

MITNE-228

NUCLEAR ENGINEERING
READING ROOM - M.I.T.

**RELIABILITY ANALYSIS OF COMPLEX TECHNICAL SYSTEMS USING THE
FAULT TREE MODULARIZATION TECHNIQUE**

**Mohammad Modarres
Norman C. Rasmussen
Lothar Wolf**

**MASSACHUSETTS INSTITUTE OF TECHNOLOGY
Department of Nuclear Engineering
Cambridge, Massachusetts 02139**

January 1980

READING ROOM - BILT

MITNE-228

RELIABILITY ANALYSIS OF COMPLEX TECHNICAL SYSTEMS USING THE
FAULT TREE MODULARIZATION TECHNIQUE

Mohammad Modarres
Norman C. Rasmussen
Lothar Wolf

January 1980

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
Department of Nuclear Engineering
Cambridge, Massachusetts 02139

Corrections to MITNE-228

pg. 203

Listing of CECA Procedure

the following PLL coding statement should be inserted
Following statement labeled PLM02850

IF (STATS(L,I) = 0 & STATED(L,I) = 0) THEN STATD(L,I) = 0 ELSE

pg 209

Listing of SHR Procedure

the following PLL coding statements should be inserted
Following statement labeled PLM03900

IF (STATS(L,I) = 0 & TTETAD(L,I) = 0) THEN STATD(L,I) = 0;
ELSE

pg 461

Table B.1

11th line from top

presently reads:	6	2	7	8	0	0	0	0
Should read	6	2	2	7	8	0	0	0

These corrections have been
discussed with Mohammad Modares

Rachel Merten 5-20-80

RELIABILITY ANALYSIS OF COMPLEX TECHNICAL SYSTEMS USING THE FAULT TREE MODULARIZATION TECHNIQUE

ABSTRACT

The method of modular decomposition of a fault tree that has been used in the PL-MOD code is extended to include several calculational features, and to remove some of the restrictions presented in its methodology. The PL-MOD's methodology consists of piecewise collapsing and modularizing portions of the tree until eventually the fault tree structure is described as a set of modular equations recursively relating the top event to its basic component inputs. In particular, the PL-MOD code demonstrated the ability to evaluate large fault trees in an economic, efficient and accurate manner. The PL-MOD code is written in PL/I language in order to use the dynamic core utilization techniques available in this language. Accordingly, the extensions of the code developed in this work have all been performed by using the PL/I language.

Time dependent models were developed to include the effect of different classes of time-dependent components in a fault tree to the top event unavailability at different instants of time. This new version of the code has been designated PL-MODT. One of the features that is incorporated into the PL-MODT is the automatic reduction of large fault trees. In this procedure a CUTOFF value is defined for the Vesely-Fussell importance measures, in order to eliminate those components and modules of the fault tree whose importances are less than this prescribed value.

A Monte-Carlo simulation package is also incorporated into the PL-MOD code for the Monte-Carlo simulation of those input parameters to the fault tree for which uncertainties can be assigned. The code obtained by incorporating this package into the PL-MOD Code, is called PL-MODMC. In the PL-MODMC Code, failure rates, repair duration, average downtime, and unavailability per demand are the failure parameters in a fault tree to which a probability density function (p.d.f.) can be assigned. The log-normal distribution is chosen to represent this p.d.f. Some of the restrictions in the PL-MOD code such as the limitation on the appearance of a component in all of the domains of a fault tree are removed.

An algorithm is developed for deriving all simple minimal cutsets from the Modular cutsets. This algorithm allows the determination of cutsets of all proper modules presented in the set of modular cutsets. Then by assigning these cutsets to the corresponding modular minimal cutsets,

a complete set of simple cutsets for the top event of the tree can be obtained. This algorithm is incorporated into the PL-MOD code in order to obtain the MODCUT code. The cutsets of higher order modules can be discriminated, and thus be reduced in the MODCUT code, based on any defined criterion. Presently the code uses the size of the cutsets as the discrimination criterion. In particular, it is found that a common cause discrimination criterion can be easily incorporated, and with a substantial saving in the computer time over other common cause failure analysis codes, a complete set of common cause candidates can be obtained.

A thorough review of common cause failure analysis is performed, and packages MOCUS-BACFIRE, and MODCUT-BACFIRE developed to identify, directly from the system fault tree, a set of cutsets which are common cause failure candidates. Common cause quantification methods are studied and a quantitative model is proposed in order to obtain the common cause failure unavailability directly from the cutsets that are identified as common cause candidates.

Several fault tree examples are evaluated by using PL-MODT, PL-MODMC, and MODCUT codes. The results obtained from these examples have emphasized the accuracy, efficiency and effectiveness of these codes.

The work presented in this report contains virtually all of the Ph.D thesis submitted by Mr. Mohammed Modarres to the Department of Nuclear Engineering at MIT in November 1979. The thesis was supervised by Professor Norman C. Rasmussen and Dr. Lothar Wolf. The funding for this study was partly provided by the U.S. Nuclear Regulatory Commission and partly by the following electric utilities.

Boston Edison Co.
Consumers Power Co.
Northeast Utilities Services Co.
Public Service Electric & Gas Co.

Their financial support is gratefully acknowledged.

TABLE OF CONTENTS

	Page
TITLE PAGE	1
ABSTRACT	2
ACKNOWLEDGMENTS	4
LIST OF TABLES	11
LIST OF FIGURES	16
CHAPTER 1 INTRODUCTION	19
1.1 Fault Tree Analysis and Its Role	19
1.2 Some Basic Definitions of the Frequently Used Terms in This Thesis	24
1.3 Fault Tree Evaluation	28
1.3.1 Qualitative Evaluation	28
1.3.2 Quantitative Evaluation	30
1.4 Fault Tree Modularization	32
1.4.1 Introduction to the Modularization Technique and its Mathematical Concepts	33
1.4.2 The PL-MOD Methodology	38
1.4.3 Other Modularization Methodologies	48
1.4.3.1 Chatterjee's Method	49
1.4.3.2 PATREC-RCM's Method	52
1.5 Objectives of This Study	54
CHAPTER 2 TIME DEPENDENT FAULT TREE ANALYSIS	62
2.1 Introduction	62
2.2 Class 1 Components: Time Independent Components	63
2.3 Class 2 Components: Nonrepairable Components	66

	Page
2.4 Class 3 Components: Repairable Components	68
2.5 Class 4 Components: Periodically Tested Components	72
2.6 General Time Dependent Relations for the Evaluation of Fault Trees by Using the Modular Concept	79
2.7 General Relations for Time Dependent Simple Modules Consisting of Only Repairable Components (Class 3 Components)	81
2.8 General Relationships for a Time-Dependent Simple Module Consisting of Only Non-Repairable Components (Class 2 Components)	88
CHAPTER 3 REDUCTION OF LARGE FAULT TREES BASED ON THE VESELY-FUSSELL IMPORTANCE MEASURES	92
3.1 Introduction	92
3.2 Importance Measures and the Use of PL-MOD to Calculate the V-F Importance Measures	92
3.3 Use of the Code PL-MOD for Reducing Large Fault Trees	103
3.4 Reductions of LPRS and HPIS Fault Trees	104
CHAPTER 4 MONTE-CARLO SIMULATION OF TOP EVENT PROBABILITY	109
4.1 Introduction	109
4.2 The Log-Normal Distribution	111
4.2.1 Justification for Use of Log-Normal Distribution	111
4.2.2 Characteristics and Some of the Mathematical Concepts of Log-Normal Distribution	111
4.3 Mathematical Concepts used in PL-MODMC for the Monte-Carlo Simulation	113
4.4 Distribution Models for the Modules of a Coherent System Fault Tree	121
4.5 The PL/I Random Number Generator in the PL-MODMC Code	128

	Page
4.6 The Monte-Carlo Algorithm in PL-MODMC	130
CHAPTER 5 DEVELOPMENT OF THE CUTSET DISCRIMINATOR CODE MODCUT	137
5.1 Introduction	137
5.2 MODCUT Methodology	138
5.3 An Example Showing the Generation of Simple Cutsets by the MODCUT Algorithm	142
CHAPTER 6 COMMON CAUSE ANALYSIS	159
6.1 Introduction	159
6.2 Review and Background Information	160
6.3 Some Qualitative Aspects of Common Cause Failure Analysis	165
6.4 MOCUS-BACFIRE, A Convenient Code Package for the Identification of Common Cause Candidates from a Fault Tree	170
6.5 MODCUT-BACFIRE - An Alternative to MOCUS-BACFIRE	173
6.6 Quantification of Common Cause Failures using the Cutsets of a Fault Tree	175
6.6.1 Quantification Methodology using a Set of Common Cause Candidates	176
6.6.2 A Method for Calculating $\lambda_{C.C.}$	184
6.6.3 Common Cause Analysis of the Aux Feed System (AFS)	185
CHAPTER 7 DESCRIPTION OF COMPUTER PROGRAMS DEVELOPED IN THIS WORK	196
7.1 Introduction	196
7.2 Organization of the Procedures used in PL-MODT	197
7.2.1 Development of the SHORE Procedure	199
7.2.2 Development of the CECA Procedure	200
7.2.3 Development of the SHR Procedure	202

	Page
7.2.4 Modifications Performed in the Old External Procedure NUMERO	210
7.2.5 Modifications Performed in the Internal Procedure IMPORTANCE	215
7.3 Organization of the Procedures used in PL-MODMC	227
7.3.1 Removal of the Restriction of Component Replication in Different Domains of Fault Trees	227
7.3.2 Incorporation of the Monte-Carlo Package NUMERO into the PL-MODMC Code	251
7.4 Organization of the Procedures used in MODCUT	267
7.4.1 CUT-SET Procedure	267
7.4.2 TOP-SET Procedure	273
7.5 Organization of the MOCUS-BACFIRE Package	289
7.6 Organization of the MODCUT-BACFIRE Package	292
CHAPTER 8 APPLICATION OF THE CODES PL-MODT, PL-MODMC, AND MODCUT TO SOME SPECIAL PROBLEMS OF INTEREST IN REACTOR SAFETY	303
8.1 Introduction	303
8.2 Examples for the Code PL-MODT	303
8.2.1 Low Pressure Recirculation System (LPRS) for PWRs (Class 1)	304
8.2.2 Comparison Between the Results of the Codes PL-MODT and PREP & KITT-1 (Components are of Class 2 and 3)	311
8.2.3 Auxiliary Feedwater System - A Comparison Between PL-MODT and FRANTIC (Class 4)	318
8.2.4 Example of a Simple Electric Circuit Using FRANTIC and PL-MODT	322
8.2.5 Differences Between FRANTIC and PL-MODT	330
8.2.6 Comments on the Vesely-Fussell Importance Measure	330

	Page
8.3 Examples for the Code PL-MODMC	332
8.3.1 Monte-Carlo Simulation of Failure Parameters of a Fault Tree Consisting of a Combination of the Four Classes of Components	334
8.3.2 Low Pressure Recirculation System (LPRS)	341
8.3.3 New B&W Reactor Protection System	341
8.4 Examples for the Code MODCUT	362
8.4.1 Generating Cutsets of the Fault Tree Shown in Figure (5.3.1) by Using the MODCUT Code	363
8.4.2 A Fault Tree with Different Types of Components and Gates	370
8.4.3 A Fault Tree to Identify Solidified High-Level Waste Release from the Containment in the RSSF	370
8.4.4 The Fault Tree for the New B&W Protection System	380
CHAPTER 9 CONCLUSIONS AND RECOMMENDATIONS	384
9.1 Summary and Conclusions	384
9.2 Recommendations	389
APPENDIX A FAULT TREE EVALUATION CODES	396
A.1 Introduction	396
A.2 Qualitative Analysis Codes	398
A.3 Quantitative Analysis Codes	404
A.4 Direct Evaluation Codes	409
A.5 Dual Purpose Codes (the PL-MOD Series)	414
A.6 Common Cause Analysis Codes	419
APPENDIX B DESCRIPTION OF INPUT DATA FOR PL-MODT	426
B.1 Description of the Code PL-MODT	426
B.2 Input Data Example	459

	Page
APPENDIX C DESCRIPTION OF THE INPUT FOR PL-MODMC	471
C.1 Input Data	471
C.2 Input Data Example	479
APPENDIX D DESCRIPTION OF INPUT TO MODCUT	481
D.1 Input Data	481
D.2 Input Data Example	485
APPENDIX E DESCRIPTION OF INPUT TO MOCUS-BACFIRE PACKAGE	487

LIST OF TABLES

<u>Table Number</u>	<u>Title</u>	<u>Page</u>
2.7.1	Component Input to a Simple AND Module	88
2.8.1	Failure Characteristics of Simple Modules Having Repairable or Non-repairable Component Inputs	90
3.2.1	Pressure Tank Rupture Fault Tree Failure Probability Data	98
3.4.1	Percentage of Increase in the Top Event Occurrence of Reduced Trees of LPRS for Different Cut-Off Values	105
4.3.1	Average Unavailability Functions used in the PL-MODMC	114
4.4.1	Log-Normal Distribution Parameters for the Unavailability of the Components Input to an OR Module	123
6.3.1	Common Links Resulting in Dependence Among Components	168
6.3.2	Mechanical or Thermal Generic Causes	168
6.3.3	Electrical or Radiation Generic Causes	169
6.3.4	Chemical or Miscellaneous Generic Causes	169
6.6.1	Failure Data for the AFS Components	190
7.2.1	Procedure SHORE	201
7.2.2	Procedure CECA	203
7.2.3	The List of Statements to Indicate the Cycle in which a Component is Operating	205
7.2.4	Procedure SHR	207
7.2.5	Modifications in the Main Routine of the NUMERO Procedure	213
7.2.6	Procedure PLUS	216
7.2.7	Procedure IMPORTANCE	219

<u>Table Number</u>	<u>Title</u>	<u>Page</u>
7.3.1	Corresponding Numeric Values for the Components and Gates of the Fault Tree in Figure (7.3.1)	230
7.3.2	Set of Statements to Count Replicated Components in Each Domain	233
7.3.3	Set of Statements to Make Replicated Gates and the Top Event Independent	237
7.3.4	Set of Statements to Transfer PER Structure Data and Minimal Modular Cutsets to MISS Structure	246
7.3.5	Removal of OFF Complementary Components and Treatment of Replications	247
7.3.6	Set of Statements to Combine Modular Minimal Cutsets	248
7.3.7	Allocation of the New PER Structure	252
7.3.8	Proper Treatment of the Disconnections Made in the TREE-IN Procedure	254
7.3.9	Set of Statements to Transfer Cutsets Stored in the Bitstrings MODCUT.AOMP(I) to the TEMP1 and TEMP2 Arrays	255
7.3.10	Input Data Treatment by the MONTE-CARLO Package	257
7.3.11	Set of Statements for the Measure of Accuracy of the Monte-Carlo Calculations	260
7.3.12	The Random Number Generator and the Sampling Procedure in the PL-MODMC Code	261
7.3.13	Set of Statements to Sort the Unavailabilities	262
7.3.14	The EXPECT Procedure	264
7.3.15	The PLUS Procedure	266
7.3.16	The DOT Procedure	268
7.4.1	List of Statements to Transfer Proper AND and OR Modules to the MCSG Structure and Generate the Bitstrings for Them	274

<u>Table Number</u>	<u>Title</u>	<u>Page</u>
7.4.2	List of Statements for Calculating the Number and Dimension of Bitstrings in the CUTSET Structure	276
7.4.3	List of Statements to Allocate CUTSET Structure for an AND Gate and Generate the Bitstrings of the CUTSET Structure	277
7.4.4	List of Statements to Transfer Data from PER Structure and Transfer Modular Minimal Cutsets to the MCSG Structure	279
7.4.5	List of Statements to Calculate Number and Length of the CUTSET Structure Bit-strings for Improper Modules	280
7.4.6	List of Statements to Generate Cutsets of Improper Modules from the CUTSET Structure	284
7.4.7	List of Statements to Account for Replication of a Component in the CUTSET.COMP(I) Bitstring	286
7.4.8	List of Statements to Perform a Size Discrimination	287
7.4.9	List of Statements to Minimize, Sort and Print the Cutsets	288
7.4.10	List of Statements to Allocate a New Cutset Structure to Store only Essential Cutsets	290
7.5.1	New Form of the MAIN and PRINTR Subroutines in the MOCUS Code	293
7.5.2	Variables that have been Put on the Common Blocks in the BACFIRE Code	298
8.2.1	Primary Failure Rates for Sample Fault Tree Shown in Figure (8.2.3)	313
8.2.2	Failure and Repair Rates for Sample Tree in Figure (2.8.3)	316
8.2.3	The Component Input Data for the Codes FRANTIC and PL-MODT [4]	320
8.2.4	Inspection Parameters for the Circuit Example	326
8.3.1	Failure Data for the Fault Tree in Figure 8.3.1	337

<u>Table Number</u>	<u>Title</u>	<u>Page</u>
8.3.2	PL-MODMC Output of Fault Tree Presented in Figure 8.3.1	338
8.3.3	LPRS Top Event Unavailability for 1200 Trials Using PL-MODMC	343
8.3.4	Comparison of the LPRS Fault Tree Simulation Using PL-MODMC and the Results Calculated by the Sample Code in WASH-1400	344
8.3.5	GO Components and Failure Data	359
8.3.6	PL-MODMC Output for the Fault Tree in Figure (8.3.7) and Data in Table (8.3.5)	361
8.4.1	Input to MODCUT with the CUTNO Variable Equal to One	364
8.4.2	Number of Cutsets in Each Module Calculated by MODCUT with CUTNO = 1	365
8.4.3	Cutsets Calculated for Fault Tree in Figure (5.3.1) with CUTNO = 0 and DEBUG = 1	366
8.4.4	MODCUT Output for the Fault Tree in Figure (8.4.1)	372
8.4.5	Outputs for the Fault Tree Shown in Figure 8.4.2	378
8.4.6	Low Order Cutsets Obtained by the MODCUT Code for the Fault Tree in Figure (8.3.7)	381
8.4.7	Corresponding Names for the Components Listed in the Minimal Cutsets	382
B.1	General Data Set Representing the Logic of the Tree	460
B.2	Data Set Following Table A.1 for only Class 1 Components (PL-MOD Case)	463
B.3	Data Set Following Table A.1 for only Class 2 Components	465
B.4	Data Set Following Table A.1 for only Class 3 Components	467

<u>Table Number</u>	<u>Title</u>	<u>Page</u>
B.5	Data Set Following Table A.1 for the only Class 4 Components or Combination of all Time Dependent Classes of Components	469
C.1	Inputs to Fault Tree Presented in Figure 8.3.1	480
D.1	MODCUT Input for the Fault Tree Shown in Figure 8.4.1	486
E.1	Input Example to the MOCUS-BACFIRE Package	489
E.2	Output from MOCUS-BACFIRE Package	492

LIST OF FIGURES

<u>Figure Number</u>	<u>Title</u>	<u>Page</u>
1.1.1	Pressure Tank Example	21
1.1.2	The Pressure Tank System Fault Tree	22
1.2.1	Fault Tree Symbols	24a
1.4.1	Sample Sub-Tree I with No Replications	37
1.4.2	Finest Modular Representation of Sample Sub-Tree	40
1.4.3	Sample Sub Tree II with Replications	42
1.4.4	Finest Modular Representation of Sample Sub Tree II	43
1.4.5	Fault Tree Modularization Algorithm	46
1.4.6	Chatterjee's Methodology for Modularizing a Fault Tree	50
2.2.1	A Markovian Model for Time Independent Components	64
2.2.2	A Markovian Model	65
2.4.1	A Markovian Model for Repairable Components (Revealed Faults)	68
2.5.1	Unavailability of a Component as a Function of Time Using Equation (2.5.8)	76
2.7.1	Unavailability of an <u>AND</u> Module as a Function of Time	87
3.2.1	Higher Order Prime Gate Super-Module	93
3.2.2	Pressure Tank Example	97
3.2.3	Pressure Tank Rupture Fault Tree and Associated Modules (Failure Rate for Component 1, $\lambda_1 = 1 \times 10^{-8} \text{ hr}^{-1}$; all others $\lambda = 1 \times 10^{-5} \text{ hr}^{-1}$)	99
3.2.4	Reduced Version of the Fault Tree in Figure (3.2.3)	102
4.4.1	Comparison of the True and Approximate Log-Normal Distribution for an OR Module	124
4.6.1	The Monte-Carlo Algorithm in PL-MODMC	131

<u>Figure Number</u>	<u>Title</u>	<u>Page</u>
4.6.2	The Execution Times of LIMITS, SAMPLE and PL-MODMC Codes	133
4.6.3	Reduced Fault Tree of the Reactor Protection System for a PWR	134
5.2.1	Cutset Generation Algorithm in MODCUT	139
5.3.1	Sample Fault Tree for the MODCUT Algorithm	143
6.6.2	Block Diagram of the Aux-Feed System	186
6.6.3	Fault Tree for the AFS	189
6.6.4	AFS Unavailability due to Random and Common Cause Failures	192
7.2.1	Unavailability as a Function of Time	212
7.3.1	Fault Tree for a Monte-Carlo Analysis	229
7.3.2	New Form of the Fault Tree Given in Figure (7.3.1)	236
7.5.3	List of Statements Changed in the CUT-SET and TOP-SET Procedures of the MODCUT Code	299
8.2.1	Simplified Flow Diagram of LPRS	305
8.2.2	LPRS Reduced Fault Tree (Sh. 1)	306
8.2.3	Fault Tree Example	312
8.2.4	Comparison of the Time-Dependent Unavailabilities for the Sample Tree Given in Figure 8.2.10 as Calculated by PREP and PL-MODT	314
8.2.5	Comparison of the Time-Dependent Unavailabilities of Repairable Components for the Sample Tree Given in Figure (8.2.3) as Calculated by FRANTIC, PREP, KITT and PL-MODT	317
8.2.6	Block Diagram of the Simplified Aux-Feed System	319
8.2.7	{ Comparison Between the Time-Dependent Unavailabilities for the Aux-Feed System as Calculated by FRANTIC and PL-MODT	321
8.2.8		
8.2.9	Sample System for Mutually Exclusive Events	323

<u>Figure Number</u>	<u>Title</u>	<u>Page</u>
8.2.10	Fault Tree for Sample System in Figure (8.2.9)	325
8.2.11	Comparison between the Unavailabilities for the Electrical System during its Operation as Calculated by FRANTIC and PL-MODT	328
8.2.12	Comparison between the Unavailabilities During Test and Repair Periods as Calculated by Both Codes	329
8.2.13	Importance of the Pump in the Aux-Feed System as Function of Time	331
8.2.14	Importance of the Valves of the Aux-Feed-System as Function of Time	333
8.3.1	Fault Tree for a Monte-Carlo Analysis	335
8.3.2	Comparison of the Top Event Distribution for the Fault Tree in Figure (8.3.1) Calculated by the Codes PL-MODMC and LIMITS	342
8.3.3	Reactor Trip System	345
8.3.4	2/4 Trip Logic	347
8.3.5	Safety Groups	348
8.3.6	Regulating Groups	350
8.3.7	B&W Reactor Protection System Fault Tree	351
8.4.1	Sample Fault Tree	371
8.4.2	A Fault Tree to Identify Potential Sequences of Events by which Solidified High-Level Waste Could be Released during Storage at the RSSF	377
A.1	Fault Tree Evaluation Codes	397

CHAPTER 1

INTRODUCTION

1.1 Fuel Tree Analysis and Its Role

Modern technology has developed much larger more complicated systems that often have the potential for accidents with substantial safety and economic consequences. Consideration of these possible economic and health risks has become of increasing importance in new technological enterprises. In the last few decades a variety of methods for evaluating these risks have been developed and applied. Typically these methods estimate the risks by calculating both the probability of system failures and their consequences. In these analyses, a quantitative measure of the system's likelihood of successfully performing the design goals is defined, and called the system "reliability". There are various definitions of reliability; for instance, reliability has been defined in [1] as follows:

"Reliability is the characteristic of an item expressed by the probability that it will perform a required function under stated conditions for a stated period of time."

The characteristic called "unreliability" is just the numerical complement of the reliability. Thus, in a general sense, the field of reliability analysis consists of determining the probability of a system failing under various conditions and in various modes.

A commonly used method for analyzing system failure modes is called fault tree analysis. In this approach the analyst defines a specific undesired state of the system. This state is called the "top event",

and the analyst then proceeds through a logical step-by-step procedure to determine what combinations of basic event failures lead to the top event. As an example of this procedure refer to Figures (1.1.1) and (1.1.2), which show a pressure tank system and the fault tree developed for the pressure tank rupture.

Although the fault tree approach is the most widely used method of system failure analysis, there are other methods which can be used. Among these are failure mode and effect analysis, success tree analysis, and reliability block diagrams. Each of these methods has applications where it may be the method of preference. However, this study is denoted to improving methods for analyzing complex fault trees, and the other methods will not be discussed here.

Especially during the last decade there have been a number of applications of the fault tree method to problems of practical interest. The System Reliability Service of the United Kingdom, under the direction of F. R. Farmer, did much of the pioneering work and applied this method especially to the nuclear and chemical industries [2]. The reliability analysis in safety studies of nuclear power plants has been advocated since the mid 1960's in the United States [3,4]. The major step towards the effective use of a quantitative reliability analysis has been made by the Reactor Safety Study (RSS) [5] (WASH-1400), which is now followed by similar efforts in various other countries, e.g., West Germany. Aside, perhaps, from the Apollo program and major defense projects, WASH-1400 is the most concentrated effort of its kind thus far. The purpose of the study was to estimate the risks to the public from potential accidents in U. S. commercial Nuclear Power Plants. The analytical backbone of the WASH-1400 methodology is fault tree and event tree analysis.

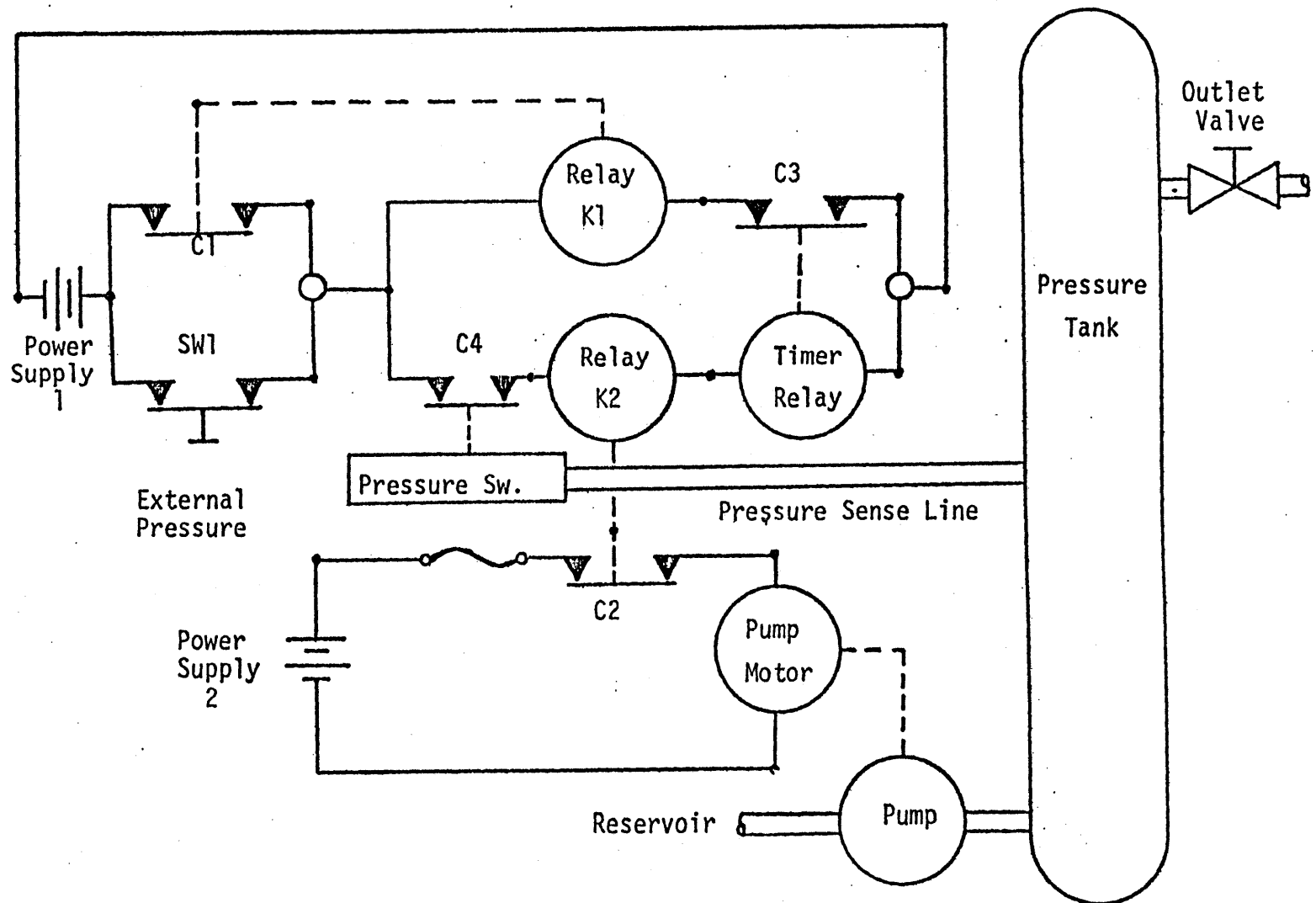


Figure 1.1.1 Pressure Tank Example

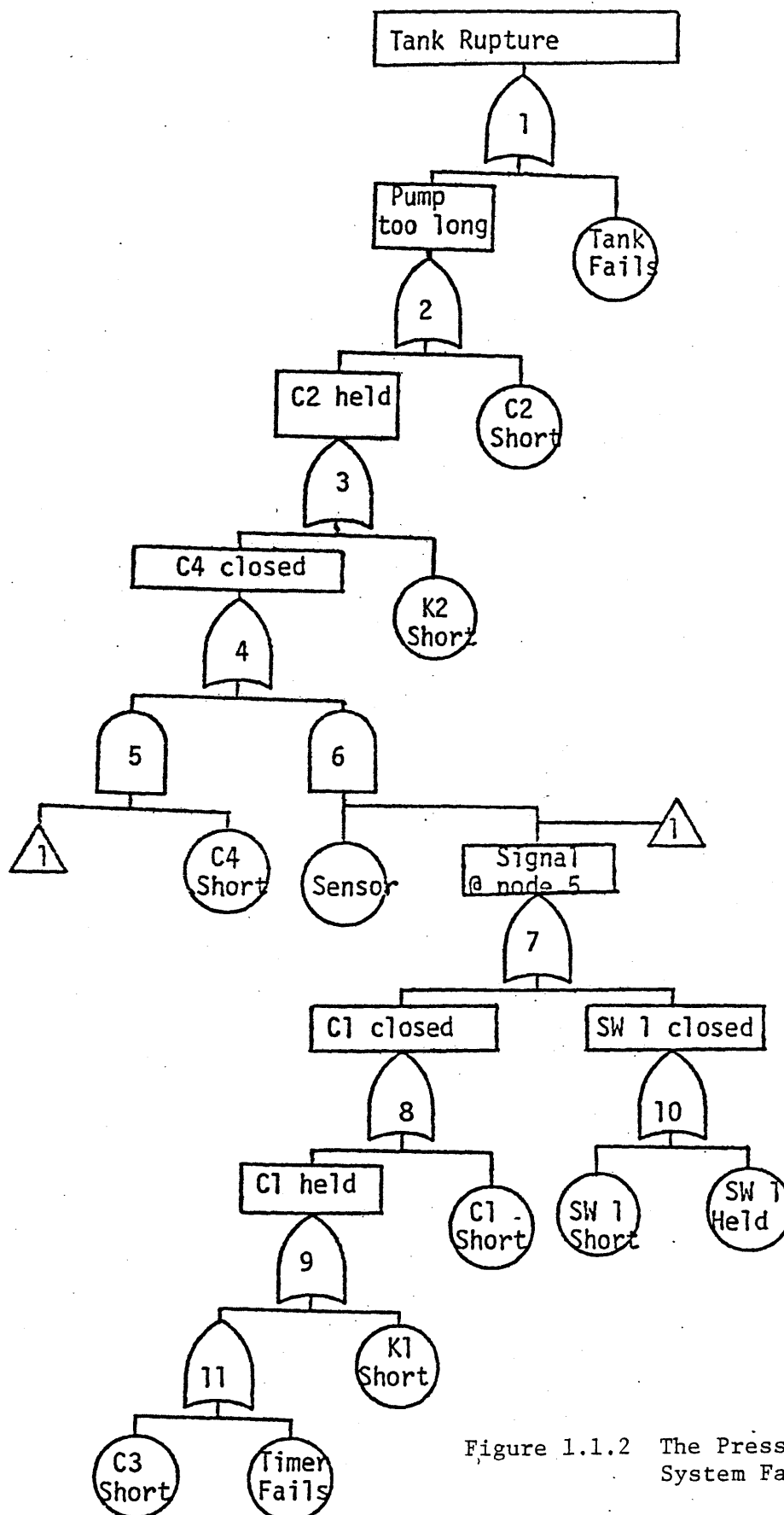


Figure 1.1.2 The Pressure Tank System Fault Tree

WASH-1400 concluded that the best estimated value for the probability of a core melt-down is 5×10^{-5} per reactor-year. This estimate seems reasonable, because the experience over past operations of nuclear power plants suggests that a core melt should be much less likely than 10^{-3} , and that the best estimate is probably less than a factor of 10 off. In a recent assessment of the RSS results by the Lewis review group [6] it has been concluded that error bounds on probability estimation are probably underestimated, mainly due to the lack of precise data base. However, the group has emphasized the substantial advance achieved by the WASH-1400 over previous attempts to estimate the risk associated with the operation of nuclear power plants. In particular, the use of the fault tree/event tree approach is recommended as the "...best available tool with which to quantify probabilities.

WASH-1400 has stimulated much interest in the use of fault tree analysis methods in safety systems. There is now widespread use of fault tree methods to achieve safety goals. Perhaps the most ambitious program in the safety area is in connection with the Clinch River Breeder Reactor (CRBR) [7]. This program is aimed at calculating different reliability measures for the Shut Down System, and the Shut Down Heat Removal System in the CRBR. Most of the methods used in WASH-1400 are also being employed in this study; in particular, the fault tree analysis forms the basis of this research program. In a study whose goal was to assess the accident risk of a conceptual high-level waste management [8] fault tree analysis has been extensively used. This study concluded that the accidental release of radioactive waste at any stage of the waste management is very small.

In many other nuclear safety studies [9-21], fault tree analysis has been the principal tool for qualitative and quantitative evaluation of the study. In summary, fault tree analysis has become an important segment of an effective system reliability analysis, and hence there is a need for further development and use of this method in the reliability analysis of complex technical systems.

1.2 Some Basic Definitions of the Frequently Used Terms in This Thesis

Because the basic elements of a fault tree are the same, regardless of the types of events or systems being analyzed, a standard set of symbols (as shown in Figure (1.2.1)) has been developed to represent the events and operations. This set of definitions is listed below and can be more fully understood by studying the sample fault tree shown in Figure (1.1.2). These definitions are given to facilitate the understanding of what follows in the thesis.

1. Basic Fault [1]: Basic fault is represented by a circle and requires no further dissection, since the probability of such an event is derived from empirical data; this probability is an input to the fault tree only when the quantitative analysis is performed.
2. Fault Assumed Basic to the Given Fault Tree [1]: This is represented by a diamond. Although this event can be further developed down to basic failures, it is not developed because of its lack of significance in such a fault tree, lack of details to develop it further, or availability of failure data.

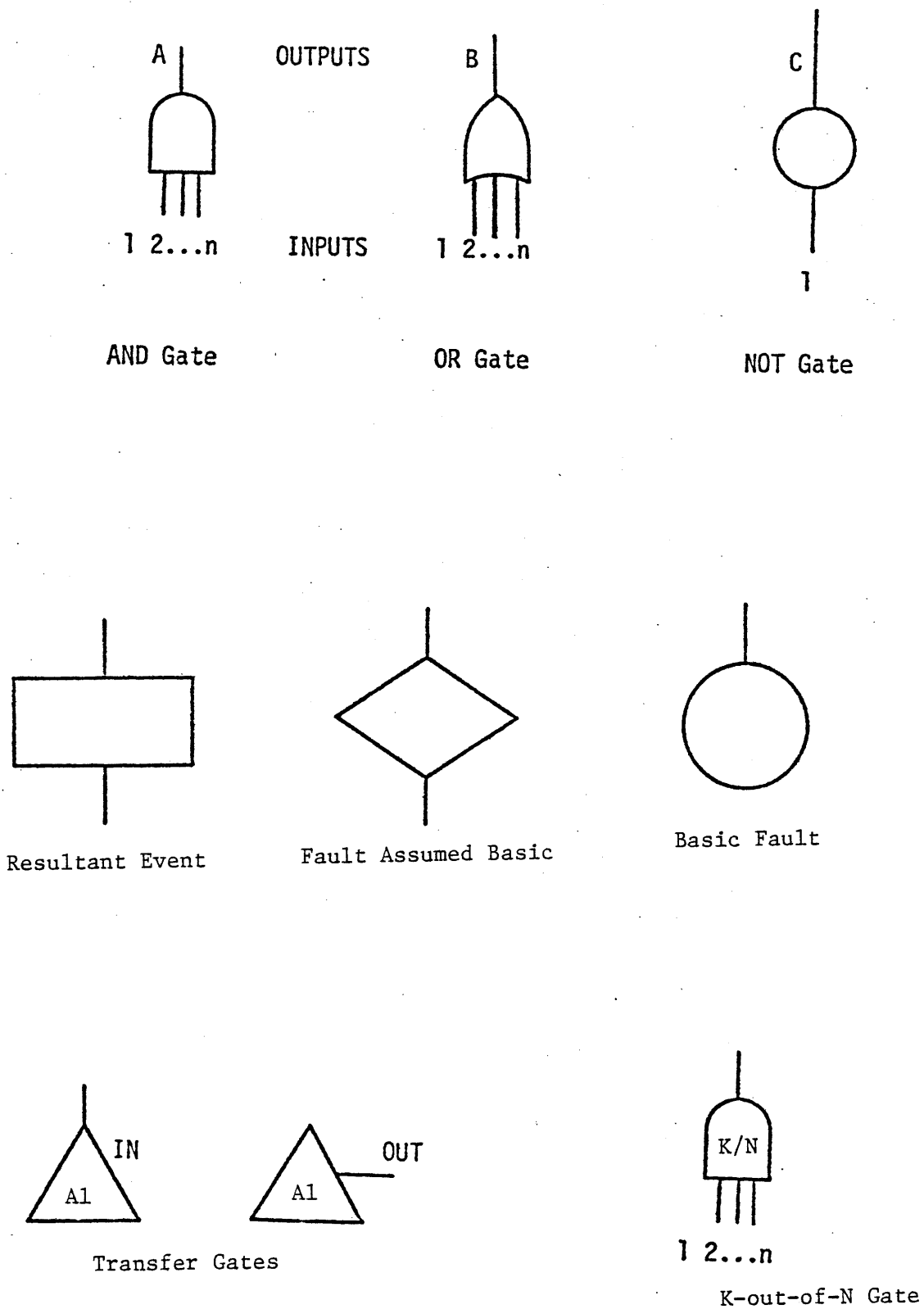


Figure 1.2.1 Fault Tree Symbols

3. Resultant Events: These events result from the combination of events of the types described above through the input of a logic gate. It is shown by a rectangle.
4. AND Gate: The AND gate represents the intersection of inputs to the gate; that is, an output event occurs if and only if all the inputs occur.
5. OR Gate: The OR gate represents the union of the inputs to the gate; that is, an output event occurs if only one of the inputs occurs.
6. K-out-of-N Gate: The K-out-of-N gate represents the condition that if any combination of K components out of the N input components occurs, it will result in the occurrence of the output from the K-out-of-N gate.
7. NOT Gate [35]: The NOT gate represents the complement of input to the gate; that is, an output event occurs if input has not occurred.
8. Transfer Gates [35]: These gates are only symbols to provide a tool to avoid repeating sections of the fault tree. The "transfer-out" gate represents the full branch that follows it, represented by a symbol, for example A1, and indicates that the branch is repeated somewhere else. The "Transfer-in"

gate represents the branch, here A1, that is already drawn somewhere else. Instead of drawing it again it is simply input at that point.

In order to avoid any misinterpretation and confusion with other definitions, some of the most frequent terms used in this thesis are explicitly defined below.

AVAILABILITY: The probability that an item, a subsystem, a module, or a system will be operational at a randomly selected future instant of time [1].

COMPONENT: A basic constituent of the system for which a failure is considered a primary failure (see the definition of a primary failure) in a particular fault tree, and has a unique defined function. Therefore, a circle, diamond, or rectangle in a fault tree could be considered a component; (e.g. a component can be anything from a wire to a complete system).

GATE DOMAIN: The set of all primary failures that logically interact to produce the resultant event of the gate in question.

FAILURE: The termination of the ability of an item or a system to perform its required function [1].

FAILURE RATE: The expected number of failures of a given type, per item, per unit of time [1] (e.g. a failure rate of 6×10^{-6} /hr for tube breaks represent 6 breaks per

million hours of operation time).

IMPORTANCE: Ranking of basic events and cutsets according to their contributions to the occurrence of the top event. Different methods of ranking exist which will be further discussed in following chapters.

MODULE: An assembly of basic components which can by itself be treated as a component of the system [29]. All basic components appearing in the module do not appear anywhere else except in the module.

MUTUALLY EXCLUSIVE EVENTS: Events that cannot exist simultaneously [1].

PRIMARY FAILURE: Failure of a component within the design envelope, i.e., failure due to the inherent characteristics of the system's element under consideration.

RELIABILITY: The characteristic of an item expressed by the probability that it will perform a required function under stated conditions for a stated period of time [1].

REPLICATED COMPONENT: A component of the system whose failure has multiple effects on the system. In a fault tree, a replicated component appears more than one time in different domains of the tree.

RISK: The expected detriment per unit time to a person or population from a given cause [1].

REPLICATED GATE: A gate that is repeated more than once in a fault tree.

SECONDARY FAILURE: Failure of a component outside its design envelope, i.e., failure due to excessive environmental characteristic or operational stress on the component, (e.g. failure of a component due to excessive heat of an accidental fire).

SYSTEM: An orderly arrangement of components that performs some task or functions.

TEST DURATION: Required time to inspect and test an item, a subsystem, or a system.

TEST INTERVAL: The elapsed time between repetitions of an identical test of the same item or system.

UNAVAILABILITY: The numerical complement of availability.

UNRELIABILITY: The numerical complement of reliability.

1.3 Fault Tree Evaluation

Fault tree evaluation consists of the analysis of the tree once it has been drawn. The evaluation can be either qualitative, quantitative, or both.

1.3.1 Qualitative Evaluation

Qualitative evaluation of a system fault tree consists basically of identifying "minimal cutsets" of the tree. A cutset is a set of basic

events whose simultaneous occurrence causes the top event to occur. A cutset is minimal if it cannot be reduced and still insure the occurrence of the top event. Minimal cutsets are very helpful in identifying different modes of system failure, and determining the design inadequacies.

The number of the cutsets dramatically increases with the size of the fault tree. For a complex fault tree the number of the cutsets is very large, thousands or more, and most of the time it is simply impossible to generate all of the cutsets in a reasonable amount of computing time.

Different methods have been proposed for the generation of the cutsets and in Appendix A some of these methods are summarized and discussed. Fault trees with only AND, OR and K-out-of-N gates are called coherent fault trees. However, some systems include gates other than AND and OR, e.g. EOR, NAND, NOR, or NOT gates. For analysis of these systems the modes of failure of any given system are no longer referred to as the cutsets, because the system fault tree becomes non-coherent. In section 1.5.1, a description of coherent and non-coherent fault trees is represented. In the non-coherent fault trees the concept of minimal cutset should be replaced by the concept of "prime implicants" [22] in Boolean Algebra. Similar to cutsets, a complete set of prime implicants gives all the failure modes of a non-coherent fault tree representing the system. Fortunately, for most complex technical systems, it is possible to represent the failure modes through the use of only coherent fault trees.

Minimal cutsets are often closely associated with each other, that is, there is a certain set of cutsets' components which are

repeated in all cutsets. If these sets of components are replaced by a super-component (i.e. a module) then minimal cutsets may be represented by "modular cutsets". However, modular cutsets can be directly found from the fault tree without generating the cutsets [23]. Modular cutsets are important because they often greatly reduce the number of cutsets needed to represent the fault tree. The method used in this study will take advantage of this property of modular cutsets.

1.3.2 Quantitative Evaluation

A second goal of the fault tree analysis is to provide a method for calculating the probability of the occurrence of the top event. Fault trees can also be used to calculate the relative importance of minimal cutsets to the top event, or the relative importance of a specified basic event to the top event.

To calculate the probability of the top event based on the minimal cutsets, one can use the Inclusion-Exclusion principle. For example, let, $K_1, K_2, \dots, K_n$ be a minimal cutset K_s of basic events for a fault tree [24]. By assuming that all basic events are subject to only primary failure, and are independent of the failure of each other, (i.e. they are statistically independent), then it follows that the probability that cutset K_i occurs would be:

$$\Pr(E_i) = \prod_{i \in K_s} P_i \quad 1.3-1$$

If the fault tree has n minimal cutsets, then

$$\Pr(\text{TOP EVENT}) = \Pr(T) = \Pr(C_1 \cup C_2 \cup C_3 \dots C_n) \quad 1.3-2$$

By the Inclusion Exclusion principle, Equation (1.3-2) can be developed as:

$$\begin{aligned} \Pr(T) = & \sum_{i=1}^K \Pr(C_i) - \sum_{i<j}^K \Pr(C_i \cap C_j) + \sum_{i<j<\ell}^K \Pr(C_i \cap C_j \cap C_\ell) \\ & + \dots + (-1)^{K-1} \Pr(C_1 \cap C_2 \cap C_3 \dots C_n) \end{aligned} \quad (1.3-3)$$

or

$$\Pr(T) = \sum_{r=1}^K (-1)^{r-1} i_r \quad (1.3-4)$$

where

$$i_r = \sum_{j_1 < j_2 < j_3 \dots < j_r} \Pr(C_{j_1} \cap C_{j_2} \dots \cap C_{j_r})$$

therefore,

$$P(T) \leq i_1 = \sum_{j=1}^K \prod_{i \in K_j} g_j \quad (1.3-5)$$

and

$$P(T) \geq i_1 - i_2 \quad (1.3-6)$$

Therefore, by using Equations (1.3-1) through (1.3-6) one would be able to calculate from the cutsets of a fault tree, the bounds on the probability of occurrence of the top event. However, the exact top event probability for statistically independent events in a cutset

would be obtained from the following equation:

$$\Pr(T) = \prod_{1 \leq i \leq K} \prod_{j \in K_s} q_j \quad (1.3-7)$$

It is clear that the use of equations (1.3-5) and (1.3-6) is more convenient than equation (1.3-7). However, for calculating the exact top event probability the use of equation (1.3-7) is inevitable.

It is clear, however, that the assumption of statistical independence is not always valid. In cases where one failure can cause another, often called common cause failure, changes in the analytical procedure are required. This will be the subject of Chapter 6 of this work. In case the components in a cutset are dependent, one can find the following useful bounds on the probability of the top event [25].

$$\max_{i \in K_s} \prod q_i \leq \Pr(\text{Top Event}) \leq \min_{i \in P_r} \prod q_i \quad (1.3-8)$$

1.4 Fault Tree Modularization

Fault tree modularization techniques have recently become very attractive and have proven to be a very efficient and reliable technique for the analysis of complex systems. This is because the modular fault tree is a compact representation of the original fault tree, and calculations based on the modular fault tree are much simpler than those based on the original tree. In this section some of the important methodologies for modularizing fault trees are discussed. Particularly, PL-MOD's methodology will be presented along with its advantages over other modularization techniques.

1.4.1 Introduction to the Modularization Technique and its Mathematical Concepts

Birnbaum and Esary [29] define a module as follows:

"A module of a system is a subset of basic components of that system which are organized into some structure of their own and which affect the system only through the performance of their subtree. Rephrasing, a module is an assembly of components which can by itself be treated as a component of the system."

Before getting into the mathematical concepts of modularization technique, and since our prime interest is coherent systems, a brief mathematical formulation of the characteristics of coherent systems will be presented.

Coherent binary systems are systems whose performance improves as the performance of their components improves. Suppose a set $C = \{c_1, c_2, \dots, c_n\}$ describes a set of components in a system. In the binary model each component is capable of two performance states, either functioning or failing. The joint performance of the components in C is indicated by

$$\vec{x} = (x_1, x_2, \dots, x_n),$$

where $x_i = 1$ indicates that component c_i functions and $x_i = 0$ indicates that c_i has failed. The whole system is capable of these two states, and the performance of the system can be represented by the so-called

Structure Function ϕ corresponding to the joint component performance $\underline{x}$. If $\phi(\underline{x}) = 1$ the system is functioning, and if $\phi(\underline{x}) = 0$ it fails. Thus, the system described above can be completely represented by the notation (C, ϕ) .

A coherent system is a system (C, ϕ) for which two conditions are satisfied. These conditions are as follows.

$$1. \quad \phi(\underline{x}) \leq \phi(\underline{y}) \text{ whenever } \underline{x} \leq \underline{y} \text{ (i.e., } x_i \leq y_i \text{)} \quad (1.4-1)$$

This condition implies that the state of the system is not degraded by changing a component from a failed state to a working state.

$$\phi(\underline{0}) = 0, \text{ where } \underline{0} = (0, 0, \dots, 0). \text{ (i.e., a state where all components have failed);} \quad (1.4-2)$$

$$\phi(\underline{1}) = 1 \text{ where } \underline{1} = (1, 1, \dots, 1). \text{ (i.e., a state where all components are functioning)} \quad (1.4-3)$$

2. Each component is essential to the system.

A component c_i is inessential to a system (C, ϕ) if the performance of the component has no effect on the performance of the structure, i.e. if

$$\phi(1_i, \underline{x}) = \phi(0_i, \underline{x}) \text{ for all } (\cdot_i, \underline{x}) \quad (1.4-4)$$

where

$$\begin{aligned}(1_i, \underline{x}) &= (x_1, \dots, x_{i-1}, 1, x_{i+1}, \dots, x_n) \\ (0_i, \underline{x}) &= (x_1, \dots, x_{i-1}, 0, x_{i+1}, \dots, x_n) \\ (\cdot_i, \underline{x}) &= (x_1, \dots, x_{i-1}, \cdot_i, x_{i+1}, \dots, x_n)\end{aligned}$$

Since non-essential components can be deleted from a system with no effect, they can be excluded from the tree. Therefore, we always assume that

$$\phi(1_i, \underline{x}) > \phi(0_i, \underline{x}) \text{ for all } (\cdot_i, \underline{x}) \quad (1.4-5)$$

In a coherent system fault tree the top event may always be represented by the Boolean Structure Function $\phi(\underline{x})$. Therefore,

$$\phi(\underline{x}) = \begin{cases} 1 & \text{if the top event occurs} \\ 0 & \text{otherwise} \end{cases} \quad (1.4-6)$$

Since AND and OR gates represent intersection and union of input components, respectively, the vector $\phi(\underline{x})$ will be defined as follows:

$$\phi \text{ AND }(\underline{x}) = x_1 \cdot x_2 \dots x_n = \prod_{i=1}^n x_i \quad (1.4-7)$$

$$\phi \text{ OR }(\underline{x}) = 1 - (1 - x_1)(1 - x_2) \dots (1 - x_n) = \prod_{i=1}^n (1 - (1 - x_i)) \quad (1.4-8)$$

A cutset of a coherent binary system (C, ϕ) is a set of components $Q \subseteq C$ such that $\phi(\underline{0}^Q, \underline{1}^{Q'}) = 0$, where Q' is the complement of Q (i.e., $C = Q \cup Q'$), and $\underline{0}^Q$, and $\underline{1}^{Q'}$ are the component performance vectors for which $x_i = 0$ when $C_i \in Q$, and $x_i = 1$ where $C_i \in Q'$. A cutset Q is a minimal cutset of (C, ϕ) if there is no cutset P such that $P \subset Q$.

From the above definition for a coherent system and by using the definition of a module, one can find [29] the Structure function of a system using its modules. Consider a subset A of components in the system (C, ϕ) , (i.e., $A \subset C$), such that A is not empty nor exhaustive. A is a modular set of system (C, ϕ) is

$$\phi(\underline{x}) = \phi(\underline{x}_A^A, \underline{x}^{A'}) = \psi[x_A(\underline{x}_A^A), \underline{x}^{A'}] \quad (1.4-9)$$

where

$$A' \cup A = C$$

(A, X_A) is a coherent system, and ψ is the structure function of a coherent binary system whose components are $C_i \in A'$. When A is a modular set we say that (A, X_A) is a module of (C, ϕ) .

The application of this definition for a fault tree is demonstrated for the sample fault tree given in Figure (1.4.1). Some of the modules in this fault tree are as follows.

$$\begin{aligned} A_1 &= \{a, b, c\}, & X_{A_1} &= x_a \cup x_b \cup x_c \\ A_2 &= \{a, b, c, d, e, f\}, & X_{A_2} &= (x_a \cup x_b \cup x_c) \cap (x_d \cup x_e \cup x_f) \\ A_3 &= \{g, h, i\}, & X_{A_3} &= x_g \cup x_h \cup x_i \\ A_4 &= \{a, b, c, d\}, & X_{A_4} &= (x_a \cup x_b \cup x_c) \cap x_d \\ &\vdots & & \\ &\vdots & & \\ &\vdots & & \end{aligned}$$

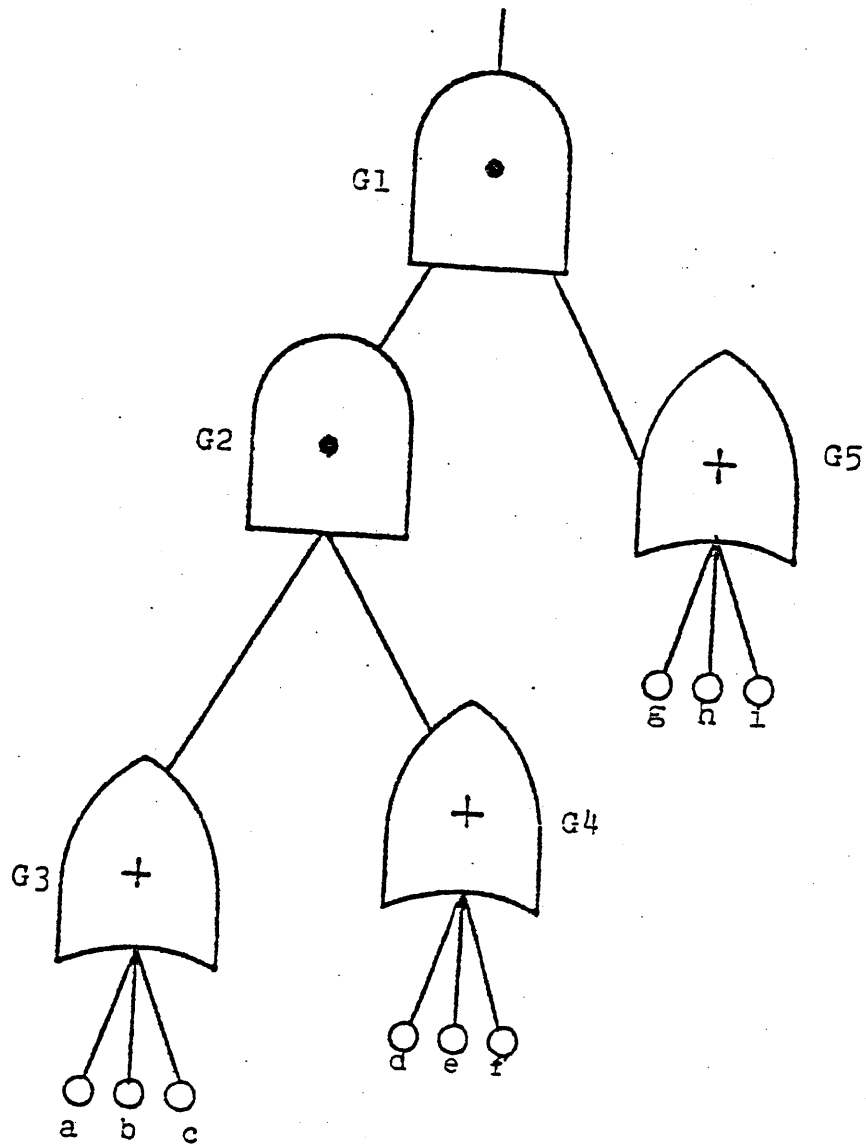


Figure 1.4.1 Sample Sub-Tree I with No Replications [23]

One of the most useful representations of a Boolean function is its pivotal decomposition about one of its coordinates, i.e.,

$$\phi(\underline{x}) = x_i \phi(1_i, \underline{x}) + (1 - x_i) \phi(0_i, \underline{x}) \quad (1.4-10)$$

If (A, X_A) is a module of (C, ϕ) , then from Equation (1.4-9) it follows that:

$$\phi(\underline{x}) = \chi_A(\underline{x}^A) (1, \underline{x}^{A'}) + [1 - \chi_A(\underline{x}^A)] \psi(0, \underline{x}^{A'}) \quad (1.4-11)$$

Equation (1.4-11) is a general form of representing mathematically the relations between the system modules and the system itself. The method of finding the modules by using the system fault tree is not unique. In the next section some of the potential methods available for finding the modules are discussed.

1.4.2 The PL-MOD Methodology

The method of modularization that is utilized in the code PL-MOD [23] to decompose a fault tree is based upon the definitions given in the previous section (1.4.1). In terms of fault tree decomposition, an intermediate gate event will be a module to the top event if the basic events contained in the domain of the gate do not appear elsewhere in the fault tree. This module would, therefore, act as a "super-component". In this case, the modularization of fault trees having no replicated events can be easily accomplished, since all of the intermediate gates may be considered as independent modules input to the top event. For trees with replicated components the procedure is much more complicated.

This case will be discussed later in this section.

If a fault tree is modularized such that no further modularization is possible, then the fault tree has reached its Finest Modular Representation. The Finest Modular Representation of a fault tree must have the following conditions.

1. All the branches of the fault tree are independent.
2. The logic function $\chi_A(x^A)$ associated with each gate is either "prime" or "simple", having no inputs from other "simple" gates of the same type.

Higher order "prime" gates are defined to be Boolean logic functions which are not further modularizable. Prime logic functions are thus characterized by an irreducible set of Boolean cutset vector equations.

As a result, from the aforementioned conditions, and for obtaining the finest modular representation of the fault tree shown in Figure (1.4.1), by using the PL-MOD's methodology one should change the fault tree representation of this figure to the one shown in Figure (1.4.2). This change is obtained by coalescing gates G_1 and G_2 and, as a result, its modular structure is given by the following set of recursive equations:

$$\begin{aligned}
 A_1^{\text{AND}} &= \{A_3, A_4, A_5\}, \quad (\vec{x}) = \chi_{A_1} = \chi_{A_3} \cdot \chi_{A_4} \cdot \chi_{A_5} \\
 A_3^{\text{OR}} &= \{a, b, c\}, \quad \chi_{A_3} = x_a \cup x_b \cup x_c \\
 A_4^{\text{OR}} &= \{d, e, f\}, \quad \chi_{A_4} = x_d \cup x_e \cup x_f \\
 A_5^{\text{OR}} &= \{g, h, i\}, \quad \chi_{A_5} = x_g \cup x_h \cup x_i
 \end{aligned}$$

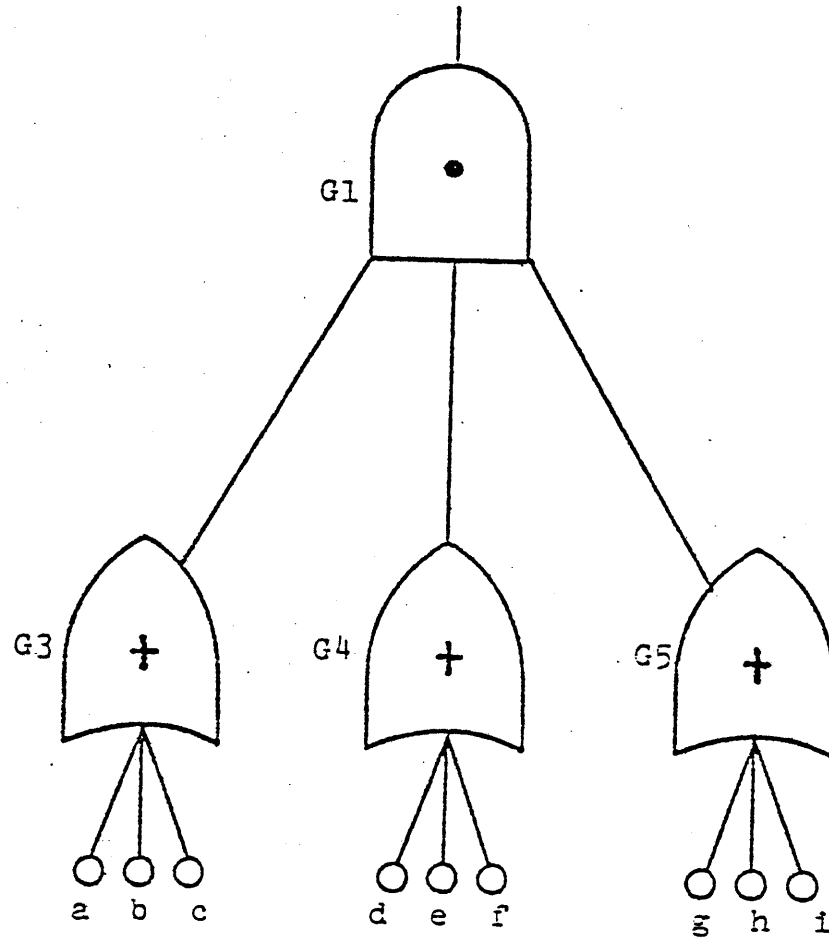


Figure 1.4.2 Finest Modular Representation of Sample Sub-Tree [23]

Suppose that the fault tree in Figure (1.4.1) has a replicated component as shown in Figure (1.4.3). Then, one must first realize that events $\{a,b\}$, $\{g,i\}$, and $\{d,e,f\}$ form modules associated with simple OR gates. Therefore:

$$\begin{aligned} A_3^{\text{OR}} &= \{a,b\}, \chi_{A_3} = x_a \cup x_b \\ A_4^{\text{OR}} &= \{d,e,f\}, \chi_{A_4} = x_d \cup x_e \cup x_f \\ A_5^{\text{OR}} &= \{g,i\}, \chi_{A_5} = x_g \cup x_i \end{aligned}$$

It should be noted that in the subset A_3 , A_4 and A_5 there are only non-replicated events. Furthermore, these modules, together with the replicated event r , will become the input to a higher prime gate G , as shown in Figure (1.4.4). Note that gate G is not a simple AND gate, rather it is a special gate representing a set of Boolean vectors for input component and modules. The structure function for gate G would therefore be:

$$\phi(\underline{x}) = (x_r, \chi_{A_3}, \chi_{A_4}, \chi_{A_5})$$

with the following set of Boolean vectors for the occurrence of gate G (i.e. Modular cutsets):

$$S_1 = (1, 0, 1, 0)$$

$$S_2 = (0, 1, 1, 0)$$

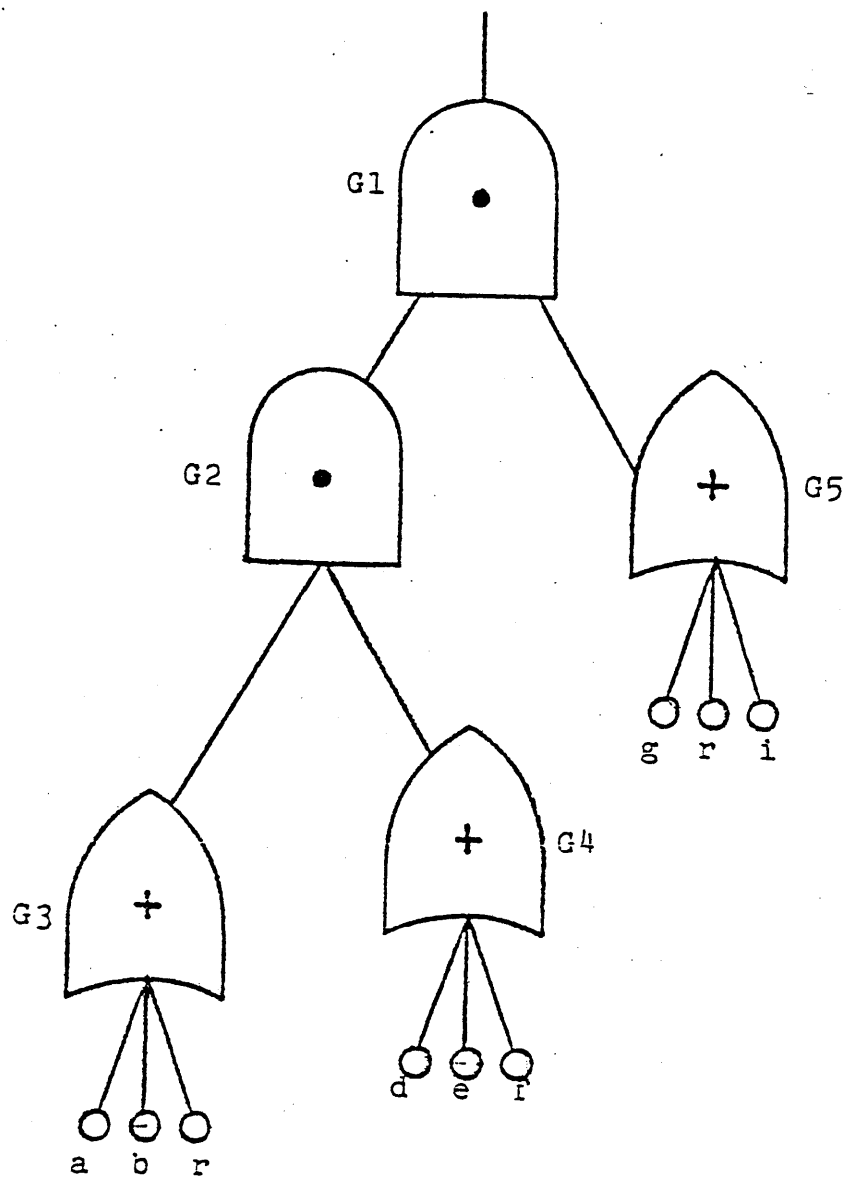


Figure 1.4.3 Sample Sub Tree II with Replications [23]

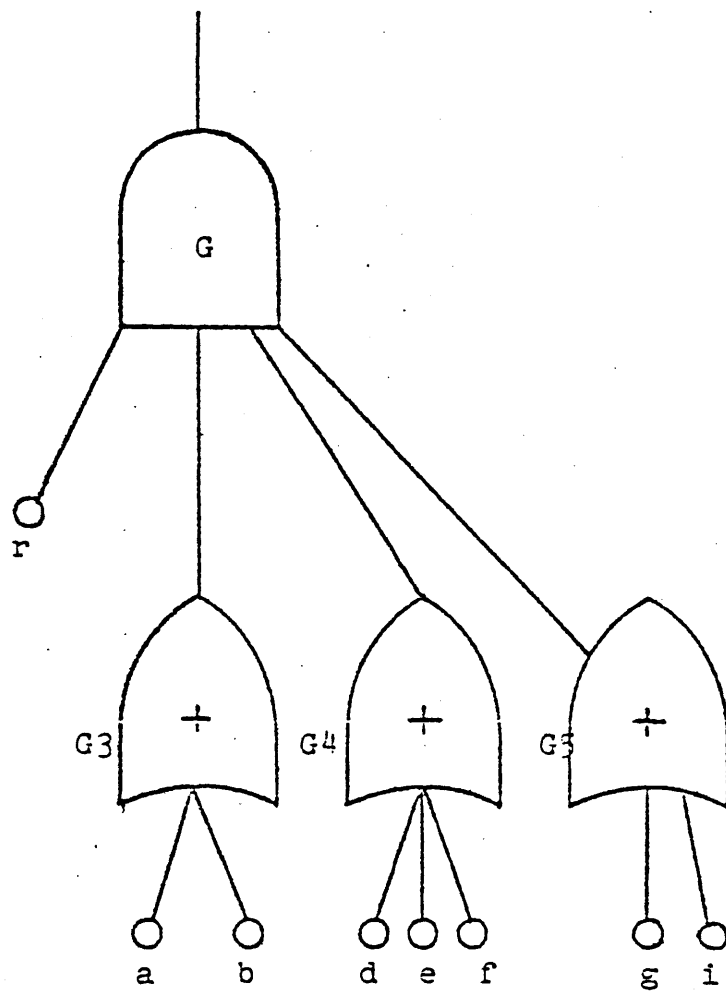


Figure 1.4.4 Finest Modular Representation of Sample Sub Tree II [23]

Obviously, these Boolean vectors represent the following modular cutsets:

- (1) r, A_4
- (2) A_3, A_4, A_5

These two modular minimal cutsets are a compact representation of the usual basic event minimal cutsets (i.e. simple cutsets) for the fault tree in Figure (1.4.3).

These simple cutsets are as follows:

- (1) r, d
- (2) r, e
- (3) r, f
- (4) a, d, g
- (5) a, d, i
- (6) a, e, g
- (7) a, e, i
- (8) a, f, g
- (9) a, f, i
- (10) b, d, g
- (11) b, d, i
- (12) b, e, g
- (13) b, e, i
- (14) b, f, g
- (15) b, f, i

Based on the methodology described above, the Code PL-MOD is constructed to modularize the tree and find the modular minimal cutsets, as well as probability calculations based on the modular minimal cutsets.

The PL-MOD Code is written in PL/I language, because PL/I is the only language that can handle the algorithms discussed above most effectively. Also, PL/I provides several features normally found only in Assembler or list processing languages. The essence of list processing is the ability to dynamically allocate blocks of core storage to link these blocks together into a structure to store and retrieve data from blocks. It should be noticed that list processing for complicated data structures such as those required by PL-MOD are very difficult, if not impossible, to achieve through manipulation using FORTRAN. In Figure (1.4.5) a flow chart is given for the structure of the PL-MOD code. The fault tree modularization is achieved by performing a series of manipulations on its nodes outlined below:

- (a) Each NODE in the fault tree is defined as a gate operator (AND, OR, or K-out-of-N) together with a set of attached input gates and basic event components.
- (b) A NODE's output will be an input to another NODE (except to the top event), defined to be its NODE.ROOT.
- (c) NODEs having common replicated inputs are interconnected. These interconnections then identify sets of nodes which are not immediately modularizable in the original form of the fault tree.
- (d) The true modular decomposition is simultaneously started at all bottom gate nodes defined to be those having no gate inputs (GATELESS NODES).

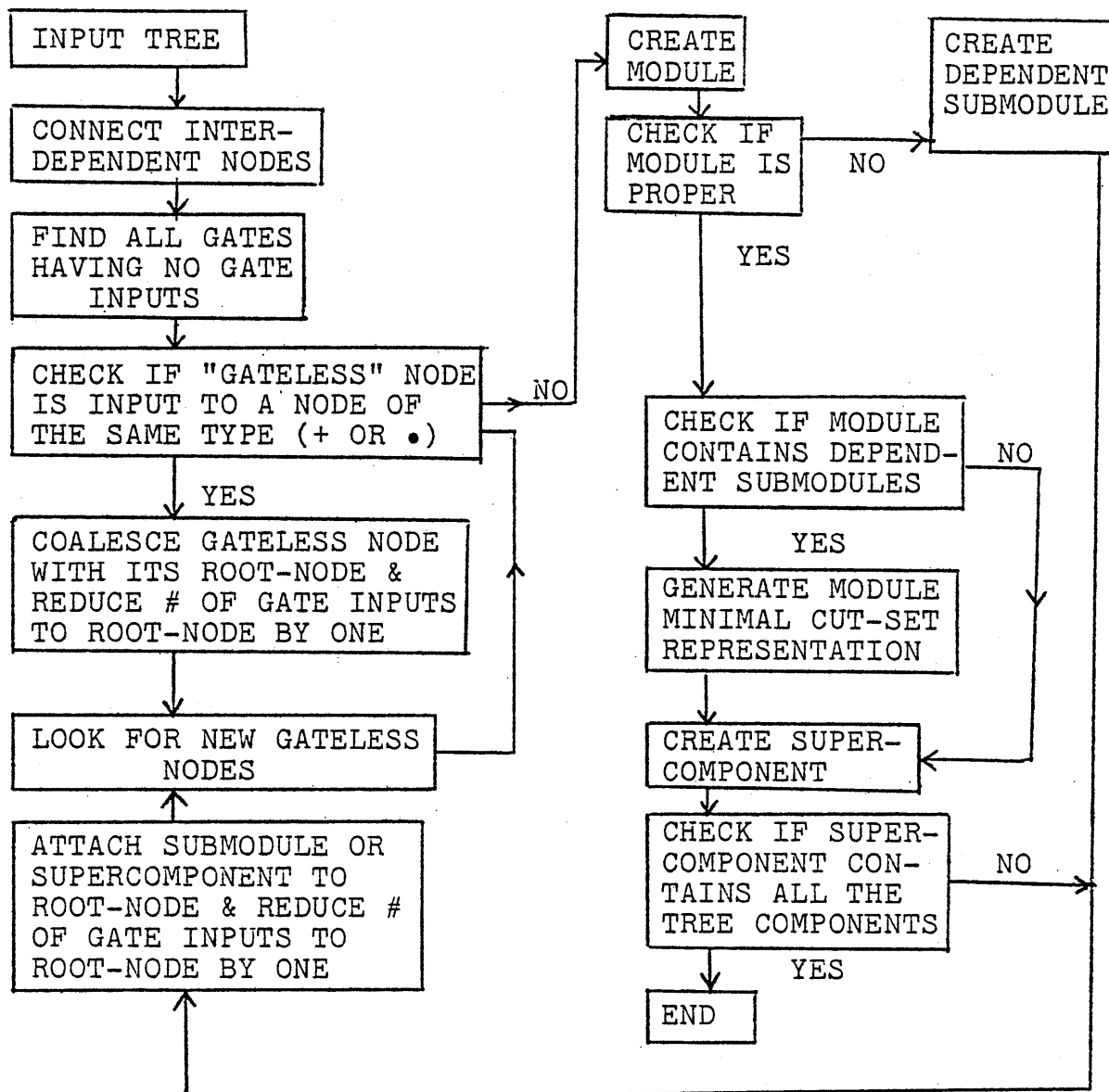


Figure 1.4.5 Fault Tree Modularization Algorithm

- (e) Simple (AND,OR) gateless nodes having as their NODE.ROOT another gate of the same type are coalesced with their NODE.ROOT by transferring all their inputs to the NODE.ROOT
- (f) Simple gateless nodes having a gate of a different type NODE.ROOT are modularized. Those gateless nodes having replicated components or submodules as inputs are temporarily transferred into "nested modules", unless they are found to contain a set of replicated events within the gate. In this case they are complete, and modular minimal cutsets can be determined.
- (g) Symmetric (K-out-of-N) gate NODEs are immediately modularized.
- (h) NODEs which have been transferred into proper modules or temporary nested submodules are attached to their NODE.ROOT.
- (i) As steps (e), (f), (g) and (h) reduce the number of gate inputs to a gateless node, a new set of gateless nodes will necessarily be obtained. Therefore, steps (e) through (h) will be successively repeated until the top event is reached.

Once the modular structure of the fault tree has been obtained, the quantitative evaluation of reliability and importance parameters of the tree can be more efficiently performed because of the reduction in the calculations. In particular, the probability of

the occurrence of the top event is obtained by means of a series of recursive calculations requiring the evaluation of the probability expectation value of each module combined in the tree.

From the foregoing evidence the computational advantages of using the PL-MOD's methodology to evaluate fault trees in a modular manner are:

- (a) The probabilities of the occurrence of the top and intermediate gate events are efficiently computed by evaluating these modular events in the same order in which they are generated.
- (b) The modular and component importance measures are easily computed by starting at the top tree event and successively using a modular importance chain rule.
- (c) The modular minimal cutsets represent all of the simple cutsets in a compact form. For qualitative fault tree evaluation, such as common cause failure analysis, the modular minimal cutsets can be efficiently used.

1.4.3 Other Modularization Methodologies

In this section two modularization methods are described; one is Chatterjee's method and next the PATREC-RCM's method.

1.4.3.1 Chatterjee's Method

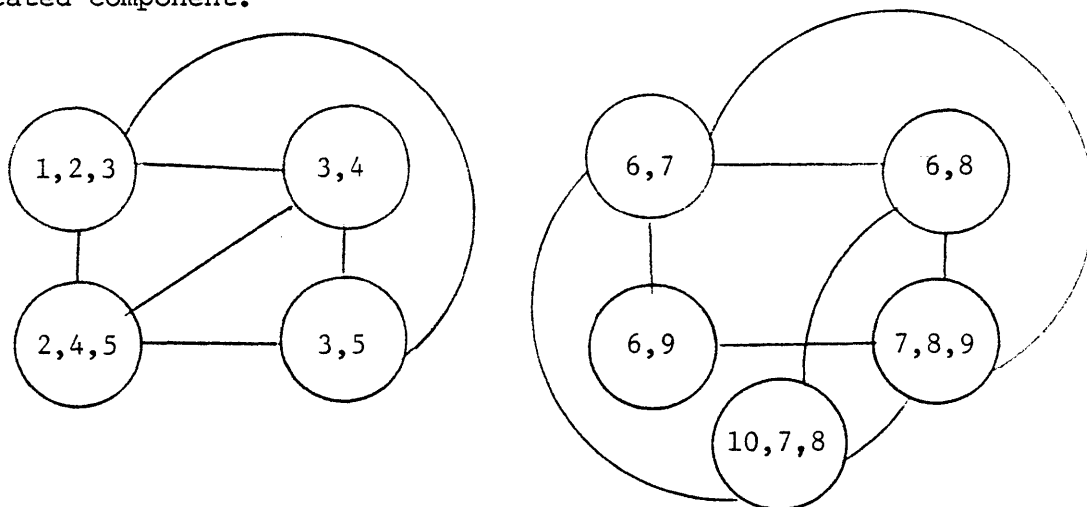
Chatterjee's algorithm [30,31] finds an equivalent representation of a given fault tree in terms of the finest modular representation of that tree. This algorithm is shown in Figure (1.4.6).

Minimal cutsets disjoint modules described in Step 2 of Figure (1.4.6) and obtained as follows:

For example, suppose the cutsets

$K = \{(1, 2, 3), (3, 4), (2, 4, 5), (3, 5), (6, 7), (6, 8), (6, 9), (7, 8, 9), (10, 7, 8)\}$ are given for a fault tree.

The disjoint min cutset modules can be obtained by using the following graph, and by connecting cutsets that share at least one replicated component.



It is evident that, due to the definition of a module, those sets of components that are independent of each other can be put into a module. Obviously, components (1, 2, 3, 4, 5) and (6, 7, 8, 9, 10) constitute two modules of the system because they are totally independent of each other.

