to 36). Chapter 6 presents a mathematical description of the dynamic gates for the user's in-depth investigation of powerful properties.

The functional dependency gate has one input, the trigger input, one or more dependent events, and a normal output. (See fig. 5.) The input event can either be a basic event or the output of some other gate. The dependent events are basic events that depend on the trigger event. When the trigger event occurs, the dependent basic events are forced to occur. The occurrence of any dependent basic event has no direct effect on the trigger event. A functional dependency gate is useful when the occurrence of some event (say a node failure) causes some other components to be unusable (e.g., sensors that can be connected to the node). For this case, the sensors are considered to have failed (but no coverage model is invoked). The nondependent output from the dependency gate reflects the status of the trigger event. This output is provided to enhance the drawing of large fault trees, and it can be used instead of the trigger input as an input to some other gate to simplify the drawing of the tree.

The priority and gate is essentially an and gate with two inputs with the added restriction that the input events have to occur in order. If the two inputs are A and B (fig. 6), then the

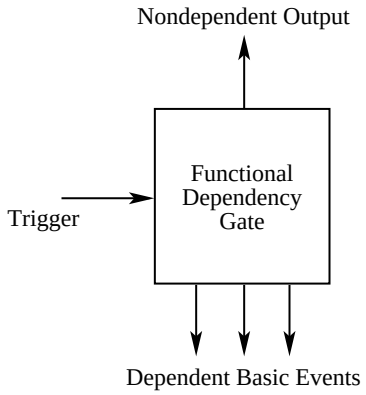


Figure 5. Functional dependency gate.

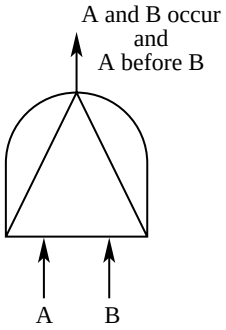


Figure 6. Priority and gate.

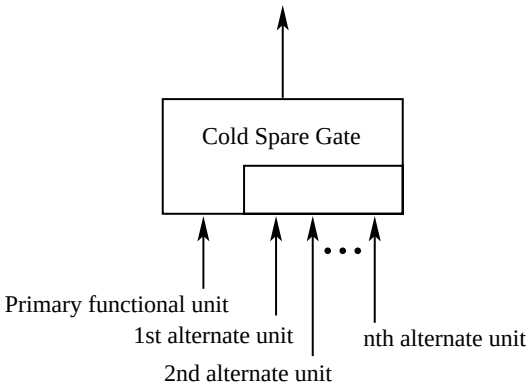


Figure 7. Cold spare gate.

priority and gate fires if both the input events occur and event A occurs before event B. The gate produces no output if event B occurs before event A.

The cold spare gate has one primary input (the primary functional unit) and one or more cold spares. (See fig. 7.) All inputs to the *cold spare gate* must be (possibly replicated) basic events.[7] The gate fires when all input events have occurred, and the gate actually controls the failure ordering of the alternate units. The active unit always fails stochastically while the alternate

[7] A replicated basic event represents multiple failure events having identical failure distributions. Using this replication notation significantly reduces HARP generated Markov models.

units are precluded from failing until they become operational. When the primary input unit fails, it is replaced by the first designated alternate unit. The alternate functional units (or cold spare units) are not allowed to fail when they are dormant.[8] However, this rule has one exception. If a cold spare is functionally dependent on another component (i.e., it is a dependent event of a functional dependency gate), the cold spare may actually be unavailable (because of the occurrence of the trigger event) when needed. Hence, a cold spare gate does not prevent one of its spares from being caused to fail by a functional dependency gate. Note also that a spare component can be shared by two or more cold spare gates (i.e., pooled spares are possible).

The sequence-enforcing gate[9] is similar to the cold spare gate but has some unique, important, and subtle properties not present in the cold spare gate. The sequence-enforcing gate controls the ordering of events in a manner similar to that of the cold spare gate. That is, the input events are constrained to occur in the left-to-right order in which they appear under the gate (i.e., the leftmost event must occur before the event on its immediate right, which must occur before the event on its immediate right is allowed to occur, etc.). There can be any number of inputs (see fig. 8), the first of which can be a (possibly replicated) basic event or the output of some other gate. All inputs other than the first are limited to being (possibly replicated) basic events. The sequence-enforcing gate differs from the cold spare gate in the way they treat shared events. Although the specification of the gate is straight forward, its modeling implications are not. The effect of failures associated with this gate can be local (relative to the component) or global (relative to the entire Markov chain). In some cases, the sequence-enforcing gate can be used to describe state-dependent FEHM's in a fault tree. (See section 4.7 for the concepts and chapter 6 for an example.)

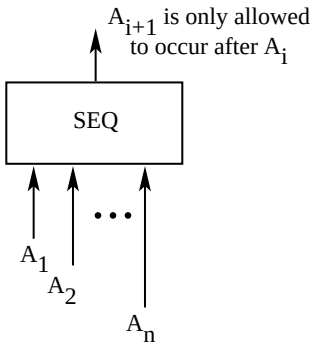


Figure 8. Sequence enforcing gate.

Note the restrictions on the inputs of the four dynamic gates previously described. All inputs to the cold spare gate must be (possibly replicated) basic events. In the functional dependency gate, the trigger input can either be a basic event or the output of some other gate, but the dependent events must be (possibly replicated) basic events. The priority and gate has no restrictions on the two inputs. They can be basic events or the output of some other gate. Thus, two or more priority and gates can be cascaded for more than two sequence dependent inputs. In the sequence-enforcing gate, all inputs except the first must be (possibly replicated) basic events. The first input can be a basic event or the output of some other gate. Like priority and gates, sequence-enforcing gates can also be cascaded. Both gates are cascaded from the left. Note that the gates cannot be cascaded from the right. (See fig. 9.)

The inv and xor gates were also implemented. The inclusion of these gates into a fault tree produces a noncoherent model that can cause the inexperienced modeler to generate

[8] This modeling assumption is often useful to arrive at a best case scenario to give an upper bound on reliability.

[9] In earlier publications, this gate is also called a *sequence gate*. The word *enforcing* was added to emphasize its function.

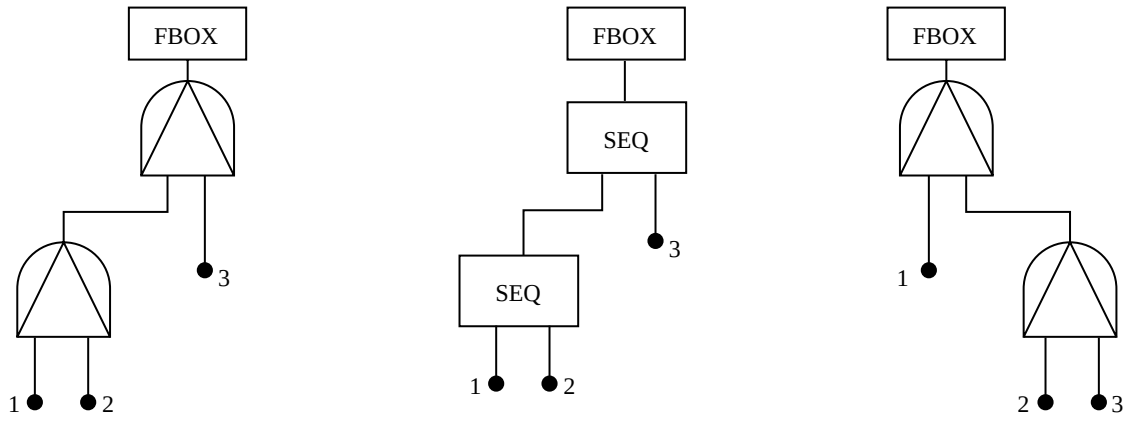


Figure 9. Cascading priority and gate and sequence-enforcing gate.

unexpected results. For example, if the top gate in a fault tree is an inv gate, HARP reports reliability probabilities (numerics) as unreliability probabilities in the output file. The values are correctly computed, but the inv gate alters the meaning of the reporting labels. These two additional gates give the user an extensive modeling capability; for example, researchers at Duke University proved that the set of HARP gates establishes a mapping into the entire noncyclic (nonrepairable) homogeneous Markov chain state space. (See ref. 36.)

Using HARP as a combinatorial fault tree solver without FEHM's is computationally inefficient, although convenient for the user accustomed to HARP. When a fault tree contains sequence dependencies, HARP provides a unique solution technique that can be difficult or computationally expensive to achieve otherwise. By using the new state truncation option (section 2.8), these applications become considerably more practical. Fault tree input is particularly useful for large fault occurrence models especially if fault/error handling is included. Because HARP converts a fault tree representation into a Markov chain, the user can always alter the generated Markov chain to include behavior not captured by the fault tree. State n-tuple notation is provided as an option to aid the user in identifying the Markov chain states.

Mission time is assumed by HARP to have the units of hours even though most fault/error handling models use a time scale of seconds. The user must therefore express the FEHM time units as specified by HARP. Chapter 3 provides more detailed information on how to input the two FORM types into HARP.

2.6. Single Fault/Error Handling Model (FEHM)

The general form of the single FEHM is shown in figure 10. The detailed fault recovery models capture in a few parameters the sequence of events that occur within the system once a fault occurs. (See figs. 11 to 13.) A fault can be permanent (always present and capable of producing errors, e.g., a broken connection), transient (present for only a short time, e.g., a glitch in the power line), or intermittent (always present but not always active, e.g., a loose connection).

All FEHM's defined later in this section except the CARE III FEHM (section 2.6.7) use time units of seconds to emphasize that these events are fast events. The CARE III FEHM uses time units of hours to be consistent with the program CARE III.

The FEHM is a connected group of fast states that is replaced by a branch point automatically in HARP. Its general structure is a single-entry, (up to) four-exit model, that is entered when a

Figure 10. Fault/error handling model.

fault occurs. The exits represent possible outcomes of the attempted system recovery. Transient restoration R is the event of correct recognition of and recovery from a transient fault before a second interfering fault occurs. Permanent coverage C is the event of successfully reconfiguring the system to eliminate a permanent, intermittent, or transient fault (mistaken as permanent), before a second fault occurs. Single-point failure S represents the event of a single fault causing the system to fail without the interference of a second fault. The near-coincident (ref. 37) fault exit N is taken when a second dependent fault[10] occurs before another exit is reached. The FEHM models are described as three exit models by the user (the R, C, and S exits). The near-coincident failure exit is automatically added by the HARP program.

Many choices are available for the specification of the FEHM model, ranging from constant exit probabilities (VALUES FEHM) to a detailed ESPN (ref. 38) model. Some FEHM's are offered to simplify the modeling and data specification process as much as possible. The simple models such as the values probabilities and moments, probability and distributions, and probabilities and empirical data are provided for users who want to perform sensitivity analyses, perhaps early in the design stage when reliable data are unavailable. The same models can be used later in the design process when hardware exists and data can be measured or estimated as probabilities and recovery time distributions. The more complex FEHM models (ARIES, CARE III, and ESPN) are superset models of the simpler ones and are useful for studying the details of fault handling and predicting the effects of various fault recovery behavior on system reliability/availability.

Parameters of a FEHM are user-specified and thus allow flexibility even while the same FEHM type is used. The tutorial gives examples of the use of these models and their typical values (vol. 2 of this TP). The user can choose a different FEHM type for each different component in the system. Selecting a different FEHM type (or different parameters for the same FEHM type) for a specific component type in the model is possible with the overriding FEHM option. (See section 4.7.) The chosen FEHM model is solved in isolation, and the probabilities for each of the four exits are derived. These probabilities are reflected in the FORM model that is then solved for the reliability/availability of the system. Some options for the fault/error handling model are shown in figures 11 to 13, and all are described in the following sections.

[10] Both faults are stochastically independent, but the second fault (called an interfering fault) was previously specified by the modeler to interact with the first to cause system failure.

2.6.1. No Coverage Model

The NONE option specifies no coverage model. This option is chosen if the user wants to assume perfect fault coverage for a particular component type.

2.6.2. Values Model

For a particular component type, the user may want to allow for imperfect fault coverage but not use a detailed coverage model. In this case, the user can input the coverage values for R, C, and S exits directly. No near-coincident faults are considered for those transitions having values the same as the coverage model type.

2.6.3. Probabilities and Moments Model

The probabilities and moments model allows the user to enter the probability that each of the three exits (R, C, and S) are reached. These three exits represent mutually exclusive events; thus, their probabilities sum to one. For each nonzero exit probability, the user is asked for the first three moments of the time (in seconds) required to reach the particular exit (given that the exit is reached). The probability of reaching the fourth (near-coincident fault) exit is derived from the given moments and the rate of occurrence of dependent near-coincident faults. (See section 2.7.) When this option is used, the FEHM can be visualized to be a single semi-Markovian (fast) state that is reduced to a branch point by HARP aggregation methods.

2.6.4. Probabilities and Distributions Model

Under this option, the user specifies an exit probability for the transient restoration, permanent coverage, and single-point failure exits. For each nonzero exit probability, a distribution of time to exit is specified as one of the following: constant, uniform, exponential, hypoexponential, hyperexponential, gamma, and Weibull. These probabilities and distributions are to be given without regard to the occurrence of a second, near-coincident fault, with a time unit of seconds. The coverage factors (reflecting the effects of near-coincident faults) are then automatically derived from this data. As for the previous option, the FEHM can be visualized as a single semi-Markovian (fast) state that is reduced to a branch point by HARP aggregation methods.

2.6.5. Probabilities and Empirical Data Model

The probabilities and empirical data model is similar to the previous two in that the exit probabilities are given for the three exits of the model. For each nonzero exit probability, the user provides a histogram listing of the time to exit, again without regard to the occurrence of a near-coincident fault. For each x, y pair listed in the histogram file, the x-value refers to the time step in seconds, and the y-value to the probability of reaching the exit during the associated time interval. The probabilities are specified as a probability mass function (not a cumulative distribution) and hence must sum to one. The coverage factors (reflecting the effects of near-coincident faults) are then automatically derived from the data and are used in subsequent calculations. As for the last two options, the FEHM can be visualized as a single semi-Markovian (fast) state that is reduced by HARP aggregation methods to a branch point.

2.6.6. ARIES Transient Fault Recovery Model

The ARIES transient fault recovery model (ref. 39) represents a multiphase recovery process that executes NP successive recovery phases. (See fig. 11.) Transition to the next phase takes

place if the present phase is not effective; the duration of each phase is constant. The recovery process terminates and normal processing begins if successful recovery is achieved in the present phase. If transient recovery is unsuccessful after all NP phases, then a permanent recovery process is initiated.

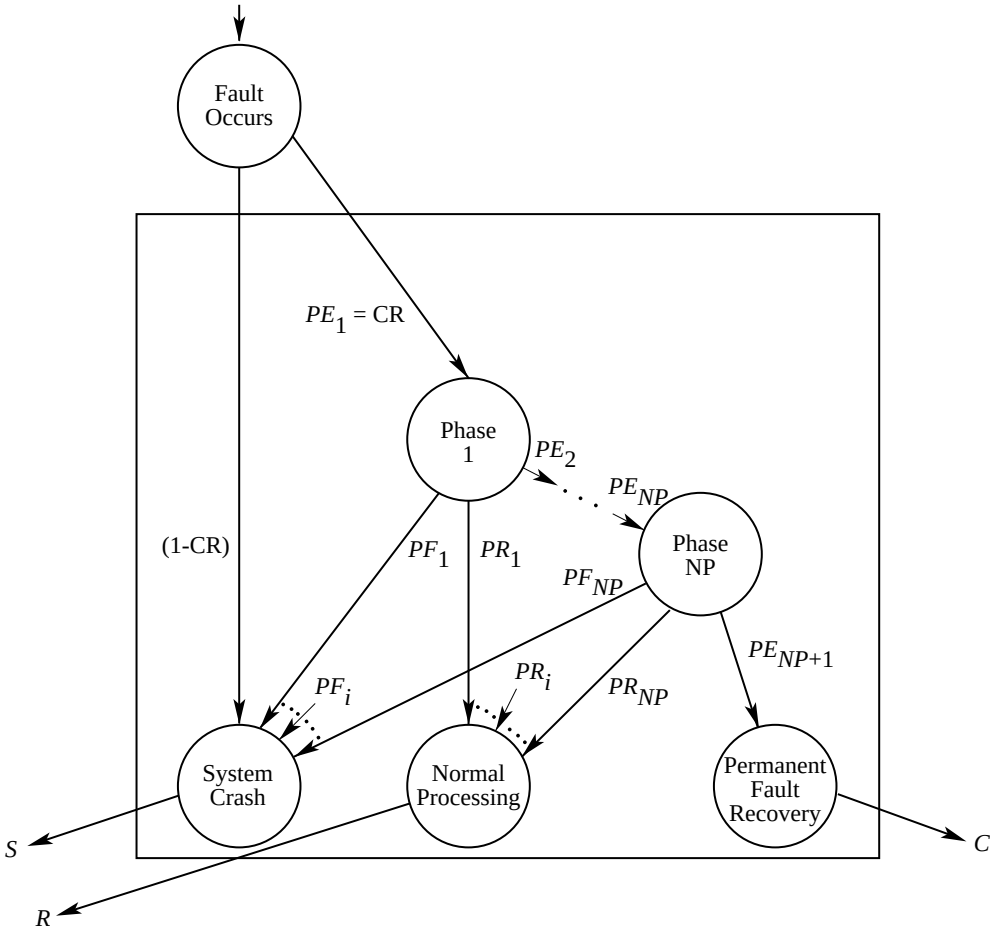


Figure 11. ARIES fault recovery model.

For the ARIES model, the user is asked for the number of phases ($\leq$11) and the constant duration and effectiveness of each phase. The effectiveness of phase i is described by the probability of successful transient recovery PR_i and the probability of the need for the next phase of transient recovery PE_i. The probability of system crash from phase i is then given by PF_i, where $PF_i = PE_i - PR_i - PE_{i+1}$. The time unit for this model is seconds. These parameters are combined together with the transient fault duration to determine whether the transition to the next phase takes place, that is, to determine whether the present phase has been successful. One other parameter is entered—the catastrophic fault recovery probability $1 - CR$. This probability is assigned to a critical fault that causes the entire system to fail because the system was unable to recover from it. The tutorial (vol. 2 of this TP) provides a detailed explanation.

2.6.7. CARE III Coverage Model

Another option for the FEHM is a Markov version of the CARE III single fault model (ref. 40), shown in figure 12. The CARE III coverage model can be used to model permanent, transient,

and intermittent faults. In the active state, a fault is both detectable (at rate δ) and capable of producing an error (at rate ρ). Once an error is produced, if it is not detected, it propagates to the output (at rate ϵ) and causes system failure. If the fault (error) is detected (probability q), the faulty element is removed from service with probability P_A or P_B. With the complementary probabilities, the element is returned to service following the detection of the fault. (This action is based on the belief that the detected fault was transient.) Note that both states A_D (active detected) and B_D (benign detected) are instantaneous states. The model is internally solved analytically for the probability of reaching each exit and for the Laplace transform of the distribution of the time to each exit. Subsequently, the effect of a near-coincident fault is incorporated by means of equations (2), (3a) and (3b) in reference 41.

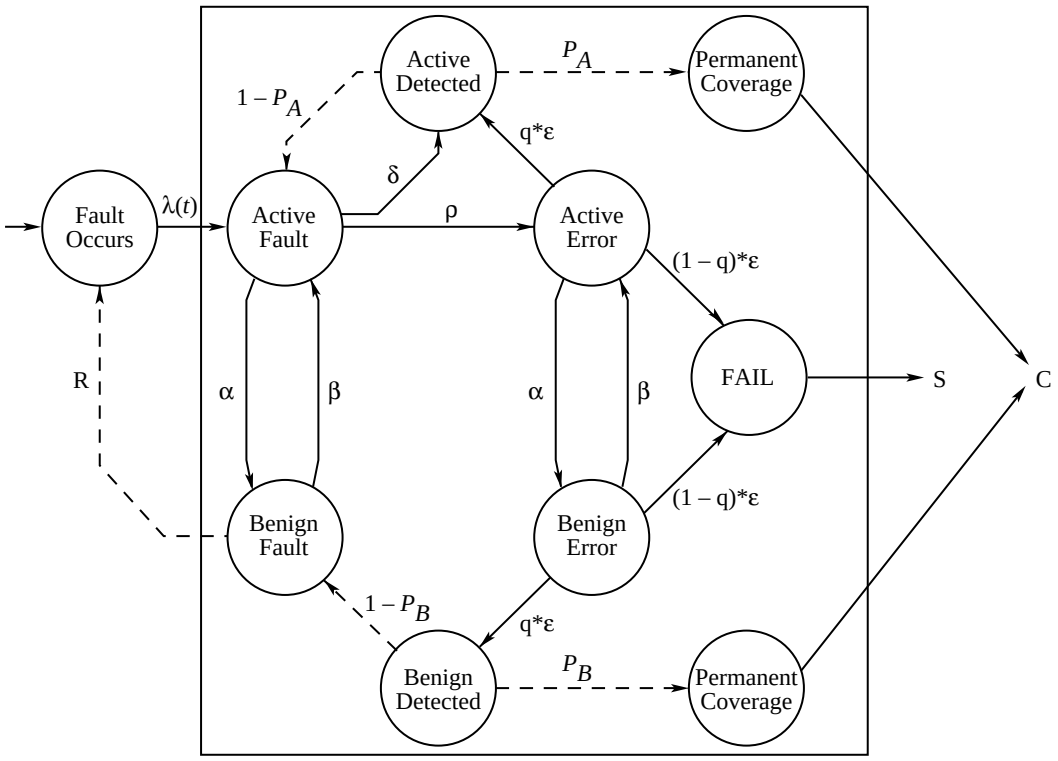


Figure 12. CARE III coverage model.

The user is asked for the probabilities for each of the three fault types (permanent, transient, and intermittent) and is asked to parameterize an instance of the CARE III model for each type that is assigned a nonzero probability. For the permanent model, α, β, and P_B are zero, and for the transient model, α is positive and β is zero. Because q, P_A, and P_B are probabilities, they lie between 0 and 1; because ϵ, δ, and ρ are rates, they are nonnegative. CARE III is the only FEHM model for which the time unit is hours. The three models are solved individually and are combined according to the assigned probabilities for each fault type.

2.6.8. ESPN Model

The HARP ESPN model is discussed in references 38 and 41 to 43 and shown in figure 13. It models three aspects of a fault recovery process: physical fault behavior, transient recovery, and recovery from a permanent fault. The fault behavior model captures the physical status of the fault, such as whether the fault is active or benign (if permanent or intermittent) and whether the fault still exists (if transient). Once the fault is detected, it is temporarily assumed

to be transient, and an appropriate recovery procedure can commence. The transient recovery procedure can be attempted more than once. If the detection-recovery cycle is repeated too many times, a permanent recovery procedure (reconfiguration) is invoked. If the reconfiguration is successful, the system is again operating correctly, although in a somewhat degraded mode.

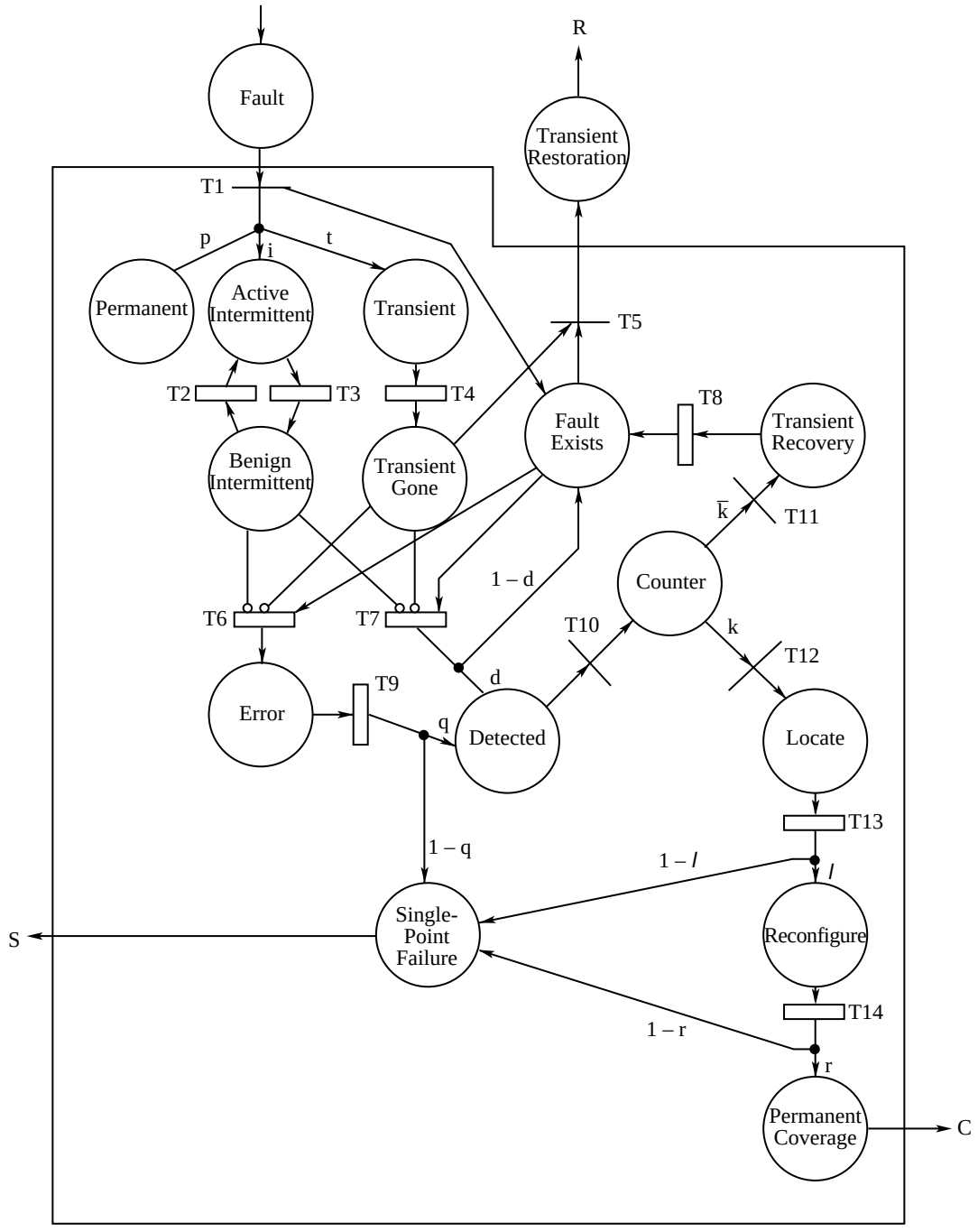


Figure 13. HARP ESPN single fault model.

The user inputs to this model are the distribution of time (T1–T14) for each activity and any associated parameters for the distribution, with a time unit of seconds. (The distributions need not be exponential.) Also requested are the probabilities of correct error detection (q), fault

detection (d), fault isolation (l), and reconfiguration (r). The user must specify the number of attempts at transient recovery, the percentage of faults that are transient, the percentage of faults that are permanent, and since this model is simulated for solution, the desired confidence level and acceptable percent error. The confidence level can be between 60 percent and 98 percent, and the acceptable percent error can be between 1 and 100. (A moderate error requirement in the range of 2 to 5 percent is suggested.)[11] The distributions available for individual transitions in the ESPN model are constant, n-stage Erlang, exponential, log-normal, normal, Rayleigh, uniform, and Weibull. For more information on these distributions, see Trivedi (ref. 44).

The ESPN is the only FEHM simulated for solution. During the simulation, a statistical analysis of the simulation data is performed. The confidence intervals about the exit probabilities are generated and compared with the allowable error. If the confidence interval is too wide, the number of trials is doubled. When the simulation has reached the desired accuracy, the results are appended to the parameter file. If the user does not change the inputs to the ESPN model in this file, then the file can be used over again with the same simulation results, thereby avoiding the simulation run each time. However, if the user has manually changed the inputs with a text editor, the previous simulation results must be discarded; that is, the lower portion of the parameter file must be deleted. Rerun *harpeng*. (See vol. 2 of this TP.)

The simulation of the model uses a random seed value that is derived from the system time. This method helps to assure a random simulation. However, it also implies that the simulation runs are not exactly reproducible. Subsequent simulations cannot match earlier runs exactly, but multiple runs should agree to within the accuracy and confidence requested. For the Convex computing platform, the user must uncomment the line `SEED = 0` and make other changes in the `harpsim` source file and recompile the code.

For this model, the coverage factor for transient restoration, is the probability of a token reaching the place labeled *Transient Recovery*. (See fig. 13.) Coverage is the probability of a token reaching the place labeled *Permanent Recovery* and single-point failure is the probability of a token reaching the place labeled *Single-Point Failure*. The fourth factor, corresponding to the N exit and representing a near-coincident fault, is derived from the relative passage time to the three exits, and is discussed in section 3.2.1. For a more detailed description of the ESPN model, refer to the tutorial (vol. 2 of this TP).

2.7. Multifault FEHM Near-Coincident Fault Rate Specification

HARP provides a number of detailed single-fault models. However, for modeling coexisting synergistic multiple faults, HARP only provides three simple computationally fast automatically generated multifault models for use with behavioral decomposition. The detailed modeling of multiple faults can be computationally expensive and tedious to specify because the modeling requires the user to input data that are typically unavailable.

The advantage of using the simple multifault HARP models is a significant reduction in model and user input data complexity, because the models are computationally fast and automatically generated. The disadvantage is a reduction in accuracy, which experience has demonstrated is typically acceptable (refs. 23 to 26, 41, and 45).[12] The increased deviation resulting from the use of the simple multifault models is always positioned to produce a conservative result as long as the significant system failure modes are properly modeled. The amount of deviation and hence degree of conservatism depends on the system. A measure of the degree of conservatism can often be determined from HARP's optimistic simple bound.

[11] See warning C155 in appendix B.

[12] Different reliability programs or ordinary differential equation solvers were used to compare the reliability of these systems.

The optimistic simple bound is always correct for the instantaneous jump model; however, when the optimistic simple bound is correct with respect to the original model, which occurs in most cases, the optimistic simple bound can be used as an error bound. (See section 3.4 for a discussion of upper and lower bounds.)

The near-coincident fault rate (the rate at which catastrophic faults occur for each fault/error handling model) is determined from the system structural model automatically by program *fiface*. The HARP aggregation technique automatically produces a conservative estimate of system reliability/availability provided that the correct multifault model is used and no increasing transition rates appear with increasing system component failures in the user's model. Such system models can be effectively modeled with HARP's behavioral decomposition but not automatically. In such cases, the user needs to modify the appropriate ASCII file for near-coincidence faults. The file is one of the following: MODELNAME.ALL, MODELNAME.SAM, or MODELNAME.USR.

A second option is for the user to use the AS IS Markov chain model to create new complex multifault models, but the cost of the increased computation and user effort can become prohibitive for all but simple system models. This method avoids the use of instantaneous coverage approximations and produces greater accuracy at greater execution times. Recovery behavior can appear to require a Markovian submodel; however, non-Markovian recovery can be approximated with the method of stages (refs. 44, 46, and 47). Another option is to use XHARP, which automatically generates the model from a user-specified full model.

HARP offers three multifault models: ALL-inclusive, SAME-type, and USER-defined. The application for the ALL-inclusive (ALL) model is the easiest to describe and is comparable with the SAME-type (SAME) in ease of use. When the ALL model is selected, the near-coincident fault rate is based on all pairs of system faults. If the system is composed of a number of different components, each with a unique failure rate, the ALL model will probably overestimate the near-coincident fault probability. The degree of over estimation depends on the system architecture and can be minor or significant. (See section 1.2 and chapter 7.) A rough estimate of the degree of conservatism can be determined by using the SAME or possibly the USER-defined (USER) models. These later models typically give an optimistic result when the system components have different failure rates, but the result is not guaranteed to be optimistic. When the system is composed of components with the same failure rates, the SAME model computes a near-coincident fault rate comprised of all pairs of system components with the same failure rate.[13] Thus, HARP produces a guaranteed conservative result. With the USER model, the user specifies specific pairs of synergistic system faults that constitute a near-coincident fault condition. The conservativeness of the USER model depends on the system architecture.

2.7.1. ALL-Inclusive Near-Coincident Fault Rate

Using the ALL model produces a conservative result if typical failure rate and recovery rate data are used, that is, no increasing transition rates with increasing system component failures. If these rates are used, editing the ASCII file MODELNAME.ALL is necessary to arrive at an accurate result. Because all pairs of failures are considered near-coincident by this model, the degree of conservatism can be large for certain systems. (See chapter 7.) Often, the optimistic simple bound can be used to quantize the conservative deviation. (See section 3.4.) Also, if the user is in doubt as to which multifault model to use, the ALL model provides a quick conservative result that can provide a baseline for comparison of further refined models. HARP automatically generates the required multifault models as follows.

[13] When all system components have the same failure rate, the ALL and SAME models are identical in effect.

Conservatively, we assume that a second near-coincident fault *anywhere in the system* (while attempting to handle the first fault) causes immediate system failure. This multifault model is simple to specify. The procedure for determining the near-coincident fault rate is described as follows.

Let the exit rates of source state i be $\sum k_r\lambda_r$ and of destination state j be $\sum \ell_r\lambda_r$. Then, a FEHM placed on an arc with rate $k_I\lambda_I$ has a near-coincident fault rate (NCFR) given by the following:

$$\text{NCFR} = \left[\sum_{r=I} \max(k_r, \ell_r)\lambda_r\right] + \max(k_I - 1, \ell_I)\lambda_I$$

Figure 14 offers some insight for the interpretation of the equation for NCFR for each multifault model in this section. The NCFR expressions are determined automatically in program *fiface*. If a rate parameter cannot be parsed because it contains unknown variables or added constants, *fiface* uses a look-ahead method to calculate all rates. This computation is provided in lieu of program termination to help the user resolve the problem. For the previous state declarations, the all-inclusive near-coincident fault rate (based on the look-ahead method) is simply the sum of the outgoing arcs of the destination state:

$$\text{NCFR} = \sum \ell_r\lambda_r$$

When only the sum of the outgoing arcs is considered, a warning is issued stating that the results may not be conservative. The ALL-inclusive model is particularly useful for approximating a multifault model where nonfailure transitions emanate from a recovery FEHM. An example of this application can be found in chapter 7.

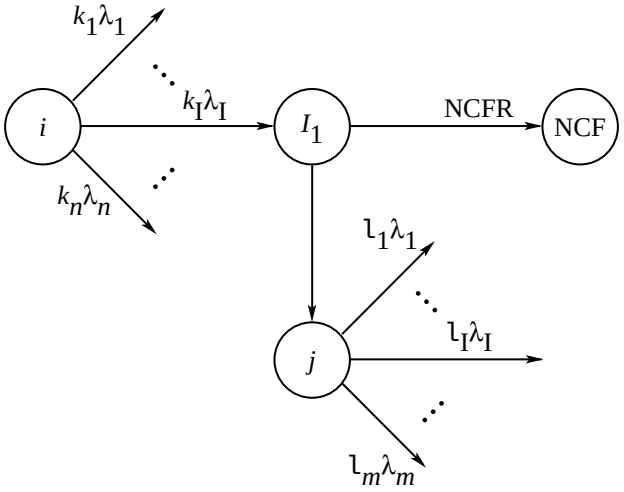


Figure 14. NCFR computation.

2.7.2. SAME-Type Near-Coincident Fault Rate

We can assume that only near-coincident faults of the same component type cause system failure (while attempting to handle a single fault). This multifault model is useful when a system is composed of subsystems where the components in one subsystem have identical failure rates but differ from components in another subsystem. This subsystem also has identical failure rates, and all of its components are synergistically coupled as near-coincident faults within

the subsystem but not across other subsystems. HARP automatically generates the required multifault models as follows.

Again, let the exit rates of source state i be $\sum k_r\lambda_r$ and the exit rates of destination state j be $\sum \ell_r\lambda_r$. For a FEHM placed on an arc with rate $k_I\lambda_I$, we have a near-coincident fault rate given by the following:

$$\text{NCFR} = \max(k_I - 1, \ell_I)\lambda_I$$

The near-coincident fault rate expressions are determined automatically in program *fiface*. If a rate parameter cannot be parsed because it contains unknown variables or added constants, *fiface* uses a look-ahead method to calculate all rates. For these state declarations, the same-type near-coincident fault rate (based on the look-ahead method) is the sum of the same-type rates emanating from the destination state:

$$\text{NCFR} = \sum \ell_I\lambda_I$$

When only the sum of the outgoing arcs is considered, a warning is issued stating that the results may not be conservative. Unlike the ALL model, the SAME model can automatically drop failure modes for certain system models. The user is cautioned to insure that no important failure modes are dropped; otherwise, a nonconservative result can be given. (See chapter 7.)

2.7.3. USER-Defined Near-Coincident Fault Rate

For some models, the user may want to define explicitly, for each component, which other components can interfere with fault recovery. In this case, the user-defined near-coincident fault rate for the FEHM between operational states depends on the user input. For example, suppose we have a system consisting of three processors, $P1$, $P2$, and $P3$ (all distinct with unique failure rates), a voter V, and a bus B. Suppose further that the processors are connected (from the monitoring point of view) in a ring network so that processor $P1$ detects errors and performs recovery for processor $P2$, processor $P2$ likewise monitors $P3$, and $P3$ monitors $P1$. Thus, a failure in processor $P1$ can interfere with recovery in processor $P2$. Similarly, a failure on processor $P2$ can interfere with recovery in $P3$. Because the processors are connected by the data bus, a bus failure can interfere with recovery on any of the processors; the bus does not rely on any other component for recovery. The voter is self-checking; no faults interfere with recovery from voter faults. This behavior cannot be captured by the all-inclusive or the same-type fault rates. It is captured by declaring that recovery in $P1$ depends on $P3$ and the bus, recovery in $P2$ depends on $P1$ and the bus, and recovery in $P3$ depends on $P2$ and the bus. HARP automatically generates the required multifault model as follows.

Let the exit rates of source state i be $\sum k_r\lambda_r$ and of destination state j be $\sum \ell_r\lambda_r$. Also let D_I be the set of interfering component types for component type I. Then, a FEHM placed on an arc with rate $k_I\lambda_I$ has a near-coincident fault rate given by the following:

$$\text{NCFR} = \left[\sum_{r=I, r\in D_I} \max(k_r, \ell_r)\lambda_r\right] + \max(k_I - 1, \ell_I)\lambda_I I* \qquad (I \in D_I)$$

where $I*$ is the indicator function that takes on value 1 if the subscript expression is true and 0 otherwise.

The near-coincident fault rate expressions are determined automatically in program *fiface*. If a rate parameter cannot be parsed because it contains unknown variables or added constants, *fiface*

uses a look-ahead method to calculate all rates. For these state declarations, the user-defined near-coincident fault rate (based on the look-ahead method) is given by the following:

$$\text{NCFR} = \sum_{r \in D_I} \ell_r \lambda_r$$

When only the sum of the outgoing arcs is considered in determining near-coincident fault rates, a warning is issued stating that the results may not be conservative. Like the SAME multifault model, the incorrect use of the USER model could produce a nonconservative result when important failure modes are dropped. (See section 3.3.2.)

2.7.4. Exact Specification of Near-Coincident Fault Rates

The ALL, SAME, and USER multifault models are provided to automatically generate the near-coincident fault rates. This automatic capability is extremely useful for all but trivial models; however, this convenience has a trade-off. The automatic multifault model is not capable of generating exact near-coincident for all possible Markov chains. In some cases, an approximating model such as the ALL model must be chosen to insure a conservative result. The user has an alternative approach if exact rates are desired and cannot be achieved with the automatic model. The user can derive these rates manually and enter them for HARP computation by editing the MODELNAME.ALL, MODELNAME.SAM, or MODELNAME.USR files. (See section 4.2.) These ASCII files are readable and easily modified.

Section 4.2.7 shows the format for the near-coincident fault rates. The expressions for the rates depend on the Markov chain of interest. Section 3.2.1 gives some insight into how the near-coincident fault rates are related to the coverage, C_i, parameters.

2.7.5. Multiple-Run Near-Coincident or No Near-Coincident Faults

In *fiface* or *harpeng*, the user can ignore any near-coincident faults. By specifying no near-coincident faults in *fiface*, the system model is much smaller. This selection may be necessary for extremely large models. (PC HARP 16-bit version does not allow the specification of near-coincident fault rates because of DOS's 640K memory limitation. Other versions do not have this restriction.) Otherwise, if the user wants to exercise several different near-coincident fault type options, none can be specified during the execution of *harpeng*.