It is evident that this algorithm is different from that of PL-MOD for deriving the modular decomposition of the fault tree. In Chatterjee's method the modules are found from minimal cut and path set of a fault tree whereas, in PL-MOD's method, the modules are found directly from the fault tree diagram. The obvious advantage of the latter is that there is no necessity to go through the time consuming cutset determination prior to the modularization procedure.

The Code FTAP (see Appendix A) finds the families of minimal cutsets based on the simple modules (easy modules), and is capable of generating the cutsets such that they can be most conveniently used in Chatterjee's algorithm for modularizing a tree (i.e. FTAP performs step 1 and obtains the MCS of the system shown in Figure (1.4.6)).

In summary, the method of fault tree modularization developed by Chatterjee is not a potentially useful algorithm for following reasons:

- (a) the algorithm is not wholly automated in the form of a computer program
- (b) the algorithm consists of determining all minimal cutsets, which by itself is very time consuming
- (c) the use of this algorithm for complex fault trees is very difficult, if not impossible; due to the presence of a large number of minimal cutsets

- (d) probability calculations from the modules obtained by using simple cutsets are more time consuming than calculating probabilities from the cutsets directly. Because, once the cutsets are obtained, calculating probabilities constitutes a negligible fraction of the computation time necessary to evaluate the fault tree through its cutsets.

1.4.3.2 PATREC-RCM's Method

PATREC-RCM [32] is a computer code based on a modularization algorithm, and is the new version of the PATREC Code [33]. The latter does not handle replicated components. The PATREC-RCM algorithm does not attempt to determine minimal cutsets or modular minimal cutsets. Rather, PATREC-RCM calculates unavailabilities directly from the modularized fault tree diagram. The effect of the presence of replicated events will be treated by a mathematical formulation rather than a logical disorientation of the fault tree (e.g. similar to the PL-MOD and Chatterjee's method).

The modularization algorithm used in the code PATREC-RCM is based on a complete modularization without treating any dependences between the components. In this case the modularization process is very simple, and is identical to the PL-MOD modularization process when no replicated component exists in the fault tree. Therefore, by starting from the bottom of the tree and replacing all simple modules by a "super-component" and repeating this procedure, the whole tree will be finally represented by a single "super-event".

As was discussed earlier, the effect of the presence of replicated components is treated mathematically during the probability calculations by the PATREC-RCM. Suppose the probability corresponding to the modular structure in terms of the primary events is as follows:

$h(\vec{P}) = h(P_a, P_b, \dots, P_{x_1}, P_{x_2}, \dots, P_{x_n})$ in which $x_1, x_2, \dots, x_n$ are replicated components, and suppose

$$h(1_i, \vec{P}) = (P_a, P_b, \dots, P_{x_1}, \dots, P_{x_i} = 1, \dots, P_{x_n}) \text{ and}$$

$$h(0_i, \vec{P}) = (P_a, P_b, \dots, P_{x_1}, \dots, P_{x_i} = 0, \dots, P_{x_n}).$$

one obtains the probability P of the occurrence of the top event by applying Bayes' theorem:

$$h_i(\vec{P}) = P_{x_i} h(1_i, \vec{P}) + (1 - P_{x_i}) h(0_i, \vec{P}) \quad (1.4-12)$$

Therefore, a single module representing the top event of a fault tree with r different replicated events could be replaced by 2^r "super-events" with no replicated events. Therefore, the total probability of the top event would be as follows:

$$\text{Pr(Top Event)} = \sum_{i=1}^r h_i(\vec{P}) \quad (1.4-13)$$

The efficiency of this method in PATREC-RCM is therefore, highly dependent upon the number of replicated events. For large fault trees with a large number of replicated events equation (1.4-12) must be evaluated for each replicated component. Therefore, the time consumed by the computer becomes prohibitive.

In summary, the use of the PATREC-RCM Code to calculate probabilities highly depends on the system fault tree. The PATREC-RCM Code is not capable of making any qualitative fault tree evaluation, because the modular minimal cutsets, or simple cutsets, are not found by the code. Further, for trees with many replicated events, the quantitative calculations cannot be efficiently performed.

1.5 Objectives of This Study

In section (1.4) the PL-MOD methodology along with the Chatterjee and PATREC-RCM methods were discussed. In general, the PL-MOD methodology has the advantage of deriving modular minimal cutsets directly from the fault tree diagram in an efficient and effective manner. Chatterjee's methodology is simply not practical and economical for large fault trees, due to the need for determining minimal cutsets prior to the modularization process. The PATREC-RCM's methodology is capable only of finding probability of the top event, and therefore is not suitable for any qualitative analysis. Even PATREC-RCM's methodology is incapable of finding the top event probability for a large fault tree consisting of many replicated events, because of the numerous mathematical calculations involved.

The objective of this thesis is mainly concentrated at developing the methodology and capabilities of the PL-MOD Code. For a complete description of the PL-MOD code the reader is referred to Appendix A of this thesis and reference [23]. The objectives of this research can be divided into the following three categories:

- i) removal of some of the important restrictions in the PL-MOD Code;
- ii) addition of more capabilities to the PL-MOD Code;
- iii) investigation for potential use of the PL-MOD Code series in the present reliability and safety analysis problems.

Developments under Category (i) will include the following efforts:

- (1) PL-MOD can only handle replicated modular gates that are independent from all other replicated gates and the fault tree itself. In other words, replicated components can only appear within the domain of a replicated gate or the top event. In practice, however, there are many fault trees where the replications are outside the replicated gate domain. This restriction will be removed in the context of this research.
- (2) PL-MOD allows the explicit treatment of symmetric (K-out-of-N) gates only if the inputs to these gates are non-replicated component or super-component events. An attempt will be made in this thesis to eliminate this restriction.
- (3) The input and output to the PL-MOD Code is based on the numerical values assigned by the user. In addition, much information about the fault tree must be applied

as input, such as number of gates, number of components, and number of replicated components. Replicated components must be given special numbers and the number of their occurrences must be specified. A change had to be made such that this input information is calculated by the computer and the user would then only input the logical corrections of the fault tree by using alpha-numeric names.

Developments under Category (ii) include the following efforts:

- (1) PL-MOD analyzes deterministically steady-state probabilities for the top event. To account for maintenance repair and test, a time-dependent (kinetic) tree analysis is implemented during the course of this research. The result of this development is the PL-MODT Code (T stands for Transient).
- (2) PL-MOD lacks the capability of accounting for uncertainties associated with the basic event input probabilities. A Monte-Carlo package is therefore implemented into this code, resulting in the version of PL-MODMC to account for uncertainties associated with component failure rates, repair time, test duration,....., etc. (MC stands for Monte-Carlo.)
- (3) PL-MOD determines the modular cutsets of a tree through its input description of the fault tree diagram. Thus far, these have not been reduced to the simple cutsets commonly

used in the FTA. Rather, all simple cutsets are buried in the modular cutsets. However, for the qualitative analysis of fault trees, one needs fault tree analysis codes in order to present the adequacy and efficiency of the methods developed in this thesis. This task has been performed by developing the MODCUT code.

- (2) For large systems, the identification of cutsets affected by common cause events are generally not feasible at the top event level due to prohibitively large numbers of cutsets. It is believed that the modularizations procedure developed for the PL-MOD Code would be well suited for the identification of common cause candidates. A thorough investigation of common cause analysis methods is performed in the context of using different computer codes, including MODCUT and MOCUS, for generating cutsets, and identifying common cause candidate cutsets. The possibility of using modular decomposition methodology in common cause analysis is also investigated.

Finally, it should be stressed that with the newly developed version of PL-MODT, PL-MODMC, and MODCUT a package has been obtained which is able to cover the broad spectrum of required tasks in present day reliability and fault tree analysis studies. Particularly, this would become more clear by reviewing the fault tree evaluation codes presented in Appendix A of this thesis. Most of these codes suffer a restriction of one kind or another in their application. In the

following chapters of this thesis, the mathematical concepts and computer implementation of these mathematical concepts are discussed.

REFERENCES

- [1] "IEEE Trial-Use Guide: General Principles for Reliability Analysis of Nuclear Power Generating Station Protection Systems" ANSI-N414, IEEE - std. 352-1972
- [2] Farmer, F. R., "Siting Criteria - A New Approach", paper presented at IAEA Symposium on the containment and Siting of Nuclear Power Plants, Vienna, SM-89/34, April, 1967
- [3] Garrick, B. J., "Unified Systems Safety Analysis for Nuclear Power Plants", Ph.D. Thesis, UCLA, 1968
- [4] Mulvihill, R. J., et. al., "A Probabilistic Methodology for Safety Analysis of Nuclear Power Plant", Planning Research Corporation, SAN-570-2, Vols. 1 and 2, February, 1966
- [5] "Reactor Safety Study", WASH-1400, U. S. Nuclear Regulatory Commission, (NUREG-75/014), 1975
- [6] "Risk Assessment Review Group Report to the U. S. Nuclear Regulatory Commission", NUREG/CR-0400, January, 1979
- [7] "Clinch River Breeder Reactor Project, preliminary safety analysis report", Project Management Corporation, July, 1975
- [8] "A Preliminary Assessment of Accident Risks in a Conceptual High-Level Waste Management System", Battelle, Richland, Washington, March, 1976
- [9] Rumble, E. T., F. L. Leverenz, Jr., R. C. Erdmann, and G. S. Lellouche "Generalized Fault Tree Analysis for Reactor Safety", Science Applications Inc., EPRI 217-2-2, June, 1975
- [10] Erdman, R. C., et. al., "Probabilistic Safety Analysis", Science Applications Inc., EPRI-217-2-4, July, 1975
- [11] Cook, B. M., "Use of Fault Tree Analysis in the Design of the Westinghouse Microprocessor Based Reactor Trip Logic System", Westinghouse Electric Corporation, presented in Probabilistic Analysis of Nuclear Reactor Safety, Newport Beach, May, 1978
- [12] Lambert, H., "Fault Trees for Decision Making in System Analysis", Lawrence Livermore Lab., Livermore, 1975
- [13] Caldarola, Wickenhauser, "Recent Advancements in Fault Tree Methodology at Karlsruhe International Conference on Nuclear System Reliability Engineering and Risk Assessment", Gatinsburg, Tennessee, June, 1977

- [14] Candolfo, C., et. al., "Some Aspects of the Risks Associated with a Mixed Oxide Fuel Production Plant", Topical Meeting on Probabilistic Analysis of Nuclear Reactor Safety, Newport Beach, CA, May, 1978
- [15] Leverenz, F. L., E. T. Rumble and R. C. Erdmann, "A Dependent Event Model for Fault Trees", Trans. Am. Nuclear Society, 21, 212, 1975
- [16] Amesz, J., et. al., "Validation of the Probabilistic Approach for the Analysis of PWR Transients", Topical Meeting Probabilistic Analysis of Nuclear Reactor Safety, Newport Beach, CA, May, 1978
- [17] Smith, T. H., et. al., "A Risk-Based Fault Tree Analysis Method for Identification, Preliminary Evaluation, and Screening of Potential Accidental Release Sequences in Nuclear Fuel Cycle Operations", BNWL-1959, Battelle (BNL) January, 1976
- [18] HTGR Accident Initiation and Progression Analysis Status Report, GA-A 13617, 1975
- [19] Wood, D. E. and N. J. Becar, "A Comparison of the Results from the GO Methodology and Fault Analysis", K77-38U(R), August, 1977
- [20] Apostolakis, G. and Y. T. Lee, "Methods for the Estimation of Confidence Bounds for the Top Event Unavailability of Fault Tree", Nuclear Eng. and Design, pp. 41-419, 1977
- [21] Cummings, G. E., "Application of the Fault Tree Technique to a Nuclear Reactor Containment System", Reliability and Fault Tree Analysis, SIAM 1975, pp. 805 - 825
- [22] Kumamoto, H., E. J. Henley, "Top-down Algorithm for Obtaining Prime Implicant Sets of No-coherent Fault Trees", IEEE Transaction on Reliability, Vol. R-27, No. 4, October, 1978
- [23] Olmos, J., "A Modular Approach to Fault Tree and Reliability Analysis", Ph.D. Thesis, MIT Dep. of Nucl. Eng., August, 1977
- [24] Barlow, R. E. and H. E. Lambert, "Introduction to Fault Tree Analysis", Reliability and Fault Tree Analysis, SIAM, 1975, pp. 7-35
- [25] Barlow, R. E. and F. Proschan, "Statistical Theory of Reliability and Life Testing", Holt, Rinehart and Winston, New York, 1975
- [26] "Fault Tree Handbook", U. S. Nuclear Regulatory Commission, 1978
- [27] Wilson, J. R., et. al., "COMCAN - A Computer Program for common cause Analysis", ANCR-1314, May, 1976
- [28] Wilson, J. R., "Selecting the 'Right' Common Cause Analysis Technique", ANS Transaction, 27, November, 1977, pp. 629-631

- [29] Birnbaum, Z. W. and J. D. Esary, "Modules of Coherent Binary Systems", J. Soc. Indust. Appl. Math., Vol. 13, No. 2, June, 1965
- [30] Chatterjee, P., "Fault Tree Analysis: Reliability Theory and Systems Safety Analysis", University of California, Berkeley, AD/A-004,209, Nov. 1974
- [31] Chatterjee, P., "Modularization of Fault Trees: A Method to Reduce the Cost of Analysis", Reliability and Fault Tree Analysis, SIAM, 1975, pp. 101-126
- [32] Koen, B. V., A. Carnino, et. al., "A Computer Code for Fault Tree Analysis: PATREC", Probabilistic Analysis and Nuclear Reactor Safety Topical Meeting, Newport Beach, CA, May, 1978
- [33] Koen, B. V., and A. Carnino, "Reliability Calculations with a List Processing Technique", "IEEE Trans. on Reliability, Vol. B-23, No. 1, April, 1974
- [34] "Methods for Reliability and Safety Engineering", Aerojet Nuclear Company, MAP #0015, Idaho Falls, Idaho, June, 1975
- [35] "A Computer-Oriented Approach to Fault-Tree Construction", EPRI-N,288, November, 1976
- [36] Wagner, D. P., "A Procedure for Qualitative Common Cause Failure Analysis of Complex Systems", M. S. Thesis, the University of Tennessee, August, 1977

CHAPTER 2

TIME DEPENDENT FAULT TREE ANALYSIS

2.1 Introduction

The use of modularization techniques in fault tree analysis provides an efficient and fast method to determine the unavailabilities of the modules and the top tree event. The modularization technique is specifically advantageous for the analysis of fault trees comprised of components with time-dependent behavior. This is because the time-dependent evaluation involves many calculations of the same kind at different time steps.

In this chapter, four different classes of components are considered for use in the PL-MOD Code for evaluating the fault tree. The four classes of components are:

1. Time-independent or constant unavailability components (i.e., $\bar{A} = \text{constant}$).
2. Nonrepairable components in which the failure rates are time independent.
3. Repairable components, failure of which is immediately detected (revealed faults). The failure and repair rates are time independent for this class of components (monitored components).
4. Repairable components, failure of which is detected upon inspection (periodically tested components).

A detailed discussion of these four classes of components is provided in Sections 2.2 through 2.5. These four classes of components are incorporated into the PL-MOD Code, and the new code is named PL-MODT, where the abbreviation, T, stands for the transient features of the code.

In the PL-MODT code, all features of the PL-MOD code are essentially kept the same. The new code is able to handle large fault trees with time-dependent components as well as small trees in an efficient, fast and economical way. During this development, an attempt has been made to incorporate most of the important features of the present time-dependent codes such as KITT [6] and FRANTIC[4] into PL-MODT.

2.2 Class 1 Components: Time Independent Components

A time-independent leaf (component) in a fault tree takes into account the presence of failures whose probability of occurrence does not change during the component's operation. These failures can occur as a result of physical phenomena such as operator faults, loss of power, faults due to the complementary components, etc. The occurrence of these faults has a specific likelihood which is not time dependent. For example, if a fault tree leaf could be in n -different failed states given that there exists only one non-failed state, then a Markovian model can be formulated such as the one shown in Figure (2.2.1). In this figure the different transition probabilities are given by α , β , γ and δ .

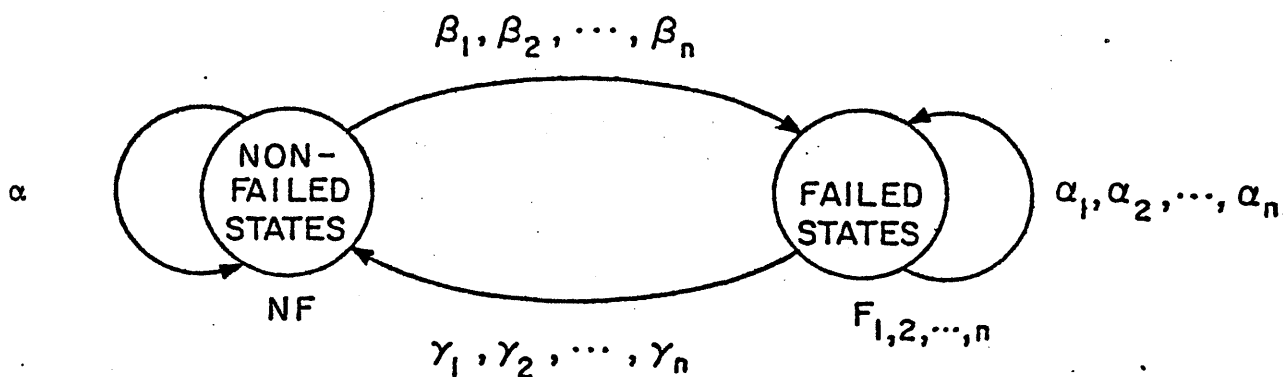


Figure 2.2.1

A Markovian Model for Time Independent Components

Each of these n failed states could be one of the failure modes of the time-independent component. For example, faults caused by an operator, accidental loss of power or by a missile produced from a turbine failure are three different mutually exclusive faults occupying three failed states of the components. Therefore the probability that the component is in its non-failed state can be calculated as follows. The transition matrix $\underline{A}$ is given by Equation (2.2-1) as follows:

$$\underline{A} = \begin{matrix} & \begin{matrix} NF & F_1 & F_2 & & F_n \end{matrix} \\ \begin{matrix} NF \\ F_1 \\ F_2 \\ \vdots \\ F_n \end{matrix} & \left(\begin{array}{ccccc} \alpha & \beta_1 & \beta_2 & \dots & \beta_n \\ \gamma_1 & \alpha_1 & 0 & \dots & 0 \\ \gamma_2 & 0 & \alpha_2 & \dots & \text{All zeros} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \gamma_n & \text{All zeros} & & & \alpha_n \end{array} \right) \end{matrix}$$

(2.2-1)

If $\underline{P} = [P_{NF} \ P_{F_1} \ \dots \ P_{F_n}]$, then the solution of the equation $\underline{P} \cdot \underline{A} = \underline{P}$ would provide a discrete value for P_{NF} which is also a time-independent value for the probability that the leaf is in its non-failed state. To illustrate this point consider the example given below.

Example: Figure (2.2.2) shows different transition probabilities for a component with one non-failed state and two failed states. The problem is to calculate the likelihood that the component is in its non-failed state.

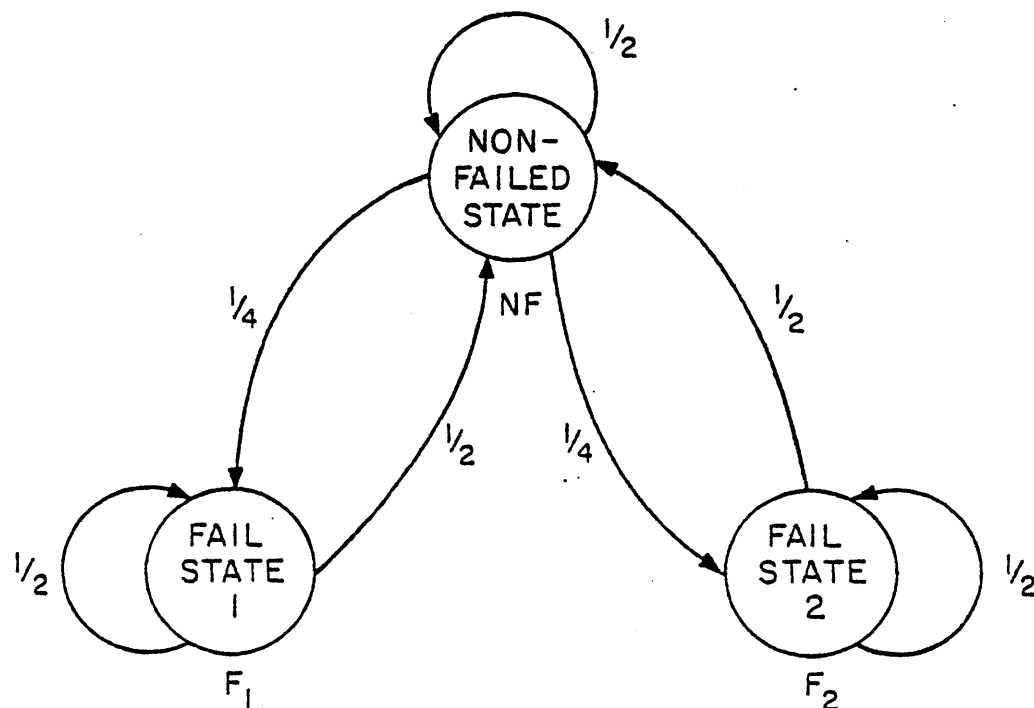


Figure 2.2.2

A Markovian Model

The transition matrix A for this example is given as:

$$\underline{A} = \begin{matrix} & \begin{matrix} NF & F_1 & F_2 \end{matrix} \\ \begin{matrix} NF \\ F_1 \\ F_2 \end{matrix} & \begin{bmatrix} 1/2 & 1/4 & 1/4 \\ 1/2 & 1/2 & 0 \\ 1/2 & 0 & 1/2 \end{bmatrix} \end{matrix}$$

Therefore, the matrix equation has the following form:

$$\begin{bmatrix} P_{NF} & P_{F_1} & P_{F_2} \end{bmatrix} \underline{A} = \begin{bmatrix} P_{NF} & P_{F_1} & P_{F_2} \end{bmatrix}$$

Solving the above equation one finds the matrix P as follows:

$$\underline{P} = [1/2 \ 1/4 \ 1/4];$$

and thus

$$P_{NF} = 1/2.$$

2.3 Class 2 Components: Nonrepairable Components

For Class 2, 3 and 4 components, the total unavailability can be divided into two separate parts: first, the probability that at an initial time ($t = 0$) the component K is down (unavailable); second, the probability that during the operation, component K becomes unavailable. Therefore, one can write

$$\bar{A}(t) = (1 - V_i) \bar{A}_{\bar{v}}(t) + V_i \bar{A}_d(t)$$

where

V_i = probability that K is down at $t = 0$

$\bar{A}_v(t)$ = probability that K is down at $t = t$ given that
K was up at $t = 0$

$\bar{A}_d(t)$ = probability that K is down at $t = t$ given that
k was down at $t = 0$

For class 2, 3 and 4 components, we assume that $V_i = 0$, which is a valid assumption in many practical cases. Therefore, it is assumed that $\bar{A}(t) = \bar{A}_v(t)$.

In the code PL-MODT the unavailability equations are formulated in terms of $\bar{A}_v(t)$ since this reduces the amount of calculations.

The value of $\bar{A}_v(t)$ for any class 2 component for a mission time t may be calculated exactly by the expression

$$\bar{A}_v = 1 - \exp \left[- \int_{t_1}^{t_2} h(t) dt \right] \quad (2.3-1)$$

where

$h(t)$ = hazard rate (instantaneous rate)

t = stated time duration of the mission which begins
at time t_1 and ends at t_2 .

Equation (2.3-1) is an exact relation for the unavailability of the class 2 components. Generally, data are not available to give a good description of the hazard rate throughout a component's lifetime especially during break in and wear out periods. However, it has been demonstrated that for most components, there is a long period of useful life during which the hazard rate is relatively constant. If the hazard

rate is constant then Equation (2.3-1) can be expressed as:

$$\bar{A}_v(t) = 1 - \exp [-\lambda t] \quad (2.3-2)$$

where λ is the constant component failure rate which is a characteristic of the exponential distribution. We will see later in Section 2.4 that, for a special case, class 3 components can be reduced to class 2 and therefore Equation (2.3-2) should be obtainable from class 3 unavailability equations. Equation (2.3-2) has been adopted in PL-MODT as an approximation for calculating the unavailability of non-repairable components. As noted above this is a good approximation in many practical cases.

2.4 Class 3 Components: Repairable Components

For this class of components, it is assumed again that $\bar{A}(t) = \bar{A}_v(t)$ (i.e. that the component is in the unfailed state at $t = 0$). For calculating $\bar{A}_v(t)$ it would be more convenient to use a Markovian approach employing a constant failure rate $\lambda(\text{hr}^{-1})$ and a constant repair rate $\mu(\text{hr}^{-1})$. Figure (2.4.1) presents a Markovian model for this class of components.

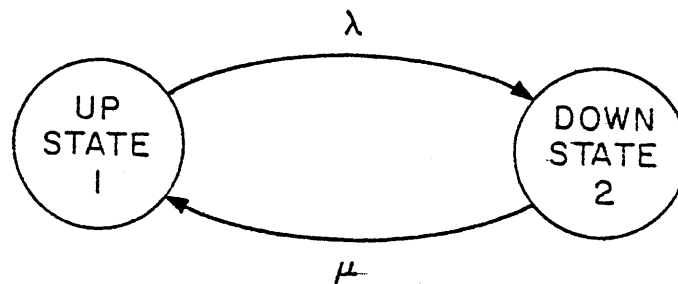


Figure 2.4.1

A Markovian Model for Repairable Components (Revealed Faults)

It is assumed that the transition rates λ and μ have a probability density function $f(t)$ and $g(t)$, respectively. Assuming that state 1 is the UP state, and state 2 is the DOWN state, then

$P_{12}(T)$ = probability that the component K goes from
1 to 2 in the time interval $\Delta t(t \text{ to } t + \Delta t)$

Therefore

$$P_{12} = \lambda \Delta t + \text{higher order terms} \quad (2.4-1)$$

$$\text{Similarly, } P_{21} = \mu \Delta t \quad (2.4-2)$$

Using Equations (2.4-1) and (2.4-2) we can determine the transition matrix $\bar{A}$ as follows:

$$\begin{array}{c} \begin{array}{cc} & \begin{array}{cc} 1 & 2 \end{array} \\ \begin{array}{c} 1 \\ 2 \end{array} & \left(\begin{array}{cc} 1 - \lambda \Delta t & \lambda \Delta t \\ \mu \Delta t & 1 - \mu \Delta t \end{array} \right) \end{array} \quad (2.4-3)$$

By subtracting 1 from the diagonal elements of the matrix $\underline{P}$ one obtains the identity matrix.

$$\begin{array}{c} \begin{array}{cc} & \begin{array}{cc} 1 & 2 \end{array} \\ \begin{array}{c} 1 \\ 2 \end{array} & \left(\begin{array}{cc} - \lambda \Delta t & \lambda \Delta t \\ \mu \Delta t & - \mu \Delta t \end{array} \right) \end{array} \quad (2.4-4)$$

From the identity matrix $\underline{P}$ we get the transition matrix which reads

$$\underline{A} = \begin{pmatrix} -\lambda & \mu \\ \mu & -\lambda \end{pmatrix} \quad (2.4-5)$$

To evaluate $P_2(t) = \bar{A}_V(t)$ the method of Laplace transform is applied.

$$(\underline{S}\underline{I} - \underline{A}) = \begin{pmatrix} S+\lambda & -\lambda \\ -\mu & S+\mu \end{pmatrix} \quad (2.4-6)$$

The inverse of the matrix, Equation (2.4-6), is calculated as follows:

$$\begin{aligned} (\underline{S}\underline{I} - \underline{A})^{-1} &= \begin{pmatrix} \frac{S + \mu}{S(S + \lambda + \mu)} & \frac{\lambda}{S(S + \lambda + \mu)} \\ \frac{\mu}{S(S + \lambda + \mu)} & \frac{S + \lambda}{S(S + \lambda + \mu)} \end{pmatrix} \\ &\equiv \frac{1}{S} \begin{pmatrix} \frac{\mu}{\lambda + \mu} & \frac{\lambda}{\lambda + \mu} \\ \frac{\mu}{\lambda + \mu} & \frac{\lambda}{\lambda + \mu} \end{pmatrix} + \frac{1}{S + \lambda + \mu} \begin{pmatrix} \frac{\mu}{\lambda + \mu} & -\frac{\lambda}{\lambda + \mu} \\ -\frac{\mu}{\lambda + \mu} & \frac{\mu}{\lambda + \mu} \end{pmatrix} \quad (2.4-7) \end{aligned}$$

Taking the inverse of Equation (2.4-7) we can evaluate $\bar{A}_V(t)$

$$P(t) = \begin{pmatrix} \frac{\mu}{\lambda + \mu} & \frac{\lambda}{\lambda + \mu} \\ \frac{\mu}{\lambda + \mu} & \frac{\lambda}{\lambda + \mu} \end{pmatrix} + e^{-(\lambda + \mu)t} \begin{pmatrix} \frac{\lambda}{\lambda + \mu} & -\frac{\lambda}{\lambda + \mu} \\ \frac{-\mu}{\lambda + \mu} & \frac{\mu}{\lambda + \mu} \end{pmatrix} \quad (2.4-8)$$

From Equation (2.4-8) it follows that

$$\bar{A}_v(t) = P_2(t) = \frac{\lambda}{\lambda + \mu} (1 - e^{-(\lambda + \mu)t}) \quad (2.4-9)$$

Equation (2.4-9) approaches an asymptotic value of

$$\bar{A}_v(t) = \frac{\lambda}{\lambda + \mu} \quad (2.4-10)$$

for large t .

Equation (2.4-10) could be reformulated as follows:

$$\bar{A}_v = \frac{\tau_r}{\theta + \tau_r} \quad (2.4-11)$$

where

$$\theta = \frac{1}{\lambda} = \text{mean time between failures (MTBF)}$$

and

$$\tau_r = \frac{1}{\mu} = \text{mean time to repair (MTTR)}$$

Sometimes Equation (2.4-10) is given in the alternate form:

$$\bar{A}_v = \frac{\lambda \tau_r}{\lambda \tau_r + 1} \quad (2.4-12)$$

Equation (2.4-9) can be used to determine unavailability of non-repairable components. It is known that for non-repairable components (class 2) the repair is zero because no repair is conducted which corresponds to $\tau_r = \infty$. Therefore, by setting $\mu = 0$ in Equation (2.4-9), it reduces to the form given by Equation (2.3-2),

i.e., $\bar{A}_v(t) = 1 - e^{-\lambda t}$.

Equation (2.4-9) is used in the code PL-MODT to determine the unavailability of repairable components.

2.5 Class 4 Components: Periodically Tested Components

Suppose that the component k is inspected at the times $t_1, t_2, \dots, t_n$. If k is found to be failed, it is then repaired.

θ_n = time needed to inspect an intact component at the n -th inspection

τ_n = time needed to repair a failed component at the n -th inspection; and

$\theta_n + \tau_n$ = time needed to inspect and return to service a failed component at the n -th inspection

This time will in general be less than the inspection interval so:

$$\theta_n + \tau_n < t_{n+1} - t_n$$

Assuming that the times t_n are known (inspection times), then the unavailability for any time t given $t_n < t < t_{n+1}$ could be calculated.

If

$$x_n = t_n - t_{n-1} - \theta_{n-1} \text{ for } n > 1$$

$$y_n = x_n - \tau_n - 1 \text{ and}$$

$$x_1 = t_1$$

Then the unavailability at different times $t_1, t_2, \dots, t_n$ would be [2]

$$\begin{aligned}
\bar{A}_v(t_1) &= F(x_1) \\
\bar{A}_v(t_2) &= F(x_1)F(y_2) + [1 - F(x_1)]F(x_2) = F(x_2) - [F(x_2) - F(y_2)]F(x_1) \\
\bar{A}_v(t_3) &= \{F(x_2) - [F(x_2) - F(y_2)]F(x_1)\}F(y_3) + \\
&\quad \{1 - F(x_2) + [F(x_2) - F(y_2)]F(x_1)\}F(x_3) \\
&= F(x_3) - [F(x_3) - F(y_3)]F(x_2) + [F(x_3) - F(y_3)][F(x_2) - F(y_2)]F(x_1) \\
&\quad \cdot \\
&\quad \cdot \\
&\quad \cdot \\
&\quad \cdot \\
\bar{A}_v(t_n) &= \sum_{n-j}^n (-1)^{n-j} F(x_j) \prod_{k=j+1}^n [F(x_k) - F(y_k)] \quad (2.5-1)
\end{aligned}$$

Equation (2.5-1) can be considerably simplified if the inspection time and inspection period are fixed values, that is:

$$\text{Inspection interval} = t_2 - t_1 = t_3 - t_2 = \dots = t_n - t_{n-1} = \eta$$

$$\begin{aligned} \text{Inspection period} &= \theta_1 = \theta_2 = \dots = \theta_n = \theta \\ \text{(Duration)} \end{aligned}$$

$$\begin{aligned} \text{Repair period} &= \tau_1 = \tau_2 = \dots = \tau_n = \tau \\ \text{(Duration)} \end{aligned}$$

Equation (2.4-1) becomes as follows:

$$\begin{aligned}
\bar{A}_v(t_n) &= F(\eta - \theta) \sum_{j=1}^n [F(\eta - \theta - t) - F(\eta - \theta)]^{n-j} \\
&\quad + [F(t_1) - F(\eta - \theta)][F(\eta - \theta - t) - F(\eta - \theta)]^{n-1} \quad (2.5-2)
\end{aligned}$$

Equation (2.5-2) can be written as follows [2]:

$$\begin{aligned} \bar{A}_v(t_n) = & F(\eta - \theta) \frac{1 - [F(\eta - \theta) - F(\eta - \theta - \tau)]^n}{1 + F(\eta - \theta) - F(\eta - \theta - \tau)} \\ & + [F(t_1) - F(\eta - \theta)][F(\eta - \theta - \tau) - F(\eta - \theta)]^{n-1} \quad (2.5-3) \end{aligned}$$

Since $F(\eta - \theta) - F(\eta - \theta - \tau)$ in Equation (2.5-3) is usually very small, then Equation (2.5.3) can be approximated by

$$\lim_{n \rightarrow \infty} \bar{A}_v(t_n) = \frac{F(\eta - \theta)}{1 + F(\eta - \theta) - F(\eta - \theta - \tau)} \quad (2.5-4)$$

The expression for the unavailability as a function of time can be written as [2]

$$\bar{A}_v(t) = \bar{A}_v(t_n) \beta(t_n, t) + [1 - \bar{A}_v(t_n)] \alpha(t_n, t) \quad (2.5-5)$$

where

$\alpha(t_n, t)$ = probability that the component k is down at time t given k was up at t_n and,

$\beta(t_n, t)$ = probability that component k is down at time t given k was down at t_n .

Thus,

$$\alpha(t_n, t) = 1(t - t_n) - 1(t - t_n - \theta_n) + F(t - t_n - \theta_n) \quad (2.5-6)$$

and

$$\beta(t_n, t) = 1(t - t_n) - 1(t - t_n - \theta_n - \tau_n) + F(t - t_n - \theta_n - \tau_n) \quad (2.5-7)$$

where

$1(t - t_n)$ is a unit step function given as follows:

$$1(x) = \begin{cases} 1 & \text{for } x \geq 0_+ \\ 0 & \text{for } x \leq 0_- \end{cases}$$

Now if $F(t) = 1 - \exp[-\lambda t]$ where λ is the constant failure rate of the component, after some approximations [2] we obtain from Equation (2.5-4)

$$\bar{A}_v(t) = 1 - e^{-(t - m\eta)\lambda_{\text{eff}}} \left[1 - e^{-\left(\frac{t - m\eta}{\theta}\right)^q} \right] \quad (2.5-8)$$

where

$$\lambda_{\text{eff}} = \frac{\eta - \theta}{\eta} \left(\frac{\eta - \theta}{\eta} + \frac{2\tau}{\eta} \right) + 2 \left[1 - \frac{\Gamma\left(\frac{1}{q}\right)}{q} \right] \frac{\theta}{\eta^2} \quad (2.5-9)$$

$\Gamma\left(\frac{1}{q}\right)$ = gamma function of $\frac{1}{q}$ and,

$$q = \ln[3 - \ln(\theta\lambda)] \quad \text{and,} \quad (2.5-10)$$

$$m = 1, 2, \dots, n$$

Figure (2.5.1) shows the unavailability calculated by Equation (2.5-8) for one inspection interval and the following data for a specific periodically tested component as the solid line assumes the system is unavailable during test process. The dashed line assumes that with a probability of 60 percent the system is not available during the test procedure.

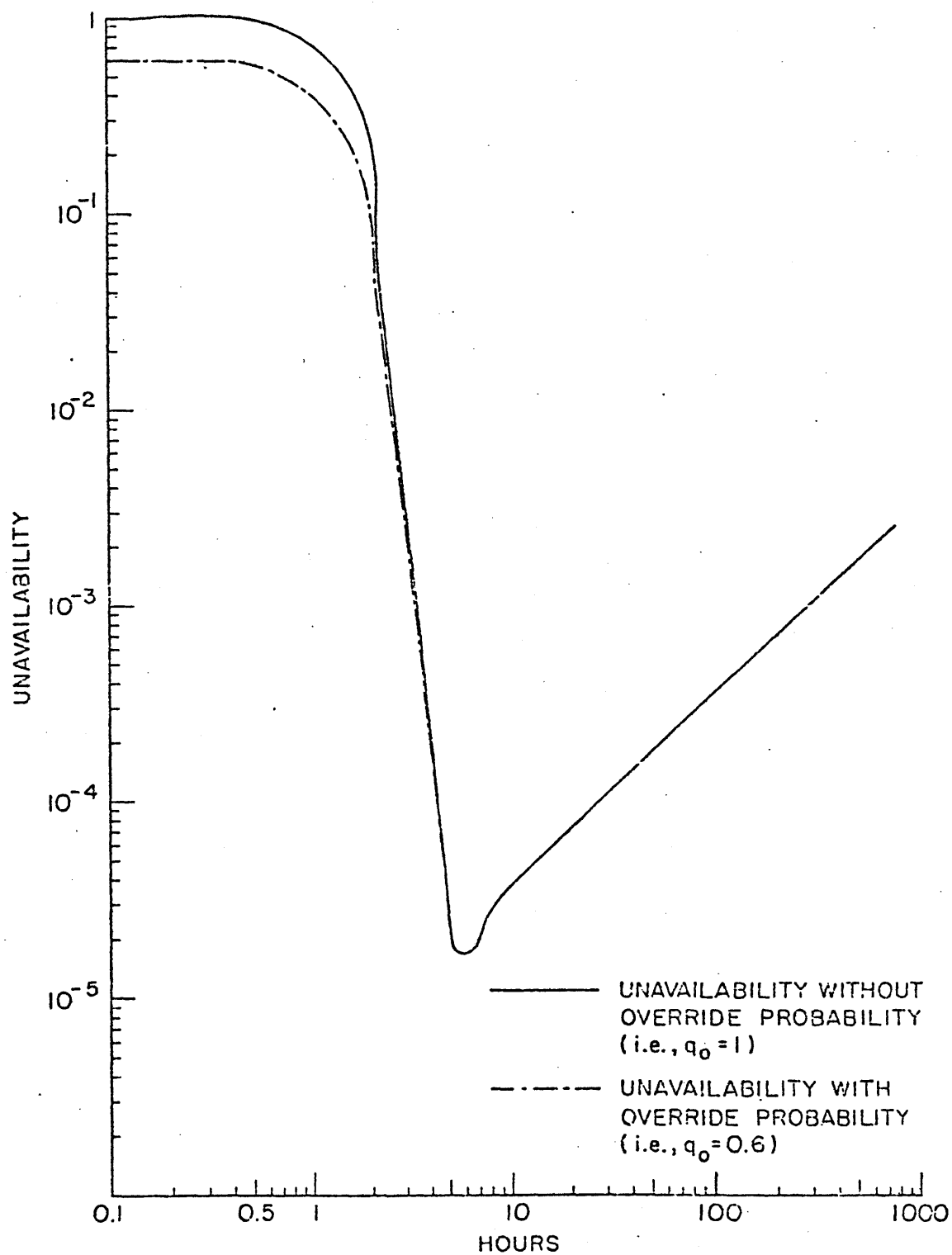


FIGURE 2.5.1

UNAVAILABILITY OF A COMPONENT AS A FUNCTION OF TIME
USING EQ. (2.5.8)

$$\theta = 1.5 \text{ hrs}, \tau = 19 \text{ hrs}, \eta = 720 \text{ hrs}, \lambda = 3 \times 10^{-6} \text{ hr}^{-1}.$$

It should be noted here that Equation (2.5-8) can be changed into a simpler form. To do this consider the term in brackets of Equation (2.5-8), that is

$$\left[1 - e^{-\left(\frac{t - m\eta}{\theta}\right)} \right]^q$$

For typical values of q , η , and θ , this part of the equation approaches unity near the end of the inspection period (i.e. for $t > m\eta + \theta$), so for this condition we can approximate Equation (2.5-8) as follows:

$$\bar{A}_V(t) \approx 1 - e^{-(t - m\eta)\lambda_{\text{eff}}} \quad 2.5-8a$$

If this approximation is used for the operating times $t > m\eta + \theta$, some computing time can be saved. Thus, for only a small fraction of time θ , Equation (2.5-8) could be used and for the rest of the test interval the simplified form (i.e. Equation 2.5.8a) can be applied. (Note that θ is always in the order of a few hours and η typically varies between several days to several months.) The simplified form of Equation (2.5-8) is not presently incorporated into PL-MODT but it is recommended as a way to release calculational time if that becomes a problem.

Equation (2.5-8) provides the unavailability of a periodically tested component with equal test intervals. It is known, however, that sometimes the first test interval is longer than the subsequent ones and, therefore, Equation (2.5-8) is not quite adequate for this interval. Equation (2.5-8) can be modified such that the longer test interval can

be handled. Specifically at $t = 0$ we have $\bar{A}_v(0) = 0$ and $\eta = \eta_1$ for the first test interval, with $\eta_2 = \eta_3 = \dots \eta_n = \eta'$ for subsequent ones. A new form of Equation (2.5-8) is actually utilized in the PL-MODT code for the first test interval by changing $t \rightarrow t + \theta$ and setting $m = 1$, and $\eta = \eta_1$. Therefore, Equation (2.5-8) becomes

$$\bar{A}_{v_1}(t) = 1 - e^{-(t + \theta - \eta_1)\lambda_{\text{eff}}} \cdot \left[1 - e^{-\left(\frac{t - \eta_1}{\theta} + 1\right)q} \right] \quad (2.5-11)$$

Also, it should be noticed that in the PL-MODT the second term in Equation (2.5-9) is approximated as follows

$$2 \left[1 - \frac{\Gamma(1/q)}{q} \right] \theta / \eta^2 \approx 0.2 \times \theta / \eta^2 \quad (2.5-12)$$

This approximation will save substantial amounts of computation time and will not change the computed unavailabilities since θ / η^2 is very small compared to the first term of Equation (2.5-9).

Override probability* P can be accounted for in Equation (2.5-8) by multiplying $\bar{A}_v(t)$ during the inspection period (θ) by the override probability P. Therefore,

$$\bar{A}_{v_o}(t) = \bar{A}_v(t) \cdot P \quad (2.5-13)$$

where P is given by

$$P = q_o + (1 - q_o)(1 - e^{-\lambda_{\text{eff}} \cdot \eta}) \quad (2.5-14)$$

*Override unavailability is the probability that a component cannot function properly during its inspection period if it is demanded. Therefore, with an override unavailability equal to one, the component is totally unavailable during the inspection interval.

and

q_o = override unavailability.

Equation (2.5-14) can be derived by using Equation (2.5-8) to calculate the unavailability at the end of the test interval. The effect of incorporating an override probability is also shown in Figure (2.5-1) for $q_o = 0.6$. The overall effect is a reduction in the average unavailability of the component and thus an increase in the system availability.

2.6 General Time Dependent Relations for the Evaluation of Fault Trees by Using the Modular Concept

Consider a module with a set $\{m_1, m_2, \dots, m_n\}$. The structure function for this module will be

$$\sigma_M(t) = \beta[\sigma_1(t), \sigma_2(t), \dots, \sigma_m(t)] \quad (2.6-1)$$

where

$$\sigma_i(t) = \sigma_{mi}(t) \quad (i = 1, 2, \dots, n) \text{ also,}$$

$$\sigma_i(t) = \begin{cases} 1 & \text{when module } i \text{ has occurred at time } t \\ 0 & \text{otherwise} \end{cases}$$

The expectation value of Equation (2.6-1) is as follows

$$h_\sigma(t) = E\{\sigma_M(t)\} \quad (2.6-2)$$

Similar to the steady state analysis, for a simple AND module in which $M = \{M_1, M_2, \dots, M_n, \Omega\}$ we have

$$h_{\sigma}(t) = E\{\sigma_M(t)\} = h_{\sigma_1}(t) \cdot h_{\sigma_2}(t) \dots h_{\sigma_m}(t) = \prod_{i=1}^n h_{\sigma_i}(t) \quad (2.6-3)$$

Similar expressions can be derived for an OR gate module, namely for a set M where:

$$M = \{M_1, M_2, \dots, M_n, U\}$$

$$h_{\beta}(t) = 1 - [1 - h_{\sigma_1}(t)][1 - h_{\sigma_2}(t)] \dots [1 - h_{\sigma_n}(t)] = \prod_{i=1}^n h_i(t) \quad (2.6-4)$$

For higher order modules, (improper modules), we have the following relationship

$$\sigma_M(t) = \bigcup_{j=1}^{N_k} \prod_{i \in k_j} \sigma_i(t) \quad (2.6-5)$$

where N_k = number of modules, replicated and free components connected to a higher order module or the top event.

Using the minimal cut-set upper bound formula, one obtains

$$h_{\beta}(t) = \bigcup_{j=1}^{N_k} \prod_{i \in k_j} h_{\sigma_i}(t) = \sum_{j=1}^{N_k} \prod_{i \in k_j} h_{\sigma_i}(t) \quad (2.6-7)$$

which is simply the union of modules and components which are attached to a higher order module or the top event.

2.7 General Relations for Time Dependent Simple Modules Consisting of Only Repairable Components (Class 3 Components)

The unavailability of a repairable component as given by Equations (2.4-9) and (2.4-10) can be used to derive an approximate failure rate Λ and a mean dead time τ for simple AND and OR gates.

A simple AND gate or module is a module consisting of only simple component inputs. From Equation (2.4-10) by employing the common assumption that $\lambda_i \tau_i \ll 1$, one obtains

$$\bar{h}_i(t) = \bar{F}_i \approx \lambda_i \tau_i \quad (2.7-1)$$

where

$\bar{h}_i(t)$ = unavailability of the component i input to a module.

Once these basic relationships have been established, the next step is to find the failure and repair rates of the module.

The primary event in an AND module can occur in the time interval t to $t + dt$, with the remaining events having already occurred at time t , or the second primary event can occur in the time interval t to $t + dt$, with the remaining events having already occurred at time t , or Keeping these observations in mind, the following equation is obtained.

$$f(t) = \text{pr}\{F \cap R\} = \sum_{j=1}^n F_j(t) \lambda_i \cdot dt \prod_{\substack{i=1 \\ i \neq j}}^n \bar{F}_i(t) \quad (2.7-2)$$

where n = number of components input to the AND gate

$$\bar{F}(t) = 1 - F(t)$$

To obtain $f(t)$ the occurrence rate of an AND gate, let $\bar{H}(t)$

be designated as:

$$\bar{H}(t) = \prod_{i=1}^n \bar{F}_i(t) \quad (2.7-3)$$

By substituting Equation (2.7-3) into Equation (2.7-2), the following expression results

$$f(t) = \sum_{j=1}^n F_j(t) \lambda_j \times \frac{\bar{H}(t)}{\bar{F}_j(t)} \quad (2.7-4)$$

Since $F_j(t)$ is close to unity and the remaining terms are very small, Equation (2.7-4) can be approximated as

$$f(t) \approx \bar{H}(t) \sum_{j=1}^n \frac{\lambda_j}{\bar{F}_j(t)} \quad (2.7-5)$$

By substituting Equations (2.7-1) and (2.7-3) into Equation (2.7-5) the following result is obtained:

$$f(t) = f = \prod_{i=1}^n \lambda_i \tau_i \sum_{j=1}^n \frac{1}{\tau_j} \quad (2.7-6)$$

By using the general hazard rate formula, it is possible to evaluate Λ for the AND gate as follows

$$\Lambda(t) = \frac{f(t)}{R(t)} \quad (2.7-2)$$

where

$R(t)$ = reliability of the AND gate.

$\Lambda(t)$ is approximately equal to $f(t)$ since $R(t)$ in most cases is close to 1. Therefore, Equation (2.7-7) reduces to the following form

$\Lambda_{\text{AND}}(t) = f(t)$ or, from Equation (2.7-6),

$$\Lambda_{\text{AND}} = \prod_{i=1}^n \lambda_i \tau_i \sum_{i=1}^n \frac{1}{\tau_i} \quad (2.7-8)$$

Similarly, by using the definition of the mean dead time for a simpler gate, it follows that

$$\tau(t) = \frac{\bar{H}(t)}{H(t)\Lambda(t)} \quad (2.7-9)$$

Therefore, the mean dead time for a simple AND gate can be obtained using Equation (2.7-9):

$$\tau_{\text{AND}} = \frac{\prod_{i=1}^n \lambda_i \tau_i}{(1 - \prod_{i=1}^n \lambda_i \tau_i) (\prod_{i=1}^n \lambda_i \tau_i \sum_{j=1}^n \frac{1}{\tau_j})} \quad (2.7-10)$$

Since $\lambda_i \tau_i \ll 1$, it can be assumed that $(1 - \prod_{i=1}^n \lambda_i \tau_i) \approx 1$ and thus Equation (2.7-10) can be written as follows

$$\tau_{\text{AND}} = \frac{1}{\sum_{j=1}^n \frac{1}{\tau_j}} \quad (2.7-11)$$

In a similar way, one can obtain the value of $\Lambda(t)$ and $\tau(t)$ for OR gates. The first primary event in an OR gate can occur in the time interval t to $t + dt$ with the remaining events not having occurred at time t , or the second primary event can occur in the time interval t to $t + dt$ with the remaining events not having occurred at time t or, ...

Similar to Equation (2.7-2), the following equation results:

$$g(t) \cdot dt = \sum_{j=1}^n G_j(t) \lambda_j dt \prod_{\substack{i=1 \\ i \neq j}}^n G_i(t) \quad (2.7-12)$$

Applying the same approximations made for Equation (2.7-2), Equation (2.7-12) can be reduced to the following form

$$g(t) = L(t) \sum_{j=1}^n \lambda_j \quad (2.7-13)$$

where

$$L(t) = \prod_{i=1}^n G_i(t) \approx 1 - \sum_{i=1}^n \bar{G}_i(t)$$

From Equation (2.7-13) it follows that*

$$\Lambda_{OR} = \sum_{i=1}^n \lambda_i \quad (2.7-14)$$

Using a definition similar to that in Equation (2.7-9) for the mean dead time

$$\tau(t) = \frac{\bar{L}(t)}{L(t)\Lambda(t)}$$

By approximating $\prod_{i=1}^n \lambda_i \tau_i$ as $\sum_{i=1}^n \lambda_i \tau_i$, then τ_{OR} follows as

$$\tau_{OR} = \frac{\sum_{i=1}^n \lambda_i \tau_i}{(1 - \sum_{j=1}^n \lambda_j \tau_j) \sum_{i=1}^n \lambda_i} \quad (2.7-15)$$

A special case in which $(1 - \sum_{i=1}^n \lambda_i \tau_i)$ is close to unity would result in very simple form for τ_{OR} , namely,

$$\Lambda_{OR} = \sum_{i=1}^n \lambda_i \quad (2.7-16)$$

and

$$\tau_{OR} = \sum_{i=1}^n \lambda_i \tau_i / \sum_{i=1}^n \lambda_i \quad (2.7-17)$$

The values obtained for Λ and τ for simple AND and OR modules can be further investigated. For example, take Equations (2.7-8) and (2.7-11) for an AND module. If we were to approximate this module behaving as a simple component, with the same approximation stated in

* Note that $\Lambda_{OR} = g(t)/L(t)$

Equation (2.7-1), it follows that the unavailability of the module is given by

$$\bar{A}_{\text{AND}} = A_{\text{AND}} \cdot \tau_{\text{AND}} = \prod_{i=1}^n \lambda_i \tau_i \quad (2.7-18)$$

Equation (2.7-18) can also be obtained by using the asymptotic unavailabilities of individual input components to this AND gate. Similarly, from Equations (2.7-16) and (2.7-17), it follows that:

$$\bar{A}_{\text{OR}} = \Lambda_{\text{OR}} \cdot \tau_{\text{OR}} = \sum_{i=1}^n \lambda_i \tau_i \quad (2.7-19)$$

which is again the exact asymptotic unavailability of this OR module.

Equations (2.7-18) and (2.7-19) show that the approximation of assuming that a simple module behaves as a simple component would have no effect whatsoever on the simple module's asymptotic unavailability, if and only if $\Lambda \cdot \tau \ll 1$.

The above discussion indicates that, within a certain range of time, Equation (2.4-9) could be used to determine the unavailability of the module. By examining many typical simple modules, it has been found that the approximation of assuming that modules would behave as components will not provide an adequate value for the unavailability of the module for times $t < 2\tau$. Figure (2.7-1) shows a comparison between the exact unavailability of a typical simple AND module with two input components; Table (2.7-1) summarizes the parametric characteristics of these input components.

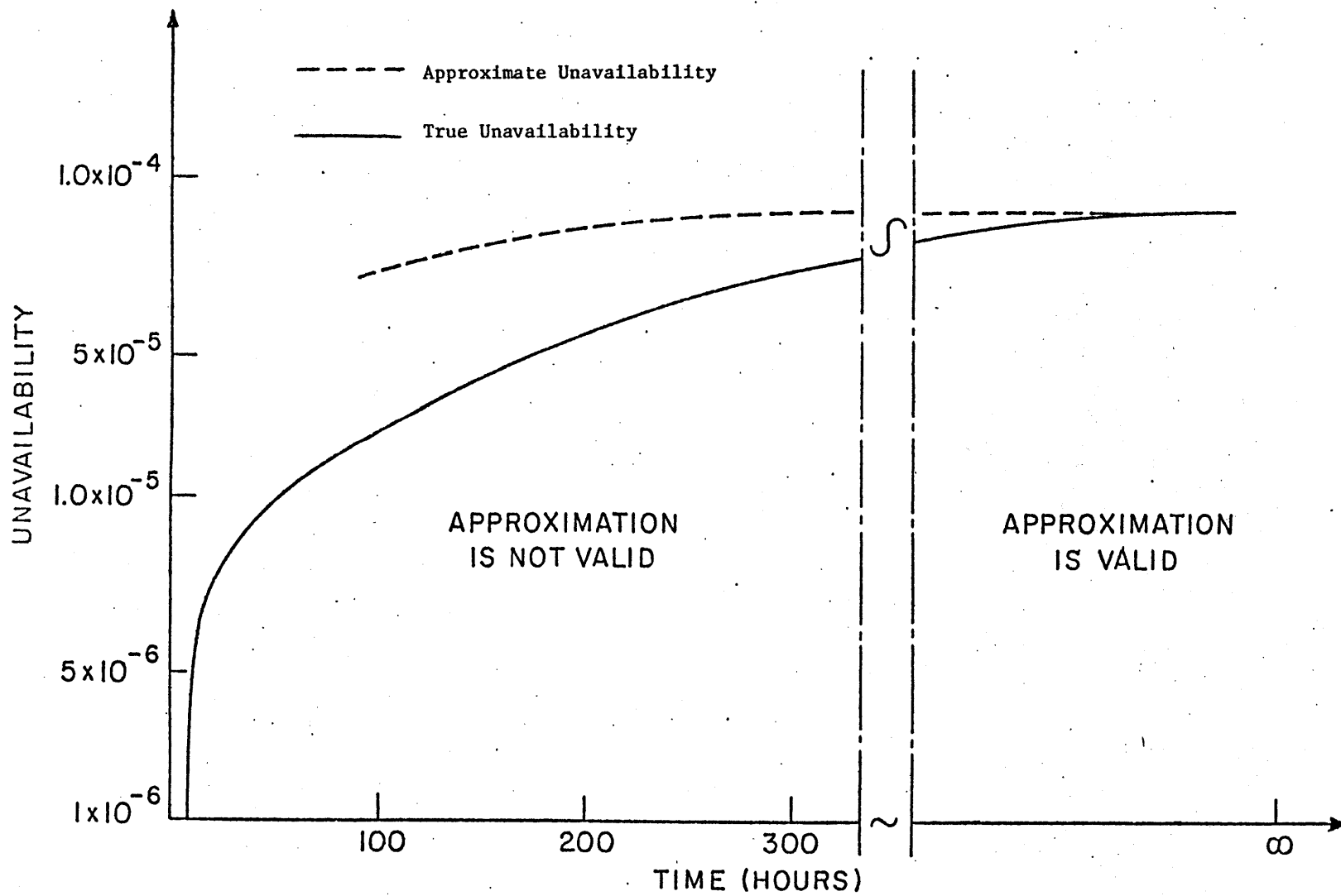


Figure 2.7.1 Unavailability of an AND Module as a Function of Time

TABLE 2.7.1Component Input to a Simple AND Module

<u>Input Component</u>	<u>Failure Rate (hr⁻¹)</u>	<u>Mean Dead Time (hr)</u>
1	3×10^{-4}	100
2	2×10^{-5}	150

Therefore,

$$\Lambda_{\text{AND}} = 9 \times 10^{-5} \times \frac{1}{60} = 1.5 \times 10^{-6} \text{ hr}^{-1}$$

$$\tau_{\text{AND}} = \frac{1}{\frac{1}{100} + \frac{1}{150}} = 60 \text{ hrs}$$

2.8 General Relationships for a Time-Dependent Simple Module

Consisting of Only Non-Repairable Components (Class 2 Components)

For a non-repairable component, the unavailability is given by Equation (2.3-2). For a simple component i , the unavailability would be

$$\bar{F}_i(t) = 1 - e^{-\lambda_i t} \quad (2.8-1)$$

For small values of $\lambda_i t$ we can approximate Equation (2.8.1) as $\bar{F}_i(t) \approx \lambda_i t$. Therefore, by using Equation (2.6-3) one obtains under these circumstances

$$\Lambda_{\text{AND}}(t) = \left(\prod_{i=1}^n \lambda_i \right) t^{n-1} \quad (2.8-2)$$

where n = number of input components to the AND gate. For the case of an OR gate, Λ_{OR} is derived by using Equation (2.6-4)

$$\Lambda_{\text{OR}} = \sum_{i=1}^n \lambda_i \quad (2.8-3)$$

Treating an OR module as a simple component will result in exactly the same unavailabilities and no approximation is involved by using Equation (2.8-3). Treating an AND gate as a simple component by using Equation (2.8-2) is possible only if $\lambda_i t$ is small, namely, $\lambda_i t < 0.1$. In that case, Equation (2.8-2) provides the failure rate of a simple AND module. Therefore, for $t > 0.1/\lambda_i$ the approximation is no longer valid and the general equation, $\bar{A}(t) = \prod_{i=1}^n \bar{F}_i(t)$ must be used. Table 2.8.1 summarizes the formulas discussed in this section.

Table 2.8.1

Failure Characteristics of Simple Modules Having
Repairable or Non-repairable Component Inputs

Type of Module	Repairable Components		Non-repairable Components
	Λ	τ	Λ
AND	$\left(\prod_{i=1}^n \lambda_i \tau_i \right) \sum_{i=1}^n \frac{1}{\tau_i}$	$\frac{1}{\sum_{i=1}^n \frac{1}{\tau_i}}$	$\left(\sum_{i=1}^n \lambda_i \right) t^{n-1}$
OR	$\sum_{i=1}^n \lambda_i$	$\frac{\sum_{i=1}^n \lambda_i \tau_i}{\sum_{i=1}^n \lambda_i}$	$\sum_{i=1}^n \lambda_i$
K-O-N*	$\sum_{k=1}^m \left(\prod_{i=1}^k \lambda_i \tau_i \right) \sum_{i=1}^k \frac{1}{\tau_i}$	$\frac{\sum_{k=1}^m \prod_{i=1}^k \lambda_i \tau_i}{\sum_{k=1}^m \left(\prod_{i=1}^k \lambda_i \tau_i \right) \sum_{i=1}^k \frac{1}{\tau_i}}$	$\sum_{i=1}^m \left(\prod_{i=1}^k \lambda_i \right) t^{k-1}$

*In the K-out-of-N Gate the value of m is given as follows:

$$m = {}^C_N = \frac{N!}{k!(N-k)!}$$

REFERENCES

- [1] Olmos, J. and L. Wolf, "A Modular Approach to Fault Tree and Reliability Analysis", Dept. of Nucl. Eng., MIT, MITNE-209, August, 1977
- [2] Caldorola, L., "Unavailability and Failure Intensity of Components", Nucl. Eng. Design, 44, 147 (1977)
- [3] "Reactor Safety Study", U. S. Atomic Energy Commission, Washington, Appendix II
- [4] Vesely, W. E. and F. F. Goldberg, "FRANTIC: A Computer Code for Time-Dependent Unavailability Analysis", NUREG-0193, October 1977
- [5] Lambert, H. E., "Fault Trees for Decision Making in System Analysis", UTRL-51829, October 1975
- [6] Vesely, W. E., "A Time-Dependent Methodology for Fault Tree Evaluation", Nuclear Engineering 13(1970), 337-360
- [7] Modarres, M. and L. Wolf, "PL-MODT: An Extended Version of the PL-MOD Code for the Modular, Transient Fault Tree Analysis and Evaluation", Department of Nuclear Engineering, MIT, Internal Report.
- [8] Modarres, M. and L. Wolf, "PL-MODT: A Modular Fault Tree Analysis and Transient Evaluation Code", Trans. Am. Nucl. Soc. 28, 510 (1978)

CHAPTER 3

REDUCTION OF LARGE FAULT TREES BASED ON THE
VESELY-FUSSELL IMPORTANCE MEASURES3.1 Introduction

In this chapter the fault tree reduction method which is incorporated into the PL-MODT code is discussed. This reduction method is based on the Vesely-Fussell importance measures which are calculated by the code. A value called the cut-off value is defined and any higher order module, simple module, or component which has an importance less than this value is eliminated. The remaining part of the original fault tree is called the reduced version. Essentially, it must have all of the characteristics of the original fault tree. The reduced version simplifies further analysis of the tree. This chapter describes the method of reduction and the selection of the cut-off range to be used.

3.2 Importance Measures and the Use of PL-MOD to Calculate the V-F Importance Measures

The code PL-MOD is able to calculate the importance measures for large fault trees very effectively and economically. Olmos and Wolf [1] have further developed V-F importance measures to calculate importance of simple and higher order modules in a fault tree.

For example, a higher order module is shown in Figure (3.2.1). Obviously, this higher order module can be shown by a set of modular minimal cutsets, as is shown in Figure (3.2.1)

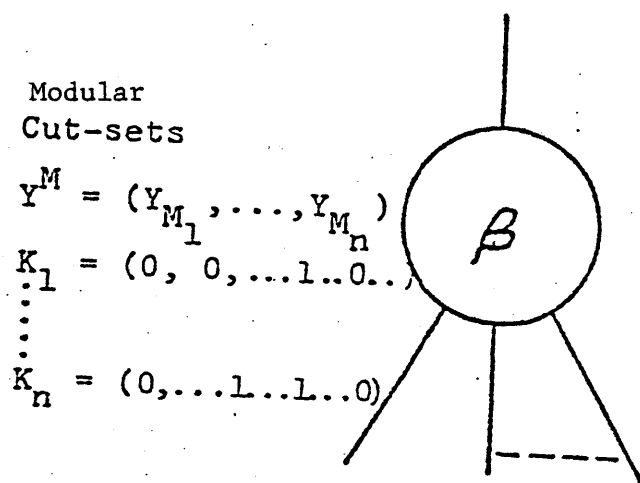


Figure 3.2.1 Higher Order Prime Gate Super-Module

To evaluate the V-F importance of modules such as the one shown in Figure (3.2.1), it follows that

$$\sigma_M^j = \beta(\sigma_1, \sigma_2, \dots, \sigma_n) = \prod_{\ell=1}^{N_k^j} \prod_{\substack{i \in k_\ell \\ j \in k_\ell}} \sigma_i \quad (3.2-1)$$

(i = 1, 2, ..., n)

where σ_M is the structure function for module j, N is modular minimal cutset. The probability that module i will contribute to the failure of its parent module M, given that the parent module has failed is the definition of the V-F importance measure; and is given by

$$I_{\beta, \sigma_j}^{VF} = \frac{\Pr[\beta_k^j(\sigma_1, \sigma_2, \dots, \sigma_n) = 1]}{\Pr[\beta(\sigma_1, \sigma_2, \dots, \sigma_n) = 1]} \quad (3.2-2)$$

now if

$$\Pr[\beta(\sigma_1, \sigma_2, \dots, \sigma_n)] = h_{\sigma_M} \quad (3.2-3)$$

then Equation (3.2-1) implies that β_k^j is given by

$$\beta_k^j = \prod_{\ell=1}^{N_k^j} \prod_{\substack{l \in k_\ell \\ j \in k_\ell}} \sigma_l \quad (3.2-4)$$

Thus, the V.F. importance for module j with respect to the top event will be

$$I_{\theta, \sigma_j}^{V.F.} = I_{\alpha, M}^{V.F.} = \frac{\Pr \left[\prod_{\ell=1}^{N_k^j} \prod_{\substack{l \in k_\ell \\ j \in k_\ell}} \sigma_l = 1 \right]}{h_{\sigma_M}} \quad (3.2-5)$$

In the code PL-MOD the procedure IMPORTANCE is constructed to evaluate the Vesely-Fussell importance ($I^{V.F.}$) for every module and basic component in the fault tree. IMPORTANCE provides these quantities by starting from the top event and taking the top event importance to be equal to 1 (i.e., $I_{TOP}^{V.F.} = 1$), and then proceeds to higher order modules, simple modules, and finally basic components of the fault tree. For a simple AND module, all of the input components and modules must fail in order to have the failure of the AND module. Therefore, given that the AND module is failed, all input components and modules are equally important to the failure of the AND module. Hence it follows from the foregoing that

$$I_{C_i}^{V.F.} = I_M^{V.F.} \quad (i = 1, 2, \dots, n)$$

and

$$I_{M_i}^{V.F.} = I_M^{V.F.} \quad (i = 1, 2, \dots, n)$$

where $I_{C_i}^{V.F.}$ and $I_{M_i}^{V.F.}$ are importances of components and lower order modules input to hand module M with a V-F importance of $I_M^{V.F.}$.

For a simple OR module we have

$$I_{C_i}^{V.F.} = I_M^{V.F.} \cdot \frac{P_i}{P_M} \quad (i = 1, 2, \dots, n)$$

$$I_{M_i}^{V.F.} = I_M^{V.F.} \cdot \frac{P_{M_i}}{P_M} \quad (i = 1, 2, \dots, n)$$

where P_i , P_{M_i} , and P_M are probability of occurrence of component i , moduls M_i and higher order module M , respectively. For the case of higher order prime gate (super-module) the following equations are used in the IMPORTANCE procedure.

$$I_{r_i}^{V.F.} = I_M^{V.F.} \frac{\sum_{j, r_i \in k_j} P(k_j)}{P(M)} \quad (3.2-6)$$

$$I_{M_i}^{V.F.} = I_M^{V.F.} \frac{\sum_{j, M_i \in k_j} P(k_j)}{P(M)} \quad (3.2-7)$$

The numerator of Equations (3.2-6) and (3.2-7) is calculated in the IMPORTANCE procedure, whereas all other probabilities are already calculated in the EXPECT procedure.

The pressure tank system [1] will be analyzed here to show the procedure for V.F. importance calculations. Figure (3.2.2) shows the pressure tank system and Figure (3.3.2) shows its fault tree. This fault tree consists of 13 free components and one replicated component with no replicated module. Failure data for the fault tree in Figure (3.2.3) are given in Table (3.2.1). The different modules which exist in the pressure tank system fault tree are presented in Figure (3.2.3). To calculate the importance measures, the following calculations are performed in three steps. It should be recalled that all probabilities are calculated prior to the use of the IMPORTANCE procedure.

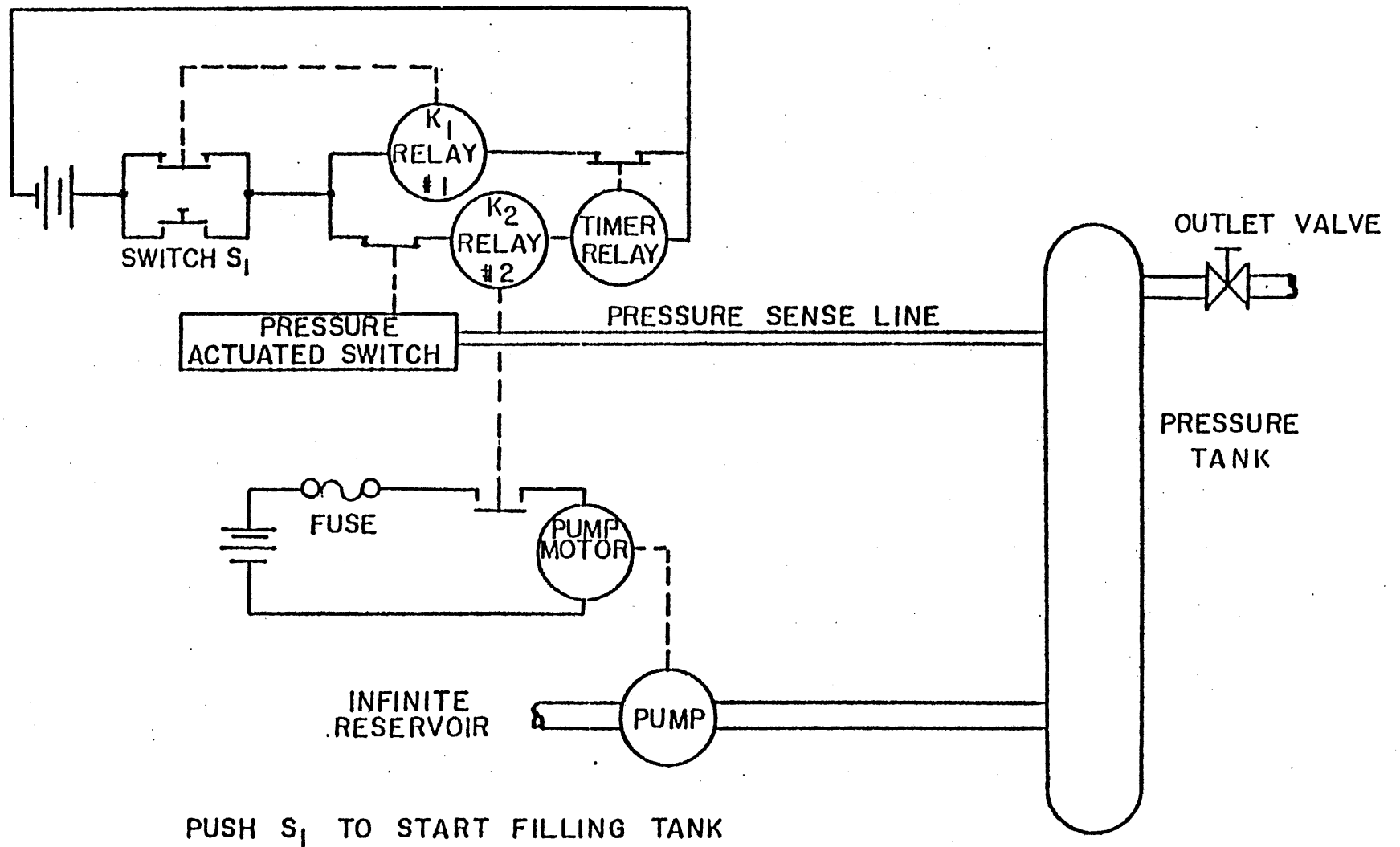


Figure 3.2.2 Pressure Tank Example

Table 3.2.1

PRESSURE TANK RUPTURE FAULT TREE FAILURE PROBABILITY DATA

<u>Basic Event i</u>	<u>Event Description</u>	<u>Failure Rate</u> (per Loading Cycle)
1	Pressure Tank Failure	10 ⁻⁸
2	Secondary failure of Pressure Tank due to Improper Selection	10 ⁻⁵
3	Secondary failure of Pressure Tank due to Out-of-tolerance Conditions	10 ⁻⁵
4	K2 relay contacts fail to open	10 ⁻⁵
5	S1 switch secondary failure	10 ⁻⁵
6	S1 switch contacts fail to open	10 ⁻⁵
7	External reset actuation force remains on switch S1	10 ⁻⁵
8	K1 relay contacts fail to open	10 ⁻⁵
9	Timer does not "time-off" due to improper setting	10 ⁻⁵
10	Timer relay contacts fail to open	10 ⁻⁵
11	Pressure switch not actuated by sensor 1	10 ⁻⁵
12	Pressure switch not actuated by sensor 2	10 ⁻⁵
13	Pressure switch not actuated by sensor 3	10 ⁻⁵
<u>Replicated Event i</u>	<u>Event Description</u>	<u>Failure Rate</u> (Per Loading Cycle)
(30001)	Common Cause failure among relays K ₁ , K ₂ and timer T	10 ⁻⁵

MODULES IN THIS TREE

GATE CORRESPONDING TO A MODULE	TYPE OF MODULE	COMPONENTS AND MODLES ATTACHED TO THE MODULE
9	SYMMETRIC	11, 12, AND 13
4	NESTED	Mg
1	OR SIMPLE MODULE	1, 2, 3, 4,
5	NESTED	5, 6, 7, 8, 9 AND 10

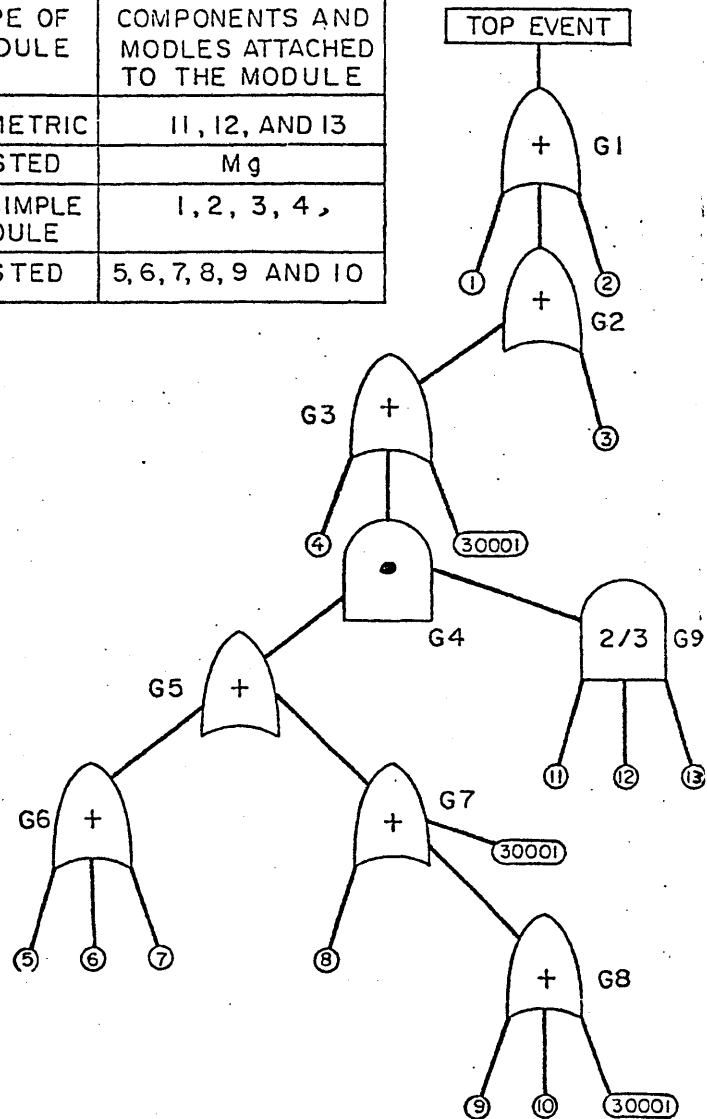


Figure 3.2.3: Pressure Tank Rupture Fault Tree and Associated Modules
 (Failure Rate for Component 1, $\lambda_1 = 1 \times 10^{-8} \text{ hr}^{-1}$;
 all others $\lambda = 1 \times 10^{-5} \text{ hr}^{-1}$)

STEP 1

$$I_{\text{TOP}}^{\text{V.F.}} = 1$$

$$I_r^{\text{V.F.}} = \frac{P_r}{P(\text{TOP})} = 2.49937 \times 10^{-1}$$

$$I_{M_1}^{\text{V.F.}} = \frac{P_{M_1}}{P(\text{TOP})} = 7.500625 \times 10^{-1}$$

$$I_{M_4}^{\text{V.F.}} = I_{M_5}^{\text{V.F.}} = \frac{P_{M_4} P_{M_5}}{P(\text{TOP})} = 4.49887 \times 10^{-10}$$

$$I_1^{\text{V.F.}} = I_{M_1}^{\text{V.F.}} \frac{P_1}{P_{M_1}} = 2.49937 \times 10^{-4}$$

$$I_2^{\text{V.F.}} = I_3^{\text{V.F.}} = I_4^{\text{V.F.}} = I_{M_1}^{\text{V.F.}} \frac{10^{-5}}{P_{M_1}} = 2.49937 \times 10^{-1}$$

STEP 2

$$I_{M_9}^{\text{V.F.}} = I_{M_4}^{\text{V.F.}} = 4.49887 \times 10^{-10}$$

$$\begin{aligned} I_5^{\text{V.F.}} &= I_6^{\text{V.F.}} = I_7^{\text{V.F.}} = I_8^{\text{V.F.}} = I_9^{\text{V.F.}} = I_{10}^{\text{V.F.}} = I_{M_5}^{\text{V.F.}} \cdot \frac{10^{-5}}{P_{M_5}} \\ &= 7.49812 \times 10^{-11} \end{aligned}$$

STEP 3

$$I_{11}^{\text{V.F.}} = I_{12}^{\text{V.F.}} = I_{13}^{\text{V.F.}} = I_{M_9}^{\text{V.F.}} \times \frac{2(10^{-5})^2}{P_{M_9}} = 2.299924 \times 10^{-10}$$

Therefore, if we were to reduce this fault tree based on the above calculations for a cut-off value of 10^{-4} , we should perform the following procedure.