Thus, whether the user chooses the Markov chain or fault tree option for specifying the system structure, the near-coincident fault rates for each instance of a fault/error handling model are generated automatically. During execution of *fiface*, the user is asked whether the all-inclusive, same-type, user-defined or no near-coincident fault rate should be used and what combinations of these, if any, are to be used in successive *harpeng* runs, for example, ALL, SAME, USER. In this way, all options can be exercised during different *harpeng* execution runs. A discussion of the various near-coincident fault rate options can be found in volume 2 of this technical paper.

As previously mentioned, an alternative is to model the systems with the AS IS Markov solution technique. This choice produces greater accuracy at greater execution times. Recovery behavior can appear to require a Markovian submodel; however, non-Markovian recovery can be approximated with the method of stages (refs. 44, 46, and 47).

2.8. Truncation of Model Entered as Fault Tree

Frequently, even for a simple model, a large number of states and transitions are produced upon conversion from a fault tree to a Markov model. This largeness problem is encountered

despite the use of behavioral decomposition. To solve a large model, HARP allows the user to truncate the model after a total of K component faults (basic event failures) have occurred and HARP then computes bounds for the model results (ref. 32). These bounds encompass the untruncated unreliability/unavailability of the system. Because most systems are designed to tolerate at most three faults and because the probability of having more than three faults occurring during certain missions is extremely small, a truncation level of three often produces values for the bounds that agree up to the third and fourth decimal places. The higher the truncation level, the tighter the bounds and the greater the computational time. Typically, the truncation level is selected to be equal to or one greater than the number of faults the system was designed to tolerate. A truncation level can be specified when the fault tree FORM is used. A truncation setting of $K = 3$ often allows solution of a system with up to 2^{71} equivalent states (a fault tree with 71 basic events). The execution time for such a truncated model can be surprisingly low while providing acceptable accuracy (ref. 48).

We integrate the use of fault trees and state truncation in the generation of a Markov model in the following way. Because a fault tree is often a more compact representation of the system than a Markov model, the user is given the option of entering the system as a fault tree. The fault tree is then converted to a Markov model automatically. In the conversion process, all states of the Markov model reachable by each number of component faults are generated together (all states reachable by one fault, then all states reachable by two faults, etc.) (ref. 49). The user can specify a number of total component faults beyond which the Markov model is to be truncated. The state generation process then proceeds in a normal fashion only as far as the number of faults specified by the user. Once this number of faults is reached (the truncation line) the state generation process changes. All transitions from the states at the user-specified truncation level that would lead to "up" states in the full-Markov model are directed to special dummy states in the truncated model. Each dummy state represents an aggregation of all states in the full model that would be reachable along the transition leading to that dummy state. These dummy states are consequently called truncation aggregation (TA) states. Transitions that would lead to failure states in the full model lead to failure states in the truncated model as well.

In general, the states represented by a TA state can include both system operational states and system failure states. An optimistic (upper) bound on the system reliability can be obtained by assuming that all states represented by the TA states are up states. Similarly, a pessimistic (lower) bound on the system unreliability can be obtained by assuming that each TA state is forced to be a system failure state ($\mathrm{TA_{tr}}$).

As the truncation level increases, more states appear explicitly in the Markov model instead of being represented in the TA states. Thus, as the truncation level increases, states that were forced to be failure states are now treated as operational states, which they really are in the full model. Hence, a model with a greater truncation level would more accurately represent the full model than would a model with a smaller truncation level. We would therefore expect the truncation bounds of the model to become tighter as the truncation level increases. In the limit, as the truncation level increases, the truncated model becomes identical to the full model and the bounds from the truncated model converge to the exact reliability value obtained from the full model.

2.9. FORM Model Parameters

Once the FORM and FEHM submodels and the near-coincident fault rates (if any) have been specified and the model is ready to be solved, the user needs to specify the parameters used in the model.

Because the user may not know the exact values of the input parameters, HARP accepts as input a nominal value and a variation on all input parameters. The nominal value is used for the

unreliability prediction, and the variation about the nominal value is used in an approximate model to generate bounds about the predicted unreliability. The various FORM parameter types that can be specified are described in the following sections: Non-Markovian models that involve Weibull distributions and warm or cold spares are best solved with MCI-HARP (refs. 18 and 19).

2.9.1. Failure Rate Specification

When the FORM model is being built, a symbolic failure rate variable (i.e., lambda or mu) is used for each component type. At run time, this symbol is defined as either a constant failure rate or a Weibull failure rate. Constant failure rate refers to the rate associated with the exponential time to failure distribution. Weibull failure rate refers to the rate associated with the Weibull failure distribution. The Weibull failure rate has two common forms; both are available in HARP. These two forms are defined as follows:

$$h(t) = \lambda \alpha t^{\alpha - 1}$$

$$h(t) = \lambda^{\alpha} \alpha t^{\alpha - 1} = \lambda \alpha \lambda t^{\alpha - 1}$$

The first form is as defined in reference 44 and the second is as used in the CARE III package (ref. 40). When a Weibull failure rate is used to model the failure of a component, the resulting solution is very slow because the time-dependent transition rate matrix must be reevaluated at each time step. (MCI-HARP can accelerate the solution significantly for large systems.) Also, the bounds computations (using the simple model and parameter variations) are not available when a Weibull failure rate is used. However, the user can still truncate the model (see section 1.4) and generate truncation bounds that enclose the unreliability. If the user tries to model systems with cold or warm spares (i.e., those whose failure rates change when switched into operation) or mixed Weibull and constant failure rates, HARP issues a warning that the user is violating the inherent Markovian modeling conditions (ref. 7).[14] Execution of the code proceeds to completion; however, the warning is listed in the results file as well as on the screen. (See chapter 7 for details and appendix D for warning messages.)

2.9.2. Repair Rate Specification

When the system being modeled is repairable, the user needs to specify the value of the repair rate at runtime. The evaluation of parametric bounds is not allowed for repairable systems unless an absorbing state exists. If any failure rate specified for a repairable system is Weibull and a constant failure rate is also specified, meaningless results can be produced. (See chapter 7.) A warning is issued in this case.

The user is reminded that a nonhomogeneous Markov chain has one time variable that is the global clock and is initiated at mission time zero to be zero. Repairable units with Weibull are subject to the same clock; thus, Weibull failure rates are not reset to time zero when repair is completed. If the user wants to reset the clock to time zero or some other time, then a more powerful model solver is required as the resulting model is non-Markovian. MCI-HARP is designed to cover such models.

2.9.3. User-Specified Coverage Parameters

If no FEHM's are used in the system being modeled, the user can specify coverage parameters during runtime. These parameters can be the coverage factor C, the transient restoration factor R, and the single-point failure factor S.

[14] A cold Weibull spare is precluded from failing even though the component failure history is reset to mission time zero. This feature can be useful as an optimistic estimate.

Chapter 3

Model Solution

3.1. Conversion of Fault Tree to Markov Chain

If the FORM is a fault tree, it is internally converted to a Markov chain for solution after it has been input to HARP. Figure 15 is an example of a fault tree for a system with three processors and two memory units. When using textual HARP, the user normally labels a fault tree with unique node labels. The node labels are assigned to basic events, gates, and the FBOX symbol, which represents the system failure events. The order of specifying the node values is not important, but uniqueness is. The GO program automatically assigns the node values.

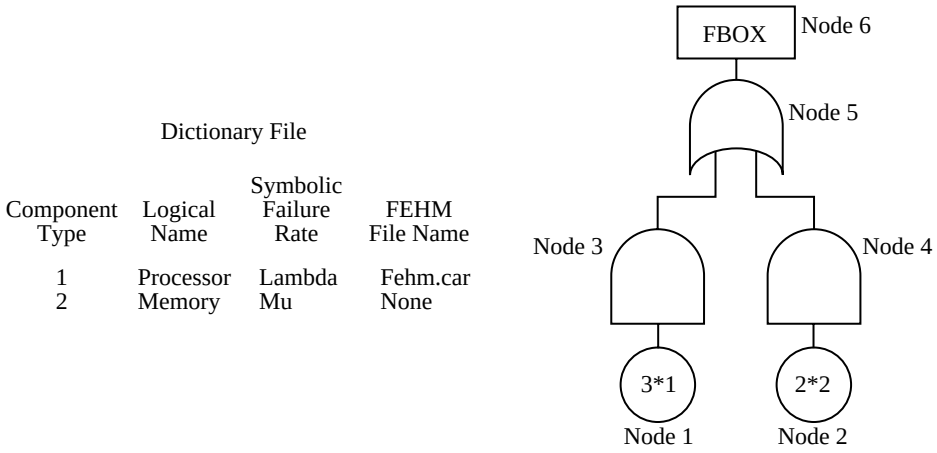


Figure 15. Three-processor two-memory system dictionary file and fault tree.

While inputting the model, the user is requested to create a dictionary file that is associated with the fault tree model. The contents of the dictionary file are shown in figure 15 and are associated with the fault tree in the same figure. In this example, the user entered the information in the last two rows of the file, with the exception of indices 1 and 2. The program assigns these indices to the component logical names (symbols $ and & are not allowed)[15] in the user inputs. The indices are used to simplify the notation of identifying the component types in the fault tree basic events. In figure 15, these are shown in the fault tree as the numbers to the right of the * symbols. For node 1, the 3*1 means that there are three identical type one components (processors) with the same failure rate symbol Lambda and FEHM described by the file, Fehm.car. When basic events are identical replications of a component type, the 3*1 notaton signals the program to simplify the Markov model by specifying one failure rate symbol instead of three separate ones. In figure 16, the effect of the 3*1 notation is to assign the failure rate $3*\lambda$ to the transition from (3,2) to (2,2). The user must differentiate component type indices from node indices. Any component type index can be assigned to any node, uniqueness is not required; however, uniqueness is required for the node indices.

The fault tree in figure 15 is converted by HARP into the Markov chain shown in figure 16. All combinations of basic events that leave the system operational are enumerated; each combination becomes a state in the Markov chain. Note that the basic event * notation has reduced the

[15] Avoid special characters because they often interfere with the operating system.

Figure 16. Three-processor two-memory system Markov chain.

number of state combinations. The user does not have to delineate all possible combinations, only those that are required. In this model, 32 combinations are possible, but only 8 are required. Exhaustion of redundancy failure states are also generated.

Also note that only a nonrepairable system can be specified by means of a fault tree. (Fault tree models with repair have not yet been developed.) To model systems with repair for availability prediction, the user must either input the model directly as a Markov chain or specify a fault tree model and then subsequently modify the Markov chain MODELNAME.INT file to include the repair transitions with a text editor. For details of the algorithm used for conversion, see chapter 6 and reference 49.

3.2. Modeling Imperfect Coverage

The possibility of imperfect fault coverage is automatically incorporated into the FORM model Markov chain as follows. Through the dictionary, each component type in the system can have associated with it a fault/error handling model that describes the recovery behavior of that particular component. The three-processor two-memory system shown as a Markov chain in figure 16 is used to demonstrate the idea of imperfect coverage. Components of type 1 represent the processors, one of which must be operational for the system to remain up. Likewise, one of component type 2 (the memories) is necessary for operation. Processors fail with rate λ and memories with rate μ. For our example, the states are labeled with a pair of numbers—the first signifying the number of operational processors and the second satisfying the number of operational memory components. Once the number of processors is exhausted, state F1 is entered. Once the number of memory components is exhausted, state F2 is entered. If the user has specified coverage, the HARP program automatically places a FEHM on the appropriate arcs, as shown in figure 17. HARP prompts the user for dictionary information to define the FEHM model to be used for a processor failure—this model is used for FEHM numbers 1, 2, 6, and 7. The FEHM model for memory failures is used in FEHM numbers 3, 4, and 5. Thus, the contents of box 1, 2, 6, and 7 are identical but may differ from the contents of box 3, 4, and 5, which are also identical to each other). (However, the user may override a FEHM on a specific arc; see section 4.6.2 and chapter 7.) The difference in the FEHM's (i.e., why they are numbered 1 to 7 rather than just 1 and 2) is in the near-coincident fault rate used to calculate the N's.

3.2.1. Automatic Incorporation of Coverage Models

The FEHM models are solved in isolation for the exit probabilities for the three exits (R, C, and S) and for some measure of time to exit. The probabilities are then adjusted according to the probability of an interfering (near-coincident) fault to produce (state-dependent) coverage probabilities, which are then used to modify the transition rates in the Markov chain. Additionally, two failure states are added to the model, one to represent single-point failures and one to represent near-coincident faults. (See fig. 17.)

Figure 17. Automatic insertion of FEHM's.

More specifically, assume that a fault of component type 1 in state (i, j) leads to state $(i-1, j)$ in the perfect coverage Markov chain. In the imperfect coverage Markov chain, this transition to state $(i-1, j)$ is completed with probability $C_{(i,j),(i-1,j)}$, and a transition to the single-point failure state occurs with probability $S_{(i,j),(i-1,j)}$. A transition back to state (i, j) occurs with probability $r_{(i,j),(i-1,j)}$, and a transition to the near-coincident failure state occurs with the following probability:

$$N_{(i,j),(i-1,j)} = 1 - C_{(i,j),(i-1,j)} - R_{(i,j),(i-1,j)} - S_{(i,j),(i-1,j)}$$

This probability of imperfect coverage is then incorporated into the Markov chain by first reducing the rate of flow from state (i, j) to state $(i-1, j)$ by multiplying the original rate γ from state (i, j) into the FEHM of component type 1 by $C_{(i,j),(i-1,j)}$ and by then adding arcs from state (i, j) to the failure states. These additional arcs represent a flow of $\gamma S_{(i,j),(i-1,j)}$ to the single-point failure state and a flow of $\gamma N_{(i,j),(i-1,j)}$ to the near-coincident fault failure state. These computations are performed for all arcs between operational states. For example, when $(i, j) = (3, 2)$ then γ is 3γ. (See fig. 17.)

These coverage failure states can be differentiated from the exhaustion of components failure state, if the user desires a comparison of the respective failure probabilities. Figure 18 shows the imperfect coverage representation of the three-processor, two-memory system of figure 15, where FSPF represents the single-point failure state, and FNCF represents the failure of the system caused by a near-coincident fault. In this figure, the coverage factors have single subscripts for ease of notation. The Markov chain of figure 18 is an approximation (see section 2.1) to the stochastic process represented by figure 17.

Figure 18. Imperfect coverage representation of three-processor, two-memory system.

The state diagram of figure 17 is automatically reduced by HARP to that of figure 18; HARP solves this Markov chain for state probabilities. The FEHM information is captured in the C, R, S, and N parameters and is passed to the engine for solution. Note that the stochastic process represented by figure 18 (fig. 17) is generally rather large and generally non-Markovian. For instance if all FEHM types chosen are semi-Markovian, then the stochastic process of figure 17 is either semi-Markov (if all failure rates are constant) or nonsemi-Markov (if one or more of the failure rates is chosen to be Weibull). Similarly, if a simulated FEHM is chosen, the stochastic process of figure 17 will be more general than a semi-Markov process. Even when a Markovian FEHM is chosen, the process represented by figure 18 (fig. 17) is generally a very stiff Markov model. The instantaneous coverage approximation in HARP (fig. 18) avoids the generation and the solution of the large and stiff stochastic process. This approximation results in considerable savings in storage and in time. At run time, the user is simply queried as to the numerical values for the failure (and repair) rates and which near-coincident fault rate calculations are to be used in the solution of the FEHM's (explained in section 2.7).

The derivation of the C, R, S, and N parameters can be illustrated by way of a simple example system architecture that is a variation of the three-processor two-memory system, that is, consideration of only the three-processor part of the three-processor two-memory system. Again, we automatically incorporate the possibility of imperfect coverage into the perfect coverage Markov chain, as shown in our three-processor example (fig. 19). While in the coverage model denoted by FEHM 1, a second processor fault is possible with rate $2 * \lambda$. Therefore, one of the exits, R, C, or S must be reached before time to the second fault (which is an exponentially distributed random variable with parameter $2 * \lambda$) if a near-coincident fault is to be avoided.

Figure 19. Three-processor system. FEHM's with C, S, R, and N exit probabilities.

Figure 20. Three-processor system showing near-coincident faults.

Figure 21. Reduced model of the three-processor system with near-coincident faults.

Likewise, while in the coverage model denoted by FEHM 2, another processor failure can occur with rate λ.

Assume that FEHM 1 and FEHM 2 in figure 19 are exponentially distributed delays with rate δ. (See fig. 20.) Thus, $S = R = 0$. Note that in the absence of a near-coincident fault, $C = 1$. However, with the near-coincident fault occurring at the rate $2 * \lambda$ from FEHM 1, the probability of a successful C exit before the occurrence of a second near-coincident fault is easily shown to be $C_3 = \frac{\delta}{\delta+2*\lambda}$. Similarly for FEHM 2, $C_2 = \frac{\delta}{\delta+\lambda}$. The reduced model is shown in figure 21. In figure 21, $N_3 = 1 - C_3$ and $N_2 = 1 - C_2$. The inclusion of interfering faults causes the coverage values to become state dependent. HARP automatically derives the

coverage factors by taking the Laplace transform of the time-to-exit distributions. We compute the transforms for the single fault model and then substitute the second near-coincident fault rate for the Laplace transform variable to obtain the state-dependent coverage values. If the time-to-exit distribution is not available in closed form, a Taylor series expansion of the Laplace transform yields an expression that depends on the powers of the next fault rate and on the moments of the distribution. These moments are easily obtained from empirical or simulation data. See reference 6 for the mathematical derivations.

We need not restrict ourselves to single-state FEHM's. Let us again look at a portion of the CARE III coverage model that was introduced in section 2.6.7. (See fig. 22.)

Figure 22. Permanent CARE III FEHM with N, C, and S exits.

Now the FEHM probabilities when replaced by a branch point are as follows:

$$C_3 = \frac{\delta}{\delta+\rho+2*\lambda} + \frac{\rho}{\delta+\rho+2*\lambda} * \frac{q*\epsilon}{\epsilon+2*\lambda} \qquad C_2 = \frac{\delta}{\delta+\rho+\lambda} + \frac{\rho}{\delta+\rho+\lambda} * \frac{q*\epsilon}{\epsilon+\lambda}$$

and

$$S_3 = \frac{\rho}{\delta+\rho+2*\lambda} * \frac{(1-q)*\epsilon}{\epsilon+2*\lambda} \qquad S_2 = \frac{\rho}{\delta+\rho+\lambda} * \frac{(1-q)*\epsilon}{\epsilon+\lambda}$$

As previously mentioned, these probabilities are determined by HARP based on the user inputs for the rates and probabilities in the model.

3.2.2. State-Dependent FEHM—Overriding the Default Model

Suppose we want to override the FEHM file associated with a particular component type. In the three-processor, two-memory example, the processors have a FEHM parameter file entitled `FEHM.HRP`. Assume that from state 2, 2 recovery from a processor failure is more closely modeled by a different FEHM model, which is stored in `FEHM.NEW`. In this case, we change the description of the Markov state transition from `2*LAMBDA;` to `2*LAMBDA:FEHM.NEW;`. This change does not affect any other transitions triggered by a processor failure. If we instead want to turn off the FEHM for this transition, we can use the keyword NONE. For this example, the state transition is `2*LAMBDA:NONE;`.

If for a particular component type the user has chosen to provide the actual coverage values (rather than use a FEHM model), has chosen to ignore near-coincident faults, or has chosen to ignore coverage, then the default model (or rather, the lack of a model) cannot be overridden. These original choices results in state-independent coverage values, which cannot be later made state dependent. Likewise, a FEHM cannot be overridden by typing `VALUES` because of its state independence.

3.3. Numerical Solution Techniques

Once the FEHM models have been solved and the state-dependent coverage factors have been automatically inserted into the model, the Markov chain representation of the FORM model remains to be solved. The Markov chain (such as the one in fig. 18) produces a linear system of ordinary differential equations as follows:

$$P'(t) = A(t)P(t) \qquad (P(0) = P_I)$$

where $P(t)$ is the column vector of the state probabilities, and $A(t)$ is the associated matrix of (possibly) time-dependent transition rates (the matrix entry $a_{ij}(t)$ represents the transition rate from state j to state i at time t). This analytic model is solved for the state probabilities $P_i(t)$ in one of two ways, which are described in the following sections. The reliability and unreliability of the system are then given by the sums of the appropriate state probabilities:

$$R(t) = \sum_{i \epsilon \text{ UP states}} P_i(t)$$

$$U(t) = \sum_{i \epsilon \text{ DOWN states}} P_i(t)$$

Additionally, the probability of each type of failure (exhaustion of redundancy, single-point failure, near-coincident faults) is reported separately. In case repair transitions are emanating from down states, HARP uses the previous equation to compute the instantaneous availability and unavailability.

3.3.1. Default Solution Technique

Under normal circumstances, the Markov chain is solved for the state probabilities with a variation of the adaptive Runge-Kutta procedure, GERK (ref. 50). GERK has been reliable and robust for a large variety of models. Although GERK solves stiff systems correctly, it can take a long time; thus, an alternative solution method has been defined for these systems.

Note that the maximum global error reported in the .RS* file is the global numerical discretization error of the differential equation solver (GERK output). It does not include the round-off error or numerical deviation resulting from behavioral decomposition. Modeling deviations contributed by behavioral decomposition are conservative. Model deviations contributed by the FORM/FEHM merging process depend upon the user's specification of the multifault model. In most cases, these deviations can be tracked with the simple bounds. (See section 3.4.1.) When the sojourn times of the FORM and the FEHM's are separated by over 2 orders of magnitude, the HARP results are much more accurate than the input data supporting the HARP results. When the sojourn times approach each other, the HARP results become increasingly more conservative. A warning message is issued to alert the user of this possibility. When behavioral decomposition is not invoked, GERK specifies the errors in the .RS* file (round-off error not included).

3.3.2. Stiff System Solution

If the largest of the transition rates in the Markov chain is bounded by q and the mission time is t, then the stiffness index of the Markov chain is defined to be their product qt. When qt is large (say greater than 100), a special stiff solver is invoked.

As mentioned in the previous section, HARP uses a variation of an adaptive Runge-Kutta procedure to solve the system of ordinary differential equations derived from the Markov chain representation of the FORM. It performs well for the transient solution of nonstiff Markov models. However, in stiff systems, the step size h may need to be intolerably small (because the time scale is chosen as a function of the slow failure rates) before acceptable accuracy is obtained. Thus, substantial computation time requirements result.

HARP uses a special method called TR-BDF2 to deal with the problem of stiffness[16] (ref. 51). This method combines the trapezoidal method (TR) with the second-order backward's difference (BDF2). Because TR-BDF2 is an implicit method[17] (whereas GERK is an explicit method), its step size can be adaptively changed. Thus, for stiff models, TR-BDF2 takes much less solution time than GERK. The TR-BDF2 method has provided good accuracy and excellent stability on stiff problems. The choice of which solver to use is made internally by HARP. MCI-HARP uses a simulation technique based on the variance reduction technique, called importance sampling, which is effective for solving stiff systems.

3.3.3. Computational Precision

The coverage value precision and hence the system unreliability computation actually depends primarily upon the FEHM being used in the model. The coverage value precision is 10^{-13} for all FEHM's except ESPN FEHM, which is specified by the user. HARP's unreliability predictions are valid from unity to 10^{-15} and are a function of the precision of the coverage computation as determined by the EPX variable. If a smaller value is computed, then a warning message is issued. If the user requires smaller predictions than 10^{-15}, the EPX value can be adjusted as required; however, the user must determine whether the computing platform is capable of producing that precision. When no FEHM's are used, the unreliability precision is determined by GERK and the round-off error.

Each coverage model in the HARP engine utilizes an epsilon variable entitled EPX. If a coverage factor falls within EPX and 0, the value is set to 0. Likewise, if the coverage factor falls within the 1.0 – EPX and 1.0, the value is set to 1. In all other cases, the actual computed coverage value is retained. Whenever the computed coverage value is changed to 0 or 1, the overall system reliability can be suspect when the unreliability is below 10^{-15}. This problem can be alleviated by changing the value of the EPX variable in the following files: aries.for, care.for, dists.for, empir.for, moments.for, and simdrv.for. The value is set twice in each file—once for the nominal computation and once for the bounds computations.

3.4. Error Bounds

Two different kinds of bounds are provided by the HARP program; simple model (parametric) bounds and truncation model bounds. Depending on the system being modeled, none, one, or both kinds of bounds are applicable.

The simple parametric bounds are computed for two distinct classes of models: (1) the AS IS model that does not use any FEHM's, where behavioral decomposition is not invoked, and

[16] Stiffness refers to a mathematical model that contains widely separated time constants associated with a system of ordinary differential equations.

[17] Implicit means dependent variable is not isolated from other terms of the equation, and explicit means it is.

(2) those models that do invoke FEHM's and behavioral decomposition. Both models can also be modified to reflect the model state reduction technique called truncation (section 3.4.2).

The AS IS model is used strictly for parametric analysis, which reports the effect on system unreliability as a function of the user-specified parametric variation. These data are useful for sensitivity analyses. The simple parametric bounds for this model class are true bounds for the original user-specified model ($M1$).

When FEHM's and behavioral decomposition are invoked, the simple bounds take on two manifestations. When no parametric variation is specified and the user selects the simple bounds computation (prompted by HARP), simple upper and lower bounds are computed based on estimated maximum and minimum imperfect coverage and lack of sufficient redundancy. If parametric variation is also specified, a combined effect is estimated, that is, imperfect coverage with insufficient redundancy and parametric variation. Unlike the AS IS model, the simple lower bound on unreliability associated with behavioral decomposition is a conditionally true bound. The conditions when it becomes questionable are delineated further in this section.

HARP does not allow bounds to be evaluated when any failure rate is Weibull. When the system being modeled has repair, bounds are evaluated only when an absorbing state is present in the model.

3.4.1. Simple Model (Parametric) Bounds

3.4.1.1. AS IS Model

Because many input parameters to the FORM model are not known exactly (e.g., the user can only know a range of values for the failure rates), HARP allows the FORM input parameters to be expressed in terms of ranges of values rather than point estimates. HARP produces upper and lower bounds on the system unreliability that are a function of these ranges of values. The model evaluates the overall system failure probability by taking the lower bound on the failure rates and the upper bound on the repair rates as the best case and by taking the upper bound on the failure rates and the lower bound on the repair rates as the worst case. It also produces the predicted unreliability based on the nominal values. The simple parametric bounds for this model class are true bounds[18] for the original user-specified model ($M1$).

3.4.1.2. Models Using Behavioral Decomposition

We approach the analysis of errors by decomposing the original model into two simpler models that can be combined to obtain a conservative unreliability estimate (refs. 9, 52, and 53). The general form of the simple bounds is given as follows:

$$P(A \cup B) \leq \min[1, P(A_{\text{high}}) + P(B_{\max})]$$

$$P(A \cup B) \geq \max[P(A_{\text{low}}), P(B_{\min})]$$

The first rule gives the conservative bound and the second rule gives the optimistic bound.[19]

The first expression gives the upper unreliability bound while the second gives the lower unreliability bound. The system failure probability $P(A)$ is caused by the lack of sufficient

[18] Mislabeling a failure transition as a repair transition or vice versa can cause inverted bounds. HARP does not check for these types of user mistakes.

[19] Validity of these bounds are subject to correct specification of multifault models, where applicable (see section 2.7).

redundancy. The probabilities $P(A_{\text{high}})$ and $P(A_{\text{low}})$ are used instead of $P(A)$ when parametric tolerance are selected to cause $P(A)$ to be maximum to get $P(A_{\text{high}})$ and $P(A)$ to be minimum to get $P(A_{\text{low}})$. The probability of system failure due to imperfect coverage is $P(B)$. When FEHM's are specified for behavioral decomposition, $P(B)$ is computed for the minimum imperfect coverage to get $P(B_{\text{min}})$ and the maximum imperfect coverage to get $P(B_{\text{max}})$. The probability $P(A)$ is further modified when transients are specified in at least one FEHM. The perfect redundancy model (coverage assumed to be perfect) transition rates are modified by coefficients that reflect transient restoration probabilities. The net effect reduces the probability of failure by redundancy exhaustion because transient restoration occurs.

The simple bounds computed by HARP are the bounds on the instantaneous jump model ($M2$, $M3$, and $M4$ of fig. 3), which produces the unreliability result ($M4$) and can also bound the user's original model $M1$ under certain conditions: The simple upper bound on the system unreliability is always a true (conservative)[20] bound with respect to both the instantaneous jump model ($M2$, $M3$, and $M4$) and the user-specified model ($M1$) (provided all failure rates are constant).[21]

When the automatically generated multifault models are used, the validity of the optimistic bound with respect to the user-specified model ($M1$) depends on the use of large numbers of fault containment regions that require the use of the ALL multifault model. (See chapter 7 for an example of a practical fault containment model.) Figure 23(a) depicts the normally expected HARP computations. With many fault containment regions, the situation shown in figure 23(b) is possible. The lower bound graph is above the full model graph but below the instantaneous jump model graph. For such system models, a valid accurate lower bound can be obtained by modifying the HARP generated ASCII files.

Remember that the HARP simple bounds are used for preliminary estimates of unreliability. They are provided as a quick-look computation that can be used in the early stages of system design when only ranges of parameter values are available. The essence of HARP output is the nominal result and not the simple bounds. We emphasize that if the model is solved AS IS, without any FEHM's or with the VALUES FEHM, the HARP bounds are true bounds for the user-specified model ($M1$), that is, the full model.

3.4.2. Truncation Bounds

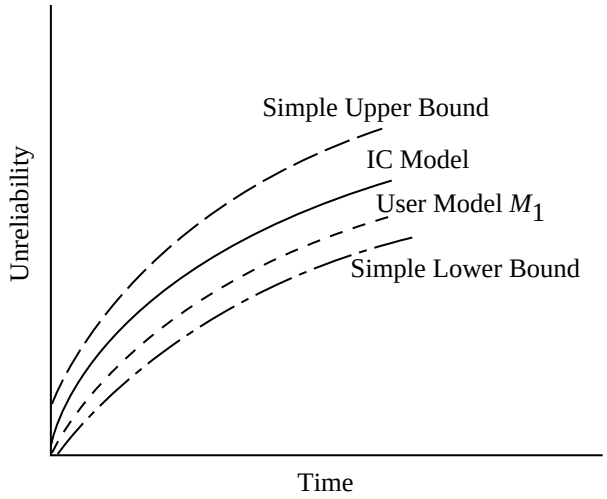
As mentioned in section 2.8, truncation bounds are obtained as follows. When the truncated model is solved, the probability of being in each of the TA states is calculated. By adding these probabilities to that of the *down* (failed) states (DS) before the truncation line, we get an upper bound on the system unreliability (SU). All states beyond the truncation line are assumed to be failed states. To get a lower bound on unreliability, we add only the probabilities of the failure states before the truncation line. Thus, the TA states are automatically considered to be functional states by HARP. To use some notation, the states in the truncated model are denoted with a subscript tr and the states in the full model have the subscript full. The bounds on the system unreliability are given by the following:

$$Pr(DS_{tr}) \leq SU_{\text{full}} \leq Pr(TA_{tr}) + Pr(DS_{tr})$$

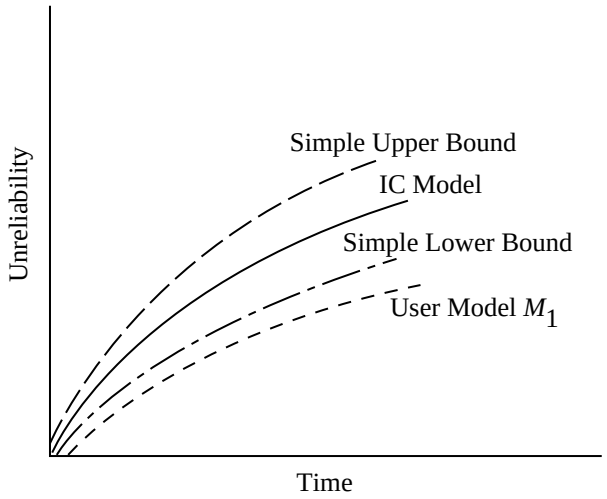
HARP not only gives the system unreliability but also provides a breakdown of individual failure probabilities. Failure causes are the exhaustion of different components, FNCF and FSPF.

[20] It is conservative in that the reliability of the system being modeled is not less than the HARP reliability estimate.

[21] HARP FEHM's and multifault models only support single recovery transitions. System models with multiple recovery transitions can cause the simple upper bound to improperly bound the HARP unreliability result ($M4$) or the original model ($M1$). For such systems, the user can edit HARP generated ASCII files or use XHARP. The HARP AS IS model can also be used to provide accurate results.



(a) Typical.



(b) Pathological.

Figure 23. Typical and pathological simple lower bounds.

In a truncated model, HARP gives bounds on the system unreliability as well as individual failure probabilities. $F1$ denotes a state where fewer components than the minimum required of component type 1 are still operational. If an $F1$ state occurs before the truncation level, we use the probability of being in the $F1$ state as a lower bound on the probability of failure due to exhaustion of component 1. All transitions due to failure of component 1 that fall on the truncation line and do not lead to state $F1$ are directed into a state called $TA1$.

Probability of failure due to exhaustion of component 1, $Pr(F1_{\text{full}})$, is bounded as follows:

$$Pr(F1_{tr}) \leq Pr(F1_{\text{full}}) \leq Pr(TA1_{tr}) + Pr(F1_{tr})$$

The bounds on the probability of exhaustion of other components are obtained in a similar manner. Now we obtain bounds for the probability of a FNCF and a FSPF.

The probability of being in the FNCF state before the truncation level is a lower bound on the FNCF probability. The upper bound is taken to be this lower bound probability added to the combined probability of all TA states:

$$Pr(FNCF_{tr}) \leq Pr(FNCF_{\text{full}}) \leq Pr(TA_{tr}) + Pr(FNCF_{tr})$$

The bounds on probability of SPF are obtained in a similar manner as follows:

$$Pr(FSPF_{tr}) \leq Pr(FSPF_{\text{full}}) \leq Pr(TA_{tr}) + Pr(FSPF_{tr})$$

3.4.3. Combined Bounds

When parametric bounds (via a simple model) are desired from a truncated model, the bounds are combined in the following way. The simple model solution uses the optimistic parameters (lowest possible failure rates, highest possible repair rates and coverage factors) to produce an upper bound on the reliability (R_{high}) of the system (ref. 53).

$$R_{\text{high}}(t) = 1 - \max[P_{\text{eshlow}}(t), P_{\text{covlow}}(t)]$$

where P_{eshlow} is the system failure probability due to exhaustion of system redundancy and P_{covlow} is the system failure probability due to minimal coverage.

If the model from which the simple bounds are derived is a truncated model, then the truncation aggregation states are taken to be operational states (for the optimistic bound). Likewise, the simple model solution uses the pessimistic parameters (highest possible failure rates, lowest possible coverage factors and repair rates) to produce a lower bound on the reliability (R_{low}) of the system (ref. 53).

$$R_{\text{low}}(t) = 1 - \min\left[P_{\text{eshhigh}}(t) + P_{\text{covhigh}}(t), 1\right]$$

If the model from which the simple model bounds are derived is a truncated model, then the truncation aggregation states are taken to be failure states (for the pessimistic bounds). The first type of bounds are reported as *simple model bounds*, the second type are reported as *truncated model bounds*, and the combined bounds are reported as *truncated simple model bounds.*

The use of behavioral decomposition and the instantaneous jump model factors have been proven to result in conservative estimates of reliability (ref. 8), when failure rates are constant (exponential times to failure). Both bounding techniques (simple and truncation) produce bounds on this conservative estimate of reliability. For practical highly reliable systems, the HARP (simple and truncation) bounds also encompass the reliability of the original model. When the disparity of the model sojourn times are too close to guarantee valid bounds, a warning message is issued.

Chapter 4

HARP Structure and User Input

4.1. Overview of Program Structure

The HARP solution of a reliability model consists of running three sequential programs: *tdrive* for model construction, *fiface* for interface, and *harpeng* for solution. (See fig. 24.)

Figure 24. HARP program structure.

In *tdrive*, the user is stepped through the model construction phase to produce the FORM, any FEHM's, and a dictionary file for the system. The FORM can be either as a fault tree (for nonrepairable systems) or a Markov chain, and the FEHM's can be any of the ones supported by HARP. For each component in the system, the dictionary file contains the logical name of the component, the symbolic name for the failure rate, the name of the FEHM parameter file, and any user-defined near-coincident fault rates. When the name of the FEHM parameter file is specified, the user can create the file at that time or specify that it already exists.

Supplied with this model representation, the *tdrive* program creates several output files that can be carefully edited and rerun or used by the interface program *fiface*. By keeping the input and interface programs separate, the user can use the fault tree or Markov chain information for any purpose by designing the appropriate interface. The *fiface* program uses the files created in *tdrive* to construct the symbolic transition rate matrix for use by the solution program *harpeng*. Additionally, symbol table information and failure state information is passed to the solution program which translates these representations into the system unreliability over a user-specified time period. If desired, the optimistic and conservative bounds are also supplied.

4.2. File Naming Conventions

Several files are created when running the HARP program, and the names of these files are derived from the user-supplied model name. Once the user has specified a model name that model name is used to create the filenames used throughout the HARP program. The model name

(up to nine characters in length, eight for PC) is appended with three character extensions to produce the reserved files. Special characters that can interfere with the user's operating system should be avoided, e.g., avoid using * or & as a model name or extension. Figure 25 shows the HARP structure and identifies the files with the program segment *tdrive*, *fiface*, or *harpeng*. The following sections give a representative listing of each of the HARP files. The typical file contents shown were obtained by running the example of figure 15 through HARP.

Figure 25. File structure of HARP.

4.2.1. MODELNAME.TXT

The symbolic textual fault tree description file is entered at the terminal by the user in program *tdrive*. It is then converted to MODELNAME.FTR so that it can be converted to a Markov chain for solution. Typical file contents using the example model in section 5 figure 14 are as follows:

```
NODE   1: TYPE BASIC,  3 OF COMPONENT   1
NODE   2: TYPE BASIC,  2 OF COMPONENT   2
NODE   3: TYPE AND          ,   1 INPUTS:   1
NODE   4: TYPE AND          ,   1 INPUTS:   2
NODE   5: TYPE OR           ,   2 INPUTS:   3  4
NODE   6: TYPE FBOX, INPUT:    5
```

4.2.2. MODELNAME.FTR

The fault tree description file is created either from the textual description file (.TXT) in the *tdrive* program or directly from the graphics program and is converted to a Markov chain for solution. Those lines beginning with an 'N' represent Markov chain nodes and those beginning with an 'A' designate arcs, arrows, or lines (connectors). The fields for the nodes are: N xcoor

ycoor type_node node_label. (See vol. 3 of this TP for more details.) Typical file contents are as follows:

```
N  1  1  16   3*1
N  2  2  16 2*2
N  3  3  19
A  1  1  3  3
N  4  4  19
A  2  2  4  4
N  5  5  17
A  3  3  5  5
A  4  4  5  5
N  6  6  22
A  5  5  6  6
```

4.2.3. MODELNAME.DIC

The dictionary file contains the logical name for each component type (e.g., processor, sensor), its symbolic failure rate parameter (e.g., lambda, mu) and the FEHM parameter filename (if any). It also contains the user-specified interfering component types. This file is created either by the textual input program or from the graphics program. Typical file contents are as follows:

```
1 PROCESSOR     LAMBDA        FEHM.CAR
  INTERFERING COMPONENT TYPES: 2
2 MEMORY        MU            NONE
  INTERFERING COMPONENT TYPES:
FEIDS     (See section 4.3.2.)
  7  6
```

The dictionary is required for fault tree FORM's. A Markov chain FORM requires the dictionary if coverage is to be included in the model. The dictionary matches failure rates with the coverage information file to correctly solve the model. It is designed as a tool for both the user and the program. The user can make changes in the dictionary file to accommodate any special modeling requirements. When creating the dictionary, do not use the symbol $ as a character in a failure rate symbol name. This symbol causes the program to ask the user to declare the meaning of the name without the symbol $ as well as with it, that is, two symbols result when only one is intended.

4.2.4. MODELNAME.INT

The symbolic textual Markov chain description file is created by *tdrive*. It is read by the interface program *fiface* and is converted to the symbolic transition rate matrix file (.MAT) for the HARP engine. The first line of the file is SORTED if the Markov chain was created from

a fault tree and either SORTED or UNSORTED if the input was a Markov chain. Typical file contents are as follows:

```
SORTED
1   2    3*LAMBDA;
1   3    2*MU;
2   4    2*LAMBDA;
2   5    2*MU;
```

Refer to section 3.3 for further information on the MODELNAME.INT file.