STEP 1

$$I_r^{V.F.} = 2.49937 \times 10^{-1} \text{ KEPT}$$

(since larger than 10^{-4})

$$I_{M_1}^{V.F.} = 7.500625 \times 10^{-1} \text{ KEPT}$$

$$I_{M_4}^{V.F.} = I_{M_5}^{V.F.} = 4.49887 \times 10^{-10} \text{ CANCELLED}$$

Therefore, module M_9 and components 5, 6, 7, 8, 9 and 10 will be automatically cancelled

$$I_1^{V.F.} = 2.49937 \times 10^{-4} \text{ KEPT}$$

$$I_2^{V.F.} = I_3^{V.F.} = I_4^{V.F.} = 2.49937 \times 10^{-1} \text{ KEPT}$$

From this discussion, the reduced version of the fault tree in Figure (3.2.3) is given in Figure (3.2.4). Naturally, the same procedure is adopted to reduce large fault trees by the use of the IMPORTANCE procedure in the PL-MOD code. The method applied is presented in the next section.

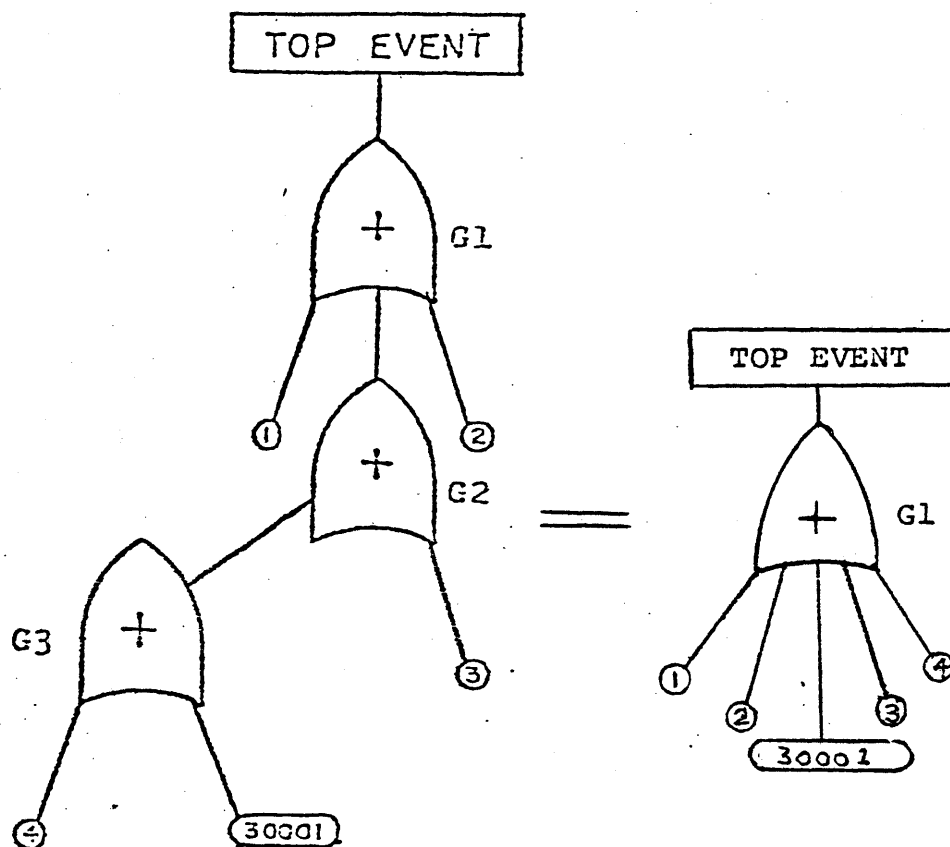


Figure 3.2.4

Reduced Version of the Fault Tree in Figure 3.2.3

3.3 Use of the Code PL-MOD for Reducing Large Fault Trees

As discussed in Section 3.2, the procedure IMPORTANCE in PL-MOD provides the V-F importance measures for all of the fault tree components as well as simple and higher order modules. This procedure is modified to a new form such that it accounts for the cut-off value to exclude those components and modules that have V-F importances less than this prescribed value. For any further evaluation, the new reduced form of the fault tree can be used.

Obviously, the cut-off value plays an important role in the reduction strategy. That is, different values of the cut-off value result in different reduced fault trees. Therefore, a great deal of attention must be paid to the selection of the cut-off value in order to achieve a desired and accurate form of the reduced fault tree.

The new form of the IMPORTANCE procedure is linked with the code PL-MODT, so that for future developments of the code, the reduced version of fault trees can be used by the code to evaluate time-dependent behavior of the fault tree.

In the modified form of the IMPORTANCE procedure, the top event importance will be set equal to 1. By starting from the top event it proceeds to the bottom. First, V-F importances of the higher modules and replicated events that are connected to the top event are calculated. Next, the code automatically removes all modules and replicated events whose V-F importances are less than the prescribed cut-off value. Each higher order module which is removed contains some other modules and free components that are attached to them. Therefore, there is no need to compute the V-F importance of any of these attached members of the

removed higher order modules, because their importances are always equal to or less than that of the parent module which is lower than the cut-off value. If, however, the higher order module is not removed because its importance is higher than the cut-off value, the code will proceed to calculate importances of simple modules and components attached to it, and to automatically remove components and simple modules with an importance below the criterion.

Thus, starting from the top event of an unreduced fault tree, all of the branches of the tree will be eliminated if their importance is below the cut-off value. The pruned tree is the reduced version of the original tree.

3.4 Reductions of LPRS and HPIS Fault Trees

In order to demonstrate the reduction process, the reduced fault tree of the Low Pressure Recirculation System presented in Appendix II of WASH-1400 is further reduced by using different values for the cut-off limit. The maximum change in the top event occurrence probability for a range of cut-off values from 10^{-2} to 10^{-5} has been calculated. The maximum change is obtained when the cut-off value is 10^{-2} . Therefore, the following inequality can be written for this range of the cut-off values.

$$P(\text{TOP})_{\text{reduced}} \leq (1 + 0.01) P(\text{TOP})_{\text{unreduced}}$$

For example, for the LPRS with 6 replicated and 61 non-replicated components, the results for the various reduced versions which follow from the application of the different cut-off values are

summarized in Table (3.4.1).

Table 3.4.1

Percentage of Increase in the Top Event Occurrence of
Reduced Trees of LPRS for Different Cut-Off Values

<u>Cut-off Value</u>	<u>Number of Free Components Remaining</u>	<u>Number of Replicated Components Remaining</u>	<u>Percentage Change in the Top Event</u>
10^{-5}	43	5	No change
10^{-4}	37	5	No change
10^{-3}	20	4	No change
10^{-2}	12	2	1.1%

It becomes obvious from Table (3.4.1) that for cut-off values in the range of 10^{-5} to 10^{-3} no change in the top event occurs and, therefore, one can safely use the upper bound (i.e., 10^{-3}) for the cut-off value without affecting the accuracy, thereby saving computation time. Even the use of 10^{-2} for the cut-off provides a still reasonable fault tree for the LPRS which results in a change of only 1.1% compared to the originally published version. Investigation of the reduced fault trees shows that they have essentially the same low order cut-sets as the original fault tree.

It is very important to understand that a component with low probability or unavailability will not necessarily result in a low V-F importance. For example, for the pressure tank example, Component 1 is found to have

$$I_r^{V.F.} = 2.49937 \times 10^{-4} \text{ and}$$

$$I_{5-10}^{V.F.} = 7.49812 \times 10^{-11}$$

However, $P_1 = 10^{-8}$ and $P_{5-10} = 10^{-5}$, and, therefore, even though the replicated event has a probability of occurrence which is by 3 orders of magnitude less than that of components 5 through 10, its importance is 7 orders of magnitude larger. This means that any reduction process which is solely based on the orders of event probabilities does not necessarily result in an accurate reduced fault tree. To have the same order of magnitude for the importance of the replicated event one needs to reduce the probability of the replicated component down to $P_r = 10^{-15}$ which results in

$$I_r^{V.F.} = 2.49937 \times 10^{-11}$$

From the above discussion it follows that the reduction schemes in the code PL-MODT provide an excellent objective means of reducing a fault tree.

The same kind of study is performed for the HPIS for a cut-off value of 10^{-3} . The HPIS fault tree consists of 142 free and 13 replicated components. The reduction process resulted in a tree with 53 free and 9 replicated components. The top event remained unchanged and the low order cut-sets are almost the same.

Since there are no calculations or iterations involved in the reduction procedure, the computer cost increase is negligibly small

compared to the modularization process. For example, for the LPRS fault tree, the CPU time for modulation and evaluation by the PL-MODT amounts to 0.46 seconds where it is about 0.47 seconds if a reduction with cut-off value equal to 10^{-2} is requested in addition to the above calculations. Although the saving does not become apparent for this steady-state example, it should be pointed out that a substantial saving will result if one uses the reduced version for time-dependent fault tree for subsequent analysis. This is particularly true for Monte Carlo simulation..

It is very important to realize the fact that the V-F importance measures are calculated by assuming that the modules and components of fault trees are statistically independent. However, if a fault tree is used for the common cause failure analysis, the reduced version obtained by using the V-F importance measure may no longer be adequate for this purpose, because, the components of the fault tree may not be statistically independent. Methods can be employed to reduce only those modules and components that are not dependent due to common causes in the system. These methods are not currently incorporated into the PL-MODT code. But, due to the importance of the common cause failure analysis, it would be highly valuable to acquire this capability.

REFERENCES

- 1 Olmos, J. and L. Wolf, "A Modular Approach to Fault Tree and Reliability Analysis", Department of Nuclear Engineering, MIT, MITNE-209, August, 1977
- 2 Lambert, H. E., "Fault Tree for Decision Making in System Analysis", UTRL-51829, October, 1975
- 3 Apostolakis, G., "Mathematical Methods of Probabilistic Safety Analysis", UCLA-ENG-7464, September, 1974

CHAPTER 4

MONTE-CARLO SIMULATION OF TOP EVENT PROBABILITY4.1 Introduction

The quantification of the probability of the top event in a fault tree is straight-forward when point values are used for the component failure probabilities. However, such a point value calculation gives no insight into the possible uncertainties of the top event probability. A common way to overcome this difficulty is to use a Monte-Carlo procedure to develop a probability distribution for the top event. In this procedure each component failure probability is represented by an appropriate distribution. In calculating the top event probability one samples, using a Monte-Carlo Procedure, each of these distributions for a particular value and then calculates the top event probability. The calculation is often repeated several thousand times to generate a distribution in probability for the top event. From this distribution one can obtain a median value and upper and lower bounds at some prescribed confidence level. The procedure is often called a Monte-Carlo simulation.

The codes SAMPLE [1], LIMITS [2] , and FRANDOM [5], are three examples of codes which have recently been used for Monte-Carlo simulation. The SAMPLE code gained widespread attention because it was used in the Reactor Safety Study (WASH-1400) [1] to calculate the reliability bounds of reactor safety systems. All the aforementioned Monte-Carlo codes require a system function as input to identify the logical dependence of components in the systems that are being analyzed. For very large

fault trees, the construction of this equation is a non-trivial process, and there is a high probability of introducing errors. Therefore, for large fault trees there is an advantage in using the fault tree itself as the input rather than the system equation derived from the cutsets.

This chapter describes the modifications made, to the PL-MOD Code, to permit such Monte-Carlo simulations. These modifications permit the simulation to be carried out for fault trees with components that are of any of the following types: constant failure probability, non repairable time-dependent, monitored, or periodically tested. In its present form it is restricted to log-normal input distribution for the failure rates, time to repair, average downtime, and test duration. The PL-MOD Code also had to be modified to permit any type of replicated event in the fault tree. This new modified code is called PL-MODMC.

The PL-MODMC uses the same sorting procedure employed by the LIMITS code developed by D. L. Shell [8]. PL-MODMC uses an efficient PL/I random number generator that is developed in this study to generate normal random numbers by using the Central Limits Theorem Approximation. The output of the PL-MODMC code consists of the mean, standard deviation, and 90% error factors for the top event probability as well as the point value of the average unavailability for different confidence levels. The following sections discuss the mathematical formulation and methodology of the PL-MODMC Code.

4.2 The Log-Normal Distribution

In this section the justification for use of log-normal distribution in this work and its mathematical concepts are discussed.

4.2.1 Justification for Use of Log-Normal Distribution

The log-normal distribution has a number of properties that make it particularly valuable for many of the problems encountered in reliability analysis. For this reason it was adopted in the Reactor Safety Study (WASH-1400)[1]. In this study it was found that the failure rate data for the various fault tree components was rarely precise enough to permit the parent data distribution to be inferred. In fact an analysis of this data revealed that any of a number of distributions were possible parent distributions. Since one of the distributions that was consistent with the observed data was a log-normal, and since it has many mathematical properties that are desirable for the calculations it was chosen. In the Reactor Safety Study it was found that the final answer for system unavailability was quite insensitive to the choice of input distribution, further justifying the use of log-normal distributions.

4.2.2 Characteristics and Some of the Mathematical Concepts of Log-Normal Distribution

A random variable t has a log-normal distribution if its logarithm follows a normal distribution. The distribution is skewed to the right. A log-normal distribution for which 90% of the data lies between t/f and $t.f$ (f is referred to as the "90% error factor" or sometimes only "error factor") can be transferred into a normal distribution centered

at $\log t$ with 90% of the data lying between $\log t - \log f$ and $\log t + \log f$. The variation of most of the failure parameters can be conveniently described by giving median m , and 90% error factor f . For example, a failure rate estimated at 10^{-6} could vary from 10^{-7} to 10^{-5} which is $(10^{-6} \times 10)$ to $(10^{-6}/10)$.

The log-normal distribution has a probability density function (Pdf) of the following form:

$$f(t) = \frac{1}{\sigma t \sqrt{2\pi}} \exp - \left[\frac{(\text{Lnt} - \mu)^2}{2\sigma^2} \right] ; \quad 0 < t < \infty \quad (4.2.1)$$

and a cumulative distribution of the form:

$$F(x) = \frac{1}{\sigma \sqrt{2\pi}} \int_0^{\text{Lnx}} \exp - \left[\frac{(y - \mu)^2}{2\sigma^2} \right] \cdot dy \quad (4.4.2)$$

For describing a log-normal distribution one can only specify median m , and 90% error factor f , i.e.:

$$\Pr(t > m.f) = 0.05; \text{ or}$$

$$\Pr(t < m/f) = 0.05$$

From variables m and f , the parameters μ and σ in equations (4.2.1) and (4.2.2) can be obtained as follows:

$$\mu = \text{Ln } m \quad (4.2.3)$$

$$\sigma = \text{Ln } f / 1.645 \quad (4.2.4)$$

The mean $\bar{t}$ and variance v of a log-normal distribution can also be computed from the variables μ and σ as follows:

$$\bar{t} = e^{\mu + \sigma^2/2} \quad (4.2.5)$$

$$v = e^{2\mu + \sigma^2} \cdot (e^{\sigma^2} - 1) \quad (4.2.6)$$

With these relationships the log-normal distribution is completely described. Mathematical operations with the log-normal distributions are easily performed as will be shown in the next sections.

4.3 Mathematical Concepts used in PL-MODMC for the Monte-Carlo Simulation

The PL-MODMC Code provides the following capabilities:

- (1) The ability to compute a point estimate for the average unavailability value of the top event and/or
- (2) The ability to propagate error or uncertainty bounds on several failure parameters including failure rate, average downtime, average repair time and average test duration.

Table (4.3.1) shows the set of four component types and corresponding unavailability equations that are employed in PL-MODMC. These equations are used in the FRANTIC Code [4]. Equations for Class 2, 3 and 4 components are all approximate equations to calculate average component unavailability over a specific period of time. In all equations, it is assumed that the failure rates are small (i.e., $\lambda \ll 10^{-2}$). Each

TABLE 4.3.1

AVERAGE UNAVAILABILITY FUNCTIONS USED IN THE PL-MODMC

Class	Component Type	Unavailability Function (Average)
1	Cyclic, Constant Unavailability	Q
2	Non-repairable	$\frac{1}{2} \lambda T_M$
3	Repairable, Monitored	λT_D
4	Repairable, Periodically Tested	$\frac{1}{2} \lambda T + \lambda T_R + \frac{\tau}{T}$

where

Q = unavailability per demand

T_M = mission length (hr)

T_D = average down time (hr)

T = test interval (hr)

T_R = average repair time (hr)

τ = average test time (duration) (hr)

λ = failure rate (hr^{-1})

parameters for which uncertainty bounds can be propagated are

Q , λ , T_D and T_R

fault tree basic event is modeled as one of these four component types. It is assumed, as a result of the discussion in the foregoing section that the log-normal distribution holds for all failure data and average unavailabilities.

In Classes one and two, uncertainty bounds can be assigned to Q and λ . The value of T_M in a Class 2 component type is the period of time that the component is operational and is independent of the physical properties of the component; thus, no uncertainty is involved in its value. In Class 3 and Class 4 component types, where addition and multiplication of lognormal distributions occur, the error factors can be obtained as follows:

(a) For Class 3 component types both λ and T_D may be associated with uncertainties and therefore the average unavailability distribution is the multiplication of the two log-normal distributions. Because of the multiplicative properties of log-normal distributions, the average unavailability for a Class 3 component is also a log-normal distribution with the following parameters:

$$\bar{Q} = m_Q = \text{median} = m_\lambda \times m_{T_D} \quad (4.3.1)$$

where m_λ and m_{T_D} are median values of λ and T_D .

$$(\ln \bar{F})^2 = (\ln F_\lambda)^2 + (\ln F_{T_D})^2 \quad (4.3.2)$$

where $F_\lambda = 90\%$ error factor for λ
 $F_{T_D} = 90\%$ error factor for T_D
 $\bar{F} = 90\%$ error factor for Q

(b) For a Class 4 component type the values of λ , T_R and T may have uncertainties associated with them, and therefore each term of the average unavailability equation is a log-normal distribution. The total average unavailability is thus the summation of these three log-normal distributions. The log-normal distribution parameters of each term can be defined as follows :

$$\bar{Q} = \text{median} = m_q = 1/2 m_\lambda \times T + m_\lambda \times m_{T_R} + \frac{m_\tau}{T} \quad (4.3.3)$$

where

m_λ = median value of λ ;

m_{T_R} = median value of T_R ; and

m_T = median value of t

$$F_1 = F_\lambda \quad (4.3.4)$$

$$(\ln F_2)^2 = (\ln F_\lambda)^2 + (\ln F_{T_R})^2 \quad (4.3.5)$$

$$F_3 = F_T \quad (4.3.6)$$

where F_1 , F_2 and F_3 are 90% error factors for the first, second and third terms of the Class 4 average unavailability equation of a component respectively. For the exact evaluation of the distribution associated with Q , samples must be taken from each of these distributions, and summation of these samples provides a sample for the average unavailability of the Class 4 Component type. However, approximations may be performed

so as to represent the average unavailability distribution of Class 4 component types by a single log-normal distribution. This approximation would significantly reduce the Monte-Carlo calculations in a fault tree with several periodically tested components. The validity and accuracy of this approximation is discussed in Section 4.4.

One of the important factors in a Monte-Carlo simulation is its accuracy. In applying the Monte-Carlo method to obtain the top event unavailability distribution, one must have confidence in the numerical values obtained. This usually involves an estimate of the statistical errors in these values. The law of large numbers in the probability theory states that the accuracy of an estimate of a quantity tends to improve as one averages larger and larger samples of observations of the value of the quantity. To apply this to our Monte-Carlo calculations, suppose that $t_1, t_2, \dots, t_n$ are sample values of the random variable t . If the sample mean

$$\bar{t} = \frac{1}{n} \sum_{i=1}^n t_i \quad (4.3.7)$$

is formed, the law of large numbers states that the sample mean, with a probability that approaches 1 as n increases to infinity, approximates the population mean (or true mean).

The law of large numbers can be demonstrated quite simply. To do this, however, we need a very general result in probability theory known as Chebyshev's inequality. Given a random variable, t belongs to a distribution with mean μ and standard deviations σ ; it is easy to deduce a quantitative value about the closeness of t to the mean μ in terms of the standard deviation σ and for all values of parameter h , i.e.,

$$\Pr \{ |t - \mu| \leq h\sigma \} \geq 1 - \frac{1}{h^2}$$

or

$$\Pr \{ |t - \mu| > h\sigma \} \leq \frac{1}{h^2} \quad (4.3.8)$$

For example, for a binomial distribution in which a random variable t takes the value 1 with probability P , and value 0 with probability $1 - P = q$, we find the Chebyshev's inequality. Suppose n samples of variable t are taken from the binomial distribution. Then

$$\bar{t} = \frac{1}{n} \sum_{i=1}^n t_i \quad (4.3.9)$$

represents the frequency of successes (i.e., when t takes the value 1) in n trials. Therefore,

$$\lim_{n \rightarrow \infty} \Pr \{ |\bar{t} - P| \leq \varepsilon \} = 1 \quad (4.3.10)$$

or, alternatively,

$$\lim_{n \rightarrow \infty} \Pr \{ |\bar{t} - P| > \varepsilon \} = 0 \quad (4.3.11)$$

Since the random variable t has mean P and variance $P(1 - P)$, it follows that $\bar{t}$ has mean P and variance $[P(1 - P)]/n$. Applying Chebyshev's inequality with $\varepsilon = h\sigma$, it follows that

$$\Pr \{ |\bar{t} - P| \leq \varepsilon \} = 1 - \frac{P(1-P)}{n \varepsilon^2} \quad (4.3.12)$$

and

$$\lim_{n \rightarrow \infty} \Pr \{ |\bar{t} - P| \leq \varepsilon \} = 1 \quad (4.3.13)$$

regardless of the value of P .

Similarly, if t_i is a random sample of random variable t with finite mean m and standard deviation σ , then the arithmetic mean of the n independent variable $t_1, \dots, t_n$ is given by equation (4.3.9). $\bar{t}$ has mean μ and variance σ^2/n . Again it follows from Chebyshev's inequality that:

$$\Pr \{ |\bar{t} - \mu| \leq \varepsilon \} \leq 1 - \frac{\sigma^2}{n \varepsilon^2}$$

or

$$\lim_{n \rightarrow \infty} \Pr \{ |\bar{t} - \mu| \leq \varepsilon \} \leq 1 - \frac{\sigma^2}{n \varepsilon^2} \quad (4.3.14)$$

Thus the (weak) law of large numbers has been demonstrated for the case of identically distributed random variables with mean and variance finite.

It is demonstrated [5,6] that equation (4.3.12) can be shown as follows:

$$\Pr \{ |\bar{t} - P| \leq \varepsilon \} = \sqrt{\frac{2}{\pi}} \int_0^t e^{-U^2/2} \cdot dU + R \quad (4.3.15)$$

which can also be given in the following form:

$$\Pr \{ |\bar{t} - P| \leq \varepsilon \} = \text{erf} (t/\sqrt{2}) + R \quad (4.3.16)$$

where

$$t = \varepsilon \sqrt{\frac{n}{pq}}$$

R = error associated with the probability measure.

erf = error function

The value of R is given by the following equation

$$|R| \leq \frac{e^{-t^2/2}}{\sqrt{2\pi npq}} + \frac{0.2 + 0.25|P - q|}{n P q} + e^{-(3/2)\sqrt{n P q}} \quad (4.3.17)$$

One may adopt [2] equations (4.3.16) and (4.3.17) to calculate the accuracy of the unavailability calculations provided by Monte- Carlo simulation of variables that are log-normally distributed. Using the same analogy, equations (4.3.16) and (4.3.17) can be used in the PL-MODMC code. However, the value $\bar{t}$ may be changed to $X(p)$, where $X(p)$ is the actual confidence limit (sometimes referred to as the probability limit). Therefore, equations (4.3.16) can be written as follows:

$$\Pr \{ |X(p) - P| \leq \varepsilon \} = \text{erf} (t/\sqrt{2}) + R \quad (4.3.18)$$

The term $\text{erf} (t/\sqrt{2})$ represents the probability that the unavailability calculated for the confidence level P lies between $P \pm \varepsilon$. (e.g. for $P = 99\%$ confidence limit, $\varepsilon = 0.5\%$, and $n = 1200$ trials the value of $\text{erf} (t/\sqrt{2})$ is 0.918, which indicates that with the probability of 91.8 percent the unavailability calculated for the 99% confidence level lies between the 98.5% and 99.5% confidence levels.) The maximum error calculated from equation (4.3.17) represents the highest error

associated with probability given by $\text{erf}(t/\sqrt{2})$. (e.g., for $P = 99\%$, $\varepsilon = 0.5\%$ and $n = 1200$, the value of $|R|$ is 0.685% . Therefore, the maximum interval probability for 99% confidence level would be 91.1%).

4.4 Distribution Models for the Modules of a Coherent System Fault Tree

In this study, through extensive data observation and mathematical formulations, it became apparent that the unavailability distribution of the modules of system fault trees are closely related to their input component unavailability distribution.

As is discussed in Chapter 2 of this thesis, there are two types of proper modules, namely:

- (a) AND modules or
- (b) OR modules.

Because of the multiplicative properties of log-normal distributions, the distribution of the unavailability of an AND module is log-normally distributed, if all of the input components have log-normal distributions. Suppose module $M(\underline{C})$ has the following set of components as input:

$$\underline{C} = [c_1, c_2, \dots, c_n]$$

The log-normal distribution which represents the unavailability of the AND module $M(\underline{C})$ has the following parameters.

$$m = \prod_{i=1}^r m_i \quad (4.4.1)$$

$$f_m = \text{EXP} \left[\sum_{i=1}^r (\text{Ln } f_i)^2 \right]^{1/2} \quad (4.4.2)$$

where

m = median value of the AND module's unavailability distributions

f_m = 90% error factor of the AND module's unavailability distribution;

f_i = 90% error factor for the i -th component unavailability distribution
input to the AND module

For the case of OR gates in which addition of log-normal distributions exists, the outcoming distribution is no longer a log-normal distribution. However, it is known from the Central Limit Theorem that the sum of a population of statistically sufficient probability distributions of any type is a normal distribution. In practice, it is found in this study that most of the OR modules created from coherent system fault trees do not have statistically sufficient input components (i.e., 15 or more input components). Thus, a normal distribution is not an ideal distribution for the OR modules. An extensive review of the component failure data and observation of typical OR modules in this study showed that log-normal distribution is still a potential candidate to represent unavailability distribution of the OR modules. Because, sum of a statistically low population of log-normal distributions can be approximated by a log-normal distribution. A broad data observation, using the LIMITS Code verified that for the practical data range the sum of a small population of log-normal distributions can be still closely approximated as a log-normal distribution. By using the 50% median value and 90% error factor obtained from the simulation of the sum of these log-normal distributions from the LIMITS code, all of the parameters of a log-normal distribution for the sum were found. Chi-square tests were performed and

showed that, for the practical range of interest, an OR module could well be represented by a log-normal distribution. Also, in all of these data observations, the approximate log-normal unavailability distribution obtained for the OR module produced slightly longer tails compared to the real distribution. For example, consider an OR module with four input components. Suppose all of the input components have an unavailability that is log-normally distributed. The log-normal distribution parameters are shown in Table (4.4.1).

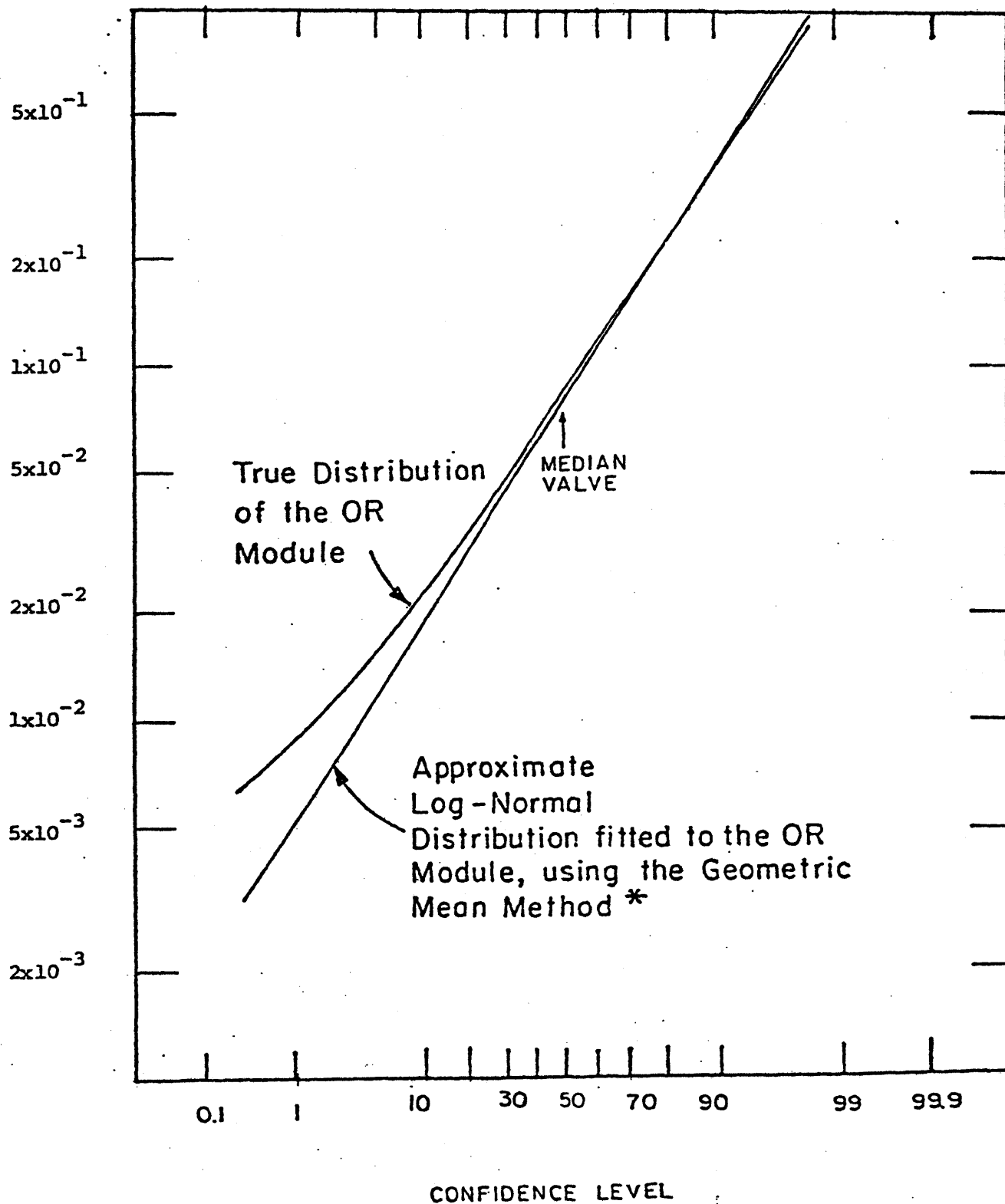
TABLE 4.4.1

LOG-NORMAL DISTRIBUTION PARAMETERS FOR THE
UNAVAILABILITY OF THE COMPONENTS INPUT TO AN OR MODULE

<u>Component Number</u>	<u>Median Value</u>	<u>90% Error Factor</u>
1	7.0×10^{-2}	9.0
2	1.3×10^{-4}	9.0
3	2.0×10^{-4}	5.0
4	7.0×10^{-3}	5.0

Figure (4.4.1) compares the true unavailability distribution obtained by a Monte-Carlo Simulation of the summation of these four input components and an approximate single log-normal unavailability assigned to the OR module. The effect of longer tails becomes negligible after 5% and before 95% confidence levels, and both approximate and true distributions estimate almost the same median value (i.e. 8.5×10^{-2}). The longer tails for the module's unavailability before 5% and after 95% confidence levels would result in a more conservative unavailability distribution for the top event. The longer tails and the adequacy of the

Figure 4.4.1 Comparison of the True and Approximate Log-Normal Distribution for an OR Module



*Median and Error Factor are Obtained from Data in Table (4.4.1) i.e.,
 $m = (7.0 \times 10^{-2} + 1.3 \times 10^{-4} + 2.0 \times 10^{-4} + 7.0 \times 10^{-3}) = 7.73 \times 10^{-2}$
 $\dots 1/4$

approximation justifies the use of this approximation in the fault tree simulation procedure.

To obtain parameters of an approximate log-normal unavailability distribution for the OR modules, the mathematical methods for the sum of distributions [7] will be used.

Consider r independent random variables $x_1, x_2, \dots, x_r$ that are summated such that

$$z = x_1 + x_2 + \dots + x_r$$

One can obtain mean and variance of variable z from $x_1, x_2, \dots, x_r$ as follows:

$$\bar{t}_z = \bar{t}_{x_1} + \bar{t}_{x_2} + \dots + \bar{t}_{x_r} \quad (4.4.3)$$

$$V_z = V_{x_1} + V_{x_2} + \dots + V_{x_r} \quad (4.4.4)$$

where

$\bar{t}_\alpha$ = mean for variate α

V_α = variance for variate α

The values of $\bar{t}$ and V are given by equations (4.2.5) and (4.2.6).

Therefore, for an OR module $M(\zeta)$, we have:

$$\bar{t}_{OR} = e^{\mu_{OR} + \sigma_{OR}^2/2} = \sum_{i=1}^r \left(e^{\mu_{c_i} + \sigma_{c_i}^2/2} \right) \quad (4.4.5)$$

and

$$v_{OR} = \bar{t}_{OR}^2 [e^{\sigma_{OR}^2} - 1] = \sum_{i=1}^r \left(e^{2\mu_{c_i} + \sigma_{c_i}^2} \right) \left(e^{\sigma_{c_i}^2} - 1 \right) \quad (4.4.6)$$

or by substituting equation (4.4.6) into (4.4.5) it follows:

$$\sigma_{OR}^2 = \exp \left[\frac{\sum_{i=1}^r \left(e^{2\mu_{c_i} + \sigma_{c_i}^2} \right) \left(e^{\sigma_{c_i}^2} - 1 \right)}{\left[\sum_{i=1}^r \left(e^{\mu_{c_i} + \sigma_{c_i}^2/2} \right) \right]^2} + 1 \right] \quad (4.4.7)$$

$$m_{OR} = e^{\mu_{OR}}, F_{OR} = \exp[1.645 \sigma_{OR}]$$

Therefore, by using equations (4.2.3) and (4.2.4), the values of μ_{c_i} and σ_{c_i} can be calculated for each component of the OR module. Finally, from Equations (4.4.7) and (4.2.4), the error factor for the approximate log-normal distribution of the OR module is found.

Another crude and simple method to calculate a more conservative value for the error factor of an OR module, would be the method of the geometric mean. It is observed in this study that the geometric mean of the error factors of the components input to an OR module provides a sufficiently accurate estimate of the error factor of the OR module itself.

For example, an OR module $M(c)$ would have a log-normal unavailability distribution with the following error factor:

$$F_{OR} = \left(\prod_{i=1}^r f_{c_i} \right)^{1/r} \quad (4.4.9)$$

where f_{c_i} is the 90% error factor of the i th component input to the OR module. Equations (4.2.4) and (4.4.5) will provide the module's median value.

Data observations proved that most of the time the crude geometric mean method provides slightly longer tails for the modules' unavailability distribution compared to the other method or the exact distribution. However, in the geometric mean method, slight errors appear when we are dealing with high input unavailabilities (e.g. 0.05 to 1) or large error factors (e.g. 30 and up). It should be noticed that in practice large unavailabilities or large error factors do not exist (the 90% error factor is often in the range of 2-20, and the unavailability in the range of 10^{-2} to 10^{-7}). In Figure (4.4.1) the unavailability distribution obtained by using the geometric mean method for determining the 90% error factor of the sum of the four log-normal distributions discussed earlier is compared to the true unavailability distribution.

The OR module approximation can also be applied to class 4 components (periodically tested components). The average unavailability equation for this class of component consists of the sum of 3 terms, each of which has a log-normal distribution. Therefore, one can use the approximation of fitting of the log-normal distribution for the class 4 average unavailability. The models discussed in this section are extensively

used in the PL-MODMC code, and several tests and comparisons with other Monte-Carlo codes have been performed to determine the accuracy of these models. Some of these comparisons are presented in Chapter 8 of this thesis.

4.5 The PL/I Random Number Generator in the PL-MODMC Code

The task of the PL-MODMC's random number generator is to provide random numbers for the sampling process. Random number generation is a time-consuming process. Thus, a random number generator must be made such that it is both efficient and accurate. The PL-MODMC code employs the Central Limit Theorem to program the random number generation procedure. First, in this method, n uniformly distributed numbers between zero and one are generated. Then if n is a statistically large number by adding these n uniform numbers, the resulting number would become a member of normally distributed data with a mean $n/2$. This results from the Central Limit Theorem which states that the sum of statistically sufficient uniformly distributed variables is normally distributed. Usually ten to twenty is assumed to be a statistically sufficient number. In most of the practical cases the value of n is taken as being 12. However, this may not be adequate for special cases, especially when we are interested in the confidence levels close to 0.0 and 1.0. In the PL-MODMC, the value of n can be given by the user in the input, and has a default value of 12.

In general, the Central Limit Theorem applies when the measurement, or random variable that we consider, is the sum of a large number of other variables. For example, let $\xi_1, \xi_2, \dots, \xi_n$ be uniformly distributed, mutually independent random variables for any value of n , with common mean μ and standard deviation σ .

$$\xi = \sum_{j=1}^n \xi_j \quad (4.5.1)$$

We have, as $n \rightarrow \infty$ [9],

$$\Pr (\xi \leq x) \approx \phi \left(\frac{x - n\mu}{\sigma\sqrt{n}} \right) \quad (4.5.2)$$

which states that ξ is asymptotically normal with $(n\mu, \sigma\sqrt{n})$. In particular, the mean $\bar{x}$ of a random sample of size n is asymptotically normal with $(\mu, \sigma/\sqrt{n})$. This is called the Central Limit Theorem Approximation.

The form of the Central Limit Theorem stated here is applicable to independent, uniformly distributed random variables with finite means and standard deviations because this version of the theorem is sufficient for our needs. Another useful version of this important theorem applies to independent random variables that are not uniformly distributed. The interested reader should consult the literature [10, 11] for a broader coverage of this theorem.

The programming configuration of the PL-MODMC's random number generator will be described in Chapter 7. Several tests proved the accuracy and efficiency of this random number generator. In particular, it is found that the mean and variance of the PL-MODMC random numbers are identical to those of the SAMPLE and LIMITS for the same value of n and identical large population of data. However, Chi-square tests showed that with a probability of 99.9 percent, 5000 random numbers generated by the PL-MODMC's random number generator with a Central Limit Theorem Approximation of 20 are normally distributed with a mean of 9.97, and a standard deviation of 1.374. The application of this

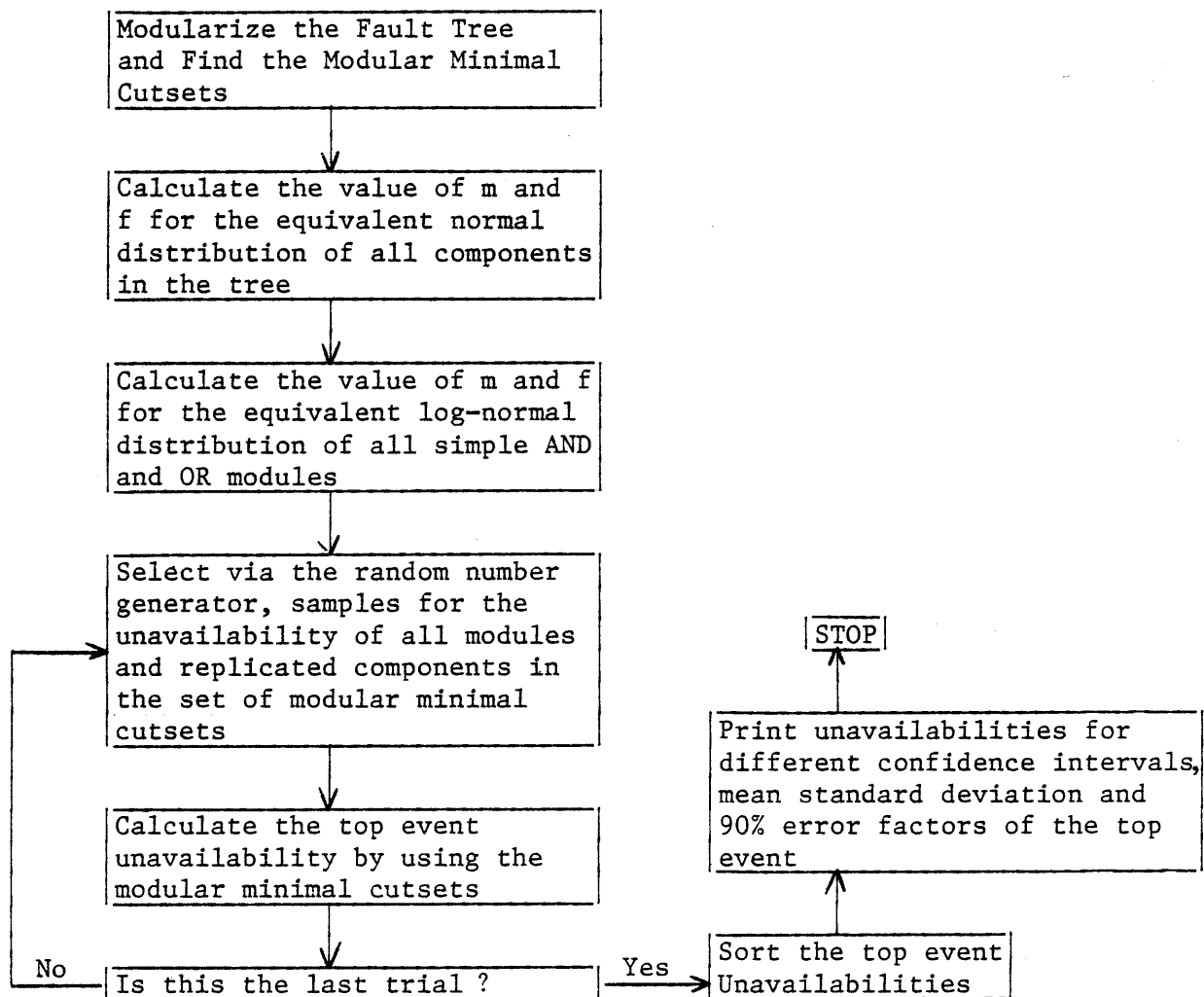
random number generator in the PL-MODMC code has provided a powerful tool with which to sample failure parameters from this log-normal distribution. Finally, it is found that this random number generator forms only a negligible fraction of the total Monte-Carlo simulation time consumed in the PL-MODMC code.

4.6 The Monte-Carlo Algorithm in PL-MODMC

The Monte-Carlo algorithm in the Code PL-MODMC is very simple and straightforward. This algorithm is shown in Figure (4.6.1). Once the log-normal distributions parameters m and f are known for the failure parameters of all components in the fault tree, the log-normal parameters for the average unavailabilities are calculated by using the mathematical formulations discussed in Section 4.3. Prior to the Monte-Carlo calculation, PL-MODMC generates the modular minimal cutsets which provide the logical information of the tree to calculate the top event probability in an efficient manner. By using equations (4.4.1) and (4.4.2) the exact log-normal distribution parameters of the AND modules in the tree can be obtained. The OR module approximation would also provide a log-normal distribution with the parameters calculated from either equation (4.4.7) or (4.4.9). Presently, for the practical range of parameters m and f , equation (4.4.9) is used in the PL-MODMC Code. However, for large error factors or large average unavailabilities, equation 4.4.7 can be used. At present, all of the members of the modular minimal cutsets are represented by equivalent log-normal distributions.

The Monte-Carlo simulation starts by selecting samples from the module or replicated component's log-normal distribution. The random

Figure 4.6.1 The Monte-Carlo Algorithm in PL-MODMC



number generators described in Section 4.5 is used to select samples from the modules or replicated events in the modular cutsets. A sample is selected by using the following equations:

$$t = \sigma \left(\sum_{i=1}^n R_i - \frac{n}{2} \right) + \mu \quad (4.6.1)$$

$$x = e^t \quad (4.6.2)$$

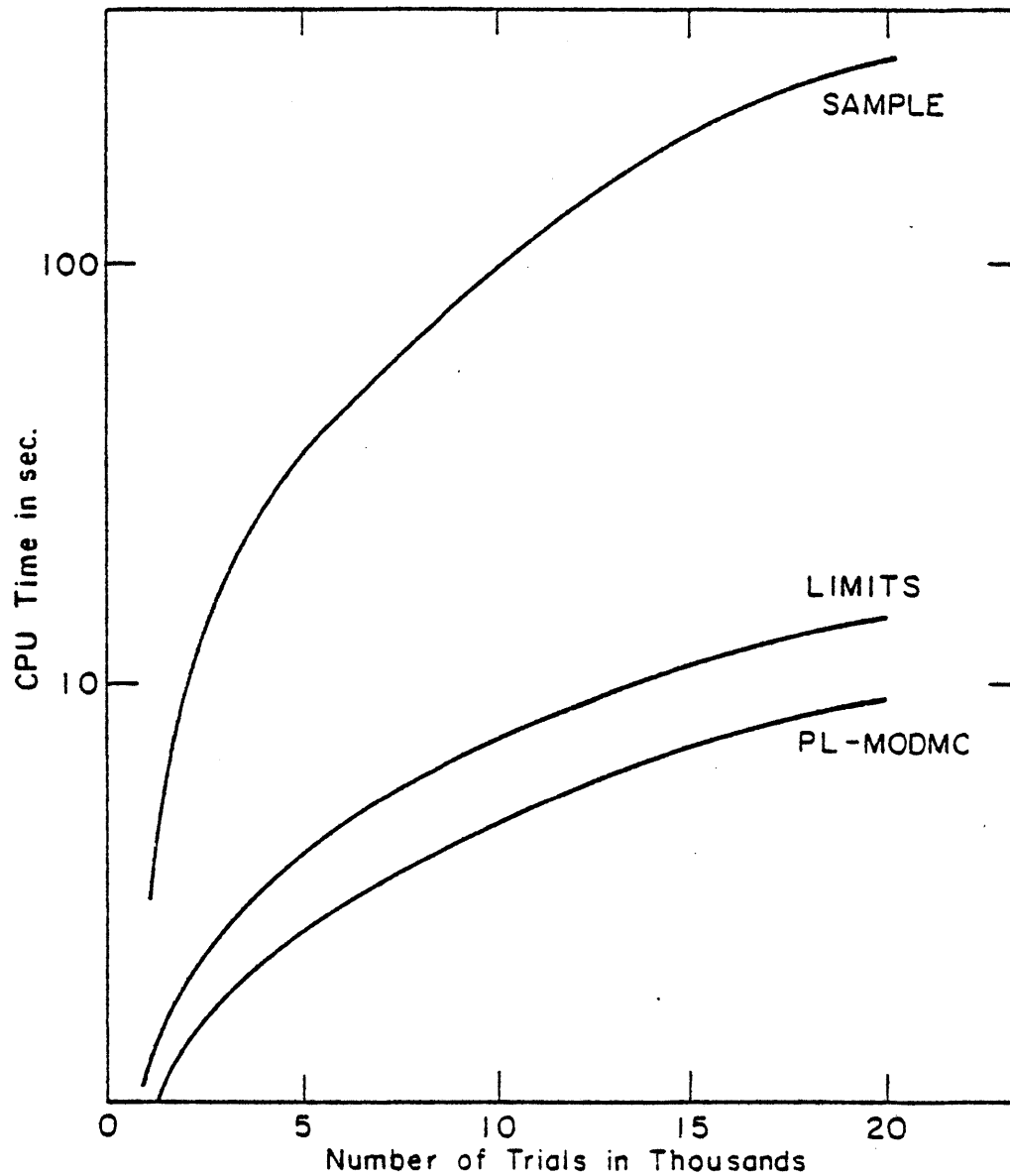
where x is the sample taken from the log-normal distribution with parameters σ and μ calculated from equations (4.2.3) and (4.2.4). R_i is a uniformly distributed random number between zero and one generated by the random number generator, and n is the number of terms in the Central Limit Theorem Approximation.

At each Monte-Carlo trial a single top event value is calculated by using the modular cutsets and by providing a single point unavailability value from Equation (4.6.2) for all modules and replicated components. This procedure is repeated for a large number of trials to make sure that an empirical distribution of the top event unavailability is adequately obtained. Top event unavailabilities obtained from the trials are sorted to obtain unavailabilities associated with different confidence levels.

Comparison with other Monte-Carlo Codes has proven the accuracy and efficiency of the PL-MODMC Code. For example, Figure (4.6.2) shows the CPU time consumed for a Monte-Carlo analysis of the fault tree shown in Figure (4.6.3). It should be noticed that input to PL-MODMC is the fault tree, and thus part of the CPU time shown in Figure (4.6.2) has gone to the modularization and modular cutset generation of the

Figure 4.6.2

The Execution Times of LIMITS, SAMPLE, and PL-MODMC Codes



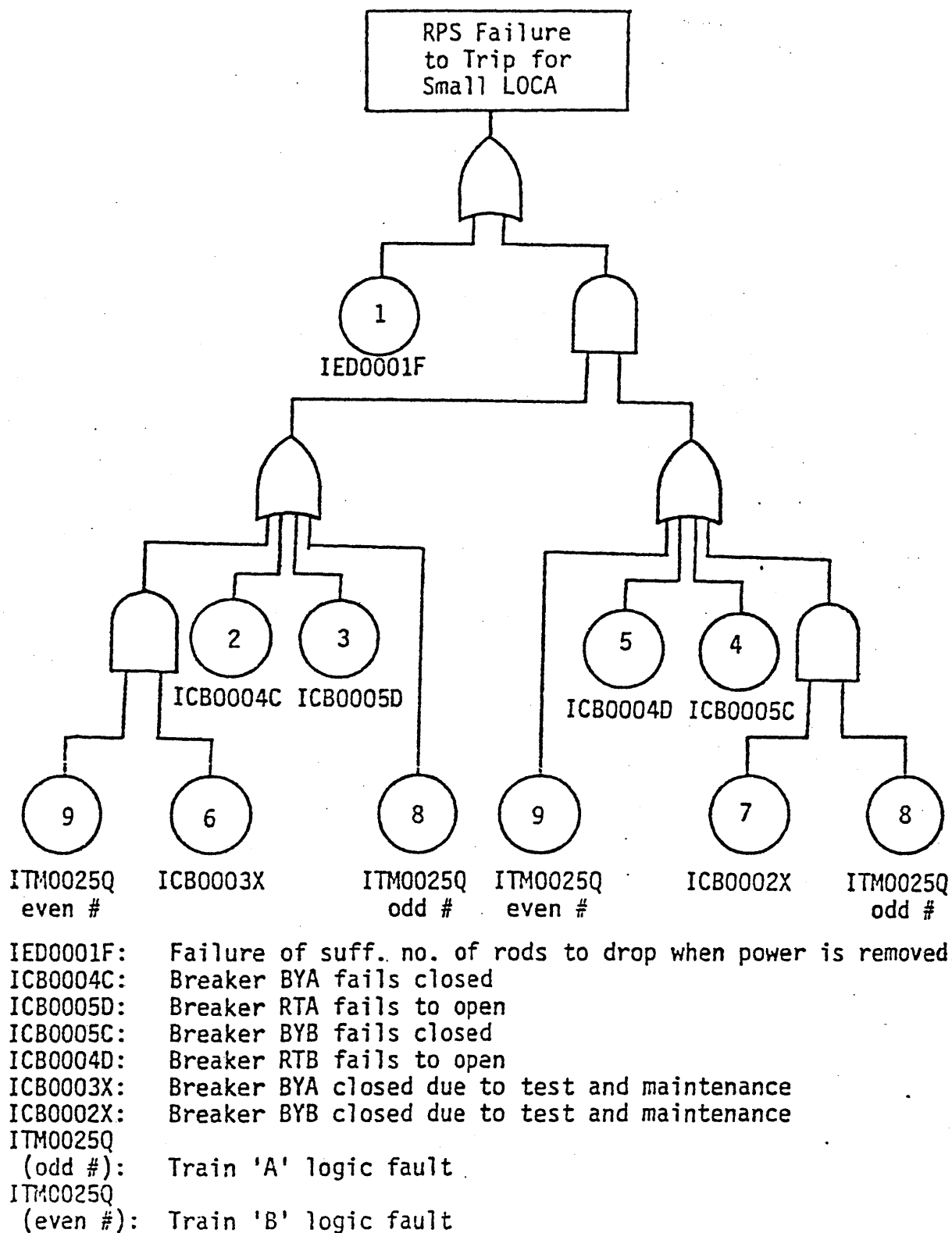


Figure 4.6.3 : Reduced Fault Tree of the Reactor Protection System for a PWR

input fault tree. Therefore, the actual simulation time has taken even less CPU time than that shown in Figure (4.6.2).

REFERENCES

- [1] "Reactor Safety Study", WASH-1400, U. S. Nuclear Regulatory Commission, (NUREG-75/014), 1975
- [2] Lee, Y. T. and S. L. Salem, "Probability Intervals for the Reliability of Complex Systems using Monte-Carlo Simulation", UCLA-ENG-7758, December 1977
- [3] Vesely, W. E. and F. F. Goldberg, "Time Dependent Availability Analysis of Nuclear Safety Systems using Randomly Sampled Failure Rates", U. S. Nuclear Regulatory Commission, NRC4/4
- [4] Vesely, W. E. and F. F. Goldberg, "FRANTIC - A Computer Code for Time Dependent Unavailability Analysis", U. S. Nuclear Regulatory Commission, NUREG-0193, October, 1977
- [5] Cashwell, E. D. and C. J. Everett, "A Practical Manual on the Monte-Carlo Method for Random Walk Problem", Pergammon Press, 1959
- [6] Dixon, W. J. and F. J. Massey, Jr.", Introduction to Statistical Analysis", 2nd Ed., McGraw Hill, 1957
- [7] Green, A. E. and A. J. Bourne, "Reliability Technology", John Wiley and Sons Ltd., 1972
- [8] Shell, D. L., "A High-Speed Sorting Procedure", Communication of the Associations for Computing Machinery, Vol. 3, No. 3, pp. 134-166, July, 1956
- [9] Lloyd, D. K. and M. Lipow, "Reliability: Management, Methods and Mathematics", Second edition, Prentice-Hall, 1962
- [10] Parzen, E., "Modern Probability Theory and its Applications", John Wiley and Sons, Ltd., New York, 1960
- [11] Loeve, M., "Probability Theory", Van Nostrand Reinhold Company, New York, 1950

CHAPTER 5

DEVELOPMENT OF THE CUTSET DISCRIMINATOR CODE MODCUT5.1 Introduction

The analysis of a large fault tree can lead to the generation of an enormous number of cutsets, if they are all included. In most analyses one desires only to retain those cutsets that make a significant contribution to the probability of the top event. Thus, it is useful to have an appropriate procedure for discriminating between various cutsets so that those contributing less than some prescribed amount to the top event probability can be eliminated. A computer code MODCUT based upon PL-MOD is developed which will systematically eliminate cutsets whose contribution lies below some specific value. MODCUT presently uses cutset size as the criterion for the cutset elimination. However, it is formulated so that other types of criteria may be used (e.g. cutset probability, or susceptibility to a common cause failure). MODCUT generates cutsets in two steps. First, cutsets of all proper modules are determined. Second, cutsets of proper modules found in the first step are inserted into the modular minimal cutsets of the top event in order to find top event cutsets. Obviously, the discrimination procedure can be applied in the first step and, if so, expansion of the number of top event cutsets in the second step can be effectively reduced to those cutsets that meet the discrimination criterion.

5.2 MODCUT Methodology

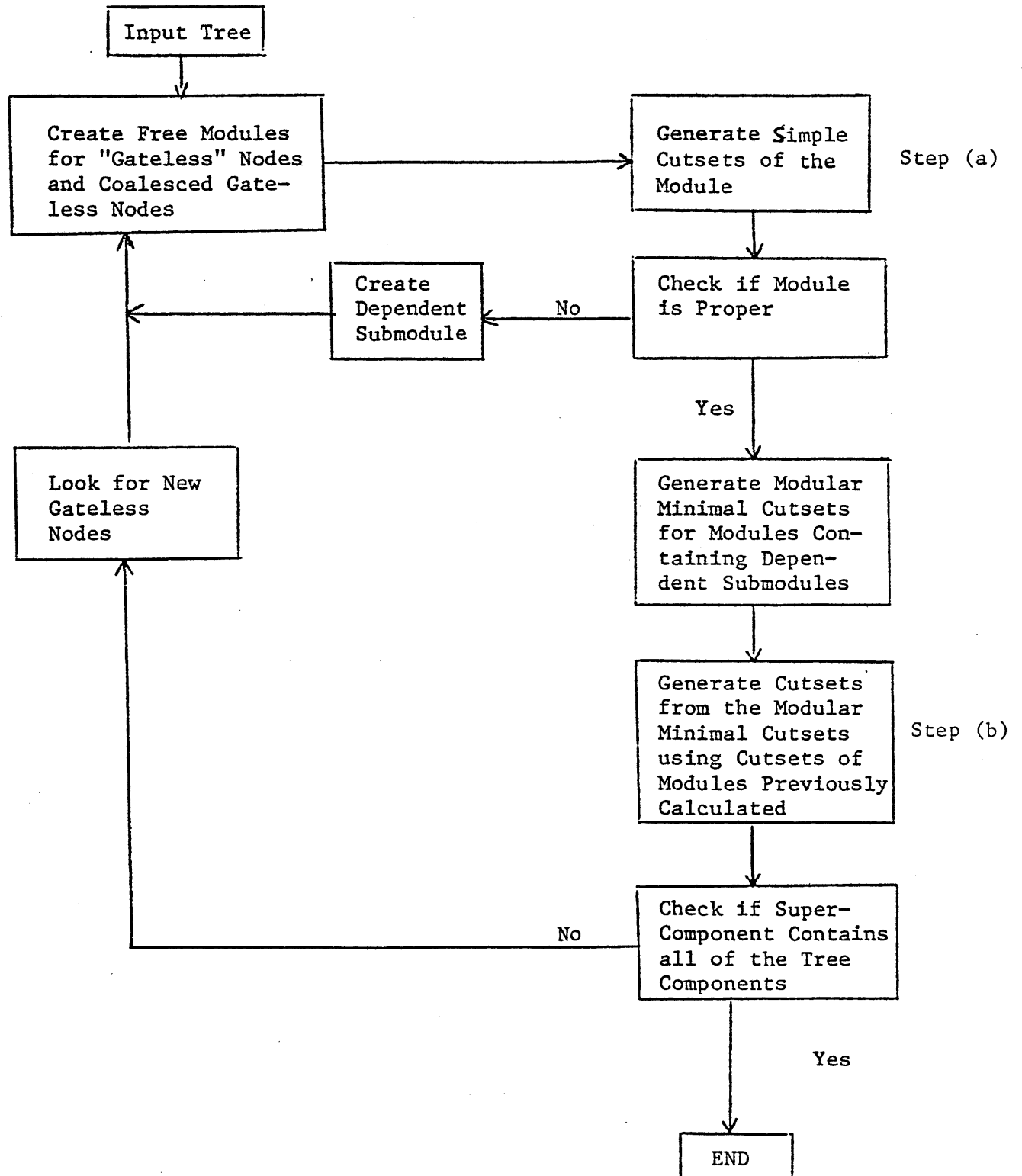
MODCUT is a computer program to reduce and generate the cutsets for the modules and top event of a complex fault tree. Since cutset generation is a time-consuming process, special methods are often used in order to reduce the problem to a manageable size. MODCUT uses the modular approach that was first developed in the PL-MOD code [1]. Bit strings are used to store the cutsets. Each cutset is represented by a bit-string. Therefore, each component in the tree occupies and is represented by only one bit in the computer. This significantly reduces core storage requirements. Bits that are 1 in a bit-string indicate a "failed", and 0, a "success" state for the corresponding component.

Figure (5.2.1) shows the algorithm which is used in the code MODCUT to generate simple cutsets from the modular cutsets. As can be seen, the cutsets are determined by a two-level process:

- (a) generation of cutsets of all proper modules;
- (b) generation of cutsets of all replicated gates, K-out-of-N gates, and the top event from their modular minimal cutsets.

These two steps are clearly shown in Figure (5.2.1). In MODCUT, cutsets at the intermediate levels can be properly reduced by using any discrimination criterion and, thus, reduce the number of top event cutsets down to a manageable size. The reduction process can be performed by applying a cutset size discrimination criterion, and/or

Figure 5.2.1
Cutset Generation Algorithm in MODCUT



any other criterion (e.g. common cause failure). Presently, reductions are made based on a cutset size discrimination.

Prior to the minimal cutset generation, the number of minimal cutsets for the intermediate levels and the top event can be efficiently calculated in MODCUT, in order to investigate the complexity, to adopt proper reduction policies, and to choose a reasonable value for the discrimination criterion. The number of cutsets for each gate is obtained by using the fact that OR gates increase the number of cutsets, and AND gates increase the size of the cutsets.

At the end of each reduction step, all of the spaces occupied by the unnecessary bit-strings, arrays, and structures will be freed in the computer, and thus an optimized computer core space will be maintained throughout the calculations. It is not necessary to indicate in the input the number of times that a component or a gate replicates, or in which domain it replicates. This will be automatically calculated by MODCUT through a special search for replicated events. However, in the code PL-MOD it is necessary to explicitly identify replicated gates and components by a special set of numbers (see Reference [1]).

Two size options may be input into MODCUT; one for the cutset size discrimination of the top event, and one for the cutset size discrimination of all replicated and K-out-of-N gates in the fault tree. In most of the large fault trees with many cutsets, the cutset size criterion for the replicated modules and K-out-of-N gates can be assigned a lower value than that of the top event. Because, if these replicated and K-out-of-N gates are interacting with some AND gates in the domain of the top event, then the size of the top event cutsets becomes larger than those of the replicated and K-out-of-N gates. However, only those top event cutsets

that are found by using the cutsets of replicated and K-out-of-N gates will have a larger cutset size than the size of the cutsets of replicated and K-out-of-N gates. In the next section (Section 5.3) it will be shown that using a lower size criterion will significantly save computer time for large fault trees.

MODCUT is capable of handling K-out-of-N gates, and also components with a complement (A and $\bar{A}$). Benchmark tests with other cutset generation codes such as MOCUS and MICSUP have proven MODCUT to be an accurate, fast and easy program for qualitative fault tree evaluation. Some of the fault trees which have been analyzed with the MODCUT code are presented in Chapter 8 of this thesis.

To clarify the algorithm described in this section an example is presented in the next section. In this example the proper modules and modular cutsets are first obtained and then simple cutsets are derived from them. The method of obtaining proper modules and modular cutsets has already been explicitly described in Reference [1] and Chapter 1 of this thesis. The example presented in the next section considers a typical fault tree and the steps shown to derive simple cutsets are exactly similar to those used in MODCUT. This example is used again in Chapter 8 as an example for using the MODCUT code, and the complete output of the MODCUT is given in that chapter. In order to more easily understand the MODCUT procedure for the fault tree described in the next section, the reader may simultaneously follow the MODCUT output for this example which is given in Table (8.4.3) of Chapter 8.

5.3 An Example Showing the Generation of Simple Cutsets by the MODCUT Algorithm

The fault tree shown in Figure (5.3.1) is an arbitrary fault tree containing replicated gate and components. Component C5 in this fault tree appears in both domains of replicated gate G8 and the top event gate TOP. In order to generate the simple cutsets of this fault tree, all of the replicated components are treated in a way similar to that of the non-replicated components. Hence, in Figure (5.3.1) gates G2 and G8 would temporarily become independent of each other. Similarly, gates G6 and G7 as well as G1 and G5 become independent.

The modularization process for such a fault tree is straightforward. First, the free modules are found and then the simple cutsets are immediately generated by calling an external procedure called CUTSET (i.e. step (a) shown in Figure (5.2.1)). The free modules in this example are G7, G9, G4, and G8. Modules G7 and G9 contain no input module, but modules G4 and G8 have modules G7 and G9 as inputs, respectively. The cutsets of free modules¹ (step (a) shown in Figure (5.2.1)) are found in the following steps:

```

Step 1   FREE MODULE NAME = G7

          TYPE = AND

          COMPONENTS INPUT = C3, C4

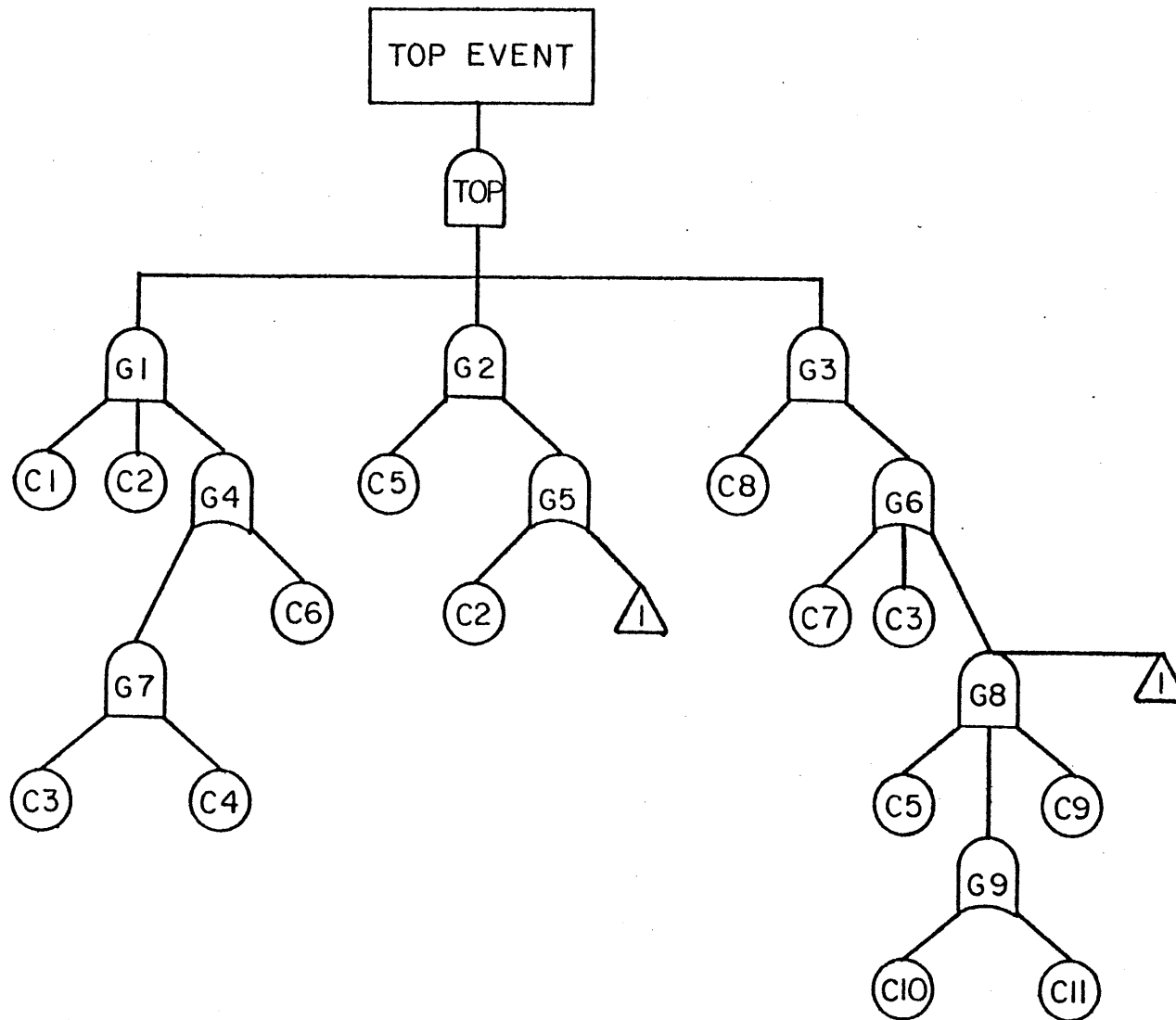
          MODULES INPUT = ——

          CUTSETS (1) C3 C4

```

¹Free, nested, and parent modules are all considered proper modules.

Figure 5.3.1 Sample Fault Tree for the MODCUT Algorithm



Step 2 FREE MODULE NAME = G9
 TYPE = OR
 COMPONENTS INPUT = C10, C11
 MODULES INPUT = —
 CUTSETS (1) C10
 (2) C11

Step 3 FREE MODULE NAME = G4
 TYPE = OR
 COMPONENTS INPUT = C6
 MODULES INPUT = G7
 CUTSETS (1) C6
 (2) C3 C4

Step 4 FREE MODULE NAME = G8
 TYPE = AND
 COMPONENTS INPUT = C5, C9
 MODULES INPUT = G9
 CUTSETS (1) C5 C9 C10
 (2) C5 C9 C11

It should be noted that, for the free and nested modules, the cutsets that are obtained will not be minimized, unless the user requests their cutsets by using the appropriate option in the input. However, the minimization procedure will always be performed automatically for the replicated gates and the top event.

Since the gate G8 in this example is a replicated gate, the minimal cutsets are automatically calculated and, if requested, they

will be printed out too. In this specific case no non-minimal cutsets exist. Also, the cutset size discrimination will be performed before finding the minimal cutsets. For example, if the maximum cutset size for the replicated and K-out-of-N gates is specified to be equal to 2, both cutsets of the replicated module G8 will be eliminated before the minimization procedure starts (in this special case no cutset remains for the minimization procedure if the cutset size discrimination is less than 3). It is obvious that this significantly reduces the computation effort for determining minimal cutsets of all proper or improper modules in which the gate G8 is an input (e.g. gates G6, G3 and TOP). The minimization method that is used in MODCUT will be discussed in more detail later in this section.

At this point all of the cutsets of free modules are calculated and minimized by the MODCUT Code. However, all of these cutsets are in the form of bit strings. For example, cutsets of the module G8 that are found in the CUT-SET procedure are stored in a structure called CUTSET. COMP(I) stores the bit-strings corresponding to the minimal cutsets. Therefore, from our discussion in Step (4) regarding finding cutsets of free modules, it transpires that the structure CUTSET for the module G8 must have the following values:

```

CUTSET. NAME = G8
CUTSET. LIL(1) = C5
CUTSET. LIL(2) = C9
CUTSET. LIL(3) = C10
CUTSET. LIL(4) = C11
CUTSET. COMP(1) = '1 1 1 0 'B → C5  C9  C10
CUTSET. COMP(2) = '1 1 0 1 'B → C5  C9  C11

```


Obviously, the first bit in the CUTSET.COMP(I)bit-string corresponds to the component stored in variable CUTSET.LIL(1) and the second bit corresponds to CUTSET.LIL(2),...,etc. It is noticed that the CUTSET structure is allocated for module G8 such that the length of the bit-strings is 4 and their dimension is 2, because there are 4 different components in the cutsets of module G8 and a total of 2 cutsets present. Similar to module G8, the CUTSET structure of the modules G7, G9 and G4 would have the following values:

CUTSET.NAME = G7

CUTSET.LIL(1) = C3

CUTSET.LIL(2) = C4

CUTSET.COMP(1) = '1 1' B $\rightarrow$ C3 C4

CUTSET.NAME = G9

CUTSET.LIL(1) = C10

CUTSET.LIL(2) = C11

CUTSET.COMP(1) = '1 0' B $\rightarrow$ C10

CUTSET.COMP(2) = '0 1' B $\rightarrow$ C11

CUTSET.NAME = G4

CUTSET.LIL(1) = C6

CUTSET.LIL(2) = C3

CUTSET.LIL(3) = C4

CUTSET.COMP(1) = '1 0 0' B $\rightarrow$ C6

CUTSET.COMP(2) = '0 1 1' B $\rightarrow$ C3 C4

After determining the simple cutsets of the free modules, the modularization algorithm continues by generating the modular minimal cutsets for the gate G1 which is a 2-out-of-3 gate. These modular cutsets are as follows:

```

SYMMETRIC MODULE NAME = G1

                TYPE = 2 out of 3

        MODULES INPUT = G4

MODULAR CUTSETS = (1)  C1  C2
                  (2)  C1  G4
                  (3)  C2  G4

```

Similar to the CUTSET structure, the modular cutsets are represented by the following bit strings in the MODCUT code;

```

(1)  '1  0  1' B
(2)  '0  1  1' B
(3)  '1  1  0' B

```

where the last bit represents the appearance of module G4, and the first two bits represent Components C1 and C2, respectively. To generate simple cutsets of module G1, MODCUT replaces the last bit with its corresponding simple cutsets calculated in step 3. The length of simple cutsets for module G4 is 3, and there have been 2 simple cutsets previously calculated for this module. Hence, the length of bit-strings representing the cutsets of module G1 is 5 (i.e. 2 components are directly input to gate G1, and 3 components represent gate G4 which is input to the symmetric gate G1; therefore, a total of 5 components represent module G1). Also, a total of 5 bit-strings can be obtained

for gate G1 (i.e. 5 simple cutsets exist for module G1). Because module G4 has 2 cutsets and the last two modular cutsets of module G1 contain module G4, therefore each represents 2 simple cutsets, and the first modular cutset represents only one simple cutset. Thus, a total of 5 simple cutsets exists for module G1. At this point MODCUT would automatically allocate the CUTSET structure with both length and dimension of the bit strings equal to 5. The following values are assigned to the CUTSET structure in MODCUT through a special procedure called TOP-SET. The TOP-SET procedure performs step (b) shown in Figure (5.2.1).

Step 5

```

CUTSET.NAME = G1

CUTSET.LIL(1) = C1
CUTSET.LIL(2) = C2
CUTSET.LIL(3) = C6
CUTSET.LIL(4) = C3
CUTSET.LIL(5) = C4

CUTSET.COMP(1) = '1 0 1 0 0' B → C1 C6
CUTSET.COMP(2) = '1 0 0 1 1' B → C1 C3 C4
CUTSET.COMP(3) = '0 1 1 0 0' B → C2 C6
CUTSET.COMP(4) = '0 1 0 1 1' B → C2 C3 C4
CUTSET.COMP(5) = '1 1 0 0 0' B → C1 C2

```

The procedure for finding the minimal cutsets is performed by MODCUT in this step and no non-minimal cutset is identified. However, if non-minimal cutsets were found, the computer storage occupied by them

would be released internally by the MODCUT code in order to maintain a minimum storage requirement. In addition, elimination of non-minimal cutsets in a gate, say G, would reduce expansion of the number of cutsets for all gates in which gate G is an input. Also, size discrimination would eliminate the non-essential minimal cutsets and thus reduce the trend in expansion of the cutsets. For example, size discrimination, of say 2, could reduce the number of cutsets in module G1 from 5 to 3 by eliminating cutsets numbers 2 and 4 that are of size three.

MODCUT continues the cutset generation procedure for further modularizing the tree. For generating the modular minimal cutsets of the top event, the presence of the replicated module G8 must be accounted for. Thus, some of the gates should become nested modules, e.g. the input replicated event G8 is removed, so as to change the modules by which gate G8 is an input to proper modules. Components C5 and C8 are coalesced because gates G2, G3 and TOP are all AND gates. The cutsets of the newly-created nested and parent modules are calculated in a way similar to that of the free modules discussed earlier in this section by calling the CUT-SET procedure. Hence, the following steps will determine the cutsets of all nested and parent modules in this tree. These steps are all automatically performed in MODCUT.

```

Step 6   PARENT MODULE NAME = *TOP

          TYPE = AND

          COMPONENTS INPUT = C5, C8

          MODULES INPUT = G1

```

```

CUTSETS (1) C5  C8  C1  C6
          (2) C5  C8  C1  C3  C4
          (3) C5  C8  C2  C6
          (4) C5  C8  C2  C3  C4
          (5) C5  C8  C1  C2

```

Since the parent module *TOP is an AND gate with input components, C5 and C8, and gate G1, it would be sufficient to add components C5 and C8 to all of the cutsets of the gate G1 to obtain cutsets of the parent module *TOP. This procedure is automatically performed in the CUT-SET procedure of the MODCUT Code. Similarly, the cutsets of all nested modules are obtained by MODCUT as follows:

```

Step 7  NESTED MODULE NAME = G5

          TYPE = OR

          COMPONENTS INPUT = C2

          MODULES INPUT = ———

          CUTSETS (1)  C2

```

```

Step 8  NESTED MODULE NAME = G6

          TYPE = OR

          COMPONENTS INPUT = C3  C7

          MODULES INPUT = ———

          CUTSETS  (1)  C3
                   (2)  C7

```

Finally, the modularization algorithm generates the modular cutsets for the top event (i.e., gate TOP). The modular cutsets are as follows:

```

Step 9    TOP EVENT = TOP

          REPLICATED MODULES INPUT = G8

          PARENT MODULE INPUT = *TOP

          NESTED MODULES INPUT = G5, G6

          MODULAR CUTSETS  (1)  *TOP  G5  G6
                           (2)  *TOP  G8

```

Due to the presence of the replicated components C2, C3, and C5, there are dependences between modules *TOP, G5, G6 and G8 in which these components are input. However, we now generate cutsets of the top event regardless of these dependences. In the next step proper action must be taken in order to treat the effect of these dependences. The "ORing" method is used to treat these dependences. In this method, bits corresponding to a specific replicated component will be ORed with each other by using the OR operator of the Boolean Algebra. The "ORing" method will be further discussed later in this section when the dependences between the bits in the bit-strings of the TOP gates are being treated.

In MODCUT the two sets of modular cutsets shown in Step 9 are represented by the following bit-strings for the gate TOP (i.e. top event).

(1) '0 1 1 1' B

(2) '1 1 0 0' B

In these two sets of bit-strings, bits 1 through 4 represent modules G8, *TOP, G5 and G6, respectively. The first set, which represents 10 simple cutsets, is obtained by multiplying the number of simple cutsets in modules *TOP, G5 and G6. Similarly, the second set consists of 10 cutsets. Therefore, a total of 20 non-minimal cutsets (with a length of the representative bit strings equal to 14) exist for the top event. The length of the bit strings is the sum of the length of bit-strings for all of the modules in the modular cutsets of the TOP gate (i.e. G8, *TOP, G5 and G6). Procedure TOP-SET will determine the simple cutsets of the top event from its modular cutsets (step (b) in Figure (5.2.1)). This process is similar to the one discussed previously for the symmetric gate G1. At this step the CUTSET structure would have the following values in the MODCUT Code:

```

Step 10  CUTSET.NAME    = TOP
          CUTSET.LIL(1) = C5
          CUTSET.LIL(2) = C9
          CUTSET.LIL(3) = C10
          CUTSET.LIL(4) = C11
          CUTSET.LIL(5) = C5
          CUTSET.LIL(6) = C8
          CUTSET.LIL(7) = C1
          CUTSET.LIL(8) = C2
          CUTSET.LIL(9) = C6

```

```

CUTSET.LIL(10) = C3
CUTSET.LIL(11) = C4
CUTSET.LIL(12) = C2
CUTSET.LIL(13) = C3
CUTSET.LIL(14) = C7
CUTSET.COMP(1) = '0 0 0 0 1 1 1 0 1 0 0 1 1 0' B
CUTSET.COMP(2) = '0 0 0 0 1 1 1 0 1 0 0 1 0 1' B
CUTSET.COMP(3) = '0 0 0 0 1 1 1 0 0 1 1 1 1 0' B
CUTSET.COMP(4) = '0 0 0 0 1 1 1 0 0 1 1 1 0 1' B
CUTSET.COMP(5) = '0 0 0 0 1 1 0 1 1 0 0 1 1 0' B
      .
      .
      .
CUTSET.COMP(20)= '1 1 0 1 1 1 1 1 0 0 0 0 0 0' B

```

It is clear now that whenever simple cutsets are to be generated from the modular minimal cutsets (i.e. improper modules), the procedure TOP-SET is called (Step (b) in Figure (5.2.1)), and if simple cutsets are to be found from proper modules (i.e., free, nested, and parent modules) the procedure CUT-SET is called (Step (a) in Figure (5.2.1)).

To treat the replicated events the ORing concept is used. For example, in cutset number 3 of the top event, the bits representing replicated components C5, C2 and C3 are shown below:

	C5		C5		C2		C3		C2	C3					
	↓		↓		↓		↓		↓	↓					
CUTSET.COMP(3) =	' 0	0	0	0	1	1	1	0	0	1	1	1	1	0	' B
		1	2	3	4	5	6	7	8	9	10	11	12	13	14

It can be seen that components C2, C3 and C5 each appear twice in this cutset, which is logically impossible. Only one is sufficient to indicate the appearance of these components in this cutset. The simultaneous appearance is due to the fact that the replicated events were made independent of each other before the modularization process started. Therefore, one of the bits representing this identical appearance of components must be changed to zero along with its corresponding CUTSET.LIL(I). In MODCUT the first appearance of a replicated component is always considered to represent that component in the bit strings. Thus, the bits representing a replicated component after its first appearance should be ORed with the first bit showing the appearance of that component, and their values must be set to zero. Obviously, if two bits that are ORed result in zero then both bits must be zero; otherwise, the outcome of ORing is always 1. The bit-string CUTSET.COMP(3) would thus change to the following form after this ORing process.

				C5			C5			C2		C3		C2	C3
				↓			↓			↓		↓		↓	↓
CUTSET.COMP(3) =	'1	0	0	0	0	1	1	1	0	1	1	0	0	0	0' B

This procedure must be repeated for all cutsets and the following form of the bitstrings is obtained for the bit strings representing the top event cutsets:

```

CUTSET.COMP(1) = '1 0 0 0 0 1 1 1 1 1 0 0 0 0' B
CUTSET.COMP(2) = '1 0 0 0 0 1 1 1 1 0 0 0 0 1' B
CUTSET.COMP(3) = '1 0 0 0 0 1 1 1 0 1 1 0 0 0' B
CUTSET.COMP(4) = '1 0 0 0 0 1 1 1 0 1 1 0 0 1' B
CUTSET.COMP(5) = '1 0 0 0 0 1 0 1 1 1 0 0 0 0' B
      .
      .
      .
CUTSET.COMP(20)= '1 1 0 1 0 1 1 1 0 0 0 0 0 0' B

```

The discrimination step would further eliminate some of the cutsets which do not meet the required criteria. For example in case of cutset size discrimination, if the maximum size of the cutsets is specified as 6 for the top event, then CUTSET.COMP(4) must be eliminated. After the discrimination procedure, the non-minimal cutsets from the remaining cutsets should also be removed. MODCUT uses the ORing method for the identification of bit-strings representing the non-minimal cutsets. For example, if $\text{CUTSET.COMP}(1) \mid \text{CUTSET.COMP}(2)$, where $\mid$ is the OR sign in Boolean Algebra, is equal to either CUTSET.COMP(1) or CUTSET.COMP(2), that CUTSET is non-minimal. For instance, it can be seen that $\text{CUTSET.COMP}(3) \mid \text{CUTSET.COMP}(4) = \text{CUTSET.COMP}(4)$, and therefore CUTSET.COMP(4) is a non-minimal cutset.

The identification of the non-minimal cutsets is the most time-consuming part of the MODCUT, and of similar cutset generation codes; this is because each bitstring must be compared to all other bit strings that are not tested through the minimization procedure.

Cutset discrimination policies can significantly affect the expansion of the simple cutsets. Sometimes lower order cutset size may be specified

as the discrimination criterion for replicated and K-out-of-N gates in the fault tree. For example, in the fault tree given in Figure (5.3.1), due to the presence of AND gates attached to the top event the lowest order of the simple cutsets that can be obtained from the proper modules input to the top event is 3. And suppose the maximum size of the cutsets requested for the top event is 5, then all the cutsets of the gates having a cutset size of more than 2 can be eliminated. One would therefore use the maximum size of 2 for the cutsets of gates G1 and G8 (i.e. K-out-of-N and the replicated gate) if the top event cutset size is 5. This reduces the number of bit strings of the top event from 20 to 11, and still results in the same set of cutsets with order 5 and less for the top event. As a result of this action, the computation time for generating cutsets from the modular structure of the tree would be reduced from 1.02 seconds to 0.4 seconds, which is a factor of 2.5 gain. This factor becomes much larger as the fault tree gets larger.

After determining the minimal cutsets for the intermediate gates and the top event, they can be printed out if requested by the user. However, before they are printed out a procedure should be used in order to sort the cutsets from the lowest to the highest order. The method of sorting in MODCUT is very simple. During the size discrimination an array called CUT is allocated to store the values of the cutset sizes that are lower than the specified maximum cutset size. Later in the minimization procedure the corresponding size values in the array CUT for the non-minimal cutsets are also removed. Finally, a DO loop orders the cutsets by selecting the lowest value in the CUT array to the highest. Therefore, the DO loop operates and selects cutsets from size 1

to the maximum size of the cutsets specified by the user for the top event (see Chapter 7 for more detail).

Minimal cutsets of up to size 5 for the top event of the fault tree shown in Figure (5.3.1) are as follows:

1)	C5	C8	C2	C6	C7
2)	C5	C8	C2	C6	C3
3)	C5	C8	C2	C3	C4
4)	C5	C8	C1	C2	C7
5)	C5	C8	C1	C2	C3

REFERENCES

- [1] Olmos, J. and L. Wolfe, "A Modular Approach to Fault Tree and Reliability Analysis", MITNE-209, August, 1977

CHAPTER 6

COMMON CAUSE ANALYSIS

6.1 Introduction

Common cause failure analysis is becoming an increasingly important segment of a complete system reliability analysis. A common cause failure (sometimes called common mode failure) is any occurrence or condition that results in multiple component failures. A common cause failure analysis attempts to identify those events which result in system failure by multiple component failures due to a single cause.

In highly redundant systems quite often common cause events are dominant contributors; hence the need for an effective method for analyzing these events. In this chapter some of the methods of common cause failure analysis from the qualitative and quantitative points of view are studied. The use of MOCUS [1], BACFIRE [2] and MODCUT codes in qualitative common cause failure analysis of complex technical systems has been studied. Further, the MOCUS-BACFIRE and MODCUT-BACFIRE packages are developed to simplify the use of these codes by coupling them.

In this chapter some of the results obtained through the review of qualitative and quantitative aspects of common cause failures in complex technical systems are summarized. In addition, a brief review of available common cause failure quantification models along with a potential model which is developed in this study is presented.

6.2 Review and Background Information

Common cause failure analysis is not new. Methods for a systematic approach to common cause failure analysis have been presented in the literature. However, the complexity of the current reactor safety systems has made this type of analysis very difficult without the substantial aid of computers. Even the computer-aided approaches are often prohibitive in terms of computation time and computer storage for complex systems.

Concern over common cause failure (CCF) in nuclear safety systems has been expressed for a number of years. In 1957, Siddall [3] reported the importance of common design and maintenance in Canadian reactor protection system. In the United States there were early developments in multichannel instrumentation systems in the naval reactor development program and at the National Laboratories. In 1961 Epler [4], in his review of the Heat-Transfer Reactor Experiment No. 3 (HTRE-3) excursion, called the phenomenon a Systematic Failure. The phenomenon was acknowledged as an issue in 1968 and called a common-mode failure during an information meeting convened by the Atomic Energy Commission (AEC) to explore the possibility of using the same instrumentation for both protection and control. In 1969, Epler [5] commented on several common cause failures that had occurred in safety systems which utilize redundant instrument channels. He stated that:

"With such an array of four or five identical channels, the probability of all, or nearly all, failing because of randomly occurring internal faults can be made extremely small....it is not evident, however, that the

probability of all of the channels failing as the result of a single external or environmental event is acceptably small."

Certainly, the most thorough CCF analysis was performed as part of the RSS [6] in each of the following analysis steps:

- Event tree construction
- Fault tree construction
- Fault tree quantification
- Event tree quantification
- Special engineering investigations

In particular, in the fault tree construction stage the following steps were considered:

- Resolution of failures to a level such that common system hardware will be identified
- Fault tree constructions which identify human interfaces, test and maintenance interfaces, and other interfaces of potential dependency.

Computerized efforts in the qualification and quantification procedure of common cause failures were begun with special emphasis after the publication of the RSS. Burdick reported on the COMCAN code [7]. This code is designed to aid in the qualitative determination of:

1. Single secondary events that could fail entire minimal cut sets.
2. Common links among components such as common circuitry, common maintenance personnel or common manufacturer for components having events in the same minimal cutsets

As a result COMCAN requires, as one of its inputs, a set of minimal cutsets which have been obtained by some other qualitative analysis code. This approach presupposes, as Burdick rightly observed, an accurate fault tree for the system and requires at the same time retention and analysis of higher order minimal cutsets than are usually required for a reasonably accurate analysis. It is quite apparent that for large fault trees with a high degree of replications and the possibilities of several common cause failures, the computational efforts involved in the minimal cut set sorting become tremendous, if not impossible, due to the large amount of minimal cutsets. This is certainly the most apparent drawback of the straightforward application of any minimal cutset approach. Furthermore, no mention was made of how minimal cutsets of arbitrarily high order are generated. COMCAN does not offer any quantitative analysis capability. However, the development of this code laid the ground for the first time of certain rules which are basic and will be common to all future efforts involving common cause handling. The most important thing which was realized during the course of the code's development was the recognition of the importance of generic classification since the possible physical sources for a specific secondary event can develop into an absolutely endless list. Only a unique generic classification makes the search for minimal cutsets sharing

susceptibilities a tractable task. For this purpose, a preprocessor has been implemented which needs as input generic categorization of secondary causes. A special category has been added which links conditions among components to locate potential common-cause failures of minimal cutsets. The other three categories considered by COMCAN are:

Mechanical or Thermal

Electrical or Radiation

Chemical or Miscellaneous

Wagner et. al., [8,9] recognized the same importance of generic classification. Moreover, they realized that existing methods such as COMCAN need to be extended because they require all the system hardware minimal cut-sets. However, the determination of all minimal cut-sets including those of highest orders for a complex system is not possible due to computer limitations. Therefore, these authors presented an alternative which does not require all such minimal cut-sets and yet predicts the same results as if those were determined. In essence, Wagner followed basically the INEL approach by describing the three broad categories of generic causes which have already been mentioned. However, Wagner et. al., [8,9] suggested a new procedure to identify common cause failure of intermediate minimal cut-sets by using the concept of a dummy event which contains all the information necessary for the analysis of the next level of the fault tree. The dummy events are expanded and common cause candidates for the TOP event are obtained on a step-by-step analysis progressing from the bottom

to the top of the fault tree. From there on the intermediate minimal cut-sets are formed by conventional means.

In a review study for the Reactor Safety Study (WASH-1400)[10] (Lewis Review Group) it is stated that "...it is of course true that even for common cause failures all the normal defenses of diversity and redundancy are at least to some extent active. We believe that the principal assurance against common cause failures must be in dealing with the initiating events".

There has been some controversy as to the adequacy of methods of quantifying CCF's. The state-of-the-art in common cause quantification is largely determined by the work described in WASH-1400 and the new developments by Vesely [11]. Whereas in WASH-1400 common cause failure probabilities were evaluated using upper and lower probability bounds, Vesely showed that common cause failure probabilities can also be evaluated directly by using the multivariate exponential model devised by Marshall and Olkin. For this purpose, Vesely developed a statistical estimation technique for cases where common cause failures are repairable. Two special cases were examined, the constant failure rate case and the binomial failure rate case. A BWR scram system served as an example. In a recent paper, Johnson and Vesely [12] applied the same technique to the common mode analysis of valve leakage. In both papers, actual data were used from EPRI records and Licensee Event Reports (LER), respectively. It should be noticed that the quantification model used is dependent upon the population and presupposes that the components from a given defined population are subject to similar failure causes.

Fleming [13] developed the β factor method for quantification of common cause failures. Fleming and Raabe [14] have compared this method to the two mentioned above. The objective of their paper was to show the validity of the common cause coupling β -method as compared to the bounding technique of the RSS and the multivariate exponential distribution by Marshall and Olkin.

It can be safely said that the quantification of common cause is still in its developmental phase where Vesely's approach shows a seemingly high potential for future application.

In summary, the trends in the qualitative common cause failure analysis are fully recognized by the industry. They also indicate that well-organized approaches are introduced for effective common cause failure qualification. However, these approaches are not fully computerized, particularly, for the complex technical systems. The quantification of CCF needs further study. Vesely's methodology seems to be a promising one.

6.3 Some Qualitative Aspects of Common Cause Failure Analysis

The common cause failure analysis literature discussed in the previous section emphasizes the significance of common cause failures, and therefore the need for an effective approach to identifying and quantifying CCFs. Since redundancy was first used in an attempt to achieve highly reliable systems, the common cause phenomenon has been inherent in system designs. The concern is that highly-reliable systems are subject to compromise by human error and environmental factors. Potential common cause failures are, most of the time, the result of adding complexity to simple system designs and external undesirable

environments (e.g. earthquakes, ..., etc). They are partly the product of a supersafe philosophy.

A significant common cause event is a secondary cause that is common to all the basic events in one or more minimal cutsets of the system fault tree. The minimal cutset for which the significant cause event is applicable is called a common cause candidate. In addition, if all the components represented by the basic events in that minimal cutset share a "common location", that minimal is a crucial common cause candidate.

Wilson, J.R., et. al. [15, 16] have used a method which is concerned only with locating crucial common cause candidates and, where possible, identify the associated significant common cause events. This method is based on the use of the system fault tree and at the present time is the most widely used method in the industry. In what follows, a description of Wilson's method which is adopted and used in this study for identification of common cause candidates is presented.

To analyze common cause failure one needs to consider only the most significant generic causes in each of four broad categories for each failure event. The categories are as follows:

- (i) mechanical/thermal
- (ii) electrical/radiation
- (iii) chemical miscellaneous
- (iv) special conditions

A suggested list of generic causes for these categories which can be easily updated without modification methodology is given in Tables (6.3.1) through (6.3.4). Using these tables, the analyst chooses those generic causes applicable to his analysis, adds quantifying details (e.g., temperatures over 500°C), and combines causes where desired (e.g., conducting medium, oxidation, and high temperature to represent steam, or impact and vibration to represent earthquake).

In Wilson's method a cutset can be a common cause candidate if all the components implied by the basic events in that cutset share a special condition as described in Table (6.3.1) (e.g. all components in a cutset produced by the same manufacturer), or if all the basic events in the minimal cutset share one or more generic causes described in Tables (6.3.2) through (6.3.4) and all these basic events are located in a common physical location in which the generic cause is applicable. It is clear that in the list of common cause candidates obtained by this method, those cutsets that have two or more components susceptible to a common cause failure are neglected. It is found in this study, however, that the effect of neglecting these types of common cause candidates might occasionally result in serious underestimation of probabilities in the quantification of CCF.

The Wilson method is used in the Idaho National Engineering Laboratory (INEL) in order to develop the code COMCAN [7]. Similarly, the code BACFIRE [1] used the same methodology and is developed by the University of Tennessee. BACFIRE has more capability than the COMCAN code. These codes will be discussed later in this chapter.

For common cause failure analysis by COMCAN or BACFIRE cutsets must be

<u>Symbol</u>	<u>Common Link</u>	<u>Example Situations</u>
E	Energy source	Common drive shaft, same power supply
C	Calibration	Misprinted calibration instructions
I	Installations contractor	Same subcontractor or crew
M	Maintenance	Incorrect procedure, inadequately trained person
O	Operator or operation	Operator disabled or overstressed, faulty operating procedures
P	Proximity	Location of all components of a cut set in one cabinet (common location exposes all of the components to many unspecified common causes)
T	Test procedure	Faulty test procedures which may affect all components normally tested together
N	Energy flow paths	Location in same hydraulic loop, location in same electrical circuit

Table (6.3.2)

MECHANICAL OR THERMAL GENERIC CAUSES

<u>Symbol</u>	<u>Generic Cause</u>	<u>Example Sources</u>
I	Impact	Pipe whip, water hammer, missiles, earthquakes, structural failure
V	Vibration	Machinery in motion, earthquake
P	Pressure	Explosion, out-of-tolerance system changes (pump overspeed, flow blockage)
G	Grit	Airborne dust, metal fragments generated by moving parts with inadequate tolerances
M	Moisture	Condensation, pipe rupture, rainwater
S	Stress	Thermal stress at welds of dissimilar metals, thermal stresses and bending moments caused by high conductivity and density of liquid sodium
T	Temperature	Fire, lightning, welding equipment, cooling system faults, electrical short circuits

ELECTRICAL OR RADIATION GENERIC CAUSES

Symbol	Generic Cause	Example Sources
E	Electromagnetic interference (EMI)	Welding equipment, rotating electrical machinery, lightning, power supplies, transmission lines
R	Radiation damage	Neutron sources, charged particle radiation
M	Conducting medium	Moisture, conductive gases
V	Out-of-tolerance voltage	Power surge
I	Out-of-tolerance current	Short circuit, power surge

Table (6.3.4)

CHEMICAL OR MISCELLANEOUS GENERIC CAUSES

Symbol	Generic Cause	Example Sources
A	Corrosion (acid)	Boric acid from neutron control system, acid used in maintenance for removing rust and cleaning
O	Corrosion (oxidation)	In a water medium or around high temperature metals (for example, filaments)
R	Other chemical reactions	Galvanic corrosion; complex interactions of fuel cladding, water, oxide fuel, and fission products; leaching of carbon from stainless steel by sodium
C	Carbonization	Hydrocarbon (hydraulic fluid, lubricating oils, diesel fuel) in liquid sodium
B	Biological hazards	Poisonous gases, explosions, missiles

[a] Sodium-water and sodium-air reactions have been left out of the table because the resulting failure modes can be represented by other generic causes included in the other tables, e.g., temperature and biological hazards. However, the analyst, for clarity, may expand the table to include sodium reactions.

input to these codes. It would be much more convenient if the user could input only the fault tree. The approach we employ is to use the code MOCUS [17] for this purpose, because MOCUS is a fast and simple program for generating cutsets, and is written in FORTRAN. It is well suited for generating the cutsets needed for the common cause analysis. A combination of MOCUS and COMCAN, or MOCUS and BACFIRE, would therefore provide a FORTRAN package which could be used conveniently to identify potential common cause failures of a system through its fault tree. The MODCUT code which is developed in this study would also provide cutsets very effectively for large fault trees, but it is written in the PL/I language so is not used here. We have chosen MOCUS because it is in FORTRAN, and both BACFIRE and COMCAN codes are also written in FORTRAN. The code BACFIRE is used here as the common cause analysis code, because it is exactly similar to COMCAN but it allows multiple appearance of components in more than one domain [18].

However, we have also used the cutset generator code MODCUT to determine cutsets for use in BACFIRE. The package obtained by coupling MODCUT and BACFIRE codes (i.e. MODCUT BACFIRE) verified the accuracy and effectiveness of the MOCUS-BACFIRE package.

In the next section the two common cause failure analysis packages, MOCUS-BACKFIRE, and MODCUT-BACFIRE are discussed.

6.4 MOCUS-BACFIRE, A Convenient Code Package for the Identification of Common Cause Candidates from a Fault Tree

In sections 6.2 and 6.3, the importance and the significance of common cause failures are discussed. The codes COMCAN and BACFIRE are available

to select from the cutsets of a fault tree, those cutsets that are common cause candidates. In the INEL a methodology has been developed in order to perform a complete CCF analysis [15]. COMCAN and BACFIRE can be effectively used as a part of this methodology. Based on this methodology, the following steps are necessary to perform a common cause analysis:

1. Determine a complete list of component minimal cutsets from the system fault tree.
2. Obtain the common cause failure data for all of the components in the cutset.
3. Search for those cutsets that are common cause failure candidates (see Section 6.3 for the proposed methodology).
4. Select significant common cause events from the common cause candidates.
5. Identify common cause mechanisms which would result in the significant common cause events.

Obviously, for the performance of these steps, maximum use of the computer is necessary. Particularly for Steps 1 and 3 several computer codes are available. However, these codes must be used separately for Steps 1 and 3. And it would be much more convenient if both steps were performed by one and the same code. In this section we will mainly discuss selection and implementation of a code to perform Steps 1 and 3 together.

As discussed in the previous section, to obtain minimal cutsets that are common cause candidates, two codes are available, namely, COMCAN and BACFIRE. The methodology utilized in these codes to obtain common cause candidates has been discussed previously. COMCAN and BACFIRE are similar, but in addition to COMCAN, the BACFIRE Code allows multiple physical locations to be input for the basic events. Therefore, the use of the BACFIRE Code seems to be more appropriate for our particular interests. BACFIRE utilizes minimal cutsets as input. Therefore, it would be more convenient, due to the high population of the cutsets, to couple a cutset generation code with BACFIRE and use the resulting package along with the system fault tree and common cause failure data.

Computer programs for determining the minimal cutsets of a fault tree are discussed in Appendix A. A review of these programs showed that the code MOCUS is a fairly good candidate for our particular case, because MOCUS is written in FORTRAN and the coupling of MOCUS and BACFIRE provides a convenient FORTRAN package for our future practical use. Therefore, the MOCUS code is coupled with BACFIRE and the resulting package is called MOCUS-BACFIRE. An alternative to this package is MODCUT-BACFIRE (next section). But MODCUT is written in PL/I, and future code development programs are under study to incorporate into the MODCUT a fast PL/I procedure to directly perform a common cause failure discrimination process. Therefore, MOCUS has been preferred to be coupled with BACFIRE. However, in the course of our research project, a MODCUT-BACFIRE code is also developed so as to test the MOCUS-BACFIRE package and make the common cause analysis of more complex fault trees possible.

The coupling procedure of MOCUS and BACFIRE is very simple and straightforward. The rows of the matrix in which the cutsets are stored in MOCUS are, one at a time, transferred to the same array that BACFIRE uses to read and store cutsets. Therefore, fault tree and common cause failure data must be first read by MOCUS and BACFIRE, then MOCUS generates all cutsets, and finally BACFIRE reads the cutsets through the array we discussed above and tests the cutsets for a possible common cause candidacy. More details with regard to the programming concepts of coupling MOCUS and BACFIRE are presented in Chapter 7. A user's manual is also provided in Appendix D of this thesis for the MOCUS-BACFIRE Package.

6.5 MODCUT-BACFIRE - An Alternative to MOCUS-BACFIRE

In the analysis of complex systems the determination of the complete list of minimal cutsets becomes a difficult task, and is for highly complex systems nearly impossible. The code MOCUS is limited to small and medium size fault trees. Thus, use of other cutset generating codes becomes necessary for complex fault trees. The MODCUT code is a potential cutset generator for this purpose. However, MODCUT is written in PL/I and confusions might arise by coupling MODCUT and BACFIRE. Hence, for simplifying the task of coupling these two codes it has been decided to use the Conversational Monitor System (CMS) using MIT's 370/168 IBM [18].

The method of coupling is simple and very few changes have been made in both MODCUT and BACFIRE to allow for the use of CMS input and output files. First, MODCUT statements that involve writing the cutsets

have been changed so as to write the cutsets on a defined output file. Second, BACFIRE statements in which the cutsets are read have been changed so as to read the cutsets, in a free form format from the output file created by the MODCUT code.

To perform the coupling procedure a file with the file name MOD-BAC and a file mode EXEC is created [18]. Object modules of the MODCUT and BACFIRE codes are obtained by compiling them separately, and storing them in separate files. In the MOD-BAC EXEC file the commands for the following purposes are stored:

- a. defining the input file to MODCUT,
- b. defining the output file from MODCUT,
- c. executing MODCUT MODULE file,
- d. defining the input files to BACFIRE; and
- e. executing the BACFIRE MODULE file.

The user automatically gets the common cause candidates after he provides input files for MODCUT (i.e. fault tree) and BACFIRE (i.e. common cause failure data), by using only mod-bac command to execute the commands that are stored in the MODBAC EXEC file. It is clear that no extra work need be performed for obtaining common cause candidates directly from the fault tree, because the provisions of input data for MODCUT and BACFIRE is sufficient to run the MODCUT-BACFIRE Code.

MODCUT-BACFIRE has been compared to MOCUS-BACFIRE and both have resulted in identical common cause candidates for the same input fault tree and the same common cause failure data. For a typical fault the having 30 components and 22 gates the MODCUT-BACFIRE package is 1.29 times

faster than the MOCUS-BACFIRE for identical susceptibility data. However, the advantage over the latter is that it is entirely written in FORTRAN IV and its use is more convenient, especially for users who have access only to FORTRAN compilers. One of the important features of the MODCUT-BACFIRE is its advantage of writing the cutsets on a file provided on auxiliary disks. Because a high amount of storage can be provided in this fashion to store as many minimal cutsets as exist in a complex fault tree.

As discussed earlier in Chapter 5, the MODCUT code is constructed for discrimination processes and a few methods of discrimination based on common cause failures have been carefully studied for future developments in the context of the MODCUT code. These methods of discrimination provide the basis for the quantification methodologies which will be discussed in the following sections. The presentation of potential methods of common cause discrimination for use in MODCUT is beyond the scope of this thesis. Future reports related to the development of the MODCUT code will discuss those issues.

6.6 Quantification of Common Cause Failures Using the Cutsets of a Fault Tree

Quantification of common causes has an important role in reliability engineering and risk assessment. Recent publications on this subject have emphasized the importance of considering potential common cause failures in estimating the probability of failure of complex redundant systems [19]. In most of the reliability calculations in the past, the probability of system failure was seriously underestimated by ignoring common cause

failure contributions to the failure of systems. A number of methods have been proposed to quantify common cause effects [20, 21, 11, 6].

In this section, by using the Markovian approach, the effect of dependancies among components of a cutset due to a common cause failure is quantified. The Markovian approach is selected because equivalent rates may be assigned to different states of system failure.

6.6.1 Quantification Methodology using a Set of Common Cause Candidates

As has been discussed previously, a MOCUS-BACFIRE package is developed to identify cutsets and potential common cause candidates. However, identification of the common cause cutsets would not provide a numerical basis for judgement on the likelihood of these common cause failures, and reduction in the availability of systems due to the presence of the common cause failures.

In what follows, a method will be demonstrated for common cause quantification, based on the cutsets of the tree which are identified as common cause candidates as well as those cutsets that are not identified as common cause candidates (i.e. these indicate only random failure). This method may be applied to the general form of common cause candidates in which two or more members of the cutsets are identified as common cause candidates from cutsets in which all of the cutsets are dependent due to a particular cause. Therefore, this method may be applied to a more general form of common cause candidates.

Suppose cutset $C_i = (C_1, C_2, \dots, C_n)$ consists of n members, and a number of these members, say n , are susceptible to cause i , and located in the same susceptibility domain, or share a similar special condition

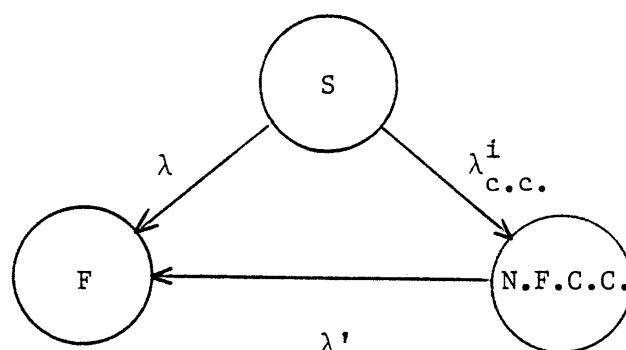
(e.g., identical manufacturer). There would be two ways in which this cutset might contribute to a system failure. One is that all members of the cutset would simultaneously fail due to a "random failure" (cause); and next if two or more of the members of this cutset fail due to a "common cause failure" and the rest of them, if any, fail due to a random failure.

The Markovian Model presented in Figure (6.6.1) shows failure of the system due to the failure of a single cutset. It is assumed that a total of three states may represent the modes of system operation due to a particular cutset, say α . First, state S represents the mode in which cutset α does not cause any system failure and that all of the members of the cutset are functioning. Second, State N.F.C.C. represents the mode in which some or all of the members of the cutset α are failed due to a common cause failure. Third, state F represents the mode in which all of the members of the cutset α are failed due to a random cause failure, common cause failure, or a combination of these. It is further assumed that transitions between the states are exponentially distributed. Thus, this verifies the use of Markovian approach. Finally, to simplify the model, the possibility of repair is neglected. The values of λ and λ' may be calculated by using Table (2.8.1) of this thesis from the following equations (see also reference [22]);

$$\lambda(t) = \frac{1}{t} \ln \left[1 - \prod_{i=1}^n (1 - e^{-\lambda_i t}) \right] \quad (6.6.1)$$

$$\lambda'(t) = \frac{1}{t} \ln \left[1 - \prod_{i=1}^{n-m} (1 - e^{-\lambda_i t}) \right] \quad (6.6.2)$$

Figure(6.6.1) Simplified Markovian Model



$\lambda_{c.c.}^i$ = common cause failure rate due to cause i which has resulted in simultaneous failure of two or more components in the cutset

λ = average random cause failure rate of the whole cutset

λ' = average random cause failure of those components in the cutset that are not involved in common cause failure

S = success state (UP)

F = failed state (DOWN)

N.F.C.C. = Non-Failed state due to Common Cause failure

where λ_i is the failure rate of individual components in the cutset, n is the total number of components in the cutset, and m is the number of components failed due to a common cause. By using average unavailabilities over a mission length T_M , constant values of λ and λ' may be obtained from equations (6.6.1) and (6.6.2) [23].

$$\lambda = (1/2 T_M)^{n-1} \prod_{i=1}^n (\lambda_i) \quad (6.6.3)$$

$$\lambda' = (1/2 T_M)^{n-m-1} \prod_{i=1}^{n-m} (\lambda_i) \quad (6.6.4)$$

By using figure (6.6.1) the differential equations representing the Markovian Model can be written as follows:

$$\frac{dP_s^i(t)}{dt} = - (\lambda_{C.C.}^i + \lambda) P_s^i(t) \quad (6.6.5)$$

$$\frac{dP_{N.F.C.C.}^i(t)}{dt} = \lambda_{C.C.}^i P_s^i(t) - \lambda' P_{N.F.C.C.}^i(t) \quad (6.6.6)$$

$$\frac{dP_f^i(t)}{dt} = \lambda' P_{N.F.C.C.}^i(t) + \lambda P_s^i(t) \quad (6.6.7)$$

where parameters $\lambda_{C.C.}^i$, λ' , and λ are defined in Figure (6.6.1) and P_A^i indicates probability associated with cause i for state A , and at time t . Differential equations presented above may be written in the following general form:

$$\frac{d\underline{P}(t)}{dt} = \underline{P}(t) \quad A \quad (6.6.8)$$

where A and $\underline{P}(t)$ are matrices. For example, $\underline{P}(t) = (P_S^i(t), P_{N.F.C.C.}^i(t), P_F^i(t))$. The initial conditions for these differential equations are as follows:

$$\underline{P}(0) = (1, 0, 0) \quad (6.6.9)$$

To obtain the exact probability that the system would go into State (N.F.C.C.) must be multiplied by P_O^i , where P_O^i is the probability that cause i would occur (e.g. fire, earthquake, ..., etc.). The solutions to the set of differential equations (6.6.5) through (6.6.7) with the initial conditions given in Equation (6.6.9) are as follows:

$$P_S^i(t) = e^{-(\lambda_{C.C.}^i + \lambda)t} \quad (6.6.10)$$

$$P_{N.F.C.C.}^i(t) = \frac{P_O^i \lambda_{C.C.}^i}{\lambda_{C.C.}^i + \lambda - \lambda'} [e^{-\lambda' t} - e^{-(\lambda_{C.C.}^i + \lambda)t}] \quad (6.6.11)$$

$$P_F^i(t) = \frac{P_O^i \lambda_{C.C.}^i}{\lambda_{C.C.}^i + \lambda - \lambda'} [1 - e^{-\lambda' t}] - \frac{P_O^i \lambda_{C.C.}^i \lambda' - \lambda(\lambda_{C.C.}^i + \lambda - \lambda')}{(\lambda_{C.C.}^i + \lambda)(\lambda_{C.C.}^i + \lambda - \lambda')} [1 - e^{-(\lambda_{C.C.}^i + \lambda)t}] \quad (6.6.12)$$

For satisfying these equations the following condition must be satisfied:

$$(\lambda_{C.C.}^i + \lambda) > \lambda' \quad (6.6.13)$$

The reason is that if λ' is greater than $(\lambda_{C.C.}^i + \lambda)$ then the rate at which a system fails due to random failures is greater than that of common cause failures and, therefore, before a system could fail due to a common cause it has already failed due to a random cause in which the state (N.F.C.C.) would become meaningless. Equation (6.6.12) can be reduced to the system unavailability for non-repairable components in which the system is not subject to common cause. In this case $\lambda_{C.C.}^i = 0$, and equation (6.6.12) reduced to the form given in equation (6.6.14) which can also be obtained by other methods as well, namely

$$P_F(t) = (1 - e^{-\lambda t}) \quad (6.6.14)$$

For calculating the total unavailability of the engineering systems through the cutset of its fault tree, it would be possible to use equation (6.6.12) for each cutset, and to form the union of all causes (i), and finally all cutsets (α) would result in the total unavailability of the system. Hence, if $\alpha P_F^i(t)$ is the unavailability of the system due to the cutsets α and cause i at time t, the total system unavailability at time t would be as follows

$$P_F(t) = \sum_{\alpha, i} \alpha P_F^i(t) \quad (6.6.15)$$

For small $\alpha P_F^i(t)$, equation (6.6.15) reduces to the form given below:

$$P_F(t) \approx \sum_{\alpha} \sum_i \alpha P_F^i(t) \quad (6.6.16)$$

Availability of the system would thus be:

$$P_A(t) = 1 - P_F(t)$$

It should be noted here that for the special case that all of the members of a cutset are identified as common cause candidate due to a single cause i , then the (N.F.C.C.) state does not exist and equation (6.6.12) for this special case can be written as follows

$$P_F^i(t) = (1 - P_O^i)(1 - e^{-\lambda t}) + P_O^i(1 - e^{-\lambda_{C.C.}^i t}) \quad (6.6.17)$$

Equation (6.6.12) can be simplified for some of the special cases of common cause failures. For example, in the case of a rare event such as fire, the probability of P_O^{FIRE} is small. However, given that fire has occurred there would be a very high probability that a set of components identified in a cutset to be susceptible to fire would simultaneously fail (i.e. $e^{-\lambda_{C.C.} t} \approx 0$). If it happens that two or more members of a cutset were involved in a fire with the probability of P_O^{FIRE} , equation (6.6.12) can be rewritten in the following form.

$$P_F^{\text{Fire}} = P_O^{\text{Fire}} \quad (6.6.19)$$

which is a logical result. Therefore, for a system with a high probability of fire in its vicinity, components in the cutsets of the system should not be placed within the same thermal barrier.

Another special case would be the situation where there are common links resulting in dependence among components. For instance, having the same power supply, same test procedure, same maintenance, same manufacturers, ..., etc. These types of common causes have a high probability for the initiation of the cause (usually in the range of 0.01 to 1). In this case equation (6.6.12) would become more simple. Specifically, most of the time the value of λ is, particularly, much smaller than $\lambda_{C.C.}$ and thus, equation (6.6.12) can be written as follows:

$$P_F^i(t) = \frac{P_O^i \lambda_{C.C.}^i}{\lambda_{C.C.}^i - \lambda'} [1 - e^{-\lambda' t}] - \frac{P_O^i \lambda'}{\lambda_{C.C.}^i - \lambda'} [1 - e^{-\lambda_{C.C.}^i t}] \quad (6.6.20)$$

Equation (6.6.20) holds for most of the practical studies concerning the quantification of common causes.

Another method for quantification of common causes through fault tree cutsets suggests the direct use of Equation (6.6.14) for each of the members of the cutset. For example, if h members of a cutset out of a total of n members, are identified as common cause contributors, then the system unavailability due to this cutset would be:

$$\begin{aligned}
 \left[\begin{array}{c} \text{Unavailability} \\ \text{due to the} \\ \text{cutset } \alpha \end{array} \right] &= \left[\begin{array}{c} \text{Unavailability due to} \\ \text{random cause given no} \\ \text{cause for a common} \\ \text{cause failure exists} \end{array} \right] + \left[\begin{array}{c} \text{Probability that} \\ \text{cause } i \text{ exists} \end{array} \right] \times \\
 &\left[\begin{array}{c} \text{Probability that cause} \\ \text{ } i \text{ fails } h \text{ members of} \\ \text{the cutset } \alpha \end{array} \right] \times \left[\begin{array}{c} \text{Probability that the rest of} \\ \text{the members of cutset } \alpha \\ \text{(i.e. } n-h \text{) fail due to a} \\ \text{random cause} \end{array} \right]
 \end{aligned}$$

This can be mathematically written as follows:

$$P_F^i(t) = (1 - P_O^i) \prod_{j=1}^m (1 - e^{-\lambda_j t}) + P_O^i (1 - e^{-\lambda_{C.C.}^i t}) \prod_{j=1}^m (1 - e^{-\lambda_j t}) \quad (6.6.21)$$

where $m = n-h$

λ_j = j th cutset member random failure rate

Average system unavailability can be obtained by using the average unavailability of each member (see Table 4.3.1):

$$P_F^i(t) = (1 - P_O^i) \left(\prod_{j=1}^n \lambda_j \right) (1/2 T_M)^n + P_O^i \lambda_{C.C.}^i \left(\prod_{j=1}^m \lambda_j \right) (1/2 T_M)^{m+1} \quad (6.6.22)$$

6.6.2 A Method for Calculating $\lambda_{C.C.}$

The difficulty of using Equation (6.6.14) or (6.6.21) for quantifying common causes is how to obtain an accurate and reliable estimate for $\lambda_{C.C.}$'s. Vesely [11] has used the method of statistical sampling from the past failure data.

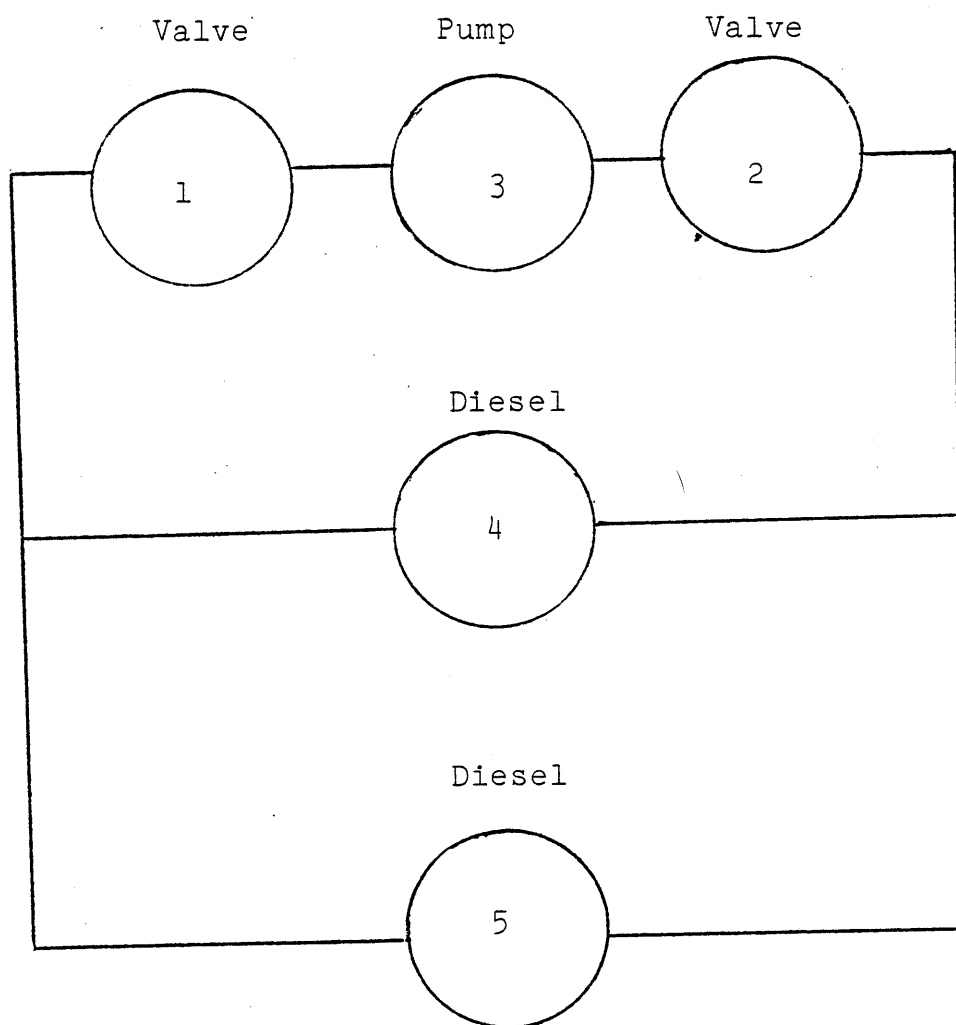
There are two basic models that Vesely has used to evaluate common cause failure rates. The First Model, the constant failure rate (CF) is a one-parameter model, and the second model, the Binomial Failure Rate (BFR), is a two parameter model. The basic assumption of the CFR model is that common cause failures are equally likely to affect any specific component set in a population. In the BFR model, the common cause occurrence rate can be dependent upon the number of components in a specific component set. Either one of the CFR or BFR models would provide a basis for calculating $\lambda_{C.C.}$. Particularly, the CFR seems more simple and still accurate for calculating $\lambda_{C.C.}$'s. Johnson and Vesely have used the BFR model to obtain the probability of valve leakage due to common causes [12]. They used the Licensee Event Report (LER) to obtain failure rates due to common cause. (LER) is computer-based data file information extracted from licensee reports on incidents reportable to the N.R.C. The LER also identifies the type of components involved and the system containing the component. Failure rates due to common causes ($\lambda_{C.C.}$) could therefore be obtained from (LER) by a sampling process. For complete and accurate values of $\lambda_{C.C.}$'s more study and failure data would be required and the N.R.C. is pursuing projects whose goal is to collect these data.

6.6.3 Common Cause Analysis of the Aux Feed System (AFS)

A simplified version of the AUX feed system is shown in Figure (6.6.2) and is described in detail in [6]. As can be seen from Figure (6.6.2) the system consists of two diesel generators in parallel with a pump and two valves. The pump and valves are in series.

Figure 6.6.2

Block Diagram of the Aux-Feed System



It is intended to calculate the total unavailability (common cause and random failure) of the AFS for the First cycle of its operation before a test will be performed (i.e. $T_M = 1440$ hrs.). By assuming that the test duration is negligible compared to the test interval (test duration is about one hour), all test intervals are equal, and at the end of the test operation the AFS becomes totally available, it transpires that the total unavailability for all test intervals would be equal. Therefore, analysis of only one test interval is sufficient. Further, in order to simplify the problem it is assumed that only fire, and the effect of common test procedures can result in a significant common cause contribution. The rest of the common cause failures will be neglected (a crude assumption).

It is known that some of the test operations are not perfect and that there is always a possibility that the system can fail due to an inadequate test procedure. Conservatively, it is assumed that one out of ten tests is non-perfect (i.e., $P_O^{\text{TEST}} = 0.1$). A failure due to this inefficiency in the testing procedure could result in a common cause between the tested components. Further, it is known that in the AFS the diesel generators are tested. A review of the operational records of diesel generators has shown that in a two-year period only one common cause failure of both diesel generators was due to faulty testing.

$$\lambda_{\text{C.C.}}^{\text{TEST}} = \frac{1}{2(365)(24)} = 5.7 \times 10^{-5} \text{ hr}^{-1}$$

Probability of fire is fairly low and it is known that it is about $P_o^{\text{FIRE}} = 0.5 \times 10^{-4}$. However, it would be logical to assume that if fire would occur with a probability of 0.5×10^{-4} , it would be almost certain that the components involved would fail and thus, with some conservatism equation (6.6.18) could be used for the analysis of fire based on common cause failures.

Figure (6.6.3) and Table (6.6.1) present the fault tree for the AFS and the failure data for the components of the fault tree, respectively. It is known that the two diesel generators are located in the same room and share the same thermal barriers.

Based on the above assumptions, the following cutsets are identified along with their common cause candidates:

1. D.G.1 D.G.2 VALVE 1

Common Causes

- a. FIRE (D.G.1, D.G.2)
- b. TEST (D.G.1, D.G.2)

2. D.G.1 D.G.2 PUMP 1

Common Causes

- a. FIRE (D.G.1, D.G.2)
- b. TEST (D.G.1, D.G.2)

3. D.G.1 D.G.2 VALVE 2

Common Causes

- a. FIRE (D.G.1, D.G.2)
- b. TEST (D.G.1, D.G.2)

Figure(6.6.3)

Fault Tree for the AFS

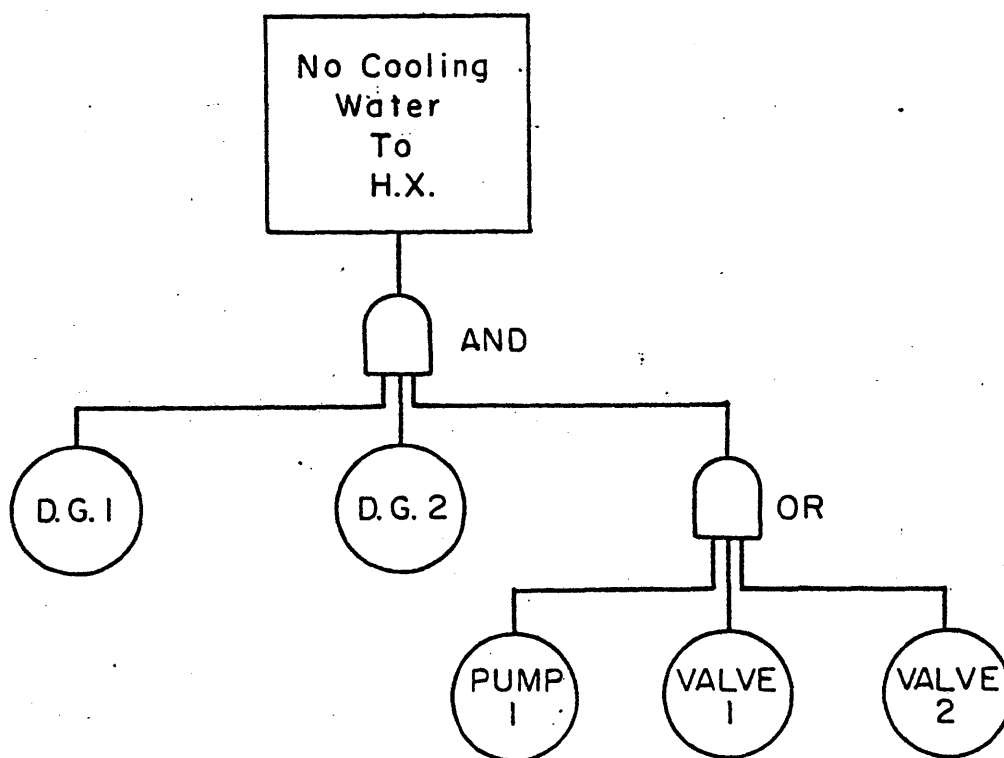


TABLE (6.6.1)

FAILURE DATA FOR THE AFS COMPONENTS

<u>Component Number</u>	<u>Component Name</u>	<u>Failure Rate (1 x 10⁻⁶)λ</u>	<u>Test Interval (days)</u>	<u>CCF Rate Test (1 x 10⁻⁶)λ</u> C.C.
1	D.G. 1	42	60	57
2	D.G. 2	42	60	57
3	PUMP 1	3	60	57
4	VALVE 1	0.3	0	-
5	VALVE 2	0.3	0	-

$$P_o^{\text{FIRE}} = 0.5 \times 10^{-4}$$

$$P_o^{\text{TEST}} = 0.1$$

For the first cutset the variables in equation (6.3.14) are as follows:

$$\lambda_{C.C.}^{TEST} = 5.7 \times 10^{-5} \text{ hr}^{-1}$$

$$P_o^{TEST} = 0.1$$

$$\lambda = (\lambda_{D.G.1} \times \lambda_{D.G.2} \times \lambda_{VALVE 1}) (1/2 \times 1440)^2 = 2.7 \times 10^{-10} \text{ hr}^{-1}$$

$$\lambda' = \lambda_{VALVE 1} = 0.3 \times 10^{-6} \text{ hr}^{-1}$$

$$\lambda_{C.C.}^{FIRE} \gg \lambda$$

$$P_o^{FIRE} = 0.5 \times 10^{-4}$$

For the second cutset one gets:

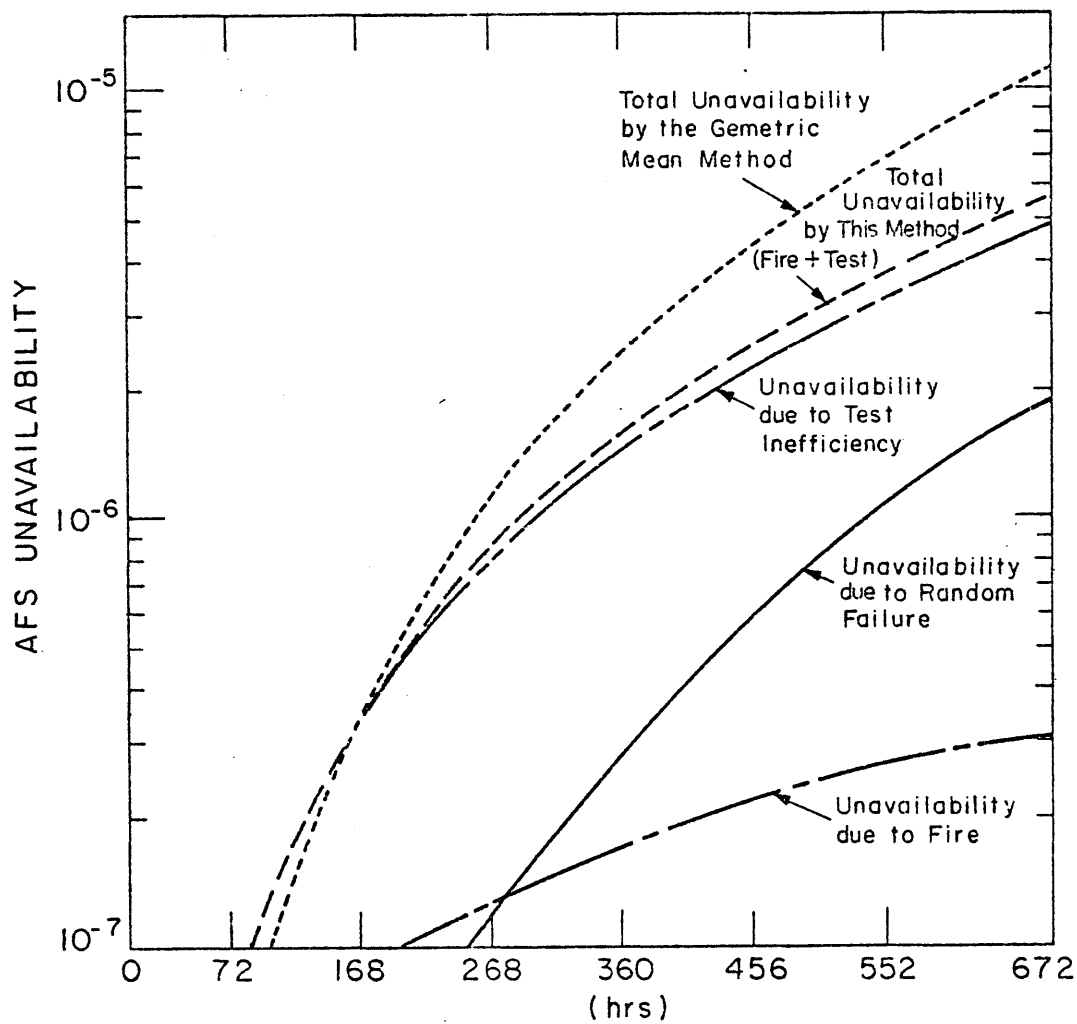
$$\lambda = (\lambda_{D.G.1} \times \lambda_{D.G.2} \times \lambda_{PUMP 1}) (1/2 \times 1440)^2 = 2.7 \times 10^{-9} \text{ hr}^{-1}$$

$$\lambda' = \lambda_{PUMP} = 3 \times 10^{-6} \text{ hr}^{-1}$$

$$\lambda_{C.C.}^{TEST} = 5.7 \times 10^{-5} \text{ hr}^{-1}$$

The third cutset is similar to the first one. The results for different time steps are shown in Figure (6.6.4) for the two causes, and the total unavailability due to both random and common cause failures. The result obtain for the system unavailability by using

Figure(6.6.4) AFS Unavailability due to Random and Common Cause Failures



the PL-MODT code for the Class 2 component type shown in Chapter 8 (i.e. non-repairable- time dependent), without considering the common cause effect, is compared with the total unavailability calculated for both random and common cause failures. The differences between the two unavailabilities are different at different instants of time. This difference is on the order of 2 to 100 larger, if the effect of the common cause failure is taken into account. Particularly, larger differences between the random and common cause failure unavailability contribution to the top event exists in the early hours of system operation after the test has been performed. This difference in unavailabilities would, therefore, prove the importance of considering the system common cause failures in reliability and risk assessment studies. To verify the results obtained for the unavailability of the AFS, the Geometric Mean Method is used to determine the unavailability due to common cause. The result are compared with the method discussed in this section (see Figure 6.6.4). Since the Geometric Mean Method is a rather rough estimate of the common cause failure probability, it is interesting that the two methods give such close agreement.

REFERENCES

- [1] Fussell, J. B., et. al., "MOCUS - A Computer Program to Obtain Minimal Sets from Fault Trees", Aerojet Nuclear Co. , ANCR-1156 (August, 1974)
- [2] Cate, C. L. and J. B. Fussell, "BACFIRE - A Computer Code for Common Cause Failure Analysis", University of Tennessee, Knoxville, Tennessee, May, 1977
- [3] Sidall, E., "Reliable Reactor Protection", Nucleonics 1616, June, 1957
- [4] Epler, P., "Dangers in Safety Systems", Oak Ridge National Lab., Tenn. pp. 51-55, October 1961
- [5] Epler, P., "Common Mode Failure Considerations in the Design of Systems for Protection and Control", Nuclear Safety 10(1), January-February 1969
- [6] "Reactor Safety Study", WASH-1400, U. S. Nuclear Regulatory Commission, October 1975
- [7] Burdick, G. R., "COMCAN - A Computer Code for Common Cause Analysis", IEEE Trans. Reliability R-26 (1977) 2, 100-102
- [8] Wagner, D. P., "A Procedure for Qualitative Common Cause Failure Analysis of Complex Systems", Master's Thesis, University of Tennessee, August 1977
- [9] Wagner, D. P., C. L. Cate and J. B. Fussell, "Common Cause Failure Analysis Methodology for Complex Systems", Nuclear Systems Reliability Engineering and Risk Assessment, SIAM, 1977
- [10] "Risk Assessment Review Group Report to the USNRC", H. W. Lewis, et. al., NUREG/CR-0400, 1978
- [11] Vesely, W. E., "Estimating Common Cause Failure Probabilities in Reliability and Risk Analysis: Marshal-Olkin Specializations", Proceedings, International Conference on Nuclear Systems Reliability Engineering and Risk Assessment, Gatinsburg, Tenn., June 1977
- [12] Johnson, J. W. and W. E. Vesely, "Common Mode Analysis of Valve Leakages", Probabilistic Analysis of Nuclear Reactor Safety Topical Meeting, May 1978, Los Angeles
- [13] Fleming, K. N., "A Reliability Model for Common Mode Failures in Redundant Safety Systems", Proceedings of the Sixth Annual Pittsburgh Conference on Modeling and Simulation, April 24, 1975

- [14] Fleming, K. W. and P. Raabe, "A Comparison of Three Methods for Quantitative Analysis of Common Cause Failures", Probabilistic Analysis of Nuclear Reactor Safety Topical Meeting, May 1978, Los Angeles
- [15] Burdick, G. R., N. H. Marshal and J. R. Wilson, "COMCAN - A Computer Program for Common Cause Analysis", Aerojet Nuclear Company, ANCR-1314, May 1976
- [16] Wilson, J. R., "Selecting the 'Right' Common Cause Analysis Techniques", ANS Transaction, 27, November 1977, pp. 629-631
- [17] Fussell, J. B. and W. E. Vesely, "A New Methodology for Obtaining Cutsets for Fault Trees", Trans. ANS, 15, p. 262, 1972
- [18] "CMS AT IPS: A Primer", VM/370 Series, VM-1, MIT-IPS, April 1979
- [19] Rasmussen, D., G.R.Burdick, and J.R.Wilson, "Common Cause Failure Analysis Techniques: A Review and Comparative Evaluation", EG&G, TREE-1349, Sept. 1979
- [20] Apostolakis, G. E., "The effect of a certain class of potential common mode failures on the reliability of redundant systems", Nuclear Engineering and Design, Vol. 36, No. 1, January 1976
- [21] "HTGR Accident Initiation and Progression Analysis Status Report", Vols. I-VII, January 1976; Vol. VIII, January 1977, GA-A13617 (UC-77)
- [22] Modarres, M. and L. Wolf, "PL-MODT and PL-MODMC, Two Codes for Reliability and Availability Analysis of Complex Technical Systems Using the Fault Tree Modularization Technique", MITNE-222, November 1978
- [23] Vesely, W. E. and F. F. Goldberg, "FRANTIC: A Computer Code for Time-Dependent Unavailability Analysis", NUREG-0193, October 1975

CHAPTER 7

DESCRIPTION OF COMPUTER PROGRAMS
DEVELOPED IN THIS WORK7.1 Introduction

As noted previously, several computational advantages will be obtained by using the modular representation to analyze fault trees [1]. PL/I language was used to develop the computer code PL-MOD because several advantages can be obtained by using the PL/I language [2,3] namely the ability to use:

- (1) structure variables (i.e. a hierarchial collection of related data items of different types) to represent the nodes and modules of the fault tree;
- (2) pointer, based and controlled variables to provide the ability to dynamically allocate blocks of core storage to link those blocks together into a structure, and to store and to retrieve data from the blocks;
- (3) bit-string variables, to represent and treat modular and simple cutsets.

In this chapter the extended versions of the PL-MOD code (i.e. PL-MODT, PL-MODMC and MODCUT) and the code developments in the area of common cause analysis (i.e. MOCUS-BACFIRE, and MODCUT-BACFIRE) are discussed. The PL-MOD Code is the backbone of the PL-MODT, PL-MODMC and MODCUT codes, and a detailed discussion of organization and procedures used in the PL-MOD is presented in reference [1]. Important

procedures used in the PL-MODT, PL-MODMC, MODCUT, MOCUS-BACFIRE and MODCUT-BACFIRE are discussed in this chapter in somewhat less detail compared to that presented for the PL-MOD code in [1]. The reader who is interested in those details is referred to this reference.

7.2 Organization of the Procedures used in PL-MODT

The original PL-MOD code handles only Class 1 components, and therefore it is only able to evaluate time-independent components. The other three classes of components have been incorporated into the PL-MOD code, and thus the extended version is called PL-MODT, which also comprises all of the other features of the PL-MOD code. The incorporation of Class 2, 3 and 4 components is performed by adding 3 internal procedures to the external procedure NUMERO, and changing some of the main routine statements in the NUMERO and EXPECT procedures to allow more flexibility to the PL-MODT code. The NUMERO and EXPECT procedures are discussed in [1] in detail. Here only the changes in the main routines of these procedures are discussed.

The IMPORTANCE procedure in the NUMERO is also changed to allow fault tree reduction based on the Vesely-Fussell importance measure. A full description of methodologies and mathematical concepts used in PL-MODT have been presented in Chapters 2 and 3 of this thesis. The following procedures are added to NUMERO in order to account for the time dependent classes of components:

- (a) Procedure SHORE to evaluate Class 2 component type (i.e. non-repairable).
- (b) Procedure CECA to evaluate Class 3 component type (i.e. repairable monitored components).
- (c) Procedure SHR to evaluate Class 4 component type (i.e. periodically tested components) and to account for a combination of Class 1, 2, 3 and 4 components in a tree.

Also, the following extensions have been added to allow more flexibility for the calculation of unavailabilities.

- (a) Further developments in the context of the old NUMERO procedure to enable it to calculate average unavailability over a period of system operation for Classes 2, 3 and 4 or combinations of them. In addition, internal procedures PLUS and MINUP were modified to allow the code treatment of unavailabilities which are close to one (e.g. during the test process of a periodically tested component).
- (b) Further developments in the IMPORTANCE internal procedure to allow fault tree reductions based on the Vesely-Fussell importance measures.

7.2.1 Development of the SHORE Procedure

The SHORE procedure is designed to evaluate fault trees consisting of only non-repairable components (Class 2 components). In the SHORE procedure, the value of the unavailability for simple AND modules can be calculated for different time steps (i.e. $DEL9T(1,J)$) by using equations (2.8.2) and (2.3.2) up to a time TIE where $TIE = 0.1/\lambda$ and λ is the largest failure rate of the corresponding components in the module. After the time TIE , the components should be considered individually and thus only equation (2.3.2) can be used.

For the simple OR modules, the unavailability can be calculated by using equations (2.8.3) and (2.3.2). No limitation exists for the OR module and hence equations (2.8.2) and (2.3.2) can be used for the entire operational period.

The total number of time steps in each time interval mesh is calculated as follows:

$$TIE(1,I) = AUN(1,I)/DEL9T(1,I);$$

where $AUN(1,I)$ is the time span in which the time step $DEL9T(1,I)$ is applied and $DEL9T(1,I)$ is the time step for the I -th mesh interval.

At each $TIE(1,I)$ time step, first the unavailability values are calculated for the various components of the fault tree using equation (2.3.2). Next, these unavailabilities are assigned to the arrays $STATE(1,I)$ and $STATD(1,I)$ for free and replicated components, respectively. Finally, the procedures `EXPECT` and `IMPORTANCE` are called respectively

to calculate the top event and the unavailability of higher order modules as well as the Vesely-Fussell importance measures. The same procedure is applied for all other time interval meshes [AUN(1,1), AUN(1,2), ..., AUN(1,N)]. At the end, unavailabilities are added to evaluate the mean unavailability over the period of $T = \sum_{I=1}^N \text{AUN}(1,I)$.

It should be noted that after the first time step where procedure SHORE is called, some of the variables that are calculated remain unchanged for other time steps. Therefore, after the first call, the procedure SHORE will be called starting from the entry procedure DOBA to avoid recalculation of some already calculated variables.

The procedure SHORE based on equation (2.3.2) is shown in Table (7.2.1)

7.2.2 Development of the CECA Procedure

The CECA procedure is designed to evaluate fault trees consisting of monitored components (Class 3 components). Similar to the SHORE procedure, different time interval meshes along with their corresponding time steps are used to evaluate the unavailability of components, modules, and the whole system for each time step. The time step interval and its corresponding time step length are stored in the allocated arrays AUN(1,I) and DEL9T(1,I), respectively. Four other arrays, STATS, STATT, STATTE, and STATED, are allocated to store the failure and repair rates of free and replicated components.

At the beginning of the system operation, the approximations discussed in Section 2.8 of this thesis are not applicable due to the small unavailability values and, therefore, unavailability of replicated

Table (7.2.1) Procedure SHORE

SHORE: PRCC:	PLM02020
P=DEL;	PLM02030
PUT EDIT('THIS IS A TIME DEPENDENT UNAILABILITY CALCULATION',	PLM02040
'FOR ONLY NONREPAIRABLE COMPONENTS')	PLM02050
(SKIP(6),X(12),A(49),X(1),A(33));	PLM02060
GET LIST(MCH);	PLM02070
ALLOCATE AUN;	PLM02080
ALLOCATE DELST;	PLM02090
DO I=1 TO MOH;	PLM02100
GET LIST(DELST(1,I),AUN(1,I));	PLM02110
END;	PLM02120
GET LIST(FUN);	PLM02130
PUT EDIT('NUM FREE EVENT INPUTS=',FUN)	PLM02140
(SKIP(2),A(22),F(5));	PLM02150
GET LIST(DUN);	PLM02160
PUT EDIT('NUM REPLICATED EVENT INPUTS=',DUN)	PLM02170
(SKIP(2),A(28),F(5));	PLM02180
ALLOCATE STATS;	PLM02190
ALLOCATE STATE;	PLM02200
ALLOCATE STAT;	PLM02210
ALLOCATE STATE;	PLM02220
DO I=1 TO FUN;	PLM02230
GET LIST(I,STAT(1,I));	PLM02240
END;	PLM02250
DO I=1 TO DUN;	PLM02260
GET LIST(I,STATS(1,I));	PLM02270
END;	PLM02280
TIE=TIE+DELST(1,1);	PLM02290
DCBA: 2NTRY:	PLM02300
PUT EDIT('FREE INPUT','RELIABILITY')	PLM02310
(SKIP(2),X(2),A(10),X(3),A(11));	PLM02320
DO I=1 TO FUN;	PLM02330
STATE(1,I)=1-EXP(-STAT(1,I)*TIE);	PLM02340
PUT EDIT(I,STATE(1,I))(SKIP(2),F(12),F(18,6));	PLM02350
END;	PLM02360
PUT EDIT('DEP INPUT','RELIABILITY')	PLM02370
(SKIP(2),X(3),A(9),X(3),A(11));	PLM02380
DO I=1 TO DUN;	PLM02390
STATD(1,I)=1-EXP(-STATS(1,I)*TIE);	PLM02400
PUT EDIT(I,STATD(1,I))(SKIP(2),F(12),F(18,6));	PLM02410
END;	PLM02420
END SHORE;	PLM02430

and free components must be calculated directly by using equation (2.4.9). However, for larger times where the components approach their asymptotic values (usually when $t \geq 3$), the approximate equations summarized in Table (2.8.1) can be used.

Unavailability values calculated at each time step will be assigned to the STATE(1,I) and STATD(1,I) arrays. After calling the internal procedures EXPECT and IMPORTANCE, the top event and the unavailability of higher order modules and the importance will be calculated as well as simple modules and component unavailabilities. The same procedure will be repeated for all other time steps. Similar to the SHORE procedure for the second time that the CECA procedure is called, the calculation of the unavailability automatically starts from the SOBA entry procedure, in order to avoid unnecessary recalculations. Table (7.2.2) summarizes the statements used in the CECA procedure to evaluate the unavailability of a system composed of only monitored components.

7.2.3 Development of the SHR Procedure

The SHR procedure is developed to evaluate fault trees composed of only periodically tested components (Class 4), or a combination of the four classes of components. Thus, the SHR procedure is a general routine which can be applied for many types of fault trees composed of time-dependent components. In the SHR procedure the use of different mesh intervals plays an important role for the accurate determination of the top event unavailability. For example, during the inspection period, very small time step meshes should be used in order to calculate a detailed behavior of the system unavailability, which usually results in large

Table (7.2.2) Procedure CECA

CECA: PROC:	PLM02440
F=DEL;	PLM02450
PUT EDIT('THIS IS A TIME DEPENDENT UNAVAILABILITY',	PLM02460
'CALCULATION FOR REPAIRABLE COMPONENTS')	PLM02470
(SKIP(6),X(10),A(39),X(1),A(37));	PLM02480
GET LIST(MCH);	PLM02490
ALLOCATE DEL9T;	PLM02500
ALLOCATE AUN;	PLM02510
DC I=1 TO MOH;	PLM02520
GET LIST(DEL9T(1,I),AUN(1,I));	PLM02530
END;	PLM02540
GET LIST(FUN);	PLM02550
PUT EDIT('NUM FREE EVENT INPUTS=',FUN) (SKIP(2),A(22),F(5));	PLM02560
GET LIST(IUN);	PLM02570
PUT EDIT('NUM REPLICATED EVENT INPUTS=',DUN)	PLM02580
(SKIP(2),A(28),F(5));	PLM02590
ALLOCATE STATS;	PLM02600
ALLOCATE STATE;	PLM02610
ALLOCATE STATI;	PLM02620
ALLOCATE STATD;	PLM02630
ALLOCATE STATTE;	PLM02640
ALLOCATE STATED;	PLM02650
TIE=TIE+1;	PLM02660
TTMC=TTMC+DEL9T(1,1);	PLM02670
DC I=1 TO FUN;	PLM02680
GET LIST(I,STATI(1,I),STATTE(1,I));	PLM02690
END;	PLM02700
DO I=1 TO DUN;	PLM02710
GET LIST(I,STATS(1,I),STATED(1,I));	PLM02720
END;	PLM02730
SCBA: ENTRY;	PLM02740
PUT EDIT('POINT NUMBER=',TIE) (SKIP(2),X(2),A(13),F(15));	PLM02750
PUT EDIT('FREE INPUT','RELIABILITY')	PLM02760
(SKIP(2),X(2),A(10),X(1),A(11));	PLM02770
DO I=1 TO FUN;	PLM02780
STATE(1,I)=(STATI(1,I)/(STATI(1,I)+STATTE(1,I)))*(1-EXP(-(STATI(1,I)+	PLM02790
STATTE(1,I))*TTMC));	PLM02800
PUT EDIT(I,STATE(1,I)) (SKIP(2),F(12),E(18,6));	PLM02810
END;	PLM02820
PUT EDIT('DEP INPUT','RELIABILITY')	PLM02830
(SKIP(2),X(3),A(9),X(1),A(11));	PLM02840
DC I=1 TO DUN;	PLM02850
STATD(1,I)=(STATS(1,I)/(STATS(1,I)+STATED(1,I)))*(1-EXP(-(STATS(1,I)	PLM02860
+STATED(1,I))*TIMO));	PLM02870
PUT EDIT(I,STATE(1,I)) (SKIP(2),F(12),E(18,6));	PLM02880
END;	PLM02890
END CECA;	PLM02900

system unavailabilities during this short interval of time (see examples in Section 8.2 of this thesis).

The arrays STATT and STATD are allocated to store the failure rates of free and replicated components respectively. The arrays ETТА and ETTAD are used to store the test interval (η), TTETA and TTETAD to store the inspection period (θ), whereas TAUU and TAUUD are employed to store the repair duration (τ).

The values of λ_{eff} and q from equations (2.5.9) and (2.5.10) are calculated for each free and replicated component and are stored in the arrays STATTE and STATTED, QUEUE and QUUED, respectively, for free and replicated components. At each time step, a special procedure is employed to determine the value of m in equation (2.5.8). The statements performing this task in the SHR procedure are summarized in Table (7.2.3). Variable TTMO indicates the time for which the unavailability will be calculated. The value of m provides the number of the specific test interval that should be used. For $m = 1$, the code will use equation (8.5.11), since it indicates that the component is in its first test interval. For $m > 1$ the equation (2.5.8) will be used with the assumption that the subsequent test intervals are identical (i.e., only the first test interval differs from the other).

The unavailabilities calculated at each time step are assigned to arrays STATE (1,I) and STATD(1,I) in order to calculate the top and higher order module unavailabilities. For periodically tested components, no approximations such as those discussed in Section 2.7 can be applied, since it seems that no correlation exists between a simple module inspection and repair duration, and its input components' parameters (i.e. η , θ , τ , and λ).

Table (7.2.3) The List of Statements to Indicate
the Cycle in which a Component is
Operating

TTMO1=0;	PLM03680
EMM=0;	PLM03690
DO WHILE (TTMC>TTMC1);	PLM03700
EMM=EMM+1;	PLM03710
TTMO1=(EMM-1)*ETTA(1,I)+FIRTM(1,I);	PLM03720
END;	PLM03730
PCCRD=1;	PLM03740
IF (EMM=1) THEN	PLM03750
MOSK=TTMO+TAVV(1,I)+TTETA(1,I);	PLM03760
ELSE MOSK=TTMO-TTMO1+ETTA(1,I);	PLM03770

If the override unavailability has a value of less than unity, and if the time step is within the inspection period of the components, then the value of the predicted unavailability will be multiplied by a value called POORD, where POORD is the override probability calculated from the following equation:

$$POORD = q_o + (1 - q_o)(1 - e^{-\lambda_{eff} \cdot \eta})$$

q_o = override unavailability

Therefore, the arrays STATE(1,I) and STATD(1,I) will change to the following form:

STATE(1,I) = POORD*STATE(1,I) and

STATD(1,I) = POORD*STATD(1,I)

The list of the statements of the SHR procedure is presented in Table (7.2.4). Similar to the procedures SHORE and CECA, after the first time step the calculation switches to the entry point SAROGH in order to avoid recalculation and reallocation of different variables that remain the same throughout the calculations.

The SHR procedure is also designed to combine different classes of time-dependent components. This enables the code to treat any combination of different classes of components. Handling of trees with different classes of components is performed by the use of the zero test interval condition for those components that are not periodically tested (i.e., $ETTA(1,I) = ETTAD(1,I) = 0$). If the zero test interval condition arises,

Table (7.2.4) Procedure SHR

SHR: PROC:	PLM02910
P=DEL;	PLM02920
PUT EDIT('THIS IS A TIME DEPENDENT UNAVAILABILITY',	PLM02930
'CALCULATION FOR PERIODICALLY TESTED COMPONENTS')	PLM02940
(SKIP(6),X(10),A(39),X(1),A(46));	PLM02950
GET LIST(MOH);	PLM02960
ALLOCATE DELST;	PLM02970
ALLOCATE AUN;	PLM02980
DO I=1 TO MOH;	PLM02990
GET LIST(DELST(1,I),AUN(1,I));	PLM03000
END;	PLM03010
GET LIST(FUN);	PLM03020
PUT EDIT('NUM FREE EVENT INPUTS=',FUN) (SKIP(2),A(22),F(5));	PLM03030
GET LIST(DUN);	PLM03040
PUT EDIT('NUM REPLICATED EVENT INPUTS=',DUN)	PLM03050
(SKIP(2),A(28),F(5));	PLM03060
ALLOCATE STAT5;	PLM03070
ALLOCATE STATE;	PLM03080
ALLOCATE STAT1;	PLM03090
ALLOCATE STATE2;	PLM03100
ALLOCATE ETTA;	PLM03110
ALLOCATE TTETA;	PLM03120
ALLOCATE TAVV;	PLM03130
ALLOCATE ETTA2;	PLM03140
ALLOCATE TTETA2;	PLM03150
ALLOCATE TAVVD;	PLM03160
ALLOCATE QUEUE;	PLM03170
ALLOCATE QUUED;	PLM03180
ALLOCATE STATTE;	PLM03190
ALLOCATE STATED;	PLM03200
ALLOCATE QUZR;	PLM03210
ALLOCATE QUZRD;	PLM03220
ALLOCATE FIRTM;	PLM03230
ALLOCATE FIRTMD;	PLM03240
TTMO=TTMO+DELST(1,1);	PLM03250
TIE=TIE+1;	PLM03260
DO I=1 TO FUN;	PLM03270
GET LIST(I,STAT1(1,I),ETTA(1,I),TTETA(1,I),TAVV(1,I),	PLM03280
FIRTM(1,I),QUZR(1,I));	PLM03290
END;	PLM03300

(continued)

Table (7.2.4)(continued)

```

DC I=1 TO EUN;
GET LIST(I,STATS(1,I),ETTAD(1,I),TTETAD(1,I),TAVVD(1,I),
FIRTM(1,I),QUZRD(1,I));
END;
DO I=1 TO FUN;
IF ETIA(1,I)=0 THEN DO;
QUEUE(1,I)=0; STATTE(1,I)=0;
END;
ELSE DO;
QUEUE(1,I)=LOG(3-LOG(TTETA(1,I)*STATTE(1,I)));
STATTE(1,I)=STATTE(1,I)*(1-TTETA(1,I)/ETIA(1,I))*(1-TTETA(1,
I)/ETIA(1,I)+2*TAVV(1,I)/ETIA(1,I))+0.2*TTETA(1,I)/ETIA(1,
I)**2;
END;
END;
DO I=1 TO EUN;
IF ETTAD(1,I)=0 THEN DO;
QUUED(1,I)=0; STATEE(1,I)=0;
END;
ELSE DO;
QUUED(1,I)=LOG(3-LOG(TTETAD(1,I)*STATS(1,I)));
STATEE(1,I)=STATS(1,I)*(1-TTETAD(1,I)/ETTAD(1,I))*(1-TTETAD(
1,I)/ETTAD(1,I)+2*TAVVD(1,I)/ETTAD(1,I))+0.2*TTETAD(1,I)/
ETTAD(1,I)**2;
END;
END;
SAROGH: ENTRY;
PUT EDIT('PCINT NUMBER=',TIE) (SKIP(2),X(2),A(13),F(15));
PUT EDIT('FREE INPUT','RELIABILITY')
(SKIP(2),X(2),A(10),X(1),A(11));
DC I=1 TO FUN;
IF ETIA(1,I)=0 THEN DO;
STATE(1,I)=(STATTE(1,I)/(STATTE(1,I)+TTETA(1,I)))*(1-EXP(-(STATTE(1,I)+
TTETA(1,I))*TMC));
PUT EDIT(I,STATE(1,I)) (SKIP(2),F(12),E(18,6));
END;
ELSE DO;
TMC=0;
DO WHILE(TMC>TMC1);
TMC=TMC+1;
TMC1=(TMC-1)*ETIA(1,I)+FIRTM(1,I);
END;
PCRD=1;
IF (TMC=1) THEN
MOSK=TMC+TAVV(1,I)+TTETA(1,I);
ELSE MOSK=TMC-TMC1+ETIA(1,I);
IF (QUZRD(1,I)=1) THEN GO TO MACH;
ELSE IF (MOSK>TTETA(1,I)) THEN GO TO MACH;
ELSE PCRD=QUZRD(1,I)+(1-QUZE(1,I))*(1-EXP(-ETIA(1,I)*STATTE
(1,I)));
MACH: STATE(1,I)=(1-EXP(-MOSK*STATTE(1,I)))*(1-EXP(-(MOSK/TTETA(1,
I))*QUEUE(1,I)))*PCRD;
PUT EDIT(I,STATE(1,I)) (SKIP(2),F(12),E(18,6));
END;

```

(continued)

Table (7.2.4) continued

END;	PLM03860
PUT EDIT('DEP INPUT ','RELIABILITY')	PLM03870
(SKIP(2),X(3),A(9),X(1),A(11));	PLM03880
DO I=1 TO LUN;	PLM03890
IF ETTAD(1,I)=0 THEN GO;	PLM03900
STATE(1,I)=(STATS(1,I)/(STATS(1,I)+TTETAD(1,I)))*(1-EXP(-(STATS(1,I)	PLM03910
+TTETAD(1,I))*TTMC));	PLM03920
PUT EDIT(I,STATD(1,I))(SKIP(2),F(12),E(18,6));	PLM03930
END;	PLM03940
ELSE DO:	PLM03950
EMM=0; TTMCI=0;	PLM03960
DO WHILE(TTMC>TTMC1);	PLM03970
EMM=EMM+1;	PLM03980
TTMO1=(EMM-1)*ETTAD(1,I)+FISTMD(1,I);	PLM03990
END;	PLM04000
PCCRD=1;	PLM04010
IF(EMM=1) THEN	PLM04020
SOSK=TTMO+TAVVD(1,I)+TTETAD(1,I);	PLM04030
ELSE	PLM04040
SOSK=TTMC-TTMC1+ETTAD(1,I);	PLM04050
IF(QUZRD(1,I)=1) THEN GO TO GACH;	PLM04060
ELSE IF(SOSK>TTETAD(1,I)) THEN GO TO GACH;	PLM04070
ELSE PCCRD=QUZRD(1,I)+(1-QUZRD(1,I))*(1-EXP(-ETTAD(1,I)*	PLM04080
STATED(1,I)));	PLM04090
GACH: STATD(1,I)=(1-EXP(-SOSK*STATED(1,I)))*(1-EXP(-(SOSK/TTETAD(1	PLM04100
,I))*QUUED(1,I)))*PCCRI;	PLM04110
PUT EDIT(I,STATD(1,I))	PLM04120
(SKIP(2),F(12),E(18,6));	PLM04130
END;	PLM04140
END;	PLM04150
END SHR;	

then PL-MODT will automatically apply equation (2.4.9) and thus will not evaluate equations (2.5.8) through (2.5.11)

It should be noted here that in the case of non-repairable components (Class 2), equation (2.4.9) reduces to the form given by Equation (2.3.2) with the repair rate equal to zero (i.e. $\mu = 0$). Therefore, for a non-repairable component we have $ETTA(1,I) = ETTAD(1,I) = TTETA(1,I) = TTETAD(1,I) = 0$.