4.2.5. MODELNAME.MAT

The symbolic textual transition rate matrix is read by the HARP engine. The HARP engine requires a specific ordering of its matrix, with row and column values of nonzero entries entered in ascending order. Matrix entry i, j represents a transition rate from state j to state i. For entry 2,1 in the second row in the following table, for example, means that $i = 2$ and $j = 1$ and `3*LAMBDA*C1` is the transition rate from state 1 to state 2. The number 10 in the first row is the number of model states. Additionally, a symbol X is created and is concatenated to those transitions leading to the failure due to exhaustion state. It serves as a flag variable for the bounds computation. The end of the matrix is flagged with value 0,0. This file is created by program *fiface*. Typical file contents are as follows:

```
 10
 2  ,  1
3*LAMBDA*C1;
 3,  1
2*MU;
 4,  2
2*LAMBDA*C2;
 5,  2
2*MU;
 5,  3
3*LAMBDA*C3;
 6,  3
MU*X;
 6,  8
```

4.2.6. MODELNAME.SYM

The symbol table and failure (and possibly operational) state information file also contains whatever symbol table information can be deduced from the graph. Specifically, for each coverage factor (i.e., C_i), it lists the symbol type number (always a 3 for FEHM types other than VALUES) and the parameter file containing the FEHM information. Additionally, for FEHM type VALUES, the corresponding R_i and S_i are listed for each component type with the "VALUES" designation. In this case, the symbol type numbers are 7 for the C_i, 8 for R_i, and 10 for S_i. If near coincident fault rates are being considered, the N_i values are also printed

with a symbol type number of 9. The symbol X, which appears in the *.INT file (not shown in MODELNAME.INT) denoting transition to the failure state, is assigned the numeric value 999.

For the following file contents, the data can be interpreted for figure 17 as follows. The coverage parameter C1 is defined by the FEHM given by the file FEHM.CAR and has the number 3 associated with it to designate that the FEHM is not a "VALUES" FEHM. Figure 17 shows C1 as a factor in the transition rate, $3\lambda C1$ between states 3, 2 and 2, 2. The C4 parameter in the file has the number 7 below it, which designates that C4 is defined by a VALUES FEHM and the value of C4 is 0.7000000000 with tolerance 0.0000000. Likewise, the probabilities and tolerances for the R4, N4, and S4 transitions are listed as well. (See section 2.6 for details of their meaning.)

The user has the option of entering the values for these parameters in program *fiface* or in program *harpeng*. If the user elects to enter the values in the engine, *fiface* lists the values as -1.00. All failure states (and operational states whose probabilities are desired) are listed in this file as well as their location in the matrix. To interpret where a failure state is located, subtract 1000 from the absolute value of the number listed. This file is created by program *fiface*. Typical file contents are as follows:

```
C1
3
FEHM.CAR
C2
3
FEHM.CAR
C3
3
FEHM.CAR
C4
7
  0.700000000000  0.000000000000
R4
8
  0.100000000000  0.000000000000
N4
9
  0.100000000000  0.000000000000
S4
10
  0.100000000000  0.000000000000
X
999
```

```
END SYMBOL DEFINITION
F1
  1007
F2
  1006
FSPF
  1009
FNCF
  1010
END FAILURE STATE DEFINITION
```

4.2.7. MODELNAME.ALL, MODELNAME.SAM, and MODELNAME.USR

The near-coincident fault rate information files are also created by program *fiface*. For each coverage factor (i.e., C_i), they list the symbolic value of the near-coincident fault rate. MODELNAME.ALL lists the symbolic value of the ALL-inclusive near-coincident fault rate, MODELNAME.SAM the SAME-type near coincident fault rate, MODELNAME.USR the USER-defined near-coincident fault rate. Section 2.7 gives more information on user-defined near-coincident fault rates. Typical file contents for MODELNAME.ALL, MODELNAME.SAM, and MODELNAME.USR are similar. The following expression C_i is the near-coincident fault rate associated with the C_i transition. It is not equal to C_i.

```
C1
 2*LAMBDA+2*MU;
C2
 LAMBDA+2*MU;
C3
 3*LAMBDA+MU;
C4
 2*LAMBDA+MU;
```

4.2.8. MODELNAME.INP

The MODELNAME.INP is an echo file containing the name of the matrix file and values for the symbolic rates defined by the user at runtime (of the HARP engine). This file can be edited after the HARP engine program has completed; thus, the need to enter parameter values during future runs is eliminated. This file is an output of the HARP engine program. Typical file contents are as follows:

```
3P2M.MAT
Symbol No. Symbol      Type       Value          Variation
    1      LAMBDA       1     0.10000000D-03  0.10000000D-06
    2      MU           1     0.10000000d-01  0.10000000D-04
```

Refer to section 3.5 for more details on the input file.

4.2.9. MODELNAME.RS*

The MODELNAME.RS* is a textual file with the reliability/unreliability values for the model is an output of the HARP engine program. Each time the HARP engine is rerun during the same session, the output is appended at the end of MODELNAME.RS*. The asterisk (*) is an integer, beginning with 1, that is incremented for each rerun of the engine during the same session. The HARPO module reads this file to make interactive graphical analysis available. (The HARPO module expects an upper case filename extension.) Up to nine runs can be executed during the same session. (Note: if the program is terminated and then rerun, all files are destroyed and rewritten.) Typical file contents are as follows:

```
---------------- HARP ----------------------
- The Hybrid Automated Reliability Predictor -
------------ Release Version 7.0 -------------
--------------- February 1993 ----------------
Modelname:
  3P2M
Input description (from dictionary file):
Component type: 1      Name: PROCESSOR
Symbolic failure rate:
LAMBDA       Constant failure rate:
                   0.10000000D-03     +/-       0.10000000D-06
FEHM file name: FEHM.CAR
   For this FEHM model, the exit probabilities are:
      (in the absence of near-coincident faults)
        Transient restoration:       0.00000000D+00
        Permanent coverage:          0.99956467D+00
        Single-point failure:        0.43532615D-03
Component type: 2      Name: MEMORY
Symbolic failure rate:
MU           Constant failure rate:
                   0.10000000D-01     +/-       0.10000000D-04
FEHM file name: NONE
ALL-INCLUSIVE near-coincident fault rate used.
Time(in Hours):    0.100D+02
State Probabilities
State name: F1             0.99203074D-09
State name: F2             0.90559086D-02
```

```
State name: FSPF           0.13012236D-05

State name: FNCF           0.13617744D-06

-------------------------------------------------

              Reliability =   0.99094265D+00

            Unreliability =   0.90573470D-02

Total failure by redundancy exhaustion =   0.90559096D-02

-------------------------------------------------

Parametric Bounds using SIMPLE Model:

    Lower Bound on Unreliability =   0.90387040D-02

    Upper Bound on Unreliability =   0.90745973D-02

See Users Guide, section 3.4.1 for interpretation.

GERK ODE solver:  global error value   0.200D-15

               relative error value   0.100D-08

See Users Guide, section 3.3.1 for interpretation.

      0 Reports from the GERK ODE solver.
```

4.2.10. MODELNAME.PT*

A textual file containing the unreliability values for the model is plotted along with the bounds values (if any) and output by the HARP engine program. In the following table, the left most column gives the times at which their corresponding unreliability values in the right column are computed. These values are provided as input data for a user's plotting program. The asterisk (*) is an integer, beginning with 1, that is incremented for each rerun of the engine during the same session. Up to nine runs may be executed during the same session. (Note: if the program is terminated and then rerun, all files are destroyed and rewritten.) The contents of this file can be created with a text editor and used as input to the HARPO module. Also, if data are generated by another program and can be put into the *.PT file format, HARPO can display that data also, possibly for comparative analysis. Typical file contents are as follows:

```
 0.00000000            0.00000000E+00

10.00000000            0.90570200E-02
```

Additionally, the input programs create the fault/error handling model parameter value definition files. These files have different formats, corresponding to the choice of the fault/error handling model specification technique. The first line of the file specifies the type of model, such as HARP.SINGLE.FAULT.MODEL, and the necessary parameters follow. Note: these files do not have the near-coincident-fault rate expressions; instead, the near-coincident fault-rate expression is an attribute of the particular coverage symbol. Different coverage symbols may have the same fault/error handling model parameter files but use different near-coincident fault-rate expressions. These files are created in the textual and graphical input programs.

4.3. Inputting a Markov Chain FORM

4.3.1. State Transition Specification

A Markov chain is entered in the following format:

```
state_x  state_y  rate_transition
```

The user can enter the information in sorted or unsorted order. If the sorted option is chosen, the state names must be integers listed in row-wise order (beginning with the number 1). First, all transitions emanating from state 1 are listed, then those from state 2, etc. If the unsorted option is chosen, the state names can be nonintegers listed in any fashion. However, the first state listed is assumed to be the initial state of the Markov chain. It is assigned an initial state probability of 1, while all other states have an initial state probability of 0. Also, if the input is unsorted, the size of the model is limited—a total of 500 states can be included and up to 2050 transitions. For sorted (or fault tree) input, the number of states is increased to 10 000 and the number of transitions to 90 000. These limits can be changed, however, as explained in a section 5.3. For either type of Markov chain input, the state names cannot be more than 13 characters. Comments can be imbedded in the text by beginning and ending a line with the asterisk (*) so that the line is printed in the MODELNAME.INT file but ignored by program *fiface*.

4.3.2. Failure State Specification

In HARP, any state whose label begins with the letter F is considered a failure state. Four types of failure states are represented. For failure by redundancy exhaustion, one failure state is associated with each component type in the system. These failure states are labeled F_i, where i is the component type number failing and F stands for "failure due to exhaustion." For those models with imperfect coverage, the occurrence of a single-point failure and near-coincident fault failure is recognized by failures states FSPF and FNCF, respectively. These latter two states are added by the interface program, *fiface*, automatically. Any other state label beginning with F contributes to the system unreliability but not to the specific failure probabilities.

When the Markov chain input type is sorted, the user must enter the state names as numbers; therefore, a state label beginning with an F is not allowed. In this instance, HARP can recognize the failure states in one of two ways. First, to run bounds in the engine program, those transitions entering failure states must have *X appended to them. Therefore, when inputting the FORM, the user can add this *X to the appropriate transitions to designate the state into which the transition goes as a failure state. Second, the user can edit the dictionary file (.DIC) by adding the following lines for failure ID's (FEIDS) to the end of the file:

```
FEIDS

  f1  f2  f3 ... fn
```

where f1, f2, f3, ..., fn are positive integers that identify the failure states F_1, F_2, F_3, ..., F_N, respectively. Note that adding *X to the failure state in an unsorted Markov chain is not allowed.

4.3.3. Solving Arbitrary Markov Chains

HARP can be used to solve arbitrary (general) Markov chains simply by stating that the model being described is to be solved AS IS. Under this designation, no FEHM models are inserted.

4.3.4. Sorted Versus Unsorted Input

When the FORM input is a fault tree, HARP converts this fault tree into a sorted Markov chain. However, many systems cannot be modeled using fault trees. Hence, the user must enter a Markov chain as input. As previously mentioned, the user can enter the Markov chain in sorted or unsorted order. If the model being evaluated is very large, then the user should input the Markov chain in sorted order because only 500 states and up to 2050 transitions can be included in an unsorted model. A sorted model, on the other hand, can have 10 000 states and up to 90 000 transitions.

Several differences should be noted about sorted and unsorted Markov chains. If in a sorted Markov chain, $*$X's are appended to the failure states, then HARP evaluates bounds on the reliability. Moreover, only the probabilities associated with the failure states are given, not those of the operational states. However, if no $*$X's or FEIDS are specified in the input, the program *harpeng* (solution stage) asks the user to specify failure states. If the user does not specify any failure state, the state probabilities of all states are printed while the system reliability is not given (since no failure states are specified). If the user does specify failure states when asked, then the system reliability and the failure state probabilities are printed. In either case, bounds are not evaluated.

For sorted input, if FEIDS are specified, then the resulting failure probabilities are listed in the order in which the states are listed in the FEIDS. For example, if state 12 is mentioned first under FEIDS in the MODELNAME.DIC file, then the failure state F1 corresponds to state 12 and F2 to the next state mentioned under FEIDS and so on.

4.3.5. Labeling Transitions

The Markov chain transitions are normally symbolically labeled with an expression of the form: constant $*$ failure rate. Failure rate transitions are denoted by a single failure rate variable (i.e., λ or μ) even though HARP does not require the failure rates to be constant. The failure distribution is specified as either exponential or Weibull at run time. In general, an arc between states (i, j) and $(i-1, j)$ is labeled with the value $i * \lambda$ (if λ is the failure rate of component type 1). Likewise, an arc between states (i, j) and $(i, j-1)$ is labeled with the value $j * \mu$ (if μ is the failure rate of component type 2).

Although most transitions are of the type previously described, transitions between arbitrary pairs of states with arbitrary labels are certainly permitted. However, the following restrictions apply:

- There can be only one level of parentheses.
- Only addition and subtraction are allowed within the parentheses.
- Only addition, subtraction, and multiplication are allowed outside the parentheses.
- The rate expression cannot exceed 23 characters.
- Other than the previously listed mathematical symbols, only alphabetic characters (upper or lower case) and numerals are understood by the HARP engine.

4.4. Inputting a Fault Tree FORM

4.4.1. Replicated Basic Events

To reduce the size of a model, HARP allows statistically identical components to be combined into single basic events. A replicated basic event is labeled with an expression of the form

$m * n$, representing m replications of redundant, functionally identical components of type n. Replication is useful when modeling statistically identical components with the same failure rate value, for example, three processors (fig. 15). Because HARP converts the fault tree to a Markov chain for solution, this combination of equivalent components reduces the size of the resulting Markov chain considerably. Suppose a fault tree has j basic events, each with a replication factor of k_i. If every component were required to fail before the system fails, then the resulting Markov chain using the multiple basic events would have $\prod(k_i + 1) - 1 + j$ states. If the basic events were all separate, then there would be $\sum k_i$ basic events and the resulting Markov chain would have $2^{\sum k_i}$ states. Consider such a system having 5 basic events, each with a replication factor of 3. The Markov chain resulting from the tree with replicated basic events would have 1028 states, and the Markov chain resulting from the fault tree without replicated basic events would have $2^{15} = 32\,768$ states.

4.4.2. Representation of Shared Events

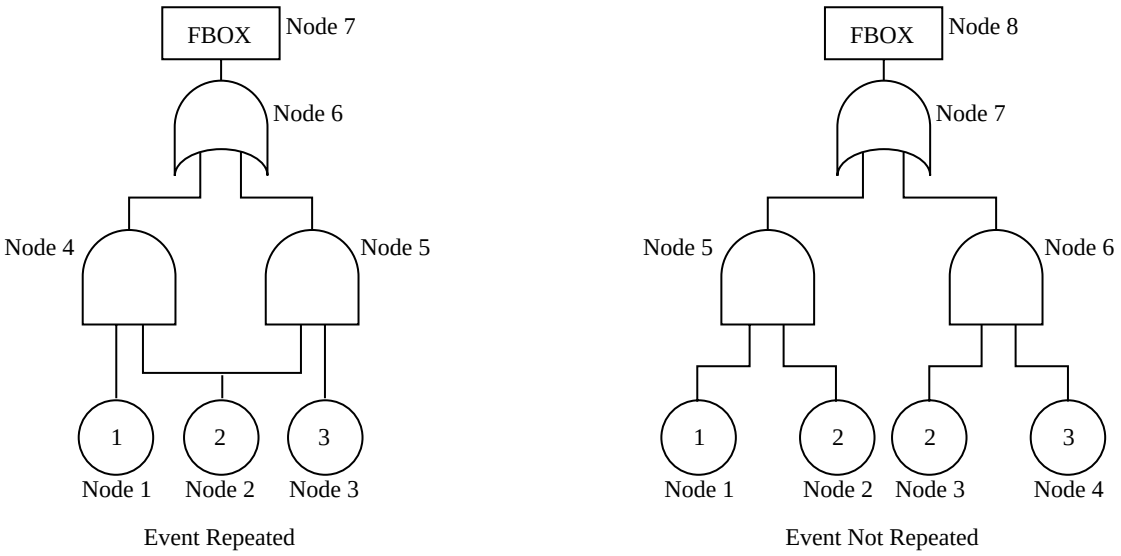
The user should be aware of a source for potential confusion when constructing fault trees. The difficulty is only evident when the fault tree contains shared events because HARP uses a representation for shared events that differs from the one often found in the literature. A shared event is a basic event that is used more than once in the fault tree, that is, a basic event that affects the failure of the system in more than one way and thus has more than one parent gate or box. In the literature, such repeated events are sometimes depicted by multiple occurrences of its basic event node in the fault tree. However, HARP uses the convention that each basic event node represents a distinct basic event that is assigned a numeral by the user. If a single basic event is used in more than one place in the fault tree, then it should still be depicted by only one basic event node, that is, the same node numeral. This basic event node has multiple outgoing arcs, one to each parent node, to represent the fact that the event is a shared event. The GO program (see vol. 3 of this TP) represents a shared event as a double circle. The shared basic event is initially drawn as a single circle. All other multiple occurring events associated with the initial basic event are referenced back to the initial single circle basic event. The double circle notation is provided for drawing convenience and to simplify the drawing by reducing connecting arcs.

Conversely, two or more basic events with the same label but different node numerals represent two or more distinct basic events that happen to be the same component type. The fact that basic events have the same label does not make them a shared event, having the same node numeral does.

In the fault tree labeled "Event Repeated" in figure 26, a single component, node 2 labeled 2 (P2 in MODELNAME.DIC), appears as an input to two different gates (node 2 is shared). In the fault tree labeled "Event Not Repeated," two individual components, nodes 2 and 3, are both labeled 2 (P2 in .DIC), each being an input to only one gate (not shared). In the latter case, the two individual components, nodes 2 and 3, are functionally different components within the fault tree, although they happen to have the same label and therefore are the same type of component.

4.4.3. Example of a Functional Dependency Gate

This section introduces a functional dependency gate that can be used to simplify the generation of a fault tree model of a system exhibiting structural dependencies of components. Suppose a system is configured such that the failure of some component (called a *trigger* component) causes other dependent components to become inaccessible or otherwise unusable. In this case, later failures of the dependent components will not further affect the system and



Dictionary File

Component Type	Logical Name	Symbolic Failure Rate	FEHM File Name
1	Processor	Lambda1	Fehm.car
2	P2	Lambda2	None
3	Memory	Lambda3	None

Figure 26. Repeated and distinct basic events in HARP.

should not be considered. A functional dependency gate has a single trigger input (either a basic event or the output of another gate in the tree), a normal output (reflecting the status of the trigger event), and one or more dependent output events. The dependent outputs are basic events that depend on the trigger event. When the trigger event occurs, the dependent basic events are forced to occur. The occurrence of any dependent basic events has no effect on the trigger event.

For an example, consider the $Cm*$ system (ref. 20) (shown in fig. 27), which consists of clusters of processors and memories connected by links. Each cluster consists of eight local switch interface controllers (S.local), each attached to one processor and one 12K-memory module. Each processor has 4K of memory on board. The K.map is a cluster controller connecting the S.locals; the clusters are connected by intercluster communications (L.inc). A fault in the K.map renders the S.locals (and their connected processors and memories) inaccessible, while a fault in the S.local makes the processors and memories connected to it inaccessible.

The development of the fault tree model for the $Cm*$ system (shown in fig. 28) is simplified by the use of the functional dependency gate. The dependence of the S.locals on the K.map can be captured by two functional dependency gates, each with a K.map trigger event and four S.local dependent events. Similarly, the dependence of the processors and memories on the S.locals is captured in eight functional dependency gates, each with an S.local as the trigger event and the associated processor and memory as the dependent events. The system is considered operational as long as three processors can communicate with three memories. As long as the L.inc is operational, the requirements can be satisfied by the components of both clusters (thus the 6/8 gates). If the L.inc fails, the requirements must be met within one cluster (thus the

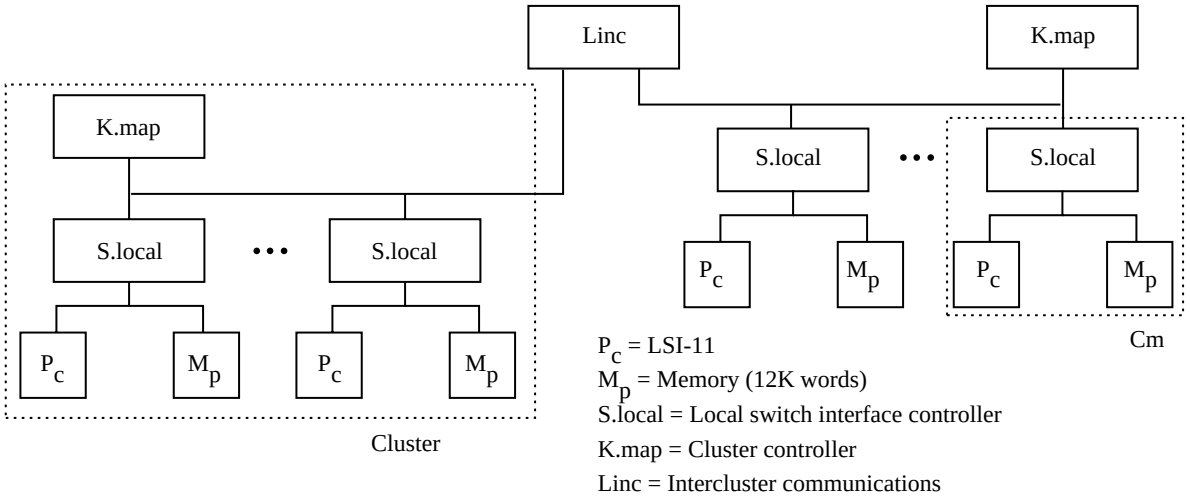


Figure 27. $Cm*$ system used as example.

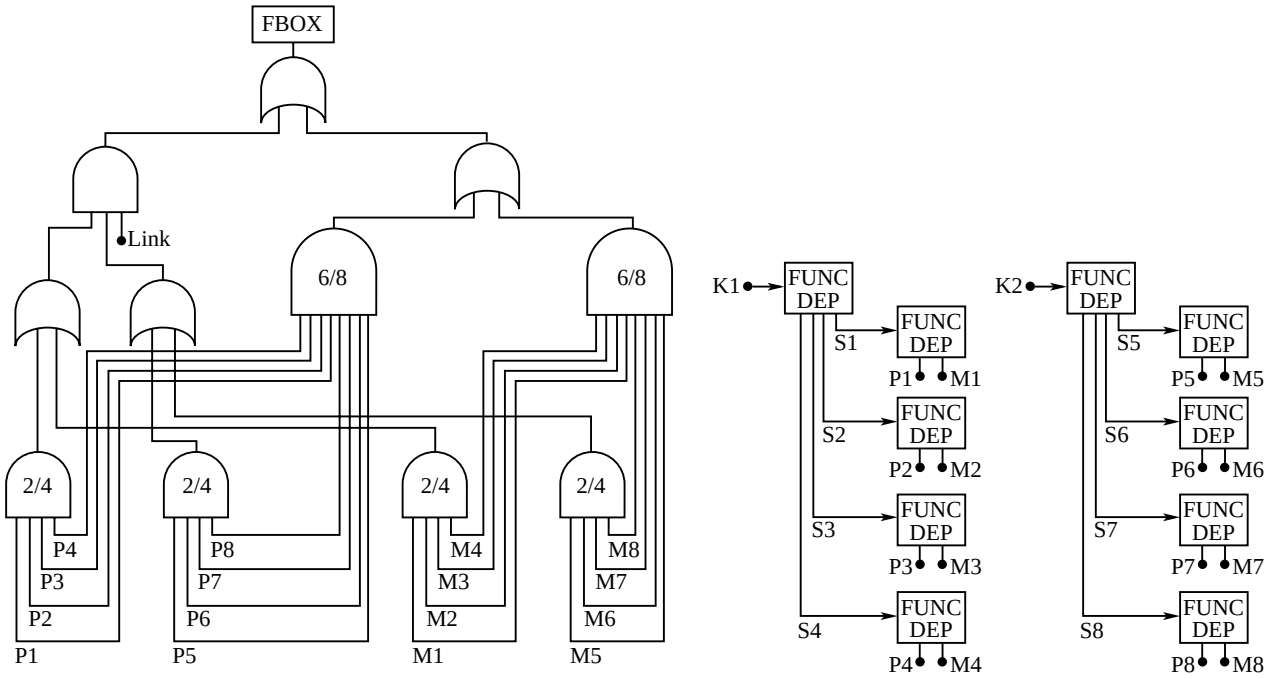


Figure 28. Fault tree model of $Cm*$ system.

2/4 gates). The outputs of the functional dependency gates need not be used as inputs to any other gates in this instance.

The fault tree to Markov chain conversion uses the functional dependency gate to alter that state descriptor. In a state representing the occurrence of a trigger event, the state descriptor is changed such that all dependent events are recorded as having failed, if they have not already done so. No coverage model is considered for these dependent component failures to insure that the possibility of imperfectly covered failure of a component that is unusable or inaccessible cannot contribute to system failure. (The absence of a coverage model produces an optimistic result because any coverage value other than unity reduces system reliability.) Furthermore,

the basic events that are forced to occur by a trigger event do not count as component failures when determining the failure level of a state. If the first component failure is a trigger event that removed two additional components, then the resulting state has three component failures. However, this state is considered to be on the first failure level of the Markov chain (i.e., it is a member of the set of states that result from the covered failure of one component only). A coverage model is included on the arc representing the failure of the trigger event. Because no explicit arc represents the occurrence of the dependent basic events, no coverage model is included for these events.

Although the functional dependency gate does not increase the modeling capacity of the fault tree, it can reduce the effort required to develop a fault tree model of a system with complex interconnections.

4.4.4. Example of Priority and Gate

The priority and gate is logically equivalent to an AND gate, with the added requirement that the input events occur in a specific order (refs. 54 and 55). In HARP, the number of inputs for a priority and gate is limited to two for implementations reasons. However, priority and gates can be cascaded together to achieve the effect of multiinput priority and gates. (See fig. 9.) As an example of the use of a priority and gate in a fault tree, consider a system that consists of two channels, as shown in figure 29. Each channel has two sensors, `A1` and `A2`, that are connected by a device interface unit (DIU). One sensor is a primary channel, the other is an alternate. The system begins by operating in channel one. Upon the first failure affecting channel one, the system switches to channel two if channel two has not experienced any component failures. After switching to channel two, the system continues to use channel two until it fails, at which time the system fails. If after the first failure on channel one, the system does not switch to channel two, then it remains on channel one until channel one fails, at which time the system fails.

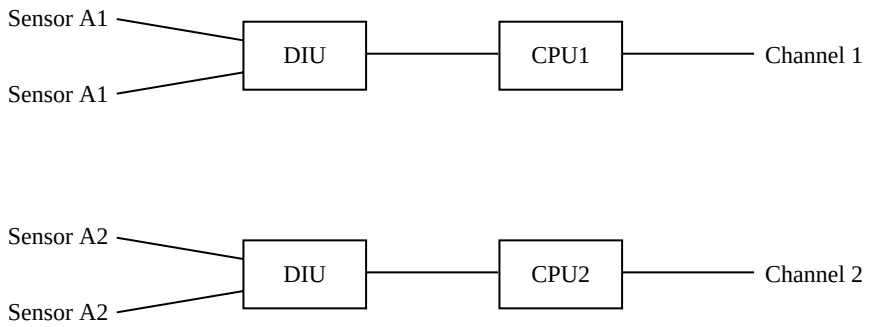


Figure 29. Two-channel system.

Figure 30 is a fault tree model of this system. The fault tree for this system utilizes two priority and gates, which input to two and gates. The leftmost priority and gate represents the situation where a failure occurs on channel one, causing a switch to channel two, and then channel two fails. The rightmost priority and gate represents the situation where something fails on channel two (and thus when a failure on channel one occurs the system stays on channel one) and then channel one fails subsequently.

4.4.5. Example of Cold Spare Gate

As an example of the use of the cold spare gate, consider a system consisting of six components: A, B, C, D, E, and F. The system operates as a triad, with A, B, and C active

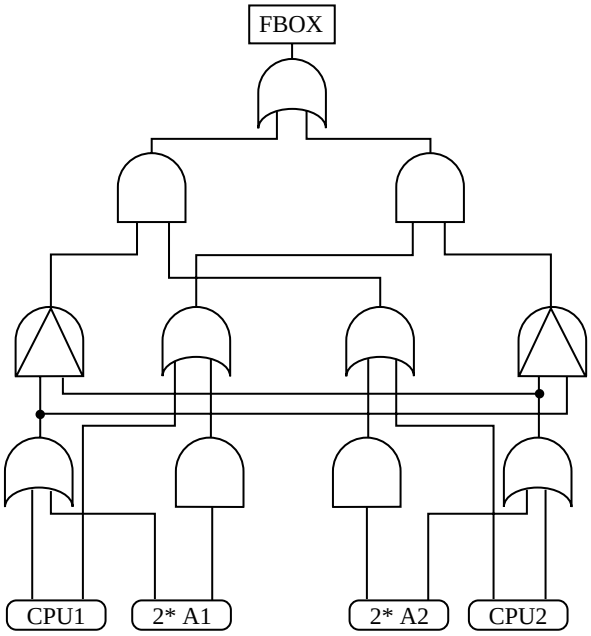


Figure 30. Fault tree model of two-channel system.

initially and D, E, and F as cold spares (inactive but not subject to failure). Components D and E can substitute for A if A suffers a covered permanent failure. Component D is the first alternate; it gets put into active use when A fails. If D then experiences a covered failure, E is switched into active use. Components D and E cannot fail before they are switched into use. Component F can substitute for either B or C, whichever fails first. Thus, F is a shared cold spare (also called a pooled spare). Note that if an external event called G causes a spare called D to fail, then component D is no longer available for the cold spare gate connected to it.

The fault tree model for this system appears in figure 31. The cold spares dependencies are captured by the boxes labeled "Cold Spare Gate." The leftmost input to the cold spare gate is the primary component, and the others are the alternates (cold spares) for the primary component. The order in which the cold spare components input to the cold spare gate (left to right) implies the order in which the spares are switched into active use. The output of the cold spare gate fires when the primary component and all its alternates have failed. For shared cold spares, the output fires if the primary component has failed, and either its alternate fails or its alternate has already been switched into active use for another component (and thus is no longer available for use as an alternate for the primary component). All inputs to cold spare gates must be basic events (possibly replicated). The input events of the cold spare gate are not allowed to have Weibull failure rates in HARP. (See section 2.9.1.)

4.4.6. Example of Sequence-Enforcing Gate

Figure 32 shows the use of sequence-enforcing gates to model state dependent FEHM's in the fault tree notation. This fault tree models a majority voting 2 out of 3 system where perfect (unity) coverage is assumed for the first failure and a user-specified FEHM is assigned to the second failure. The details of this model are discussed in chapter 7.

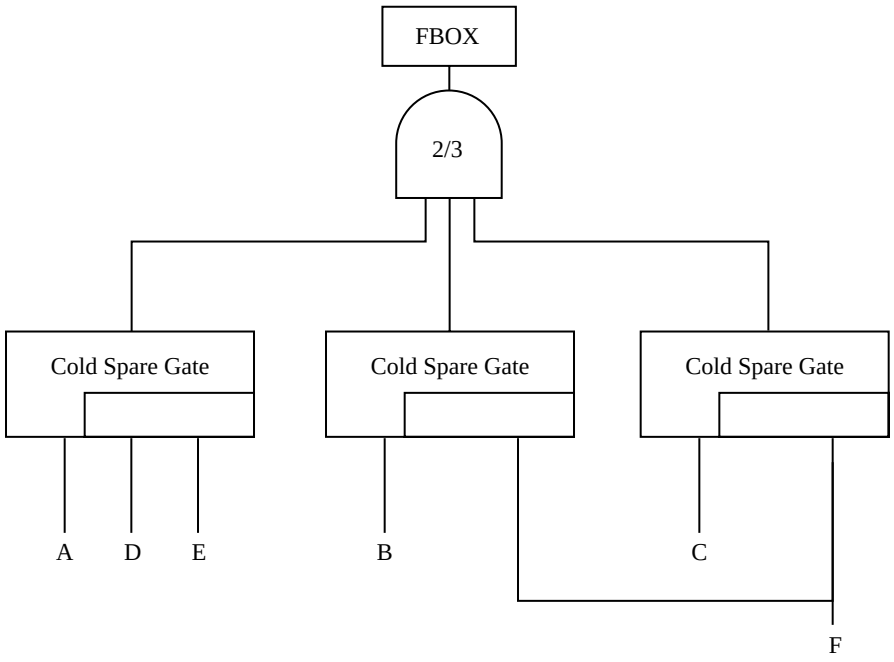


Figure 31. Fault tree model of system with cold spares.

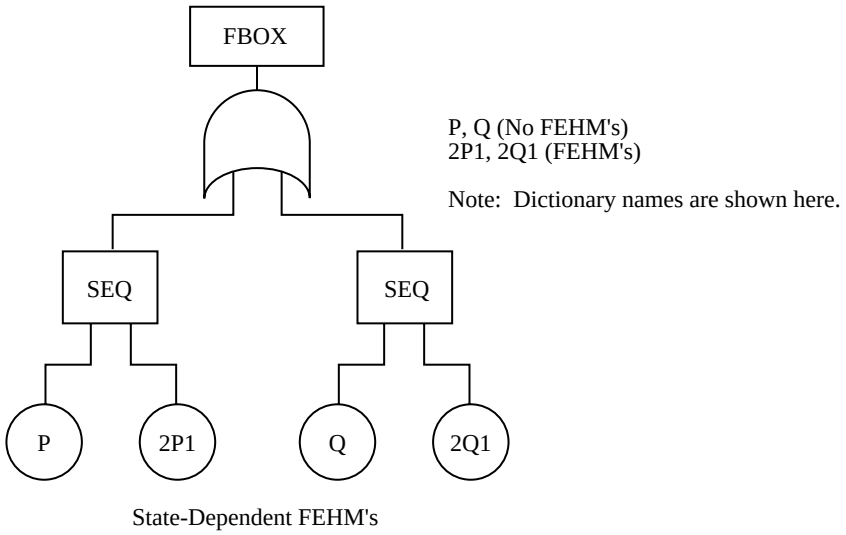


Figure 32. Fault tree model of system with a sequence-enforcing gate.

4.5. Editing MODELNAME.INP File

The MODELNAME.INP file is an output of *harpeng*. It is a text file containing the name of the matrix file and the type and values for the symbolic rates defined by the user at runtime (of the HARP engine). This file can be edited after the HARP engine program run is complete. This eliminates the need to enter the parameter values again during future runs of the same model.

Because the old format of the MODELNAME.INP file was inconvenient for the user, it is now written in tabular form, as shown in section 4.2. For a Weibull failure rate, the value of the

symbol refers to the lambda parameter and the variation is the alpha parameter. Also, symbol type 21 denotes that the symbol has a Weibull failure rate of type 1 and symbol type 22 denotes that the symbol has a Weibull failure rate of type 2. The HARP engine program still accepts any MODELNAME.INP file written in the old format and rewrites it in the new format.

When the system being modeled is large, the symbolic matrix generated by the model can be too large to store in the data structures internal to HARP. Then, any calculations that require reevaluation of the symbolic matrix, for example, the computation of bounds, are not possible. In such cases, the HARP engine does not need the parameter variation values in the input. If the engine requires the variation value and it has not been specified, then it is assumed to be zero.

The exact size of the model that causes the generated symbolic matrix to be too large for the HARP internal data structures cannot be determined because the size of the symbolic matrix for the model depends on the size and number of transitions in the model.

4.6. Entering Dictionary in *tdrive*

The dictionary is both a tool for the user and for the HARP program. As previously mentioned, the dictionary is required for fault tree input and for Markov chain input if coverage modeling is desired. Each component type that can fail in the system should be listed, that is, processors, sensors, and actuators. (For Markov chain FORM's, do not enter component repair information in the dictionary.) For each component type, the failure rate symbol is given, that is, lambda, mu, rho, etc. The *tdrive* program then asks for the coverage model to be used (ESPN, ARIES, CARE, distributions, moments, empirical, values, none) for the component type. The user can then specify a preexisting file containing the appropriate parameter information or create the FEHM file by supplying a filename into which the model parameters should be stored. Once this information is given for each component type in the system, the user is asked about user-defined near-coincident faults (only if there are coverage models other than NONE or VALUES). For each appropriate component type, the user lists all component types that can affect the given component type in terms of a second fault crashing the system. Once the dictionary is complete, the FORM is entered.

To model certain peculiar features of the system under study, the modeler can alter the dictionary manually. However, care must be taken in making any changes to the dictionary. For example, the length of the rate parameters and component names cannot exceed 12 characters. The user must ensure that the number of component types in the dictionary equals the number of failure exhaustion states in the MODELNAME.INT file. The grammar associated with the MODELNAME.DIC file is restrictive; therefore, while making changes, the user should not delete the blanks at the end of each line. If the interfering component type numbers exceed more than one line in the dictionary, they should be continued on the next line starting from the first position (i.e., no blanks at the beginning of the line).

4.7. State-Dependent FEHM—Overriding Default Model

Overriding the FEHM associated with a component type on a specific failure transition is possible. To do so, a colon is inserted into the transition label before the semicolon, followed by the name of the new FEHM file or by the word NONE (signifying no FEHM for this transition). For instance, if the transition label is `4*GAMMA;`, we can change the label of the Markov state transition to `4*GAMMA:FEHM.NEW;`. This change does not affect any other transitions triggered by a failure of the specific component type. If we want, we can instead turn off the FEHM for this transition by using the keyword NONE. For this example, the state transition label then becomes `4*GAMMA:NONE;`.

In some cases, state dependent FEHM's can be described at the fault tree level with the sequence gate. (See section 2.5 and chapter 7.) As an example, consider a three-processor system where each processor has a failure rate of λ. When all three processors are operational, the FEHM associated with the transition is say, FEHM.1. After the first fault (i.e., two operational processors), the FEHM used is FEHM.2. The system fails when two out of three processors have failed. This model can be described by a sequence gate with two inputs A and B, where A occurs before B. Input A has associated failure rate of λ_1, which is assigned the numerical value of $3 * \lambda$ at run time. Input B has associated failure rate of λ_2, which is assigned the numerical value of $2 * \lambda$ at run time. In the dictionary file, FEHM.1 is assigned to λ_1 and FEHM.2 is assigned to λ_2.

If for a particular component type the user has specified type VALUES in the dictionary, (rather than a FEHM model), has chosen to ignore near-coincident faults (NONE), or has chosen to ignore coverage completely, then the default model (or lack of a model) cannot be overridden. This restriction exists because each of these choices results in state-independent coverage values, which cannot be later made state dependent. Likewise, a transition cannot be overridden by typing VALUES because of its state independence.

To make it easier for the user to decipher the state of individual components for a particular Markov chain state, the MODELNAME.INT file can be optionally augmented by comment lines. If the user responds affirmatively to the *tdrive* question "Include state tuples as comments in the .INT file?", then each line (arc designation) in the MODELNAME.INT file is preceded by a comment line (beginning with a "*" in the first column). This comment line shows the state descriptor for the source and destination states for the arc. For example, suppose there is an arc from some state 41 to some other state 56 in the MODELNAME.INT file. Suppose further that state 41 represents a configuration with three components of type 1, two components of type 2, and zero components of type 3, and the arc represents a failure of component type 2. Then, the entry in the MODELNAME.INT file is: `41 56 2*LAMBDA2;`. If the MODELNAME.INT file is commented, then the line preceding this line is: `* 3 2 0 -> 3 1 0 2*LAMBDA2;`.

Chapter 5

Technical Information

5.1. Error and Warning Messages

An electronic file called MESSAGES.TXT is included on the tape with the source code; this file explains the meaning of each error or warning message from HARP. Within each program, error messages are numbered beginning with 500; warning message numbers are less than 500. To discern the meaning of an error or warning message, simply search the MESSAGES.TXT file for the corresponding message number. The text lists the subroutine name and source file from which the message originated, an explanation of the message, and a course of action (where possible) for correcting the error.