One final note discusses the usefulness of the procedures SHORE and CECA. Since the SHR procedure can handle not only periodically tested components, but also repairable (revealed fault) and non-repairable components, there is seemingly no need for the procedures SHORE and CECA. It should be pointed out, however, that the use of SHORE and CECA saves computation time due to the fact that no procedure exists in these subroutines to recognize the class of each component at each individual time step. Furthermore, if the fault tree components would consist of only one class, then the corresponding procedure should better be used because of its compactness and simplicity.

7.2.4 Modifications Performed in the Old External Procedure NUMERO

The modifications in the external procedure NUMERO include the addition of a special routine to the main routine of the NUMERO procedure to select one of the procedures STAT-IN, SHORE, CECA or SHR depending on the fault tree component type(s). Also, this special routine is designed to select time steps and change the time step length for different periods of operation. In addition, calculation of the average unavailability over the period of operation is carried out in this routine.

The unavailability as a function of time can be calculated by assuming that after unavailabilities of two successive time steps are found for the top event, the unavailability changes linearly between these two points. Therefore, the unavailability as a function of time for the top event of a fault tree would look like the curve shown in Figure (7.2.1). In Figure (7.2.1) the effect of smaller time steps for the calculation of higher unavailability values during a small period of time is shown. Particularly if all time steps were large, a portion of the two peaks in the unavailability curves shown in Figure (7.2.1) could not be evaluated accurately; and by using a small time step for the whole operation time, the cost of computation would become very high. The total area under the curve in Figure (7.2.1) is calculated by adding the areas occupied by each time step interval, and using the linear unavailability approximation between the two time steps. The total area is then divided by the total operation time t , where $T = \sum_{i=1}^N AUN(1,I)$, to get the average unavailability. The set of statements in Table (7.2.5) shows the routine added to the old procedure NUMERO. In these statements variable ESF is read in the input, and indicates the class of the components in the fault tree. Variable REY represents the top event unavailability. At each time step, the value of REY is determined by calling the EXPECT procedure. TTMO and TIE variables represent the time and number of time steps for which the calculation is performed.

Other changes in the procedure NUMERO are related to the internal procedures PLUS and MINUP. These procedures calculate the unavailability of OR modules and evaluate the unavailability of improper modules through

Figure 7.2.1 Unavailability as a Function of Time

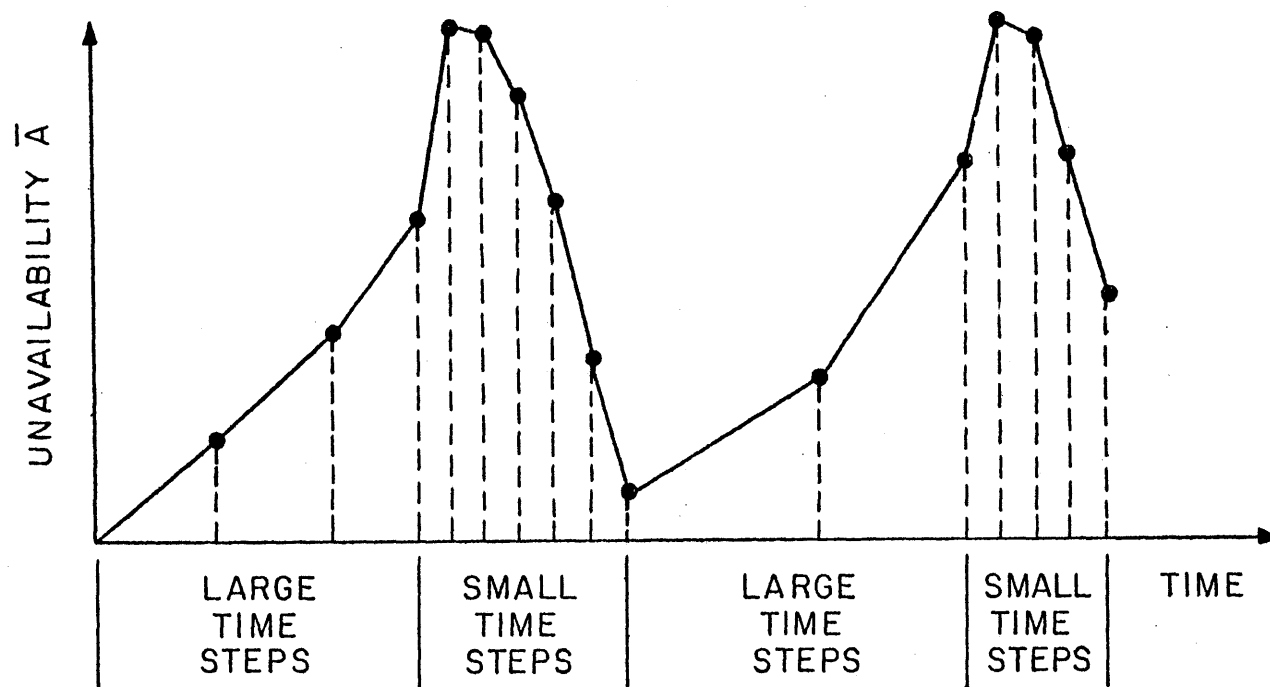


Table (7.2.5) Modifications in the Main Routine of the NUMERO Procedure

/* NUMERC */	PLM00880
GET LIST (FOX);	PLM00890
IF (FOX=0) THEN GO TO STAC91;	PLM00900
RETURN;	PLM00910
STAC91: GET LIST (ESF);	PLM00920
IF (ESF=1) THEN GO TO TM9DE;	PLM00930
ELSE DO;	PLM00940
CALL STAT_IN;	PLM00950
CALL EXPECT;	PLM00960
CALL IMFCRTANCE;	PLM00970
END;	PLM00980
RETURN;	PLM00990
TM9DE: IF (ESF=2) THEN GO TO TM9DE2;	PLM01000
ELSE DO;	PLM01010
CALL SHCEE;	PLM01020
CALL EXPECT;	PLM01030
JAVI=(REY/2)*DEL9T(1,1);	PLM01040
MACAR=REY;	PLM01050
CALL IMPORTANCE;	PLM01060
DO WHILE (SISI<MOH);	PLM01070
SISI=SISI+1;	PLM01080
DO WHILE (TIE<AUN(1,SISI));	PLM01090
TIE=TIE+DEL9T(1,SISI);	PLM01100
PUT EDIT('THIS IS A NEW TIME STEP','TIE=', TIE)	PLM01110
(SKIP(4),A(23),X(3),A(4),F(10));	PLM01120
CALL COEA;	PLM01130
CALL EXPECT;	PLM01140
JAVI=JAVI+((MACAR+REY)/2)*DEL9T(1,SISI);	PLM01150
MACAR=REY;	PLM01160
CALL IMPORTANCE;	PLM01170
END;	PLM01180
END;	PLM01190
END;	PLM01200
JAVI=JAVI/TIE;	PLM01210
PUT EDIT('MEAN UNAVAILABILITY=',JAVI) (SKIP(5),X(19),A(20)	PLM01220
,F(18,6));	PLM01230
RETURN;	PLM01240
TM9DE2: IF (ESF=3) THEN GO TO TM9DE3;	PLM01250
ELSE DO;	PLM01260
CALL CECA;	PLM01270
CALL EXPECT;	PLM01280
JAVI=(REY/2)*DEL9T(1,1);	PLM01290
MACAR=REY;	PLM01300
CALL IMFCRTANCE;	PLM01310
DO WHILE (SISI<MOH);	PLM01320
SISI=SISI+1;	PLM01330
MASHAD=MASHAD+AUN(1,SISI)/DEL9T(1,SISI);	PLM01340
DO WHILE (TIE<MASHAD);	PLM01350
TTMC=TTMC+DEL9T(1,SISI);	PLM01360
TIE=TIE+1;	PLM01370

(continued)

Table (7.2.5) continued

PUT EDIT('THIS IS A NEW TIME STEP', 'TIE=', TIE, 'TTMC=',	PLM01380
TTMO) (SKIP(4), A(23), X(3), A(4), F(10), X(8), A(5), F(10));	PLM01390
CALL SOBA;	PLM01400
CALL EXPECT;	PLM01410
JAVI=JAVI+((MACAR+REY)/2)*DEL9T(1, SISI);	PLM01420
MACAR=REY;	PLM01430
CALL IMPORTANCE;	PLM01440
END;	PLM01450
END;	PLM01460
END;	PLM01470
JAVI=JAVI/TTMO;	PLM01480
PUT EDIT('MEAN UNAVAILABILITY=', JAVI) (SKIP(5), X(19), A(20)	PLM01490
, F(18,6));	PLM01500
RETURN;	PLM01510
TM9DE3: IF (ESF=4) THEN GO TO TM9DE4;	PLM01520
EISE DO;	PLM01530
CALL SHR;	PLM01540
CALL EXPECT;	PLM01550
JAVI=(REY/2)*DEL9T(1, 1);	PLM01560
MACAR=REY;	PLM01570
CALL IMFCRTANCE;	PLM01580
DO WHILE (SISI<MOH);	PLM01590
SISI=SISI+1;	PLM01600
MASHAD=MASHAD+AUN(1, SISI)/DEL9T(1, SISI);	PLM01610
DO WHILE (TIE<MASHAD);	PLM01620
TTMO=TTMO+DEL9T(1, SISI);	PLM01630
TIE=TIE+1;	PLM01640
PUT EDIT('THIS IS A NEW TIME STEP', 'TIE=', TIE, 'TTMO=',	PLM01650
TTMO) (SKIP(4), A(23), X(3), A(4), F(10), X(8), A(5), F(10));	PLM01660
CALL SABCGH;	PLM01670
CALL EXPECT;	PLM01680
JAVI=JAVI+((MACAR+REY)/2)*DEL9T(1, SISI);	PLM01690
MACAR=REY;	PLM01700
CALL IMFCRTANCE;	PLM01710
END;	PLM01720
END;	PLM01730
END;	PLM01740
JAVI=JAVI/TTMO;	PLM01750
PUT EDIT('MEAN UNAVAILABILITY=', JAVI) (SKIP(5), X(19), A(20)	PLM01760
, F(18,6));	PLM01770
RETURN;	PLM01780
TM9DE4: PUT EDIT('OPTICN NOT KNOWN') (SKIP(1), A(16));	PLM01790
GO TO SOSCL;	PLM01800

the modular minimal cutsets, respectively. In the old version of the PLUS and MINUP procedures, the unavailabilities for components and modular cutsets were found by adding them together. For time-dependent analysis, since there are periods of time in which the unavailability becomes large (greater than 0.2), the approximation of adding unavailabilities is no longer valid. Therefore, the procedures PLUS and MINUP had to be changed to evaluate the union of component unavailability in the PLUS procedure, and the union of the modular cutsets in the MINUP procedure. Table (7.2.6) shows the new form of the PLUS procedure. Changes in the procedure MINUP are very similar to the PLUS procedure and, therefore, only the MINUP procedure is shown here.

7.2.5 Modifications Performed in the Internal Procedure IMPORTANCE

As has been discussed in Section 3.2, the procedure IMPORTANCE in the PL-MOD Code provides the V-F importance measures for all of the fault tree components as well as simple and higher order modules. This procedure is modified to a new form such that it accounts for the CUT-OFF value to exclude those components and modules that have V-F importances of less than this prescribed value. For any further evaluation, the new reduced form of the fault tree can be used.

Obviously, the cut-off value plays an important role in the reduction strategy. That is, different values of the cut-off value result in different reduced fault trees. Therefore, a great deal of attention must be paid to the selection of the cut-off value in order to achieve a desired and yet accurate form of the reduced tree.

Table (7.2.6) Procedure PLUS

```

PLUS: PROC(EAT, EXA);
DECLARE EAT PCINTER;
DECLARE EXA LABEL;
PT=BAT;
REX=0;
IF (PRCP.LIM=1 & PROP.TIL(1)=0) THEN GO TO PLUA;
DO J=1 TO PROP.IIM;
  REX=REX+STATE(1, PRCP.TIL(J));
END;
PLUA: IF (PROP.MIM=1 & PRCP.FIM(1)=NULL) THEN GO TO PLUB;
      DO J=1 TO PRCP.MIM;
        IF (PROP.PIM(J)->PRCP.HCST~=NULL) THEN DO;
          PR=PROP.FIM(J)->PROP.HOST;
          REX=REX+PER.REL(1);
        END;
      ELSE REX=REX+PRCP.PIM(J)->PROP.REL(1);
      END;
PLUB: IF REX<0.2 THEN GO TO ZACH;
      REX=1;
      IF (PROP.LIM=1 & PROP.TIL(1)=0) THEN GO TO SLUA;
      DO J=1 TO PRCP.LIM;
        REX=REX*(1-STATE(1, PRCP.TIL(J)));
      END;
SLUA: IF (PROP.MIM=1 & PROP.PIM(1)=NULL) THEN GO TO SLUB;
      DO J=1 TO PROP.MIM;
        IF (PRCP.PIM(J)->PROP.HOST~=NULL) THEN DO;
          PR=PROP.FIM(J)->PROP.HOST;
          REX=REX*(1-PER.REL(1));
        END;
      ELSE REX=REX*(1-PROP.PIM(J)->PRCP.REL(1));
      END;
SLUB: REX=1-REX;
ZACH:  PROP.REL(1)=REX;
      GO TO EXA;
END PLUS;

```

```

PLM04440
PLM04450
PLM04460
PLM04470
PLM04480
PLM04490
PLM04500
PLM04510
PLM04520
PLM04530
PLM04540
PLM04550
PLM04560
PLM04570
PLM04580
PLM04590
PLM04600
PLM04610
PLM04620
PLM04630
PLM04640
PLM04650
PLM04660
PLM04670
PLM04680
PLM04690
PLM04700
PLM04710
PLM04720
PLM04730
PLM04740
PLM04750
PLM04760
PLM04770
PLM04780

```

The new form of the IMPORTANCE Procedure is linked with the code PL-MODT, so that for future developments of the code, the reduced version of fault trees will be used by the code to evaluate time-dependent behaviour of the fault tree.

In the modified form of the IMPORTANCE subroutine, the top event importance will be set equal to 1. Starting from the top event it proceeds to the bottom. First V-F importances of the higher order modules and replicated events that are connected to the top event are calculated. Next, the code automatically removes all modules and replicated events whose V-F importances are less than the prescribed cut-off value. Each higher order module which is removed contains some other proper modules and free components that are attached to them. Therefore, there is no need to compute the V-F importance of any of these attached members of the removed higher order modules because their importances are always equal to or less than that of the parent module, which by itself is lower than the cut-off value. If, however, the higher order module is not removed because of its importance being higher than the cut-off value, the code will proceed to calculate the importances of simple modules and components attached to it, and to automatically remove any component or simple module with an importance low enough to be cancelled.

Therefore, starting from the top event of an unreduced fault tree, all of the branches of the tree with low importances will be cut. Those lower order branches and leaves which show low importances compared to the cut-off value will be cancelled. The pruned tree is the reduced version of the original tree.

The pruning process is performed by setting the variable STATE(2,I) which is the measure of V-F importance for components equal to zero. If the module is reduced, then automatically all components input to this module are cancelled (i.e. STATE(2,I) = 0).

For example, the removal of a simple AND module proceeds as follows:

```
IF(PROP.LIM=1 & PROP.TIL(1)=0)THEN DO;
PROP.REL(2) = 0;
GO TO EMEL;
```

(To calculate the importance of the NOT gate if any)

```
END;
ELSE PROP.REL(2) = PER.REL(2);
IF(PROP.REL(2) < CUT-OFF)THEN PROP.REL(2) = 0;
DO IT = 1 TO PROP.LIM;
STATE(2,PROP.TIL(IT)) = PROP.REL(2);
END;
```

Therefore, all of the components input to the AND gate will have an importance which is either equal to zero (i.e. cancelled) or equal to the importance of the AND module.

Table (7.2.7) lists a set of statements that form the new IMPORTANCE procedure. The following sets of statements in the new procedure are particularly added to the old IMPORTANCE Procedure to identify modules and components that should be cancelled:

Table (7.2.7) Procedure IMPORTANCE

219

```

/*      IMPORTANCE      (VESELY-  FUSSELL)      */
IMPORTANCE:  PROC;

  IF (DEL=1) THEN GC TO ASAC;
  DECLARE CUT_OFF FLOAT DECIMAL(16);
  GET LIST(CUT_OFF);
  IF (CUT_OFF=0) THEN
    PUT SKIP LIST('NO REDUCTION BASED ON IMPORTANCE IS REQUESTED');
  ELSE PUT SKIP LIST('AUTOMATIC REDUCTION WILL BE PERFORMED');
    BUG=1;
    PT=STCRK;
    IF PROP.HOST~=NULL THEN GC TO IMA;
    BUM=PROP.MIM;
    ALLOCATE CLM(BUM);
    OLM=PRCP.FIM;
    PRCP.REL(2)=1;
    PUT EDIT('MODULE=',PROP.NAME,'IMP=',PROP.REL(2))
      (SKIP(1),A(7),F(5),A(4),F(18,6));
    IF PROP.VALUE=1 THEN DC;
    IF (PROP.LIM=1 & PROP.TIL(1)=C) THEN GO TO IME;
    DO I=1 TO PRCP.LIM;
      STATE(2,PROP.TIL(I))=1;
    END;
    IF PRCP.VALUE=2 THEN DC;
    IF (PROP.LIM=1 & PROP.TIL(1)=0) THEN GO TO IME;
    DO I=1 TO PRCP.LIM;
      STATE(2,PROP.TIL(I))=STATE(1,PRCP.TIL(I))/PROP.REL(1);
    IF (STATE(2,PROP.TIL(I))<CUT_OFF) THEN STATE(2,PROP.TIL(I))=C ;
    END;
  END;
GO TO IMA;

/*      CUT SET CASE      */
IMA:  PR=PRCP.HCST;
  IF (PROP.MIM=1 & PRCP.FIM(1)=NULL) THEN DO;
    BUM=0;
    BUN=0;
  END;
  ELSE DO;
    BUM=PROP.MIM;
    BUN=1;
  END;
  BUM=BUM+PR.FER.LEAL;
  BUZ=BUM;
  DO IK=1 TO FER.RAM;
    MA=PR.TAR(IK);
    DA=-CEIL(-MA/10000);
    JA=-CEIL(-MA/1000);
    JAK=JA-10*DA;
    IF JAK=9 THEN DO;
      BUM=BUM+1;
    END;
  END;
  BUZ=BUM-BUZ;
  ALLOCATE OLM(BUM);
  IF BUN=0 THEN DO;
    I=1;
  GO TO IMA0;

```

```

PLM06040
PLM06050
PLM06060
PLM06070
PLM06080
PLM06090
PLM06100
PLM06110
PLM06120
PLM06130
PLM06140
PLM06150
PLM06160
PLM06170
PLM06180
PLM06190
PLM06200
PLM06210
PLM06220
PLM06230
PLM06240
PLM06250
PLM06260
PLM06270
PLM06280
PLM06290
PLM06300
PLM06310
PLM06320
PLM06330
PLM06340
PLM06350
PLM06360
PLM06370
PLM06380
PLM06390
PLM06400
PLM06410
PLM06420
PLM06430
PLM06440
PLM06450
PLM06460
PLM06470
PLM06480
PLM06490
PLM06500
PLM06510
PLM06520
PLM06530
PLM06540
PLM06550
PLM06560
PLM06570
PLM06580
PLM06590
PLM06600

```

(continued)

```

END:
DO I=1 TO PROP.MIM;
OLM(I)=PRCP.PIM(I);
ENC:
PLM06610
PLM06620
PLM06630
PLM06640
PLM06650
PLM06660
PLM06670
PLM06680
PLM06690
PLM06700
PLM06710
PLM06720
PLM06730
PLM06740
PLM06750
PLM06760
PLM06770
PLM06780
PLM06790
PLM06800
PLM06810
PLM06820
PLM06830
PLM06840
PLM06850
PLM06860
PLM06870
PLM06880
PLM06890
PLM06900
PLM06910
PLM06920
PLM06930
PLM06940
PLM06950
PLM06960
PLM06970
PLM06980
PLM06990
PLM07000
PLM07010
PLM07020
PLM07030
PLM07040
PLM07050
PLM07060
PLM07070
PLM07080
PLM07090
PLM07100
PLM07110
PLM07120
PLM07130
PLM07140
PLM07150

IMAO: DO IL=I TO EUM-BUZ;
OLM(IL)=PEF.KIM(IL-I+1);
ENC:
IF (BUZ=0) THEN DO;
DO IK=1 TO PER.RAM;
MA=PER.TAB(IK);
DA=-CEIL(-MA/10000);
JA=-CEIL(-MA/10000);
JAK=JA-10*DA;
IF (JAK=9) THEN DO;
DO IX=1 TO EMCD;
IF (TRIM(IX)=MA) THEN GO TO IMA4;
END;
IM4: OLM(IL)=PRIN(IX)->AP.SPIT;
PUT EDIT('INDEX=',IL,'PROP=',OLM(IL)->PROP.NAME)
(SKIP(1),A(6),F(5),A(5),F(5));
IL=IL+1;
END;
END;
END;
PER.REL(2)=1;
PUT EDIT('PATR=',PROP.NAME,'IMP=',PER.REL(2))
(SKIP(1),A(5),F(5),A(4),E(18,6));
IF PROP.VALUE>2 THEN GO TO IMA2;
IF PROP.VALUE=1 THEN DO;
IF (PROP.LIM=1 & PRCP.TIL(1)=0) THEN DO ;
PROP.REL(2)=0;
GO TO IMA1;
END;
PROP.REL(2)=1;
DO I=1 TO PROP.LIM;
STATE(2,PRCP.TIL(I))=1;
END;
END;
IF PRCP.VALUE=2 THEN DO;
IF (PROP.LIM=1 & PROP.TIL(1)=0) THEN DO;
PROP.REL(2)=0;
GO TO IMA1;
END;
PRCP.REL(2)=PROP.REL(1)/PRP.REL(1);
IF (PROP.REL(2)<CUT_CFF) THEN PRCP.REL(2)=0 ;
DO I=1 TO PROP.LIM;
STATE(2,PROP.TIL(I))=STATE(1,PRCP.TIL(I))/PER.REL(1);
IF (STATE(2,PROP.TIL(I))<CUT_CFF) THEN STATE(2,PROP.TIL(I))=0 ;
END;
END;
IM41: DO I=1 TO PER.RAM;
EX=I;
TIERRA=PR;
QT=PER.DEXTER;
CALL MINUF(EX);

```

(continued)

```

PUT EDIT('I=',I,'PER.TAR=',PER.TAR(I),'REY=',REY)
(SKIP(2),A(2),F(5),A(3),F(5),A(4),E(18,6));
MA=PER.TAR(I);
DA=-CEIL(-MA/10000);
JA=-CEIL(-MA/1000);
JAK=JA-10*JA;
NA=MA-(1000)*JA;
IF (JAK=2) THEN STATE(2,NA)=REY/PER.REL(1);
IF (STATE(2,NA)<CUT_OFF) THEN STATE(2,NA)=0;
IF (JAK=2) THEN DC;
SNCT=REY/PER.REL(1);
PUT EDIT('NOTSTATE=',MA,'IMP=',SNCT)
(SKIP(2),A(9),F(5),X(2),A(4),E(18,6));
END;
END;
GO TO IME;
/* SYMMETRIC CASE */
IMA2: PROP.REL(2)=0;
IF (PER.RAM=1 & PER.TAR(1)=0) THEN GO TO IME;
ELSE DC I=1 TO PER.RAM;
EX=I;
TIERRA=PR;
QT=PER.DEXTER;
CALL MINUP(EX);
PUT EDIT('I=',I,'PER.TAR=',PER.TAR(I),'REY=',REY)
(SKIP(2),A(2),F(5),A(8),F(5),A(4),E(18,6));
STATE(2,PER.TAR(I))=REY;
IF (STATE(2,PER.TAR(I))<CUT_OFF) THEN STATE(2,PER.TAR(I))=0;
END;
GO TO IME;
/* ICCP STARTS HERE */
IM2: DO WHILE(BUG=0);
BUG=BUM;
PUT LIST('BUG=',BUG);
IF (BUG=0) THEN GO TO IME;
ALLOCATE GOLD(BUG);
DO I=1 TO BUG;
GOLD(I)=OLM(I);
PUT EDIT('GOLD=',I,'PROP=',GOLD(I)->PROP.NAME)
(SKIP(1),A(5),F(5),A(5),F(5));
END;
FREE OLM;
BUM=0;
DO I=1 TO BUG;
PT=GOLD(I);
IF PROP.HOST=NULL THEN DO;
IF (PROP.MIM=1 & PROP.PIM(1)=NULL) THEN GO TO IME3;
ELSE BUM=BUM+PROP.MIM;
GO TO IME3;
END;
ELSE PR=PROP.HOST;
IF (PROP.MIM=1 & PROP.FIM(1)=NULL) THEN GO TO IME2;
ELSE BUM=BUM+PROP.MIM;
IME2: IF (PER.LEAL=1 & PER.KIM(1)=NULL) THEN GO TO IME1;
ELSE BUM=BUM+PER.LEAL;

```

PLM07160
 PLM07170
 PLM07180
 PLM07190
 PLM07200
 PLM07210
 PLM07220
 PLM07230
 PLM07240
 PLM07250
 PLM07260
 PLM07270
 PLM07280
 PLM07290
 PLM07300
 PLM07310
 PLM07320
 PLM07330
 PLM07340
 PLM07350
 PLM07360
 PLM07370
 PLM07380
 PLM07390
 PLM07400
 PLM07410
 PLM07420
 PLM07430
 PLM07440
 PLM07450
 PLM07460
 PLM07470
 PLM07480
 PLM07490
 PLM07500
 PLM07510
 PLM07520
 PLM07530
 PLM07540
 PLM07550
 PLM07560
 PLM07570
 PLM07580
 PLM07590
 PLM07600
 PLM07610
 PLM07620
 PLM07630
 PLM07640
 PLM07650
 PLM07660
 PLM07670
 PLM07680
 PLM07690
 PLM07700

(continued)

```

IME1:  DO IX=1 TO PER.RAM;
      MA=PER.TAR(IX);
      DA=-CEIL(-MA/10000);
      JA=-CEIL(-MA/1000);
      JAK=JA-10*DA;
      IF JAK=9 THEN DC;
      BUM=BUM+1;
      END;
      END;
IME3:  END;
      IF BUM=0 THEN GO TO IMI3;
      ALLOCATE OLM(BUM);
      IL=0;
      DO I=1 TO EUG;
      PT=GOLD(I);
      IF PRCP.HCST=NULL THEN DO;
      IF (PROP.MIM=1 & PRCP.PIM(1)=NULL) THEN GO TO IMI4;
      DO IT=1 TO PROP.MIM;
      IL=IL+1;
      OLM(IL)=PRCP.PIM(IT);
      END;
      GO TO IMI4;
      END;
      ELSE PR=PRCP.HCST;
      PUT EDIT('HOST','PROP=',PRCP.NAME)
      (SKIP(1),A(4),A(5),F(5));
      IF (PROP.MIM=1 & PROP.PIM(1)=NULL) THEN GO TO IMI2;
      DO IT=1 TO PROP.MIM;
      IL=IL+1;
      OLM(IL)=PROP.PIM(IT);
      END;
IMI2: IF (PER.LEAL=1 & PER.KIM(1)=NULL) THEN GO TO IMI1;
      DO IT=1 TO PER.LEAL;
      IL=IL+1;
      OLM(IL)=PER.KIM(IT);
      END;
IMI1:  DO IK=1 TO PER.RAM;
      MA=PER.TAR(IK);
      DA=-CEIL(-MA/10000);
      JA=-CEIL(-MA/1000);
      JAK=JA-10*DA;
      IF JAK=9 THEN DC;
      DO IX=1 TO EMOD;
      IF (TRIM(IX)=MA) THEN GO TO IMK1;
      END;
IMK1:  CLM(IL+1)=PRIN(IX)->AP.SPIT;
      IL=IL+1;
      END;
      END;
IMI4:  END;
/* ASSIGN IMPORTANCES OF CLDER GENERATION */
IMI3:  DO I=1 TO BUG;
      PUT EDIT('I=',I) (SKIP(1),A(2),F(5));
      PT=GOLD(I);
      CAT=PRCP.RCCT;

```

PLM07710
 PLM07720
 PLM07730
 PLM07740
 PLM07750
 PLM07760
 PLM07770
 PLM07780
 PLM07790
 PLM07800
 PLM07810
 PLM07820
 PLM07830
 PLM07840
 PLM07850
 PLM07860
 PLM07870
 PLM07880
 PLM07890
 PLM07900
 PLM07910
 PLM07920
 PLM07930
 PLM07940
 PLM07950
 PLM07960
 PLM07970
 PLM07980
 PLM07990
 PLM08000
 PLM08010
 PLM08020
 PLM08030
 PLM08040
 PLM08050
 PLM08060
 PLM08070
 PLM08080
 PLM08090
 PLM08100
 PLM08110
 PLM08120
 PLM08130
 PLM08140
 PLM08150
 PLM08160
 PLM08170
 PLM08180
 PLM08190
 PLM08200
 PLM08210
 PLM08220
 PLM08230
 PLM08240
 PLM08250

(continued)

Table (7.2.7) continued

```

IF (CAT->PROP.TIPO=0) THEN EC;
  APT=CAT;
  MA=AP.NAF;
  JA=-CEIL (-MA/1000);
  NA=MA-(1000)*JA;
  IF (PROP.HCST~=NULL) THEN EC ;
  PR=PRCP.HCST;
  TIERRA=PR;
  QT=PER.DEXTER;
  PER.REL(2)=STATD(2,NA);
  GO TO IMK3;
END;
ELSE PROP.REL(2)=STATD(2,NA);
IF (PROP.REL(2)<CUT_OFF) THEN PROP.REL(2)=0 ;
GO TO IMK2;
END;
IF CAT->PROP.HOST~=NULL THEN GO TO EMA2;
IF PROP.HOST~=NULL THEN GO TO EMA1;
IMK4: IF CAT->PROP.VALUE=1 THEN PROP.REL(2)=CAT->PROP.REL(2);
ELSE PROP.REL(2)=PROP.REL(1)*CAT->PROP.REL(2)/CAT->PROP.REL(1);
IF (PROP.REL(2)<CUT_OFF) THEN PROP.REL(2)=0 ;
IMK2: IF PROP.VALUE=1 THEN DO;
  IF (PROP.LIM=1 & PROP.TIL(1)=0) THEN GC TC AME;
  DO IT=1 TC PRCP.LIM;
  STATE(2,PROP.TIL(IT))=PROP.REL(2);
END;
GO TO AME;
END;
ELSE DC;
  IF (PROP.LIM=1 & PRCP.TIL(1)=0) THEN GO TO AME;
  DO IT=1 TC PRCP.LIM;
  IF (PROP.REL(2)=0) THEN STATE(2,PRCP.TIL(IT))=0 ;
  ELSE STATE(2,PRCP.TIL(IT))=STATE(1,PROP.TIL(IT))*PROP.REL(2)/
    PROP.REL(1);
  IF (STATE(2,PROP.TIL(IT))<CUT_CFF) THEN STATE(2,PROP.TIL(IT))
    =0;
END;
GO TO AME;
END;
EMA1: PR=PRCP.HCST;
TIERRA=PR;
QT=PER.DEXTER;
IF CAT->PROP.VALUE=1 THEN PER.REL(2)=CAT->PROP.REL(2);
ELSE PER.REL(2)=PER.REL(1)*CAT->PRCP.REL(2)/CAT->PROP.REL(1);
IMK3: IF PRCP.VALUE=1 THEN DC;
  IF (PROP.LIM=1 & PROP.TIL(1)=0) THEN DO;
    PROP.REL(2)=0;
  GO TO EMA1;
  END;
  ELSE PROP.REL(2)=PER.REL(2);
  IF (PROP.REL(2)<CUT_OFF) THEN PRCP.REL(2)=0 ;
  DO IT=1 TO PROP.LIM;
  STATE(2,PRCP.TIL(IT))=PROP.REL(2);
  END;
  GO TO EMA1;

```

(continued)

Table (7.2.7) (continued)

END;	PLM08810
IF PROP.VALUE=2 THEN DO;	PLM08820
IF (PROP.LIM=1 & PROP.TIL(1)=0) THEN DO;	PLM08830
PRCP.REL(2)=0;	PLM08840
GO TO EME1;	PLM08850
END;	PLM08860
ELSE PROP.REL(2)=PER.REL(2)*PRCP.REL(1)/PER.REL(1);	PLM08870
IF (PROP.REL(2)<CUT_OFF) THEN PRCP.REL(2)=0;	PLM08880
DO IT=1 TO PROP.LIM;	PLM08890
IF (PROP.REL(2)=0) THEN STATE(2,PROP.TIL(IT))=0;	PLM08900
ELSE STATE(2,PROP.TIL(IT))=STATE(1,PRCP.TIL(IT))*PROP.REL(2)/	PLM08910
PRCP.REL(1);	PLM08920
IF (STATE(2,PROP.TIL(IT))<CUT_OFF) THEN STATE(2,PROP.TIL(IT))	PLM08930
=0;	PLM08940
END;	PLM08950
GO TO EME1;	PLM08960
END;	PLM08970
IF PROP.VALUE>2 THEN DO;	PLM08980
PRCP.REL(2)=0;	PLM08990
IF (PER.RAM=1 & PER.TAR(1)=0) THEN GO TO AME;	PLM09000
DO IT=1 TO PER.RAM;	PLM09010
EX=IT;	PLM09020
CALL MINUP(EX);	PLM09030
STATE(2,PER.TAR(IT))=REY*PER.REL(2)/PER.REL(1);	PLM09040
IF (STATE(2,PER.TAR(IT))<CUT_OFF) THEN STATE(2,PER.TAR(IT))=0;	PLM09050
END;	PLM09060
GO TO AME;	PLM09070
END;	PLM09080
EME1: DO IT=1 TO PER.RAM;	PLM09090
EX=IT;	PLM09100
CALL MINUP(EX);	PLM09110
MA=PER.TAR(IT);	PLM09120
DA=-CEIL(-MA/10000);	PLM09130
JA=-CEIL(-MA/1000);	PLM09140
JAK=JA-10*DA;	PLM09150
NA=MA-1000*JA;	PLM09160
IF (JAK=2) THEN STATE(2,NA)=REY*PER.REL(2)/PER.REL(1);	PLM09170
IF (STATE(2,NA)<CUT_OFF) THEN STATE(2,NA)=0;	PLM09180
IF JAK=2 THEN DO;	PLM09190
SNOT=REY*PER.REL(2)/PER.REL(1);	PLM09200
PUT EDIT('NOTSTATE=',MA,'IMP=',SNOT	PLM09210
(SKIP(2),A(9),F(5),X(2),A(4),E(18,6));	PLM09220
END;	PLM09230
END;	PLM09240
GO TO AME;	PLM09250
/* NESTED CASE */	PLM09260
EMA2: PR=CAT->PRCP.HCST;	PLM09270
TERRA=PE;	PLM09280
QT=PER.DEXTER;	PLM09290
DO IT=1 TO PER.LEAL;	PLM09300
IF PER.KIM(IT)=GOLF(I) THEN GO TO EMA3;	PLM09310
END;	PLM09320
GO TO IMK4;	PLM09330
EMA3: IF CAT->PROP.VALUE<=2 THEN EX=IT+1+PER.RAM;	PLM09340
ELSE IF (PER.RAM=1 & PER.TAR(1)=0) THEN EX=IT;	PLM09350

(continued)

```

ELSE EX=IT+PER.RAM;
CALL MINUP(EX);
IF (PROP.HOST=NUIL) THEN DO;
  PRCP.HOST->PER.REL(2)=REY*PER.REL(2)/PER.REL(1);
IF (PROP.HOST->PER.REL(2)<CUT_OFF) THEN
  PROP.HOST->PER.REL(2)=0;
  PR=PROP.HOST;
  TIERRA=PR;
  OT=PER.DEXTER;
  GO TO IMK3;
END;
ELSE PROP.REL(2)=REY*PER.REL(2)/PER.REL(1);
IF (PROP.REL(2)<CUT_OFF) THEN PROP.REL(2)=0;
  IF PROP.VALUE=1 THEN DO;
    IF (PROP.LIM=1 & PROP.TIL(1)=0) THEN GO TO AME;
    ELSE DO IT=1 TO PROP.LIM;
    STATE(2,PRCP.TIL(IT))=PROP.REL(2);
    END;
  END;
  IF PROP.VALUE=2 THEN DO;
    IF (PROP.LIM=1 & PROP.TIL(1)=0) THEN GO TO AME;
    ELSE DO IT=1 TO PROP.LIM;
    IF (PROP.REL(2)=0) THEN STATE(2,PROP.TIL(IT))=0;
    ELSE STATE(2,PROP.TIL(IT))=STATE(1,PROP.TIL(IT))*PROP.REL(2)/
      PROP.REL(1);
    IF (STATE(2,PROP.TIL(IT))<CUT_OFF) THEN STATE(2,PROP.TIL(IT))
      =0;
    END;
  END;
AME:   END;
FREE GCLD;
END;
  PUT SKIP(2) LIST('VESELY-FUSSELL IMPORTANCES');
  PUT SKIP(2) LIST('FREE EVENTS');
  IF (CUT_OFF=0) THEN GO TO ENAS;
  DO I=1 TO FUN;
  PUT SKIP DATA(I,STATE(2,I));
  IF (STATE(2,I)=0) THEN PUT SKIP LIST('CANCELED');
END;
  PUT SKIP(2) LIST('REPLICATED EVENTS');
  DO I=1 TO DUN;
  PUT SKIP DATA(I,STATE(2,I));
  IF (STATE(2,I)=0) THEN PUT SKIP LIST('CANCELED');
  END;
  PUT SKIP(2) LIST('MODULES');
  DO I=1 TO IB;
  PT=BOST(I);
  PUT EDIT('MODULE NAME=',PROP.NAME,'IMP=',PROP.REL(2))
    (SKIP(1),A(12),F(5),X(2),A(4),E(18,6));
  IF (PROP.REL(2)=0) THEN
    PUT SKIP(2) LIST('THIS MODULE AND ITS COMPONENTS ARE CANCELED');
  IF (PROP.HOST=NUIL) THEN DO;
    PUT EDIT('IMP=',PRCP.HOST->PER.REL(2))
      (X(6),A(4),E(18,6));
  IF (PROP.HOST->PER.REL(2)=0) THEN

```

PLM09360
 PLM09370
 PLM09380
 PLM09390
 PLM09400
 PLM09410
 PLM09420
 PLM09430
 PLM09440
 PLM09450
 PLM09460
 PLM09470
 PLM09480
 PLM09490
 PLM09500
 PLM09510
 PLM09520
 PLM09530
 PLM09540
 PLM09550
 PLM09560
 PLM09570
 PLM09580
 PLM09590
 PLM09600
 PLM09610
 PLM09620
 PLM09630
 PLM09640
 PLM09650
 PLM09660
 PLM09670
 PLM09680
 PLM09690
 PLM09700
 PLM09710
 PLM09720
 PLM09730
 PLM09740
 PLM09750
 PLM09760
 PLM09770
 PLM09780
 PLM09790
 PLM09800
 PLM09810
 PLM09820
 PLM09830
 PLM09840
 PLM09850
 PLM09860
 PLM09870
 PLM09880
 PLM09890
 PLM09900

(continued)

Table (7.2.7) continued

PUT SKIP(2) LIST('THIS MODULE AND ITS COMPONENTS ARE CANCELED');	PLM09910
END;	PLM09920
END;	PLM09930
ENAS: DO I=1 TO FUN;	PLM09940
PUT SKIP DATA(I,STATE(2,I));	PLM09950
END;	PLM09960
PUT SKIP(2) LIST('REPLICATED EVENTS');	PLM09970
DO I=1 TO DUN;	PLM09980
PUT SKIP DATA(I,STATE(2,I));	PLM09990
END;	PLM10000
PUT SKIP(2) LIST('MODULES');	PLM10010
DO I=1 TO IB;	PLM10020
PT=BOST(I);	PLM10030
PUT EDIT('MODULE NAME=',ERCF.NAME,'IMP=',PROP.REL(2))	PLM10040
(SKIP(1),A(12),F(5),X(2),A(4),E(18,6));	PLM10050
IF (PROP.HCST=)NULL THEN DC;	PLM10060
PUT EDIT('IMP=',PROP.HOST->PER.REL(2))	PLM10070
(X(6),A(4),E(18,6));	PLM10080
END;	PLM10090
END;	PLM10100
ASAC: END IMPORTANCE;	PLM10110
SOSOL: END NUMERC;	PLM10120

- (1) PLM06270 through PLM06330
- (2) PLM06990 through PLM07100
- (3) PLM07240 through PLM07290
- (4) PLM07330 through PLM07440
- (5) PLM08310 through PLM09250
- (6) PLM09380 through PLM09650

7.3 Organization of the Procedures used in PL-MODMC

PL-MODMC is a computer program to perform Monte-Carlo analysis of complex fault trees. In addition, this code has the capability, if requested, to calculate the point unavailability or the average unavailability of the top event over a specific period of time. This code is constructed in two separate steps. In the first step, the restriction of not allowing a replicated component to appear in the domains of two or more replicated gates or the top event is removed. In the second step, a Monte-Carlo package is added to the code in order to perform a Monte-Carlo simulation of the top event occurrence probability. In this section, some of the logical and computer programming concepts that are implemented are discussed.

7.3.1 Removal of the Restriction of Component Replication in Different Domains of Fault Trees

This restriction has been removed by temporarily disconnecting the dependences between the replicated gates and the top event, caused by the presence of identical replicated components in their domains. Components which have a complement in the tree have been widely used to perform the disconnection process. Basically, two distinct procedures

perform all the logical changes for handling this extension. They are as follows;

- (1) TREE-IN Procedure;
- (2) TOP-SET Procedure.

The TREE-IN procedure existed already in the PL-MOD code, but it has been completely changed here in order to perform the disconnection process prior to the modularization process. The TOP-SET procedure is used, after the modularization of the fault tree, to properly treat the effect of disconnection that has been performed by the TREE-IN procedure.

To clarify the calculations and treatments performed in the TREE-IN procedure, the fault tree shown in Figure (7.3.1) will be studied. This fault tree consists of seven components where components C_2 , C_3 and C_5 are replicated, and have appeared in the domains of replicated gate G7 and the top event G1. A preprocessor assigns proper numerical values to the gates and components of the fault tree, such that the input becomes acceptable to the old version of the PL-MODMC code. Except for the components that are replicated, the preprocessor assigns regular numbers (instead of numbers in the form of, say, 30005). However, the same number will be assigned to other places where this replicated component appears. Therefore, Table (7.3.1) shows the numeric values that are assigned through the preprocessor to the alphanumeric names shown in Figure (7.3.1)

Figure 7.3.1

FAULT TREE FOR A MONTE-CARLO ANALYSIS

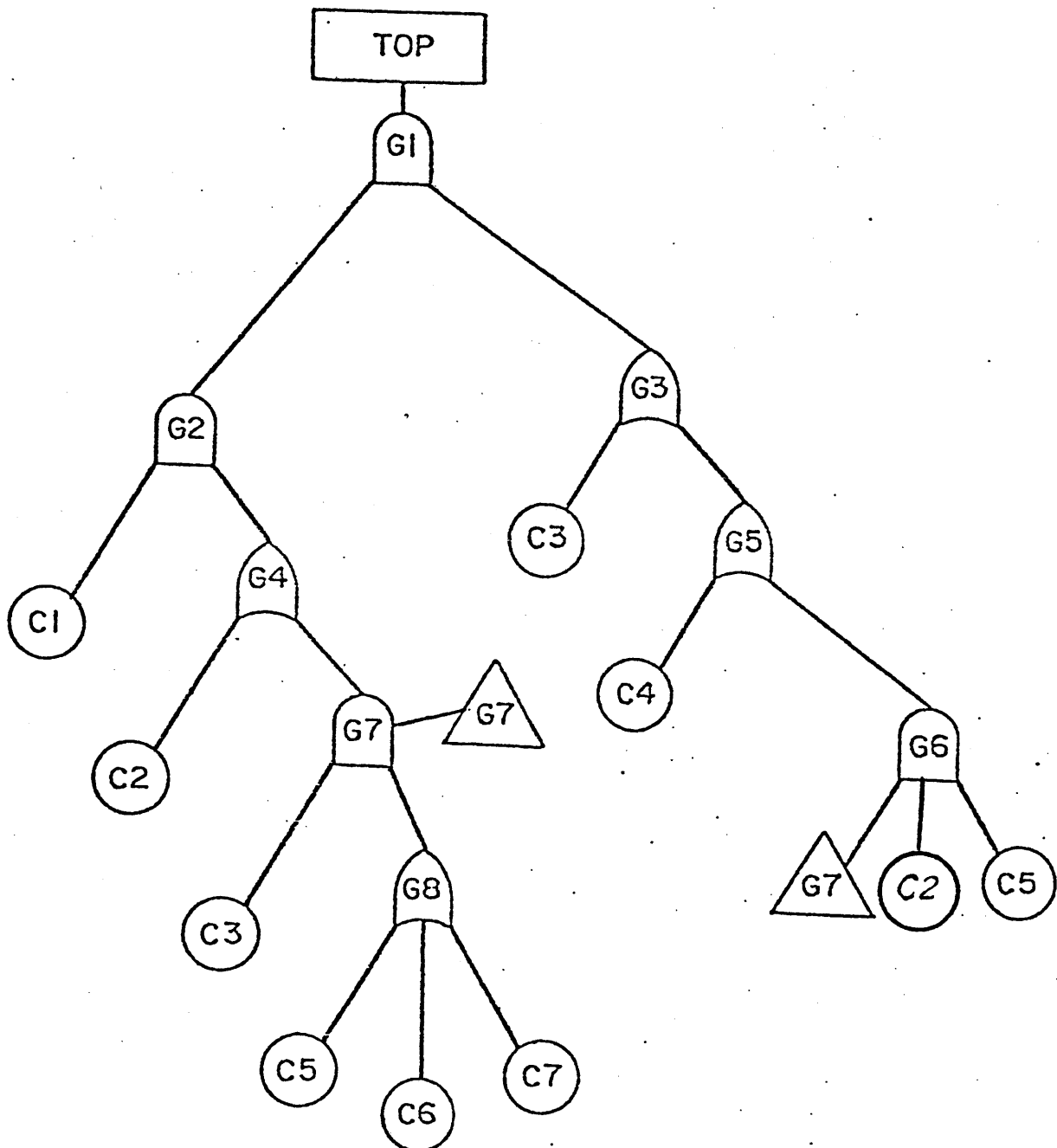


TABLE 7.3.1

Corresponding Numeric Values for the
Components and Gates of the Fault Tree
in Figure (7.3.1)

<u>Component Name</u>	<u>Corresponding Numeric Value</u>	<u>Gate Name</u>	<u>Corresponding Numeric Value</u>
C1	1	G1 (TOP)	1
C2	2	G2	2
C3	3	G3	3
C4	4	G4	4
C5	5	G5	5
C6	6	G6	6
C7	7	G7	7
		G8	8

At the end, the preprocessor transfers these numeric values of the nodes and components of the tree, to the controlled NODEIN structure. A NODE based structure would be allocated for each gate of the tree to transfer all information with regard to the nodes of the fault tree. The composition of the NODEIN and NODE structures are the same as in the PL-MOD, and the details of these structures are clearly discussed in Reference [1]. All the pointers identifying the NODE structure are stored in the SPINE array [1]. This array will be used in the TREE-IN procedure to refer to the NODE structure for changing some of the values that are modified during the disconnection process.

The TREE-IN procedure starts by assigning name, value, gate inputs, and component inputs to a NODE structure from the NODEIN structure. Then the following DO loop assign NODE.ROOT and NODE.SPIT values of the gates to the NODE structure in question (see Reference [1]);

```
DO L=1 TO GINO;
  NODE.SPIT(L) = SPINE(NODEIN.PIT(I,L));
  AT = NODE.SPIT(L);
  AT -> NODE.ROOT = NT;
END;
```

GINO indicates the number of gates input to the NODE structure with the pointer NT, and NODE.SPIT(1) stores pointers NT of the input gate's NODE structure.

Arrays BOOK and NPOINT are allocated and have the following declaration form:

```

DECLARE BOOK(NRG,NCOMP) FIXED BINARY CONTROLLED;
DECLARE NPOINT(NRG,NCOMP,15) POINTER CONTROLLED;
DECLARE GATNO (GUM)POINTER CONTROLLED;

```

NRG and NCOMP are the number of replicated gates and the top gate, and number of components in the fault tree, respectively. The BOOK array will store the number of times that a component appears in each domain, and the NPOINT array stores the NT pointers for each appearance. GATNO is an auxiliary array to store NT pointers during the counting of the number of appearances of the components in each domain. Table (7.3.2) summarizes a set of statements in the TREE-IN procedure which performs the counting of the components in different domains of a fault tree. For the fault tree in Figure (7.3.1) the following values will be assigned by the set of statements shown in Table (7.3.2)

$$\text{NRG} = 1 + 1 = 2$$

(there are two main domains in this fault tree: one is the domain of the top event TOP and one is the domain of the replicated gate G1)

$$\text{BOOK} = \begin{matrix} & \begin{pmatrix} 1 & 2 & 1 & 1 & 1 & 0 & 0 \end{pmatrix} \\ \begin{pmatrix} 0 & 0 & 1 & 0 & 1 & 1 & 1 \end{pmatrix} & \\ \begin{matrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{matrix} & \end{matrix}$$

The first row shows the appearances of each component in the TOP domain and the second row shows the appearance of the components in the gate G7. Therefore, from the BOOK array it is clear that components 2, 3 and 5

Table(7.3.2) Set of Statements to Count Replicated Components in Each Domain

NRG=RMOD+1;	WHO06220
DCOR=GUM;	WHO06230
ALLCCATE BOOK;	WHO06240
ALLOCATE NPOINT;	WHC06250
BOOK=0;	WHO06260
NFCINT=NULL;	WHC06270
ALLOCATE GATNO;	WHO06280
GATNO=NULL;	WHC06290
MANO=0;	WHO06300
GNO=0;	WHO06310
TREE=NULL;	WHO06320
AT=SPINE(TCFENO);	WHO06330
AGAIN: IF(AT->NODE.GINT/=0) THEN DO;	WHC06340
GINO=AT->NCDE.GINT;	WHO06350
DO L=1 TO GINO;	WHC06360
MANO=MANO+1;	WHO06370
GATNO(MANO)=AT->NODE.SPIT(L);	WHC06380
END;	WHO06390
END;	WHO06400
MCOM=AT->NODE.LIL;	WHO06410
DC M=1 TO MCOM;	WHC06420
MISH=AT->NODE.TIL(M);	WHO06430
IF(MISH=0) THEN GO TO FINE;	WHO06440
BCKK(1,MISH)=BOOK(1,MISH)+1;	WHO06450
N=BOOK(1,MISH);	WHO06460
NPCINT(1,MISH,N)=AT;	WHO06470
FINE: END;	WHO06480
GNO=GNO+1;	WHO06490
AT=GATNO(GNO);	WHO06500
IF(AT=NULL) THEN GO TO AGAIN;	WHC06510
DO IX=1 TO RMOD;	WHO06520
GATNO=NULL;	WHO06530
GNO=0;	WHO06540
MANO=0;	WHO06550
RG=TRIN(IX);	WHO06560
AT=SPINE(RG);	WHO06570
BGAIN: IF(AT->NCDE.GINT/=0) THEN GO;	WHO06580
GINO=AT->NCDE.GINT;	WHC06590
DC L=1 TO GINO;	WHO06600
MANO=MANO+1;	WHO06610
GATNO(MANO)=AT->NCDE.SPIT(L);	WHO06620
END;	WHO06630
END;	WHO06640
MCCM=AT->NODE.LIL;	WHO06650
DO M=1 TO MCCM;	WHO06660
MISH=AT->NODE.TIL(M);	WHO06670
IF(MISH=0) THEN GO TO ANZ;	WHO06680
BOOK(IX+1,MISH)=BOOK(IX+1,MISH)+1;	WHO06690
N=BOOK(IX+1,MISH);	WHO06700
NPOINT(IX+1,MISH,N)=AT;	WHO06710
ANZ: END;	WHO06720
GNO=GNO+1;	WHO06730
AT=GATNO(GNO);	WHO06740
IF(AT=NULL) THEN GO TO BGAIN;	WHO06750
END;	WHO06760
FREE GATNO;	WHO06770

are replicated. Component 2 is repeated twice in the domain of the top event, and components 3 and 5 are repeated once in each TOP and G7 domain. It is clear that components 3 and 5 cannot be handled by the old PL-MOD code. To remove the dependences between gates TOP and G7 one can use, temporarily, different numbers for the components 3 and 5 in each domain. Particularly, any replicated component that is repeated only once in a specific domain can be treated as an ON component. However, an OFF pair will be temporarily assigned to all ON components without an OFF pair. The process of adding this OFF pair is more fully discussed later in this section. The NPOINT assay stores the following pointers:

$$\text{NPOINT} = \begin{pmatrix} \text{NT}_2, \dots, \text{Null} & \text{NT}_4, \text{NT}_6, \dots, \text{Null} & \text{NT}_3, \dots, \text{Null} & \text{NT}_5, \dots, \text{Null} \\ & \text{NT}_6, \dots, \text{Null} & \text{Null}, \dots, \text{Null} & \text{Null}, \dots, \text{Null} \\ \text{Null}, \dots, \text{Null} & \text{Null}, \dots, \text{Null} & \text{NT}_7, \dots, \text{Null} & \text{Null}, \dots, \text{Null} \\ & \text{NT}_8, \dots, \text{Null} & \text{NT}_8, \dots, \text{Null} & \text{NT}_8, \dots, \text{Null} \end{pmatrix}$$

For example, NT_4 and NT_6 indicate the NODE pointers for gates numbers 4 and 6 to which component 2 is attached. Proper numbers for the replicated components 2, 3 and 5 for disconnecting dependences between the gates TOP and G7 are as follows

$$\text{BOOK} = \begin{pmatrix} 1 & 20001 & 21002 & 1 & 21004 & 0 & 0 \\ 0 & 0 & 21003 & 0 & 21005 & 1 & 1 \end{pmatrix}$$

Where, for instance, 21002 means that there is a pair of ON and OFF replicated components in which the ON part is identified by 21002 and the OFF part has a value of 22002. The OFF pairs of all temporary ON components must be attached to a new gate added to the main event domain (i.e. replicated event domain or the top event domain) with a gate type opposite to the previous main event. For example, gate G7, which is an OR gate, will be input to a new OR gate with all of the OFF temporary components in the G7 domain input to this new gate. Therefore, the fault tree in Figure (7.3.1) will be changed to the form illustrated in Figure (7.3.2). This fault tree is clearly an independent one which can be handled by the PL-MOD Code.

To account for the changes in the structure of the tree and by using the data in arrays BOOK and NPOINT, those NODE structures that are not reflecting the new form of the nodes of the tree must be changed (e.g. gates 4, 7, 8, 3 and 6 must be accounted for in the new replicated event inputs). Also, a new NODE structure must be allocated for the gates added to the top and replicated events with added OFF components in their domains (e.g. nodes 9 and 10). The set of statements in the TREE-IN procedure to handle the changes in the NODE structure data are shown in Table (7.3.3).

To keep track of all these changes a based structure named NOT and an array BKEEP are used. The NOT structure and BKEEP arrays are declared as follows:

```

DECLARE 1 NOT BASED (KT),
        2 DIR FIXED BINARY,
        2 TIR (LIRO REFER (NOT.DIR)) FIXED BINARY;
```

Figure (7.3.2)

New Form of the Fault Tree Given in Figure (7.3.1)

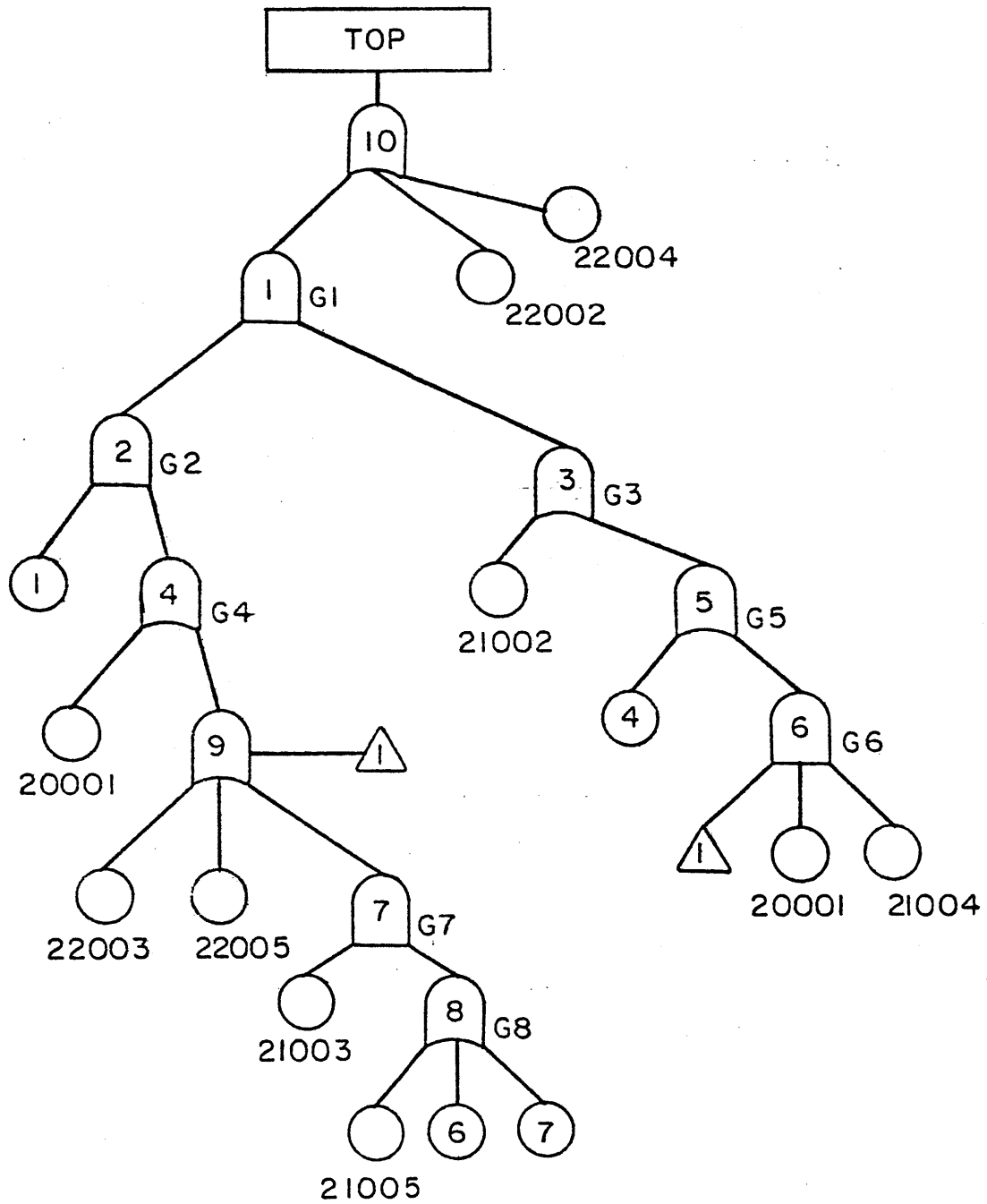


Table (7.3.3) Set of Statements to Make Replicated Gates and the
Top Event Independent

ALLOCATE CCMPNO;	WHO06780
NMANG=EMOD+NNOT+1;	WHO06790
NIA=0;	WHO06800
ALLOCATE GATNA;	WHO06810
GATNA=NULL;	WHO06820
ALLOCATE EAM;	WHO06830
BAM=0;	WHO06840
ALLOCATE NODA;	WHO06850
NODA=0;	WHO06860
DO J=1 TO NCCMP;	WHO06870
COUNT=0;	WHO06880
DO I=1 TO NRG;	WHO06890
COUNT=CCOUNT+BOOK(I,J);	WHO06900
END;	WHO06910
IF(COUNT<=1) THEN GO TO FATM;	WHO06920
NIA=NIA+1;	WHO06930
CCMPNO(NIA)=J;	WHO06940
DO K=1 TO NRG;	WHO06950
IF(BOOK(K,J)~=1) THEN GO TO GOOD;	WHO06960
BAM(K)=BAM(K)+1;	WHO06970
BOOK(K,J)=21000+NMANG;	WHO06980
NMANG=NMANG+1;	WHO06990
ZAP=NPOINT(K,J,1);	WHO07000
LILO=ZAP->NODE.LII-1;	WHO07010
IF(LILO=0) THEN LILO=1;	WHO07020
GINC=ZAP->NODE.GIN;	WHO07030
IF(ZAP->NODE.TIR(1)~=?) THEN	WHO07040
LILO=ZAP->NODE.LIR+1;	WHO07050
ELSE LILO=1;	WHO07060
ALLOCATE NCDE;	WHO07070
LILO=ZAP->NODE.LII;	WHO07080
DO M=1 TO LILO;	WHO07090
KAP=ZAP;	WHO07100
MASH=0;	WHO07110
MISH=ZAP->NODE.TIL(M);	WHO07120
DO WHILE(KAP~=NULL);	WHO07130
MASH=MASH+1;	WHO07140
KAP=NPOINT(K,MISH,MASH);	WHO07150

Continued

Table (7.3.3) continued

```

IF (KAP=ZAP) THEN NPCINT(K, MISH, MASH)=NT;
END;
END;
LAMB=ZAP->NODE.LIR;
DO M=1 TO LAMB;
  NODE.TIR(M)=ZAP->NODE.TIR(M);
END;
NODE.TIR(LIRO)=BOOK(K, J);
L=0;
JAP=ZAP->NODE.LIL;
DO M=1 TO JAP;
  IF (J=ZAP->NODE.TIL(M)) THEN DC;
  L=L+1;
  NODE.TIL(L)=ZAP->NODE.TIL(M);
END;
END;
NODE.TIPC=1;
NODE.NAME=ZAP->NODE.NAME;
NODE.VALUE=ZAP->NODE.VALUE;
NODE.GINT=ZAP->NODE.GINT;
NODE.LILT=ZAP->NODE.LILT-1;
NODE.LIRT=ZAP->NODE.LIRT+1;
NODE.LIMD=ZAP->NODE.LIMD;
NODE.LIMT=ZAP->NODE.LIMT;
NODE.NEST='';
NODE.WHIZ=0;
NODE.ROOT=ZAP->NODE.ROOT;
NODE.LIP=NULL;
NODE.LID=NULL;
NODE.NAIL=NULL;
NODE.WHIP=NULL;
  GINO=ZAP->NODE.GINT;
DO M=1 TO GINO;
  AT=ZAP->NODE.SPIT(M);
  NODE.SPIT(M)=AT;
  AT->NODE.ROOT=NT;
END;
LAMB=ZAP->NODE.NAME;
SPINE(LAMB)=NT;
AT=ZAP->NODE.ROOT;
LAMB=AT->NODE.GINT;
IF (AT=NULL) THEN
  DO M=1 TO LAMB;
    IF (ZAP=AT->NODE.SPIT(M)) THEN
      AT->NODE.SPIT(M)=NT;
    END;
  FREE ZAP->NODE;
  NODA(K, BAM(K))=22000+NMANG-1;
  GO TO FATN;
GOOD: IF (BOOK(K, J)<=1) THEN GO TO FATN;
COUNT=BOOK(K, J);
BOOK(K, J)=BOOK(K, J)*1000+NMANG;
NMANG=NMANG+1;
DO N=1 TO COUNT;
  ZAP=NPOINT(K, J, N);

```

(continued)

WHO07160
 WHO07170
 WHO07180
 WHO07190
 WHO07200
 WHO07210
 WHO07220
 WHO07230
 WHO07240
 WHO07250
 WHO07260
 WHO07270
 WHO07280
 WHO07290
 WHO07300
 WHO07310
 WHO07320
 WHO07330
 WHO07340
 WHO07350
 WHO07360
 WHO07370
 WHO07380
 WHO07390
 WHO07400
 WHO07410
 WHO07420
 WHO07430
 WHO07440
 WHO07450
 WHO07460
 WHO07470
 WHO07480
 WHO07490
 WHO07500
 WHO07510
 WHO07520
 WHO07530
 WHO07540
 WHO07550
 WHO07560
 WHO07570
 WHO07580
 WHO07590
 WHO07600
 WHO07610
 WHO07620
 WHO07630
 WHO07640
 WHO07650
 WHO07660
 WHO07670
 WHO07680
 WHO07690
 WHO07700

Table (7.3.3)(continued)

```

LILC=ZAP->NODE.LIL-1;
IF (LILC=0) THEN LILC=1;
GINC=ZAP->NODE.GIN;
IF (ZAP->NODE.TIR(1) /= 0) THEN
  LIRC=ZAP->NODE.LIR+1;
  ELSE LIRC=1;
ALLOCATE NCDE;
LILC=ZAP->NODE.LIL;
DO M=1 TO LILC;
  MASH=0;
  KAP=ZAP;
  MISH=ZAP->NODE.TIL(M);
  DO WHILE (KAP /= NULL);
    MASH=MASH+1;
    KAP=NPOINT(K,MISH,MASH);
    IF (KAP=ZAP) THEN NPOINT(K,MISH,MASH)=NT;
  END;
  LAMB=ZAP->NODE.DIR;
  DO M=1 TO LAMB;
    NODE.TIR(M)=ZAP->NODE.TIR(M);
  END;
  NODE.TIR(LIRC)=BOOK(K,J);
  L=0;
  JAPE=ZAP->NODE.LIL;
  DO M=1 TO JAPE;
    IF (J /= ZAP->NODE.TIL(M)) THEN DO;
      I=I+1;
      NODE.TIL(I)=ZAP->NODE.TIL(M);
    END;
  END;
  NODE.TIPO=1;
  NODE.NAME=ZAP->NODE.NAME;
  NODE.VALUE=ZAP->NODE.VALUE;
  NODE.GINT=ZAP->NODE.GINT;
  NODE.LILT=ZAP->NODE.LILT-1;
  NODE.LIRT=ZAP->NODE.LIRT+1;
  NODE.LIMD=ZAP->NODE.LIMD;
  NODE.LIMT=ZAP->NODE.LIMT;
  NODE.NEST=0;
  NODE.WHIZ=0;
  NODE.ROOT=ZAP->NODE.ROOT;
  NODE.LIP=NULL;
  NODE.LID=NULL;
  NODE.NAIL=NULL;
  NODE.WHIP=NULL;
  GINO=ZAP->NODE.GINT;
  DO M=1 TO GINO;
    AT=ZAP->NODE.SPIT(M);
    NODE.SPIT(M)=AT;
    AT->NODE.FOOT=NT;
  END;
  LAMB=ZAP->NODE.NAME;
  SPINE(LAMB)=NT;
  AT=ZAP->NODE.ROOT;

```

(continued)

```

WHO07710
WHO07720
WHC07730
WHO07740
WHC07750
WHO07760
WHC07770
WHO07780
WHO07790
WHC07800
WHO07810
WHO07820
WHO07830
WHO07840
WHO07850
WHO07860
WHO07870
WHC07880
WHO07890
WHC07900
WHO07910
WHC07920
WHO07930
WHO07940
WHO07950
WHO07960
WHC07970
WHO07980
WHO07990
WHO08000
WHO08010
WHO08020
WHO08030
WHO08040
WHO08050
WHC08060
WHO08070
WHC08080
WHO08090
WHC08100
WHO08110
WHO08120
WHO08130
WHO08140
WHC08150
WHO08160
WHC08170
WHO08180
WHO08190
WHO08200
WHO08210
WHO08220
WHO08230
WHC08240
WHO08250

```

Table (7.3.3) (continued)

IAMB=AT->NODE.GINT;	WHO08260
IF(AT=)NULL) THEN	WHC08270
DO M=1 TC IAMB;	WHO08280
IF(ZAP=AT->NODE.SPIT(M)) THEN	WHC08290
AT->NODE.SPIT(M)=NT;	WHO08300
END;	WHC08310
FREE ZAP->NODE;	WHO08320
END;	WHO08330
FATN: END;	WHO08340
FATH: END;	WHO08350
DC K=1 TC NRG;	WHO08360
IF(NODA(K,1)=) THEN GO TO BALLY;	WHO08370
LIRC=BAM(K);	WHO08380
GINO=1;	WHO08390
LILQ=1;	WHC08400
ALLOCATE NODE;	WHO08410
IF(K=1) THEN GO TO FAMN;	WHO08420
GATNA(1)=NT;	WHO08430
NODE.ROOT=NULL;	WHO08440
NODE.TIL=0;	WHO08450
NODE.WHIP=NULL;	WHO08460
NODE.NAIL=NULL;	WHO08470
NODE.WHIZ=0;	WHO08480
NODE.LIRT=LIRO;	WHO08490
NODE.LILT=0;	WHO08500
NODE.LIMT=0;	WHC08510
NODE.LIME=0;	WHO08520
NODE.LIP=NULL;	WHO08530
NODE.LIE=NULL;	WHO08540
GUM=GUM+1;	WHO08550
NODE.NAME=GUM;	WHC08560
NODE.GINT=1;	WHO08570
NODE.TIPC=1;	WHC08580
NODE.NEST=0;	WHO08590
AT=SPINE(TOPENO);	WHO08600
IF(AT->NODE.VALUE=1) THEN NODE.VALUE=2;	WHO08610
ELSE NODE.VALUE=1;	WHO08620
AT->NODE.ROOT=NT;	WHO08630
NODE.SPIT(1)=SPINE(TOPENO);	WHO08640
TREE=NT;	WHO08650
DO I=1 TO BAM(K);	WHC08660
NODE.TIR(I)=NODA(K,I);	WHO08670
END;	WHC08680
GO TO BALLY;	WHO08690
FAMN: GATNA(K)=NT;	WHO08700
NODE.TIL=0;	WHO08710
NODE.WHIP=NULL;	WHO08720
NODE.NAIL=NULL;	WHC08730
NODE.WHIZ=0;	WHO08740
NODE.ROOT=NULL;	WHO08750
NODE.LIRT=LIRO;	WHO08760
NODE.LILT=0;	WHO08770
NODE.LIMT=0;	WHO08780
NODE.LIME=0;	WHO08790
NODE.LIP=NULL;	WHC08800

(continued)

Table (7.3.3) continued

NODE.LID=NULL;	WHO03810
GUM=GUM+1;	WHO03820
NCDE.NAME=GUM;	WHO08830
NODE.GINT=1;	WHO08840
NODE.TIPC=1;	WHC08850
NODE.NEST=0;	WHO08860
RG=TRIN(K-1);	WHO08870
PRIM(K-1)=NT;	WHO08880
TRIN(K-1)=GUM;	WHO08890
SPINA(K-1)=NT;	WHO08900
AT=SPINE(RG);	WHO08910
IF(AT->NODE.VALUE=1) THEN NCDE.VALUE=2;	WHO08920
ELSE NODE.VALUE=1;	WHO08930
AT->NODE.ROOT=NT;	WHO08940
NODE.SPIT(1)=SPINE(RG);	WHO08950
DO I=1 TC BAM(K);	WHC08960
NODE.TIR(I)=NODA(K,I);	WHO08970
END;	WHC08980
BALY: END;	WHO08990
FREE NODA;	WHC09000
FREE BAM;	WHO09010
FREE NPOINT;	WHO09020
NRG=NRG+1;	WHO09030
ALLOCATE BKEEP;	WHO09040
BKEEP=0;	WHO09050
DO J=1 TC NIA;	WHO09060
BKEEP(1,J)=CCMPNO(J);	WHO09070
N=CCMPNO(J);	WHC09080
DO I=2 TC NRG;	WHC09090
BKEEP(I,J)=BOOK(I-1,N);	WHO09100
END;	WHC09110
END;	WHO09120
FREE BOOK;	WHO09130
FREE COMPNO;	WHO09140
NRG=NRG-1;	WHO09150
ALLOCATE ADDNOT;	WHO09160
ADDNOT=NULL;	WHO09170
DO I=1 TC NEG;	WHC09180
IF(GATNA(I)~=NULL) THEN DO;	WHO09190
LIRO=GATNA(I)->NODE.DIR;	WHC09200
ALLOCATE NOT;	WHO09210
ADDNOT(I)=KT;	WHO09220
DO J=1 TC LIRO;	WHO09230
NOT.TIR(J)=GATNA(I)->NCDE.TIR(J);	WHO09240
END;	WHO09250
END;	WHO09260
END;	WHC09270
FREE GATNA;	WHO09280
NOR=NMA NG-1;	WHO09290

DECLARE BKEEP (NRG,NIA) FIXED BINARY CONTROLLED EXTERNAL;

The NOT structure stores the OFF components presented in the new gate added to the top and replicated gates. For example, in the fault tree of Figure (7.3.2) the following values are assigned to the NOT structure for gate (9):

NOT.TIR(1) = 22003

NOT.TIR(2) = 22005

The BKEEP array has one more row than the BOOK array, and its number of columns is equal to the number of replicated events in the original tree. In this example, the number of rows is 3 and the number of columns is also 3. Therefore, the BKEEP would store the following values for the fault tree:

$$\text{BKEEP} = \begin{pmatrix} 2 & 3 & 5 \\ 20001 & 21002 & 21004 \\ 0 & 21003 & 21005 \end{pmatrix}$$

The BKEEP array and NOT structure will be used in the TOP-SET procedure to identify the replicated components that are identical, for the proper treatment of the disconnection process performed in the procedure TREE-IN. The rest of the calculations in the TREE-IN are very similar to the PL-MOD code as discussed in Reference [1].

The TOP-SET procedure is used to treat the effect of changes made in the structural organization of the fault tree during the procedure TREE-IN. Procedure TOP-SET is called whenever an improper module of

the fault tree is generated. In the TOP-SET procedure all replicated modules are replaced by their own modular cutsets in the improper module. For example, when the top event of the fault tree shown in Figure (7.3.2) is reached, the modular cutsets of the replicated gate G7 that is previously calculated will be combined with the modular minimal cutsets of the top event, in order to find a unique set of modular minimal cutsets for the top event. Replicated events 20001, 21002, 21003, 21004, 21005, 22002, 22003, 22004 and 22005 are obviously replicated components appearing in the set of modular cutsets. Three steps will treat the effect of making the fault tree independent:

- (1) By using the NOT structure. In this case all of the bits corresponding to the added imaginary OFF components must be set equal to zero (i.e. bits corresponding to components 22002, 22003, 22004 and 22005).
- (2) By using the BKEEP array. In this case the corresponding values for replicated components should be set equal to their real values that are given in the first row of the BKEEP array (e.g. change 20001 to 2, 21003 and 21003 to 3, and 21005 and 21004 to 5).
- (3) Due to the replication of one component in a modular cutset (e.g. components 3 and 5), the bits corresponding to it must be ORed with the first appearance of the component in each modular cutset. The rest of the bits should be set equal to zero after the component's first

appearance in the cutset.

The method of combining the modular cutsets is similar to the method of generating simple cutsets in the MODCUT code. Similarly, Step (3) for treating the effect of replications in the improper modular cutsets is identical to the method utilized by the MODCUT for the removal of this effect. Due to the more general method used in the TOP-SET procedure in the MODCUT code, the reader is referred to Section 7.3 of this thesis for a more detailed discussion.

Two important structures exist in the TOP-SET procedure of the PL-MODMC; one is the MISS based structure and the other is the MODCUT based structure. These structures have the following compositions:

```

DECLARE 1 MISS BASED (ALM),
        2 MOS FIXED BINARY,
        2 SIN FIXED BINARY,
        2 SON FIXED BINARY,
        2 MIN FIXED BINARY,
        2 LIL (NUB REFER (MESS.SON)),
        2 AOMP(JUMP REFER (MISS.MOS))
        BIT (LARG REFER (MISS.SIN)),
        2 TIL (WEST REFER (MISS.MIN));

DECLARE 1 MODCUT BASED (RA),
        2 MOS FIXED BINARY,
        2 SIN FIXED BINARY,
        2 AOMP (MOSE REFER (MODCUT.MOS))
        BIT (SINE REFER (MODCUT.SIN));

```

The MISS structure is used to store the information in the PER structure and the modular minimal cutsets after the TOP-SET is called for an improper module. The MODCUT structure is used to store a combination of the modular cutsets, if any, due to the presence of replicated modules in the modular cutsets of the improper module. The set of statements for the TOP-SET procedure which transfers data from the PER structure and modular minimal cutsets of an improper module to the MISS structure is shown in Table (7.3.4).

To remove the OFF complementary components by using the NOT structure, the list of components in the TOP-SET procedure to perform this task is shown in Table (7.3.5).

The set of statements to find the simple cutsets from the modular minimal cutsets by inserting the cutsets of all modules in modular minimal cutsets is listed in Table (7.3.6).

Finding minimal modular cutsets, and treating the replication of a component in the resulting combination of modular minimal cutsets is similar to the procedures TOP-SET and CUT-SET of the MODCUT code which will be extensively discussed in the next section. However, it should be noted that if the replication of a component in the set of modular cutsets does not exist then the set of combined modular cutsets is minimal and there would be no need for further minimization. Because if there are no identical components in two or more sets of modular minimal cutsets, the combination of these minimal cutsets is still minimal. For example, if components 5 and 3 were not replicated in both domains of the top event and replicated gate G7, then the final set of modular minimal cutsets obtained by combining the modular minimal cutsets of TOP and G7 modules is still minimal.