5.2. Installation of HARP Program

The HARP package, which is received on magnetic media has six directories: TDRIVE, FIFACE, HARPENG, TESTDIR, EXECUTE, FSOLVER. For a DEC VMS installation, the user has two options for compiling and linking: using command files or using MMS (Module Management System) files. In the TDRIVE, FIFACE, and HARPENG directories contain source files and a FORTIT.COM file and a LINKIT.COM file. The former creates the needed object modules, and the latter creates the executable. The user can invoke the *.com files by typing @FORTIT.COM then @LINKIT.COM in that order for each subdirectory. The MMS file is entitled DESCRIP.MMS and also appears in each directory along with the *.COM files. The user can invoke MMS by typing mms in each subdirectory. For a UNIX installation, Makefiles are included in each directory. The user can invoke the Makefiles by typing make in each subdirectory. The executables are entitled TDRIVE for the driver portion of the code, FIFACE for the interface and HARPENG for the engine. Once compiled, the user can move these executable files to a new location. Generally, we operate with these files in the EXECUTE directory and put this directory in our path. The code is configured to model systems with at most 10 000 states and up to 90 000 transitions (excluding diagonals of the matrix as HARP automatically calculates the diagonals). These limits can be changed using the information provided in the next section.

Once installed, the version can be tested against the three examples in directory TESTDIR. In addition, scripts of actual runs are found in the EXECUTE directory, named SCRIPT.FT and SCRIPT.MC. These files create 3P2M1BFT and 3P2M1BMC, respectively. The output files of these runs are also in directory EXECUTE.

The directory FSOLVER contains the source code for the CFEHM program, an editor for FEHM's. Use the FORTIT.COM and LINKIT.COM files or DESCRIP.MMS listed therein to create the DEC VMS executable. Makefiles can be used for UNIX installation. Section 5.4 contains information on the CFEHM program. The accompanying tutorial will help familiarize the user with running the HARP program.

5.3. Changing Limit Sizes for HARP Program

The HARP program as received is configured for up to 10 000 states and up to 90 000 transitions.[22] In addition, the HARP engine program has a limit of 15 000 symbols in the model. These parameters can all be changed as described in the following sections.

5.3.1. Program *tdrive*

To change the number of states in program *tdrive*, perform the following steps:

1. In the ft2mc.for source file, locate the following line (occurs only once) with an editor:

   ```
   PARAMETER (MSTATS = oldval)
   ```

   where "oldval" is an integer and represents the maximum number of states TDRIVE is currently set to handle. Change "oldval" to the new value "newval" desired for the number of states.

2. Recompile and relink.

Internally, *tdrive* uses linked lists implemented by routines that allocate and manage individual regions of large integer arrays. Two such arrays are defined in the subroutine FT2MC() in file ft2mc.f. The array POOL() has its length defined by parameter PLEN; similarly array DPOOL() has its length defined by parameter DPLEN. If a HARP error message indicates an operation failed because of insufficient memory, increasing PLEN and/or DPLEN and recompiling the program may prove sufficient.

5.3.2. Program *fiface*

To change program *fiface* so that it can run larger problems, the following variables must be changed. The program *fiface* has two state and transition sizes—those for SORTED input and those for UNSORTED (or with symbolic state names) input. If input is in sorted order (a fault tree converted to a Markov chain from TDRIVE is always in sorted order), then the state size can be up to 10 000 and transitions size up to 90 000. On the other hand, if the input is not in row-wise order or if the state names are symbolic, then the limits are 500 for state size and 2050 for transition size. If your model is UNSORTED and does not fit in the data structures, first try to put the MODELNAME.INT file in row-wise order with state names having increasing integer values beginning with 1. (This scheme is more efficient and easier than altering the code.) If the state size and transition size are still too small, increase the following sizes for SORTED input.

- NODES: in common block DATACB

  The size of this array represents the number of TRANSITIONS in a SORTED model. DATACB contains the transitions of the SORTED model. Files with this common block are

  ```
  covs, fiface, ld, nxt, printit, transpose
  ```

[22] By using the HARP truncation option, considerably greater system models can be solved. System models with 71 basic events can be solved with a truncation at level 3 by HARP on a VAX or Sun workstation. A truncation level 2 or 1 allows the solution of even larger system models (more than 71 basic events) with possible decrease in accuracy. Larger systems can be solved on larger computing platforms, up to 2^{96} states (96 basic events) with appropriate truncation. A system of this size, however, requires expanding the default state size of 10 000 to well over 60 000. (See section 2.8.)

- ROWPTR: in common block DATACB

  The size of this array equals the number of STATES+1 in a SORTED model. ROWPTR is an array of matrix row offsets, for each state in the model it tells how many transitions emanate from it. Files with this common block are

  `covs, fiface, ld, nxt, printit, transpose`

- PARMS: in common block CHRDAT

  The size of this array equals the number of TRANSITIONS in a SORTED model. PARMS contains the rate parameters of the SORTED model. Files with this common block are

  `covs, ld, nxt, printit, transpose`

- JAT: in common block TRARRY

  The size of this array equals the number of TRANSITIONS in a SORTED model. Generally, JAT is the row pointer array. Files with this common block are

  `ld, printit, transpose`

- AT: in common block TRARRY

  The size of this array equals the number of TRANSITIONS in a SORTED model. AT points to the character rate parameters for each transition in the model. Files with this common block are

  `ld, printit, transpose`

- IAT: in common block TRARRY

  The size of this array equals the number of STATES+1 in a SORTED model. Generally, IAT is the column pointer array. Files with this common block are

  `ld, printit, transpose`

- SYMBOL: in common block COMSYM

  The size of this array equals the number of STATES allowed in an UNSORTED model. SYMBOL contains the symbolic name for each state in the UNSORTED model. Files with this common block are

  `ld, nxt`

- MCPARM: in common block COMSYM

  The size of this array equals the number of TRANSITIONS allowed in an UNSORTED model. MCPARM contains the rate parameters for each transition in the UNSORTED model. Files with this common block are

  `ld, nxt`

- MCNODE: in common block DATACA

  The size of this array represents the number of TRANSITIONS allowed in an UNSORTED model. MCNODE contains the transitions of the UNSORTED model. Files with this common block are

  `ld`

In addition, in routine INITSZ of fiface.for, the following four limits must be changed. Note: It is unnecessary to change each occurrence of each variable in the file; instead, change the declaration.

- STSIZ—new number of states for sorted input
- TRSIZ—new number of transitions for sorted input
- MCSTZ—new number of states for unsorted input
- MCTRZ—new number of transitions for unsorted input

We also have the following file names and variables that must be changed:

- covs.for: NODES, ROWPTR, PARMS
- fiface.for: NODES, ROWPTR, STSIZ, TRSIZ, MCSTZ, MCTRZ
- ld.for: NODES, ROWPTR, PARMS, JAT, AT, IAT, SYMBOL, MCPARM, MCNODE
- nxt.for: NODES, ROWPTR, PARMS, SYMBOL, MCPARM
- printit.for: NODES, ROWPTR, PARMS, JAT, AT, IAT
- transpose.for: NODES, ROWPTR, PARMS, JAT, AT, IAT

5.3.3. Program *harpeng*

To increase the number of states in the HARPENG program, the user must change the following parameters and array dimensions:

- MAXST—the number of states (originally set to 10 000)
- TRANS—the number of transitions (originally set to 90 000)
  (We generally assume an average of nine transitions per state, but this number can be lower or higher.)
- MAXSYM—the number of symbols (originally set to 15 000) Estimate the number of symbols in the model. A good estimate is 100 + total number of FEHM instances (i.e., C_1, C_2, ..., C_N in the model means N FEHM instances) +3*number of coverage symbols of type VALUES + number of failure states (number of component types +2) + number of active states whose probabilities the user wants to see (only applies for unsorted Markov chain input—see section 2.7.1). The number of symbols can be reduced tremendously by telling program FIFACE that the user is not interested in near-coincident faults. If so, the coverage values is state independent, and the number of coverage symbols is reduced from (total number of FEHM instances) to (number of components with FEHM models).
- MAXFAC—the number of factors (originally set to 15 000)
- MAXTRM—the number of terms (originally set to 7500)

- DIAG: in common block MATRXN

  The size of this array equals MAXST. DIAG contains the value of each diagonal entry (which is the negative sum [23] of the outgoing arcs for the state; that is, DIAG(2,2) is the negative sum of all arcs leaving state number 2). Files with this common block are

  `bounds, eval, fill, gcall, harpeng, set, store, sym`

- ENTVAL: in common block MATRXN

  The size of this array equals TRANS. ENTVAL contains the value of each off diagonal entry of the transition rate matrix. Files with this common block are

  `bounds, eval, fill, gcall, harpeng, set, store, sym`

- SYMENT: in common block MATRXN

  The size of this array equals MAXFAC. Along with with FACTYP and NXTFAC, SYMENT is a vector of symbolic entries. It either contains a pointer to the symbol table or a constant (integer, float, or double). Files with this common block are

  `bounds, eval, fill, gcall, harpeng, set, store, sym`

- SYMVAL: in common block MATRXN

  The size of this array equals MAXSYM. SYMVAL contains the variation value of each symbol in the model. Files with this common block are

  `bounds, eval, fill, gcall, harpeng, set, store, sym`

- SYMVAR: in common block MATRXN

  The size of this array equals MAXSYM. SYMVAR contains the variation (if any) of each symbol in the model. For Weibull failure rates, SYMVAR is the vector of the alpha parameter value. Files with this common block are

  `bounds, eval, fill, gcall, harpeng, set, store, sym`

- SYMNOM: in common block MATRXN

  The size of this array equals MAXSYM. SYMNOM contains the nominal value of each symbol in the model. Files with this common block are

  `bounds, eval, fill, gcall, harpeng, set, store, sym`

- ROWPTR: in common block COMM1

  The size of this array equals MAXST+1. ROWPR contains row pointers into the sparse matrix data structure. The difference between ROWPTR(i) and ROWPTR(i+1) is the number of nonzero entries stored for the corresponding row. Files with this common block are

  `bounds, eval, fill, gcall, harpeng, set`

[23] The negative of the sum of the absolute values of the outgoing transition rates.

- COLIND: in common block COMM1

  The size of this array equals TRANS. COLIND contains pointers to the sparse matrix columns. Files with this common block are

  `bounds, eval, fill, gcall, harpeng, set`

- FACLHD: in common block COMM2

  The size of this array equals MAXTRM. Together with NXTTRM, FACLHD constitutes a term node. The term node stores pointers to symbolic expressions. FACLHD points to the head of a factor list containing the symbolic expression (see FACTYP, SYMENT, or NXTTRM). NXTTRM points to the next term in the expression (terms are separated by a plus or minus). Files with this common block are

  `eval, fill, gcall, get, harpeng, hrputil, set, store, sym`

- NXTTRM: in common block COMM2

  This size of this array equals MAXTRM. Together with FACLHD, NXTTRM constitutes a term node. The term node stores pointers to symbolic expressions. FACLHD points to the head of a factor list containing the symbolic expression (see FACTYP, SYMENT, or NXTTRM). NXTTRM points to the next term in the expression (terms are separated by a plus or minus). Files with this common block are

  `eval, fill, gcall, get, harpeng, hrputil, set, store, sym`

- FACTYP: in common block COMM2

  The size of this array equals MAXFAC. Combined with NXTFAC and SYMENT, FACTYP is a vector of symbolic entries. It contains an integer specifying the type of factor pointed to by FACLHD: 0 = Constant, 1 = x, 2 = $-x$, 3 = ‘ and 4 = ’. Files with this common block are

  `eval, fill, gcall, get, harpeng, hrputil, set, store, sym`

- NXTFAC: in common block COMM2

  The size of this array equals MAXFAC. Combined with FACTYP and SYMENT, NXTFAC is a vector of symbolic entries. It contains a pointer to the next factor in the term. Files with this common block are

  `eval, fill, gcall, get, harpeng, hrputil, set, store, sym`

- ETLHD: in common block COMM3

  The size of this array is between 2 and COLIND, that is, $2 \leq ETLHD \leq COLIND$. ETLHD determines how much of the matrix can be read in at one time. If the size of the array is small, the evaluation takes longer; however, the size of the program is smaller. If the size of the array is small enough to allow only a portion of the matrix to be read in at a time, bounds and Weibull failure processes are disallowed. The size of this array should be stored in MAXENT. Files with this common block are

  `eval, fill, gcall, get, harpeng, hrputil, set, store, sym`

- SYMNAM: in common block MATRXC

  The size of this array equals MAXSYM. SYMNAM contains the name of each symbol in the model. Files with this common block are

  `bounds, eval, fill, gcall, get, harpeng, hrputil, sym`

- SYMFN: in common block MATRXC

  The size of this array equals MAXSYM. SYMFN contains the filename of each symbol in the model, if the symbol is a coverage factor. Files with this common block are

  `bounds, eval, fill, gcall, get, harpeng, hrputil, sym`

- STDEF

  The dimension of STDEF should be changed to MAXST in bounds .for, set.for, and gcall.for

- WORK

  The dimension of WORK should be changed to (8*MAXST+3) in gerk.for and gcall.for

Also, in routine INITMX of hrputil.for, the following five limits must be changed:

- MAXST—new number of states
- MAXTRM—new number of terms
- MAXFAC—new number of factors
- MAXSYM—new number of symbols
- MAXENT—new number of size of ETLHD array

We also have the following file names and variables that must be changed. Note: It is unnecessary to change each occurrence of each variable in the file; instead, change the declaration.

- bounds.for: DIAGS, ENTVAL, SYMENT, SYMVAL, SYMVAR, SYMNOM, ROWPTR, COLIND, SYMNAM, SYMFN, STDEF
- eval.for: DIAGS, ENTVAL, SYMENT, SYMVAL, SYMVAR, SYMNOM, ROWPTR, COLIND, FACLHD, NXTTRM, FACTYP, NXTFAC, ETLHD, SYMNAM, SYMFN
- fill.for: DIAGS, ENTVAL, SYMENT, SYMVAL, SYMVAR, SYMNOM, ROWPTR, COLIND, FACLHD, NXTTRM, FACTYP, NXTFAC, ETLHD, SYMNAM, SYMFN
- gcall.for: DIAGS, ENTVAL, SYMENT, SYMVAL, SYMVAR, SYMNOM, ROWPTR, COLIND, SYMNAM, SYMFN, STDEF, WORK
- gerk.for: WORK
- get.for: FACLHD, NXTTRM, FACTYP, NXTFAC, ETLHD, SYMNAM, SYMFN
- harpeng.for: DIAGS, ENTVAL, SYMENT, SYMVAL, SYMVAR, SYMNOM, ROWPTR, COLIND, SYMNAM, SYMFN, WORK
- hrputil.for: FACLHD, NXTTRM, FACTYP, NXTFAC, ETLHD, SYMNAM, SYMFN, MAXST, MAXTRM, MAXFAC, MAXSYM, MAXENT

- set.for: DIAGS, ENTVAL, SYMENT, SYMVAL, SYMVAR, SYMNOM, ROWPTR, COLIND, FACLHD, NXTTRM, FACTYP, NXTFAC, ETLHD, STDEF
- store.for: DIAGS, ENTVAL, SYMENT, SYMVAL, SYMVAR, SYMNOM, FACLHD, NXTTRM, FACTYP, NXTFAC, ETLHD
- sym.for: DIAGS, ENTVAL, SYMENT, SYMVAL, SYMVAR, SYMNOM, FACLHD, NXTTRM, FACTYP, NXTFAC, ETLHD, SYMNAM, SYMFN

If the symbolic transition rate matrix fits in these data structures, then bounds and Weibull failure processes are allowed by the program. If the matrix does not fit, then the portion of the matrix that is read in at each step is evaluated and the space is reused. Once the entire matrix is evaluated, the unreliability (or unavailability) is computed.

5.4. CFEHM—An Editor for FEHM Models

The stand-alone program CFEHM allows the user to create new FEHM files or change parameter values in existing ones. A user creating a new FEHM file is stepped through the input as in *tdrive*, with the same models available, as described in section 4.2. In addition to creating the file, the FEHM is solved immediately and the exit probabilities are displayed on the terminal. CFEHM allows the user to change any existing FEHM files and supports change of single parameters, adding phases as in the ARIES Transient Recovery Model or changing distributions as in the ESPN model. The FEHM file is displayed line by line followed by the (Y/N) option, where *yes* means that the user wants to change a value and *no* means that the user wants to retain the old value. If the change option is chosen, the user is prompted for the new input and subsequent dependent values.

5.5. Solving Large Models

This version of HARP is configured for a problem that requires up to 10 000 states.[24] A problem of this size can be solved easily on a DEC VAX 750 computer, which is the machine on which HARP was developed. The limiting factor on the size of the problem is the amount of storage required to store the symbolic transition rate matrix. On a DEC VAX 8600 computer, we have solved problems as large as 25 000 states (one user reported success in solving a system with 45 000 states on a DEC VAX 11-785). This section discusses some methods that the user can utilize to solve very large problems.

If the symbolic matrix is too large to store in the data structures internal to HARP (but there are still fewer than 10 000 states), the portion of the matrix that has been stored is evaluated, and the space is reused. This method allows the user to solve larger problems (without increasing the data storage requirements) but disallows any calculations that require reevaluation of the symbolic matrix (bounds, Weibull failure rates). If the problem is still too large for HARP to solve, for example if there are too many symbols, the user can ignore the consideration of near-coincident faults. This action significantly reduces the number of distinct symbols in the model because the coverage factors are now state independent.

To solve models that are larger than 10 000 states, see the section 4.3, which discusses how to change the limits on the various variables used in HARP.

5.6. System Resources

When HARP is executed on a DEC VAX 11-700 series computer, the following resources are required for the default 10 000 state limits:

[24] The configured state size of 10 000 is the actual number of state occupancy probabilities that are computed. By using the truncation technique of chapters 1 and 4, systems with much larger states can be solved. The maximum system size is 2^{96} equivalent Markovian states; however, the computational resources required to use this size model may be unavailable.

- 4096 physical pages (2 MB of real memory)
- 40 000 virtual pages (20 MB virtual address space)

Under a UNIX environment, the same bytes of real memory and virtual address space are required. The system parameters may also need to be changed to allow a single user a large working set size. These parameters include the working set maximum and the virtual page count. For execution under MS DOS, see section 1.2.4.

Chapter 6

Dynamic Fault Tree Gates

6.1. Modeling Nonrepairable Systems of Arbitrary Complexity With Fault Trees

The dynamic dependency gates for fault tree modeling are not traditional fault tree gates and will be unfamiliar to most HARP users. This chapter familiarizes the user with the detailed specification and mathematical model for each dynamic dependency gate. The original research on the dynamic fault tree gate models is reported in reference 36. As with any modeling language, the modeler must properly apply these gates according to their specifications.

For the purposes of our discussion we use the following notation. We identify a particular Markov chain state by listing a set of the components that failed while producing the state in question. For example, if the system has five different components (one of each type present) and if components 2 and 5 have failed (first component 2 followed by component 5), then the system is left in a state denoted by an index $I = (2, 5)$. Because the new fault tree gates model system behavior for which the sequence in which the components fail is important, the order in which the components fail can be significant. Therefore, we note that state $(2, 5)$ generally cannot be equivalent to state $(5, 2)$.

States can also be denoted with tuples, indicating which components are still working and which have failed. We call these *component status tuples* because they denote the working status of all system components. It is not possible to denote the sequence in which the components have failed with only the component status tuple notation. For example, both state $(2, 5)$ and state $(5, 2)$ can be denoted by the tuple 10110. For this reason, if the sequence of component failures is significant, then the Markov chain state labeling method may need to be extended by appending to the component status tuple some additional information indicating the sequence in which certain events took place. The form of this additional information is determined by the structure of the fault tree. For example, an additional element may be added to the component status region of the tuple for each priority and gate (see section 6.1.3) in the fault tree to indicate which of the gate's inputs (left or right) fired first (or whether any inputs have fired at all yet). In this way, states can be denoted that are identical in terms of components working or failed but which are distinguished by the sequence in which component failures occurred, as expressed by one or more priority and gates. Yet another type of additional information (described in section 6.1.4) is appended to the component status region of the state tuple for each cold spare gate in the fault tree that shares any of its spares with one or more other cold spare gates in the fault tree.

Components of identical type that serve a redundant function in the system can be grouped together in what are called replicated basic events. The previous notation is easily extended to accommodate these: the tuple notation is nonbinary with redundant sets of components denoted by members of the tuple whose value is greater than one. Conversely, the component failure list notation can simply contain multiple occurrences of component type i in the list denoting the failure of more than one member of the redundant set of components of type i. For example, if the system containing five types of components has a group of two redundant components of type 2, the original state tuple (denoting "all components working") is 12111. If one component of type 2 fails, followed by one component of type 5 failing, followed by the other component of type 2 failing, the resulting state is denoted by either the component failure list $(2, 5, 2)$ or the state tuple 10110. If the sequence in which these components fail is significant (for example, if

state $(2,5,2)$ is distinct from state $(2,2,5)$ in the Markov chain), then additional information needs to be appended to the tuple 10110 to indicate the sequence in which the components failed. In general, the form of that additional information is determined by the structure of the fault tree, as previously described. In most cases, the component failure list notation are preferred over that of the component status tuple notation for reasons of simplicity and clarity.

We assume that component i fails at a constant rate λ_i. A state labeled Fj is entered whenever a component of type j fails and there are no spare components to take its place (exhaustion of redundancy); thus, a system crash results. We denote the probability of being in state I at time t by $P_I(t)$, and the Laplace transform for the state by $L_I(s)$. For example, the probability and Laplace transform for state $(2,5)$ are $P_{2,5}(t)$ and $L_{2,5}(s)$, respectively. The initial state (the one in which all components are operational) is denoted by the index 0 (e.g., $P_0(t)$ and $L_0(t)$).

6.1.1. Functional Dependency Gate

The functional dependency gate is the simplest of the sequence-dependent gates to define. It has an input, called the trigger input, which can be any general event (e.g., the output of any other fault tree gate or any basic event). For ease of presentation, we assume without loss of generality that the trigger event is a single unreplicated basic event. The gate also has a number of dependent events, which must be (possibly replicated) basic events. Finally, the gate has an output (which we call the nondependent output) whose value is always identical to the value of the input event (i.e., the nondependent output event occurs if and only if the input event occurs). This output is provided to simplify the display of complex fault trees where the trigger event is required as an input to another gate.

Figure 33 depicts the functional dependency gate as described here. The trigger event is event 1, the dependent events are events 3 through n, which can be replicated (i.e., groups of m_i redundant components of type i), and the output event is denoted by the outgoing arc at the top of the gate. Figure 34 depicts the Markov model that defines the behavior of the functional dependency gate shown in figure 33, where the states have been labeled with failed component lists. Figure 35 depicts the same Markov model with the states labeled with component status tuples. Figure 35 gives a better indication of the component failures than figure 34. The Markov model contains two states: the initial state depicting the situation before the trigger event occurs and the final state representing the situation after the trigger event occurs.

Figure 33. Functional dependency gate.

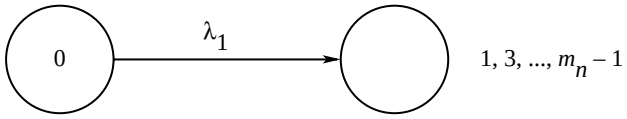


Figure 34. Markov model defining action of functional dependency gate with failed component lists.

Figure 35. Markov model defining action of functional dependency gate with component status tuples.

In figure 35, the initial state tuple indicates that neither the trigger event nor any of the dependent events (interpreted here as component failures) has occurred by showing that the trigger component and all dependent components are still operational (all tuple members are still greater than 0). The final state indicates that the trigger event has occurred (trigger component has failed), and the action of the gate causes all dependent events to occur as well (all dependent components are forced to fail). Thus, all members of the tuple correspond to dependent events becoming zero (note that the tuple member corresponding to component 2 is still nonzero because component 2 was not dependent on the trigger component). The rate at which this occurs is the rate of occurrence of the trigger event λ_1. The output event of the functional dependency gate is defined to be equal to the trigger event. The Chapman-Kolmogorov equations and the single-sided Laplace transform equations are given for state 0 in equations (1) and for state 1 in equations (2) as follows:

$$\left.\begin{aligned} \frac{dP_0(t)}{dt} &= -\lambda_1 P_0(t) \\ L_0(s) &= \frac{1}{s+\lambda_1} \end{aligned}\right\} \tag{1}$$

$$\left.\begin{aligned} \frac{dP_1(t)}{dt} &= \lambda_1 P_0(t) \\ sL_1(s) &= \lambda_1 L_0(s) \\ L_1(s) &= \frac{1}{s}\lambda_1 \frac{1}{s+\lambda_1} \\ &= \lambda_1\left(\frac{1/\lambda_1}{s} + \frac{-1/\lambda_1}{s+\lambda_1}\right) \\ &= \frac{1}{s} - \frac{1}{s+\lambda_1} \end{aligned}\right\} \tag{2}$$

The probability of the output event of the gate is as follows:

$$P_1(t) = 1 - e^{-\lambda_1 t} \tag{3}$$

Although this example is simple enough to be solved by inspection, we have used a three-step analysis to obtain a mathematical expression for the gate's output event (and hence a definition of the action of the gate) to illustrate the general procedure used to analyze all these gates.

We first identify a minimal Markov model that defines the action of the gate. We then use the Laplace transform equation for the output event of the gate. We finally obtain an expression for the probability of the output event by inverting the Laplace transform.

6.1.2. Sequence-Enforcing Gate

We next consider the sequence-enforcing gate. This gate can have any number of inputs. The leftmost input can be any general event (e.g., the output of any other fault tree gate or any basic event). Again, for ease of presentation, we assume without loss of generality that this leftmost event is a single unreplicated basic event. All other inputs to the gate must be (possibly replicated) basic events. The gate has an output which is on when all gate inputs are on (i.e., have occurred). We note that when an input leads to a descendent node that is a replicated basic event, the input event is not considered to occur until all redundant components of the replicated basic event have failed. Figure 36 shows a sequence-enforcing gate for which the inputs all lead to unreplicated basic events (representing components 1 through n). The sequence-enforcing gate constrains the occurrence of its input events to follow the left-to-right order in which they appear as inputs to the gate. For example, event 2 is not permitted to occur before event 1. Similarly, event $i+1$ is not permitted to occur before event i. This process is accomplished by not including in the Markov model any states for which event $i+1$ has occurred and event i has not. The resulting Markov model that corresponds to figure 36 is shown in figure 37.

Figure 36. Sequence-enforcing gate with unreplicated basic events.

Figure 37. Markov model defining action of n-input sequence-enforcing gate for output event of gate.

Using the Chapman-Kolmogorov equations, we can derive the Laplace transform equation as follows:

$$L_0(s) = \frac{1}{s+\lambda_1} \tag{4}$$

$$\left.\begin{aligned} \frac{dP_1(t)}{dt} &= \lambda_1 P_0(t) - \lambda_2 P_1(t) \\ L_1(s) &= \frac{1}{s+\lambda_2}\lambda_1 L_0(s) \\ &= \lambda_1 \frac{1}{s+\lambda_2}\frac{1}{s+\lambda_1} \end{aligned}\right\} \tag{5}$$

$$\left.\begin{aligned} \frac{dP_{1,\ldots,k}(t)}{dt} &= \lambda_k P_{1,\ldots,k-1}(t) - \lambda_{k+1} P_{1,\ldots,k}(t) \\ L_{1,\ldots,k}(s) &= \frac{1}{s+\lambda_{k+1}}\lambda_k L_{1,\ldots,k-1}(s) \\ &= \frac{1}{s+\lambda_{k+1}}\lambda_k \frac{1}{s+\lambda_k}\cdots\frac{1}{s+\lambda_1} \end{aligned}\right\} \tag{6}$$

$$\left.\begin{aligned} \frac{dP_{F_n}(t)}{dt} &= \lambda_n P_{1,\ldots,n-1}(t) \\ sL_{F_n}(s) &= \lambda_n L_{1,\ldots,n-1}(s) \\ L_{F_n}(s) &= \frac{1}{s}\prod_{i=1}^{n}\frac{\lambda_i}{s+\lambda_i} \end{aligned}\right\} \tag{7}$$

Equation (7) is the Laplace transform equation for the output event of the sequence-enforcing gate. Using the following partial fraction expansion method from reference 44:

$$\frac{N(s)}{\prod_{i=1}^{n}(s+a_i)} = \sum_{i=1}^{n}\frac{C_i}{s+a_i}$$

where

$$C_i = \frac{N(s)}{\prod_{i=1}^{n}(s+a_i)}(s+a_i)|_{s=-a_i}$$

and taking $\lambda_0 = 0$, we can obtain a form of equation (7) that is easily invertible:

$$L_{F_n}(s) = \prod_{i=1}^{n}\lambda_i \sum_{k=0}^{n}\frac{1/\prod_{j=0,j=k}(\lambda_k - \lambda_j)}{s+\lambda_k} \tag{8}$$

Inverting equation (8) gives the probability of the output event of the sequence-enforcing gate:

$$P_{F_n}(t) = \prod_{i=1}^{n}\lambda_i \sum_{k=0}^{n}\frac{e^{-\lambda_k t}}{\prod_{j=0,j=k}(\lambda_k - \lambda_j)} \tag{9}$$

These figures and equations are easily modified to accommodate inputs leading to replicated basic events. Figure 38 depicts such a sequence-enforcing gate. The corresponding Markov chain is similar to that depicted in figure 37 except that m_1 failures of the components of type 1 occur first followed by m_2 failures of the components of type 2, and so on until the m_n components of type n are the last to fail. A straight-forward application of the Chapman-Kolmogorov equations

shows that the Laplace transform for the gate output when the gate inputs lead to replicated basic events generalizes from equation (7) to the following:

$$L_{F_n}(s) = \frac{1}{s} \prod_{i=1}^{n} \left[\prod_{j=0}^{m_i - 1} \frac{(m_i - j)\lambda_i}{s + (m_i - j)\lambda_i} \right] \tag{10}$$

from which an expression for the probability $P_{F_n}(t)$ can be obtained by using the same partial fraction expansion procedure that was used before.

Figure 38. An n-input sequence-enforcing gate with replicated basic events.

6.1.3. Priority And Gate

The sequence-enforcing gate described in the previous section forces component failures to follow a prescribed sequence in the Markov model by disallowing inclusion of any states in the model for which component failures have occurred in other than the prescribed sequence. In contrast, the priority and gate allows these states from the Markov model. However, if components fail in other than the prescribed sequence for a particular gate, the gate never fires (i.e., the output event of the gate never occurs). The output event only occurs if all input events of the gate occur in the left-to-right sequence in which they appear as inputs to the gate. Our implementation requires that each priority and gate has a maximum of two inputs. However, two or more priority and gates can be cascaded together to achieve the effect of a multiinput gate. Therefore, we assume a multiinput priority and gate (an unlimited number of inputs) for the purpose of our analysis. The inputs of the priority and gate can be any general event (e.g., the output of any other fault tree gate or any basic event). As previously noted, when an input leads to a replicated basic event, that input is not considered to be on (i.e., the event occurred) until all redundant components in the replicated basic event have failed. Again, for ease of presentation, we assume without loss of generality that the inputs to the priority and gate lead to unreplicated basic events. Figure 39 depicts the Markov model that defines the action of a multi-input priority and gate for which the leftmost input is event 1, the next leftmost is event 2, up to the rightmost event, which is event n. The Laplace transform equation for the output event of the gate can be obtained by again using the Chapman-Kolmogorov equations as follows:

$$L_0(s) = \frac{1}{s + \sum_{i=1}^{n} \lambda_i} \tag{11}$$

$$\left.\begin{aligned} \frac{dP_1(t)}{dt} &= \lambda_1 P_0(t) - \sum_{k=1} \lambda_k P_1(t) \\ L_1(s) &= \frac{1}{s + \sum_{k=1} \lambda_k} \lambda_1 L_0(s) \\ &= \lambda_1 \frac{1}{s + \sum_{k=1} \lambda_k} \frac{1}{s + \sum_{i=1}^{n} \lambda_i} \end{aligned}\right\} \tag{12}$$

$$\left.\begin{aligned} \frac{dP_{1,\ldots,k}(t)}{dt} &= \lambda_k P_{1,\ldots,k-1}(t) - \left(\sum_{j\in(1,\ldots,k)} \lambda_j\right) P_{1,\ldots,k}(t) \\ L_{1,\ldots,k}(s) &= \frac{1}{s + \sum_{j\in(1,\ldots,k)} \lambda_j} \lambda_k L_{1,\ldots,k-1}(s) \\ &= \frac{1}{s + \sum_{j\in(1,\ldots,k)} \lambda_j} \lambda_k \frac{1}{s + \sum_{j\in(1,\ldots,k-1)} \lambda_j} \ldots \lambda_1 \frac{1}{s + \sum_{i=1}^{n} \lambda_i} \end{aligned}\right\} \tag{13}$$

$$\left.\begin{aligned} \frac{dP_{F_n}(t)}{dt} &= \lambda_n P_{1,\ldots,n-1}(t) \\ sL_{F_n}(s) &= \lambda_n L_{1,\ldots,n-1}(s) \\ L_{F_n}(s) &= \frac{1}{s} \lambda_n \frac{1}{s + \sum_{j\in(1,\ldots,n-1)} \lambda_j} \ldots \lambda_1 \frac{1}{s + \sum_{i=1}^{n} \lambda_i} \\ &= \frac{1}{s} \prod_{i=1}^{n} \frac{\lambda_i}{s + \sum_{k\in(1,\ldots,i-1)} \lambda_k} \\ &= \prod_{i=1}^{n} \lambda_i \prod_{i=0}^{n} \frac{1}{s + \sum_{k\in(1,\ldots,i)} \lambda_k} \end{aligned}\right\} \tag{14}$$

Taking $a_0 = 0$ and $a_i \sum_{j\in(1,\ldots,i)} \lambda_j$, we obtain a form of equation (14) that is easily invertible:

$$L_{F_n}(s) = \prod_{i=1}^{n} \lambda_i \sum_{k=0}^{n} \frac{1/\prod_{j=0, j=k} (a_k - a_j)}{s + a_k} \tag{15}$$

Inverting equation (15) gives the probability of the output even of the priority and gate:

$$P_{F_m}(t) = \prod_{i=0}^{n} \lambda_i \sum_{i=0}^{n} \frac{e^{-\lambda_k t}}{\prod_{j=0, j=k} (a_k - a_j)} \tag{16}$$

We note here that equations (15) and (16) are identical to the expressions derived by Fussell (ref. 54) except that our method of numbering the input events of the gate is different.

6.1.4. Cold Spare Gate

The cold spare gate is the most complex gate in the set of sequence dependency gates. All inputs to the cold spare gate must be (possibly replicated) basic events. The leftmost input

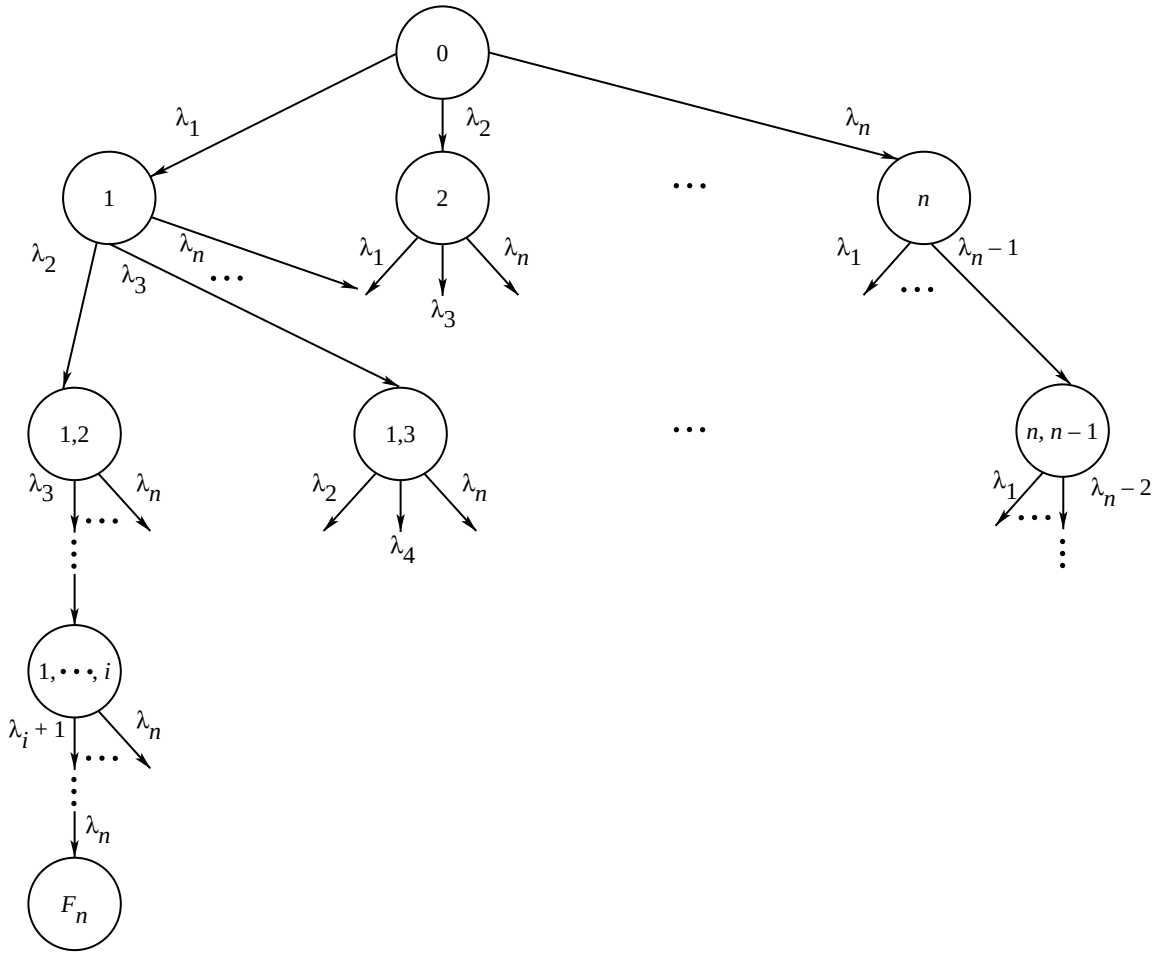


Figure 39. Markov model defining action of n-input priority and gate.

represents one or more primary units that are initially on-line. All inputs to the right of the leftmost input represent alternate (spare) units or groups of redundant units available as pooled spares that are initially powered down (i.e., that are cold spares). Upon the failure of any of the units that are active, replacements are selected from the set of spare units that have not yet been placed on-line. The spare units must be switched into operation in the left-to-right sequence in which they appear as inputs to the cold spare gate.

For example, all spare units from the second leftmost input must be activated as spares for failed components before any units from the third leftmost input can be activated. So long as at least m_1 nonfailed components are present in the set of all components that are inputs to the cold spare gate (whether on-line or powered down), m_1 active (i.e., on-line) components are always being "used" by the cold spare gate. Once enough failures occur so that all remaining components are active (i.e., no spares remain that are powered down; they are all on-line replacing components that have previously failed), then the number of components being used by the cold spare gate is the number of the spare components that have not yet failed. This number decreases from m_1 down to 1 as subsequent failures of the spare components occur. Only when all components from all inputs to the cold spare gate have failed does the output of the cold spare gate turn on (i.e., the output event occurs). Figure 40 depicts the general form of the cold spare gate described here.

Figure 40. An n-input cold spare gate with replicated basic events.

6.1.4.1. CSP Gate Behavior Without Intergate Interactions

When none of the inputs to a cold spare gate are shared with any other gate in the fault tree, the operation of this general form of the cold spare gate is defined by the Markov chain shown in figure 41. The probability of the output event of the gate is obtained by solving the Markov chain for and summing the probabilities of being in state F_i. This computation can be achieved with the analysis procedure (used in the previous examples) of deriving the Laplace transform equation for each F_i state, performing a partial fraction expansion, and then inverting the resulting expression to obtain the probability expression (in the time domain) of the state F_i.

The action of the cold spare gate is often more clearly illustrated when the Markov chain is labeled with component status tuple notation rather than the list of component failures notation. As mentioned in section 6.1, a component status tuple indicates how many of each type of component in the system is still nonfailed. The term nonfailed refers to both active (on-line) and cold (off-line) components. Additional information can be added to the component status tuple of each state to further clarify the action of the cold spare gate. That additional information has the form of a second tuple containing one element for each input of the cold spare gate (for fault trees that have several cold spare gates, one such auxiliary tuple can be added for each cold spare gate in the fault tree). The value of each element indicates how many units are currently being used (i.e., on-line and active) for each input of the cold spare gate. The component status tuple is separated from the in-use descriptor tuple by a double bar. For example, a state labeled $m_1 m_2 \cdots m_n \| u_1 u_2 \cdots u_n$ is one in which m_1 components of type 1 are nonfailed (i.e., either operating on-line or powered down and awaiting activation), m_2 components of type 2 are nonfailed, u_1 primary units (of component type 1) are active and on-line (i.e., "in use"), and u_2 units of type 2 are on-line, etc. Note that $u_i \leq m_i$ for $1 \leq i \leq n$ at all times. The initial "all components working" state of a Markov chain for the cold spare gate in figure A8 always has the form $m_1 m_2 \cdots m_n \| m_1 0 \cdots 0$. This form indicates that the m_1 primary units are initially all on-line and working correctly and all spare units used by the cold spare gate are initially powered down and therefore not yet in use by the gate. In general, a state is vulnerable to failures of components under a cold spare gate in accordance with nonzero values for the in-use tuple (i.e., the second tuple denoted by the $u_1 u_2 \cdots u_n$) of the gate because this tuple is the one that indicates which components are active for the cold spare gate (and hence eligible to fail).

Figure 42 shows the transitions that can be experienced by a general state in the Markov chain for the cold spare gate of figure 40 when the component status tuple notation is used.

Figure 41. Markov chain defining action of n-input cold spare gate with replicated basic events.

(We note that $u_i \leq w_i$ and $w_i \leq m_i$ for $1 \leq i \leq n$, where w_i denotes the number of components of type i that are still working (i.e., nonfailed).) The incoming transitions come from upstream states where for each transition at least one of the u_i primary components has failed.

As an example, the action of the cold spare gate shown in figure 43 is defined by the Markov chain shown in figure 44. Each state is labeled with a two-part state tuple. The first part of the state tuple is the component status tuple. A double bar separates the first part of the tuple from the second part. The second part of the state tuple is the in-use descriptor tuple for the cold spare gate and contains the number of components of each type that are currently being used by the cold spare gate. That is, the components being used are on-line and performing the functions of the primary units of the cold spare gate that were initially on-line. The second part of the tuple shows exactly which spares are currently active (replacing failed components).

Figure 42. General state from Markov chain for cold spare gate with replicated basic events.

Figure 43. Example cold spare gate with replicated basic events.

Initially, the system begins with all three primary units (all of component type 1) operating correctly. The two spare components of type 2 and the one spare component of type 3 are off-line. This status is indicated by the value 300 in the second part of the state tuple of the initial state in the Markov chain. Eventually, one of the three primary units fails, and one of the spare components of type 2 is switched on-line to take the place of the failed component. The system moves to the state labeled 221||210. In this case, the second part of the state tuple indicates that two components of type 1 and one spare component of type 2 are in use by the gate. The other spare component of type 2 and the spare component of type 3 are still off-line (powered down) and hence not in use by the gate. Consequently, these two components cannot fail yet because it is assumed that powered down components do not fail. Once the spare component of type 2 that is selected to replace the failed component is activated, it can fail at any time after it is placed on-line. Therefore, the next failure the system experiences can be either one of the two remaining components of type 1 (in which case the system goes to the state labeled 121||120) or the active component of type 2 (leading the system to go to the state labeled 211||210).

When one of the components of type 1 fails, the remaining component of type 2 is activated to replace the second failed component of type 1, and the system is left operating with one component of type 1 and two components of type 2 (as indicated by the second part of the

Figure 44. Markov chain defining action of cold spare gate with replicated basic events.

state tuple, 120). When the active component of type 2 fails, the second component of type 2 is activated to replace the first component of type 2 (which just failed), and the system is left operating with two components of type 1 and one component of type 2 active. This status is indicated by the second part of the state tuple, 210. Note that the second part of the state tuple for this state (211||210) has not changed from the second part of the tuple for the previous state (221||210) even though the system has one less component of type 2 in working order. The second part of the state tuple records only the number of each type of component that is in use by the cold spare gate, and from the previous state to the current state the number of components of each type that are in use has not changed. One component of type 2 has failed and been replaced by the other component of type 2; thus, the count of active components are unchanged.

While in state 211||210 the system can experience either a failure of one of the two components of type 1 or a failure of the component of type 2. Either kind of failure results in the last spare (of component type 3) being activated and placed on-line. If one of the components of type 1 fails, then the system goes to state 111||111 in which one of each component type is operating on-line. If the component of type 2 fails, the system goes to state 201||201 in which two components of type 1 and the component of type 3 are all operating on-line. Note that all or some components of type 2 can fail before all components of type 1 fail, and components of type 3 can be activated before all components of type 1 fail. The cold spare gate with replicated basic events differs from the sequence-enforcing gate with replicated events in this respect. The fact that the cold spare gate enforces the sequence of component activation rather than failure accounts for this difference. The cold spare gate prevents any components that have not been activated from failing. Once activated, however, components can fail at any time. Consequently; once components A and B are both active, component B can fail before component A even though component A may have been activated before component B. By contrast, the sequence-enforcing gate enforces the sequence of allowed component *failures* (or, more generally, the sequence of event *occurrences*); thus, all components of type 1 must fail before any components of type 2 or type 3 can fail.

The remainder of the Markov chain in figure 44 can be similarly interpreted. Note that the sum of all components in use by the cold spare gate (as indicated by the second part of the state tuple) always equals the original number of primary units (in this case, three) until fewer than that number of components remain that have not yet failed. Then, all remaining spare components have been activated and placed on-line, and subsequent failures are not replaced by spares (there are no spares left). Thus, degraded system performance results. The output of the cold spare gate turns on only when one of the states labeled F_1, F_2, or F_3 is reached.

The system modeled by the fault tree in figure 43 remains operating as long as is at least one component among the primary and spare units is working. However, sometimes the system to be modeled has a critical minimum component count of components that must be operating for the system to remain functional. For example, suppose the system shown in figure 43 can only remain operational if at least three components from among the primary and spare units remained operational. This system can be modeled easily by adding an *M-out-of-N* gate (where M is one greater than the critical minimum number of components that need to be operational, and N is the total number of components that are inputs to the cold spare gate) to the cold spare gate. All inputs to the cold spare gate must also be inputs to the *M-out-of-N* gate. The outputs of the *M-out-of-N* gate and the cold spare gate should then become inputs to an or gate. Figure 45 shows the system for which at least three components from among the primary and spare units must be operational in order for the system to be operational. Figure 46 shows the resulting Markov chain.

6.1.4.2. CSP Gate Behavior With Spares Shared Between CSP Gates

Unlike the sequence-enforcing gate, the cold spare gate can interact with other gates of the fault tree in two special ways. The first occurs when two or more cold spare gates share an alternate (spare) unit (or group of pooled spare units). Figure 47 depicts a situation where $n-1$ system components share a spare unit between them. In the figure, the first cold spare gate to have its primary unit fail does not "fire" (i.e., the output event does not occur) until the spare unit subsequently fails. In the meantime, any subsequent failure primary unit of any other cold spare gate (before the spare unit fails) causes that cold spare gate to fire immediately because the shared spare is no longer available to replace failing primary units (it has already been used to replace the first failed primary unit). Figure 48 shows the equivalent Markov model that defines this interaction between cold spare gates sharing spare units. The probability of

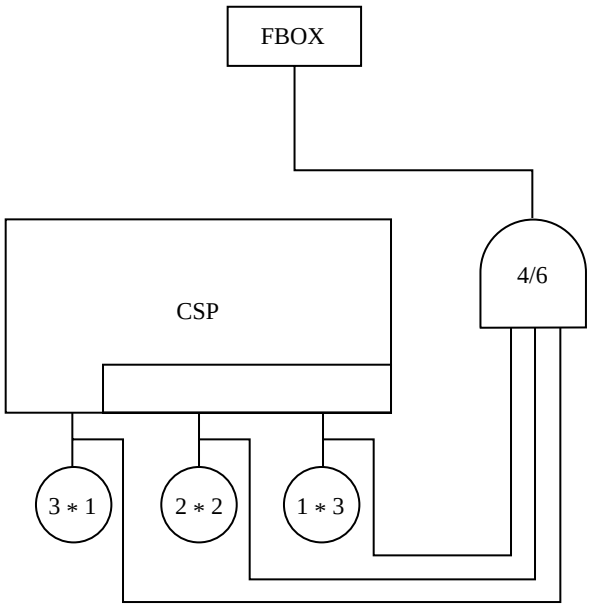


Figure 45. Cold spare gate with critical minimum complement.

Figure 46. Markov chain defining action of cold spare gate with critical minimum complement.

the top event of the fault tree of figure 47 is obtained by solving the Markov model in figure 48 for the probability of each of the states labeled F_i and then summing the probabilities of those

Figure 47. Fault tree with $n-1$ cold spare gates sharing a spare.

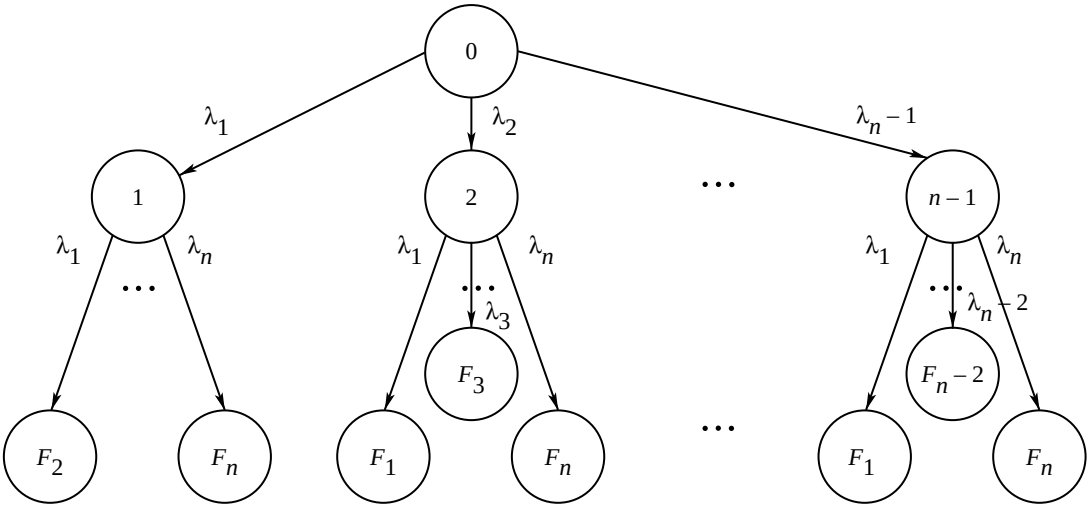


Figure 48. Markov model defining action of $n-1$ spare-sharing cold spare gates.

states. This computation can be accomplished with the analysis method used in the preceding sections to analyze the Markov models for the other gates.

An additional consideration arises when a replicated basic event representing a group of pooled spares is shared between two or more cold spare gates. The order in which failures occur among activated members of such a group of pooled spares can be significant and must be accounted for in the Markov chain. To illustrate this point, consider the fault tree in figure 49. The Markov chain for this system is shown in figure 50. The first part of the label of each state

Figure 49. Fault tree with cold spare gates sharing pooled spares.

is the component status tuple. The next part, separated by a double bar from the first part, is an in-use tuple denoting the components that are in use by the leftmost cold spare gate in figure 49. The last part of the state label (again separated by a double bar from the previous part) is an in-use tuple denoting the components that are in use by the rightmost cold spare gate in figure 49. In this example, two spare units of type 3 are pooled together and shared by the two cold spare gates in the fault tree. If component 1 fails first, one of the components of type 3 is activated to replace it. If component 2 fails next, the other component of type 3 is activated to replace it. If a component of type 3 (one of the units in the replicated basic event labeled $2*3$) fails next, the order in which the two components of type 3 fail is now significant. If the component selected to replace the failed component of type 1 fails first, the system behavior can be different than if the component selected to replace the failed component of type 2 fails first. The order of failures is illustrated in figure 50 in the transitions from state 00211||010||010 to states 00111||001||010 and 00111||010||001. The set of descendent states of state 00111||001||010 (and hence the behavior of the system once it has reached state 00111||001||010) differs from the set of descendent states of state 00111||010||001. This difference in system behavior is accounted for by the fact that the process of activating and replacing failed components has transformed the two components of type 3 from being functionally equivalent units (which they were when they were off-line pooled spares that were powered down) to being functionally distinct units in the system. This transformation need not always occur. For example, suppose that in figure 49 two components of type 1 instead of one were attached to the leftmost input of the leftmost cold spare gate. Further, suppose that both of these components of type 1 have failed and that both of the components of type 3 were activated to replace them. If the next failure is one of the components of type 3, then which one of the two actually fails first is not significant. The

Figure 50. Markov chain for fault tree with cold spare gates sharing pooled spares.

subsequent behavior of the system is the same no matter which component of type 3 fails. On the other hand, suppose only one of the components of type 1 fails and is replaced by one of the components of type 3, and then the component of type 2 fails and is replaced by the other component of type 3; then suppose the next failure is a component of type 3. In this case, the system behavior may depend on which of the two components of type 3 fails first.

This type of sequence dependency can be subtle and difficult to track in the Markov chain for a fault tree that has cold spare gates that share pooled spares. However, the use of the auxiliary in-use tuples previously described provides a satisfactory way of accounting for these sequence dependencies in the Markov chain. This state labeling method, which is perhaps not the most efficient method of recording sequence dependencies in Markov chain states, makes it comparatively easy for human modeling engineers to understand what is going on in the Markov chain (and consequently the fault tree) model of the system.

6.1.4.3. CSP Gate Behavior for Spare Shared With Functional Dependency Gate

The second special intergate interaction of a cold spare gate occurs when a spare unit for a cold spare gate is also a dependent event for a functional dependency gate. Normally, the cold spare gate disallows the spare from failing before the primary unit fails. However, if the spare unit is disabled as a result of the failure of some other component through the action of a functional dependency gate, then the failure of the spare is permitted even though the primary unit may still be operational. This apparent exception to the cold spare gate definition is needed because the disabling of a spare unit (modeled with the functional dependency gate in this way) is expressed by considering the spare unit as functionally failed, even though the unit may not actually have failed. Figure 51 shows the simplest fault tree that models this situation, and figure 52 shows the equivalent Markov model that defines this interaction between the functional dependency gate and the cold spare gate. The normal action of the cold spare gate would have prevented state 1,3 (and any of its descendent states) from being generated. The effect is that component 3 is prevented from failing before component 2 fails, and component 1 is prevented from failing before component 2. However, because component 3 (the spare of the cold spare gate) is a dependent event of the functional dependency gate, state 1,3 is generated and included in the Markov chain along with its descendent states. The probability of the top event of the fault tree of figure 51 is obtained by solving the Markov model in figure 52 for the probability of each of the states labeled F_i and then summing the probabilities of those states.

Figure 51. Fault tree interaction between functional dependency gate and cold spare gate.

6.2. Fault-Tree-to-Markov-Chain Conversion Algorithm

An arbitrary fault tree can be converted into an equivalent Markov chain with a fault-tree-to-Markov-chain conversion algorithm. The original version of this algorithm is described in detail in reference 49. This original algorithm has been expanded to allow the addition of sequence-dependency gates to the standard set of traditional fault tree gates. A sketch of the updated algorithm is as follows:

Figure 52. Markov model interaction between functional dependency gate and cold spare gate.

```
Algorithm ft2mc
    input component failure rates;
    input and build internal representation for fault tree;
    determine number of basic even nodes;
    determine system ‘‘initial operational state’’;
    open output file;
    initialize state queue and state table;
    place ‘‘initial operational state’’ onto the state queue;
    while (state queue not empty) do
    {
        remove next originating state from queue;
        for each component i in the state tuple do
        {
            simulate a failure of one of component i;
            evaluate the effect on the resulting state of
             Functional Dependency gates;
            evaluate the effect on the resulting state of
             Cold Spare gates;
            evaluate the effect on the resulting state of
             Priority-AND gates;
            evaluate the effect on the resulting state of
             Sequence Enforcing gates;
            look up resulting state in the state table;
```

```
            if (resulting state is new, i.e. not in
                                        state table) then
            {
                if (not system failure) then
                    add resulting state to queue;
                record resulting state in state table;
            }
            output arc from originating state to resulting state;
            undo simulated failure of one of component i;
        } /* End of for loop */
    } /* End of while loop*/
     close output file;
end ft2mc.
```

This algorithm has been modified somewhat to allow lumping together of multiple transitions from an originating state to a single resulting state. However, this algorithm is the simplest conceptual expressure of the fault-tree-to-Markov-chain conversion algorithm and suffices for the purpose of describing our work here.

Chapter 7

Advanced Modeling Techniques

This chapter informs the advanced user of some important modeling features of HARP. The first section illustrates a technique for specifying FEHM's directly from a fault tree instead of the typical FEHM specification. The extent to which the FEHM specification can be implemented in the fault tree notation has not been explored.

The second section addresses the application of HARP's multifault models to a specific class of system architectures that use nearly independent fault containment regions. A study has shown that as the number of fault containment regions increase, HARP's multifault ALL model produces an increasing greater conservative result.

7.1. State-Dependent FEHM's in Fault Trees

Figure 53 shows the use of the sequence-enforcing gate to force state-dependent FEHM insertion in a fault tree. The FORM without FEHM's for the model in the figure is an or gate with the same top event (FBOX) and basic events 3*1 and 3*2. This notation specifies three units of type 1, shown as P (e.g., a processor) in the figure and three units of type 2, shown as Q (e.g., buses). By splitting the number of P and Q units and renaming the groups of two units as $2P1$ and $2Q2$ (1*1, 1*2, 2*3, 2*4 in HARP notation), different FEHM's can be assigned to the units. In this example, no FEHM's are assigned to P and Q, but the same or different FEHM's can be assigned to the $2P1$ and $2Q1$ units, respectively. The sequence-enforcing gates enable this capability by precluding the $2P1$ and $2Q1$ units from failing until the P or Q unit fails. The modeling effect is that because no FEHM's were specified for the first P or Q failure, a unity fault/error recovery probability (coverage) is modeled for the first of these failures. Subsequent failures have the specified FEHM's inserted into the resulting Markov chain as usual.

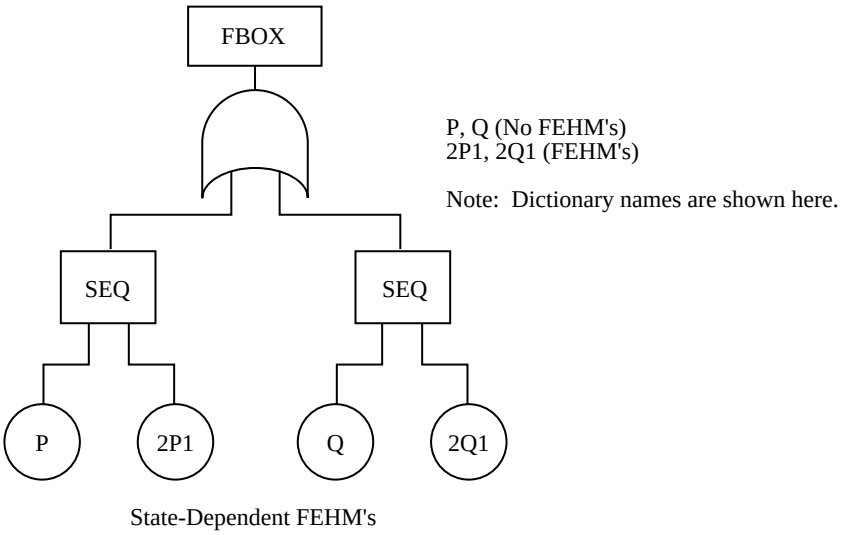


Figure 53. State-dependent FEHM's fault tree. (This figure is identical to figure 32.)

7.2. Approximating Multifault Models

As discussed in section 2.7, HARP offers three simplified models, ALL-inclusive, SAME-type, and USER-defined, for automatic multifault model generation when invoking behavioral

decomposition. The penalty for utilizing a simplified multifault model when all failure modes are captured is an increase in the conservatism of the unreliability predictions. The degree of conservatism is a function of the disparity of the FORM and FEHM sojourn times. The advantage of using a simplified multifault model is a considerable reduction in computation and user effort to define the input for the detailed model. For most systems the simplified models provide acceptable results (refs. 27, 32, and 34). When this is not the case, the user can modify the HARP generated ASCII files to get an accurate model with behavioral decomposition. Using the AS IS model and using the X Window System (XHARP) are alternative ways to obtain more accurate results (ref. 5).

Figure 54 is an example of the use of the ALL-inclusive model for a system where transitions out of recovery states is possible (ref. 5). With behavioral decomposition, HARP ignores these transitions unless the user invokes the ALL-inclusive multifault model, as shown in figure 54. The system consists of two triads with failure rates $\lambda_1 = \lambda_2 = 0.25 \times 10^{-4}$/hr and the recovery rate $\delta = 0.72 \times 10^{4}$/hr ($\frac{1}{2}$-sec mean recovery). Recovery is always successful, unless a near-coincident fault occurs. A near-coincident fault in the same triad causes that triad to go off-line, but the system remains operational. A near-coincident fault in the other triad causes a system failure. Both triads cannot be executing recovery procedures simultaneously, and the system is operational if at least one triad is operational. For a mission time of 100 hr, the unreliability is 0.504×10^{-9}. Figure 55 shows the instantaneous model when the ALL-inclusive model is chosen.

Figure 54. Two triad fault-tolerant system—full model.

Figure 55. Two triad fault-tolerant system—instantaneous HARP model.

Because HARP uses a simple multifault model (ALL-inclusive model in this case) and behavioral decomposition, the instantaneous coverage requires that the near-coincident fault transitions (e.g., from $R2$ to $F3$) be modeled as system failures. This instantaneous model solution produced an unreliability of 0.608×10^{-9}, which is a conservative error of 20.6 percent. Conservative errors of this magnitude should be acceptable for most applications. If not, the XHARP system applied to this model yields an unreliability of 0.504×10^{-9}.

7.3. Markovian Models With Hot Weibull Spares

Markov models with Weibull hot spares behave differently than Markov models with constant failure rate hot spares. Constant failure rate models exhibit the so called memoryless property. That is, when a spare (warm or cold) is switched in, the spare that has not failed behaves as if it were brand new. By definition, this condition is guaranteed for a cold spare. The constant failure rate spare does not remember its past use history.

A Weibull spare, by contrast, does remember its use history. So when a hot Weibull spare is switched in, it behaves as if it were operating from time zero with the exception that it was not allowed to fail until it switched in. If the Weibull is a decreasing failure rate, the part that is switched in has a lower instantaneous failure rate than a brand new part. The opposite is true for a cold Weibull spare. When the cold Weibull spare is switched in, its instantaneous failure rate is at maximum. Thus, Weibull cold spares may not increase system reliability as much as a hot Weibull spare for decreasing rates. This failure behavior differs from what constant failure rate parts exhibit (ref. 19).

7.4. Non-Markovian Models With Weibull Failure Rates

A Markov chain with nonconstant failure rates such as the Weibull is called a nonhomogeneous Markov chain. This stochastic process has one time variable (mission clock) that starts at time zero with a value of zero. When a cold or warm Weibull spare is introduced into such a model,

another time variable is required that is initiated when the cold or warm spare is activated. Stochastic processes of this type are called mixed-Markov processes, that is, Markov and semi-Markov. Mixed-Markov models are difficult to solve analytically, and HARP does not have this capability. MCI-HARP uses a specialized Monte Carlo simulation engine that was designed to solve mixed-Markov models.

A nonhomogeneous Markov chain with Weibull failure distributions is no longer Markovian if repair is introduced. As in the models previously discussed, a separate clock must be established to track repaired components. Although MCI-HARP has not been applied in this manner, the basic capability exists.

NASA Langley Research Center
Hampton, VA 23681-0001
June 27, 1994

Appendix A

Known Bugs in HARP Version 6.1

This appendix lists known bugs in HARP version 6.1. These bugs were fixed in version 6.2.

1. An infinite loop is encountered if the mission time, sampling interval, or variation is entered as anything but a number.
2. The ESPN FEHM model has the following bugs.
   - It does not recognize a net in which all transition firing times are constant and therefore cannot solve it.
   - When only one exit is reachable in the net or the probability of reaching the exit is sufficiently close to 1.0, an infinite loop occurs as the program keeps doubling the trials.
   - Higher moments of time to reach various exits are sometimes incorrectly set to zero.
   - In cases where S exit is a rare event, the outcome of the simulation may vary drastically and cause the unreliability of the overall system to change dramatically.
   - In the case of transient faults, the program does not always simulate the net exactly as the drawing indicates.
   - The seed for the random generator in the UNIX version can repeat itself; hence, the simulation can have undesired correlation.
3. The ARIES model has the following bugs:
   - It does not accurately calculate each exit probability correctly and ignores the variable that pertains to failure of the recovery hardware. However, the model is correct as defined in this Technical Paper.
   - It incorrectly calculates the moments for each exit in the case that one or more component failure times are Weibull distributed.
4. Certain combinations of cold spare gates and functional dependency gates give rise to a CSP gate behavior that has not yet been implemented in HARP. The affected combinations of gates are somewhat unusual and should rarely be needed. The behavior of the cold spare gate and its proper uses are defined in full in this Technical Paper. In particular, users of HARP version 6.1 should avoid combining functional dependency gates and cold spare gates with shared spares in such a way that an input event of one of the cold spare gates is also a dependent event of the functional dependency gate.

Appendix B

Warning and Error Messages

This appendix contains the warning and error messages for the HARP program.

```
+++ WARNING C100: ILLEGAL PARAMETER IN WEIBULL DEVIATE GENERATOR SUBROUTINE,
     ALPHA = 0 in DVWEBL +++
   File:           DEVGEN
   Subroutine:     DVWEBL
```

Meaning: If the FEHM model chosen is the HARP default ESPN model, this model is simulated for solution. In the course of the simulation, deviates of the distributions for the timed transitions are generated. When the specified distribution is Weibull and alpha, the shape parameter (ref. 44), is zero, the resulting function is not a distribution. In this case, the deviate returned is set to zero.

```
+++ WARNING C150: MORE TRIALS NEEDED FOR NORMAL APPROXIMATION IN SIMULATOR +++
   File:           HARPSIM
   Subroutine:     STATS
```

Meaning: If the FEHM model chosen is the HARP default ESPN model, this model is simulated for solution. This warning may appear during the statistical analysis of the simulation data. In estimating the confidence intervals about the exit probabilities, a normal approximation to the binomial distribution is used. This approximation is valid if n*p is greater than 5. (This rule is discussed more fully in ref. 44.) If n*p is less than 5, then the number of trials is doubled and the simulation continues. The initial number of simulation trials run is 1000, so this message appears if any of the exit probabilities are less than 0.005.

```
+++ WARNING C155: MORE SIMULATION TRIALS ARE NEEDED TO REDUCE PERCENT**** ERROR
     TO WITHIN THE VALUE SPECIFIED BY USER +++
   File:           HARPSIM
   Subroutine:     STATS
```

Meaning: If the FEHM model chosen is the HARP default ESPN model, then this model is simulated for solution. This warning may appear during the statistical analysis of the simulation data. Confidence intervals about the exit probabilities are generated, and a check is made as to the relative size of the interval. If the band/estimate (*100) is greater than the percent error specified by the user, then more trials are needed to reduce the width of the interval (band). In this case, the number of trials is doubled, and the simulation continues. The initial number of simulation trials run is 1000; this number is doubled until percent error in all three exit probabilities is less than the value specified by the user. If this message appears more than 5 or 6 times, the simulation may be long (maybe an hour), and the user may want to reduce the percent error requested. Each time this message appears, the percent error for the exit being analyzed is displayed.

**** ERROR C500: ILLEGAL VALUE FOR COVVAL ***

File: COVFAC

Subroutine: COVNOM

Meaning: This error is displayed whenever the numeric value for a coverage factor is greater than 1 or less than zero. If all the inputs are correct, this error signifies that the basic justification for the behavioral decomposition has been violated. That is, the average amount of time spent in the coverage model is comparable with the time between failures rather than relatively different orders of magnitude.

**** ERROR C600: CALLING FOR A DEVIATE FOR NODIS ***

File: DEVGEN

Subroutine: DEVGEN

Meaning: If the FEHM model chosen is the HARP default ESPN model, then this model is simulated for solution. During the simulation, deviates of the distributions for the timed transitions are generated. If the user has specified "No Distribution" (believing that the corresponding transition would never be enabled), and a deviate is called for, then this error is printed and execution halts.

**** ERROR C750: FEHM FILE NOT FOUND ***

File: COVFAC

Subroutine: COVNOM or COVVAR

Meaning: This error appears if the file containing the parameters to be used for the FEHM file does not exist. The name of the file requested is displayed. The obvious correction is to be certain that the FEHM files listed in the dictionary do indeed exist.

**** ERROR C755: UNRECOGNIZED FEHM TYPE ***

File: COVFAC

Subroutine: COVNOM or COVVAR

Meaning: The first line of a file containing the parameters for a FEHM file states the model type being described. If the COVFAC routine does not recognize the file type, this message is printed, as is the first line of the FEHM parameter file, and the name of the FEHM file.

**** ERROR C900: INVALID RATE FOR EXPONENTIAL DIST

File: DISTS

Subroutine: EXP or MEXP

Meaning: This message appears if the rate parameter (1/mean) for the (negative) exponential distribution (for DISTRIBUTIONS FEHM model) is less than or equal to zero. To correct this error, check the FEHM file(s) to be certain that the specified value of any rate parameter is positive.

**** ERROR C905: SCALE PARAMETER FOR WEIBULL IS ZERO

File: DISTS

Subroutine: WEIBL or MWEIBL

Meaning: This message appears if the parameter for the Weibull distribution of time to exit the FEHM model is illegal. To correct this error, check the FEHM parameter file(s) to be certain that the scale (rate) parameter for the Weibull distribution is positive.

**** ERROR C910: SHAPE PARAMETER FOR WEIBULL IS ZERO

File: DISTS

Subroutine: WEIBL or MWEIBL

Meaning: This message appears if the parameter for the Weibull distribution of time to exit the FEHM model is illegal. To correct this error, check the FEHM parameter file(s) to be certain that the shape (alpha) parameter for the Weibull distribution is positive.

**** ERROR C915: HIGH VALUE < LOW VALUE FOR UNIFORM DIST.

File: DISTS

Subroutine: UNIFRM or MUNIF

Meaning: This message appears if the parameters for the uniform distribution of time to exit the FEHM model are illegal (i.e., if the upper limit is less than or equal to the lower limit). To correct this error, check the FEHM parameter file(s) to be certain that the upper and lower limits are correct.

**** ERROR C920: SCALE PARAMETER FOR GAMMA IS ZERO

File: DISTS

Subroutine: GAMDST or MGMDST

Meaning: This message appears if the scale parameter for the gamma distribution of time to exit the FEHM model is zero. To correct this error, check the FEHM parameter file(s) to be certain that the scale (rate) parameter for the gamma distribution is positive.

**** ERROR C925: SHAPE PARAMETER FOR GAMMA IS ZERO

File: DISTS

Subroutine: GAMDST or MGMDST

Meaning: This message appears if the shape parameter for the gamma distribution of time to exit the FEHM model is zero. To correct this error, check the FEHM parameter file(s) to be certain that the shape (alpha) parameter for the gamma distribution is positive.

**** ERROR C930: ILLEGAL PARAMETER FOR HYPEREXP DIST.

File: DISTS

Subroutine: HYPER or MHYPER

Meaning: This message appears if a rate parameter or probability for the hyperexponential distribution of time to exit the FEHM model is less than or equal to zero. To correct this

error, check the FEHM parameter file(s) to be certain that the probabilities and rates for the hyperexponential distribution are positive.

```
**** ERROR C935: ILLEGAL PARAMETER FOR HYPOEXP DIST.

   File:           DISTS

   Subroutine:     HYPO or MHYPO
```

Meaning: This message appears if a rate parameter for the hypoexponential distribution of time to exit the FEHM model is less than or equal to zero. To correct this error, check the FEHM parameter file(s) to be certain that the probabilities for the hypoexponential distribution are positive.

```
++++ WARNING E031, WEIBULL FAILURE RATE USED WITH REPAIR

     File:       FILL

     Subroutine: FILSYM
```

Meaning: The model contains both a (time varying) Weibull failure rate transition and a constant repair rate transition. The results may be meaningless with this combination.

Action: The user needs to analyze the model and determine if the combination of time-varying transitions with repair transitions is correct.

```
++++ WARNING E032, WEIBULL FAILURE RATE USED WITH COLD SPARES

     File:       FILL

     Subroutine: FILSYM
```

Meaning: The model contains both a (time varying) Weibull failure rate transition and a cold spare (as specified in the fault tree). The results of the solution of this model are suspect.

Action: Check the model to be sure that it is the one intended.

```
++++ WARNING E033, BEHAVIORAL DECOMPOSITION ASSUMPTIONS VIOLATED

     File:       HARPENG

     Subroutine: HARPENG
```

Meaning: The model contains states that are too fast (relative to the slowest FEHM). This warning arises when the fastest mean time to exit for any FEHM is less than 1000 times the mean sojourn time in the fastest state. The warning is issued to alert the user that predicted unreliabilities/availabilities may be overly conservative. The "magic number" 1000 was chosen based on observed typical system models so that this message does not appear too often.

Action: Check the model to be sure that it is the one intended.

```
++++ WARNING E060, INVALID INPUT CHARACTER -x- IS IGNORED

     File:       SCAN

     Subroutine: ICLASS
```

Meaning: A character in the input stream of a symbolic expression cannot be classified as a digit, upper or lower case alphabetic, operation sign, or parenthesis. It is ignored.

Action: If an input file is being used, it should be edited to remove or correct the offending character. If the character is not printable (i.e., a blank or null character appears between the dashes in the error message), the offending character may be a control character.

```
++++ SKIP: WARNING E061 - UNEXPECTED END OF FILE
     File:       FEHMUTL
     Subroutine: SKIP
```

Meaning: The SKIP() subroutine skips a specified number of lines in an input file. If an EOF is encountered unexpectedly during this operation, this warning message is produced.

Action: Check the input files for *harpeng*; one of them might be corrupted.

```
++++ GERK WARNING E200, MANY STEPS
     File:       GCALL
     Subroutine: GCALL
```

Meaning: This message reflects an error code of 3 from the GERK ODE solver. Thus, the integration was not completed because more than 9000 derivative evaluations were needed ($\approx$500 steps). The model may be too stiff for GERK to handle accurately and/or the mission time may be too long.

Action: Determine whether the stiffness is inherent in the model formulation and/or if the mission time can be reduced. If the model cannot be changed, then another ODE solver may be more appropriate.

```
++++ GERK WARNING E201, TOLERANCES RESET: x.xxx-xx y.yyy-yy
     File:       GCALL
     Subroutine: GCALL
```

Meaning: This message reflects an error code of 4 or 5 from the GERK ODE solver. Code 4 means that the integration was not completed because the solution vanished, making a pure relative error test impossible. Thus, GERK must use a nonzero absolute error tolerance to continue. Code 5 means that the integration was not completed because the requested accuracy could not be achieved with the smallest allowable stepsize. Thus, GERK must increase the error tolerance before continued integration can be attempted.

Action: No user action is required. The GCALL subroutine automatically sets a positive absolute error tolerance for a code 4 return and increases the relative error tolerance by a factor of 10 for a code 5 return.

```
++++ GERK WARNING E202, MUCH OUTPUT
     File:       GCALL
     Subroutine: GCALL
```

Meaning: This message reflects an error code of 6 from the GERK ODE solver. Thus, GERK is being used inefficiently in solving the model; too much output is restricting the natural stepsize choice.

Action: If convenient, reduce the mission time.

++++ GERK WARNING E300, (BND) MANY STEPS

File: GCALL

Subroutine: GCALL2

Meaning: This message reflects an error code of 3 from the GERK ODE solver. Thus, the integration was not completed because more than 9000 derivative evaluations were needed (≈500 steps). The model may be too stiff for GERK to handle accurately and/or the mission time may be too long.

Action: Determine whether the stiffness is inherent in the model formulation and/or if the mission time can be reduced. If the model cannot be changed, then another ODE solver may be more appropriate.

++++ GERK WARNING E301, (BND) TOLERANCES RESET: x.xxx-xx y.yyy-yy

File: GCALL

Subroutine: GCALL2

Meaning: This message reflects an error code of 4 or 5 from the GERK ODE solver. Code 4 means that the integration was not completed because the solution vanished, making a pure relative error test impossible. Thus, GERK must use a nonzero absolute error tolerance to continue. Code 5 means that the integration was not completed because the requested accuracy could not be achieved with the smallest allowable stepsize. Thus, GERK must increase the error tolerance before attempting continued integration.

Action: No user action is required. The GCALL2 subroutine automatically sets a positive absolute error tolerance for a code 4 return and increases the relative error tolerance by a factor of 10 for a code 5 return.

++++ GERK WARNING E302, (BND) MUCH OUTPUT

File: GCALL

Subroutine: GCALL2

Meaning: This message reflects an error code of 6 from the GERK ODE solver. Thus, GERK is being used inefficiently in solving the model; too much output is restricting the natural stepsize choice.

Action: If convenient, reduce the mission time.

**** ERROR E510, NUMBER OF STATES OUT OF RANGE

File: FILL

Subroutine: FILL

Meaning: The number of states specified for the model was either less than one or more than the maximum number allowed by the program.

Action: Redefine the number of states for the model. If a larger model is to be run, consult section 5.3 of this Technical Paper.

**** ERROR E511, ROW AND COLUMN OUT OF ORDER

File: FILL

Subroutine: FILL

Meaning: The program requires that matrix entries be entered in row-major order so that the sparse matrix data structure can be built properly (i.e., rows must be entered in ascending order, and within a row columns must be in ascending order). This message indicates that the program detected an entry out of this order.

**** ERROR E512, ROW AND COLUMN OUT OF RANGE

File: FILL

Subroutine: FILL

Meaning: The row or column index used to specify a matrix entry is greater than the number of states.

Action: Rerun *fiface* for the model.

**** ERROR E530, PARAMETER TYPE UNRECOGNIZED

File: FILL

Subroutine: FILSYM

Meaning: When reading from an echo file, the parameter type given does not match a defined value.

Action: Rerun the model through *harpeng* without the echo file.

**** ERROR E550, NEW SYMBOL ADDED IN NEXT FAULT RATE EXPRESSION

File: GET

Subroutine: GETNF

Meaning: To reduce the number of symbol definition and evaluation passes over the symbol table, no new parameters may be defined by the next fault rate symbolic expression. This error indicates that a new parameter was introduced in the next fault rate expression.

Action: Be sure that any symbols that appear in the near-coincident fault rate expression appear in the dictionary.

**** ERROR E570, INCORRECT SYNTAX - EXPRESSION INCOMPLETE

File: SYM

Subroutine: SYMINP

Meaning: The symbolic expression was prematurely terminated by a semicolon. The semicolon may have appeared after an operation symbol ($+, -, *$) or the expression may lack closing parentheses.

Action: Rerun *fiface* for the model.

```
**** ERROR E580, INCORRECT SYNTAX, UNEXPECTED SYMBOL
       OFFENDING SEQUENCE IS:
    File:       STORE
    Subroutine: STORE
```

Meaning: The symbolic expression does not conform to the proper syntax as implemented in the syntax table.

Action: Correct the symbolic expression. The following are usual suspects:

1. No multiplicative sign between a token and its coefficient
2. Double $--$, $++$, or ** signs
3. No semicolon

```
**** ERROR E590, TRYING TO DEALLOCATE WRONG TERM
    File:       ALLOC
    Subroutine: DALLCT
```

Meaning: A call to the DALLCT subroutine has been made with a pointer to a TERM node that was not the most recently allocated.

Action: A serious logical error has occurred in the program and should be reported along with a copy of the input files for this model to the first author of this Technical Paper.

```
**** ERROR E591, POINTER TO TERM LIST NEGATIVE
    File:       ALLOC
    Subroutine: DALLCT
```

Meaning: A call to the DALLCT subroutine has been made before a TERM node has been allocated.

Action: A serious logical error has occurred in the program and should be reported along with a copy of the input files for this model to the first author of this Technical Paper.

```
**** ERROR E600, OUT OF SPACE FOR TERM LIST
    File:       ALLOC
    Subroutine: ALLOCT
```

Meaning: More TERM nodes are required to represent the model than are currently allocated by the program.

Action: To run such a large model, the program must be recompiled with more storage space for the data structures FACLHD and NXTTRM. See section 5.3 of this Technical Paper.

```
**** ERROR E610, OUT OF SPACE FOR FACTOR LIST
    File:       ALLOC
    Subroutine: ALLOCF
```

Meaning: More FACTOR nodes are required to represent the model than are currently allocated by the program.

Action: To run such a large model, the program must be recompiled with more storage space for the data structures FACTYP, SYMENT, and NXTFAC. See section 5.3 of this Technical Paper.

```
**** ERROR E620, OUT OF SPACE FOR SYMBOL TABLE
     File:       ALLOC
     Subroutine: ALLOCS
```

Meaning: More symbol table entries are required to represent the model than are currently allocated by the program.