Table (7.3.4) Set of Statements to Transfer PER Structure Data and Minimal Modular Cutsets to MISS Structure 246

```

MARK=1;
X=20;
  IF (DEL=1) THEN DC;
    ALLOCATE BABA;
    ALLOCATE JAJA;
    ALLOCATE DADA;
  DO I=1 TO 20;
    BABA(I)=NULL;
    JAJA(I)=NULL;
    DADA(I)=0;
  END;
  END;
  DEL=DEL+1;
  IF (DEL>2) THEN DEL=2;
  I=1;
  VIT=BABA(I);
  DO WHILE (VIT<=>NULL);
    I=I+1;
    VIT=BABA(I);
  END;
  BABA(I)=PR;
  JAJA(I)=PT;
  DADA(I)=STORK->PROP.NAME;
  DMISH=PER.RAM;
  JUMP=0;
  PER.DEXTER=STORK;
  VIT=PER.HECTOR;
  DO WHILE (VIT<=>NULL);
    VIC=VIT;
    VIT=VIC->FLOOR;
    JUMP=JUMP+1;
  END;
  NUD=PER.RAM;
  WEST=PER.LEAL;
  VIT=PER.HECTOR;
  LARG=VIT->VECTOR.LORG;
  IF (PROP.VALUE<=2) THEN
    WEST=WEST+1;
  ALLOCATE MISS;
  MISS.LIL=PER.TAR;
  IF (PROP.VALUE<=2) THEN
    MISS.TIL(1)=STORK->PROP.NAME;
  DO I=2 TO WEST;
    IF (PROP.VALUE<=2) THEN
      J=I-1;
      ELSE J=I;
    MISS.TIL(I)=PER.JIM(J);
  END;
  N=0;
  DO WHILE (VIT<=>NULL);
    VIC=VIT;
    N=N+1;
    MISS.AOMP(N)=VIC->VECTOR.CGMP;
    VIT=VIC->FLOOR;
  END;
  JOSA(STORK->PROP.NAME)=ALM;

```

WHO23340
 WHC23350
 WHO23360
 WHO23370
 WHO23380
 WHO23390
 WHO23400
 WHO23410
 WHC23420
 WHO23430
 WHC23440
 WHO23450
 WHO23460
 WHO23470
 WHO23480
 WHO23490
 WHO23500
 WHC23510
 WHO23520
 WHO23530
 WHO23540
 WHO23550
 WHC23560
 WHO23570
 WHC23580
 WHO23590
 WHC23600
 WHO23610
 WHO23620
 WHO23630
 WHO23640
 WHO23650
 WHO23660
 WHO23670
 WHO23680
 WHO23690
 WHO23700
 WHO23710
 WHO23720
 WHO23730
 WHC23740
 WHO23750
 WHC23760
 WHO23770
 WHC23780
 WHO23790
 WHO23800
 WHO23810
 WHO23820
 WHC23830
 WHO23840
 WHC23850
 WHO23860
 WHC23870
 WHO23880
 WHO23890

Table (7.3.5) Removal of OFF Complementary Components and Treatment of Replications

K=0;	WHO23900
DC N=1 TO NFG;	WHC23910
IF (ADDNOT (N) -> NULL) THEN IO;	WHC23920
AMISH=ADDNCT(N) -> NCT.DIR;	WHO23930
DO I=1 TO AMISH;	WHO23940
MA=ADDNCT(N) -> NOT.TIE(I);	WHO23950
DO J=1 TO BMISH;	WHO23960
IF (MISS.LIL(J) = MA) THEN	WHO23970
DO M=1 TO MISS.MOS;	WHO23980
SUBSTR(MISS.ACMP(M), J, 1) = '0'B;	WHO23990
K=1;	WHC24000
END;	WHO24010
END;	WHO24020
END;	WHO24030
END;	WHO24040
JUMP1=JUMP;	WHC24050
IF (K=1) THEN DO I=1 TO MISS.MOS;	WHO24060
IF MISS.AOMP(I) = REPEAT('0'B, LARG-1) THEN JUMP1=JUMP1-1;	WHC24070
END;	WHO24080
IF (JUMP1 < JUMP) THEN DO;	WHO24090
JUMP=JUMP1;	WHO24100
ALLOCATE MISS;	WHO24110
VIC=JCSA (STCRK -> PROP.NAME);	WHO24120
JUMP1=VIC -> MISS.SIN;	WHO24130
JUMP=VIC -> MISS.MOS;	WHO24140
MISS.LIL=VIC -> MISS.LIL;	WHO24150
MISS.TIL=VIC -> MISS.TIL;	WHC24160
J=1;	WHO24170
DO M=1 TO JUMP;	WHC24180
IF VIC -> MISS.ACMP(M) -> REPEAT('0'B, JUMP1-1) THEN DO;	WHO24190
MISS.AOMP(J) = VIC -> MISS.AOMP(M);	WHO24200
J=J+1;	WHC24210
END;	WHO24220
END;	WHO24230
JCSA (STCRK -> PROP.NAME) = ALM;	WHO24240
FREE VIC -> MISS;	WHO24250
END;	WHO24260
	WHO24270

Table (7.3.6) Set of Statements to Combine Modular Minimal Cutsets

```

      JAM=ALM;
DO I=1 TO BMISH;
  MA=PER.TAR(I);
  DA=-CEIL (-MA/10000);
  JA=-CEIL (-MA/1000);
  IF ((JA-10*DA)=9) THEN MARK=C;
END;
IF (MARK=1) THEN GO TO SKAP;
MOSE=1;
AMISH=MISS.MOS;
BMISH=MISS.SON;
DO J=1 TO AMISH;
  JEST=1;
  MISH=0;
  MASH=0;
  DO I=1 TO BMISH;
    MA=MISS.LIL(I);
    DA=-CEIL (-MA/10000);
    JA=-CEIL (-MA/1000);
    IF ((JA-10*DA)=9) THEN DC;
    DO IX=1 TO RMOD;
      IF (MA=TRIM(IX)) THEN SAM=TRIM(IX);
    END;
    IF (SUBSTR(MISS.ACOMP(J),I,1)='1'E) THEN DO;
      MISH=JOSA(SAM)->MISS.SIN;
      IF (J=1) THEN SINE=MISH+MASH;
      AKME=JOSA(SAM)->MISS.MOS;
      JEST=JEST*AKME;
      MASH=MISH+MASH;
    END;
    ELSE IF (J=1) THEN DC;
    JEST=MOSE;
    MISH=JOSA(SAM)->MISS.SIN;
    SINE=MISH+MASH;
    MASH=MISH+MASH;
  END;
  END;
  ELSE IF (J=1) THEN DC;
  JEST=MOSE;
  SINE=MASH+1;
  MASH=MASH+1;
END;
END;
IF (J=1) THEN SINE=MASH+MISS.MIN;
  IF (J=1 & JEST=1) THEN MCSE=1;
ELSE MOSE=MOSE+JEST;
END;
EH=0;

```

```

WHC24280
WHO24290
WHC24300
WHO24310
WHO24320
WHO24330
WHO24340
WHO24350
WHO24360
WHO24370
WHO24380
WHO24390
WHO24400
WHO24410
WHO24420
WHO24430
WHC24440
WHO24450
WHC24460
WHO24470
WHC24480
WHO24490
WHO24500
WHO24510
WHO24520
WHC24530
WHO24540
WHO24550
WHO24560
WHO24570
WHO24580
WHO24590
WHO24600
WHO24610
WHC24620
WHO24630
WHO24640
WHO24650
WHO24660
WHO24670
WHO24680
WHO24690
WHO24700
WHC24710
WHO24720
WHO24730
WHO24740
WHC24750

```

continued

Table (7.3.6) continued

```

ALLOCATE MODCUT;
DO J=1 TO AMISH;
MISH=0;
MASH=0;
MANO=MH+1;
MH=MANO;
NH=MANO;
DO I=1 TO BMISH;
MA=MISS.LIL(I);
DA=-CEIL(-MA/10000);
JA=-CEIL(-MA/1000);
IF(JA-10*DA)=9) THEN DO;
DO IX=1 TO RMOD;
IF(MA=TRIM(IX)) THEN SAM=TRIM(IX);
END;
IF(SUBSTR(MISS.ACMP(J),I,1)='1'B) THEN DO;
MISH=JOSA(SAM)->MISS.SIN;
MMISH=JOSA(SAM)->MISS.MCS;
ALLOCATE FOLL;
ALLOCATE FOLLA;
IF(J=1) THEN DO;
NUB=JOSA(SAM)->MISS.SON;
IC K=1 TO NUB;
MODCUT.LIL(MASH+K)=JOSA(SAM)->MISS.LIL(K);
END;
WEST=JOSA(SAM)->MISS.MIN;
DO K=1 TO WEST;
MODCUT.LIL(MASH+NUB+K)=JOSA(SAM)->MISS.TIL(K);
END;
END;
DO K=MANO TO MH;
FOLLA=MODCUT.ACMP(K);
DO N=1 TO MMISH;
FOLL=JOSA(SAM)->MISS.ACMP(N);
MODCUT.ACMP(NH)=FOLLA||FOLL;
NH=NH+1;
END;
END;
FREE FOLLA;
FREE FOLL;
MASH=MISH+MASH;
IF(MANO=NH) THEN MH=NH-1;
NH=MANO;
END;
ELSE DO;
MISH=JOSA(SAM)->MISS.SIN;
ALLOCATE FOLL;
ALLOCATE FOLLA;
IF(J=1) THEN DO;
NUB=JOSA(SAM)->MISS.SON;
IC K=1 TO NUB;
MODCUT.LIL(MASH+K)=JOSA(SAM)->MISS.LIL(K);
END;
WEST=JOSA(SAM)->MISS.MIN;
IC K=1 TO WEST;

```

```

WHO24760
WHO24770
WHO24780
WHO24790
WHO24800
WHO24810
WHO24820
WHO24830
WHO24840
WHO24850
WHO24860
WHO24870
WHO24880
WHO24890
WHO24900
WHO24910
WHO24920
WHO24930
WHO24940
WHO24950
WHO24960
WHO24970
WHO24980
WHO24990
WHO25000
WHO25010
WHO25020
WHO25030
WHO25040
WHO25050
WHO25060
WHO25070
WHO25080
WHO25090
WHO25100
WHO25110
WHO25120
WHO25130
WHO25140
WHO25150
WHO25160
WHO25170
WHO25180
WHO25190
WHO25200
WHO25210
WHO25220
WHO25230
WHO25240
WHO25250
WHO25260
WHO25270
WHO25280
WHO25290
WHO25300

```

continued

Table (7.3.6) continued

MODCUT.LIL(MASH+NCB+K)=JOSA(SAM)->MISS.TII(K);	WHO25310
END;	WHO25320
END;	WHO25330
DO K=MANG TO MH;	WHO25340
FOLLA=MODCUT.AOMP(K);	WHO25350
FOLL=REPEAT('O'B,MISH-1);	WHO25360
MODCUT.ACMP(K)=FOLLA FCLL;	WHC25370
END;	WHO25380
FREE FOLL;	WHC25390
FREE FOLLA;	WHO25400
MASH=MISH+MASH;	WHC25410
END;	WHO25420
END;	WHO25430
ELSE DO;	WHO25440
ALLOCATE FOLLA;	WHO25450
IF(J=1) THEN	WHC25460
MODCUT.LIL(MASH+1)=MA;	WHO25470
DO K=MANG TO MH;	WHO25480
FOLLA=MODCUT.AOMP(K);	WHO25490
MODCUT.ACMP(K)=FOLLA SUBSTR(MISS.ACMP(J),I,1);	WHO25500
END;	WHO25510
FREE FOLLA;	WHO25520
MASH=MASH+1;	WHC25530
END;	WHO25540
END;	WHO25550
ALLOCATE FOLLA;	WHO25560
IF(J=1) THEN	WHO25570
WEST=MISS.MIN;	WHO25580
DO K=1 TO WEST;	WHC25590
MODCUT.LIL(MASH+K)=MISS.TII(K);	WHO25600
END;	WHC25610
MARD=MISS.SON+1;	WHO25620
NARD=MISS.SIN;	WHO25630
NARD=NARD-MARD;	WHO25640
DO K=MANG TO MH;	WHO25650
FCLLA=MODCUT.AOMP(K);	WHC25660
MODCUT.AOMP(K)=FOLLA SUBSTR(MISS.ACMP(J),MARD,NARD);	WHO25670
END;	WHC25680
END;	WHO25690

At the end of the TOP-SET procedure a new PER structure is allocated to incorporate all changes that are made during the TOP-SET procedure in the PER structure. The new PER structure will be further used in the numerical calculations performed in the NUMERO procedure. The list of computer statements for this purpose is given in Table (7.3.7).

7.3.2 Incorporation of the Monte-Carlo Package NUMERO into the PL-MODMC Code

The NUMERO external procedure is used to propagate failure parameters of the components of a fault tree. The NUMERO procedure consists of a main routine and four internal procedures as follows:

- (a) MONTCA procedure
- (b) EXPECT procedure
- (c) DOT procedure
- (d) PLUS procedure

In the main routine of the NUMERO procedure, one of the three different processes should be performed, depending on the three types of fault trees. These three types of fault trees are as follows:

- (1) A fault tree containing no replicated component or replicated gate
- (2) A fault tree containing only replicated and free components, but no replicated gate (i.e. only one improper module exists)

Table (7.3.7) Allocation of the New PER Structure

VIT=PR;	WHO26730
NUB=K;	WHO26740
IF (PROP.VALUE>2) THEN GO TO SIBEH;	WHO26750
WEST=MODCUT.SIN-K;	WHO26760
ALLOCATE PER;	WHO26770
PER.DEXTER=VIT->PER.DEXTER;	WHC26780
NUB=1;	WHO26790
PER.KIM=NULL;	WHO26800
DO I=1 TO K;	WHO26810
PER.TAR(I)=MODCUT.LIL(I);	WHC26820
END;	WHO26830
DO I=K+1 TO MODCUT.SIN;	WHO26840
PER.JIM(NUB)=MODCUT.LIL(I);	WHO26850
NUB=NUB+1;	WHO26860
END;	WHC26870
J=1;	WHO26880
N=1;	WHO26890
DO WHILE (BABA(J) /= NULL);	WHO26900
WEST=BABA(J)->PER.LEAL;	WHO26910
DO I=1 TO WEST;	WHO26920
DO K=1 TO PER.LEAL;	WHO26930
IF (PER.JIM(K)=BABA(J)->PER.JIM(I)) THEN	WHC26940
PER.KIM(K)=BABA(J)->PER.KIM(I);	WHO26950
END;	WHO26960
END;	WHO26970
ZAP=BABA(J);	WHO26980
J=J+1;	WHC26990
END;	WHO27000
J=1;	WHC27010
WEST=PER.LEAL;	WHO27020
DO I=1 TO WEST;	WHO27030
IF (PER.KIM(I)=NULL)	WHO27040
THEN DO WHILE (DADA(J) /= 0);	WHO27050
IF (PER.JIM(I)=DADA(J)) THEN PER.KIM(I)=JAJA(J);	WHO27060
J=J+1;	WHO27070
END;	WHO27080
END;	WHO27090
SIBEH: AT=RA;	WHO27100
DME=RA;	WHO27110
SKAP: ZAF=PB;	WHO27120
KAP=PT;	WHO27130

- (3) A fault tree containing free components replicated gates and/or replicated components.

For the fault trees with no replicated gate or component (type 1) there are no improper modules and therefore the procedure TOP-SET has never been called prior to the NUMERO procedure. In this case the variable BECHE, which will be set to NULL if the TOP-SET procedure is called, is not equal to NULL and thus no additional process is required to be performed in the main routine of the NUMERO procedure.

For the fault trees with no replicated gate (type 2), proper treatments which are made in the TOP-SET procedure for the disconnections made in the TREE-IN procedure have not been performed in the TOP-SET. Therefore, the set of statements in Table (7.3.8) is used to account for these disconnections in NUMERO. This process is very similar to the one in the TOP-SET procedure and uses the BKEEP array for the proper treatment of the replicated events.

For fault trees with replicated gates and/or replicated components (type 3), two arrays TEMP1 and TEMP2 are used to store the cutsets represented by the bit strings in the MODCUT structure in terms of the corresponding number of components in the fault tree. Declarations for the TEMP1 and TEMP2 arrays are as follows:

```
DECLARE TEMP1(ALAS) FIXED BINARY CONTROLLED;
DECLARE TEMP2(ALAS) FIXED BINARY CONTROLLED;
```

Table (7.3.9) shows the set of statements to transfer modular cutsets from the MODCUT.AOMP(I) bitstrings to the TEMP1 and TEMP2 arrays. The TEMP2 array will be used in the MONTE-CARLO simulation procedure to

Table (7.3.8) Proper Treatment of the Disconnections Made
in the TREE-IN Procedure

PT=PER.DEXTER;	WHO28240
RA=AT;	WHO28250
IF DME=NULL THEN DO;	WHO28260
ALM=MAM;	WHC28270
SINE=MISS.SIN;	WHO28280
MOSE=MISS.MOS;	WHC28290
ALLOCATE MODCUT;	WHO28300
MODCUT.ACMP=MISS.ACME;	WHO28310
FREE MISS;	WHO28320
DO I=1 TO PER.RAM;	WHO28330
MA=PER.TAR(I);	WHC28340
DO J=1 TO NIA;	WHO28350
IF (BKIEP(2,J)=MA) THEN DO;	WHC28360
PER.TAR(I)=BKIEP(1,J);	WHO28370
J=NIA+1;	WHC28380
END;	WHO28390
END;	WHO28400
END;	WHO28410
FREE BKIEP;	WHO28420

Table (7.3.9) Set of Statements to Transfer Cutsets Stored in the
Bitstrings MODCUT.AOMP(I) to the TEMP1 and TEMP2 Arrays

DEL=1;	WHC28430
NUB=PER.RAM;	WHO28440
WEST=PER.LEAL+1;	WHC28450
IF (PROP.VALUE<=2) THEN DC;	WHO28460
ALLOCATE PER;	WHC28470
PER.HECTOR=ZAP->PER.HECTOR;	WHO28480
PER.DEXTER=ZAP->PER.DEXTER;	WHO28490
PER.TAR=ZAP->PER.TAR;	WHO28500
PER.JIM(1)=PROP.NAME;	WHO28510
PER.KIM(1)=PT;	WHO28520
DO I=2 TO WEST;	WHO28530
PER.JIM(I)=ZAP->PER.JIM(I-1);	WHC28540
PER.KIM(I)=ZAP->PER.KIM(I-1);	WHO28550
END;	WHC28560
END;	WHO28570
FREE ZAP->PER;	WHO28580
END;	WHC28590
MISH=MODCUT.SIN;	WHO28600
TEMP1=0;	WHO28610
COUNT=0;	WHO28620
IF (MCNT=0) THEN GO TO MAGIC;	WHO28630
DO I=1 TO MODCUT.MOS;	WHC28640
IF MODCUT.ACMP(I)~=REPEAT('0'B,MISH) THEN DO;	WHO28650
COUNT=COUNT+1;	WHC28660
DO J=1 TO MODCUT.SIN;	WHO28670
IF (SUBSTR(MODCUT.AOMP(I),J,1)='1'B) THEN DO;	WHO28680
TEMP1(COUNT)=J;	WHO28690
COUNT=COUNT+1;	WHO28700
END;	WHO28710
END;	WHC28720
END;	WHO28730
END;	WHO28740
ALAS=COUNT;	WHC28750
ALLOCATE TEMP2;	WHO28760
TEMP2=0;	WHO28770
DO I=1 TO COUNT;	WHO28780
TEMP2(I)=TEMP1(I);	WHO28790
END;	WHO28800
FREE MODCUT;	WHO28810
MAGIC: CALL MONTCA;	WHO28820
RETURN;	WHO28830

calculate the top event at each trial. For example, if a fault tree consists of two sets of modular cutsets as follows:

$$(1) \quad M_1 \quad M_2;$$

$$(2) \quad M_2 \quad M_3 \quad M_4;$$

then the TEMP2 array stores the following numbers corresponding to the modules in the modular cutsets:

TEMP2 =	1	2	0	2	3	4	0

- (a) MONTCA Procedure: This procedure consists of statements to read failure parameters of the components in the fault tree. There are two options for the unavailability calculations. The first option is used when the point unavailability of the top event only is calculated, whereas the second option represents the Monte-Carlo simulation. The Monte-Carlo simulation option is the more general form of the point unavailability calculation option. In particular, equivalent error factor calculations for the multiplication of the lognormal distributions has been performed during the input process. The set of statements to input failure parameters, and to calculate equivalent mean and error factors for Class 3 and 4 components are shown in Table (7.3.10).

Table (7.3.10) Input Data Treatment by the MONTE-CARLO Package

```

IF (ROM=0) THEN DO;
  RCM=21;
  ALLOCATE CONPNT;
  ALLOCATE TORI2;
  CCNPNT(1)=0.5;      CONPNT(2)=1;      CONPNT(3)=2.5;
  CONPNT(4)=5;        CCNPNT(5)=10;     CONPNT(6)=15;
  CONPNT(7)=20;       CCNPNT(8)=25;
  CONPNT(9)=30;
  CCNPNT(10)=40;      CCNPNT(11)=50;    CONPNT(12)=60;
  CONPNT(13)=70;      CONPNT(14)=75;    CCNPNT(15)=80;
  CONPNT(16)=85;      CCNPNT(17)=90;    CONPNT(18)=95;
  CCNPNT(19)=97.5;    CCNPNT(20)=99;    CONPNT(21)=99.5;
END;
ELSE DO;
  ALLOCATE CONPNT;
  ALLOCATE TCRI2;
  DO I=1 TO ROM;
    GET LIST(I,CCNPNT(I));
  END;
  END;
  GET LIST(IPP);
  PSUM=0;
  PSUM2=0;
  GET LIST(NRAND,NTERM);
  VEN=PUN+DUN;
  IF (VEN<MISH) THEN VEN=MISH;
  ALLOCATE MEDIAN;
  MEDIAN=0;
  ALLOCATE FEN1;
  ALLOCATE FEN2;
  FEN1=0;
  FEN2=0;
  DO K=1 TO NCOMP+NNOT;
    GET LIST(AAA,OPTICN);
    IF (AAA='END') THEN GO TO BEGA;
    DO I=1 TO NCCME;
      IF (AAA=COMPENAM(I)) THEN GO TO MARMOT1;
      IF (OPTICN=1) THEN DO;
        GET LIST(MEDIAN(I),FEN2(I));
      GO TO MINK1;
      END;
      IF (OPTION=2) THEN DO;
        GET LIST(MED1,FEN2(I),MED2);
        MEDIAN(I)=MED1*MED2*0.5;
      GO TO MINK1;
      END;
      IF (OPTION=3) THEN DO;
        GET LIST(MED1,FAC1,MED2,FAC2);
        MEDIAN(I)=MED1*MED2;
        FEN2(I)=10**((SQRT((LOG10(FAC1))**2+(LOG10(FAC2))**2)));
      GO TO MINK1;
      END;
      IF (OPTION=4) THEN DO;
        GET LIST(MED1,FAC1,MED2,FAC2,MED3,FAC3,MED4);
        MEDIAN(I)=MED1*MED4*0.5+MED1*MED2+MED3/MED4;

```

```

WHO29710
WHO29720
WHC29730
WHO29740
WHC29750
WHO29760
WHC29770
WHO29780
WHO29790
WHO29800
WHO29810
WHO29820
WHO29830
WHC29840
WHO29850
WHC29860
WHO29870
WHO29880
WHO29890
WHO29900
WHC29910
WHO29920
WHO29930
WHO29940
WHO29950
WHO29960
WHO29970
WHC29980
WHO29990
WHC30000
WHO30010
WHC30020
WHO30030
WHC30040
WHO30050
WHO30060
WHC30070
WHO30080
WHO30090
WHO30100
WHO30110
WHO30120
WHC30130
WHO30140
WHC30150
WHO30160
WHO30170
WHO30180
WHO30190
WHO30200
WHO30210
WHC30220
WHO30230
WHO30240
WHO30250

```


Table (7.3.10) continued

```

FAC2=10** (SQRT ((LOG10 (FAC1)) **2+ (LOG10 (FAC2)) **2));
FEN2 (I) = (FAC1 * FAC2 * FAC3) ** (1/3);
GO TO MINK1;
END;
MARMCT1: END;
IF (BECHE=NULL) THEN
  DO I=1 TO PER.RAM;
    IF (PER.TAR (I) < 10000) THEN GO TO AWL1;
    MA=PER.TAR (I);
    DA=-CEIL (-MA/10000);
    JA=-CEIL (-MA/1000);
    JAK=JA-10*DA;
    NA=MA-1000*JA;
    IF (JAK=1 | AAA=REPNAME (NA)) THEN GO TO AWL1;
    NA=I;
    IF (OPTION=1) THEN DO;
      J=NA+FUN;
      GET LIST (MEDIAN (J), FEN2 (J));
      GO TO MINK1;
    END;
    IF (OPTION=2) THEN DO;
      J=NA+FUN;
      GET LIST (MED1, FEN2 (J), MED2);
      MEDIAN (J) = MED1 * MED2 * .5;
      GO TO MINK1;
    END;
    IF (OPTION=3) THEN DO;
      J=NA+FUN;
      GET LIST (MED1, FAC1, MED2, FAC2);
      MEDIAN (J) = MED1 * MED2;
      FEN2 (J) = 10 ** (SQRT ((LOG10 (FAC1)) **2+ (LOG10 (FAC2)) **2));
      GO TO MINK1;
    END;
    IF (OPTION=4) THEN DO;
      J=NA+FUN;
      GET LIST (MED1, FAC1, MED2, FAC2, MED3, FAC3, MED4);
      MEDIAN (J) = MED1 * MED4 * .5 + MED1 * MED2 + MED3 / MED4;
      FAC2 = 10 ** (SQRT ((LOG10 (FAC1)) **2+ (LOG10 (FAC2)) **2));
      FEN2 (J) = (FAC1 * FAC2 * FAC3) ** (1/3);
      GO TO MINK1;
    END;
    AWL1: END;
  MINK1: END;
  IF (BECHE=NULL) THEN
    DO I=1 TO PER.RAM;
      IF (PER.TAR (I) < 10000 & PER.TAR (I) > 0) THEN DO;
        K=PER.TAR (I);
        MEDIAN (FUN+I) = MEDIAN (K);
        FEN2 (FUN+I) = FEN2 (K);
        MEDIAN (K) = 1;
        FEN2 (K) = 0;
      END;
    ELSE DO;
      MA=PER.TAR (I);
      DA=-CEIL (-MA/10000);
      JA=-CEIL (-MA/1000);
      JAK=JA-10*DA;
      NA=MA-1000*JA;
      IF (JAK=2) THEN STATE (1, I) = 1 - STATE (1, DA);
    END;
  END;
END;

```

258

WHO30260
 WHC30270
 WHO30280
 WHC30290
 WHC30300
 WHO30310
 WHO30320
 WHO30330
 WHC30340
 WHO30350
 WHC30360
 WHO30370
 WHO30380
 WHO30390
 WHO30400
 WHC30410
 WHO30420
 WHC30430
 WHO30440
 WHC30450
 WHO30460
 WHO30470
 WHO30480
 WHO30490
 WHO30500
 WHO30510
 WHO30520
 WHO30530
 WHO30540
 WHO30550
 WHC30560
 WHO30570
 WHC30580
 WHO30590
 WHO30600
 WHO30610
 WHO30620
 WHO30630
 WHO30640
 WHC30650
 WHO30660
 WHC30670
 WHO30680
 WHO30690
 WHO30700
 WHO30710
 WHO30720
 WHO30730
 WHC30740
 WHO30750
 WHC30760
 WHO30770
 WHO30780
 WHO30790
 WHO30800
 WHO30810
 WHO30820
 WHO30830
 WHC30840
 WHO30850
 WHC30860

A part of the MONTCA procedure deals with the calculation of the accuracy of data for the unavailabilities calculated for different probability intervals. The mathematical concepts of these calculation have been discussed in Chapter 4 of this thesis. The set of statements for the MONTCA procedure for these calculations is listed in Table (7.3.11).

The sampling process is one of the most important processes in the MONTCA procedure. In this process, samples are taken from the unavailability distributions found for the input components. In this process the random number generator provides normal random numbers (variable AA) with a Central Limit Theorem Approximation that is shown by the variable NTERM. By using the variable AA, as discussed in Section 4.6 of this thesis, one would be able to find a sample unavailability from the unavailability distribution of the component. The random number generator and the sampling process in the MONTCA procedure are shown in Table (7.3.12).

Immediately after a set of samples is found for the modules and components in the fault tree, the internal procedure EXPECT is called in order to calculate the unavailability of the top event. The value of the unavailability is stored in the variable REY which will be transferred to the TOP-P(I) array in each trial. Therefore, the TOP-P(I) array must be allocated such that its dimension is equal to the number of trials NRAND. The TOP-P(I) array is used to sort the unavailabilities calculated in each trial. The method of sorting is simple and has been discussed in Chapter 4 of this thesis. Mean, standard deviation and the TOP event error factors are also calculated in this part of the MONTCA procedure. The set of statements shown in Table (7.3.13) is used to store

Table (7.3.11) Set of Statements for the Measure of Accuracy of the Monte-Carlo Calculations

BECA: IF (NRAND=0) THEN NRAND=2000;	WHO30870
IF (NTERM=0) THEN NTERM=12;	WHO30880
ALLOCATE TORI1;	WHO30890
ALLOCATE TORI3;	WHO30900
ALLOCATE TORI4;	WHO30910
DO I=1 TO ROM;	WHO30920
TORI2(I)=100-CONPNT(I);	WHO30930
DUMMY1=CONPNT(I)*TORI2(I)*2.0;	WHO30940
DUMMY1=NRAND/DUMMY1;	WHO30950
TORI1(I)=CONPNT(I)*((DUMMY1)**0.5);	WHO30960
IF (TORI1(I)>13) THEN TORI1(I)=13;	WHO30970
TORI4(I)=ERF(TORI1(I));	WHO30980
DUMMY1=NRAND*CONPNT(I)*TORI2(I)*0.001;	WHO30990
TORI3(I)=(EXP(-TORI1(I)**2))/(SQRT(DUMMY1*6.283186))+ (0.2+	WHO31000
0.0025*(ABS(CONPNT(I)-TORI2(I))))/DUMMY1+EXP(-(1.5*SQRT(DUMMY1	WHO31010
))));	WHO31020
END;	WHO31030

Table (7.3.12) The Random Number Generator and the Sampling Procedure in the PL-MODMC Code

DO I=1 TO VEN;	WHO31450
IF (MEDIAN(I)=0) THEN FEN1(I)=0;	WHO31460
ELSE FEN1(I)=LOG (MEDIAN(I));	WHC31470
IF (FEN2(I)=-0) THEN	WHO31480
FEN2(I) = LOG (FEN2(I)) / 1.645;	WHC31490
END;	WHO31500
AMGIS=SQRT(12/NTERM);	WHO31510
ALLOCATE TOP_P;	WHO31520
IY=IPP/0.499977;	WHO31530
IY1=IY;	WHC31540
DO WHILE (ASACH<NRAND);	WHO31550
DO I=1 TO VEN;	WHC31560
AA=0;	WHO31570
DO IRAND=1 TO NTERM;	WHO31580
IY1=CEIL(IY1);	WHC31590
IY=IY-IY1+1;	WHO31600
AA=AA+IY;	WHC31610
IY1=IY*65539;	WHO31620
IY=IY1;	WHO31630
END;	WHO31640
XCCMP=AMGIS*(AA-0.5*NTERM)*FEN2(I)+FEN1(I);	WHO31650
IF(XCCMP=0) THEN GO TO BACON;	WHO31660
MEDIAN(I)=EXP(XCCMP);	WHO31670
BACCN: END;	WHC31680

Table (7.3.13) Set of Statements to Sort the Unavailabilities

```

      REY=1;
      REX=1;
      IF (BECHS=NUIL) THEN
DO N=1 TO ALAS;
      IF (TEMP2(N)~=0) THEN REX=REX*MELIAN(TEMP2(N));
      ELSE DO;
        REY=REY*(1-REX);
        REX=1;
      END;
      END;
      ELSE DO;
        REY=MEDIAN(1);
        GO TC TOPP;
      END;
      ZND;
      REY=1-REY;
      TCPP: TOP_P(ASACH)=REY;
      PSUM=PSUM+REY;
      PSUM2=PSUM2+REY**2;
      ASACH=ASACH+1;
      END;
      XMEAN=PSUM/NRAND;
      STDDEV=(PSUM2-PSUM*XMEAN)/(NRAND-1);
      STDDEV=SQRT(STDDEV);
      M1=NRAND;
      MAYA2: M1=M1/2;
      M1=CEIL(M1);
      K1=NRAND-M1;
      DO I= 1 TO K1 ;
      J=I;
      MAYA1: IM1=J+M1;
      IF ((TOP_P(J)-TOP_P(IM1)) <=0) THEN GO TO MAYA;
      PROBAB=TOP_P(J);
      TOP_P(J)=TOP_P(IM1);
      TOP_P(IM1)=PROBAB;
      J=J-M1;
      IF (J>0) THEN GO TO MAYA1;
MAYA: END;
      IF (M1>1) THEN GO TO MAYA2;
      INDEX=NRAND*0.5;
      INDEX=CEIL(INDEX);
      XMED=TOP_P(INDEX);
      INDEX=CEIL(NRAND*0.05);
      F05=TOP_P(INDEX);
      F1=XMED/F05;
      INDEX=CEIL(NRAND*(.95));
      P95=TOP_P(INDEX);
      F2=P95/XMED;
      PUT EDIT('MEAN PROB=' ,XMEAN,'STANDARD DEVIATION=' ,STDDEV,
'MEDIAN PROB=' ,XMED) (SKIP(5),A(10),E(18,6),X(10),A(19),
E(18,6),X(10),A(12),E(18,6));
      PUT EDIT('ERROR FACTOR (5%)=' ,F1,'ERROR FACTOR (95%)=' ,F2)
(SKIP(2),A(17),P(12,5),X(10),A(18),P(12,5));
      PUT EDIT('CONFIDENCE LEVEL','UNAVAILABILITY','INTVL PROBABILITY',
'MAX ERROR')
(SKIP(2),X(15),A(16),X(10),A(14),X(7),A(17),X(10),A(9));
      NRAND=NRAND/100;
      DO I=1 TO ROM;
      INDEX=CEIL(NRAND*CONPNT(I));
      PUT EDIT(CONPNT(I),TOP_P(INDEX),TCRI4(I),TCRI3(I))
(SKIP(2),X(10),P(12,5),X(10),E(18,6),X(4),E(18,6),X(3),E(18,6));
      END;
      NMONT: RETURN;
      END MONTCA;

```

262 WHO31690
 WHC31700
 WHO31710
 WHC31720
 WHO31730
 WHC31740
 WHO31750
 WHO31760
 WHO31770
 WHO31780
 WHC31790
 WHO31800
 WHC31810
 WHO31820
 WHO31830
 WHO31840
 WHO31850
 WHC31860
 WHO31870
 WHC31880
 WHO31890
 WHO31900
 WHO31910
 WHO31920
 WHO31930
 WHO31940
 WHO31950
 WHO31960
 WHO31970
 WHO31980
 WHO31990
 WHO32000
 WHO32010
 WHO32020
 WHO32030
 WHC32040
 WHO32050
 WHC32060
 WHO32070
 WHO32080
 WHO32090
 WHO32100
 WHC32110
 WHO32120
 WHC32130
 WHO32140
 WHO32150
 WHO32160
 WHO32170
 WHO32180
 WHO32190
 WHO32200
 WHO32210
 WHO32220
 WHO32230
 WHO32240
 WHO32250
 WHC32260
 WHO32270
 WHO32280
 WHO32290
 WHO32300
 WHO32310

unavailabilities in the TOP-P(I) array and to sort them.

(b) EXPECT Procedure: This procedure uses the PER structure, and TEMP2 array or MODCUT-AOMP(I) bitstrings for calculating the top event unavailability. The TEMP2 array is used for the Monte-Carlo simulation and MODCUT.AOMP(I) bitstrings are used only for point unavailability calculations.

The PER structure is used to identify the nested and parent modules, and the BOST(I) array (see Chapter 3 of Reference [1]) is used to identify free modules. Whenever an AND module is found the procedure DOT is called to obtain its equivalent unavailability distribution of the AND modules. For OR modules the procedure PLUS is called for the same reason. Finally, the EXPECT procedure uses the TEMP2 array or MODCUT.AOMP(I) bitstrings to obtain the unavailability of the top event from the values obtained in the DOT and PLUS procedures for the modules of the fault tree. The set of statements for calculating the top event unavailability is summarized in Table (7.3.14).

(c) PLUS and DOT procedures: The PLUS Procedure finds equivalent unavailability distributions for the OR modules (see Chapter 4). Obviously, the unavailability of an OR module is found by calculating the union of the unavailabilities of input components to that module. Table (7.3.15) shows the list of statements of the PLUS procedure. Similar to the PLUS procedure, the DOT procedure is internally called from the EXPECT Procedure to calculate the equivalent unavailability distributions for AND modules, by multiplying the unavailabilities of input components to that AND module. The set of statements shown in

Table (7.3.14) The EXPECT Procedure

EXPECT: PROC;	WHC32930
IF (MCNT=0) THEN	WHO32940
MEDIAN=0;	WHO32950
DO I=1 TO IB;	WHO32960
CAT=BOST(I);	WHO32970
PT=CAT;	WHO32980
IF (PROP.HOST=NULL) THEN GO TO SHOH;	WHO32990
IF (PROP.VALUE=1) THEN CALL DOT(CAT,ENTA);	WHO33000
IF (PROP.VALUE=2) THEN CALL PLUS(CAT,ENTA);	WHO33010
ENTA: REY=PROP.REL(1);	WHO33020
END;	WHO33030
SHOH: IF (MCNT=0 & BECHE=NULL) THEN GO TO EXCEL;	WHO33040
IF (MCNT=0 & BECHE=NULL) THEN DO;	WHO33050
MEDIAN(1)=REY;	WHO33060
FEN2(1)=PROP.TIPC;	WHO33070
GO TO EXCEL;	WHO33080
END;	WHO33090
IF (MCNT=0 & MISH>VEN) THEN DO;	WHC33100
VEN=MISH;	WHO33110
FREE MEDIAN;	WHO33120
ALLOCATE MEDIAN;	WHO33130
MEDIAN=0;	WHO33140
FREE FEN1;	WHO33150
ALLOCATE FEN1;	WHO33160
FEN1=0;	WHC33170
END;	WHO33180
DO I=1 TO PER.LEAL;	WHC33190
CAT=PER.KIM(I);	WHO33200
PT=CAT;	WHO33210
IF (PT=NULL) THEN GO TO CUTS;	WHO33220
IF (PROP.VALUE=1) THEN CALL DOT(CAT,ESTA);	WHO33230
IF (PROP.VALUE=2) THEN CALL PLUS(CAT,ESTA);	WHO33240
ESTA: J=PER.LAN+I;	WHO33250
PROY(J)=PROP.REL(1);	WHG33260
IF (MCNT=0) THEN GO TO CUTS;	WHO33270
MEDIAN(J)=PROY(J);	WHO33280
FEN1(J)=PROP.TIPC;	WHO33290
CUTS: END;	WHO33300

continued

Table (7.3.14) continued

DO I=1 TO PER.RAM;	WHO33310
PROY(I)=STATD(1,I);	WHO33320
IF(MONT=0) THEN GO TO MONIC;	WHO33330
MEDIAN(I)=FFOY(I);	WHO33340
FEN1(I)=FEN2(I+FUN);	WHO33350
MONIC: END;	WHO33360
IF(MCNT=0) THEN DO;	WHO33370
FREE FEN2;	WHO33380
ALLCCATE FEN2;	WHO33390
FEN2=FEN1;	WHO33400
FEN1=0;	WHO33410
END;	WHO33420
REY=1;	WHO33430
REX=1;	WHO33440
IF(MONT=0) THEN	WHO33450
DO I=1 TO ALAS;	WHO33460
IF(TEMP2(I)=0) THEN REX=REX*PROY(TEMP2(I));	WHO33470
ELSE IG;	WHO33480
REY=REY*(1-REX);	WHO33490
REX=1;	WHO33500
END;	WHO33510
END;	WHO33520
ELSE DO I=1 TO MODCUT.MOS;	WHO33530
IF MODCUT.AOMP(I)=REPEAT('0'B,MISH) THEN DO;	WHO33540
DO J=1 TO MODCUT.SIN;	WHO33550
IF(SUESTR(MODCUT.AOMP(I),J,1)='1'B) THEN	WHO33560
REX=REX*PROY(J);	WHO33570
INI;	WHO33580
REY=REY*(1-REX);	WHO33590
REX=1;	WHO33600
END;	WHO33610
END;	WHO33620
REY=1-REY;	WHO33630
EXCEL: RETURN;	WHO33640
END EXPECT;	WHO33650
END NUMERO;	WHO33660

Table (7.3.15) The PLUS Procedure

```

/*      RELIABILITY CALCULATION      */
PLUS: PROC (BAT, EXA);
DECLARE BAT POINTER;
DECLARE EXA LABEL;
  PT=BAT;
  REX=1;
  P=1;
  COUNT=0;
  IF (PROP.LIM=1 & PROP.TIL(1)=0) THEN GO TO SLUA;
  DO J=1 TO PROP.LIM;
    IF (STATE(1, PROP.TIL(J))<1) THEN DO;
      REX=REX*(1-STATE(1, PROP.TIL(J)));
      IF (MONT=0) THEN DO;
        P=P*PEN2 (PROP.TIL(J));
        COUNT=COUNT+1;
      END;
    END;
  END;
  SLUA: IF (PROP.MIM=1 & PROP.PIM(1)=NULL) THEN GO TO SLUB;
        DO J=1 TO PROP.MIM;
          REX=REX*(1-PROP.PIM(J)->PROP.REL(1));
          IF (MONT=0) THEN DO;
            K=PROP.PIM(J)->PROP.TIFO;
            P=P*K;
            COUNT=COUNT+1;
          END;
        END;
  SLUB: REX=1-REX;
  ZACH: PROP.REL(1)=REX;
        IF (MONT=0 & COUNT>0) THEN DO;
          P=P** (1/COUNT);
          PROP.TIPO=P;
        END;
        ELSE PROP.TIPO=0;
        GO TO EXA;
  END PLUS;

```

```

WHO32320
WHC32330
WHO32340
WHO32350
WHO32360
WHC32370
WHO32380
WHO32390
WHO32400
WHO32410
WHO32420
WHO32430
WHO32440
WHO32450
WHO32460
WHO32470
WHO32480
WHO32490
WHO32500
WHO32510
WHO32520
WHC32530
WHO32540
WHO32550
WHO32560
WHC32570
WHO32580
WHC32590
WHO32600
WHO32610
WHO32620
WHO32630
WHC32640
WHO32650
WHO32660
WHO32670

```

Table (7.3.16) forms the procedure DOT used in the external procedure NUMERO.

7.4 Organization OF The Procedures used in MODCUT

In Chapter 5 the methodology used in the code MODCUT has been discussed in detail. In this section the programming of this methodology is presented. Two procedures exist in the MODCUT code:

- (1) CUT-SET Procedure
- (2) TOP-SET Procedure

The CUT-SET procedure is used to generate cutsets of proper modules (i.e. free, nested and parent modules), and the TOP-SET procedure is used to generate cutsets of the improper modules, that is modules represented by minimal modular cutsets (see Figures 5.2.1). Each of the CUT-SET and the TOP-SET procedures is called immediately after a proper or improper module is generated. The procedure to determine proper and improper modules of a fault tree in MODCUT is exactly similar to the procedure contained in PL-MOD and will not be further discussed in this thesis. However, the reader is referred to reference [1] for more information with regard to proper and improper modules and the methodology used for their determination.

7.4.1 CUT-SET Procedure

The CUT-SET procedure is constructed such as to find cutsets of proper modules. The cutsets for each module will be represented by a set of bit-string vectors. The modules are either AND or OR; and they have either a set of modules as input or not. Therefore a proper module

Table (7.3.16) The DOT Procedure

DOT: PROC (BAT, EXA);	WHO32680
DECLARE EAT POINTER;	WHO32690
DECLARE EXA LABEL;	WHO32700
PT=BAT;	WHO32710
REX=1;	WHO32720
P=0;	WHO32730
IF (PROP.LIM=1 & PROP.TIL(1)=0) THEN GC TC ECTA;	WHO32740
DO J=1 TO PROP.LIM;	WHO32750
REX=REX*STATE(1, PROP.TIL(J));	WHO32760
IF (MCNT=0 & FEN2(PROP.TIL(J)) > 0) THEN	WHO32770
P=P+(LOG10(FEN2(PROP.TIL(J))))**2;	WHO32780
END;	WHO32790
DOTA: IF (PROP.MIM=1 & PROP.PIM(1)=NULL) THEN GO TO DOTE;	WHC32800
DO J=1 TO PROP.MIM;	WHO32810
REX=REX*PROP.PIM(J) -> PROP.REL(1);	WHC32820
IF (MONT=0 & PROP.PIM(J) -> PROP.TIFC > 0) THEN	WHO32830
P=P+(LOG10(PROP.PIM(J) -> PROP.TIPO))**2;	WHC32840
END;	WHO32850
DOTB: PROP.REL(1)=REX;	WHC32860
IF (MONT=0) THEN GC TO EXA;	WHO32870
IF (P=0) THEN	WHC32880
PROP.TIPO=10**SQRT(P);	WHO32890
ELSE PROP.TIFC=0;	WHO32900
GC TO EXA;	WHO32910
END DOT;	WHO32920

can be one of the following categories:

- (1) OR module with no input modules
- (2) OR module with one or more input modules
- (3) AND module with no input module and
- (4) AND module with one or more input modules.

For categories (1) and (3) the generation of simple cutsets is easy whereas for categories (2) and (4) the determination of the cutsets is more involved.

Some of the important structures, arrays, and variables which are used in the CUT-SET procedure are defined below.

Variable CUTNO is read from the input data and, if it is 1, this indicates that only the number of cutsets in each module should be obtained; otherwise, the cutsets should be determined in the form of bitstrings.

SIZE indicates the maximum size of the cutsets for the module whose cutsets are generated.

Based structure MCSG stores the PROP structure data [1] and evaluates bit strings for the cutsets (i.e. bitstrings for the combination of simple component and proper modules). The PROP structure is allocated in the modularization process to store the data of proper modules. The MCSG has the following configuration:

```

DECLARE 1 MCSG BASED (AW),
        2 LIM FIXED BINARY
        2 TIL(LILE REFER(MCSG.LIM))FIXED BINARY,
        2 MIM FIXED BINARY,
        2 TIM(LIME REFER(MCSG.MIM))FIXED BINARY,
        2 JIM FIXED BINARY
        2 AOMP(JIME REFER(MCSG.JIM))BIT(LILE REFER(MCSG.LIM)),
        2 KIM FIXED BINARY
        2 NAME FIXED BINARY
        2 PIM FIXED BINARY,
        2 BOMP(PIME REFER(MCSG.PIM))BIT(TIME REFER(MCSG.KIM));

```

MCSG.TIL(I) and MCSG.TIM(I) store simple input components and input modules, respectively. MCSG.AOMP(1) and MCSG.BOMP(1) stores bitstrings represent input components and modules. For example, if 3 components and 2 modules are input to an AND module then MCSG.AOMP(1) = '1 1 1 1 1' B and the first three bits represent input components, and the rest represent the input modules.

Based structure CUTSET stores the cutsets and components of a module. The difference between CUTSET and MCSG structures is that the CUTSET structure stores only bit-strings representing components, but the MCSG structure stores bit-strings representing components and modules. Therefore, for categories (1) and (3) in which no modules are inputted, CUTSET and MCSG structures store identical information, but for categories (2) and (4) CUTSET data must be derived from the MCSG data by replacing input modules to the module which is represented by the MCSG structure with their simple cutsets. The cutset structure has the following configuration:

```

DECLARE 1 CUTSET BASED(AY),
        2 MOS FIXED BINARY (31,0),
        2 SIN FIXED BINARY,
        2 MIN FIXED BINARY,
        2 JIN FIXED BINARY,
        2 KIN FIXED BINARY,
        2 COMP(SINE REFER(CUTSET.MOS))BIT(SINE REFER(CUTSET.SIN)),
        2 LIL(SINE REFER(CUTSET.MIN))FIXED BINARY,
        2 AOMP(MOSE REFER(CUTSET.JIN))BIT(SINE REFER(CUTSET.KIN));

```

The LIL array is used to store components and CUTSET.COMP and CUTSET.AOMP are bit strings to store cutsets of the module. For example, if the MSCG structure for an AND Module (module M₁) has stored MCSG.AOMP(1) = '1 1 1 1 1'B, for input components C₁, C₂, C₃ and input modules M₁ and M₂; and modules M₁ and M₂ have the following cutsets (i.e. previously found by calling CUT-SET procedures):

```

CUTSET.COMP(1) = '1 1'B (Module M1)
                  C5 C6

CUTSET.COMP(1) = '0 1'B
CUTSET.COMP(2) = '1 0'B } (Module M2)
                  C8 C3

```

Then the following values will be assigned to the CUTSET structure of the module M1 in two steps:

Step 1) To assign cutsets of module M1

CUTSET.LIL(1) = C1

CUTSET.LIL(2) = C2

CUTSET.LIL(3) = C3

CUTSET.LIL(4) = C5

CUTSET.LIL(5) = C6

CUTSET.LIL(6) = C8

CUTSET.LIL(7) = C3

CUTSET.AOMP(1) = '1 1 1 1 1 0 0'B

CUTSET.AOMP(2) = '0 0 0 0 0 0 0'B

CUTSET.COMP(1) = '0 0 0 0 0 0 0'B

CUTSET.COMP(2) = '0 0 0 0 0 0 0'B

Step 2) To assign cutsets of module M2

Same CUTSET.LILs

CUTSET.AOMP(1) = '1 1 1 1 1 0 0'B

CUTSET.AOMP(2) = '0 0 0 0 0 0 0'B

CUTSET.COMP(1) = '1 1 1 1 1 0 1'B

CUTSET.COMP(2) = '1 1 1 1 1 1 0'B

It can be seen that CUTSET.AOMP is a set of intermediate bitstrings to help generate the final set of bitstrings which are stored in the CUTSET.COMP.

JOSA(GUM) is an array of pointers to store pointer (AY) related to the CUTSET structure.

The Operator `||` has been extensively used in developing the bit strings. The function of this operator is to join two bitstrings together. For example:

`'1 0 1'B || '0 0 1 1'B = '1 0 1 0 0 1 1'B`

The set of statements in Table (7.4.1) transfers proper AND and OR modules to the MCSG structure, and generates the bit strings for their representation.

For generating cutsets from MCSG structures with one or more input modules the set of statements in Table (7.4.2) calculates the number of bitstrings and the length of the CUTSET.LIL array in the CUTSET structure.

The statements listed in Table (7.4.3) allocate the CUTSET structure for the AND module, and generate the bit strings from the bitstrings of the MCSG structure.

The CUTSET structure for OR modules is allocated and obtained from a somewhat similar approach to the AND module shown in Table (7.4.3). The procedures for minimizing and sorting the cutsets stored in the form of bit-strings in the CUTSET structure are discussed in the following section.

7.4.2 TOP-SET Procedure

The TOP-SET procedure is constructed so that cutsets of improper modules can be found from their modular minimal cutsets. Each improper module has a corresponding PER structure which is similar to the PROP structure for proper module [see Reference 1]. In

Table (7.4.1) List of Statements to Transfer Proper AND and OR Modules
to the MCSG Structure and Generate the Bitstrings for them

IF (PROP.VALUE=1) THEN GO TO ARO;	
JIME=1;	CUT00750
PIME=1;	CUT00760
IF (PROP.TIM(1)=0) THEN	CUT00770
TIME=PROP.LIM;	CUT00780
ELSE TTME=PRCP.LIM+PRCP.MIM;	CUT00790
LILE=PROP.LIM;	CUT00800
LIME=PROP.MIM;	CUT00810
ALLOCATE MCSG;	CUT00820
DME=AW;	CUT00830
MCSG.NAME=PROP.NAME;	CUT00840
MCSG.TIL=PROP.TIL;	CUT00850
MCSG.TIM=PROP.TIM;	CUT00860
IF (MCSG.TIL(1)~=0) THEN DO;	CUT00870
AMISH=MCSG.LIM;	CUT00880
AMISH=AMISH-1;	CUT00890
MCSG.AOMP(1)=REPEAT('1'E,AMISH);	CUT00900
END;	CUT00910
ELSE MCSG.ACMP(1)='0'E;	CUT00920
IF (MCSG.TIM(1)=0) THEN	CUT00930
MCSG.BOMP(1)=REPEAT('1'E,AMISH);	CUT00940
ELSE DO BMISH=AMISH+PROP.MIM;	CUT00950
MCSG.BCMP(1)=REPEAT('1'E,BMISH);	CUT00960
END;	CUT00970
GO TO BORC;	CUT00980
ARO: IF (PROP.TIM(1)=0) THEN DO;	CUT00990
JIME=PRCP.LIM;	CUT01000
TIME=JIME;	CUT01010
PIME=TIME;	CUT01020
END;	CUT01030
ELSE DO;	CUT01040
JIME=PRCP.LIM;	CUT01050
TTME=JIME+PROP.MIM;	CUT01060
PIME=TTME;	CUT01070
END;	CUT01080
LILE=PRCP.LIM;	CUT01090
LIME=PROP.MIM;	CUT01100

continued

Table (7.4.1) continued

ALLOCATE MCSG;	CUT01110
DME=AW;	CUT01120
MISH=0;	CUT01130
MCSG.NAME=PRCP.NAME;	CUT01140
MCSG.TIL=PROP.TIL;	CUT01150
MCSG.TIM=PRCP.TIM;	CUT01160
IF(MCSG.TIL(1)~=0) THEN DO;	CUT01170
MISH=JIME;	CUT01180
ALLOCATE FOLL;	CUT01190
DO I=1 TO MISH;	CUT01200
FOLL=REPEAT('0'B,MISH);	CUT01210
SUBSTR(FOLL,I,1)='1'B;	CUT01220
MCSG.AOMP(I)=FOLL;	CUT01230
END;	CUT01240
FREE FOLL;	CUT01250
END;	CUT01260
ELSE MCSG.ACMP(1)='0'E;	CUT01270
IF(MCSG.TIM(1)=0) THEN DO;	CUT01280
AMISH=JIME;	CUT01290
MISH=JIME;	CUT01300
MCSG.BOMP=MCSG.ACMP;	CUT01310
END;	CUT01320
ELSE DO;	CUT01330
MISH=MCSG.LIM+MCSG.NIM;	CUT01340
ALLOCATE FOLL;	CUT01350
DO I=1 TO MISH;	CUT01360
FOLL=REPEAT('0'B,MISH);	CUT01370
SUBSTR(FOLL,I,1)='1'B;	CUT01380
MCSG.BOMP(I)=FOLL;	CUT01390
END;	CUT01400
FREE FOLL;	CUT01410
END;	CUT01420
	CUT01430

Table (7.4.2) List of Statements for Calculating the Number and Dimension fo Bitstrings in the CUTSET Structure

BOFO: IF (MCSG.TIM(1)-=0) THEN EC;	CUT02320
CCUNT=1;	CUT02330
SINE=MCSG.IIM;	CUT02340
BMISH=MCSG.MIM;	CUT02350
DO I=1 TO EMISH;	CUT02360
SAM=MCSG.TIM(I);	CUT02370
ZAP=JOSA(SAM);	CUT02380
COUNT=COUNT*ZAP->CUTSET.MCS;	CUT02390
END;	CUT02400
MOSE=COUNT;	CUT02410
DO I=1 TC BMISH;	CUT02420
SAM=MCSG.TIM(I);	CUT02430
ZAP=JCSA(SAM);	CUT02440
SINE=SINE+ZAP->CUTSET.SIN;	CUT02450
END;	CUT02460

Table (7.4.3) List of Statements to Allocate CUTSET Structure
for an AND Gate and Generate the Bitstrings of the
CUTSET Structure

IF (CUTNO=1) THEN GO TO FRCG1;	CUT02470
ALLCCATE CUTSET;	CUT02480
DSI=AY;	CUT02490
MANO=MCSG.LIL;	CUT02500
MASH=MANO;	CUT02510
KAP=AY;	CUT02520
CUTSET.LIL=MCSG.TIL;	CUT02530
DO I=1 TO BMISH;	CUT02540
SAM=MCSG.TIM(I);	CUT02550
ZAP=JOSA(SAM);	CUT02560
AMISH=ZAP->CUTSET.SIN;	CUT02570
DO K=1 TO AMISH;	CUT02580
MANO=MANO+1;	CUT02590
MCSY=ZAP->CUTSET.LIL(K);	CUT02600
CUTSET.LIL(MANO)=MCSY;	CUT02610
END;	CUT02620
END;	CUT02630
CUTSET.CCMP(1)=MCSG.ACMP(1);	CUT02640
NH=1;	CUT02650
DO I=1 TO BMISH;	CUT02660
SAM=MCSG.TIM(I);	CUT02670
ZAP=JOSA(SAM);	CUT02680
NH=0;	CUT02690
MISH=ZAP->CUTSET.SIN;	CUT02700
ALLOCATE FOLLA;	CUT02710
ALLOCATE FCLL;	CUT02720
MMISH=ZAP->CUTSET.MCS;	CUT02730
DO J=1 TO NH;	CUT02740
DO K=1 TO MMISH;	CUT02750
FCLL=ZAP->CUTSET.COMP(K);	CUT02760
IF (FOLL=REPEAT('0'B,MISH)) THEN GO TO FATAK;	CUT02770
NH=NH+1;	CUT02780
FOLLA=CUTSET.CCMP(J);	CUT02790
CUTSET.ACMP(NH)=FOLLA FCLL;	CUT02800
FATAK: END;	CUT02810
END;	CUT02820
MASH=MASH+MISH;	CUT02830
FREE FCLL;	CUT02840
FREE FOLLA;	CUT02850
CUTSET.COMP=CUTSET.ACMP;	CUT02860
NH= NH;	CUT02870
END;	CUT02880

the PER structure of an improper module input components, nested and parent modules are stored. Every time that a set of minimal modular cutsets is obtained for an improper higher order module (i.e. improper top event, K-out-of-N gates or replicated gates), the procedure TOP-SET is called.

The structure MCSG has exactly the same composition as in the CUT-SET procedure. But, in the TOP-SET procedure the MCSG structure is used to transfer data from the PER structure, and transfers modular minimal cutsets to the bit-string arrays MCSG.AOMP(I). Therefore, the MCSG structure should be allocated such that the number of bitstrings MCSG.AOMP(I) is equal to the number of minimal modular cutsets for the corresponding module. Array MCSG.TIL(I) stores the list of replicated modules, ON-OFF complementary components, parent and nested modules input to the improper module. Array MCSG.TIM(I) stores only parent and nested modules. The set of statements listed in Table (7.4.4) transfers the data of the PER structure and modular minimal cutsets to the MCSG structure.

From the MCSG structure and the cutsets generated for modules input to the improper module, through the CUT-SET procedure, it would be possible to calculate the number and length of the CUTSET structure for the improper module in question.

The statements summarized in Table (7.4.5) are part of the TOP-SET procedure to find the dimensions of the CUTSET structure.

The process for finding simple cutsets from the MCSG structure of the improper module and the cutsets of the input module's is as follows:

Table (7.4.4) List of Statements to Transfer Data from PER Structure
and Transfer Modular Minimal Cutsets to the MCSG Structure

IF (SYM1=1) THEN LILE=PER.BAM+PER.LEAL;	CUT04980
ELSE LILE=PER.BAM+PER.LEAL+1;	CUT04990
LIME=LILE;	CUT05000
N=0;	CUT05010
VIT=PER.HECTOR;	CUT05020
DO WHILE (VIT~=NULL);	CUT05030
VIC=VIT;	CUT05040
VIT=VIC->FLOOR;	CUT05050
N=N+1;	CUT05060
END;	CUT05070
JIME=N;	CUT05080
PIME=1;	CUT05090
TIME=0;	CUT05100
ALLOCATE MCSG;	CUT05110
DME=AW;	CUT05120
MCSG.NAME=STORK->PROP.NAME;	CUT05130
MCSG.TIL=PER.TAB;	CUT05140
I=PER.BAM+1;	CUT05150
IF (SYM1=1) THEN GC IC FAT;	CUT05160
MCSG.TIL(I)=MCSG.NAME;	CUT05170
I=I+1;	CUT05180
RAT: K=1;	CUT05190
DO J=I TO LILE;	CUT05200
MCSG.TIL(J)=PER.JIM(K);	CUT05210
K=K+1;	CUT05220
END;	CUT05230
MCSG.TIM=PER.JIM;	CUT05240
N=1;	CUT05250
VIT=PER.HECTOR;	CUT05260
DO WHILE (VIT~=NULL);	CUT05270
VIC=VIT;	CUT05280
VIT=VIC->FLOOR;	CUT05290
MCSG.AOME(N)=VIC->VECTCE.COMP;	CUT05300
N=N+1;	CUT05310
END;	CUT05320

Table (7.4.5) List of Statements to Calculate Number and Length of the CUTSET Structure Bit-strings for Improper Modules

COUNT=1;	CUT05330
SINE=0;	CUT05340
AMISH=MCSG.JIM;	CUT05350
BMISH=PER.RAM;	CUT05360
CMISH=BMISH+1;	CUT05370
DMISH=MCSG.LIM;	CUT05380
I=1;	CUT05390
DO J=1 TO BMISH;	CUT05400
MA=MCSG.TIL(J);	CUT05410
DA=-CEIL(-MA/10000);	CUT05420
JA=-CEIL(-MA/1000);	CUT05430
IX=MA-JA*1000;	CUT05440
IF((JA-10*DA)=9) THEN DO;	CUT05450
SAM=TRIN(IX);	CUT05460
ZAP=JOSA(SAM);	CUT05470
SINE=SINE+ZAP->CUTSET.SIN;	CUT05480
IF(SUBSTR(MCSG.ACMP(I),J,1)='1'E) THEN	CUT05490
COUNT=COUNT*ZAP->CUTSET.MCS;	CUT05500
END;	CUT05510
ELSE SINE=SINE+1;	CUT05520
END;	CUT05530
DO K=CMISH TO DMISH;	CUT05540
SAM=MCSG.TIL(K);	CUT05550
ZAP=JOSA(SAM);	CUT05560
IF(SUBSTR(MCSG.ACMP(I),K,1)='1'E) THEN	CUT05570
COUNT=COUNT*ZAP->CUTSET.MCS;	CUT05580
SINE=SINE+ZAP->CUTSET.SIN;	CUT05590
END;	CUT05600
MOSE=COUNT;	CUT05610
DO I=2 TO AMISH;	CUT05620
CCOUNT=1;	CUT05630
DO J=1 TO BMISH;	CUT05640
MA=MCSG.TIL(J);	CUT05650
DA=-CEIL(-MA/10000);	CUT05660
JA=-CEIL(-MA/1000);	CUT05670
IX=MA-JA*1000;	CUT05680
IF((JA-10*DA)=9) THEN DO;	CUT05690
SAM=TRIN(IX);	CUT05700
ZAP=JOSA(SAM);	CUT05710
IF(SUBSTR(MCSG.ACMP(I),J,1)='1'E) THEN	CUT05720
COUNT=COUNT*ZAP->CUTSET.MCS;	CUT05730
END;	CUT05740
END;	CUT05750
DO K=CMISH TO DMISH;	CUT05760
SAM=MCSG.TIL(K);	CUT05770
ZAP=JOSA(SAM);	CUT05780
IF(SUBSTR(MCSG.ACMP(I),K,1)='1'B) THEN	CUT05790
COUNT=COUNT*ZAP->CUTSET.MCS;	CUT05800
END;	CUT05810
MOSE=MOSE+COUNT;	CUT05820
END;	CUT05830

- (1) All of the components representing the modules
input to the improper module are transferred to the
CUTSET.LIL(I) array.
- (2) All bit strings representing the modules input
to the improper module are transferred to the CUTSET.COMP(I)
and CUTSET.AOMP(I) bitstrings.

For example the following modular cutsets are given for an improper
module M_i with the following data:

```

MCSG.NAME = Mi
MCSG.LIL(1) = 29001 (This represents the input replicated module 29001)
MCSG.LIL(2) = *Mi (This represents the proper section of the improper
                    module)
MCSG.LIL(3) = Mi (This represents the nested module input to
                    improper module Mi)
MCSG.AOMP(1) = '0 1 0'B
MCSG.AOMP(2) = '1 0 1'B

```

Also, suppose the following cutsets are determined through the previous
CUT-SET and TOP-SET procedures for modules 29001, *M_i and M_i;

- (1) for 29001 CUTSET.COMP(1) = '0 1'B
 CUTSET.COMP(2) = '1 0'B
- (2) for *M_i CUTSET.COMP(1) = '0 1'B
- (3) for M_i CUTSET.COMP(1) = '1 1 0'B
 CUTSET.COMP(2) = '0 1 1'B

There is a total of 5 bit-strings and 7 components for M_i , and to obtain simple cutsets the following calculations must be performed for the evaluation of the bitstrings for M_i :

- (1) Put CUTSETS that represent replicated module 29001

CUTSET.AOMP(1) = '0 0 0 0 0 0 0'B

CUTSET.AOMP(2) = '0 1 0 0 0 0 0'B

CUTSET.AOMP(3) = '1 0 0 0 0 0 0'B

CUTSET.AOMP(4) = '0 0 0 0 0 0 0'B

CUTSET.AOMP(5) = '0 0 0 0 0 0 0'B

CUTSET.AOMP(1) represents the first modular minimal cutset for the M_i module, and the rest of the CUTSET.AOMP(I) bitstrings represent the second modular minimal cutset of the M_i module. In this step all of the CUTSET.COMP(I) bitstrings will be set equal to the CUTSET.AOMP(I) bitstrings given above.

- (2) In this step the calculations will be started by using the

CUTSET.COMP(I) bit strings for inputting the module $*M_i$

cutsets. As a result one gets the following bitstrings:

CUTSET.COMP(1) = '0 0 0 1 0 0 0'B

CUTSET.COMP(2) = '0 1 0 0 0 0 0'B

CUTSET.COMP(3) = '1 0 0 0 0 0 0'B

CUTSET.COMP(4) = '0 0 0 0 0 0 0'B

CUTSET.COMP(5) = '0 0 0 0 0 0 0'B

Again, at this point the CUTSET.AOMP(I) bitstring will be set equal to the CUTSET.COMP(I) bitstrings given above.

- (3) Finally, put cutsets representing the nested module M_1 by starting from CUTSET.AOMP bitstrings:

CUTSET.AOMP(1) = '0 0 0 1 0 0 0'B

CUTSET.AOMP(2) = '0 1 0 0 1 1 0'B

CUTSET.AOMP(3) = '0 1 0 0 0 1 1'B

CUTSET.AOMP(4) = '1 0 0 0 1 1 0'B

CUTSET.AOMP(5) = '1 0 0 0 0 1 1'B

All CUTSET.COMP(I) will be set equal to CUTSET.AOMP(I). Statements shown in Table (7.4.6) are part of the TOP-SET procedure to perform the aforementioned operations.

To remove the effect of the replication of a component in the array CUTSET.LIL(I) the statements in Table (7.4.7) are used, based on the discussion in Chapter 5, to properly treat this effect.

By using the methodology for discriminating the cutsets, the statements in Table (7.4.8) are used in the TOP-SET procedure to perform a size discrimination.

All of the cutsets which meet the size criterion have their corresponding size stored in an array called CUT(I). This array will be used in the minimization process to remove nonminimal cutsets based on the OR method which has been fully discussed in Chapter 5. The set of statements in Table (7.4.9) is used to perform the minimization process, sorting and printing of minimal cutsets.

Table (7.4.6) List of Statements to Generate Cutsets of Improper Modules from the CUTSET Structure

284

MH=0;	CUT06270
DC I=1 TC CUTSET.MCS;	CUT06280
CUTSET.CCMP(I)=REPEAT('C'E,SINE);	CUT06290
END;	CUT06300
CUTSET.ACMP=CUTSET.CCMP;	CUT06310
DC I=1 TO AMISH;	CUT06320
MANO=MH+1;	CUT06330
MH=MANO;	CUT06340
NH=MANO;	CUT06350
MASH=0;	CUT06360
DO J=1 TC BMISH;	CUT06370
MA=MCSG.TIL(J);	CUT06380
DA=-CEIL(-MA/10000);	CUT06390
JA=-CEIL(-MA/1000);	CUT06400
IX=MA-JA*1000;	CUT06410
IF ((JA-10*DA)=-9) THEN GO TO FAT;	CUT06420
IF (SUBSTR(MCSG.AOMP(I),J,1)='1'E) THEN DO;	CUT06430
SAM=TRIN(IX);	CUT06440
ZAP=JOSA(SAM);	CUT06450
MISH=ZAP->CUTSET.SIN;	CUT06460
ALLOCATE FOLL;	CUT06470
ALLOCATE FOLLA;	CUT06480
MMISH=ZAP->CUTSET.MOS;	CUT06490
DC K=MANO TC MH;	CUT06500
DC N=1 TC MMISH;	CUT06510
FCLLA=CUTSET.CCMP(K);	CUT06520
FCLL=ZAP->CUTSET.CCMP(N);	CUT06530
IF FOLL=REPEAT('O'B,MISH) THEN GO TC FATAJ;	CUT06540
CUTSET.AOMP(NH)=FCLLA FCLL;	CUT06550
NH=NH+1;	CUT06560
FATAJ: END;	CUT06570
END;	CUT06580
MASH=MASH+MISH;	CUT06590
IF (MANO=-NH) THEN	CUT06600
MH=NH-1;	CUT06610
NH=MANO;	CUT06620
END;	CUT06630
ELSE DO;	CUT06640
SAM=TRIN(IX);	CUT06650
ZAP=JOSA(SAM);	CUT06660
MISH=ZAP->CUTSET.SIN;	CUT06670
ALLOCATE FOLLA;	CUT06680
ALLOCATE FCLL;	CUT06690
DO K=MANO TO MH;	CUT06700
FOLLA=CUTSET.CCMP(K);	CUT06710
FCLL=REPEAT('O'B,MISH);	CUT06720
CUTSET.ACMP(K)=FOLLA FCLL;	CUT06730
END;	CUT06740
MASH=MASH+MISH;	CUT06750
END;	CUT06760
FREE FOLL;	CUT06770
FREE FOLLA;	CUT06780
DO M=MANO TO MH;	CUT06790
CUTSET.CCMP(M)=CUTSET.ACMP(M);	CUT06800
END;	CUT06810
GO TO THIN;	CUT06820

continued

FAT:	IF(SUBSTR(MCSG.ACMP(I),J,1)='1'E) THEN DC;	CUT06830
	ALLOCATE FOLIA;	CUT06840
	DC K=MANO TC MH;	CUT06850
	FOLLA=CUTSET.CCMP(K);	CUT06860
	CUTSET.ACMP(K)=FOLLA '1'E;	CUT06870
	END;	CUT06880
	MASH=MASH+1;	CUT06890
	END;	CUT06900
	ELSE DO;	CUT06910
	ALLOCATE FOLLA;	CUT06920
	DC K=MANO TC MH;	CUT06930
	FOLLA=CUTSET.COMP(K);	CUT06940
	CUTSET.ACMP(K)=FOLLA '0'B;	CUT06950
	END;	CUT06960
	MASH=MASH+1;	CUT06970
	END;	CUT06980
	FREE FOLL;	CUT06990
	FREE FOLIA;	CUT07000
	DC M=MANO TC MH;	CUT07010
	CUTSET.CCMP(M)=CUTSET.ACMP(M);	CUT07020
	END;	CUT07030
THIN:	END;	CUT07040
	DO J=CMISH TO DMISH;	CUT07050
	SAM=MCSG.TIL(J);	CUT07060
	ZAP=JOSA(SAM);	CUT07070
	IF(SUBSTR(MCSG.ACMP(I),J,1)='1'E) THEN DC;	CUT07080
	MISH=ZAP->CUTSET.SIN;	CUT07090
	ALLOCATE FOLL;	CUT07100
	ALLOCATE FOLLA;	CUT07110
	MMISH=ZAP->CUTSET.MCS;	CUT07120
	DC K=MANO TC MH;	CUT07130
	DO N=1 TC MMISH;	CUT07140
	FOLLA=CUTSET.CCMP(K);	CUT07150
	FOLL=ZAP->CUTSET.CCMP(N);	CUT07160
	IF FOLL=REPEAT('C'E,MISH) THEN GO TO FATAG;	CUT07170
	CUTSET.ACMP(NH)=FOLLA FOLL;	CUT07180
	NH=NH+1;	CUT07190
FATAG:	END;	CUT07200
	END;	CUT07210
	MASH=MASH+MISH;	CUT07220
	IF (MANO<=NH) THEN	CUT07230
	MH=NH-1;	CUT07240
	NH=MANO;	CUT07250
	END;	CUT07260
	ELSE DO;	CUT07270
	MISH=ZAP->CUTSET.SIN;	CUT07280
	ALLOCATE FOLL;	CUT07290
	ALLOCATE FOLLA;	CUT07300
	DC K=MANO TC MH;	CUT07310
	FOLLA=CUTSET.CCMP(K);	CUT07320
	FOLL=REPEAT('0'B,MISH);	CUT07330
	CUTSET.ACMP(K)=FOLLA FOLL;	CUT07340
	END;	CUT07350
	MASH=MASH+MISH;	CUT07360
	END;	CUT07370
	DC M=MANO TC MH;	CUT07380
	CUTSET.COMP(M)=CUTSET.ACMP(M);	CUT07390
	END;	CUT07400
	END;	CUT07410
	END;	CUT07420

Table (7.4.7) List of Statements to Account for Replication of
a Component in the CUTSET.COMP(I) Bitstring

MISH=CUTSET.SIN;	CUT07590
MANO=CUTSET.MOS;	CUT07600
AMISH=MISH-1;	CUT07610
DC I=1 TC AMISH;	CUT07620
K=I+1;	CUT07630
DC J=K TO MISH;	CUT07640
N=CUTSET.LIL(I);	CUT07650
IF (CUTSET.LIL(J)=N) THEN DO;	CUT07660
DC M=1 TC MANO;	CUT07670
SUBSTR(CUTSET.CCMP(M),I,1)=SUBSTR(CUTSET.COMP(M),I,1);	CUT07680
SUBSTR(CUTSET.COMP(M),J,1);	CUT07690
SUBSTR(CUTSET.CCNE(M),J,1)='0'B;	CUT07700
END;	CUT07710
END;	CUT07720
END;	CUT07730
END;	CUT07740

Table (7.4.8) List of Statements to Perform a Size Discrimination

NCTA=0;	CUT07750
ALLOCATE CUT;	CUT07760
ALLOCATE FCIL;	CUT07770
CUT=0;	CUT07780
DO I=1 TO MANC;	CUT07790
SCORE=0;	CUT07800
IF CUTSET.COMP(I) = REPEAT('C'E,MISH) THEN	CUT07810
DO N=1 TO MISH;	CUT07820
IF(SUBSTR(CUTSET.COMP(I),N,1)='1'B) THEN SCORE=SCORE+1;	CUT07830
END;	CUT07840
IF(SCORE<=SIZE) THEN CUT(I)=SCORE;	CUT07850
ELSE CUT(I)=0;	CUT07860
END;	CUT07870

Table (7.4.9) List of Statements to Minimize, Sort and Print the Cutsets

DC I=1 TC MANO;	CUT07880
EMISH=I+1;	CUT07890
IF CUT(I)=0 THEN GO TO FATAH;	CUT07900
DO J=EMISH TC MANO;	CUT07910
IF CUT(J)=0 THEN GO TO FATAN;	CUT07920
FOLL=CUTSET.CCMP(I);	CUT07930
FOLL=FOLL CUTSET.CCMP(J);	CUT07940
IF (FOLL=CUTSET.CCMP(J)) THEN DO;	CUT07950
CUTSET.CCMP(J)=REPEAT('0'E,MISH);	CUT07960
CUT(J)=0;	CUT07970
END;	CUT07980
ELSE IF (FOLL=CUTSET.CCMP(I)) THEN DO;	CUT07990
CUTSET.CCMP(I)=REPEAT('0'E,MISH);	CUT08000
J=MANO;	CUT08010
NOTA=1;	CUT08020
CUT(I)=0;	CUT08030
END;	CUT08040
FATAN: END;	CUT08050
IF NOTA=0 THEN MOSE=MOSE+1;	CUT08060
ELSE NOTA=0;	CUT08070
FATAH: END;	CUT08080
MARK=0;	CUT08090
IF (SPINE(MCSG.NAME)=NULL) THEN	CUT08100
DO J=1 TC SIZE;	CUT08110
DO I=1 TC MANO;	CUT08120
IF (CUT(I)=J) THEN DO;	CUT08130
MARK=MARK+1;	CUT08140
PUT FILE(Y) EDIT(MARK) (SKIP(2),F(4),SKIP(1));	CUT08150
DC N=1 TC MISH;	CUT08160
IF (SUBSTR(CUTSET.CCMP(I),N,1)='1'B) THEN	CUT08170
IF (CUTSET.LIL(N)<10000) THEN PUT FILE(Y) EDIT(COMPNAME(CUTSET.LIL(N))	CUT08180
(A(12)));	CUT08190
ELSE DO;	CUT08200
MA=CUTSET.LIL(N);	CUT08210
DA=-CEIL(-MA/10000);	CUT08220
JA=-CEIL(-MA/1000);	CUT08230
JAK=JA-10*DA;	CUT08240
NA=MA-1000*JA;	CUT08250
IF (JAK=1) THEN PUT FILE(Y) EDIT(REPNAME(NA))(A(12));	CUT08260
ELSE PUT FILE(Y) EDIT('-' REPNAME(NA))(A(12));	CUT08270
END;	CUT08280
END;	CUT08290
END;	CUT08300
END;	CUT08310
END;	CUT08320

If the TOP event has not been reached yet, a special procedure will be used to allocate a new CUTSET structure, so as to store only bit strings that remain through the discrimination and minimization processes and, thereby, release space which is occupied by the "non-essential" bit strings. The set of statements shown in Table (7.4.10) is performing this task.

Finally, it should be noticed that the treatment of replications, the discrimination, the minimization and the sorting of the cutsets are exactly similar for the CUT-SET and TOP-SET procedures.

7.5 Organization of the MOCUS-BACFIRE Package

The MOCUS-BACFIRE Package is obtained by coupling the codes MOCUS[4] and BACFIRE [5]. The code MOCUS is a computer program to generate cutsets of a fault tree (see also Appendix A of this thesis). This code is written in FORTRAN IV for the IBM 360/370 computer series. BACFIRE is a computer program to search among the basic events of a minimal cutset for potential causes of failure. A minimal cutset with a common potential cause of failure is called a common cause candidate (see Chapter 6 of this thesis). BACFIRE is written in FORTRAN IV for the IBM 360/370 computers. BACFIRE reads as input the cutsets provided by a cutset generator code. Therefore, it would be much easier, due to the numerous cutsets in a fault tree, to couple a cutset generator code with BACFIRE. The package resulting from this coupling would be very simple to use, because only the fault tree and common cause susceptibility data would have to be inputted to that package.