Action: To run such a large model, the program must be recompiled with more storage space for the symbol table. See section 5.3 of this Technical Paper.

```
**** ERROR E700, IMPROPER CALL TO GERK
     File:       GCALL
     Subroutine: GCALL
```

Meaning: This message reflects an error code of 7 from the GERK ODE solver. Thus, a call to the GERK subroutine has been made with invalid input parameters. Possible reasons are NEQN <= 0, T = TOUT and IFLAG = /, +1, or −1, RELERR < 0, ABSERR < 0, IFLAG = 0, IFLAG < − 2, IFLAG > 7.

Action: A serious logical error has occurred in the program and should be reported along with a copy of the input files for this model to the first author of this Technical Paper. There is no intermediate circumvention.

```
**** ERROR E710, WRONG FUNCTION CODE TO SETVAL
     File:       SET
     Subroutine: SETVAL
```

Meaning: A call to the SETVAL subroutine has been made with an undefined function code.

Action: A serious logical error has occurred in the program and should be reported along with a copy of the input files for this model to the first author of this Technical Paper. There is no intermediate circumvention.

```
**** ERROR E720, NEGATIVE TRANSITION RATE
     File:       EVAL
     Subroutine: SYMEVL
```

Meaning: One of the off-diagonal symbolic transition rates has appeared as a negative number.

Action: Check the values assigned to the symbols, and check the numeric evaluation of the rate.

```
**** ERROR E800, (BND) IMPROPER CALL TO GERK
     File:       GCALL
```

```
Subroutine: GCALL2
```

Meaning: This message reflects an error code of 7 from the GERK ODE solver. Thus, a call to the GERK subroutine has been made with invalid input parameters. Possible reasons are NEQN $< = 0$, T = TOUT and IFLAG = /, $+1$, or -1, RELERR < 0, ABSERR < 0, IFLAG $= 0$, IFLAG < -2, IFLAG > 7.

Action: A serious logical error has occurred in the program and should be reported along with a copy of the input files for this model to the first author of this Technical Paper. There is no intermediate circumvention.

```
++++ RDDICT: WARNING F100 - FAILURE RATE VARIABLE, rate,
       TOO LONG - TRUNCATED TO size CHARACTERS ++++
     File:        RDDICT
     Subroutine:  RDDICT
```

Meaning: The character string representing a failure rate variable that was read from the dictionary was longer than the allowable size for a failure rate variable and was truncated to the maximum legal size.

Action: Correct the offending failure rate variable in the dictionary or use the truncated one.

```
++++ FT2MC: WARNING F102 - STATE TABLE TOO SMALL TO
        REMEMBER ALL SYSTEM STATES, RETRYING... ++++
     File:        FT2MC
     Subroutine:  FT2MC
```

Meaning: To prune the state search tree, system states are remembered by storing them in a State Table when they are first generated. If a previously generated state is regenerated at some point, it does not need to be retested for system failure or added to the queue for later expansion. Both unique nonfailure states and states that cause overall system failure are stored in the State Table. When the fault tree causes so many of these states to be generated that the State Table is filled, this message is printed. FT2MC() tries to reprocess the fault tree by placing itself in a mode where the states representing system failure are not individually stored, but instead only a single FE (failure due to exhaustion) state is stored for each component. The unique nonfailure states are still stored in the State Table. Depending on the fault tree, the conversion process to a Markov chain may take longer, but this method may allow larger systems to be processed than could be handled before.

Action: None. If error F508 occurs after FT2MC() makes its second attempt, then the State Table is too small and the number of states parameters (MSTATS in FT2MC, TABLEN and PRIME2 in INISTA (in CKSTAT.FOR source file)) must be increased to handle a fault tree of this size. See section 5.3 of this Technical Paper for information on increasing the number of states HARP can handle.

```
++++ FT2MC: WARNING F103 - MARKOV CHAIN TRUNCATED BEFORE ANY FAILURE
      EXHAUSTION STATES WERE REACHED ++++
     File:        FT2MC
     Subroutine:  FT2MC
```

Meaning: When truncation of the Markov chain to a user-specified number of states is enabled, the generation of states is stopped after the specified number of components have failed throughout the system. If the redundancy of each component type in the system is larger than the user-specified truncation cutoff number of failures, than no FE states have been reached in the Markov chain when FT2MC stops generating states. This warning message is produced in that situation.

```
++++ CKSTAT: WARNING F200 - STATE n FOUND ALREADY IN STATE TABLE
   WHILE ATTEMPTING TO ADD TO THE STATE TABLE ++++
     File:        CKSTAT
     Subroutine:  CKSTAT
```

Meaning: When adding a (supposedly) new system state to the State Table, CKSTAT() found that the state was already present in the State Table. This action has no functional effect on the operation of the conversion process, but it may indicate an internal programming problem somewhere in the FT2MC subsystem.

Action: Report warning message to the first author of this Technical Paper.

```
++++ INPTRE: WARNING F300 - MAX BASIC COMPONENTS ALLOWED
       IN FAULT TREE = maxcmpts ++++
     File:        INPTRE
     Subroutine:  INPTRE
```

Meaning: When entering a textual description of a fault tree, the user has attempted to specify more basic component nodes than the system currently allows.

Action: Rebuild the FT2MC subsystem with a larger value for the maximum number of basic component nodes allowed; that is, increase the MCMPTS parameter in all FORTRAN source files, recompile, and relink.

```
++++ INPTRE: WARNING F301 - MAX NO. NODES ALLOWED IN FAULT TREE = maxnodes ++++
     File:        INPTRE
     Subroutine:  INPTRE
```

Meaning: When entering a textual description of a fault tree, the user has attempted to specify more fault tree nodes than the system currently allows.

Action: Rebuild the FT2MC subsystem with a larger value for the maximum number of fault tree nodes allowed; that is, increase the MNODES parameter in all FORTRAN source files, recompile, and relink.

```
**** ENCSTA: ERROR F400 - # FAILURES EXCEED DECLARED LENGTH OF AN ENCODED STATE
     File:        ENCSTA
     Subroutine:  ENCSTA/DECSTA
```

Meaning: ENCSTA() encodes a system state from a tuple of working components to a list of the components that failed to bring the system from its original state to the current STATE. DECSTA() performs the corresponding inverting conversion. Both routines declare a vector of

a certain length to hold the encoded state. If more component failures have occurred than can be stored in the declared vector, this error message is produced. As currently implemented, this error represents an internal programming error within FT2MC.

Action: Report error to the first author of this Technical Paper.

```
**** DECSTA: ERROR F401 - CMPNT TYPE cmp SPECIFIED IN ENCODED STATE IS
    OUT OF RANGE
     File:        ENCSTA
     Subroutine:  DECSTA
```

Meaning: ENCSTA() encodes a system state from a tuple of working components to a list of the components that failed to bring the system from its original state to the current STATE. DECSTA() performs the corresponding inverting conversion. If during the decoding process DECSTA() encounters in the list of failed components a component type that is not present in the system, this error message is produced. This error represents an internal programming error within FT2MC.

Action: Report error to the first author of this Technical Paper.

```
**** FT2MC: ERROR F500 - FAULT TREE NAME TOO LONG ****
     File:        FT2MC
     Subroutine:  FT2MC
```

Meaning: The modelname passed to FT2MC() was too long and was rejected.

Action: Specify a shorter modelname.

```
**** FT2MC: ERROR F504 - ERROR OPENING MARKOV CHAIN OUTPUT FILE filename ****
     File:        FT2MC
     Subroutine:  FT2MC
```

Meaning: The FT2MC() subroutine encountered an error while trying to open the Markov chain output file. This error is an operating system error rather than an FT2MC error.

Action: Consult the operating system manuals for the cause and possible solutions.

```
**** OUTARC: ERROR F505 - ERROR retcode RETURNED BY INTCHR() ****
     File:        FT2MC
     Subroutine:  OUTARC
```

Meaning: The OUTARC() subroutine outputs a Markov chain arc between an originating system state and a new system state produced when a component fails in the originating state. For the component that fails, the number of those components operational before the failure influences the transition rate of the arc. Thus, the number of components operational before the failure must be converted to a character string and printed. The INTCHR() subroutine encountered an error while attempting to convert this number of components to a character string. This error represents an internal programming error in FT2MC.

Action: Report error to the first author of this Technical Paper.

```
**** FT2MC: ERROR F510 - TOO MANY STATES GENERATED, MAX = mstats
     File:        FT2MC
     Subroutine:  FT2MC
```

Meaning: FT2MC() has generated more than the maximum allowable number of states during the conversion of the fault tree model into a Markov chain.

Action: Increase the MSTATS parameter in the *tdrive* source files and recompile the entire program.

```
**** EXTID: ERROR F511 - STATE state_tuple NOT FOUND IN LINKED LIST
     File:        FT2MC
     Subroutine:  EXTID
```

Meaning: As states are generated during the conversion process, they are stored in a linked list. This list makes it possible to determine whether each state has been generated before or is being generated for the first time. If the state has been generated before, it will have been assigned an external ID number when it was created. EXTID() looks up such a previously generated state in the linked list to determine its ID number. This error message occurs when a state that should be in the linked list is not found there.

Action: Report error to the first author of this Technical Paper.

```
**** FRSTIM: ERROR F512 - INCOMPATIBLE ROOT STATE (state_tuple) FOR
     STATE state_tuple
     File:        CKSTAT
     Subroutine:  FRSTIM
```

Meaning: FRSTIM() uses a STATE and its parent state RSTATE in performing its function. At several places, FRSTIM() performs a consistency check between STATE and RSTATE to ensure that STATE is indeed a proper descendent state of RSTATE. If this consistency check fails (i.e., it is determined that STATE could not possibly be a descendent state of RSTATE), then this error message is produced. This error represents an internal programming error in FT2MC.

Action: Report error to the first author of this Technical Paper.

```
**** RDDICT: ERROR F600 - DICTIONARY FILE NOT FOUND ****
     File:        DICT
     Subroutine:  RDDICT
```

Meaning: The RDDICT() subroutine was unable to find the dictionary file.

Action: Make sure the dictionary file for the fault tree (modelname.DIC) exists before the FT2MC subsystem is called.

```
**** RDDICT: ERROR F601 - DICTIONARY OVERFLOW, MAX NUMBER OF
      COMPONENT TYPES = num ****
     File:        DICT
     Subroutine:  RDDICT
```

Meaning: The RDDICT() subroutine found that the dictionary file for the fault tree contains more than the maximum number of component types allowed in a fault tree.

Action: This fault tree cannot be run through FT2MC unless the FT2MC subsystem is rebuilt with a larger value for MTYPES, which is the limit for the maximum number of component types allowed in a fault tree.

```
**** RDDICT: ERROR F602 - ERROR OPENING DICTIONARY FILE ****
     File:        DICT
     Subroutine:  RDDICT,WRTDCT
```

Meaning: The RDDICT() subroutine encountered an error while trying to open the dictionary file. This error is an operating system error rather than an FT2MC error.

Action: Consult the operating system manuals for the cause and possible solutions.

```
**** RDDICT: ERROR F603 - UNEXPECTED EOL ENCOUNTERED WHILE PARSING NEXT
         ITEM FROM INPUT LINE, OFFSET = offset ****
     File:        DICT
     Subroutine:  RDDICT
```

Meaning: The NXTWRD() subroutine encountered an unexpected End-Of-Line while reading an input line from the dictionary. The dictionary file may be corrupted.

Action: Check the dictionary file. Recreate it if necessary.

```
**** RDDICT: ERROR F604 - ERROR ENCOUNTERED WHILE PARSING NEXT ITEM FROM
     INPUT LINE, OFFSET = offset ****
     File:        DICT
     Subroutine:  RDDICT
```

Meaning: The RDDICT() subroutine encountered an error while reading an input line from the dictionary. The dictionary file may be corrupted.

Action: Check the dictionary file. Recreate it if necessary.

```
**** RDDICT: ERROR F605 - COMPONENT ENTRIES OUT OF ORDER IN DICTIONARY ****
     File:        DICT
     Subroutine:  RDDICT
```

Meaning: While reading the dictionary file, the RDDICT() subroutine found that an entry for a component was not in consecutive order with the other entries. The dictionary file may be corrupted or the user may have made an error when creating the dictionary file.

Action: Check the dictionary file. Recreate it if necessary.

```
**** PASS1: ERROR F700 - FAULT TREE DESCRIPTION FILE (filename) NOT FOUND ****
     File:        BLDLST
     Subroutine:  PASS1
```

Meaning: The PASS1() subroutine could not find the graphics specification language input file containing the fault tree specification.

Action: Make sure that an input file containing a fault tree specification (either a .FTR graphics format input file or a .TXT textual specification language format input file) exists before the FT2MC subsystem is called.

```
**** PASS1: ERROR F701 - UNEXPECTED FIRST INPUT ITEM - OBJECT TOO LONG ****

     File:        BLDLST

     Subroutine:  PASS1
```

Meaning: The first item of every line in a graphics specification language format input file is either 'N' (indicating that this line describes a fault tree node) or 'A' (indicating that this line describes an arc). Either item is only one character in length. If the first item read from an input line is longer than one character in length, then there is an error. The input file is either corrupted, or it is not a graphics specification language format input file.

Action: Make sure the input file <ftreename.FTR> has the correct format (graphics specification language).

```
**** PASS1: ERROR F702 - ERROR OPENING INPUT FILE filename ****

     File:        BLDLST

     Subroutine:  PASS1
```

Meaning: The PASS1() subroutine encountered an error while trying to open the input file containing the fault tree specification in graphics specification language format. This error is an operating system error rather than an FT2MC error.

Action: Consult the operating system manuals for the cause and possible solutions.

```
**** PASS1: ERROR F703 - ERROR ENCOUNTERED WHILE PARSING NEXT WORD FROM

    INPUT LINE:  line ****

     File:        BLDLST

     Subroutine:  PASS1
```

Meaning: The NXTWRD() subroutine encountered an error while reading an input line from the fault tree specification input file. The input file may be corrupted or contain errors.

Action: Check the input file for format errors. Recreate it if necessary with *tdrive* or the graphics fault tree input facility.

```
**** PASS1: ERROR F704 - INDEX LIST OVERFLOW, TOO MANY FT NODES ****

     File:        BLDLST

     Subroutine:  PASS1
```

Meaning: The fault tree specification read from the input file has so many nodes that it overflowed the index list table.

Action: Rebuild the FT2MC subsystem with a larger index list size. The FT2MC() subroutine calls BLDLST() so that the QUEUE array is used for the Index List by BLDLST.

This double duty for the QUEUE array is possible because BLDLST does not use a queue and FT2MC does not use the index list after BLDLST returns. Therefore, space can be saved by using the same array for both purposes. Consequently, to increase the index list size, increase QLEN, the size of the QUEUE array, in the FT2MC() subroutine.

```
**** PASS2: ERROR F705 - UNEXPECTED FIRST INPUT ITEM - OBJECT TOO LONG ****
```

```
     File:         BLDLST
```

```
     Subroutine:   PASS2
```

Meaning: The first item of every line in a graphics specification language format input file is either 'N' (indicating that this line describes a fault tree node) or 'A' (indicating that this line describes an arc). Either item is only one character in length. If the first item read from an input line is longer than one character in length, then there is an error. The input file is either corrupted, or it is not a graphics specification language format input file.

Action: Make sure the input file <MODELNAME.FTR> has the correct format (graphics specification language).

```
**** PASS2: ERROR F706 - ILLEGAL FORM FOR M/N GATE LABEL: label ****
```

```
     File:         BLDLST
```

```
     Subroutine:   PASS2
```

Meaning: This error occurs if the length of the token that is supposed to be a label for an M/N gate is less than three. An M/N gate label has the form: m/n, where m and n are integers. The label therefore must have a length of at least three. If it does not, then it cannot possibly be a valid M/N gate label.

Action: Check the input file <ftreename.FTR> for an M/N gate label with an illegal format and correct it.

```
**** PASS2: ERROR F707 - TOO MANY BASIC COMPONENTS (LEAVES) IN FAULT TREE ****
```

```
     File:         BLDLST
```

```
     Subroutine:   PASS2
```

Meaning: The fault tree specification read from the input file contains more Basic Component nodes than the maximum number allowed in a fault tree.

Action: FT2MC cannot process this fault tree unless FT2MC is rebuilt with a larger value for MCMPTS, the maximum number of Basic Component nodes allowed in the fault tree.

```
**** PASS2: ERROR F708 - ILLEGAL FAULT TREE NODE TYPE: nodetype ****
```

```
     File:         BLDLST
```

```
     Subroutine:   PASS2
```

Meaning: A fault tree node type read from the input file is not one of the supported types defined in the GATES COMMON block. The value for <nodetype> printed as part of the previous message is an integer value.

Action: Check the input file <MODELNAME.FTR> for an error in one of the fault tree node description lines.

```
**** PASS2: ERROR F709 - LOOKUP IN INDEX LIST FAILED FOR ITEM AT
    LOCATION (x,y) ****
     File:        BLDLST
     Subroutine:  PASS2
```

Meaning: The PASS2() subroutine could not find an entry in the index list for one of the fault tree nodes. This error represents an internal programming error in PASS1 and PASS2.

Action: Report error to the first author of this Technical Paper.

```
**** PASS2: ERROR F710 - DEST COORD MISMATCH FOR INCOMING ARC ****
     File:        BLDLST
     Subroutine:  PASS2
```

Meaning: The graphics fault tree input facility records both incoming and outgoing arcs for fault tree nodes in the fault tree specification file that it produces for FT2MC. An incoming arc description specifies both the node at the source of the arc and the node at the destination of the arc. The destination node for an incoming arc is the node currently being processed. An outgoing arc description specifies only the node at the destination of the arc. FT2MC only needs to concern itself with incoming arcs. FT2MC can determine whether an arc description is for an incoming arc or an outgoing arc by looking for the destination node to be the same as the node currently being processed. If it is not and no destination node is specified, then the arc is an outgoing arc and can be ignored. However, if the destination node is specified and it is not the same as the node currently being processed, then there is an error in the fault tree specification. This error message is produced in that event.

Action: Check the fault tree specification in the input file <MODELNAME.FTR> and correct it.

```
**** PASS2: ERROR F711 - ERROR OPENING INPUT FILE filename ****
     File:        BLDLST
     Subroutine:  PASS2
```

Meaning: The PASS2() subroutine encountered an error while trying to open the input file containing the fault tree specification in graphics specification language format. This error is an operating system error rather than an FT2MC error.

Action: Consult the operating system manuals for the cause and possible solutions.

```
**** PASS2: ERROR F712 - ERROR PARSING NEXT WORD FROM INPUT LINE:  line ****
     File:        BLDLST
     Subroutine:  PASS2
```

Meaning: The NXTWRD() subroutine encountered an error while reading an input line from the fault tree specification input file. The input file may be corrupted or contain errors.

Action: Check the input file for format errors. Recreate it if necessary with *tdrive* or the graphics fault tree input facility.

```
**** LOOKUP: ERROR F713 - ILLEGAL FUNCT: f ****

     File:        BLDLST

     Subroutine:  LOOKUP
```

Meaning: The value of f in the previous message is an integer. The LOOKUP() subroutine looks up a fault tree node in the index list and allows the calling routine to either read or write to the pointer field (points into place table) of the index list entry for the fault tree node. The calling routine specifies which operation it wants to do through a subroutine argument FUNCT, where FUNCT = 0 means read and FUNCT = 1 means write. Any other value of FUNCT is unsupported and produces this error message. This error represents a programming error within PASS2.

Action: Report error to the first author of this Technical Paper.

```
**** LOOKUP: ERROR F714 - PLACE AT (x,y) NOT FOUND IN INDEX LIST ****

     File:        BLDLST

     Subroutine:  LOOKUP
```

Meaning: The LOOKUP() subroutine could not find an entry in the index list for one of the fault tree nodes. This error represents an internal programming error in PASS1 and PASS2.

Action: Report error to the first author of this Technical Paper.

```
**** PASS3: ERROR F715 - CARDINALITY OF INCOMING ARCS (c) FOR m/n GATE
    DO NOT MATCH N = n of M/N GATE   ****

     File:        BLDLST

     Subroutine:  PASS3
```

Meaning: The values of c, m, and n in the previous message are integers. The fault tree specified in the modelname.FTR input file contained an m/n gate whose number of incoming arcs did not match the parameter n of the gate. Compound arcs (i.e., arcs whose sources are basic component nodes representing several redundant components) count as several individual arcs rather than as one arc. For example, a compound incoming arc whose source node is a basic component node representing three redundant components of type 1 (a 3*1 basic component node) counts as three individual arcs rather than as one arc.

Action: Examine the MODELNAME.FTR or the MODELNAME.TXT input file and correct any m/n gates that do not have exactly n incoming arcs (considering compound arcs as described previously).

```
**** PASS3: ERROR F716 - DEPENDENT EVENTS FOR A FUNCTIONAL DEPENDENCY
    GATE MUST BE BASIC EVENTS; ARC arc OF NODE node
    IS NOT A BASIC EVENT

     File:        BLDLST

     Subroutine:  PASS3
```

Meaning: Functional Dependency gates can have only basic event nodes as dependent events (i.e., all events after the first, or leftmost, incoming arc to the gate). The trigger event (first, or leftmost, incoming arc) can be any type of event (i.e., the trigger arc may come from any legal

type of gate or node). If any dependent events of the Functional Dependency gate are not basic events (i.e., the arc does not come from a basic component node), then this error message is printed.

Action: Examine the MODELNAME.FTR or the MODELNAME.TXT input file and correct any Functional Dependency gates with dependent events that are not basic events.

```
**** PASS3: ERROR F717 - ALL DESCENDENT EVENTS FOR A COLD SPARE GATE MUST
    BE UNREPLICATED BASIC EVENTS; ARC arc OF NODE node
    IS NOT A BASIC EVENT
     File:        BLDLST
     Subroutine:  PASS3
```

Meaning: Cold Spare gates can have only unreplicated basic event nodes as descendent events (i.e., all incoming arcs must come from basic component nodes). If any descendent events of the Cold Spare gate are not basic events, this error message is printed.

Action: Examine the MODELNAME.FTR or the MODELNAME.TXT input file and correct any Cold Spare gates with descendent events that are not basic events.

```
**** PASS3: ERROR F718 - PRIORITY-AND GATES MUST HAVE 2 INCOMING ARCS;
    NODE node HAS numarcs ARCS
     File:        BLDLST
     Subroutine:  PASS3
```

Meaning: Priority And gates must have exactly two incoming arcs. If a Priority And gate has any number of incoming arcs other than two, this error message is printed.

Action: Examine the MODELNAME.FTR or the MODELNAME.TXT input file and correct any Priority And gates that have other than exactly two incoming arcs.

```
**** PASS3: ERROR F719 - ALL DESCENDENT EVENTS FOR A COLD SPARE GATE MUST
    BE UNREPLICATED BASIC EVENTS; ARC arc OF NODE node
    IS A REPLICATED BASIC EVENT
     File:        BLDLST
     Subroutine:  PASS3
```

Meaning: Cold Spare gates can have only unreplicated basic event nodes as descendent events (i.e., all incoming arcs must come from basic component nodes). If any descendent events of the Cold Spare gate are replicated basic events, this error message is printed.

Action: Examine the MODELNAME.FTR or the MODELNAME.TXT input file and correct any Cold Spare gates with descendent events that are replicated basic events.

```
**** PASS3: ERROR F720 - COLD SPARE GATES SHARING A SPARE WITH
    OTHER COLD SPARE GATE(S) MUST HAVE 2
    INCOMING ARCS; NODE node HAS numarcs ARCS
     File:        BLDLST
     Subroutine:  PASS3
```

Meaning: Cold Spare gates whose spare (dependent) component is shared with another Cold Spare gate are restricted to having only one spare. The gate can therefore have only two incoming arcs—one for the primary component and one for its spare (each of these must be an unreplicated basic event, as described for error F719). If a Cold Spare gate that shares a spare with any other Cold Spare gate(s) has any number of incoming arcs other than two, this error message is printed. NOTE: Cold Spare gates that do not share any spares with other Cold Spare gates are not subject to this restriction; they may have any number of incoming arcs up to the maximum (specified by the MARCS parameter).

Action: Examine the MODELNAME.FTR or the MODELNAME.TXT input file and correct any Cold Spare gates with shared spares that have other than exactly two incoming arcs.

```
**** PASS3: ERROR F721 - ALL DESCENDENT EVENTS EXCEPT THE LEFTMOST

    EVENT OF A SEQUENCE GATE MUST BE BASIC EVENTS;

    ARC arc OF NODE node IS NOT A BASIC EVENT

     File:        BLDLST

     Subroutine:  PASS3
```

Meaning: Sequence gates can have only (possibly replicated) basic event nodes as descendent events (i.e., all incoming arcs must come from basic component nodes) except for the leftmost descendent event, which can be any general event. If any descendent events of a Sequence gate other than the leftmost are not basic events, this error message is printed.

Action: Examine the MODELNAME.FTR or the MODELNAME.TXT input file and correct any Sequence gates with descendent events that are not basic events.

```
**** TRVTRE: ERROR F800 - TABLE TOO SMALL TO HOLD STACK ****

     File:        TRVTRE

     Subroutine:  TRVTRE,DEPCHK,CSPCHK,PACHK
```

Meaning: The TRVTRE() subroutine uses a stack to simulate a recursive traversal of the fault tree. This stack is stored at the end of the place table constructed by BLDLST(). If TRVTRE() finds that the stack it needs is too big to fit onto the end of the place table array, this error message is produced.

Action: FT2MC cannot process this fault tree unless FT2MC is rebuilt with a larger value for FTLEN, which is the size of the array containing the place table.

```
**** TRVTRE: ERROR F801 - ILLEGAL PLACE TYPE (nodetype) FOR PLACE node AT

     OFFSET offset IN PLACE TABLE ****

     File:        TRVTRE

     Subroutine:  TRVTRE,DEPCHK,CSPCHK,PACHK
```

Meaning: The values for <nodetype>, <node>, and <offset> in the previous message are all integers. The TRVTRE() subroutine detected an illegal fault tree node type stored in the place table. Thus, the place table is probably corrupted. This error represents an internal programming error in BLDLST and/or TRVTRE.

Action: Report error to the first author of this Technical Paper.

```
**** TRVTRE: ERROR F806 - INTERNAL ERROR: PLACE TABLE OFFSET < 0 ( neg val )
     File:         TRVTRE
     Subroutine:   TRVTRE,DEPCHK,CSPCHK,PACHK
```

Meaning: This internal error indicates that a fault tree arc has been flagged as severed and incorrectly handled.

Action: Report error to the first author of this Technical Paper.

```
**** TRVTRE: ERROR F807 - CSP GATE AT OFFSET place IN PLACE TABLE
     NOT FOUND IN TABLE OF CSP GATES (CSPTAB)
     File:         TRVTRE
     Subroutine:   TRVTRE,DEPCHK,PACHK,SEQCHK
```

Meaning: This internal error indicates a Cold Spare gate in the fault tree was not found in the table of Cold Spare gates (CSPTAB).

Action: Report error to the first author of this Technical Paper.

```
**** CSPCHK: ERROR F808 - LINKED LIST OF CSP GATE DESCENDENTS
     (CSPRNT or REPEAT) IS CORRUPTED:
     COMPONENT cmpnt IS NOT A DESCENDENT
     OF CSP GATE gate AT LOCATION cspoff
     IN PLACE TABLE
     File:         TRVTRE
     Subroutine:   CSPCHK,DETUSD
```

Meaning: This internal error indicates that a component that was supposed to be a descendent of a Cold Spare gate according to either the REPEAT linked list or one of the CSPRNT() linked lists was in fact found not to be a descendent of the specified Cold Spare gate according to the fault tree data structure (the Place Table). The two internal data structures therefore do not agree.

Action: Report error to the first author of this Technical Paper.

```
**** CSPCHK: ERROR F809 - COMPONENT cmpnt HAS SEVERAL CSP GATE
     PARENTS, INCLUDING CSP GATE gate
     (AT LOCATION cspoff IN THE PLACE
     TABLE) WHICH IS SUPPOSED TO HAVE
     NO SHARED SPARES
     File:         TRVTRE
     Subroutine:   CSPCHK
```

Meaning: This internal error occurs when a component is supposed to be a descendent of several Cold Spare gates (according to the CSPRNT() array of linked lists) and one of these Cold Spare gates is not supposed to share any of its spares with any other Cold Spare gate

according to the SHRDSP field of its entry in the fault tree data structure (the Place Table). By definition, the component has to be shared between the several Cold Spare gates to be their descendent; thus, the Cold Spare gate is either incorrectly marked as not sharing spares, or the CSPRNT() data structure is corrupted.

Action: Report error to the first author of this Technical Paper.

```
**** DETUSD: ERROR F810 - CANNOT TELL HOW MANY OF COMPONENT TYPE
     compnt ARE BEING USED BY CSP GATE gate
     (AT LOC cspoff IN THE PLACE TABLE)
     File:        TRVTRE
     Subroutine:  DETUSD
```

Meaning: This internal error occurs when subroutine DETUSD() is called to try to determine how many components of a basic event are on-line and in use by a Cold Spare gate that does not share any of its spares with any other Cold Spare gate. Cold Spare gates that share spares with other Cold Spare gates have a "components-in-use" descriptor in the state tuple. Cold Spare gates that do not share any of their spares do not have such a descriptor in the state tuple. Since DETUSD() determines how many of a components are in use by examining the appropriate descriptor, it cannot determine how many of the requested components are in use (because there is no descriptor to examine). Since DETUSD() should never be called for a Cold Spare gate that does not share its spares, this error is an internal error.

Action: Report error to the first author of this Technical Paper.

```
**** CVRTXT: ERROR F900 - FAULT TREE NAME TOO LONG ****
     File:        CVRTXT
     Subroutine:  CVRTXT
```

Meaning: The modelname passed to CVRTXT() was too long and was rejected.

Action: Specify a shorter modelname.

```
**** CVRTXT: ERROR F901 - "NODE" MISSING ON INPUT LINE line ****
     File:        CVRTXT
     Subroutine:  CVRTXT
```

Meaning: A syntax error occurred in an input line read from the MODELNAME.TXT file; the offending line is printed for the user's inspection. The keyword "NODE" is not present where it should be in the input line.

Action: Edit the input file MODELNAME.TXT (containing the textual description of the fault tree) and fix the syntax error in the appropriate line.

```
**** CVRTXT: ERROR F902 - TOO MANY NODES, MAX NO. NODES ALLOWED = maxnodes ****
     File:        CVRTXT
     Subroutine:  CVRTXT
```

Meaning: The fault tree described in the MODELNAME.TXT input file contains too many nodes.

Action: Rebuild the FT2MC subsystem with a larger value for the maximum number of fault tree nodes allowed; that is, increase the MNODES parameter in all FORTRAN source files, recompile, and relink.

```
**** CVRTXT: ERROR F903 - NODE NUMBER MISSING ON INPUT LINE line ****
```

```
     File:        CVRTXT
```

```
     Subroutine:  CVRTXT
```

Meaning: A syntax error occurred in an input line read from the MODELNAME.TXT file; the offending line is printed for the user's inspection. The number that identifies a node is not present where it should be in the input line.

Action: Edit the input file MODELNAME.TXT (containing the textual description of the fault tree) and fix the syntax error in the appropriate line.

```
**** CVRTXT: ERROR F904 - "TYPE" MISSING ON INPUT LINE line ****
```

```
     File:        CVRTXT
```

```
     Subroutine:  CVRTXT
```

Meaning: A syntax error occurred in an input line read from the MODELNAME.TXT file; the offending line is printed for the user's inspection. The keyword "TYPE" is not present where it should be in the input line.

Action: Edit the input file MODELNAME.TXT (containing the textual description of the fault tree) and fix the syntax error in the appropriate line.

```
**** CVRTXT: ERROR F905 - TOO MANY BASIC COMPONENTS, MAX NO. = maxcmpts ****
```

```
     File:        CVRTXT
```

```
     Subroutine:  CVRTXT
```

Meaning: The fault tree described in the MODELNAME.TXT input file contains too many basic component nodes.

Action: Rebuild the FT2MC subsystem with a larger value for the maximum number of basic component nodes allowed; that is, increase the MCMPTS parameter in all FORTRAN source files, recompile, and relink.

```
**** CVRTXT: ERROR F906 - "OF" MISSING ON INPUT LINE line ****
```

```
     File:        CVRTXT
```

```
     Subroutine:  CVRTXT
```

Meaning: A syntax error occurred in an input line read from the MODELNAME.TXT file; the offending line is printed for the user's inspection. The keyword "OF" is not present where it should be in the input line.

Action: Edit the input file MODELNAME.TXT (containing the textual description of the fault tree) and fix the syntax error in the appropriate line.

```
**** CVRTXT: ERROR F907 - "COMPONENT" MISSING ON INPUT LINE line ****
```

```
     File:        CVRTXT
```

```
     Subroutine:  CVRTXT
```

Meaning: A syntax error occurred in an input line read from the modelname.TXT file; the offending line is printed for the user's inspection. The keyword "COMPONENT" is not present where it should be in the input line.

Action: Edit the input file MODELNAME.TXT (containing the textual description of the fault tree) and fix the syntax error in the appropriate line.

```
**** CVRTXT: ERROR F908 - "INPUT" MISSING ON INPUT LINE line ****

     File:        CVRTXT

     Subroutine:  CVRTXT
```

Meaning: A syntax error occurred in an input line read from the MODELNAME.TXT file; the offending line is printed for the user's inspection. The keyword "INPUT" is not present where it should be in the input line.

Action: Edit the input file MODELNAME.TXT (containing the textual description of the fault tree) and fix the syntax error in the appropriate line.

```
**** CVRTXT: ERROR F909 - SOURCE NODE NUMBER MISSING ON INPUT LINE line ****

     File:        CVRTXT

     Subroutine:  CVRTXT
```

Meaning: A syntax error occurred in an input line read from the MODELNAME.TXT file; the offending line is printed for the user's inspection. The number that identifies the node at the source of an input arc is not present where it should be in the input line.

Action: Edit the input file MODELNAME.TXT (containing the textual description of the fault tree) and fix the syntax error in the appropriate line.

```
**** CVRTXT: ERROR F910 - "LABEL" MISSING ON INPUT LINE line ****

     File:        CVRTXT

     Subroutine:  CVRTXT
```

Meaning: A syntax error occurred in an input line read from the MODELNAME.TXT file; the offending line is printed for the user's inspection. The keyword "LABEL" is not present where it should be in the input line.

Action: Edit the input file MODELNAME.TXT (containing the textual description of the fault tree) and fix the syntax error in the appropriate line.

```
**** CVRTXT: ERROR F911 - NUMBER INCOMING ARCS MISSING ON INPUT LINE line ****

     File:        CVRTXT

     Subroutine:  CVRTXT
```

Meaning: A syntax error occurred in an input line read from the MODELNAME.TXT file; the offending line is printed for the user's inspection. The number of incoming arcs for a fault tree node is not present where it should be in the input line.

Action: Edit the input file MODELNAME.TXT (containing the textual description of the fault tree) and fix the syntax error in the appropriate line.

```
**** CVRTXT: ERROR F912 - TOO MANY INCOMING ARCS, MAX NO. = maxarcs ****
     File:        CVRTXT
     Subroutine:  CVRTXT
```

Meaning: The fault tree described in the MODELNAME.TXT input file contains one node that has too many incoming arcs.

Action: Rebuild the FT2MC subsystem with a larger value for the maximum number of incoming arcs per node allowed; that is, increase the MARCS parameter in all FORTRAN source files, recompile, and relink.

```
**** CVRTXT: ERROR F914 - ERROR ENCOUNTERED READING SOURCE OF INCOMING
     ARC arc IN LINE: line ****
     File:        CVRTXT
     Subroutine:  CVRTXT
```

Meaning: The NXTWRD() subroutine encountered an error while reading the number of the node at the source of an incoming arc.

Action: Examine the offending input line in the MODELNAME.TXT input file for syntax errors.

```
**** CVRTXT: ERROR F915 - ILLEGAL TYPE FOR FAULT TREE NODE ---> type ****
     File:        CVRTXT
     Subroutine:  CVRTXT
```

Meaning: A syntax error occurred in an input line read from the MODELNAME.TXT file; the offending line is printed for the user's inspection. The input line specifies a fault tree node whose type is unsupported.

Action: Edit the input file MODELNAME.TXT (containing the textual description of the fault tree) and fix the syntax error in the appropriate line.

```
**** CVRTXT: ERROR F916 - "SYSTEM-FAILURE" BOX NOT SPECIFIED, FAULT TREE
     INCOMPLETE ****
     File:        CVRTXT
     Subroutine:  CVRTXT
```

Meaning: The fault tree description contained in the MODELNAME.TXT file does not include a "System-Failure" box (FBOX) at the top node (root) of the fault tree. The FT2MC subsystem requires all fault trees to have an FBOX or they cannot be processed.

Action: Edit the input file MODELNAME.TXT (containing the textual description of the fault tree) and add an FBOX node at the top of the fault tree.

```
**** CVRTXT: ERROR F917 - ERROR OPENING TEXT DESCRIPTION FILE filename ****
     File:        CVRTXT
     Subroutine:  CVRTXT
```

Meaning: The CVRTXT() subroutine encountered an error while trying to open the text description file. This error is an operating system error rather than an FT2MC error.

Action: Consult the operating system manuals for the cause and possible solutions.

```
**** CVRTXT: ERROR F918 - ERROR OPENING FAULT TREE FILE filename ****
     File:        CVRTXT
     Subroutine:  CVRTXT
```

Meaning: The CVRTXT() subroutine encountered an error while trying to open the fault tree file. This error is an operating system error rather than an FT2MC error.

Action: Consult the operating system manuals for the cause and possible solutions.

```
**** CVRTXT: ERROR F919 - NXTWRD() ENCOUNTERED ERROR err PARSING NEXT
     ITEM, OFFSET = offset ****
     File:        CVRTXT
     Subroutine:  CVRTXT
```

Meaning: The NXTWRD() subroutine encountered an error while parsing the next item on the input line from the text description file. The offset within the input line where the error occurred is printed.

Action: Check the text description file. Edit it if necessary to correct any errors.

```
**** INPTRE: ERROR F920 - FAULT TREE NAME TOO LONG ****
     File:        INPTRE
     Subroutine:  INPTRE
```

Meaning: The modelname passed to INPTRE() was too long and was rejected.

Action: Specify a shorter modelname.

```
**** INPTRE: ERROR F921 - RDDICT() RETURNED ERROR err WHILE TRYING TO
     READ DICTIONARY FILE ****
     File:        INPTRE
     Subroutine:  INPTRE
```

Meaning: The RDDICT() subroutine returned an error while trying to read the dictionary file. Several conditions may cause this message. Look at the error messages that occur immediately before this message to determine the cause.

Action: Report error to the first author of this Technical Paper.

```
**** INPTRE: ERROR F922 - ERROR OPENING DICTIONARY FILE filenams ****
     File:        INPTRE
     Subroutine:  INPTRE
```

Meaning: The INPTRE() subroutine encountered an error while trying to open the dictionary file. This error is an operating system error rather than an INPTRE error.

Action: Consult the operating system manuals for the cause and possible solutions.

```
**** INPTRE: ERROR F923 - ERROR OPENING TEXT DESCRIPTION FILE filename ****

     File:         INPTRE

     Subroutine:   INPTRE
```

Meaning: The INPTRE() subroutine encountered an error while trying to open the text description file. This error is an operating system error rather than an INPTRE error.

Action: Consult the operating system manuals for the cause and possible solutions.

```
**** CVRTXT: ERROR F940 - UNEXPECTED OR INVALID TOKEN (token) ON LINE:

     line

     File:         CVRTXT

     Subroutine:   CVRTXT
```

Meaning: An unexpected or invalid token was detected during the parsing of the indicated line from the .TXT input file.

Action: Check the text description file. Edit it if necessary to correct any error(s).

```
+++ WARNING I20: CANNOT FIND THE PARAMETER ', *************

   IN THE DICTIONARY.  RESULTS MAY BE INCORRECT.

     File:          COVS

     Subroutine:    NEWREP
```

Meaning: The user has specified no repair in the model. However, there are rates in the model not included in the dictionary.

Action: To insure that the results are correct, run the program again and respond yes when questioned about repair.

```
+++ WARNING I110: ASSUMING FIRST STATE ENCOUNTERED IN THE .INT

     FILE TO BE THE INITIAL STATE OF THE MODEL!

     File:          FIFACE

     Subroutine:    MAIN
```

Meaning: The first line of the .INT file reads either SORTED or UNSORTED. The next line begins the actual Markov chain entries of the form STATE1 STATE2 RATE; regardless of the initial FORM type (fault tree or Markov chain), the first state listed (i.e., STATE1) must be the initial state of the system. HARP gives the initial state a probability of 1.0. This warning is only given for UNSORTED input.

Action: If the first state listed is not the initial state, edit the .INT file so that it is the first state.

```
+++ WARNING I120: ONLY 96 COMPONENTS ALLOWED ALL OTHERS ARE IGNORED'

     files:         FIFACE

     Subroutine:    READIC
```

Meaning: Only 96 components are acknowledged by HARP. Any additional component types will not be read into the data structure.

Action: If the model has more than 96 component types, HARP will not run. The model must be reconstructed with fewer component types.

```
+++ WARNING I130:  ************** IS GREATER THAN 12 CHARACTERS
   IT WILL BE TRUNCATED TO: ************
     files:        FIFACE
     Subroutine:   READIC
```

Meaning: Component word length in the dictionary is restricted to 12 characters. This restriction does not affect the outcome of the program.

```
+++ WARNING I135: EXTRANEOUS LINE FOUND AT END OF DICTIONARY FILE: line
     files:        FIFACE
     Subroutine:   READIC
```

Meaning: At the bottom of the dictionary file are listed the state id numbers of all the FEn (failure due to exhaustion) states and/or the TAn (truncation aggregation) states. There should be no other lines of data after these state id numbers. If any other lines are found following them, this warning message is printed.

Action: If the model has more than 96 component types, HARP will not run. The model must be reconstructed with fewer component types.

```
+++ WARNING I150 - CAN''T PARSE *******, NCF RATES MAY NOT BE CONSERVATIVE
     File:         PARSE, SUMCOF
     Subroutine:   PARSE, ALLSET
```

Meaning: The string passed to the parser cannot be converted to a numerical value. Therefore, the NCF rates will be calculated only by the arcs emanating from the target state and may not be conservative.

Action: To insure conservative rates, rates should be of the form coefficient*dictionary_rate. A single rate may also be a numerical value or a repair rate.

```
+++ WARNING I170 - CAN''T FIND OVERRIDING FEHM FILENAME FOR LINE ,WHERE
     File:         NXTFLT
     Subroutine:   HIRFND
```

Meaning: HARP cannot find the overriding FEHM file that was declared in the .INT file.

Action: Send copies of all input files along with the version of the program to the first author of this Technical Paper.

```
**** WARNING I190:  USE OF OVERRIDING FEHMs
     File:        COVS
     Subroutine:  TOADD
```

Meaning: Overriding FEHM's are being used in a model where there may be more than one rate going from the same source state to the same destination state on the same input line. The program *fiface* cannot accommodate this situation. If there is only one rate, continue by answering yes when prompted to continue.

Action: List the rates explicitly for each transition as follows:

```
YES: 1  2  3*LAM:NEW;
     1  2  2*MU:OLD;
 NO: 1  2  3*LAM:NEW+2*MU:OLD;
```

```
*** ERROR I510: ALLOWABLE NUMBER OF TRANSITIONS
   EXCEEDED FOR SORTED INPUT
     File:         LD
     Subroutine:   LDSORT
```

Meaning: The number of allowable transitions for the program has been exceeded. This number, TRSIZ, is originally set to 10 000.

Action: Change the value of TRSIZ in routine INITSZ in fiface.for, recreate the object module, and recompile the program.

```
*** ERROR I511: ALLOWABLE NUMBER OF TRANSITIONS
   EXCEEDED FOR UNSORTED OR SYMBOLIC INPUT
     File:         LD
     Subroutine:   LODFIL
```

Meaning: The number of allowable transitions for unsorted input has been exceeded. This number, MCTRZ, is originally set to 2050.

Action: Change the value of MCTRZ in routine INITSZ in fiface.for, recreate the object module, and recompile the program. Note, it would be better to sort the input (or use numeric rather than symbolic input) to reduce the run time.

```
*** ERROR I520: STATE SIZE EXCEEDED FOR SORTED INPUT
     File:         LD
     Subroutine:   LDSORT
```

Meaning: The number of allowable states for the program has been exceeded. This number, STSIZ, is originally set to 1000.

Action: Change the value of STSIZ in routine INITSZ in fiface.for, recreate the object module, and recompile the program.

```
*** ERROR I521: STATE SIZE EXCEEDED FOR
   UNSORTED OR SYMBOLIC INPUT
     File:         LD
     Subroutine:   STNUM
```

Meaning: The number of allowable states for unsorted input has been exceeded. This number, MCSTZ, is originally set to 500.

Action: Change the value of MCSTZ in routine INITSZ in fiface.for, recreate the object module, and recompile the program. Note, it would be better to sort the input (or use numeric rather than symbolic input) to reduce the run time.

```
*** ERROR I530: PARAMETER EXCEEDS ALLOWABLE'
   SIZE OF ',PSIZE-9, '  TRUNCATING.'
     File:          LD
     Subroutine:    LDPARM
```

Meaning: The variable PSIZE, set in routine INITSZ of file fiface.for has been exceeded. The rate parameter is set to 32 with 9 characters being allowed for the coefficient, 12 for the rate symbol, and the rest for coverage and the multiplicative signs and semicolon (i.e., 1.2345678*RATPARAMETER*C1234567;). The program actually allows only PSIZE-9, so it can add the coverage factor without exceeding the size of 32.

Action: The value of PSIZE can be changed in routine INITSZ. However, the HARP engine also has a hard limit of 13 characters for the rate symbol. Also, change the data structure for PARMS and MCPARM (currently set to 32).

```
***  FIFACE:  ERROR I610 - MISSING KEYWORD "UNSORTED" OR "SORTED"
   IN THE .INT FILE. SHOULD BE THE FIRST LINE.
     File:          FIFACE
     Subroutine:    MAIN
```

Meaning: The first line of the .INT file must be one of two keywords: UNSORTED or SORTED.

Action: Edit the .INT file so that the first line reads SORTED if the .INT file is a converted fault tree or a Markov chain without symbolic input and in row-wise order or UNSORTED if it is a Markov chain with symbolic input or not in row-wise order.

```
*** READIC:  ERROR I620 - ERROR IN DICTIONARY FILE LINE:
     File:          FIFACE
     Subroutine:    READIC
```

Meaning: There must be four entries on the dictionary line in the following format:

```
1 COMPONENT          RATE          FEHM
```

The program lists the offending line that has either fewer than or more than 4 entries. FEHM may be a filename or the keyword NONE or VALUES.

Action: Edit the .DIC file so that it conforms to the above rules.

```
*** RDIDS: ERROR I625 - NUMBER OF FEIDS/TAIDS DOES NOT MATCH
   THE NUMBER OF COMPONENT TYPES IN THE DICTIONARY
     File:          FIFACE
     Subroutine:    RDIDS
```

Meaning: At the bottom of the dictionary file are listed the state id numbers of all FEn (failure due to exhaustion) states and/or TAn (truncation aggregation) states. Since there can be one FE/TA state for each type of component in the system, the number of FEn and/or TAn state ids should be the same as the number of dictionary entries (which define the types of components in the system). If there are not the same number of FEn or TAn state ids listed than as component types defined in the dictionary file, this error message is produced.

Action: The .DIC file is probably corrupted. Rerun *tdrive* to recreate it.

```
*** ERROR I710: A ZERO ROW IN ROUTINE ORDER
     File:          TRANSPOSE
     Subroutine:    ORDER
```

Meaning: During the transposition, the routine was trying to set the zeroth entry of an array.

Action: Check the input file for an error.

```
*** ERROR I725: ERROR IN EXPRESSION, CHAR
     File:          PARSE
     Subroutine:    TOCNVT
```

Meaning: A character has been found that should not be in the expression.

Action: Edit the input file to remove the offending sequence.

```
*** ERROR I730: ILLEGAL CHARACTER, ****,  IN EXPRESSION
     File:          PARSE
     Subroutine:    OPRTOR
```

Meaning: A character has been found that should not be in the expression.

Action: Edit the input file to remove the offending sequence.

```
*** ERROR I735: STACK IS EMPTY
     File:          PARSE
     Subroutine:    POP
```

Meaning: During parsing an attempt has been made to pop a value off the empty stack.

Action: If a reason for the error cannot be found, then send a copy of the input files and version number to the first author of this Technical Paper.

```
++++ CONVRT: WARNING U100 - CHAR WORD TO BE CONVERTED TO NUMERIC
       CONTAINS NO DIGITS
     File:         TFHUTL
     Subroutine:   CONVRT
```

Meaning: CONVRT() converts a character string representation of a real number into its numeric data type representation. If the character string that is purported to contain a real

number in fact contains no digits, this warning is produced. This error probably represents an internal programming error in *tdrive*.

Action: In the calling routine, check for a nonnumeric character string being passed to CONVRT().

```
++++ CONVRT: WARNING U101 - EXPONENT CONTAINS NO DIGITS ---> string

     File:        TFHUTL

     Subroutine:  CONVRT
```

Meaning: CONVRT() converts a character string representation of a real number into its numeric data type representation. If the character string that is purported to contain a real number contains exponential notation and the exponent contains no digits, this warning message is produced.

Action: In the calling routine, check for an invalid numeric character string being passed to CONVRT().

```
++++ CONVRT: WARNING U102 - INVALID CHARACTERS: characters DETECTED DURING

       CONVERSION OF string FROM CHAR TO NUMERIC

     File:        TFHUTL

     Subroutine:  CONVRT
```

Meaning: CONVRT() converts a character string representation of a real number into its numeric data type representation. If the character string that is purported to contain a real number contains nonnumeric characters, this warning message is produced.

Action: In the calling routine, check for an invalid numeric character string being passed to CONVRT().

```
++++ DBCHR:  WARNING U104 - ... ROUNDING TO ZERO

     File:        TFHUTL

     Subroutine:  DBCHR
```

Meaning: DBCHR converts a double precision number (RNUM) to a character representation. To convert the decimal portion of the number, at each iteration the number (RNUM) is multiplied by 10 and the integer value subtracted (RNUM = RNUM − INT(RNUM)). Once RNUM is less than EPSIL, RNUM is rounded to zero.

Action: If this is not satisfactory, change the value of EPSIL. It is initially set to 1.0e-6.

```
**** ALLOC: ERROR U400 - BOUNDS OF MEMORY POOL EXCEEDED

     File:        DYNMEM

     Subroutine:  ALLOC,PSHPTR
```

Meaning: ALLOC() allocates regions of a large buffer array for use in linked list type applications (emulates a simple dynamic memory facility). If an attempt is made to allocate space beyond the end of the buffer array, this error message is produced.

Action: Increase the size of the buffer array in the calling routine where it is defined (in this case, increase the MPLEN or DMPLEN parameter, whichever applies, in FT2MC()) and recompile the entire program.

```
**** POPPTR: ERROR U401 - STACK UNDERFLOW
     File:        DYNMEM
     Subroutine:  POPPTR
```

Meaning: An underflow situation (stack is empty) was encountered while trying to pop an item from a stack. This error represents an internal programming error in the *tdrive* program.

Action: Report error to the first author of this Technical Paper.

```
**** GETNOD:  ERROR U402 - NODSIZ nodesize IS > MAX NODE SIZE max_nodesize
     File:        DYNMEM
     Subroutine:  GETNOD,DSPNOD
```

Meaning: While requesting the allocation of memory for a node from the dynamic memory emulation routines, the caller has requested a node size greater than the declared maximum allowable node size. This error represents an internal programming error in the *tdrive* program.

Action: Report error to the first author of this Technical Paper.

```
**** GETLIN: ERROR U500 - INPUT LINE TOO LONG, MUST BE <= vldlen CHAR'S
     File:        TFHUTL
     Subroutine:  GETLIN
```

Meaning: GETLIN() reads a line from an input file and checks that the line is not longer than a certain valid length. If such a line is longer than the program is expecting, the program simply truncates the input line to the valid length, losing some of the input. When this occurs, this error message is produced.

Action: Ensure that the input file contains data of the proper format expected by the program.

```
**** GETLIN: ERROR U501 - ERROR ENCOUNTERED READING INPUT LINE FROM FILE
     File:        TFHUTL
     Subroutine:  GETLIN
```

Meaning: GETLIN() encountered a read error while trying to read a line from an input file. This error represents an operating system error rather than a GETLIN() error.

Action: Consult the operating system manuals for the cause and possible solutions.

```
**** ADD2Q: ERROR U601 - QUEUE OVERFLOW ****
     File:        QUEUE
     Subroutine:  ADD2Q
```

Meaning: A queue overflow occurred during an attempt to add a node to a queue that is already full.

Action: In the calling routine, check for an infinite loop if the queue length is large enough to hold the queue contents. If the queue really is not large enough to hold it contents, then increase the size of the queue array.

```
**** POPQ: ERROR U602 - QUEUE UNDERFLOW ****
```

```
     File:        QUEUE
```

```
     Subroutine:  POPQ
```

Meaning: A queue underflow occurred during an attempt to remove a node from a queue that is already empty.

Action: In the calling routine, check for a programming error.

```
**** WRTQ: ERROR U603 - EXCEEDED BOUNDS OF QUEUE ARRAY ****
```

```
     File:        QUEUE
```

```
     Subroutine:  WRTQ
```

Meaning: While attempting to copy a queue node into the queue array, the bounds of the queue array were exceeded before the entire node was copied. This error indicates an internal programming error in the queue package.

Action: Report error to the author of the queue package.

```
**** PUSH: ERROR U701 - STACK OVERFLOW ****
```

```
     File:        STACK
```

```
     Subroutine:  PUSH
```

Meaning: A stack overflow occurred during an attempt to add a node to a stack that is already full.

Action: In the calling routine, check for an infinite loop if the stack length is large enough to hold the stack contents. If the stack really is not large enough to hold it contents, increase the size of the stack array.

```
**** POP: ERROR U702 - STACK UNDERFLOW ****
```

```
     File:        STACK
```

```
     Subroutine:  POP
```

Meaning: A stack underflow occurred during an attempt to remove a node from a stack that is already empty.

Action: In the calling routine, check for a programming error.

```
*** NXTWRD: ERROR U801 - ERROR PARSING NUMERIC VALUE (token) IN LINE: line ***
```

```
     File:        UTIL
```

```
     Subroutine:  NXTWRD
```

Meaning: An inappropriate character was found in what is supposed to be a numeric token while parsing that token from an input line (e.g., if an alphabetic character appears in what is

supposed to be a real or integer number). This error indicates either an error in the data in the input line or a programming error in the calling routine.

Action: Check the data on the input line passed to NXTWRD to make sure it is correct. Then check the calling routine to be sure it is looking for the correct type of token in the input line.

```
**** NXTWRD: ERROR U802 - ILLEGAL FUNCTION ---> f ****
```

```
File:        UTIL
```

```
Subroutine:  NXTWRD
```

Meaning: An illegal value was passed to NXTWRD in the FUNCT argument. FUNCT indicates whether NXTWRD should look for a numeric value (FUNCT = 'N') or a character value (FUNCT = 'C') for the next token on the input line. Any other value for FUNCT is not supported.

Action: In the calling routine, correct the FUNCT argument in the call to NXTWRD. FUNCT must be either 'N' or 'C'.

```
**** OPRNDS: ERROR U803 - OPERATOR op NOT FOUND IN WORD word ****
```

```
File:        UTIL
```

```
Subroutine: OPRNDS
```

Meaning: The character specified as the operator character in a binary operator expression was not found in the expression.

Action: In the calling routine, check the value of the binary operator expression passed to OPRNDS.

```
**** OPRNDS: ERROR U804 - ILLEGAL FORM FOR BINARY OPERATOR EXPRESSION ****
```

```
File:        UTIL
```

```
Subroutine:  OPRNDS
```

Meaning: The binary operator expression passed to OPRNDS did not have the form: OPERAND1 op OPERAND2

Action: In the calling routine, check the value of the binary operator expression passed to OPRNDS.

```
**** INTCHR: ERROR U805 - NUMBER OVERFLOWS CHAR STRING ****
```

```
File:        TFHUTL
```

```
Subroutine:  INTCHR
```

Meaning: The character representation of the integer value passed to INTCHR is too long to fit in the output character variable provided to hold it.

Action: In the calling routine, provide a longer character variable to receive the converted numeric value.

```
**** DBCHR : ERROR U806 - NUMBER OVERFLOWS CHAR STRING ****
     File:        TFHUTL
     Subroutine:  INTCHR
```

Meaning: The character representation of the double value passed to DBCHR is too long to fit in the output character variable provided to hold it.

Action: In the calling routine, provide a longer character variable to receive the converted numeric value.

```
**** DBCHR: ERROR U807 - MAXLEN TOO SMALL FOR ROUTINE ****
     File:        TFHUTL
     Subroutine:  DBCHR
```

Meaning: The length of the character array is too small for DBCHR. DBCHR concatenates two arrays, each of size 10—one on each side of a decimal point.

Action: Set value of MAXLEN in calling routine to 21.

```
**** SKPICT: ERROR U900 - UNEXPECTED END-OF-FILE ENCOUNTERED WHILE READING
     DICTIONARY FILE
     File:        DICUTL
     Subroutine:  SKPICT
```

Meaning: While the dictionary file was being read, an EOF was encountered before it should have been. The dictionary file is probably corrupted.

Action: Rerun *tdrive* and recreate the dictionary file.

References

1. Bavuso, Salvatore J.: Hybrid Automated Reliability Predictor Integrated Work Station (HiRel). *Technology 2001: The Second National Technology Transfer Conference and Exposition*, Volume 1, NASA CP-3136, Jan. 1991, pp. 385–394.

2. Bavuso, Salvatore J.; and Dugan, JoAnne B.: HiRel—Reliability/Availability Integrated Workstation Tool. *Proceedings of the Annual Reliability and Maintainability Symposium*, IEEE, 1992, pp. 491–500.

3. Platt, M. E.; Lewis, E. E.; and Boehm, F.: *General Monte Carlo Reliability Simulation Code Including Common Mode Failures and HARP Fault/Error-Handling.* NASA CR-187587, 1991.

4. Somani, A. K.; Ritcey, J.; and Au, S.: Computationally-Efficient Phased-Mission Reliability Analysis for Systems With Variable Configurations. *IEEE Trans. Reliab.*, vol. 41, no. 4, Dec. 1992, pp. 504–511.

5. Geist, Robert: Extended Behavioral Decomposition for Estimating Ultrahigh Reliability. *IEEE Trans. Reliab.*, vol. 40, Apr. 1991, pp. 22–28.

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

7. Trivedi, Kishor S.; and Geist, Robert M.: Decomposition in Reliability Analysis of Fault-Tolerant Systems. *IEEE Trans. Reliab.*, vol. 32, no. 5, Dec. 1983, pp. 463–468.

8. McGough, J.; Smotherman, M.; and Trivedi, K. S.: The Conservativeness of Reliability Estimates Based on Instantaneous Coverage. *IEEE Trans. Comput.*, vol. 34, no. 7, July 1985, pp. 602–609.

9. Smotherman, Mark; Geist, Robert M.; and Trivedi, Kishor S.: Provably Conservative Approximations to Complex Reliability Models. *IEEE Trans. Comput.*, vol. 35, no. 4, Apr. 1986, pp. 333–338.

10. S. G. Corps: A320 Flight Controls. *The 1985 Report to the Aerospace Profession*, Soc. Exp. Test Pilots, Sept. 1985, pp. 196–210.

11. Harschburger, H. E.; Glaser, B.; and Hammel, J. R.: Backup Modes for the F/A-18 Digital Flight Control System. *Proceedings of the 6th Digital Avionics Systems Conference*, AIAA, 1984, pp. 108–115. (Available as AIAA-84-2622.)

12. Devlin, B. T.; and Girts, R. D.: MD-11 Automatic Flight System. *IEEE Aerosp. & Electron. Syst. Mag.*, vol. 8, no. 3, Mar. 1993, pp. 53–56.

13. Kowal, B. W.; Scherz, C. J.; Quinlivan, R.: C-17 Flight Control System Overview. *Proceedings of the IEEE 1992 National Aerospace & Electronics Conference*, IEEE, vol. 2, 1992, pp. 829–835.

14. Dugan, JoAnne Bechta: Automated Analysis of Phased-Mission Reliability. *IEEE Trans. Reliab.*, vol. 40, no. 1, Apr. 1991, pp. 45–52, 55.

15. Esary, J. D.; and Ziems, H.: *Reliability Analysis of Phased Missions.* NPS-55EY-75021, U.S. Navy, Feb. 1975. (Available from DTIC as AD A008 290.)

16. Pedar, A.; and Sarma, V. V. S.: Phased-Mission Analysis for Evaluating the Effectiveness of Aerospace Computing-Systems. *IEEE Trans. Reliab.*, vol. 30, Dec. 1981, pp. 429–437.

17. Clark, C. E.: Importance Sampling in Monte Carlo Analyses. *Oper. Res.*, vol. 9, no. 5, Sept.–Oct. 1961, pp. 603–620.

18. Boyd, Mark A.; and Bavuso, Salvatore J.: Modeling a Highly Reliable Fault-Tolerant Guidance, Navigation, and Control System for Long Duration Manned Spacecraft. *IEEE/AIAA 11th Digital Avionic Systems Conference*, 1992, pp. 464–469.

19. Boyd, Mark A.; and Bavuso, Salvatore J.: Simulation Modeling for Long Duration Spacecraft Control Systems. *Proceedings of the 1993 Annual Reliability and Maintainability Symposium*, IEEE, 1993.

20. Siewiorek, Daniel P.; and Swarz, Robert S.: *The Theory and Practice of Reliable System Design.* Digital Press, 1982.

21. Hecht, Herbert; and Hecht, Myron: Reliability Prediction for Spacecraft. RADC-TR-85-229, U.S. Air Force, Dec. 1985. (Available from DTIC as AD A164 747.)

22. Don't Take This Reliability Model as Gospel. *High Perform. Sys.*, Nov. 1989, p. 22.

23. Bjurman, B. E.; Jenkins, G. M.; Masreliez, C. J.; McClellan, K. L.; and Templeman, J. E.: *Airborne Advanced Reconfigurable Computer System (ARCS)—Final Report, Mar. 1975–Apr. 1976.* NASA CR-145024, 1976.

24. Bavuso, S. J.: *CARE III Hands-On Demonstration and Tutorial.* NASA TM-85811, 1984.

25. Baker, Robert; and Scheper, Charlotte: *Evaluation of Reliability Modeling Tools for Advanced Fault Tolerant Systems.* NASA CR-178067, 1986.

26. Bavuso, Salvatore J.; Dugan, JoAnne Bechta; Trivedi, Kishor; Rothmann, Beth; and Boyd, Mark: *Applications of the Hybrid Automated Reliability Predictor.* NASA TP-2760, 1987.

27. Dugan, JoAnne Bechta; Bavuso, S. J.; and Boyd, M. A.: Modeling Advanced Fault-Tolerant Systems With HARP. Topics in Reliability & Maintainability & Statistics, Consolidated Lecture Notes—1991 "Tutorial Notes," *Proceedings of the Annual Reliability and Maintainability Symposium*, IEEE, 1991, pp. FTS-1–FTS-25.

28. Dugan, JoAnne B.: Fault Trees and Imperfect Coverage. *IEEE Trans. Reliab.*, vol. 38, June 1989, pp. 177–185.

29. Dugan, JoAnne Bechta; Bavuso, Salvatore J.; and Boyd, Mark A.: Fault Trees and Sequence Dependencies. *Proceedings of the Annual Reliability and Maintainability Symposium—1990*, IEEE, 1990, pp. 286–293.

30. Dugan, JoAnne Bechta; Bavuso, Salvatore J.; and Boyd, Mark A.: Fault Trees and Markov Models for Reliability Analysis of Fault-Tolerant Digital Systems. *Reliab. Eng. & Sys. Safety*, vol. 39, no. 3, 1993, pp. 291–307.

31. Veeraraghavan, Malathi: Modeling and Evaluation of Fault-Tolerant Multiple Processor System. Ph.D. Diss., Duke Univ., 1988.

32. Dugan, JoAnne Bechta; Boyd, Mark A.; and Bavuso, Salvatore J.: Fault Trees and Sequence Dependencies. *Proceedings of the 1990 Annual Reliability and Maintainability Symposium*, IEEE, 1990, pp. 286–293.

33. Barlow, R. E.; and Lambert, H. E.: Introduction to Fault Tree Analysis. *Reliability and Fault Tree Analysis—Theoretical and Applied Aspects of System Reliability and Safety Assessment.* Richard E. Barlow, Jerry B. Fussell, and Nozer D. Singpurwalla, eds., Soc. of Ind. & Appl. Math., 1975, pp. 7–35.

34. Bavuso, Salvatore J.; Dugan, JoAnne Bechta; Trivedi, Kishor S.; Rothmann, Elizabeth M.; and Smith, W. Earl: Analysis of Typical Fault-Tolerant Architectures Using HARP. *IEEE Trans. Reliab.*, vol. 36, June 1987, pp. 176–185.

35. Dugan, JoAnne Bechta; Veeraraghavan, Malathi; Boyd, Mark; and Mittal, Nitin: Bounded Approximate Reliability Models for Distributed Systems. *Proceedings of the 8th Symposium on Reliable Distributed Systems*, IEEE Comput. Soc. Press, Oct. 1989, pp. 1–20.

36. Boyd, Mark Allen: Fault Tree Models—Techniques for Analysis of Advanced Fault Tolerant Computer Systems. Ph.D. Diss., Duke Univ., 1991.

37. McGough, J.: Effects of Near-Coincident Faults in Multiprocessor Systems—Reliability Engineering for Commercial Aircraft Flight Control. *Proceedings of the 5th Digital Avionics Systems Conference*, IEEE, 1983, pp. 16.6.1–16.6.7.

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

39. Ng, Ying-Wah; and Avižienis, Algirdas: A Model for Transient and Permanent Fault Recovery in Closed Fault-Tolerant Systems. *Proceedings—FTCS-6*, IEEE, 1976, pp. 182–188.

40. Stiffler, J. J.; and Bryant, L. A.: *CARE III Phase II Report—Mathematical Description.* NASA CR-3566, 1982.

41. Trivedi, Kishor; and Dugan, JoAnne Bechta: Hybrid Reliability Modeling of Fault-Tolerant Computer Systems. *Comput. & Electr. Eng.*, vol. 11, no. 2/3, 1984, pp. 87–108.

42. Geist, Robert; Trivedi, Kishor; Dugan, JoAnne Bechta; and Smotherman, Mark: Design of the Hybrid Automated Reliability Predictor. *Proceedings of the 5th Digital Avionics Systems Conference*, IEEE, 1983, pp. 16.5.1–16.5.8.

43. Geist, Robert; Trivedi, K. S.; Dugan, JoAnne Bechta; and Smotherman, Mark: Modeling Imperfect Coverage in Fault Tolerant Systems. *The 14th International Conference on Fault-Tolerant Computing—Digest of Papers*, IEEE Comput. Press, 1984, pp. 77–82.

44. Trivedi, Kishor Shridharbhai: *Probability and Statistics With Reliability, Queuing, and Computer Science Applications.* Prentice-Hall, 1982.

45. Bavuso, Salvatore J.; and Martensen, Anna L.: A Fourth Generation Reliability Predictor. *Proceedings of the 1988 Annual Reliability and Maintainability Symposium*, 1988, pp. 11–16.

46. Cox, D. R.: A Use of Complex Probabilities in the Theory of Stochastic Processes. *Proceedings of the Cambridge Philosophical Soc.*, vol. 51, 1955, pp. 313–319.

47. Costes, A.; Doucet, J. E.; Landrault, C.; and Laprie, J. C.: SURF—A Program for Dependability Evaluation of Complex Fault-Tolerant Computing Systems. *The 11th Annual International Symposium on Fault-Tolerant Computing*, IEEE, 1981, pp. 72–78.

48. Boyd, Mark A.; Veeraraghavan, Malathi; Dugan, JoAnne Bechta; and Trivedi, Kishor S.: An Approach to Solving Large Reliability Models. AIAA-88-3905, 1988.

49. Boyd, Mark A.: Converting Fault Trees to Markov Chains for Reliability Prediction. M.S. Thesis, Duke Univ., 1986.

50. Shampine, L. F.; and Watts, H. A.: Global Error Estimation for Ordinary Differential Equations. *ACM Trans. Math. Softw.*, vol. 2, no. 2, June 1976, pp. 172–186.

51. Reibman, Andrew; and Trivedi, Kishor: Numerical Transient Analysis of Markov Models. *Comput. Oper. Res.*, vol. 15, no. 1, 1988, pp. 19–36.

52. Smotherman, Mark Kelly: Parametric Error Analysis and Coverage Approximations in Reliability Modeling (Sensitivity). Ph.D. Diss., Univ. of North Carolina, 1984.

53. Smotherman, Mark: Error Analysis in Analytic Reliability Modeling. *Microelectron. Reliab.*, vol. 30, no. 1, 1989, pp. 141–149.

54. Fussell, J. B.; Aber, E. F.; and Rahl, R. G.: On the Quantitative Analysis of Priority-AND Failure Logic. *IEEE Trans. Reliab.*, vol. 25, no. 5, Dec. 1976, pp. 324–326.

55. Henley, Ernest J.; and Kumamoto, Hiromitsu: *Reliability Engineering and Risk Assessment.* Prentice-Hall 1981.

GLOSSARY

Most terms unique to reliability modeling and fault-tolerant systems are defined within the body of each volume of this Technical Paper. The meaning of some terms are well known to researchers and users of these technologies but may not be familiar to new users of Hybrid Automated Reliability Predictor (HARP) integrated reliability (HiRel) tool system. Thus, the purpose of this glossary is to primarily aid new users.

Availability

Availability is a probabilistic quantity that predicts the operational life of a system that is subject to line maintenance (repair). Availability is the probability that a system under repair is operational at a specified time. In a Markov chain model representation, repair is modeled by adding transitions from states with $n+1$ failed components to states with n components. The transition rate is given as a repair rate. No fault tree model representation has yet been developed to represent an availability model; therefore, a Markov chain model must be given to HARP for solution. A fault tree model can be used to specify and generate a preliminary Markov chain model that the user needs to modify.

Behavioral Decomposition

Behavioral decomposition is a mathematical approximation technique that reduces a complex fault/error handling model (FEHM) to a branch point in a Markov chain. The effects of the FEHM are compensated for by modifying state transition rates. The advantage of this technique is that it greatly reduces the size of Markov models for solution and complex FEHM behavior that can be non-Markovian modeled.

Bounds or Mathematical Bounds

Large or complex mathematical models often require approximations to keep their solutions tractable. Bounds are the numerical expressions of the variation in a computed result due to mathematical approximation or uncertainty in the accuracy of the input data to the models.

Combinatorial Model

A combinatorial model is a stochastic model that relates combinatorial component failure or success events to a subsystem or system failure or success, respectively. Combinatorial models do not distinguish the order of failure events.

Coincident Fault

A coincident fault exists at the same time one or more other faults are present. A coincident fault is not a simultaneous fault.

Conservative Unreliability Result

Mathematical quantities can be expressed in two forms, in exact form, which is usually a symbolic representation such as the symbol π, or in an approximate form such as a decimal representation for π as 3.14159. When approximations are necessary, the difference between the exact quantity (which may not be obtainable) and the computed result (which is obtainable) is called the error. A conservative unreliability result is one where the error in the computed result is in the direction of increased unreliability.

Critical-Pair Fault

A critical fault is a near-coincident fault involving two faults. HARP uses three multifault models to account for critical-pair faults: ALL, SAME, and USER.

Extended Behavioral Decomposition

Extended behavioral decomposition is a generalized behavioral decomposition technique that allows multiple FEHM entry/exit transitions and multifault near-coincident modeling.

Fault Tree

A fault tree is a notational model that uses symbols resembling logic gates that relates failure events of components or subsystems to failure events of a system composed of components and subsystems.

Instantaneous Jump Model

An instantaneous jump model is a Markov model that is an approximation of a more complex semi-Markov model that produces a conservative result with respect to the semi-Markov model that is operated on mathematically to become the instantaneous jump model.

Multifault Model

A multifault model is a fault/error handling model that accounts for two or more faults, none occurring simultaneously.

Near-Coincident Fault

A near-coincident fault is second fault that occurs during the time between the occurrence of a first fault and its recovery.

Near-Coincident Failure

A near-coincident failure is system failure resulting from a near-coincident fault. To reduce modeling complexity, a near-coincident failure is assumed to result from a near-coincident fault. Typically, this assumption results in a conservative result.

Optimistic Unreliability Result

An optimistic unreliability result occurs when the error in the computed result is in the direction of decreased unreliability.

Primitive

A primitive is any screen image that is an entity that can be manipulated without dissection, for example, a line, a circle, a fault tree gate, etc.

Semi-Markov Models

Semi-Markov models are generalizations of Markov models. In particular, semi-Markov models allow generalized state holding time distributions. Semi-Markov models are required

for fault-tolerant system models to account for fault/error handling times that may not be exponential.

Sequence-Dependent Model

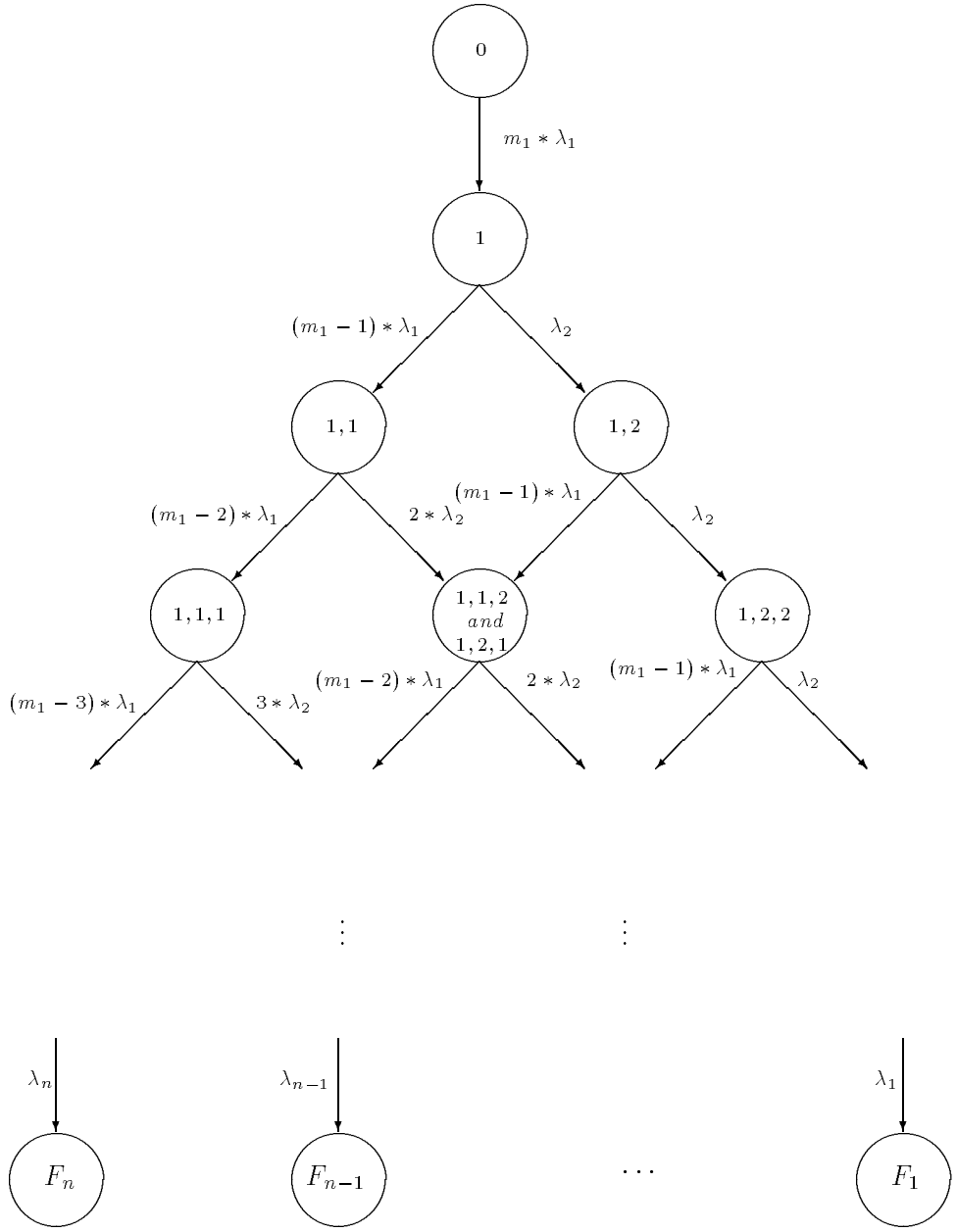
A sequence-dependent model is a stochastic model that relates ordered component failure or success events to a subsystem or system failure or success, respectively. Sequence-dependent models distinguish the order of failure events. These models are more complex than combinatorial models and are also more difficult to solve.

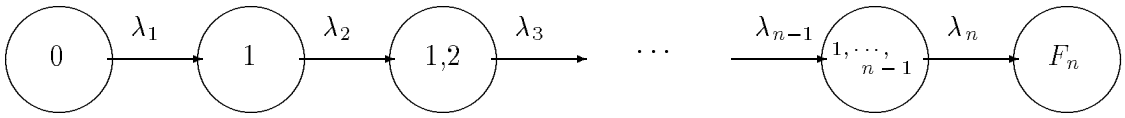
Simultaneous Fault

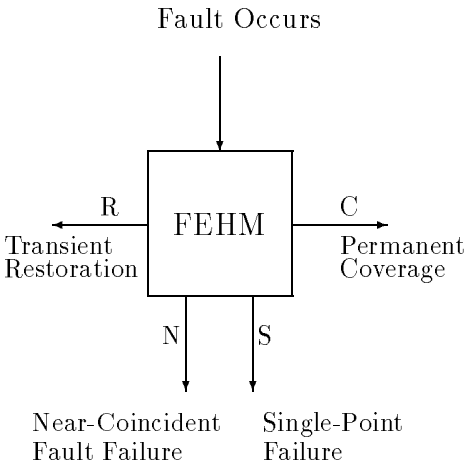
A simultaneous fault is second fault that occurs at exactly the same instant in time as a first fault. Markov chain models do not allow such faults.

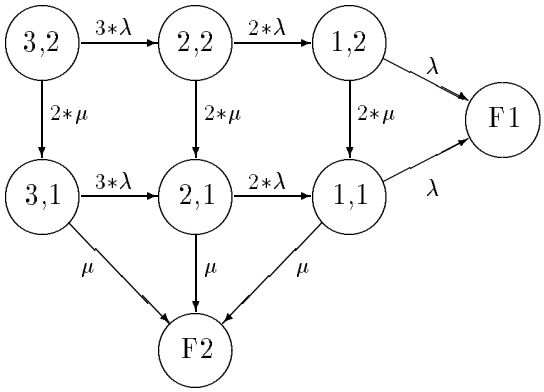
Weibull Distribution

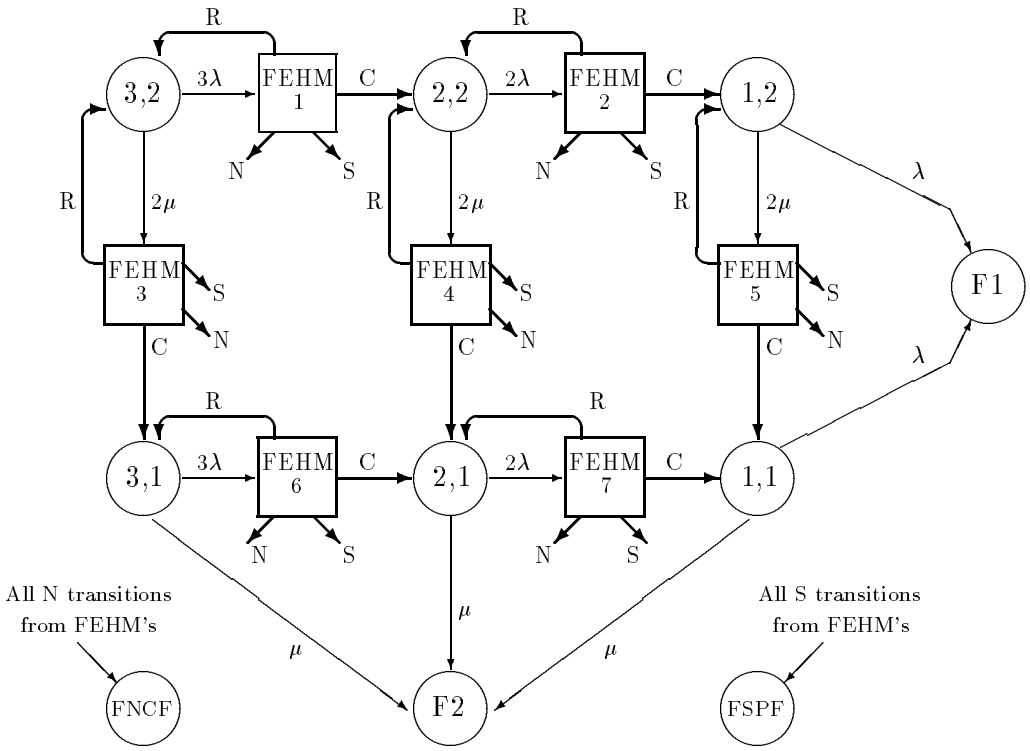
A Weibull distribution is a two parameter distribution that can exhibit time increasing, decreasing, or constant failure rates.