Table (7.4.10) List of Statements to Allocate a New CUTSET Structure
to Store only Essential Cutsets

MANO=1;	CUT08330
IF(MOSE<CUTSET.MOS & GATENAM(MCSG.NAME) ~~NE~~ TOP') THEN	CUT08340
DO;	CUT08350
SINE=CUTSET.SIN;	CUT08360
ALLOCATE CUTSET;	CUT08370
DSI=AY;	CUT08380
JOSA(MCSG.NAME)=AY;	CUT08390
CUTSET.LII=KAP->CUTSET.LII;	CUT08400
AMISH=KAP->CUTSET.MOS;	CUT08410
DO I=1 TC AMISH;	CUT08420
IF(CUT(I) ~~NE~~ 0) THEN DO;	CUT08430
CUTSET.CCMP(MANC)=KAP->CUTSET.CCMP(I);	CUT08440
MANC=MANC+1;	CUT08450
END;	CUT08460
END;	CUT08470
FREE KAP->CUTSET;	CUT08480
END;	CUT08490

By carefully reviewing the present cutset generation codes, the code MOCUS has been selected for coupling with BACFIRE. Also, enough information, experience and literature is available for the MOCUS code, so that the user can easily familiarize himself with the background information.

The process of coupling the two codes is very simple and straightforward. The input section in which BACFIRE reads the cutsets and the output from which MOCUS writes cutsets have been changed such as to make it possible for BACFIRE to read the cutsets directly from the NAM(I), and ICS(I,J) arrays in MOCUS after they have been generated.

In BACFIRE, array CSM(I) reads the cutsets. Each time that a set of cutsets is transferred to CSM(I), the BACFIRE Code is called to search for common cause candidates in that cutset. Hence, in the PRINTR subroutine of the MOCUS code which prints the minimal cutsets, the following statements transfer cutsets to the CSM(I) array and call the BACFIRE code for the identification of common cause failures.

Most of the variables that are used in both MOCUS and BACFIRE codes are set in the COMMON blocks for all subroutines for which these variables are essential. The main routine in the MOCUS Code in which the fault tree data are read has been slightly changed, so that in addition to fault tree data, susceptibility data can also be read by calling only that portion of the BACFIRE code that reads the susceptibility data. The changes in the main routine of the MOCUS Code is shown below.

In the MOCUS-BACFIRE package, the BACFIRE code has been considered as a subroutine of the MOCUS code that is called whenever a minimal cutset is assigned to the CSM(I) array. The statements in the two subroutines of the MOCUS code which have been changed (i.e. Main and PRINTR subroutines) are listed in Table (7.5.1). Table (7.5.2) summarizes the parameters in the common blocks and parts of the BACFIRE Code that have been changed.

7.6 Organization of the MODCUT-BACFIRE Package

The MODCUT-BACFIRE package is an alternative to the MOCUS-BACFIRE Package. As has been discussed in Section 7.4 and Chapter 5 of this thesis, the MODCUT code is a computer program to generate cutsets of a fault tree using the modularization technique. In particular, MODCUT is capable of handling a variety of fault tree types. MODCUT is written in PL/I language, and although direct coupling of MODCUT and BACFIRE is possible with the inter-language facilities of the PL/I language, it has been decided to generate the package by creating intermediate auxiliary files to store minimal cutsets generated by the MODCUT code. The reason is that the direct coupling results in some handling difficulties, and furthermore the package would be highly dependent upon the type of PL/I compiler available. However, by using the conversational Monitor System (CMS) of the IBM 370/168 computers and by defining an intermediate file, the coupling of MODCUT-BACFIRE has been easily performed.

A file named Y is defined for the MODCUT code to store all minimal cutsets generated from the fault tree. The statements in Table(7.5.3) constitute the changes in the CUTSET, and TOP-SET procedures which print the cutsets in the MODCUT Code.

Table (7.5.1) New Form of the MAIN and PRINTR Subroutines in the MOCUS Code 293

```

C      ARRAY W IN LABELED COMMON /WORK/. THE STARTING LOCATIONS MOC00010
C      OF THE VARIOUS ARRAYS ARE COMPUTED BY THE ROUTINE ALOCT1. MOC00020
C      TO MODIFY THE WORK SPACE AVAILABLE TO MOCUS, IT IS NECESSARY MOC00030
C      ONLY TO CHANGE THE DIMENSIONS OF W. THE VALUE ASSIGNED TO W MOC00040
C      BY THE DATA STATEMENT MUST CORRESPOND TO THE DIMENSIONS OF W. MOC00050
C
COMMON/WORK/ W(20000) MOC00060
REAL*8 W MOC00070
DATA M/20000/ MOC00080
C
INTEGER ERROR MOC00090
COMMON/ERRORS/ NAME,CARD,ERRCR MOC00100
COMMON/TWO/MAST,MAST MOC00110
DIMENSION IW(50) MOC00120
EQUIVALENCE (IW(1),W(1)) MOC00130
EQUIVALENCE (ITMAX,IW(3)),(NROW,IW(4)),(NCOL,IW(5)) MOC00140
DIMENSION MAXIMP(30,2),SUMIMP(30,2),AVGIMP(30,2),MINIMP(30,2) MOC00150
DIMENSION C(101),MFG(101),LG(101,10),GCS(101,4,5) MOC00160
DIMENSION LOC(1001),MAP(1001,10),IMP(100,4,5),MIMP(4,10,5) MOC00170
DIMENSION ASSUS(101),CTG(101),MP(101),SUS(101,3) MOC00180
DIMENSION CGCS(30,3),CSC(5,3) MOC00190
DIMENSION MANU(37,3),AMAN(37) MOC00200
DIMENSION DA(10,10),D(10),CAT(4,10,5),D1(10) MOC00210
COMMON /LT/ NCUT,NCOMF,NIN,NLINE,D,LO,LOC,C,CAT,GCS MOC00220
COMMON /LY/ MIMP,IMP,A2,IMPCO,ASSUS,CTG,NMAP,MP MOC00230
COMMON/LP/ YES,QMAP,SUS MOC00240
COMMON/LK/MAIMP,SIMP,NUM,MFG,AMAN MOC00250
COMMON/LO/ MANU,DD1,DD2 MOC00260
COMMON/IMPO/ MAXIMP,SUMIMP,AVGIMP,MINIMP MOC00270
COMMON /CNE/ CGCS,CSC,CMFG MOC00280
C
MAXWRK=2*M MOC00290
DO 10 J=1,M MOC00300
W(J)=0.D0 MOC00310
10 CONTINUE MOC00320
C
NS1=1 MOC00330
MAST=0 MOC00340
MAST=0 MOC00350
NS2=0 MOC00360
CALL BACFR(NS1,NS2) MOC00370
NS2=0 MOC00380
CALL GETNG(W,NG,NI) MOC00390
IW(6)=MAXWRK MOC00400
CALL ALOCT1(W,NG,NI) MOC00410
C
N=IW(7) MOC00420
CALL PREP(NG,W(IW(N+6)),W(IW(N+1)),W(IW(N+2)),W(IW(N)),W(IW(11)), MOC00430
W(IW(N+7)),W(IW(N+8)),W(IW(N+9))) MOC00440
IF(ERROR.EQ.1) GO TO 200 MOC00450
C
30 CONTINUE MOC00460
IW(10)=1 MOC00470
IF(IW(9).EQ.1) CALL ANDCR(W(IW(N+3))) MOC00480
CALL SCAN(NG,W(IW(N+3)),W(IW(N+4)),W(IW(N+5)),W(IW(N+6))) MOC00490
MOC00500
MOC00510
MOC00520
MOC00530
MOC00540
MOC00550

```

Table (7.5.1) continued

```

C      CALL ALOCT2(W,NG)
C      ITMAX=MINO(ITMAX,NCOL)
C      IF (IW(9) .EQ. 1) ITMAX=NCOL
C      IF (ITMAX .LE. 0) ITMAX=NCOL
C      IW(19)=0
C      IF (ITMAX .LT. NCOL) IW(19)=1
C      NAVAIL=((NCOL + 1)*NROW)/IW(12)
C      L=IW(N+6)
C      DO 20 J=L,M
20    W(J)=0.D0
C      CALL MOCUS(W(IW(N)),W(IW(N+1)),W(IW(N+2)),W(IW(N+3)),W(IW(N+4)),
&      W(IW(N+5)),W(IW(N+6)),W(IW(N+7)))
C      LFTOVR=NAVAIL - ((NCOL + 1)*IW(10))/IW(12)
C      NUSED=M - LFTOVR
C      PRINT 100, M,NUSED,LFTOVR
100    FORMAT('THE W ARRAY IS DIMENSIONED FOR ',I5,' REAL*8 WORDS.',
&      '// I6,' WORDS WERE USED AND THERE WERE ',I5,' EXCESS.')
C      WRITE(6,110) IW(10),NCOL
110    FORMAT('MOCUS USED ',I5,' ROWS AND ',I5,
&      ' COLUMNS IN THE ICS MATRIX.')
C      IF (IW(20) .EQ. 0) WRITE(6,120)
C      IF (IW(20) .EQ. 1) WRITE(6,130)
120    FORMAT(' / -EXTERNAL SCRATCH STORAGE WAS NOT REQUIRED.')
130    FORMAT(' / -EXTERNAL SCRATCH STORAGE WAS REQUIRED.')
C      IF (IW(9) .NE. 2) GO TO 200
C      IW(9)=1
C      IW(20)=0
C      GO TO 30
200    CONTINUE
C      WRITE(6,220)
220    FORMAT(' /// -',35X,' - - - CONCLUSION OF MOCUS OUTPUT - - -
& -')
C      STOP
C      END
C      SUBROUTINE PRINTR(KOUNT,MAX,NAM,IT,ICS)
C      ROUTINE TO PRINT MINIMAL CUT SETS. ALL CUT SETS WITH 'K' TO
C      'KOUNT' COMPONENTS WILL BE PRINTED.
C      COMMON /BRANCH/ IK,NTST,NGTX(20),IPLG(20)
C      COMMON /VERSN/ VERSN(6)
C      COMMON /WORK/ IW(50)
C      EQUIVALENCE (NG,IW(1)),(NI,IW(2)),(ITMAX,IW(3)),(NROW,IW(4))
C      INTEGER*2 IT(NROW),ICS(NROW,11),LT(12)
C      INTEGER CUT/' CUT'//, PATH/'PATH'/
C      LOGICAL CSW/F/, PSW/F/
C      LOGICAL FIRST/T/
C      REAL*8 NAM(1)
C      DATA MAXLIN/55/

```

continued

Table (7.5.1) continued

	DIMENSION CSM(10)	MOC01110
	COMMON /BK/ CSM	MOC01120
	DIMENSION MAXIMP(30,2),SUMIMP(30,2),AVGIMP(30,2),MINIMP(30,2)	MOC01130
	DIMENSION C(101),MFG(101),LG(101,10),GCS(101,4,5)	MOC01140
	DIMENSION LOC(1001),MAP(1001,10),IMP(100,4,5),MIMP(4,10,5)	MOC01150
	DIMENSION ASSUS(101),CTG(101),MP(101),SUS(101,3)	MOC01160
	DIMENSION CGCS(30,3),CSC(5,3)	MOC01170
	DIMENSION MANU(37,3),AMAN(37)	MOC01180
	DIMENSION DA(10,10),D(10),CAT(4,10,5),D1(10)	MOC01190
	COMMON /LT/ NCUT, NCOMP, NIN, NLINE, D, LO, ICC, C, CAT, GCS	MOC01200
	COMMON /LY/ MIMP, IMP, A2, IMPCG, ASSUS, CTG, NMAP, MP	MOC01210
	COMMON /LP/ YES, QMAP, SUS	MOC01220
	COMMON /LK/ HAIMP, SIMP, NUM, MFG, AMAN	MOC01230
	COMMON /LO/ MANU, DE1, DD2	MOC01240
	COMMON /IMPO/ MAXIMP, SUMIMP, AVGIMP, MINIMP	MOC01250
	COMMON /ONE/ CGCS, CSC, CMFG	MOC01260
	COMMON /TWO/ NAST, MAST	MOC01270
	COMMON MAP, NCSM, DA	MOC01280
	REAL*8 CSM, LMDA(NI), TAU(NI)	MOC01290
C	IW(9)=0 FOR CUT SETS	MOC01300
C	IW(9)=1 FOR PATH SETS	MOC01310
	IF (IW(9) .EQ. 1) GO TO 10	MOC01320
	IF (CSW) GO TO 20	MOC01330
	CSW=.TRUE.	MOC01340
	KIND=CUT	MOC01350
	GO TO 15	MOC01360
C		MOC01370
10	IF (PSW) GO TO 20	MOC01380
	PSW=.TRUE.	MOC01390
	KIND=PATH	MOC01400
C		MOC01410
15	NT=0	MOC01420
	KK=1	MOC01430
	LINES=7	MOC01440
C		MOC01450
C	FOR PATH SETS, DELETE PRINTOUT FOR THOSE WHICH ARE EMPTY	MOC01460
C	(CONTROLLED BY PRSW)	MOC01470
C		MOC01480
20	CONTINUE	MOC01490
	DO 50 N=KK, KCUNT	MOC01500
	IF (IW(9) .NE. 1) GO TO 25	MOC01510
	DO 22 I=1, MAX	MOC01520
	IF (IT(I) .EQ. N) GO TO 25	MOC01530
22	CONTINUE	MOC01540
	GO TO 50	MOC01550
C		MOC01560
25	CONTINUE	MOC01570
	IF (IW(17) .NE. 1) GO TO 26	MOC01580
	IF (LINES .GT. MAXLIN-10) LINES=MAXLIN - 3	MOC01590
	LINES=MOD(LINES+3, MAXLIN)	MOC01600
	IF (LINES .EQ. 0) WRITE(6,180) VERSN	MOC01610
	WRITE(6,110) KIND, N	MOC01620
26	L=0	MOC01630
	DO 40 I=1, MAX	MOC01640
	IF (IT(I) .NE. N) GO TO 40	MOC01650

continued

Table (7.5.1) continued

```

L=L + 1
IF (IW(17) .NE. 1) GO TO 28
LINES=MOD(LINES+1,MAXLIN)
IF (LINES .EQ. 0) WRITE(6,190) VERPN,KIND,N
WRITE(6,120) L,(NAM(ICS(I,J)),J=1,N)
DO 27 J=1,N
CSM(J)=NAM(ICS(I,J))
27 CONTINUE
NCSH=N
IF (N .LT. 2) GO TO 28
MAST=2
WAST=0
CALL BACFR(2,0)
-IF (N .GT. 10) WRITE(6,270)
28 IF (IW(18) .NE. 1) GO TO 40
IF (IFLG(IK) .NE. 0) GO TO 40
DO 30 J=1,N
30 LT(J)=ICS(I,J) - NG
WRITE(10,140) N,(LT(J),J=1,N)
WRITE( 7,140) N,(LT(J),J=1,N)
40 CONTINUE
NT=NT + L
IF (L .NE. 0) GO TO 50
WRITE(6,130)
LINES=LINES + 5
50 CONTINUE
KK=KOUNT + 1
IF (KOUNT .LT. ITMAX) RETURN
IF (IW(18) .EQ. 1 .AND. IFLG(IK) .EQ. 0) END FILE 10
WRITE(6,150) KIND,NT
IF (IW(19) .EQ. 0) WRITE(6,160) KIND
IF (IW(19) .NE. 0) WRITE(6,170) KIND,ITMAX,IW(5)
NSER=1
MAST=2
WAST=1
CALL BACFR(2,NSER)
RETURN
C
110 FORMAT('-',A4,' SETS WITH ',I3,' COMPONENTS' / )
120 FORMAT( 5X,I7,' ',4X,10(A8,1X) / (17X,10(A8,1X)))
130 FORMAT(6X,'NONE EXIST.')
140 FORMAT(9I10)
150 FORMAT('-', // '-TOTAL NUMBER OF ',A4,' SETS FOUND WAS',I6)
160 FORMAT('-',ALL ',A4,' SETS HAVE BEEN DETERMINED.')
170 FORMAT('-',ONLY ',A4,' SETS WITH ',I3,' OR LESS COMPONENTS HAVE BEEN
    EN DETERMINED.' / '0SETS WITH UP TO ',I3,' COMPONENTS MAY EXIST.')
180 FORMAT('1',105X,6A4)
190 FORMAT('1',105X,6A4 / ' ',A4,' SETS WITH ',I3,' COMPONENTS' / 1
C
C * * * * *
C
C THIS ENTRY FOR OUTPUTTING COMPONENT KEYS
C
C ENTRY PRTN(NAM)
C
    continued

```

MOC01660
MOC01670
MOC01680
MOC01690
MOC01700
MOC01710
MOC01720
MOC01730
MOC01740
MOC01750
MOC01760
MOC01770
MOC01780
MOC01790
MOC01800
MOC01810
MOC01820
MOC01830
MOC01840
MOC01850
MOC01860
MOC01870
MOC01880
MOC01890
MOC01900
MOC01910
MOC01920
MOC01930
MOC01940
MOC01950
MOC01960
MOC01970
MOC01980
MOC01990
MOC02000
MOC02010
MOC02020
MOC02030
MOC02040
MOC02050
MOC02060
MOC02070
MOC02080
MOC02090
MOC02100
MOC02110
MOC02120
MOC02130
MOC02140
MOC02150
MOC02160
MOC02170
MOC02180
MOC02190
MOC02200

Table (7.5.1) continued

	FIRST=FIRST .AND. IW(18).EQ. 1	MOC02210
	IF (.NOT. FIRST) RETURN	MOC02220
	FIRST=.FALSE.	MOC02230
C		MOC02240
	NM=8	MOC02250
	K=50*NM	MOC02260
C		MOC02270
	DO 250 J=1,NI,K	MOC02280
	M=MINO(K,NI-J+1)	MOC02290
	N=(M + NM - 1)/NM	MOC02300
	MM=J + N - 1	MOC02310
C		MOC02320
	DO 240 L=J,MM	MOC02330
	IF (MOD(L,10) .NE. 1) GO TO 210	MOC02340
	IF (MOD(L,50) .NE. 1) GO TO 220	MOC02350
	WRITE(6,260)	MOC02360
	GO TO 220	MOC02370
C		MOC02380
	210 WRITE(6,270)	MOC02390
	220 M=L + (NM-1)*N	MOC02400
	225 IF (M .LE. N1) GO TO 230	MOC02410
	M=M - N	MOC02420
	GO TO 225	MOC02430
	230 WRITE(6,280) (I,NAM(I),I=L,M,N)	MOC02440
	240 CONTINUE	MOC02450
	250 CONTINUE	MOC02460
C		MOC02470
	260 FORMAT(' IDENTIFICATION KEY FOR COMPONENT DATA WRITTEN TO UNIT 10	MOC02480
	&' /)	MOC02490
	270 FORMAT(1X)	MOC02500
	280 FORMAT(2X,8(I5,1X,A8,2X))	MOC02510
C		MOC02520
	RETURN	MOC02530
C		MOC02540
C	*****	MOC02550
C		MOC02560
C	THIS ENTRY WRITES PRELUDE TO MICS ON UNITS 10 AND 7	MOC02570
C		MOC02580
	ENTRY PRELUD(LMDA,TAU)	MOC02590
C		MOC02600
	KK=1	MOC02610
	NT=0	MOC02620
C		MOC02630
	IF (IW(18) .EQ. 0) RETURN	MOC02640
	IF (IFLG(IK) .NE. 0) RETURN	MOC02650
C		MOC02660
	WRITE(10,140) NI	MOC02670
	WRITE(10,350) (LMDA(I),I=1,NI)	MOC02680
	WRITE(10,350) (TAU(I),I=1,NI)	MOC02690
	WRITE(7,140) NI	MOC02700
	WRITE(7,350) (LMDA(I),I=1,NI)	MOC02710
	WRITE(7,350) (TAU(I),I=1,NI)	MOC02720
	RETURN	MOC02730
C		MOC02740
	350 FORMAT(1P8E10.3)	MOC02750
	END	

Table (7.5.2) Variables That Have Been Put on the Common Blocks
in the BACFIRE Code

C C C C C C C C C C C C	<pre> SUBROUTINE BACFR(MARD,NARD) BACFIRE BACFIRE BACFIRE THIS PROGRAM WAS WRITTEN IN DECEMBER, 1975, BY T.J. ROZEK AND C.L. CATE FOR THE NUCLEAR ENGINEERING DEPARTMENT, UNIVERSITY OF TENNESSEE THIS PROGRAM TAKES GENERIC AND SPECIAL CONDITIONS FOR SECONDARY CAUSES OF FAILURE AND LOCATES POSSIBLE CRUCIAL COMMON CAUSES FOR SUPPLIED CUT SETS AND MAP LOCATIONS. IMPLICIT INTEGER*2 (A-Z) REAL*8 C,SUS,CSM,MANU,CHEND,A2,CGCS,CSC,CMFG,NONE,CHBL8 INTEGER NUM INTEGER MAST,NAST INTEGER DD1,DD2 INTEGER*4 LOC,MAP,LO,NIN,BUG,QMAP INTEGER*4 SAME LOGICAL YES,PRNT1 DIMENSION MAXIMP(30,2),SUMIMP(30,2),AVGIMP(30,2),MINIMP(30,2) DIMENSION C(101),MFG(101),LG(101,10),GCS(101,4,5) DIMENSION LOC(1001),MAP(1001,10),IMP(100,4,5),MIMP(4,10,5) DIMENSION ASSUS(101),CTG(101),MP(101),SUS(101,3) DIMENSION CGCS(30,3),CSC(5,3) DIMENSION MANU(37,3),AMAN(37) DIMENSION CSM(10),DA(10,10),D(10),CAT(4,10,5),D1(10) COMMON /LT/ NCUT,NCOMP,NIN,NLINE,D,LO,LOC,C,CAT,GCS COMMON /LY/ MIMP,IMP,A2,IMPCG,ASSUS,CTG,MAP,MP COMMON/LP/ YES,QMAP,SUS COMMON/LK/MAIMP,SIMP,NUM,MFG,AMAN COMMON/LO/ MANU,DD1,DD2 DATA CHBL/' '//'CHBL8/' '//'PRNT2/'P'//N/1/ DATA CHEND/'END '//'CHAST/'*'/ COMMON/BK/ CSM DATA SAME/'SAME'/ COMMON /IMPO/ MAXIMP,SUMIMP,AVGIMP,MINIMP COMMON /ONE/ CGCS,CSC,CMFG COMMON MAP,NCSM,DA COMMON/TWO/NAST,MAST DATA BLOCK LIMITS. CAN BE REDUCED OR ENLARGED WITH APPROPRIATE ARRAY DIMINSIONS ALSO REQUIRING CHANGE. MAST=MAST+1 IF(MARD.GT. 1) GO TO 209 IF(MAST.GT. 1) GO TO 209 NCUT=0 MNCOMP=100 MNCMMP=100 </pre>	MOC02780 MOC02790 MOC02800 MOC02810 MOC02820 MOC02830 MOC02840 MOC02850 MOC02860 MOC02870 MOC02880 MOC02890 MOC02900 MOC02910 MOC02920 MOC02930 MOC02940 MOC02950 MOC02960 MOC02970 MOC02980 MOC02990 MOC03000 MOC03010 MOC03020 MOC03030 MOC03040 MOC03050 MOC03060 MOC03070 MOC03080 MOC03090 MOC03100 MOC03110 MOC03120 MOC03130 MOC03140 MOC03150 MOC03160 MOC03170 MOC03180 MOC03190 MOC03200 MOC03210 MOC03220 MOC03230 MOC03240 MOC03250 MOC03260 MOC03270 MOC03280 MOC03290 MOC03300

C
C
C
C

Figure 7.5.3

List of Statements Changed in the CUT-SET and
TOP-SET Procedures of the MODCUT Code

```

PUT FILE(Y) EDIT(J)(F(4));
DO N=1 TO MISH;
  IF(SUBSTR(CUTSET.COMP(1),N,1)='1'B) THEN
  IF(CUTSET.LIL(N)<10000) THEN PUT FILE(Y) EDIT(COMPNAME(CUTSET.LIL(N)))
    (X(1), (12));
  ELSE DO;
    MA=CUTSET.LIL(N);
    DA=-CEIL(-MA/10000);
    JA=-CEIL(-MA/1000);
    JAK=JA-10*DA;
    NA=MA-1000*JA;
    IF(JAK=1) THEN PUT FILE(Y) EDIT(REPNAME(NA))(X(1),A(12));
    ELSE PUT FILE(Y) EDIT(' '||REPNAME(NA))(X(1),A(12));
  END;
END;
END;
END;
END;

```

In the statements shown in Table(7.5.3), the Variable J is changed through a DO loop and indicates the number of components contained in the minimal cutset. Y is the name of the file over which the minimal cutsets are to be printed. The file Y is declared as follows in the CUT-SET and TOP-SET procedures:

```
DECLARE(Y)FILE EXTERNAL;
```

The statements by which the BACFIRE code reads the cutsets have also been changed to the following form:

```
C      READ DATA SET FIVE HARDWARE MINIMAL CUTSETS.
221    READ (5, 434) A1
      IF (A. NE. CHAST) GO TO 221
C      READ IN MINIMAL CUTSETS AND COMPONENTS
231    READ (9, *, END = 661) NCSM, (CSM(I), I = 1, NCSM)
      IF (NCSM.EQ.CHEND) GO TO 661
      .
      .
      .
```

It can be seen that the cutsets are read from unit 9 by the BACFIRE code, and therefore unit 9 can be defined to be the file Y which is created by the MODCUT code prior to the execution of the BACFIRE Code. By using the CMS of the IBM one can use a file with a file name MOD-BAC and a file type EXEC to store all necessary commands for coupling the MODCUT and BACFIRE Codes. If the file type of a CMS file is EXEC then all of the commands that are stored in that

file can be applied by issuing only the name of the EXEC File (e.g., MOD-BAC here). For compiling MODCUT and BACFIRE, the MOD-BAC EXEC file stores the following commands.

TOF:

```
FILEDEF SYSIN DISK TREE DATA (LRECL 80 RECFM FB BLOCK 800
```

```
FILEDEF Y      DISK Y      DATA (LRECL 80 RECFM FB BLOCK 800
```

```
MODCUT
```

```
FILEDEF 05 DISK BACFR DATA
```

```
FILEDEF 09 DISK Y      DATA
```

```
FILEDEF 11 DISK .
```

```
FILEDEF 12 DISK
```

```
BACFIRE
```

EOF:

In the first and second commands, the file tree data in which fault tree data are stored for use in the MODCUT code, and file Y data over which MODCUT writes the minimal cutsets are defined, respectively. The MODCUT module which is stored in the file MODCUT MODULE is then executed. At this point all of the cutsets are generated by the MODCUT code and transferred to the file Y data.

Next, files BACFR DATA and Y DATA are defined as units 5 and 9 respectively. Units 11 and 12 are also defined to be internally used by the BACFIRE code. Finally, the BACFIRE command executes the BACFIRE code and its output will be printed. All of these commands are operated by issuing the command MOD-BAC. Therefore, one should only provide the TREE DATA and BACFR DATA files for the input fault tree and input common cause failure data, and issue the MOD-BAC command to get the common cause candidates.

REFERENCES

- [1] Olmos, J. and L. Wolf, "A Modular Approach to Fault Tree and Reliability Analysis", MITNE-209, Dept. of Nucl.Eng., MIT, August 1977
- [2] Pollack, S. V. and T. D. Sterling, "A Guide to PL/I", Holt, Rinehart and Winston, Second Edition, 1976
- [3] Fike, C. T., "PL/I for Scientific Programmers", Prentice-Hall, 1970
- [4] Fussell, J. B., et. al., "MOCUS - A Computer Program to Obtain Minimal Sets from Fault Trees", Aerojet Nuclear Co., ANCR-1156 (August 1974)
- [5] Cate, C. L. and J. B. Fussell, "BACFIRE - A Computer Code for Common Cause Failure Analysis", University of Tennessee, Knoxville, Tennessee, May 1977

CHAPTER 8

APPLICATION OF THE CODES PL-MODT, PL-MODMC, AND MODCUT
TO SOME SPECIAL PROBLEMS OF INTEREST IN REACTOR SAFETY8.1 Introduction

In Chapters 2, 4 and 5 the mathematical models used in the codes PL-MODT, PL-MODMC and MODCUT were discussed. Also, the computer code developments with regard to these three codes were verified and discussed in Chapter 7. In this chapter several examples will be evaluated by using these codes. Some of these examples are also solved by using other codes, so as to provide a basis for evaluating the efficiency, adequacy and the capabilities of these three PL-MOD based codes. For most of the examples the CPU time spent by any of these codes is given along with the output of results from the codes.

8.2 Examples for the Code PL-MODT

In order to more easily comprehend the use of the PL-MODT code, four examples are presented in this section. These four examples are as follows:

- (1) Low Pressure Recirculation System (LPRS) for PWRs
(All components are of Class 1)[3].
- (2) An arbitrary fault tree has been constructed and is used to determine the unavailability of the top event by assuming, first, that all components are of Class 2, and then that they are a combination of Class 2 and Class 3.

The results for this example are compared with the results stemming from the KITT-1 Code.

(3) An Auxiliary Feedwater System is calculated by the PL-MODT code and the results are compared with those calculated by the FRANTIC Code (all components are Class 4)[4].

(4) A Simple Electric Circuit System is evaluated and the results are compared to those of the FRANTIC Code. This simple system is discussed in Reference [5].

8.2.1 Low Pressure Recirculation System (LPRS) for PWRs (Class 1)

This system is part of the Emergency Core Cooling Recirculation System and consists of the containment pump and two redundant pumps in parallel. Figure(8.2.1)shows a simplified flow diagram and Figure (8.2.2)shows its associated reduced fault tree. It should be referred to in connection with the following discussion. All valves with the exception of V_{24} and V_{25} are aligned for injection into the cold legs. V_{24} and/or V_{25} should be open in order to start the LPRS and close V_2 . These valves, whether locally or remotely operated, can be manually operated. Details of the system as well as of its fault tree can be obtained from WASH-1400 [3].

The LPRS unavailability estimates are given in WASH-1400 as follows:

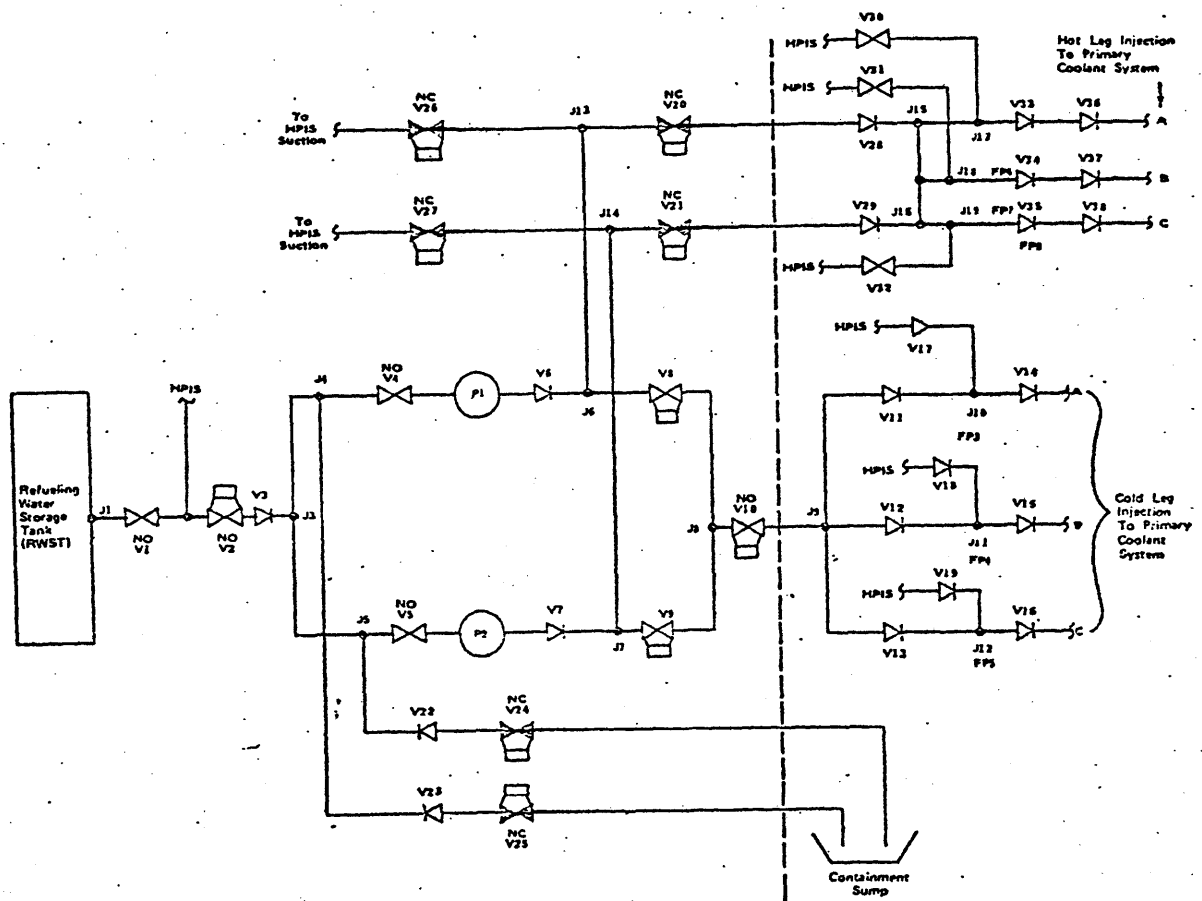


Figure (8.2.1) Simplified Flow Diagram of LPRS [3]

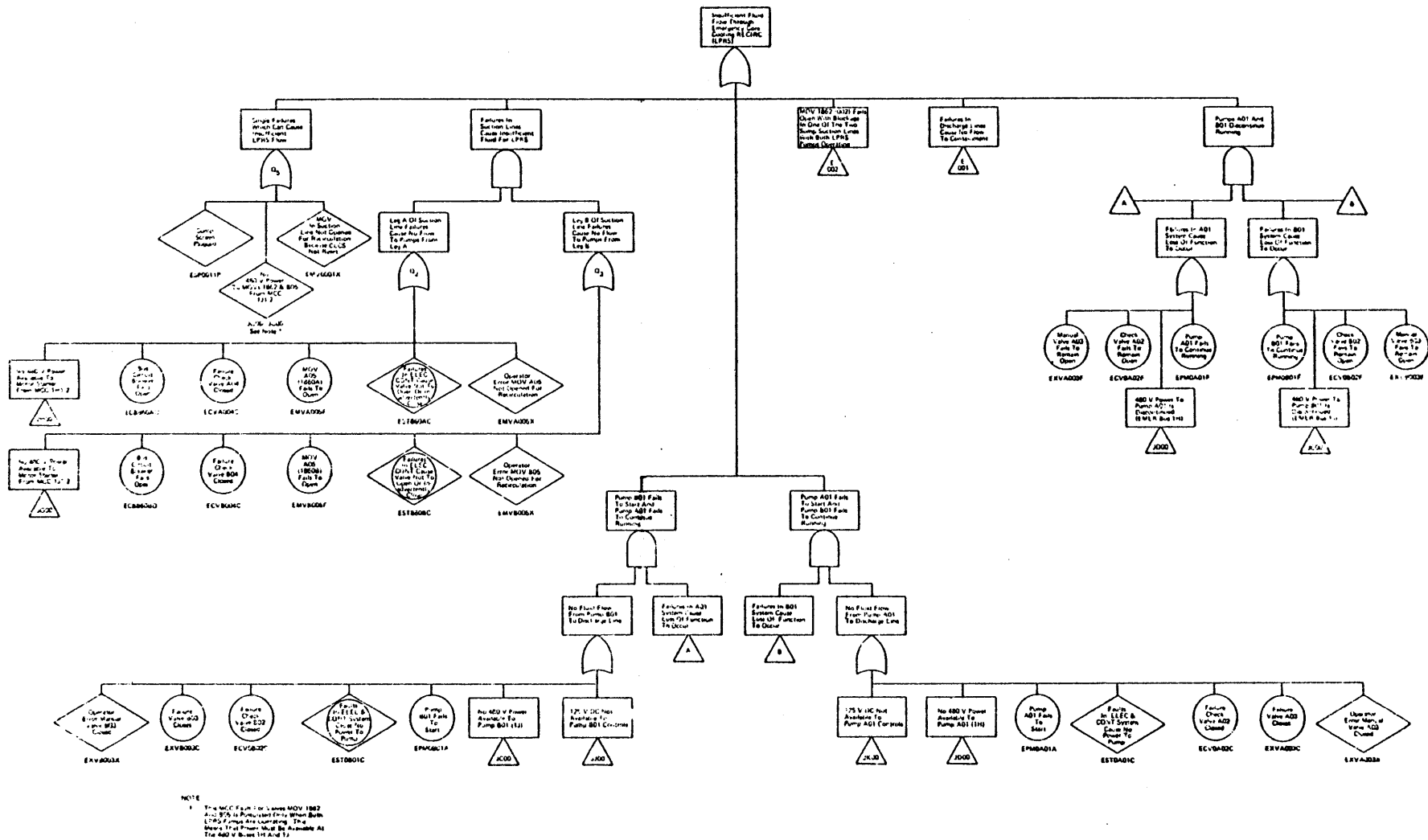
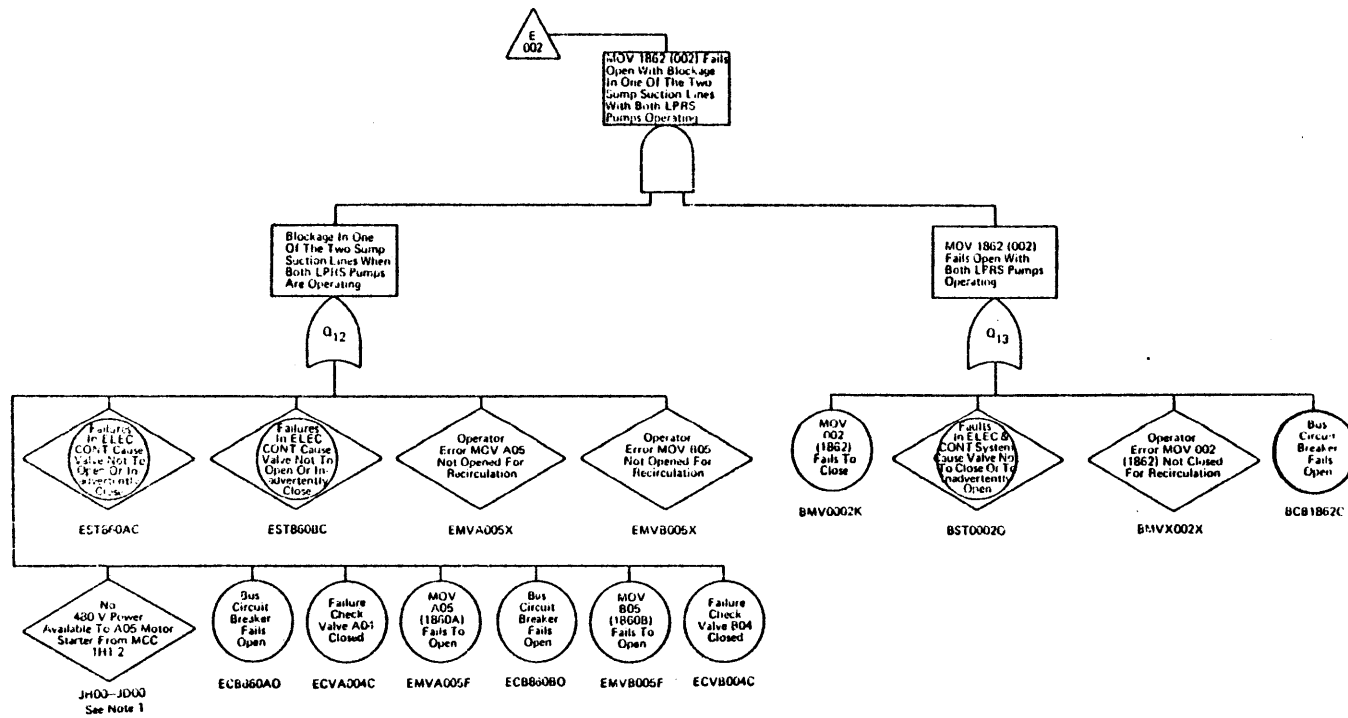


Figure (8.2.2) LPRS Reduced Fault Tree (Sh. 1) [3]



NOTE:

1. These MCC Faults Postulated Only When Both LPRS Pumps Are Operating. This Means That Power Must Be Available At The 480 V Buses 1H And 1J.

Figure (8.2.2) continued

LPRS Reduced Fault Tree (Sh. 2)

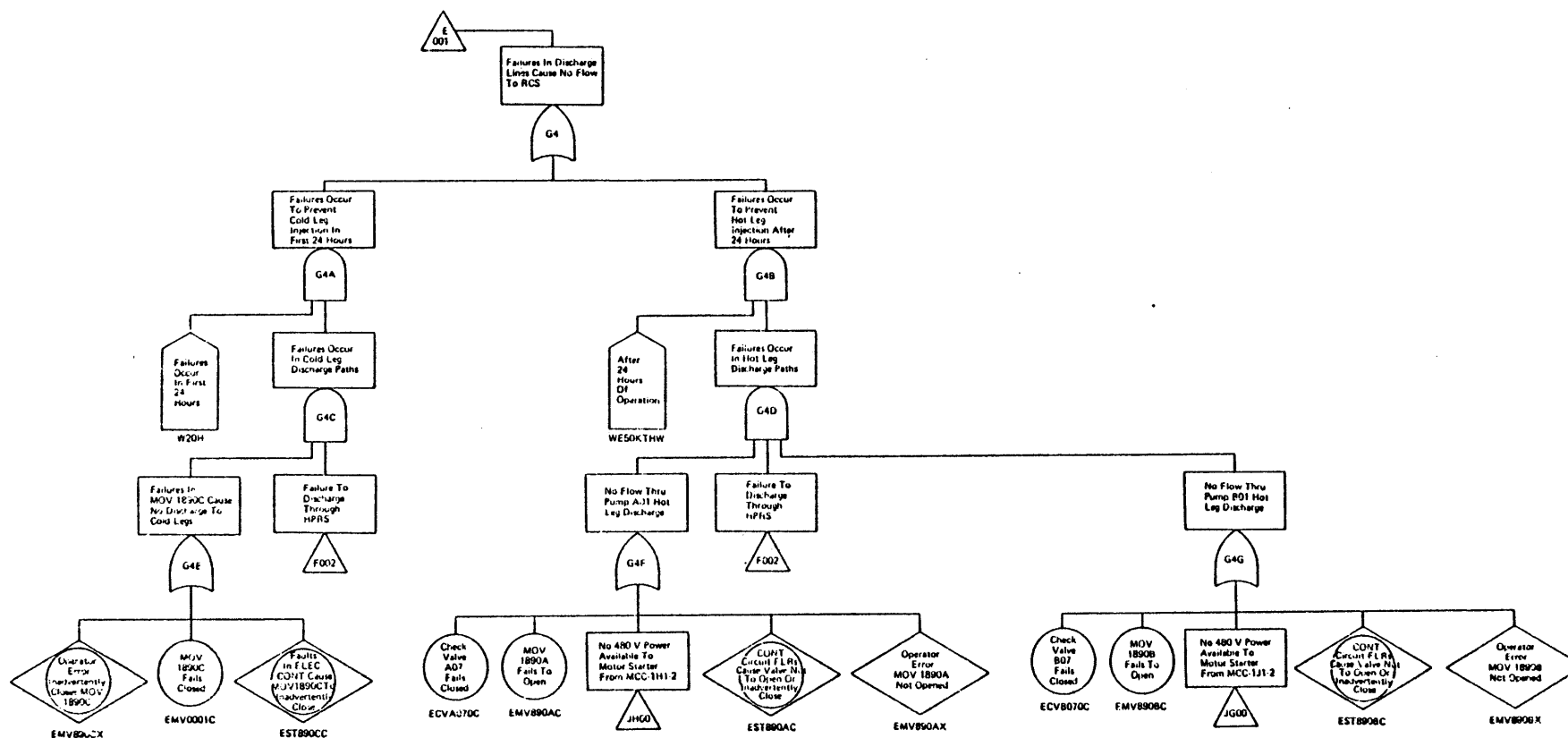


Figure (8.2.2) continued

LPRS Reduced Fault Tree (Sh. 3)

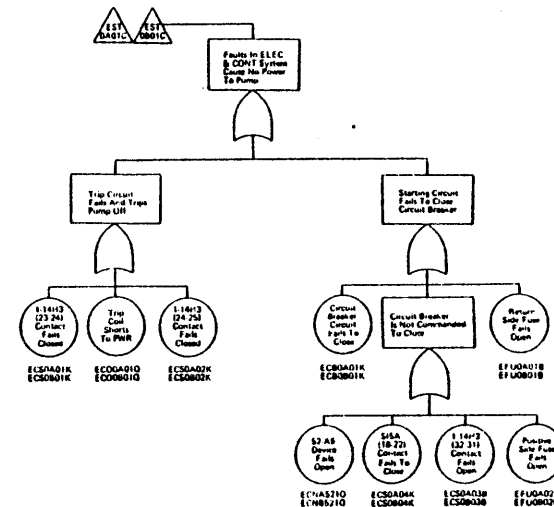
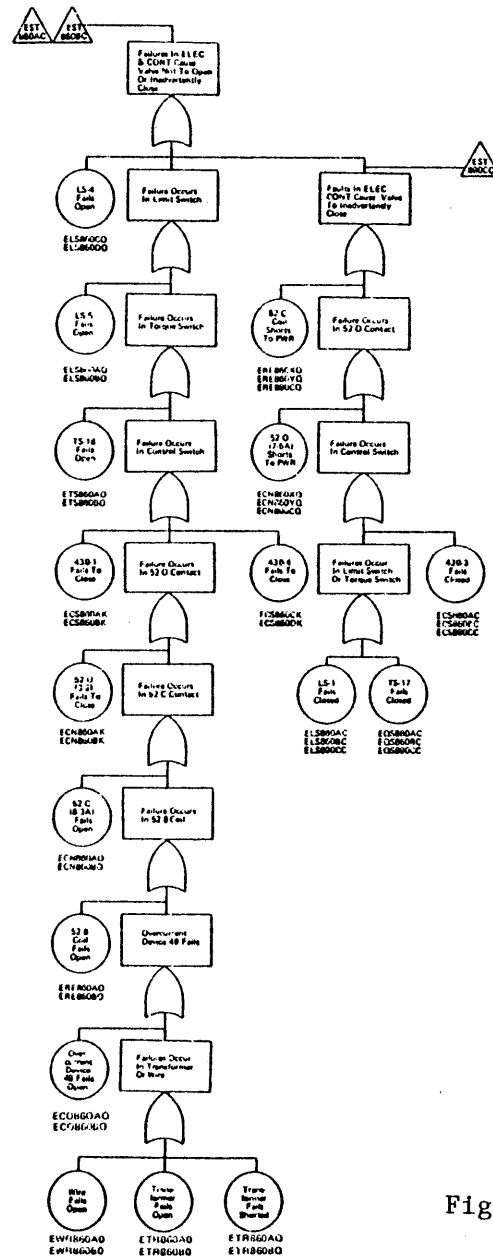


Figure (8.2.2) continued LPRS Subtrees

$$Q_{\text{med}}^{\text{WASH}} = 1.3 \times 10^{-2}$$

$$Q_{\text{lower}}^{\text{WASH}} = 4.4 \times 10^{-3}$$

$$Q_{\text{upper}}^{\text{WASH}} = 2.7 \times 10^{-2}$$

where the lower and upper bounds were evaluated by a Monte-Carlo simulation by using the minimal cut-set approach. The point estimates for the unavailability as a result of single and double failures, test and maintenance as well as common mode failures are:

$$Q_{\text{single}} = 1.1 \times 10^{-5}$$

$$Q_{\text{double}} = 2.7 \times 10^{-3}$$

$$Q_{\text{test \& main}} = 1.0 \times 10^{-4}$$

$$Q_{\text{common}} = 6.0 \times 10^{-3}$$

respectively.

This system has been re-analyzed by PL-MODT to determine the point estimate probability for the occurrence of the top event and the Vesely-Fussell importance of the components. The reduced fault tree contains a total of 61 non-replicated basic events, 4 replicated events and 2 replicated modular gates.

The point unavailability computed by PL-MODT using the modular approach is

$$Q = 2.83 \times 10^{-3}$$

The total computation time (CPU) for this example was 0.46 sec. and included the modularization, the evaluation of the top event probability and the determination of the importance measure for all components and modules in the fault tree. The results obtained for the unavailability of the top event by using the PL-MODT code is in close agreement with those of the WASH-1400. It should be noticed that the top event unavailability obtained by the PL-MODT code shows only random failure contribution and does not include the common cause contribution (i.e. $Q_{\text{WASH-1400}} = 2.81 \times 10^{-3}$).

8.2.2 Comparison Between the Results of the Codes PL-MODT and PREP & KITT-1 (Components are of Class 2 and 3)

This comparison is based upon an example considered in Ref. [6]. The associated fault tree is shown in Figure (8.2.3). It has been analyzed and evaluated by PL-MODT for the data summarized in Table (8.2.1) by assuming all components have a time-dependent failure behavior, but are non-repairable (i.e. Class 2).

Figure (8.2.4) compares the results of the two codes. Those for PREP & KITT are taken from Reference [6]. For the first 3000 hours PL-MODT calculates unavailabilities which are higher than those computed by PREP & KITT. Thereafter, the trend reverses and PREP & KITT gives higher values. Which code comes closer to the exact answer can only be answered by benchmarking these codes against a code which employs the Markovian approach.

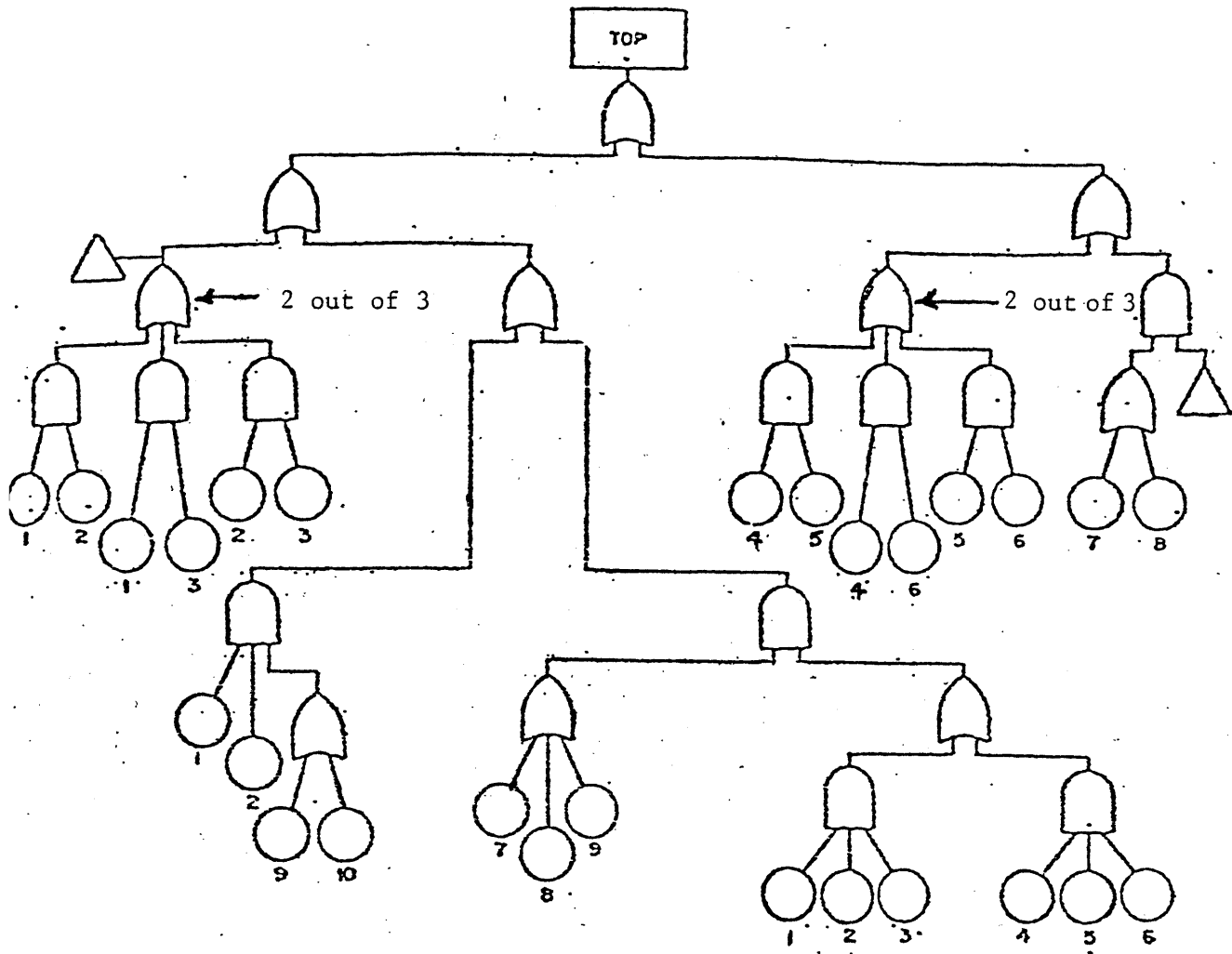


Figure (8.2.3) Fault Tree Example [6]

TABLE 8.2.1

PRIMARY FAILURE RATES FOR SAMPLE
FAULT TREE SHOWN IN FIGURE (8.2.3)

<u>Primary Failure Index</u>	<u>$\lambda(\text{hr}^{-1})$</u>
1	2.6×10^{-6}
2	2.6×10^{-6}
3	2.6×10^{-6}
4	3.5×10^{-5}
5	3.5×10^{-5}
6	3.5×10^{-5}
7	3.5×10^{-6}
8	5.0×10^{-6}
9	8.0×10^{-6}
10	8.0×10^{-6}

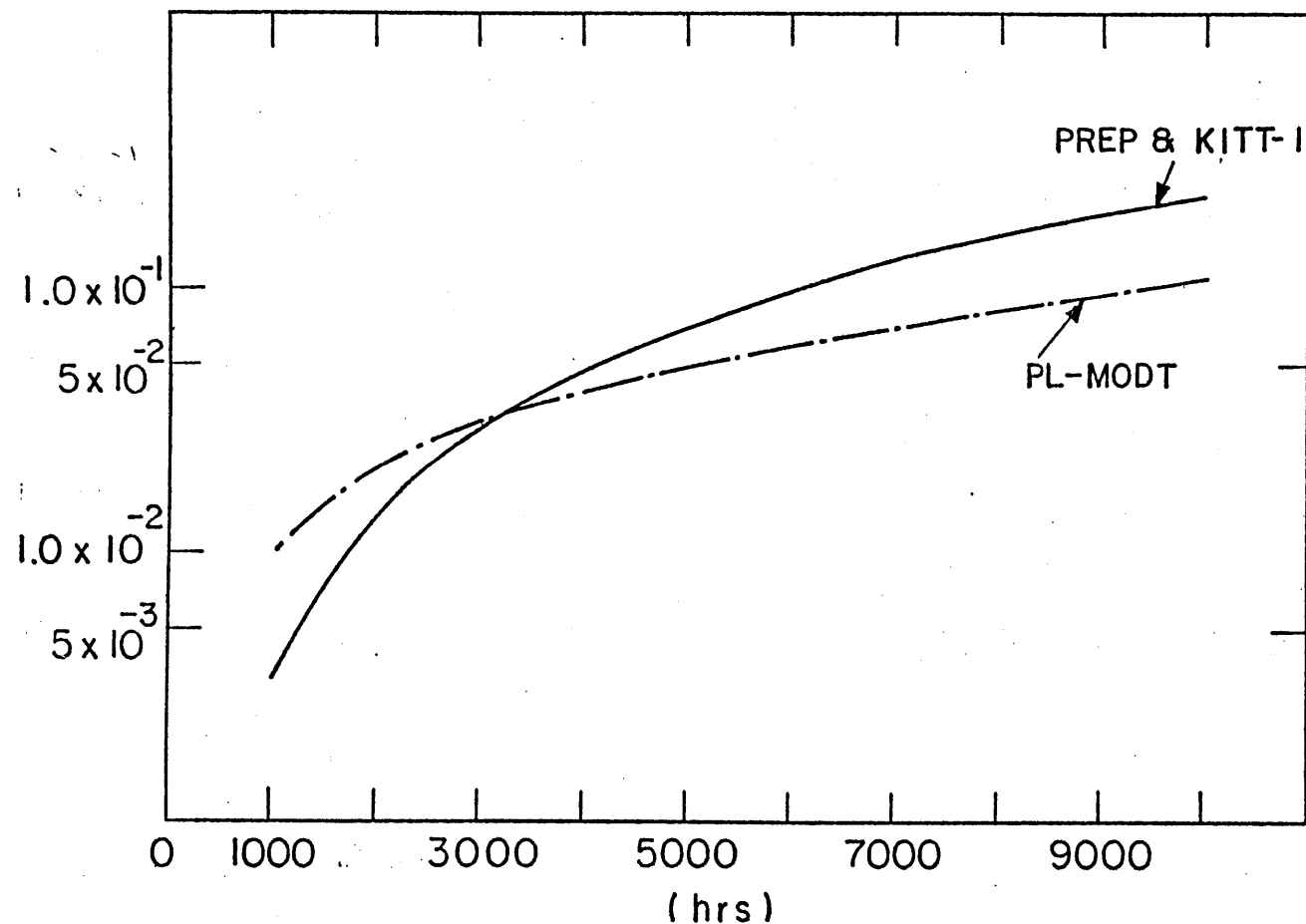


FIGURE 8.2.4 COMPARISON OF THE TIME-DEPENDENT UNAVAIL-
ABILITIES FOR THE SAMPLE TREE GIVEN IN
FIGURE 8.2.10 AS CALCULATED BY PREP AND PL-MODT

The computation time for PL-MODT was 0.64 seconds for modularizing the tree, finding the unavailabilities for the top event, components and modules for 10 time steps. Advantage was taken of the fact that PL-MODT is capable of handling directly K-out-of-N gates. Therefore, the two 2/3 gates were input to PL-MODT, rather than the one shown in Figure (8.2.3). This approach naturally saves computer storage and computation time.

The same fault tree in Figure (8.2.3) is used to evaluate the top event unavailability for the case that components 1 to 6 in Figure (8.2.3) are repairable and 7 to 10 are non-repairable (i.e. a combination of Class 2 and 3 components). The data listed in Table (8.2.2) are used for this analysis.

First, KITT-1 is used and its results compared with PL-MODT. Next, FRANTIC results are obtained for the same example. Results are shown in Figure (8.2.5).

Small differences exist due to different approximations used in these codes. However, these three codes give essentially the same asymptotic values for unavailabilities. The job running time for PL-MODT to calculate the top event unavailability for 15 time steps was 0.62 seconds. The curve labeled EXACT demonstrates the unavailability calculated by using Equations (2.3.2) and (2.4.9) for Class (2) and Class (3) components, respectively. FRANTIC uses the asymptotic value for Class (3) components throughout the system operation (i.e. $Q \approx \lambda\tau$). KITT-1 uses some approximations that result in an unavailability closer to the exact unavailability. The PL-MODT prediction for the system unavailability is much closer to the exact value of the system unavailability,

Table (8.2.2)

FAILURE AND REPAIR RATES FOR SAMPLE TREE IN FIGURE (2.8.3)

<u>Primary Failure Index</u>	<u>(hr⁻¹)</u>	<u>(hr⁻¹)</u>
1	2.6×10^{-6}	4.1×10^{-2}
2	2.6×10^{-6}	4.1×10^{-2}
3	2.6×10^{-6}	4.1×10^{-2}
4	3.5×10^{-5}	1.66×10^{-1}
5	3.5×10^{-5}	1.66×10^{-1}
6	3.5×10^{-5}	1.66×10^{-1}
7	5.0×10^{-6}	0
8	5.0×10^{-6}	0
9	8.0×10^{-6}	0
10	8.0×10^{-6}	0

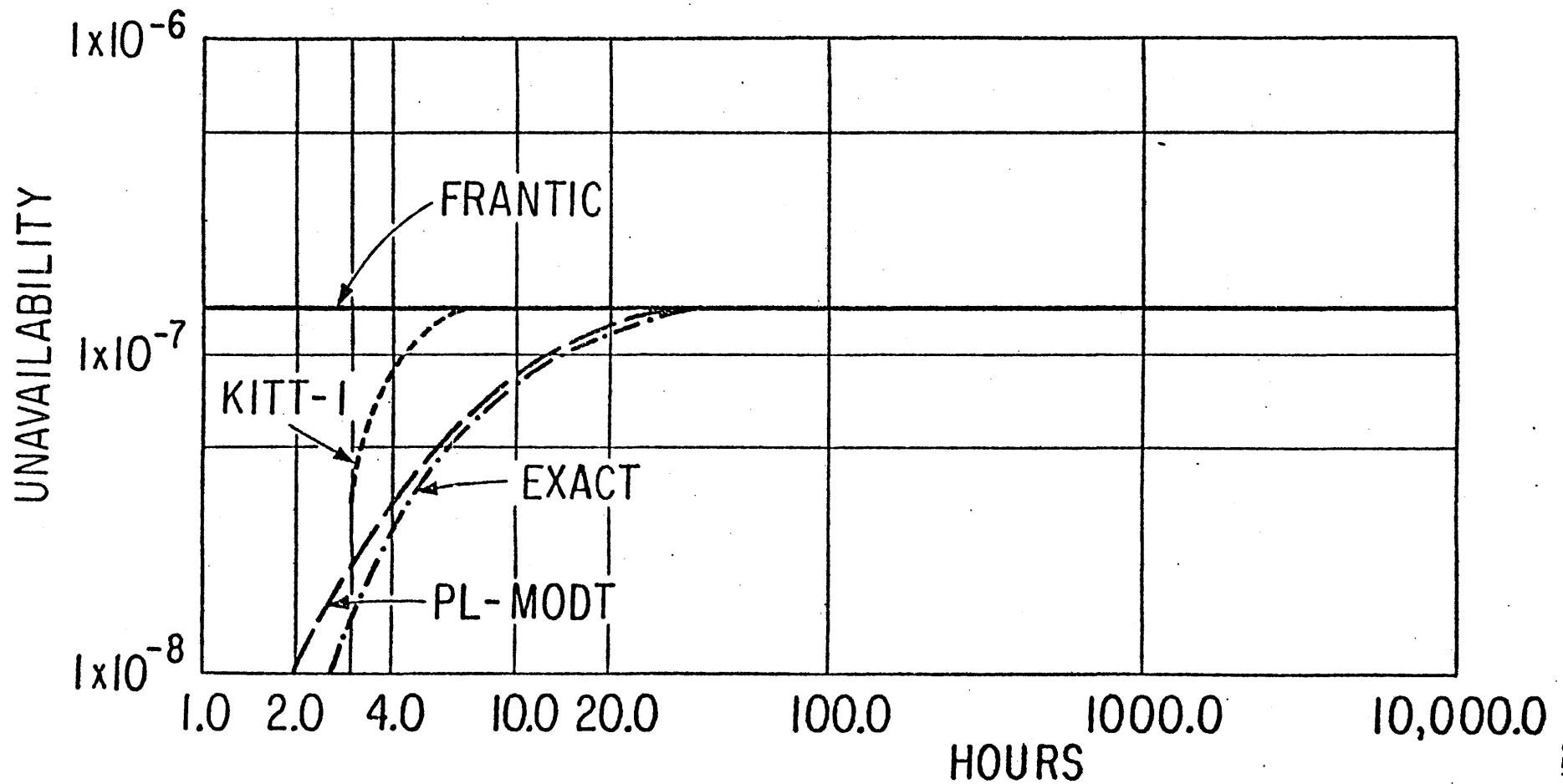


Figure (8.2.5) Comparison of the Time-Dependent Unavailabilities of Repairable Components for the Sample Tree given in Figure (8.2.3) as Calculated by FRANTIC, PREP, KITT and PL-MODT

because of its more adequate approximations for obtaining the time-dependent unavailability functions for the Class 3 components

8.2.3 Auxiliary Feedwater System - A Comparison Between PL-MODT and FRANTIC (Class 4)

A simplified block diagram of the Aux-Feed System of a PWR is shown in Figure (8.2.6) and is described in detail in Reference [4]. As can be seen from Figure(8.2.6), the system consists of two diesel driven pumps in parallel with a pump and two valves. The pump and valves are in series. The block diagram shown in Figure (8.2.6) is a simplified version of the one shown in WASH-1400. It is assumed that all of the components of the system are periodically tested. The data are summarized in Table (8.2.3).

Figure (8.2.7) compares the results for the unavailability computed by the two codes FRANTIC and PL-MODT. It should be noticed that PL-MODT gives slightly higher values for the unavailability of the system during the operation time.

The lower value obtained for the unavailability by FRANTIC is due to the fact that FRANTIC uses the approximate versions of the general unavailability functions used in PL-MODT. However, the difference is negligible for small failure rates. During the inspection time of the components in the AUX-Feed system fault tree (e.g. at $720 \leq t \leq 721.5$ for pump), both codes give essentially the same unavailability. However, for the repair time interval, a larger difference appears, as can be seen from Figure (8.2.8). This is mainly due to the fact that the continuous analytical equation for Class 4 components in PL-MODT

Figure (8.2.6)

Block Diagram of the Simplified Aux-Feed System [4]

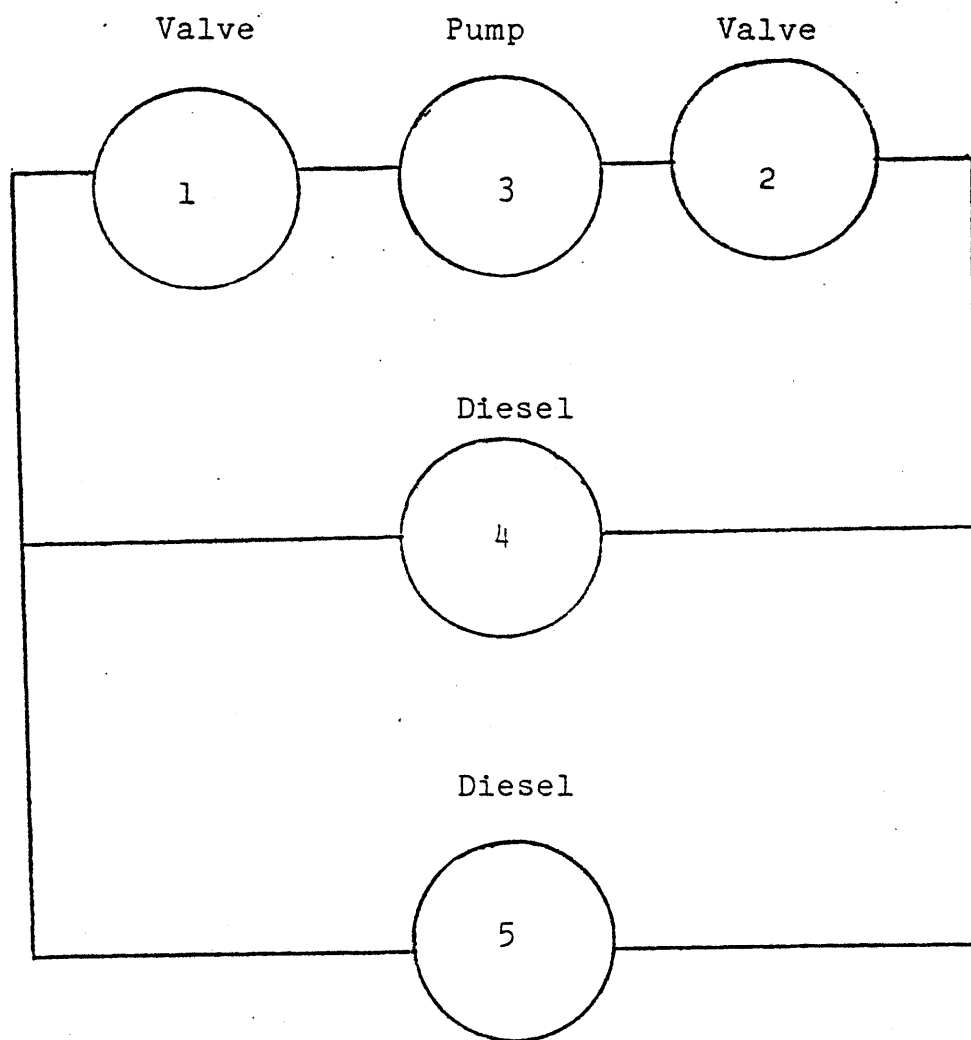


Table (8.2.3)

THE COMPONENT INPUT DATA FOR THE CODES FRANTIC AND PL-MODT [4]

<u>Component Number</u>	<u>Component Name</u>	<u>Failure Rate (λ) (x 10^{-6})</u>	<u>Test Interval (η) (days)</u>	<u>First Test Interval q.T.</u>	<u>Inspection Duration (θ) (hours)</u>	<u>Repair Duration (τ) (hours)</u>	<u>(q_0) Override Unavailability</u>
1	Valve	0.3	30	30	1.5	7	1.0
2	Valve	0.3	30	30	1.5	7	1.0
3	Pump	3	30	30	1.5	19	1.0
4	Diesel	42	60	30	1.5	21	1.0
5	Diesel	42	60	60	1.5	21	1.0

Figure (8.2.7)

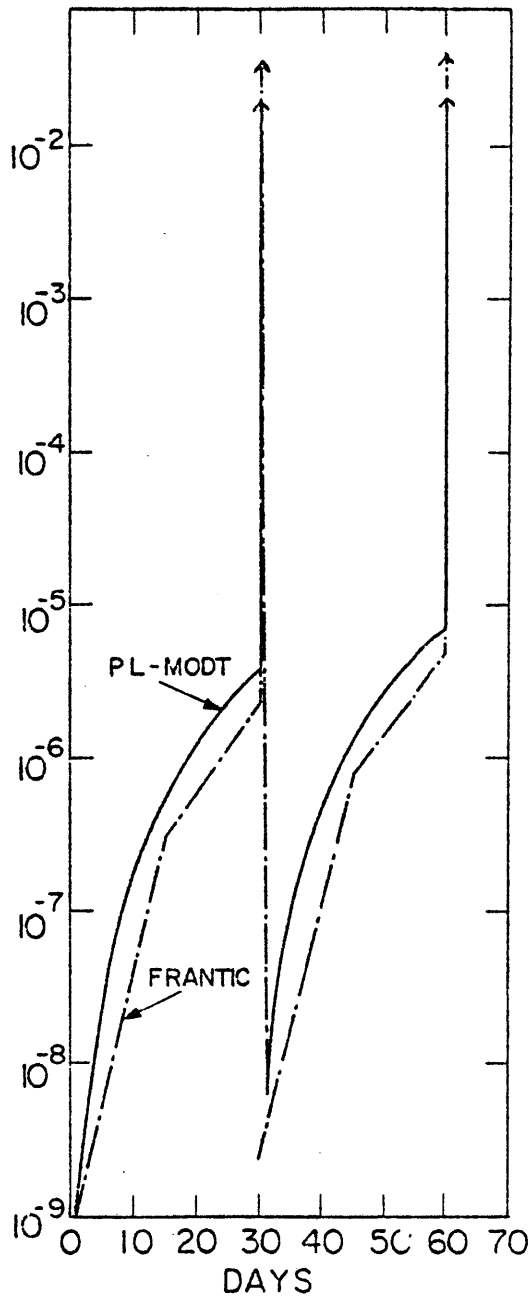
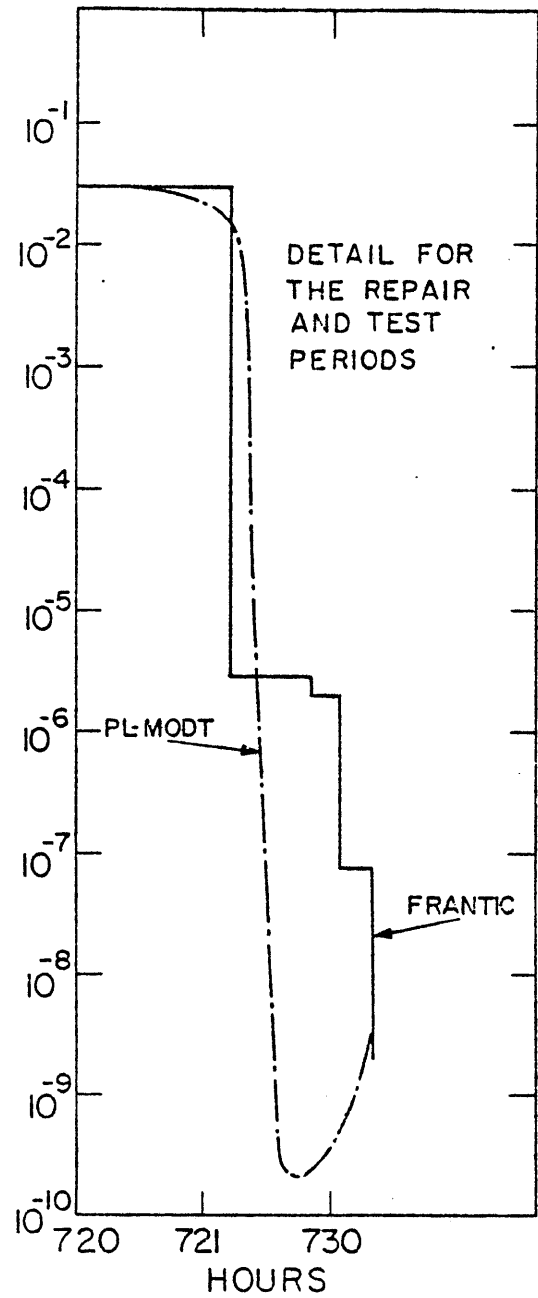


Figure (8.2.8)



COMPARISON BETWEEN THE TIME-DEPENDENT UNAVAIL-
ABILITIES FOR THE AUX-FEED SYSTEM AS CALCULATED
BY FRANTIC AND PL-MODT

(i.e. Equation (2.5.8) gives a lower value of the unavailability due to inspection. This difference vanishes as the end of the inspection period is approached. Thereafter, both codes predict about the same value for the unavailability as depicted in Figure (8.2.8). To evaluate unavailabilities shown in Figure (8.2.8), a finer time step mesh was used.

PL-MODT treats the group "valve 1, pump, and valve 2" as a module and therefore the unavailability of this module will be given automatically in the output. Therefore, the unavailability for each branch of the AUX-Feed System block diagram can automatically be obtained from the same computer run for unavailability calculations of the top event.

The CPU time for FRANTIC was 0.57 seconds over a period of 180 days. The CPU time for PL-MODT was 0.57 seconds for modularizing the fault tree and calculating the unavailabilities for 40 time steps as well as the importance for components and modules.

It is noteworthy to mention that as the number of components in the fault tree increases and more inspected components become involved, the differences between PL-MODT and FRANTIC during the repair time vanishes. This will become more apparent in the next example.

8.2.4 Example of a Simple Electric Circuit Using FRANTIC and PL-MODT

Figure (8.2.9) shows a simple electric system which has been discussed in Reference [5]. The purpose of the system is to provide light by the bulb when the switch is closed, the relay 1 contacts closed and the contacts of relay 2 (a normally closed relay) are open. Should

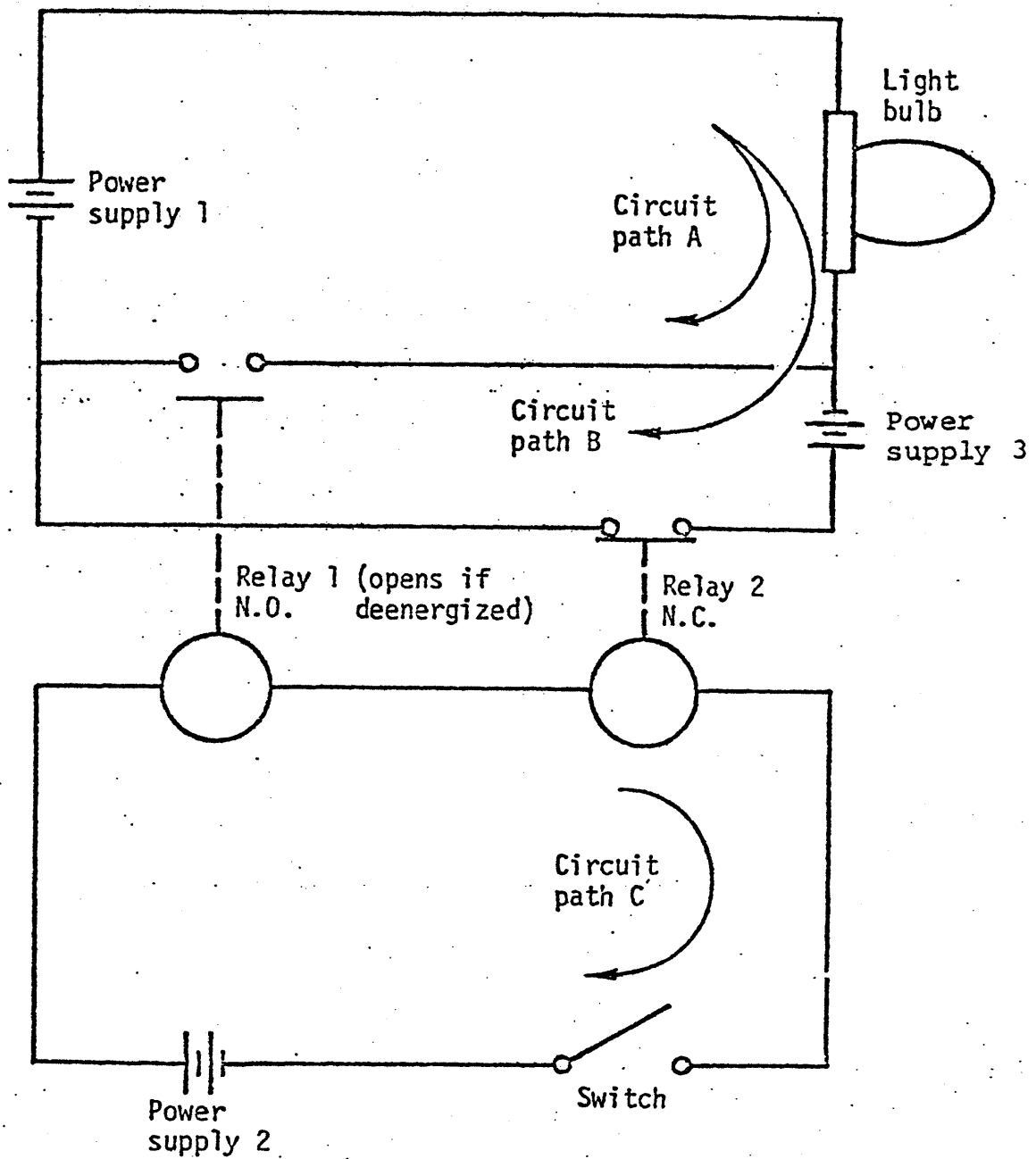


Figure (8.2.9) Sample System for Mutually Exclusive Events [5]

the contacts of relay 1 open, the light will go out and the operator will immediately open the switch which in turn causes the contacts of relay 2 to close, which restores the light. In what follows, operator failures, wiring failures as well as secondary failures will be neglected. The fault tree for this system is shown in Figure (8.2.10).

Failure rates, repair times and test periods for the various components are summarized in Table (8.2.4). It should be noted that no data base exists for the data in Table (8.2.4) and they are chosen arbitrarily. Also, no replicated component or module exists in this system.