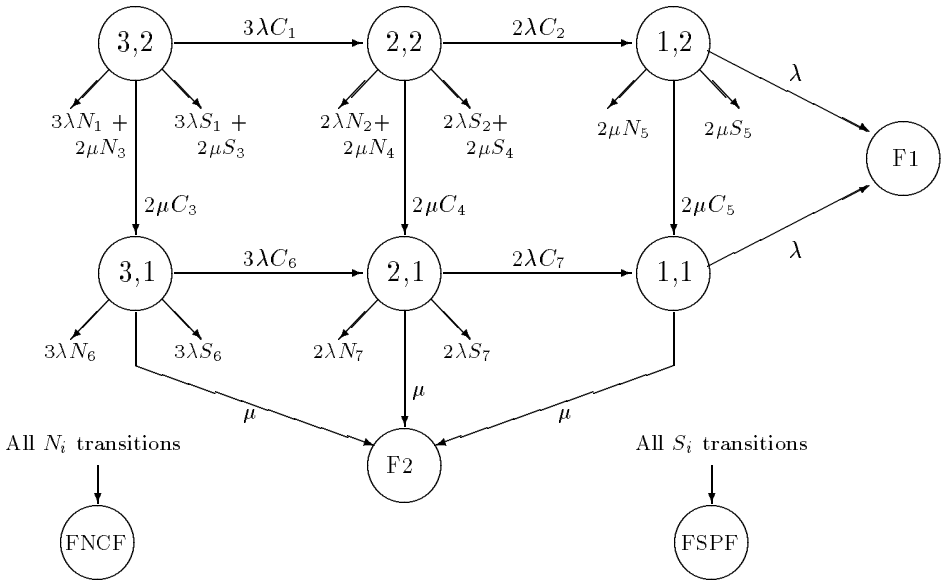

0
$m_1 * \lambda_1$
1
$(m_1 - 1) * \lambda_1$
λ_2
1, 1
1, 2
$(m_1 - 2) * \lambda_1$
$2 * \lambda_2$
$(m_1 - 1) * \lambda_1$
λ_2
1, 1, 1
1, 1, 2 and 1, 2, 1
1, 2, 2
$(m_1 - 3) * \lambda_1$
$3 * \lambda_2$
$(m_1 - 2) * \lambda_1$
$2 * \lambda_2$
$(m_1 - 1) * \lambda_1$
λ_2
λ_n
λ_{n-1}
λ_1
F_n
F_{n-1}
F_1

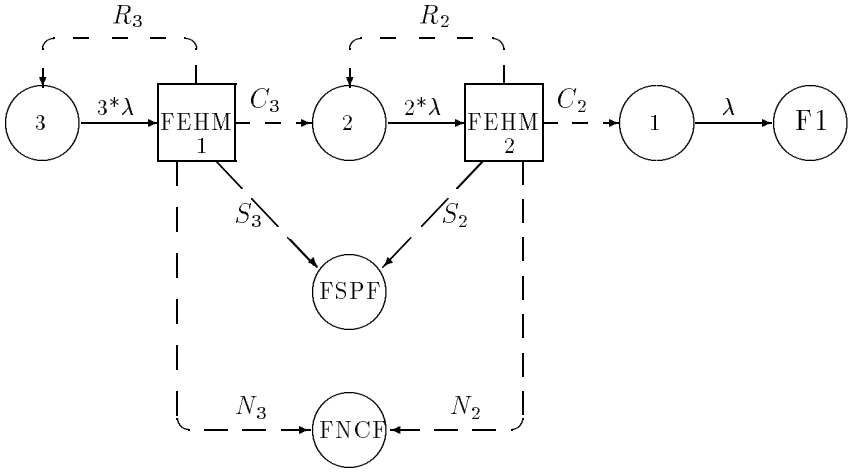

0
λ_1
1
λ_2
1,2
λ_3
$\cdots$
λ_{n-1}
$1, \cdots, n-1$
λ_n
F_n

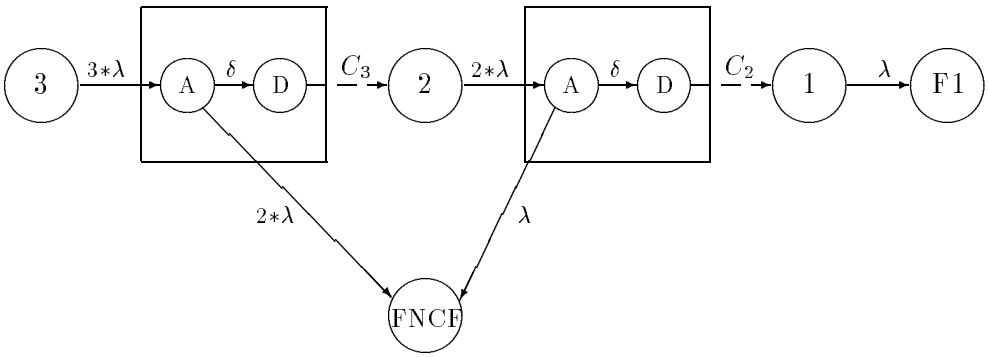

Fault Occurs
FEHM
R
Transient
Restoration
C
Permanent
Coverage
N
S
Near-Coincident
Fault Failure
Single-Point
Failure

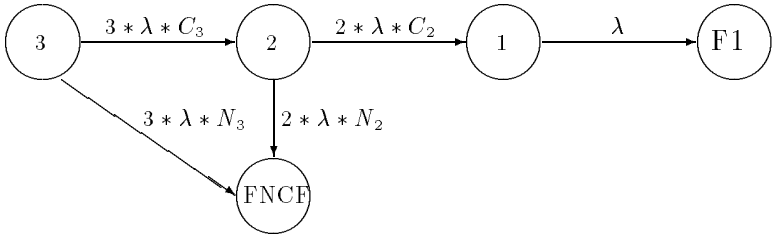

3,2
2,2
1,2
3,1
2,1
1,1
F1
F2
3∗λ
2∗λ
2∗μ
λ
μ

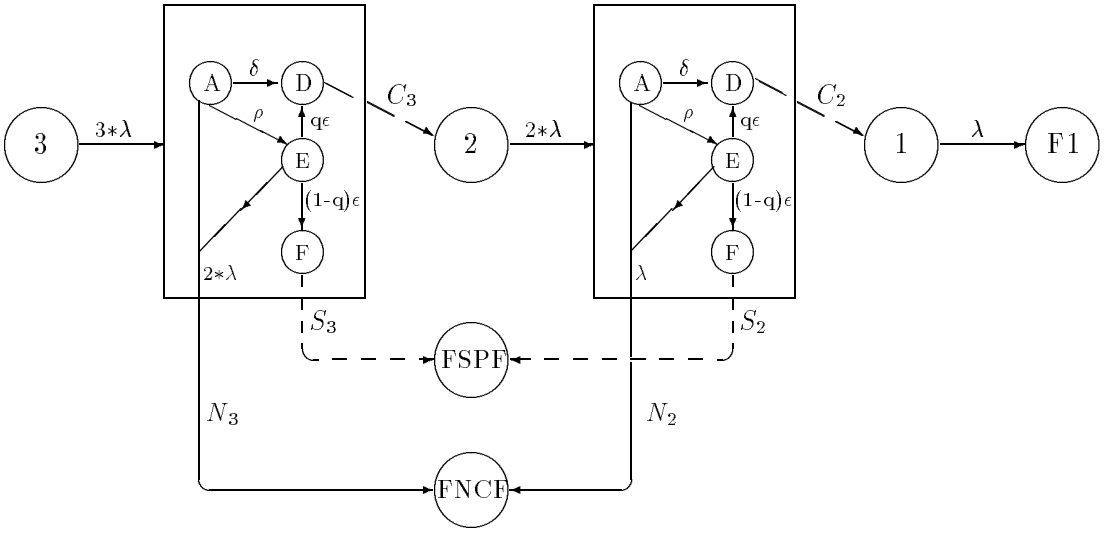

R
3,2
3λ
FEHM 1
C
2,2
2λ
FEHM 2
C
1,2
N
S
N
S
λ
R
2μ
R
2μ
R
2μ
FEHM 3
S
N
FEHM 4
S
N
FEHM 5
S
N
F1
C
C
C
λ
R
R
3,1
3λ
FEHM 6
C
2,1
2λ
FEHM 7
C
1,1
N
S
N
S
All N transitions from FEHM's
All S transitions from FEHM's
μ
μ
μ
FNCF
F2
FSPF


3,2
2,2
1,2
3,1
2,1
1,1
$3\lambda C_1$
$2\lambda C_2$
$3\lambda C_6$
$2\lambda C_7$
$2\mu C_3$
$2\mu C_4$
$2\mu C_5$
λ
λ
$3\lambda N_1 + 2\mu N_3$
$3\lambda S_1 + 2\mu S_3$
$2\lambda N_2 + 2\mu N_4$
$2\lambda S_2 + 2\mu S_4$
$2\mu N_5$
$2\mu S_5$
$3\lambda N_6$
$3\lambda S_6$
$2\lambda N_7$
$2\lambda S_7$
μ
μ
μ
F1
F2
All N_i transitions
All S_i transitions
FNCF
FSPF

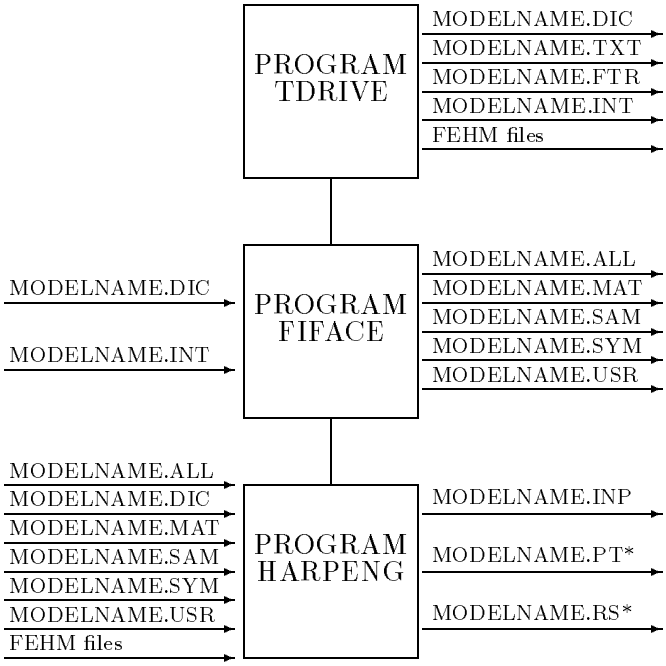

R_3
R_2
3
2
1
$3*\lambda$
$2*\lambda$
λ
FEHM 1
FEHM 2
C_3
C_2
F1
S_3
S_2
FSPF
N_3
N_2
FNCF

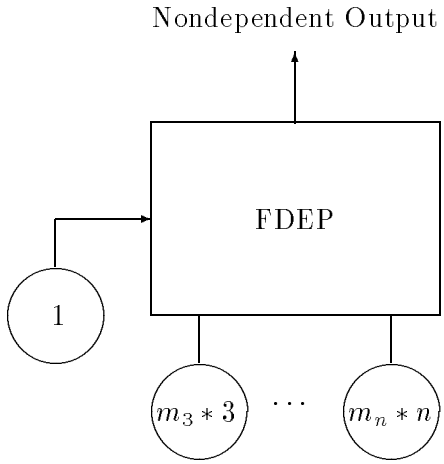

3
3*λ
A
δ
D
C3
2
2*λ
A
δ
D
C2
1
λ
F1
2*λ
λ
FNCF

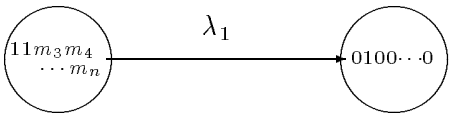

3
3 * λ * C3
2
2 * λ * C2
1
λ
F1
3 * λ * N3
2 * λ * N2
FNCF

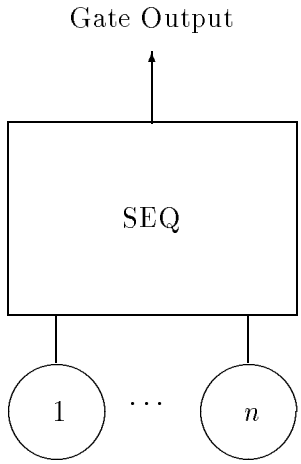

3
3*λ
A
δ
D
ρ
qϵ
E
(1-q)ϵ
F
2*λ
C3
2
2*λ
A
δ
D
ρ
qϵ
E
(1-q)ϵ
F
λ
C2
1
λ
F1
S3
S2
FSPF
N3
N2
FNCF

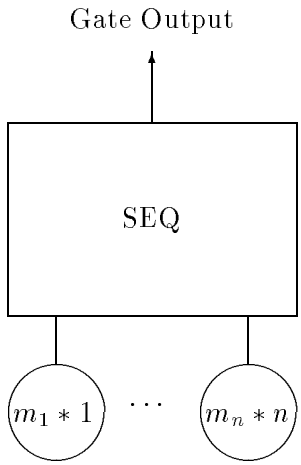

PROGRAM
TDRIVE
PROGRAM
FIFACE
PROGRAM
HARPENG

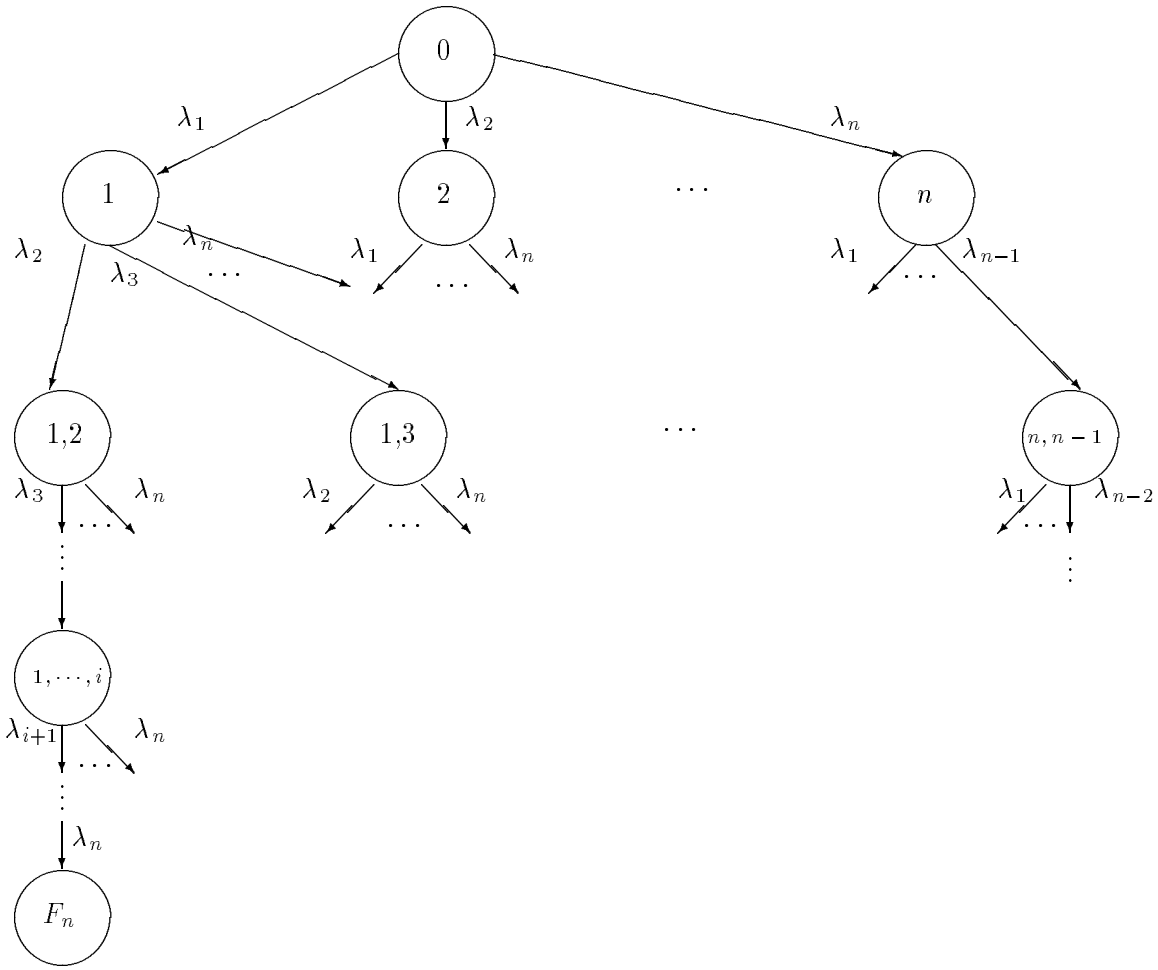

PROGRAM
TDRIVE
MODELNAME.DIC
MODELNAME.TXT
MODELNAME.FTR
MODELNAME.INT
FEHM files
PROGRAM
FIFACE
MODELNAME.DIC
MODELNAME.INT
MODELNAME.ALL
MODELNAME.MAT
MODELNAME.SAM
MODELNAME.SYM
MODELNAME.USR
PROGRAM
HARPENG
MODELNAME.ALL
MODELNAME.DIC
MODELNAME.MAT
MODELNAME.SAM
MODELNAME.SYM
MODELNAME.USR
FEHM files
MODELNAME.INP
MODELNAME.PT*
MODELNAME.RS*

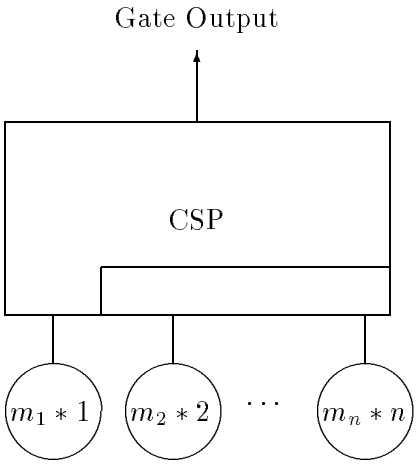

Nondependent Output
FDEP
1
$m_3 * 3$
$\cdots$
$m_n * n$

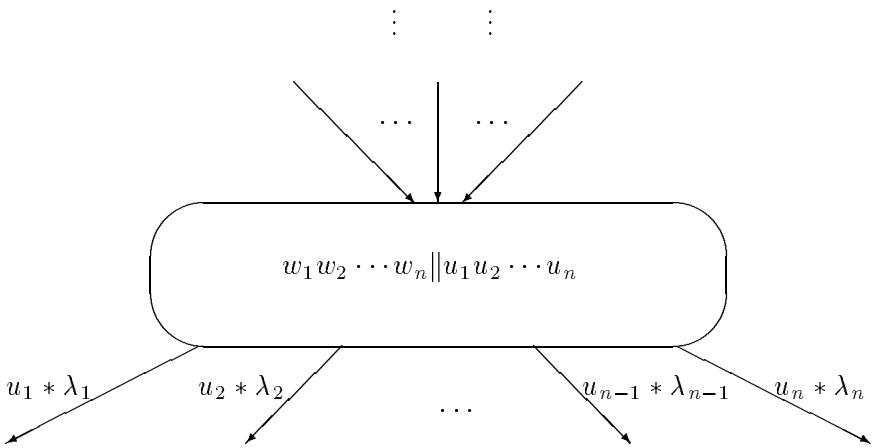

$11m_3m_4$
$\cdots m_n$
λ_1
0100$\cdots$0

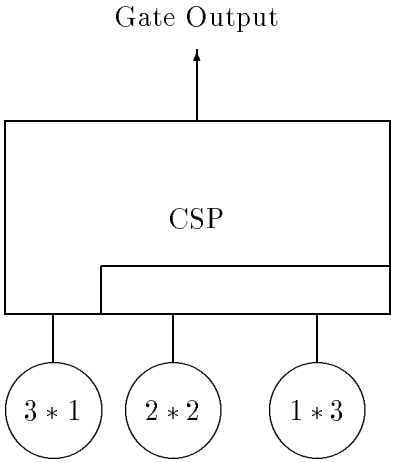

Gate Output
SEQ
1
...
n

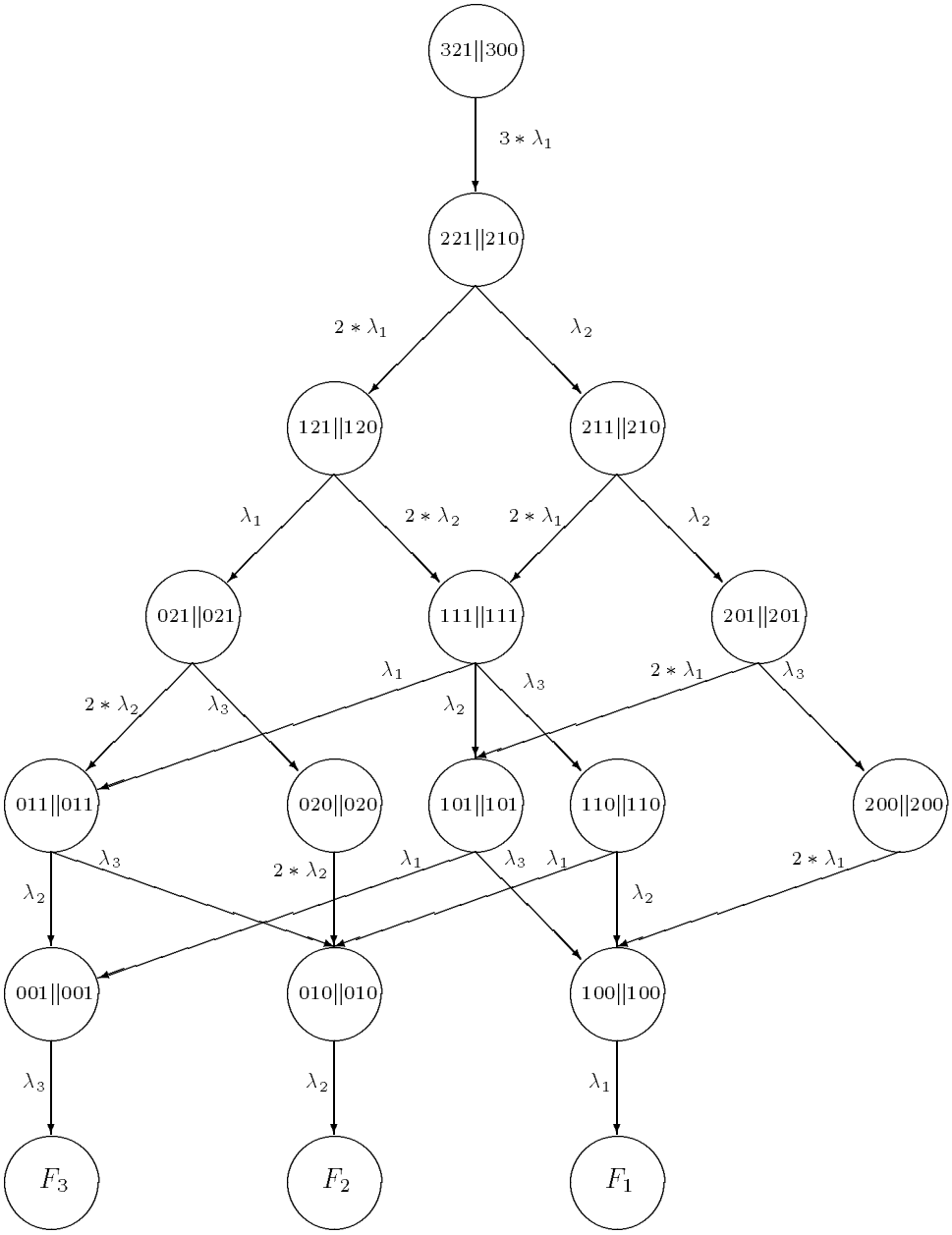

Gate Output
SEQ
$m_1 * 1$
...
$m_n * n$

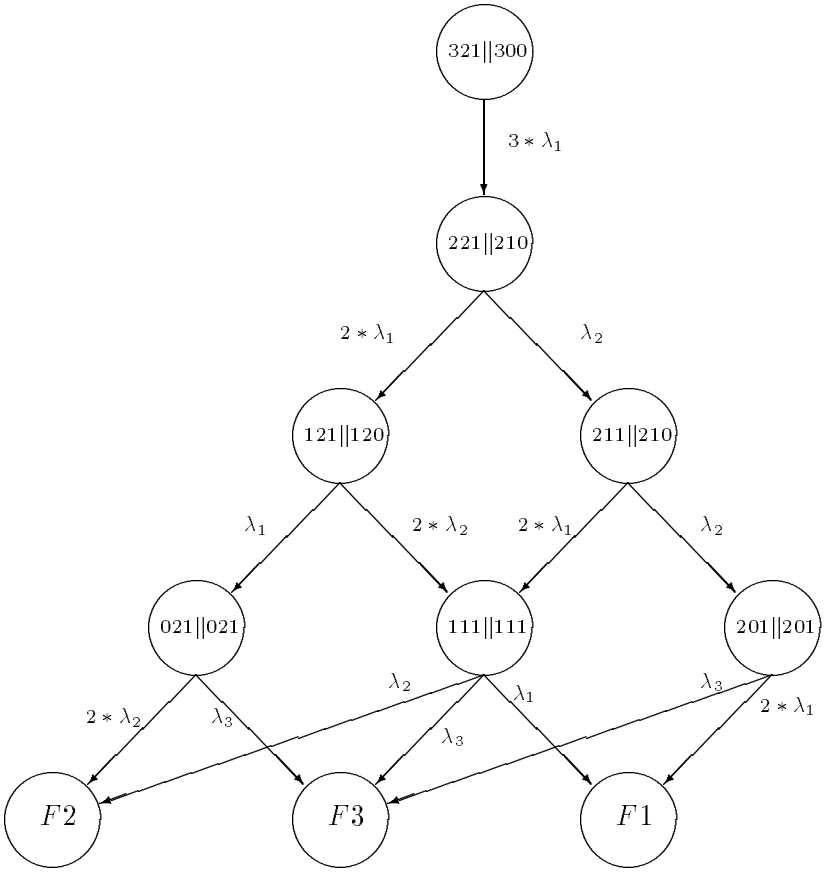

0
1
2
n
1,2
1,3
n, n − 1
1, · · · , i
F_n
λ_1
λ_2
λ_n
λ_3
λ_{i+1}
λ_{n-1}
λ_{n-2}

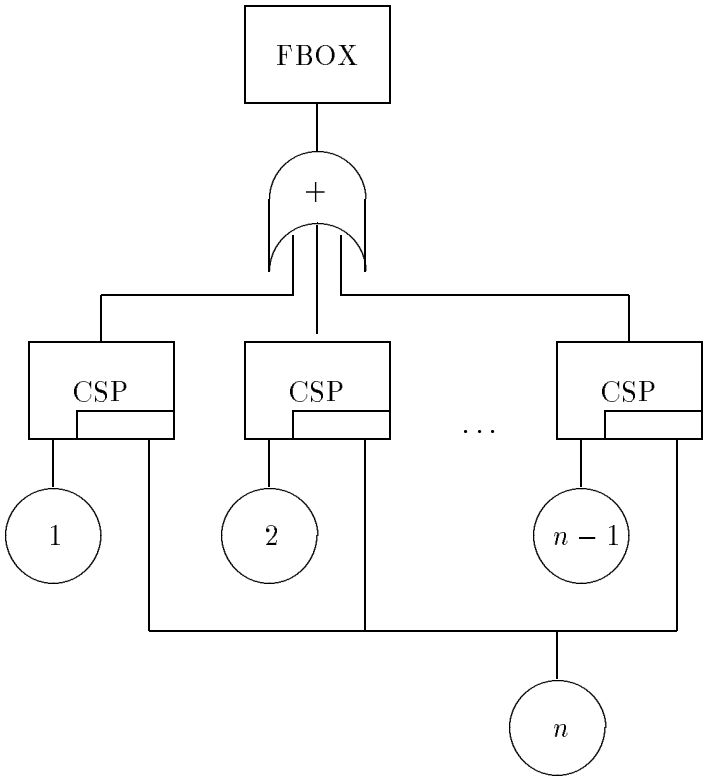

Gate Output
CSP
$m_1 * 1$
$m_2 * 2$
$m_n * n$

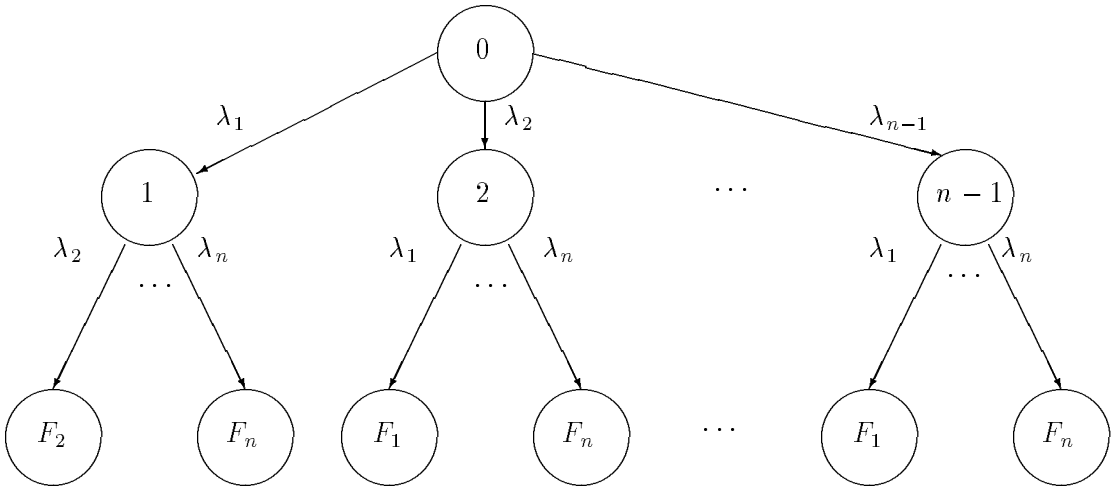

$w_1 w_2 \cdots w_n || u_1 u_2 \cdots u_n$
$u_1 * \lambda_1$
$u_2 * \lambda_2$
$u_{n-1} * \lambda_{n-1}$
$u_n * \lambda_n$

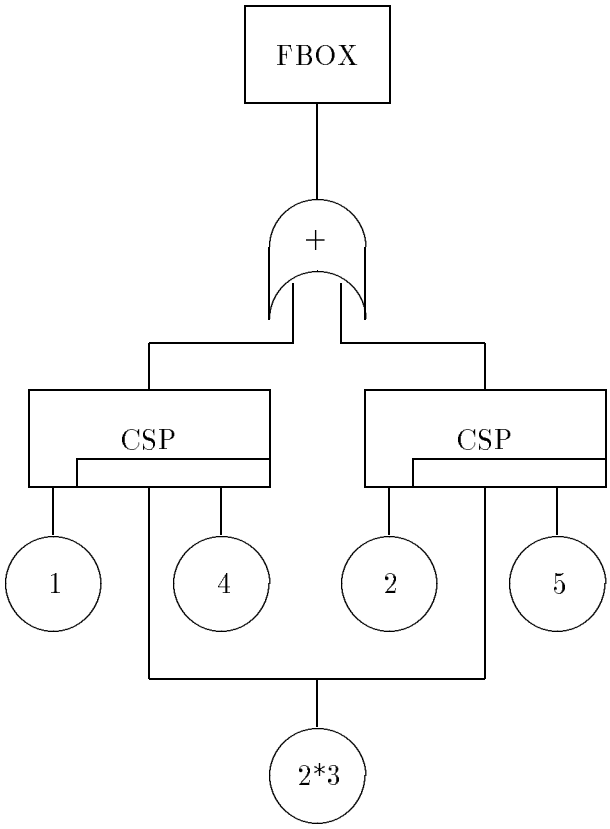

Gate Output
CSP
$3 * 1$
$2 * 2$
$1 * 3$

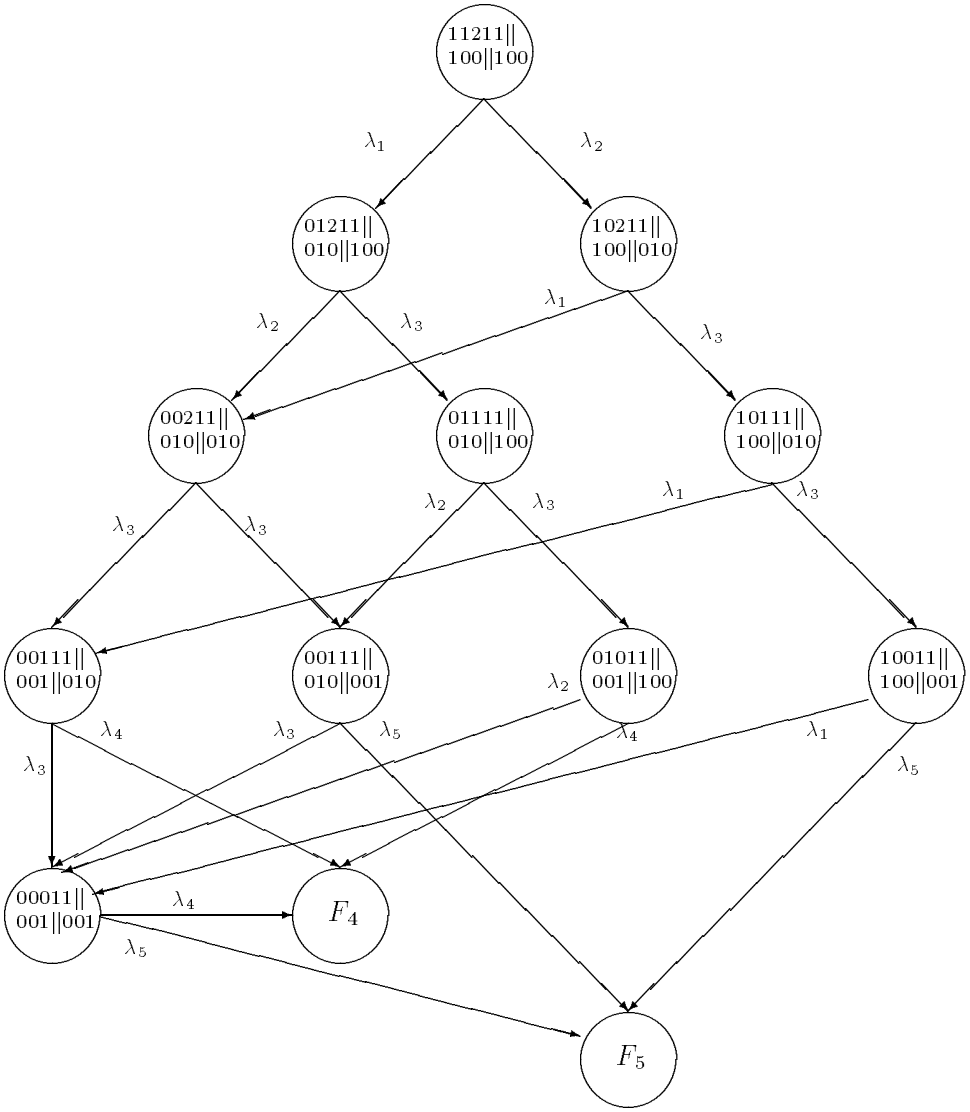

321||300
3 * λ1
221||210
2 * λ1
λ2
121||120
211||210
λ1
2 * λ2
2 * λ1
λ2
021||021
111||111
201||201
λ1
λ3
2 * λ1
λ3
2 * λ2
λ3
λ2
011||011
020||020
101||101
110||110
200||200
λ3
2 * λ2
λ1
λ3
λ1
2 * λ1
λ2
λ2
001||001
010||010
100||100
λ3
λ2
λ1
F3
F2
F1

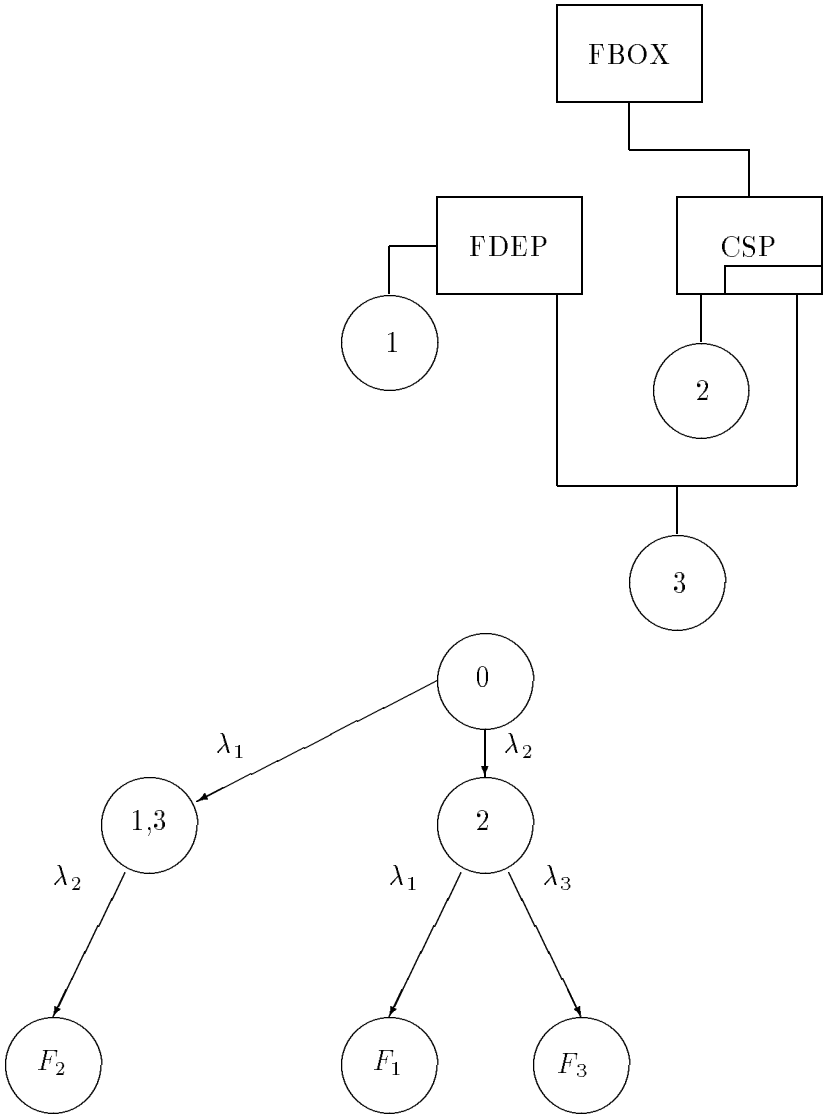

321||300
3 * λ1
221||210
2 * λ1
λ2
121||120
211||210
λ1
2 * λ2
2 * λ1
λ2
021||021
111||111
201||201
λ2
λ1
λ3
2 * λ2
λ3
λ3
2 * λ1
F2
F3
F1


FBOX
+
CSP
CSP
CSP
...
1
2
n − 1
n


0
λ1
λ2
λn−1
1
2
...
n − 1
λ2
λn
λ1
λn
λ1
λn
...
...
...
F2
Fn
F1
Fn
...
F1
Fn


FBOX
+
CSP
CSP
1
4
2
5
2*3


11211||
100||100
λ_1
λ_2
01211||
010||100
10211||
100||010
λ_2
λ_3
λ_1
λ_3
00211||
010||010
01111||
010||100
10111||
100||010
λ_3
λ_3
λ_2
λ_3
λ_1
λ_3
00111||
001||010
00111||
010||001
01011||
001||100
10011||
100||001
λ_2
λ_4
λ_3
λ_5
λ_4
λ_1
λ_3
λ_5
00011||
001||001
λ_4
F_4
λ_5
F_5


FBOX
FDEP
CSP
1
2
3
0
λ_1
λ_2
1,3
2
λ_2
λ_1
λ_3
F_2
F_1
F_3