In order to enable FRANTIC to analyze this system, the unavailability function of the system must be provided as input. This function was found to be:

$$QS = \{1.0 - (1 - Q(1))\{1 - [(1 - Q(2))(1 - Q(4))(1 - Q(7))(1 - Q(8)) * (1 - Q(9))(1 - Q(10))][1 - (1 - Q(3))(1 - Q(6))(1 - Q(5))]\}$$

Naturally, for PL-MODT the fault tree was directly input, because there is no need for a system function. As output, PL-MODT modularizes the tree and gives the following four modules.

OR Module #4: components 5, 6 and 3

OR Module #3: components 4, 7, 8, 9, 10 and 2

AND Module #2: modules 4 and 3

OR Module #1: component 1 and module 2

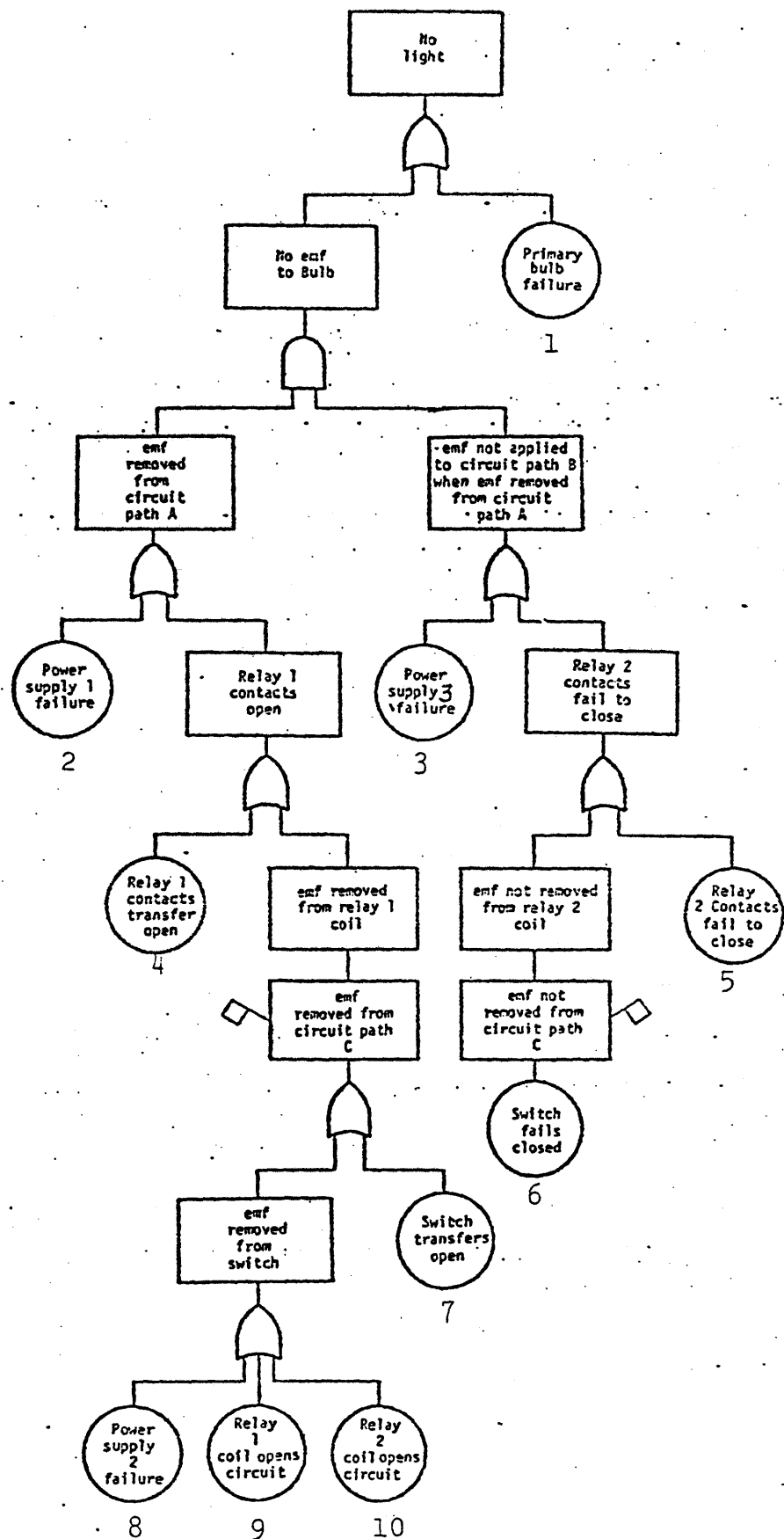


Figure (8.2.10) Fault Tree for Sample System in Figure (8.2.9) [5]

TABLE 8.2.4

INSPECTION PARAMETERS FOR THE CIRCUIT EXAMPLE

<u>Component Number</u>	<u>Failure Rate hr⁻¹</u>	<u>Inspection Time (hrs)</u>	<u>First Time Interval (days)</u>	<u>Repair Time (hrs)</u>	<u>Override Unavailability</u>	<u>Time Interval (days)</u>
1	2.0×10^{-4}	1.0	7	2.0	1.0	7
2	2.8×10^{-5}	0.5	7	1.0	1.0	14
3	2.8×10^{-5}	1.0	7	1.5	1.0	14
4	3.2×10^{-3}	0.5	7	2.0	1.0	7
5	4.1×10^{-4}	0.5	14	1.5	1.0	21
6	3.2×10^{-4}	1.0	28	1.5	1.0	28
7	2.8×10^{-3}	0.5	14	2.0	1.0	21
8	2.8×10^{-3}	1.0	14	1.5	1.0	14
9	4.5×10^{-3}	0.5	7	2.5	1.0	7
10	4.5×10^{-3}	1.5	7	3.0	1.0	7

Figure (8.2.11) compares the output of both codes for one complete period of 28 days. As can be seen, the results are overall in very close agreement during the operational period. Differences are attributed to the fact that for this example the failure rates are comparatively high. For these high values, the FRANTIC prediction, by using a linear approximation for the unavailability, is not very reliable at large times.

Again, during the inspection period both codes give essentially the same results. However, for the repair period (see Figure 8.2.12) differences show up again which were observed already in Example (8.2.3). It should be noticed that these differences are not as pronounced as in Example(8.2.3) because more components are involved in the present example.

The CPU time for PL-MODT was 0.98 seconds for modularizing the tree, evaluating the unavailabilities for components, modules and the top event for 32 time steps and for determining the importances for various components and modules.

For the same tree and data, FRANTIC needed 1.12 seconds alone for calculating the system unavailability over the period of 180 days and to determine the mean unavailability of the system over this period. However, it should be noted here that PL-MODT would also be an efficient and convenient code for evaluating large fault trees consisting of periodically tested components or any other class of components. Examples 8.2.3 and 8.2.4 showed that the code is also fairly fast for evaluating small fault trees as compared to other state-of-the-art computer codes.

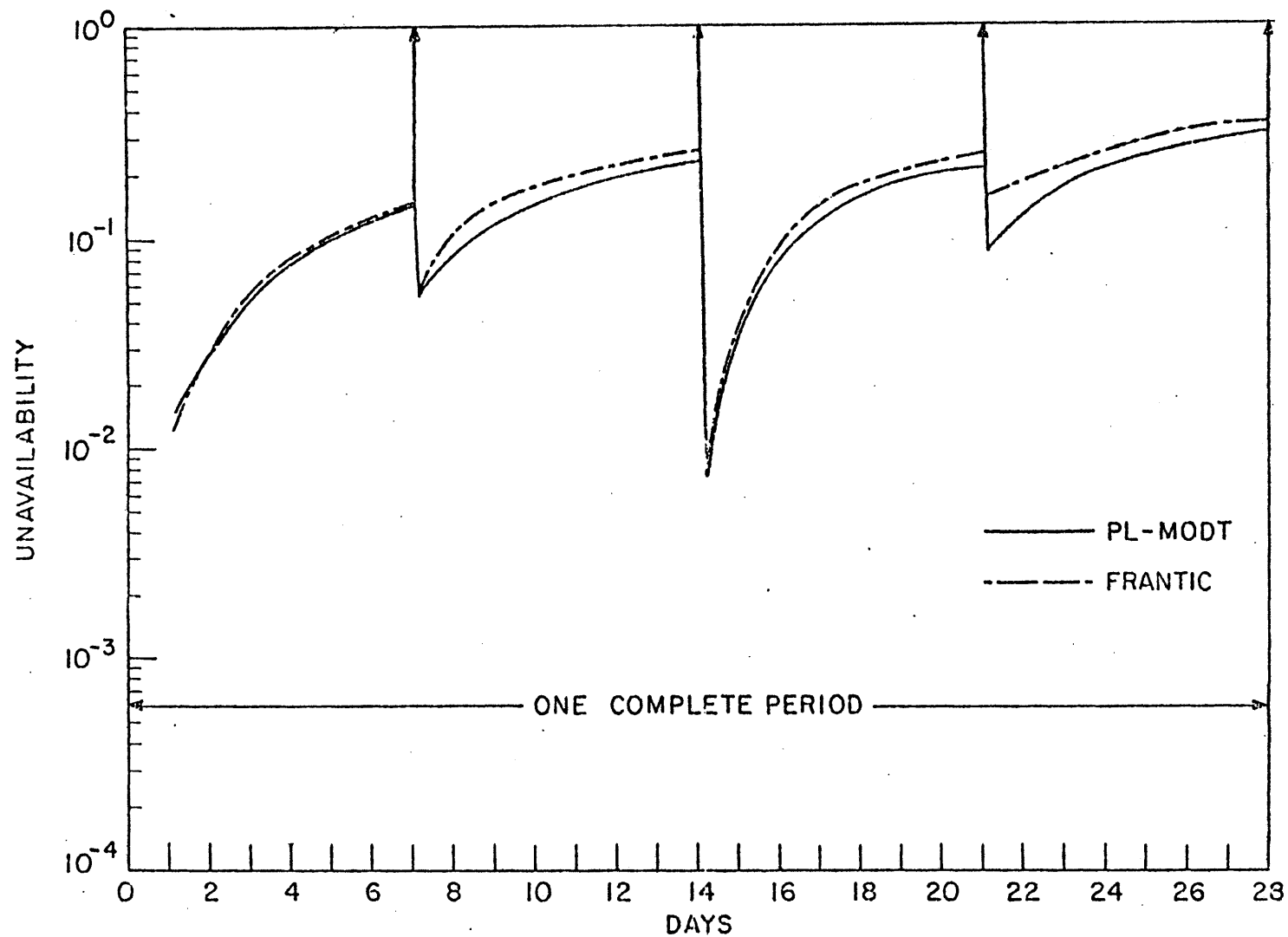


Figure (8.2.11) Comparison Between the Unavailabilities for the Electrical System During Its Operation as Calculated by FRANTIC and PL-MODT

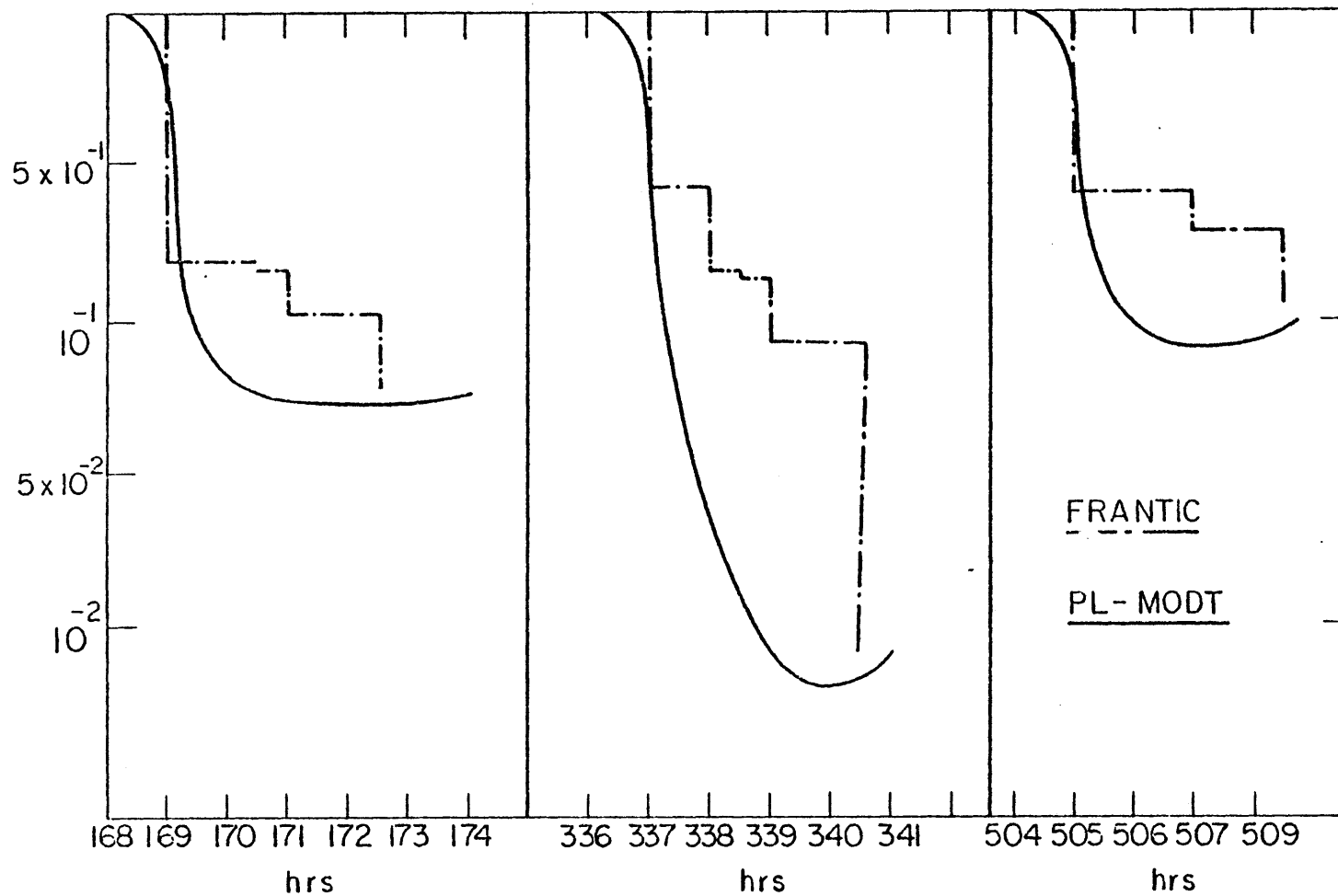


Figure (8.2.12) COMPARISON BETWEEN THE UNAVAILABILITIES DURING TEST AND REPAIR PERIODS AS CALCULATED BY BOTH CODES

8.2.5 Differences Between FRANTIC and PL-MODT

The obvious difference between the two codes is that whereas PL-MODT is capable of analyzing large fault trees and evaluating them at the same time, the use of FRANTIC is mainly confined to the analysis of small systems for which the system unavailability function is known in advance. This off-line approach is not only time consuming but has the additional disadvantage that the user may introduce spurious errors. The mathematical functions used for evaluating system unavailability in PL-MODT are continuous, whereas in FRANTIC they are approximate and noncontinuous.

In a case where a system, for example, the Aux-Feed System, is to be evaluated in more detail than each valve, pump, and diesel would be further developed down to the level of subcomponents which are periodically tested. For this purpose, PL-MODT is especially suited.

8.2.6 Comments on the Vesely-Fussell Importance Measure

PL-MODT enables the user to select the option for the determination of the V.F. importance in steady-state and transient evaluations. The Vesely-Fussell importance measure is defined as the probability that component C_i contributes to the system failure given that the system has failed. As an example, Figure (8.2.13) shows the importance of the pump in the Aux-Feed system as a function of time. The importance stays about constant through the operational period. After a sudden increase in system unavailability due to the testing of the pump, the valves, and the diesel, the importance of the pump sharply decreases,

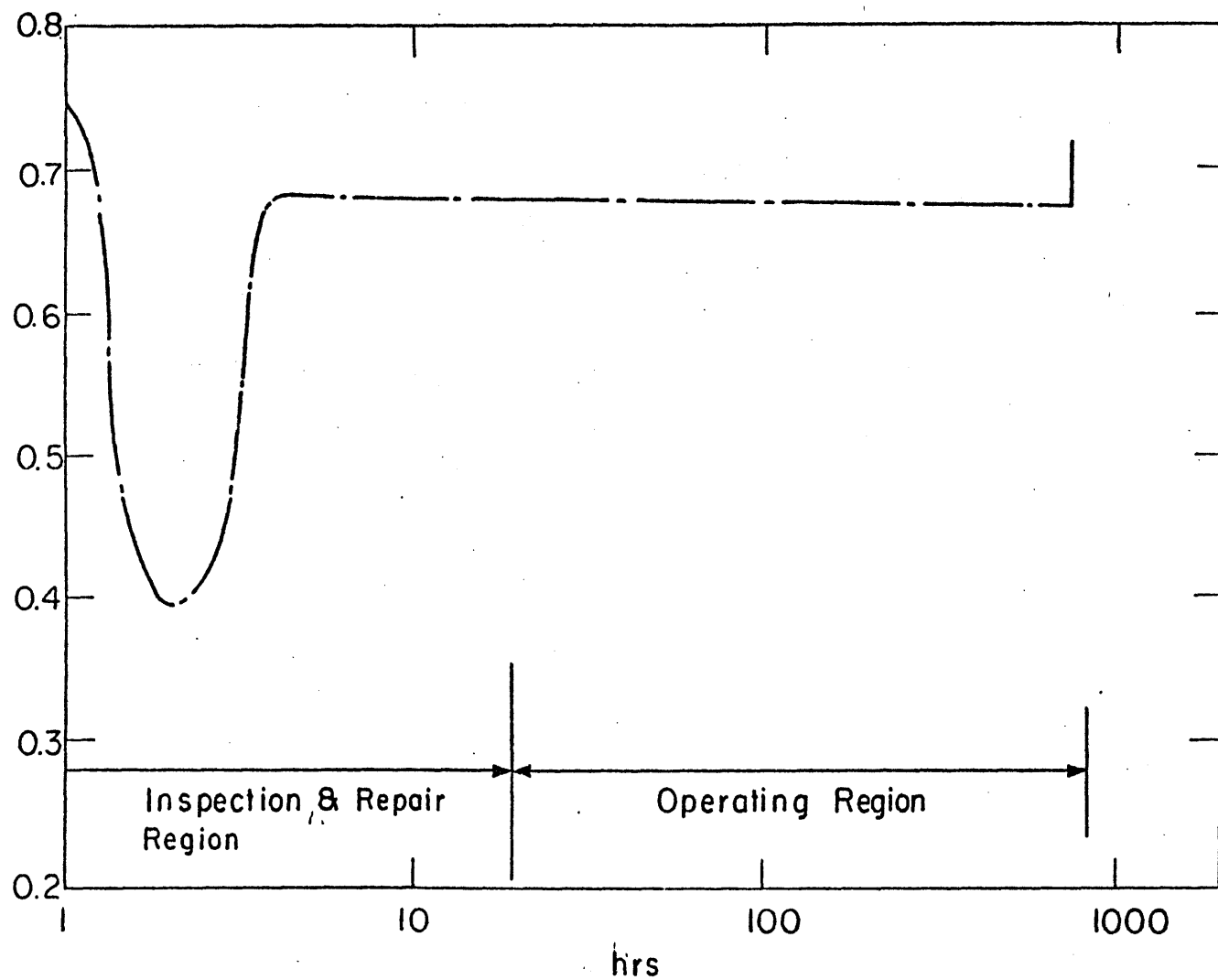


FIGURE 8.2.13 IMPORTANCE OF THE PUMP IN THE AUX-FEED SYSTEM AS FUNCTION OF TIME

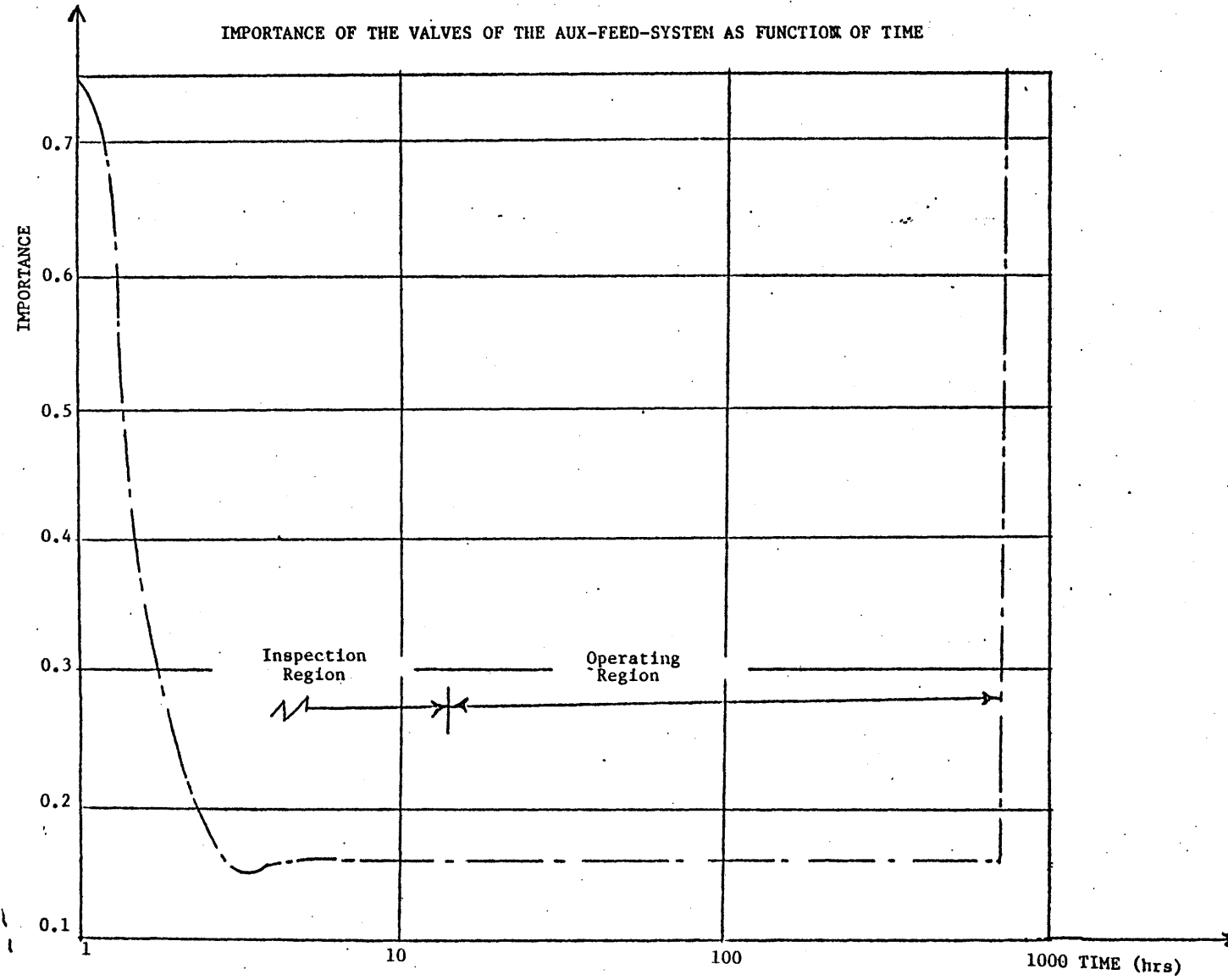
and increases again once the inspection has been finished. Thereafter, it remains fairly constant during repair and operation periods. The opposite behavior can be seen in Figure (8.2.14) for the valves. The reason for this behavior is that the combination of valve 1, pump and valve 2 constitutes a simple module which is directly connected to the top event and thus its total importance is 1 over the whole operational period. The same holds for diesels 4 and 5. As a result, any increase in pump importance is accompanied at the same time by a decrease in valve importance.

8.3 Examples for the Code PL-MODMC

In reference [12] the PL-MODMC code has been discussed and comparisons between the PL-MODMC, LIMITS [13] and SAMPLE [3] codes are presented. In this section the new version of the PL-MODMC in which some of the restrictions of the old version of the code have been removed. And approximation and mathematical formulations discussed in Chapter 4 of this thesis are incorporated, and is testing by presenting some examples. These examples are as follows:

- (1) An arbitrarily designed fault tree consisting of a combination of different classes of components with uncertainty values associated with failure rates, average downtime, average repair time, and unavailability per demand is analyzed.

Figure 8.2.14



- (2) The Low Pressure Recirculation System fault tree discussed in Section 8.2 with 61 free components and 6 replicated components is analyzed by the PL-MODMC Code. All components are of Class 1 and the results are compared with those given in WASH-1400. The values of failure rates and 90% error factors are taken from WASH-1400.
- (3) The new B&W Reactor Protection System fault tree [9] with 170 basic events and 157 gates is evaluated by the PL-MODMC code. The results from PL-MODMC are compared to the median value found by the GO methodology [9]. The components in this fault tree are a combination of Class 1 and 2 components.

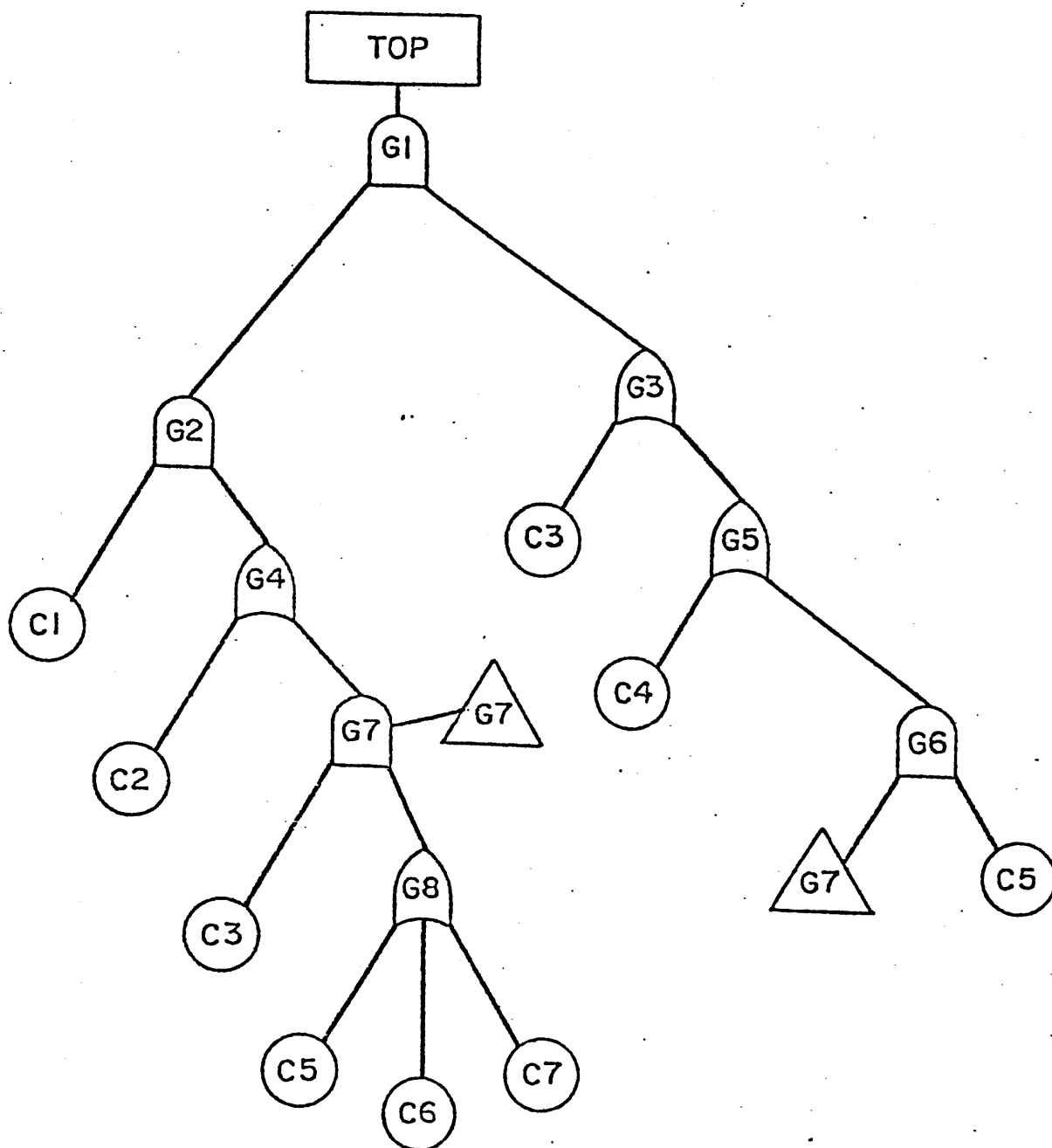
8.3.1 Monte-Carlo Simulation of Failure Parameters of a Fault Tree Consisting of a Combination of the Four Classes of Components

Consider the fault tree in Figure (8.3.1). This fault tree consists of 7 components and 8 gates. Components C3, C5 and Gate G7 are replicated. It should be noted that components C3 and C5 appear in both the domain of the replicated gate and the domain of the top event. Therefore, this fault tree obviously could not be handled by the old version of the PL-MODMC Code. In this fault tree the components are a mixture of different classes of components. The following quantities of interest will be calculated:

- (a) Average top event unavailability
- (b) Unavailability associated with different confidence intervals

Figure 8.3.1)

FAULT TREE FOR A MONTE-CARLO ANALYSIS



For calculating the average top event point unavailability associated with the confidence intervals, Table shows the input component failure data.

Table (8.3.2) presents the output from the PL-MODMC Code for 1200 trials and for the fault tree shown in Figure (8.3.1) and its failure data shown in Table (8.3.1). The output consists of the results:

- (a) Mean unavailability, and Standard deviation of unavailability calculated in each trial.
- (b) Confidence levels and unavailability associated with them.
- (c) 5% and 95% error factors for the top event which are defined as follows:

$$F_{5\%} = \frac{P_{50\%}}{P_{5\%}} \text{ and } F_{95\%} = \frac{P_{45\%}}{P_{50\%}}$$

- (d) Interval probability and Maximum Error associated with the interval probability (see Chapter 4).

For comparing the accuracy of the PL-MODMC code the Code LIMITS has been run for comparison for the same example. However, since LIMITS handles only Class 1 components, by using the mathematical equations discussed in Chapter 4 the value of the equivalent average unavailability and its associated error factor for the Class 3 and 4 components was calculated by using equations (4.3-1) through (4.3-6), prior to the LIMITS run. These values are then used by LIMITS to evaluate the top

Table 8.3.1

Failure Data for the Fault Tree in Figure 8.3.1

Component Name	Class Number	λ or Q (hr ⁻¹)	F_λ	T_D (hr)	F_{T_D}	T_R (hr)	F_{T_R}	τ (hr)	F_τ	T_M (hr)	T (hr)
'C1'	2	2.0×10^{-4}	3	-	-	-	-	-	-	700	-
'C2'	1	1.3×10^{-4}	3	-	-	-	-	-	-	-	-
'C3'	3	1.47×10^{-4}	3	21	7	-	-	-	-	-	-
'C4'	1	2.0×10^{-4}	3	-	-	-	-	-	-	-	-
'C5'	4	2.0×10^{-5}	3	-	-	30	5	2.5	5	-	720
'C6'	2	2.0×10^{-5}	3	-	-	-	-	-	-	700	-
'C7'	4	1.0×10^{-5}	10	-	-	20	3	1.5	3	-	1440

Table 8.3.2

PL-MODMC Output of Fault Tree Presented in Figure 8.3.1

TREE ANALYSIS BY MODULES

DATE: 08/29/79
TIME: 21:41.05.000

RUN TITLE: TEST OF ACCURACY

ESTIMATED MAX TOTAL GATES:	100
ESTIMATED MAX TOTAL COMPONENTS:	200
ESTIMATED MAX GATES INPUT TO A GATE:	5
ESTIMATED MAX COMPONENTS INPUT TO A GATE:	10

FAULT TREE INPUT

GATE NAME	TYPE	INPUTS
TOP	A	G2 G3
G2	A	G4

Table 8.3.2 (continued)

03	0	05 C3		
04	0	07 C2		
05	0	06 C4		
06	A	07 C5		
07	A	08 C3		
08	0	C5	C6	C7

NUMBER OF GATES= 8

NUMBER OF COMPONENTS= 7

REPLICATED GATES: 07

REPLICATED COMPONENTS: NONE

GATE NAME MODULR MINIMAL CUTSETS

CALCULATION OF THE MEDIAN POINT VALUE

NUM FREE EVENT INPUTS= 5

NUM REPLICATED EVENT INPUTS= 2

FREE INPUT	MEDIAN VALUE	SPREAD

C1	6.999999E-02	3

C2	1.300000E-04	3

C4	2.000000E-04	3

C6	6.999999E-03	3

C7	0.441664E-03	7

REP INPUT	MEDIAN VALUE	SPREAD

C3	3.087000E-03	9

C5	1.127222E-02	5

Table (8.3.2) (continued)

TOP EVENT POINT AVERAGE UNAVAILABILITY= 5.789721E-06

THE MONTE-CARLO SIMULATION STARTS NOW

TOTAL NUM OF TRIALS= 1200

MEAN PROB= 2.328730E-05 STANDARD DEVIATION= 6.005271E-05 MEDIAN PROB= 7.127903E-06
 ERROR FACTOR(5%)= 14.13585 ERROR FACTOR(95%)= 11.38871

CONFIDENCE LEVEL	UNAVAILABILITY	INTVL. PROBABILITY	MAX ERROR
0.50000	1.393705E-07	7.859364E-01	1.005607E-01
1.00000	1.919755E-07	9.182771E-01	6.857973E-02
2.50000	3.400596E-07	7.327424E-01	5.512181E-02
5.00000	5.196343E-07	5.732234E-01	4.600069E-02
10.00000	1.052895E-06	4.362971E-01	3.619882E-02
15.00000	1.492676E-06	3.723742E-01	3.112384E-02
20.00000	2.001615E-06	3.349944E-01	2.803759E-02
25.00000	2.530474E-06	3.108435E-01	2.599579E-02
30.00000	3.225246E-06	2.945430E-01	2.458902E-02
40.00000	4.889805E-06	2.763264E-01	2.295169E-02
50.00000	7.127903E-06	2.709655E-01	2.235820E-02
60.00000	1.095650E-05	2.763264E-01	2.295169E-02
70.00000	1.709651E-05	2.945430E-01	2.458902E-02
75.00000	2.089908E-05	3.108435E-01	2.599579E-02
80.00000	2.631526E-05	3.349944E-01	2.803759E-02
85.00000	3.427277E-05	3.723742E-01	3.112384E-02
90.00000	4.816501E-05	4.362971E-01	3.619882E-02
95.00000	8.782063E-05	5.732234E-01	4.600069E-02
97.50000	1.488157E-04	7.327424E-01	5.512181E-02
99.00000	2.371144E-04	9.182771E-01	6.857973E-02
99.50000	4.627793E-04	9.859364E-01	1.005607E-01

event unavailability distribution. The results obtained from both codes are very close and, in particular, PL-MODMC has a slightly longer tail due to the approximations discussed in Chapter 4. Figure (8.3.2) compares the results obtained for the top event unavailability distribution using the two codes PL-MODMC and LIMITS. However, the median value of the top event is almost the same for both codes.

8.3.2 Low Pressure Recirculation System (LPRS)

The LPRS fault tree has been shown in Figure (8.2.2). This fault tree is input into the code PL-MODMC for the data shown in Table II 5-32 of WASH-1400 and 1200 trials were simulated. The results are presented in Table (8.3.3). The input data are listed in WASH-1400 [3]. In general, the results agree well with those calculated by the SAMPLE Code and given in WASH-1400, as can be seen by the comparison presented in Table (8.3.4). The total computation time including the modularization process is 8 seconds.

8.3.3 New B&W Reactor Protection System

An overview of the new reactor protection system is shown in Figure (8.3.3). The nine trip parameters in four independent channels feed a bistable trip string with an output to the reactor trip model which includes the two-out-of-four logic interactions with the other channels. The nine trip parameters are as follows:

1. High Reactor Coolant Pressure (normal operation)
2. High Reactor Coolant Pressure (low setting) for shutdown
bypass mode

Figure (8.3.2) Comparison of the Top Event Distribution for the Fault Tree in Figure (8.3.1) Calculated by the Codes PL-MODMC and LIMITS

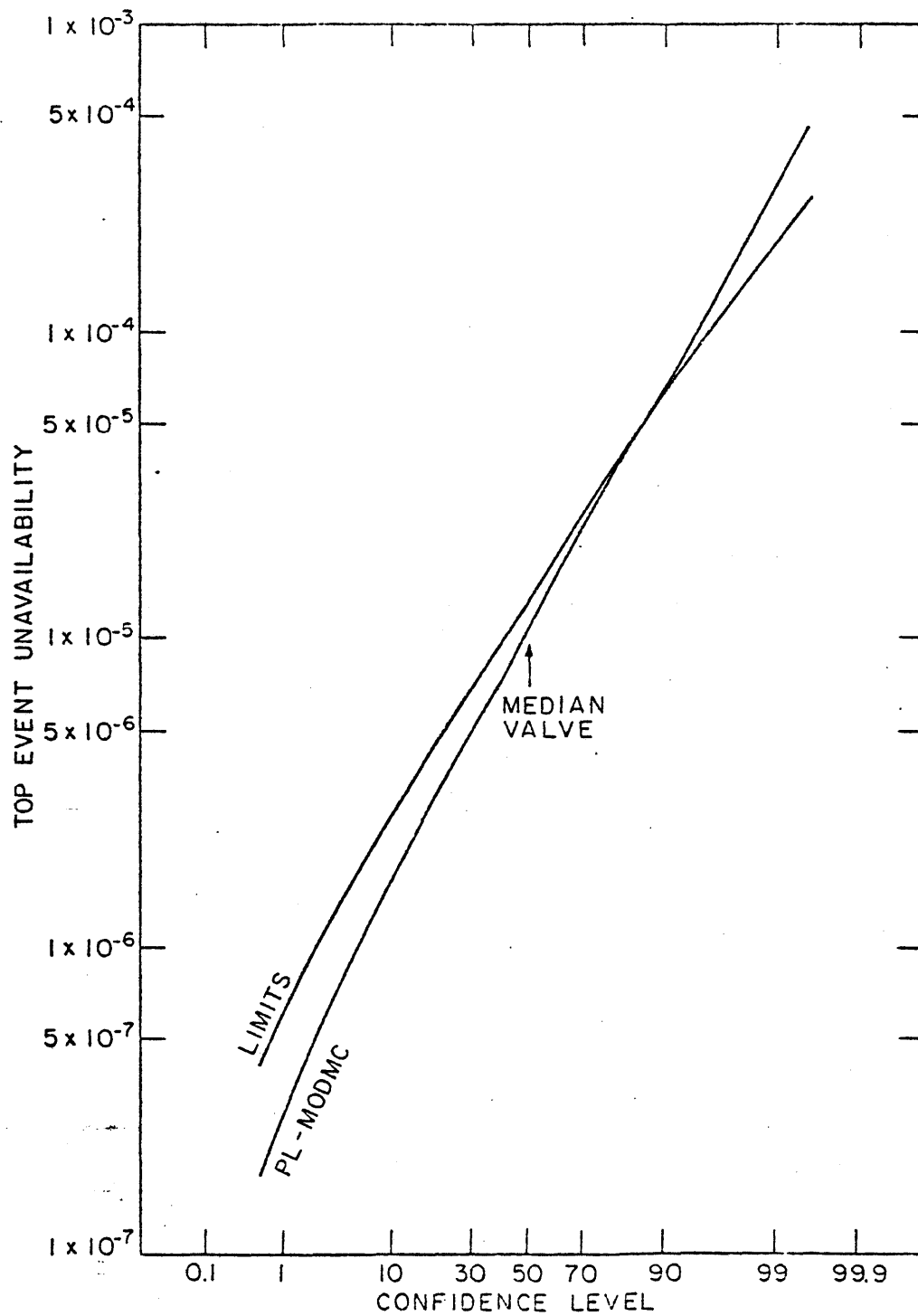


Table 8.3.3 : LPRS Top Event Unavailability for 1200 Trials Using PL-MODMC

MEAN PROB= 1.410070E-02 STANDARD DEVIATION= 7.909754E-03
 ERROR FACTOR (5%)= 2.13365 ERROR FACTOR (95%)= 2.21611 MEDIAN PROB= 1.243022E-02

343	CONFIDENCE LEVEL	PROBABILITY	MAX ERROR	MIN PROBABILITY
	0.50000	3.575392E-03	1.893921E-01	7.656468E-01
	1.00000	4.353750E-03	1.224911E-01	7.222902E-01
	2.50000	5.138319E-03	8.370483E-02	5.512648E-01
	5.00000	5.825788E-03	6.371158E-02	4.198762E-01
	10.00000	6.662294E-03	4.763011E-02	3.150179E-01
	15.00000	7.352761E-03	4.019815E-02	2.677403E-01
	20.00000	8.052737E-03	3.585986E-02	2.404665E-01
	30.00000	9.505350E-03	3.113346E-02	2.112458E-01
	40.00000	1.096696E-02	2.891828E-02	1.982517E-01
	50.00000	1.243022E-02	2.810337E-02	1.945992E-01
	60.00000	1.385169E-02	2.891828E-02	1.982517E-01
	70.00000	1.591348E-02	3.113346E-02	2.112458E-01
	80.00000	1.868913E-02	3.585986E-02	2.404665E-01
	85.00000	2.036301E-02	4.019815E-02	2.677403E-01
	90.00000	2.358368E-02	4.763011E-02	3.150179E-01
	95.00000	2.754679E-02	6.371158E-02	4.198762E-01
	97.50000	3.381504E-02	8.370483E-02	5.512648E-01
	99.00000	3.927734E-02	1.224911E-01	7.222902E-01
	99.50000	4.923805E-02	1.893921E-01	7.656468E-01

Table (8.3.4)

COMPARISON OF THE LPRS FAULT TREE SIMULATION
USING PL-MODMC AND THE RESULTS CALCULATED BY
THE SAMPLE CODE IN WASH-1400

<u>Confidence Level</u>	<u>WASH-1400 (SAMPLE)</u>	<u>PL-MODMC</u>
50%	1.3×10^{-2}	1.24×10^{-2}
5%	4.4×10^{-3}	5.82×10^{-3}
95%	3.1×10^{-2}	2.75×10^{-2}

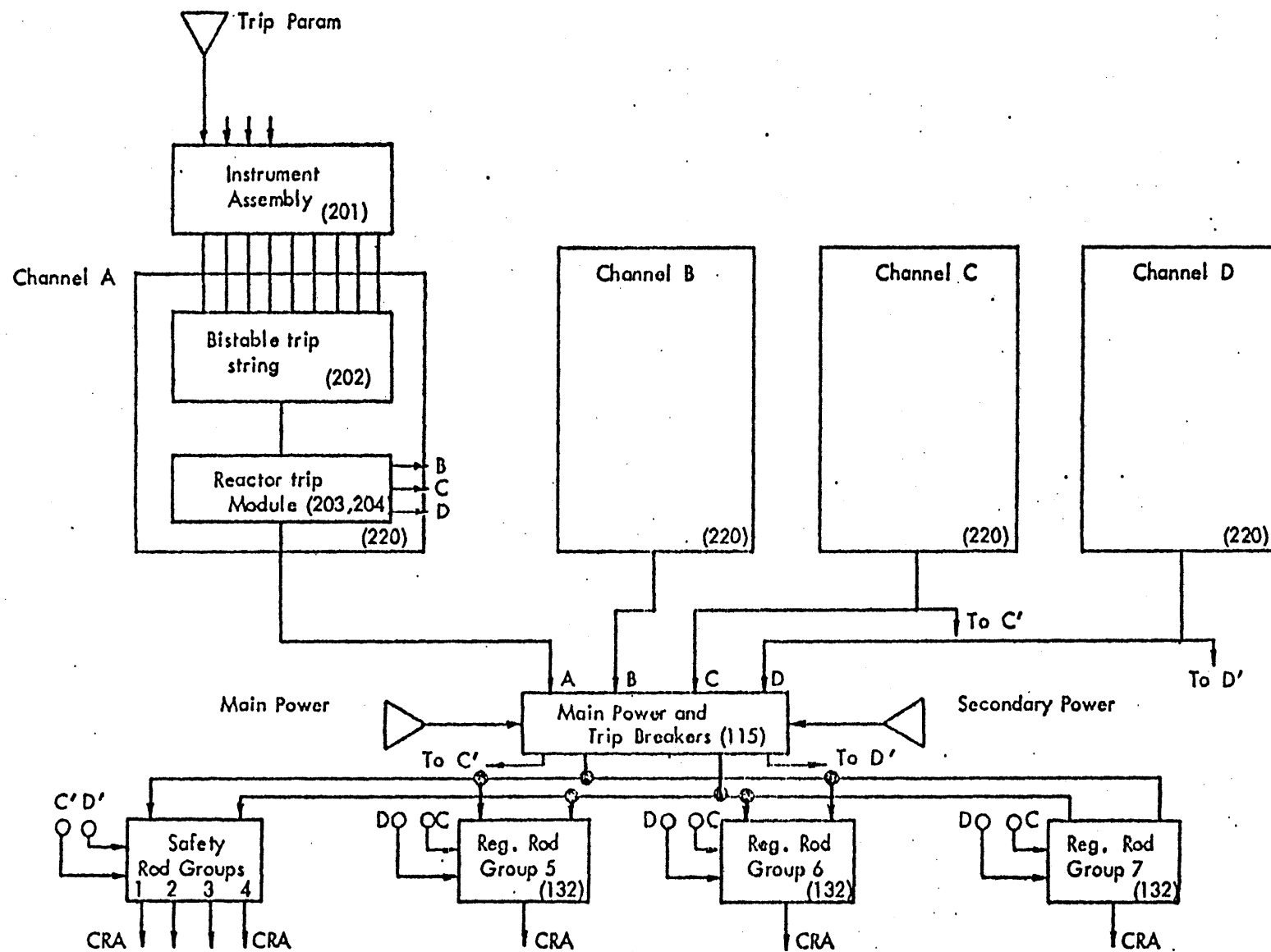


FIGURE 8.3.3
REACTOR TRIP SYSTEM

3. Low Reactor Coolant Pressure.
4. High Reactor Coolant Pressure.
5. Variable Reactor Coolant Low Pressure (Temperature Comparison)
6. Overpower
7. Power/Reactor Coolant Pumps Comparison (Pump Monitor)
8. Power/Imbalance/Flow (Unbalanced Power)
9. Reactor Containment Pressure High

There are actually four primary physical parameters (coolant pressure, coolant temperature, reactor power and building pressure) combined with other auxiliary parameters (number of pumps, power imbalance, flow) to provide those nine trip parameters.

Trip signals interrupt main and secondary power to the power supplies for four safety rod groups and three regulating rod groups. Removal of holding power from the windings of the Control Rod Assemblies (CRA) allows the rod to drop for a safe shutdown.

The channel trip relays are connected in 2-out-of-4 trip logic as shown in Figure (8.3.4). Each relay de-energizes four auxiliary trip relays (KA1, KA2, etc.) one in each channel. Power is provided to the undervoltage windings of circuit breakers through the dual 2-out-of-4 matrix which provides power interrupt on any combination of two channels tripped.

The logic for the safety groups is shown in Figure (8.3.5). Each control rod drive assembly has six windings driven by six-phase power from the three-phase buses through a delta/star transformer and appropriate power supplies. Safety rods are held out by applying power to two windings from the independent main and secondary buses. Trip from the

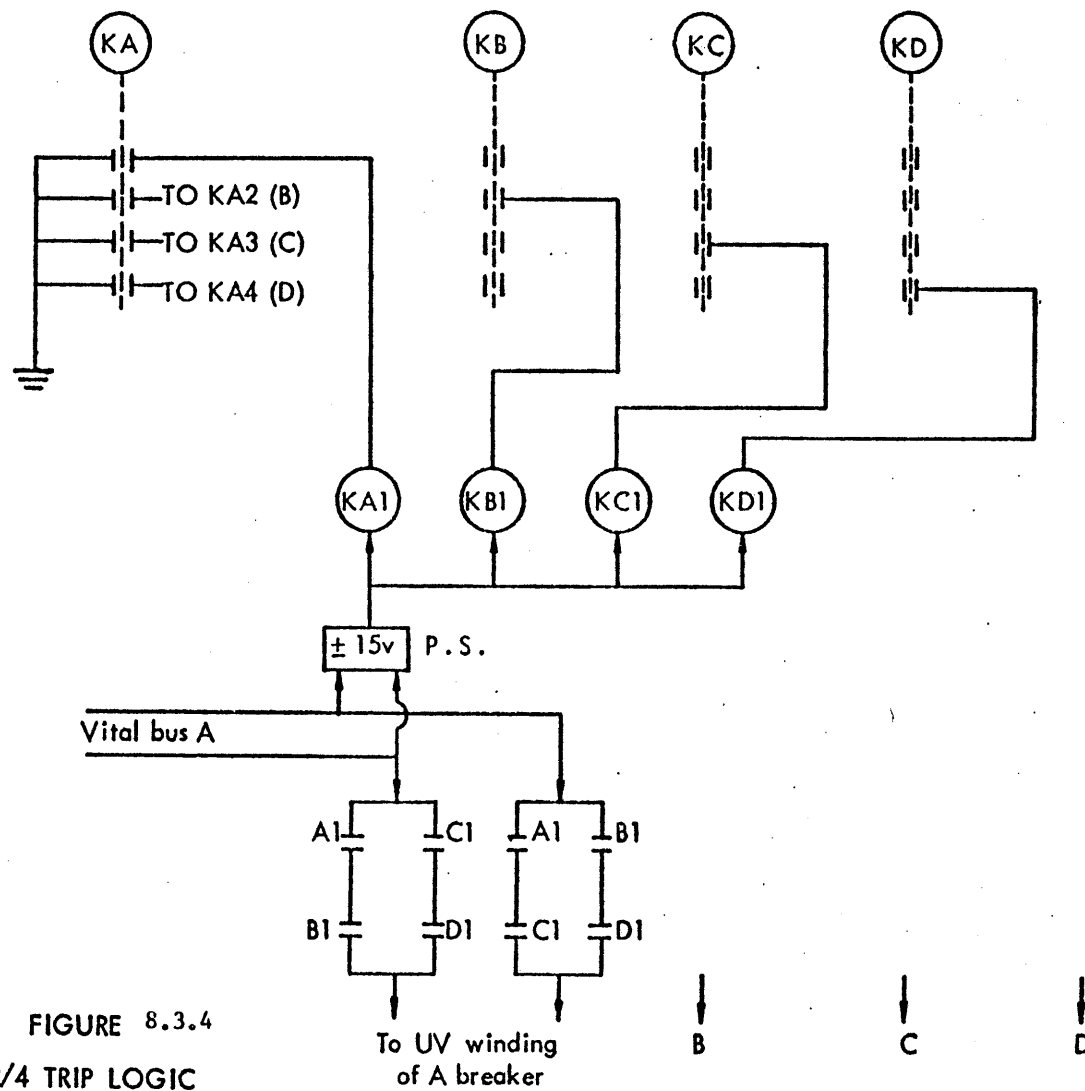


FIGURE 8.3.4
2/4 TRIP LOGIC

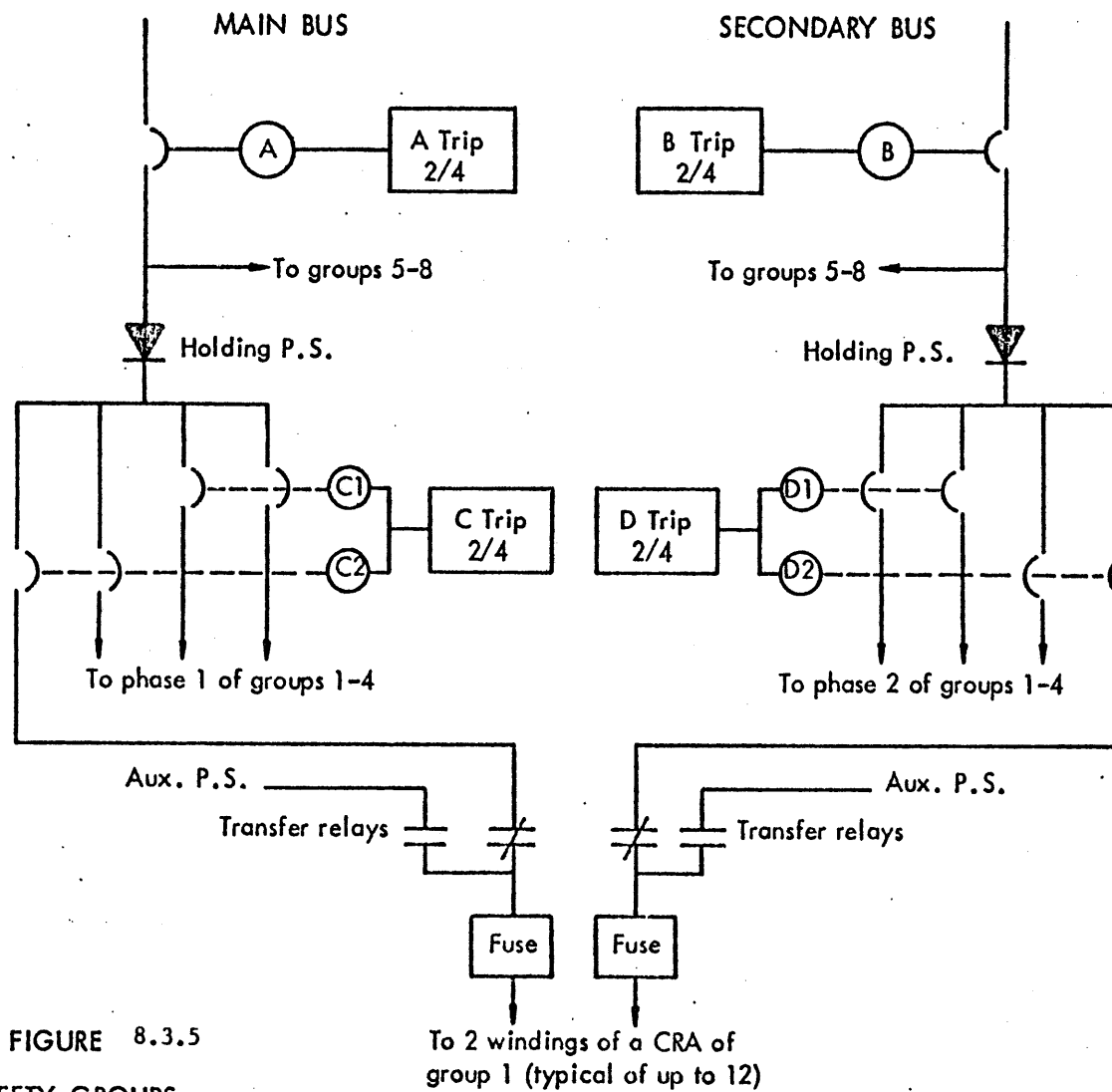


FIGURE 8.3.5
SAFETY GROUPS

A and B channels removes power from the undervoltage windings of the A and B breakers to interrupt both ac buses. (The A and B breakers also interrupt power to the regulatory rod groups, discussed in the next paragraph). Dual dc breakers (C_1 , C_2 , D_1 , D_2) connected to the C and D trip channels provided a second trip mechanism by interrupting dc power to both CPA windings. Transfer relays are provided to allow rod withdrawal at startup with six-phase power from the auxiliary power supply to one group at a time.

Regulatory rod group logic is shown in Figure (8.3.6). Power is available to all six windings, although only two or three are energized at any one time, depending on the rotational position of the rod drive. Thus power to all six windings is interrupted to trip. Three groups of regulating rods are inserted on reactor trip.

The power supplies include six SCR's per group supplied by the main bus through the A breaker and six parallel SCR's from the secondary bus and B breaker. dc control power to the SCR's is supplied through lamps and optical encoders (six per group) controlled by programmers (one per group) to the gate drives (six per group). Programmers, lamps and encoders are not shown in the figure. Independent trip is provided from the C and D channels through E and F relays to interrupt dc control power to the SCR's.

The fault tree for this system consists of 170 basic events and 157 gates. This fault tree is shown in Figure (8.3.7). The fault tree has been previously evaluated for a point estimate of the system unavailability using the GO methodology and a fault tree analysis code [9]. In this study this fault tree is also evaluated by using the PL-MODMC Code. The top event point unavailability calculation and a Monte-Carlo simulation

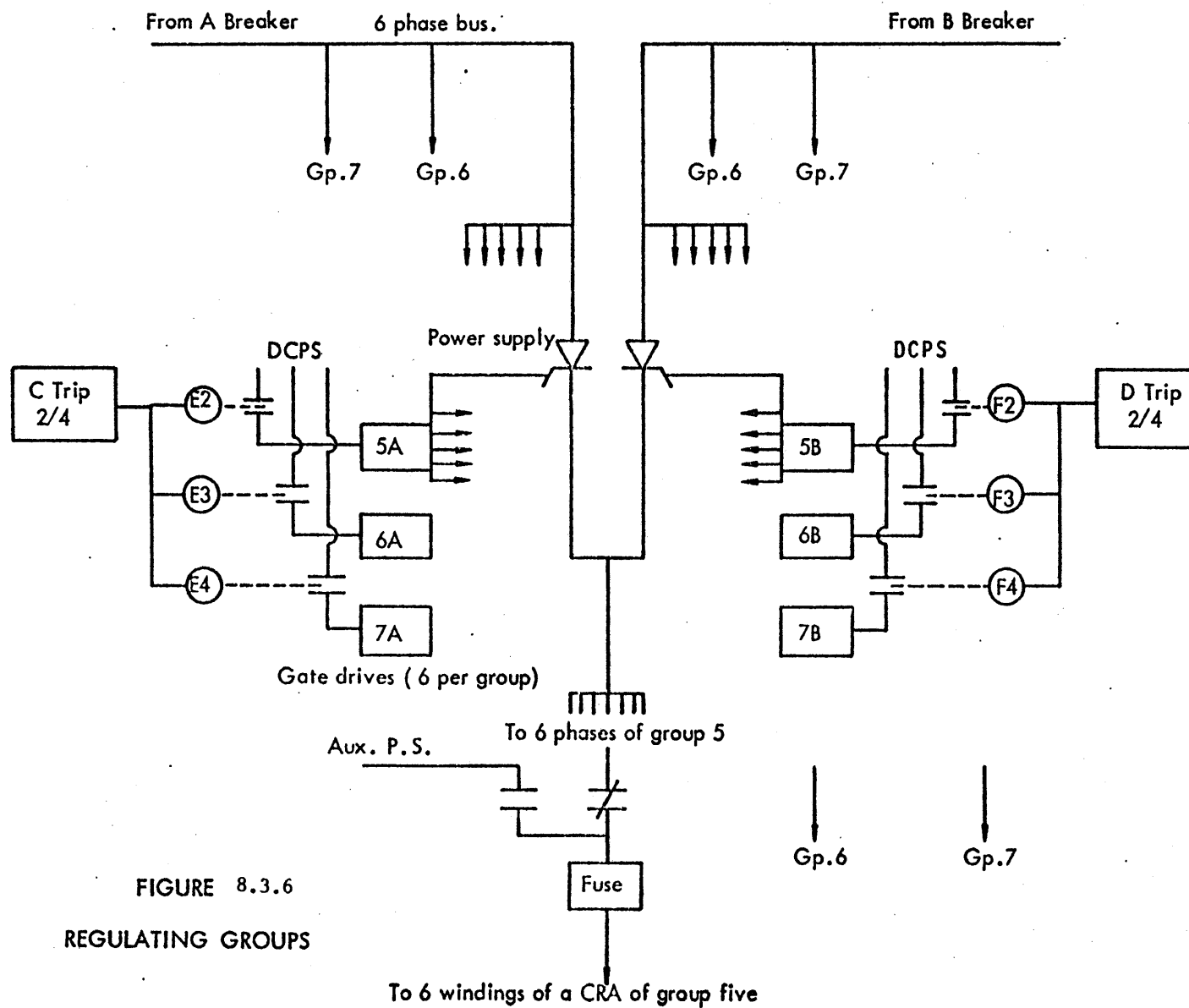
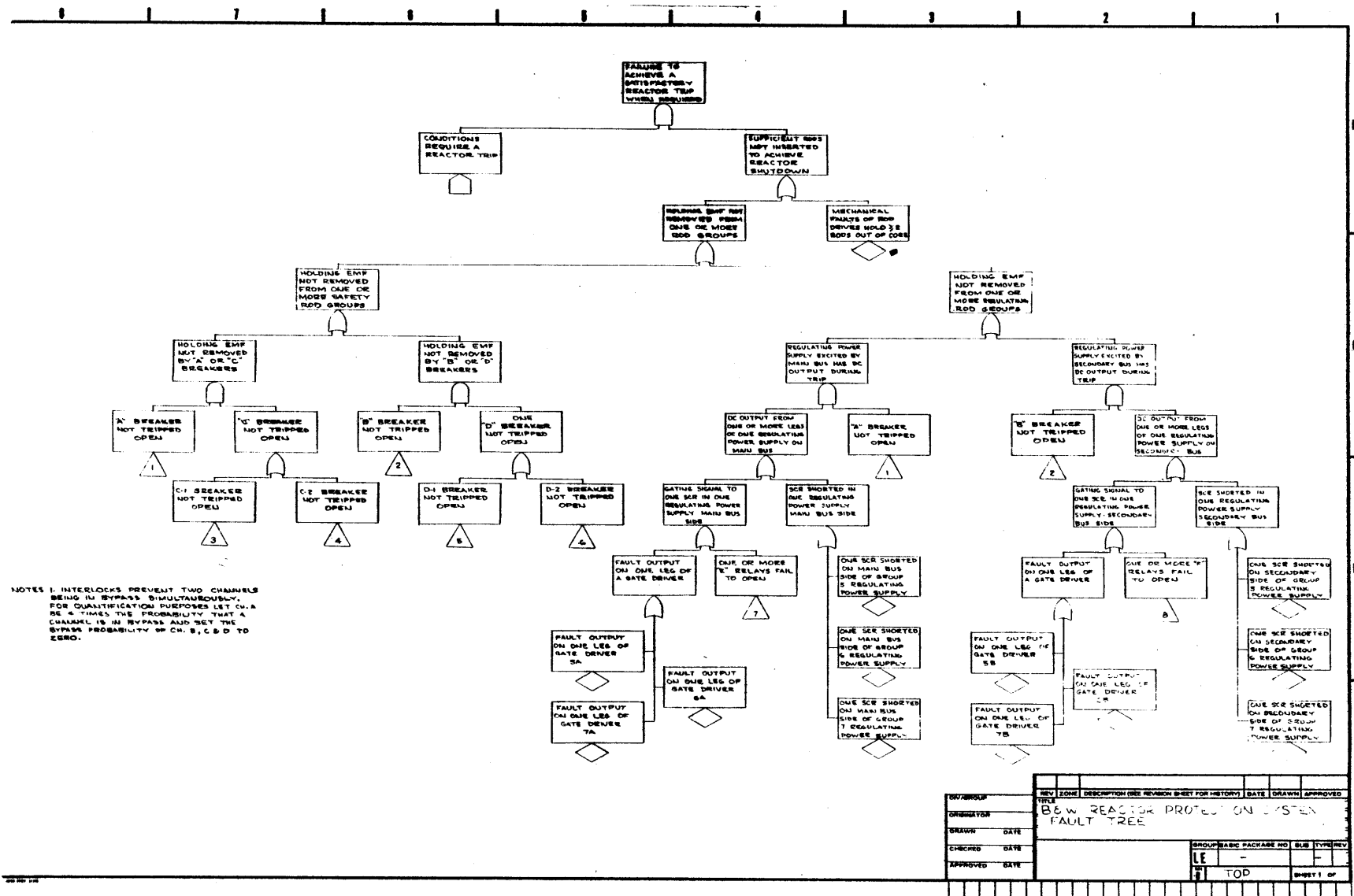


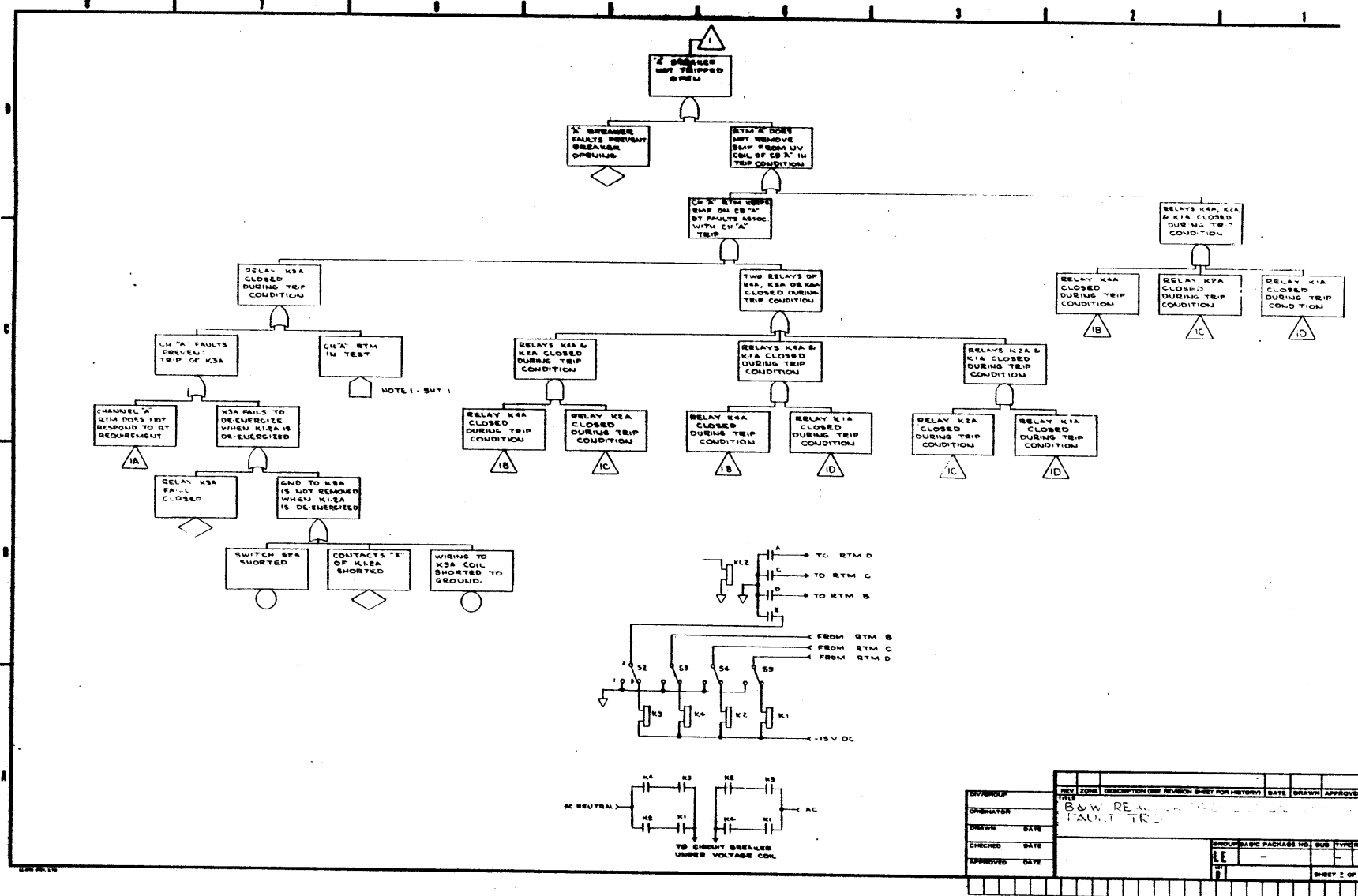
FIGURE 8.3.6
REGULATING GROUPS

Figure (8.3.7) B&W Reactor Protection System Fault Tree



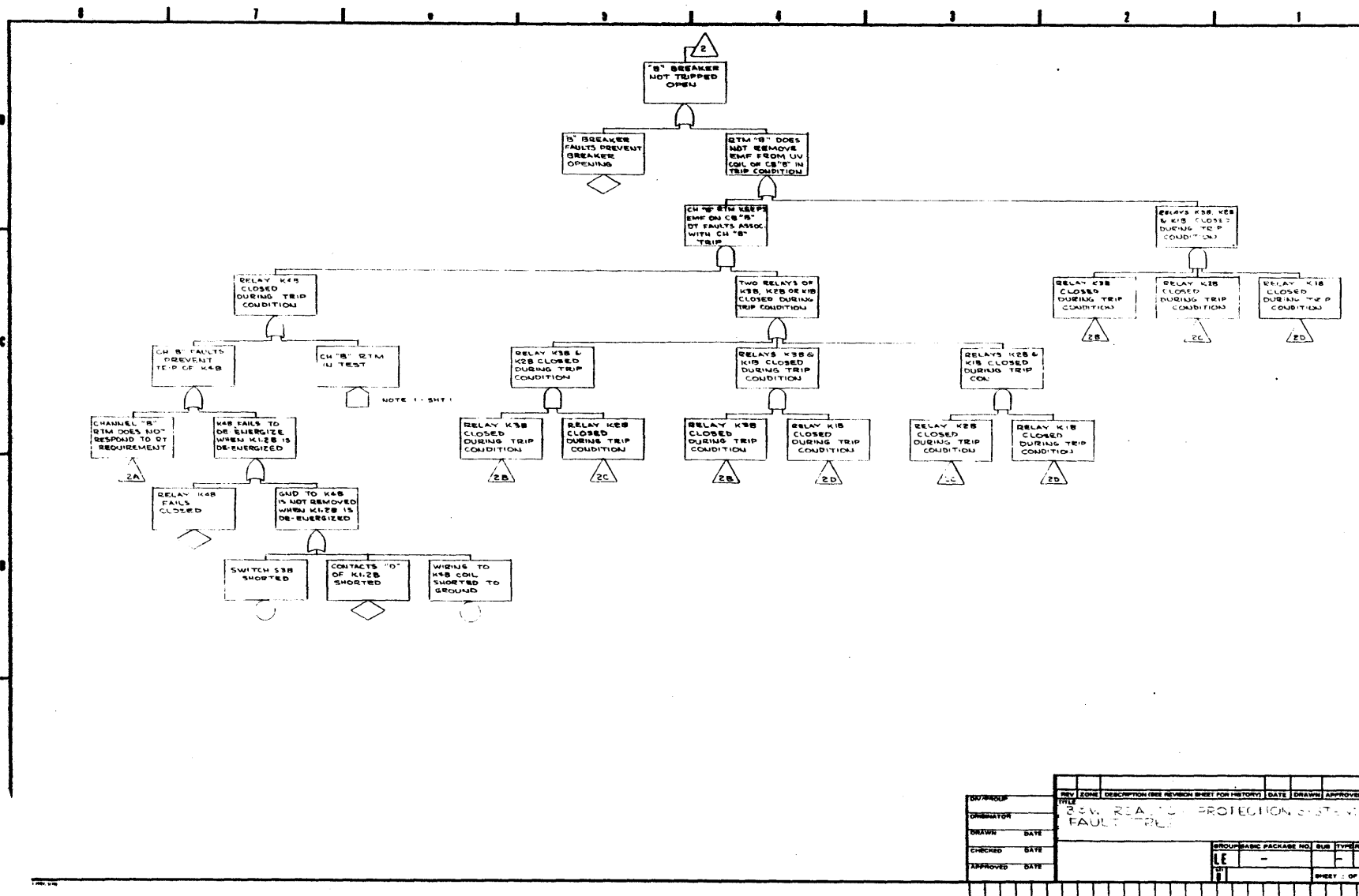
(continued)

Figure : (8.3.7) (continued)



(continued)

Figure (8.3.7) (continued)



(continued)

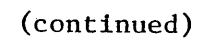
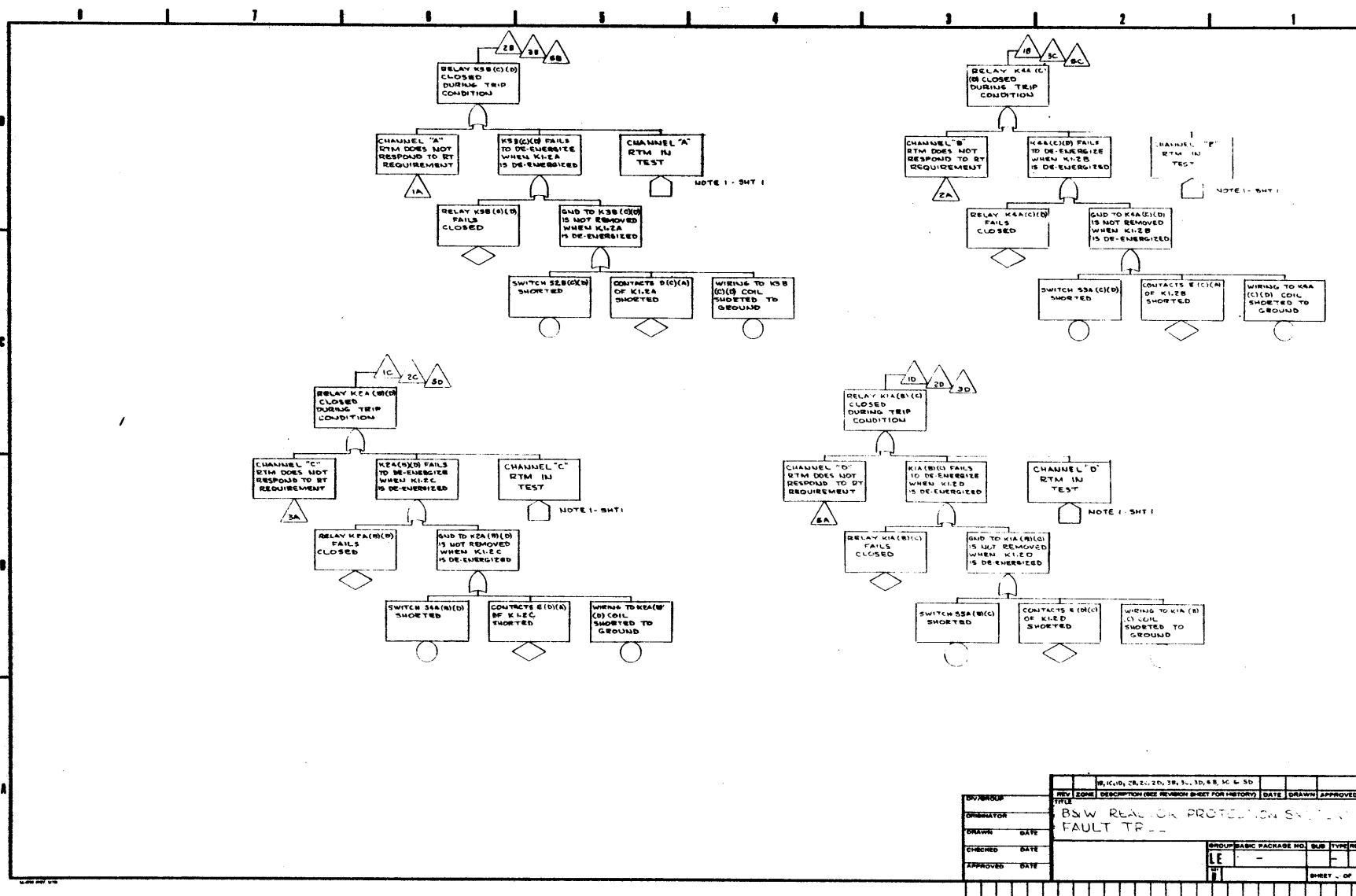
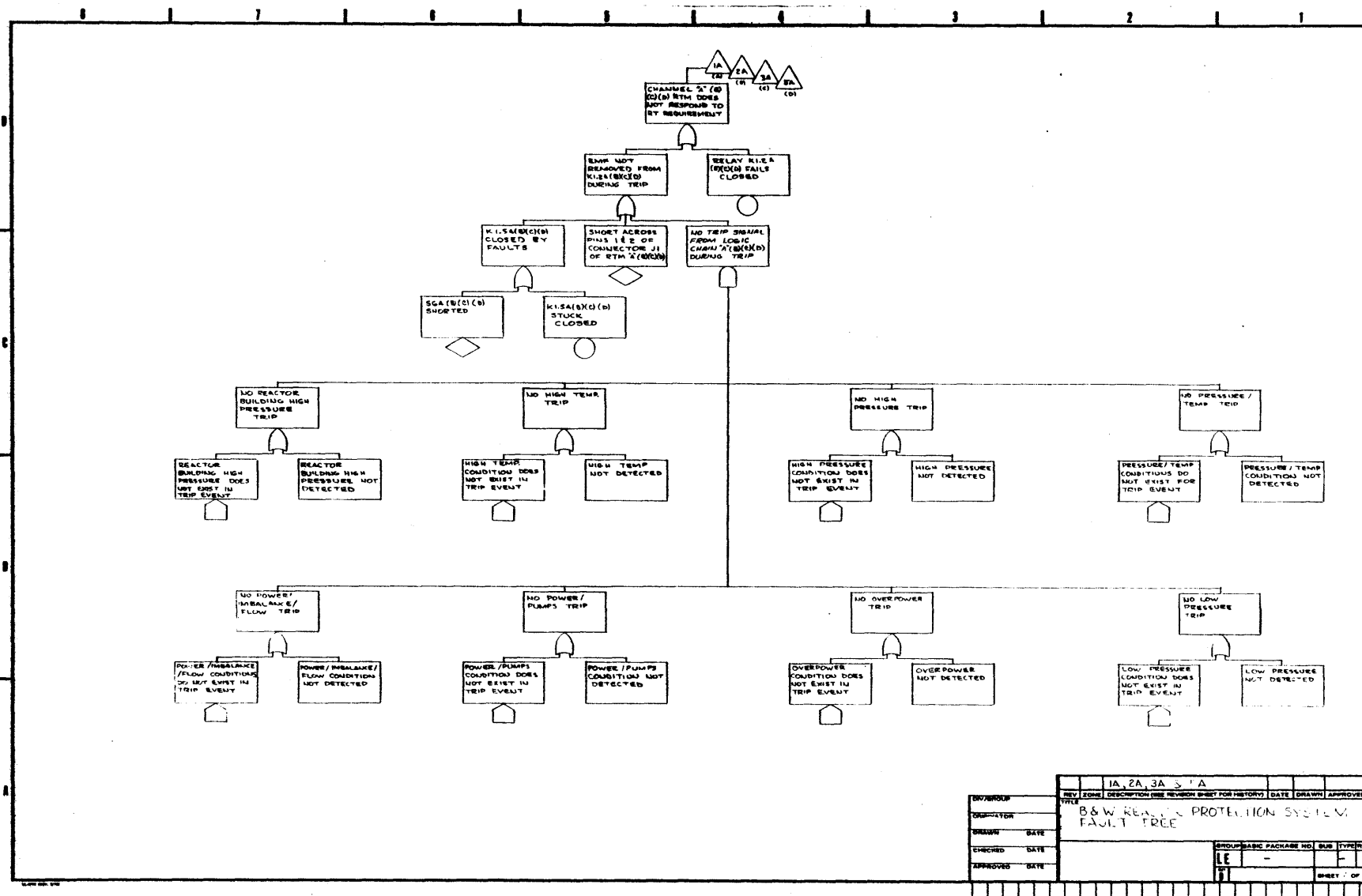


Figure (8.3.7) (continued)



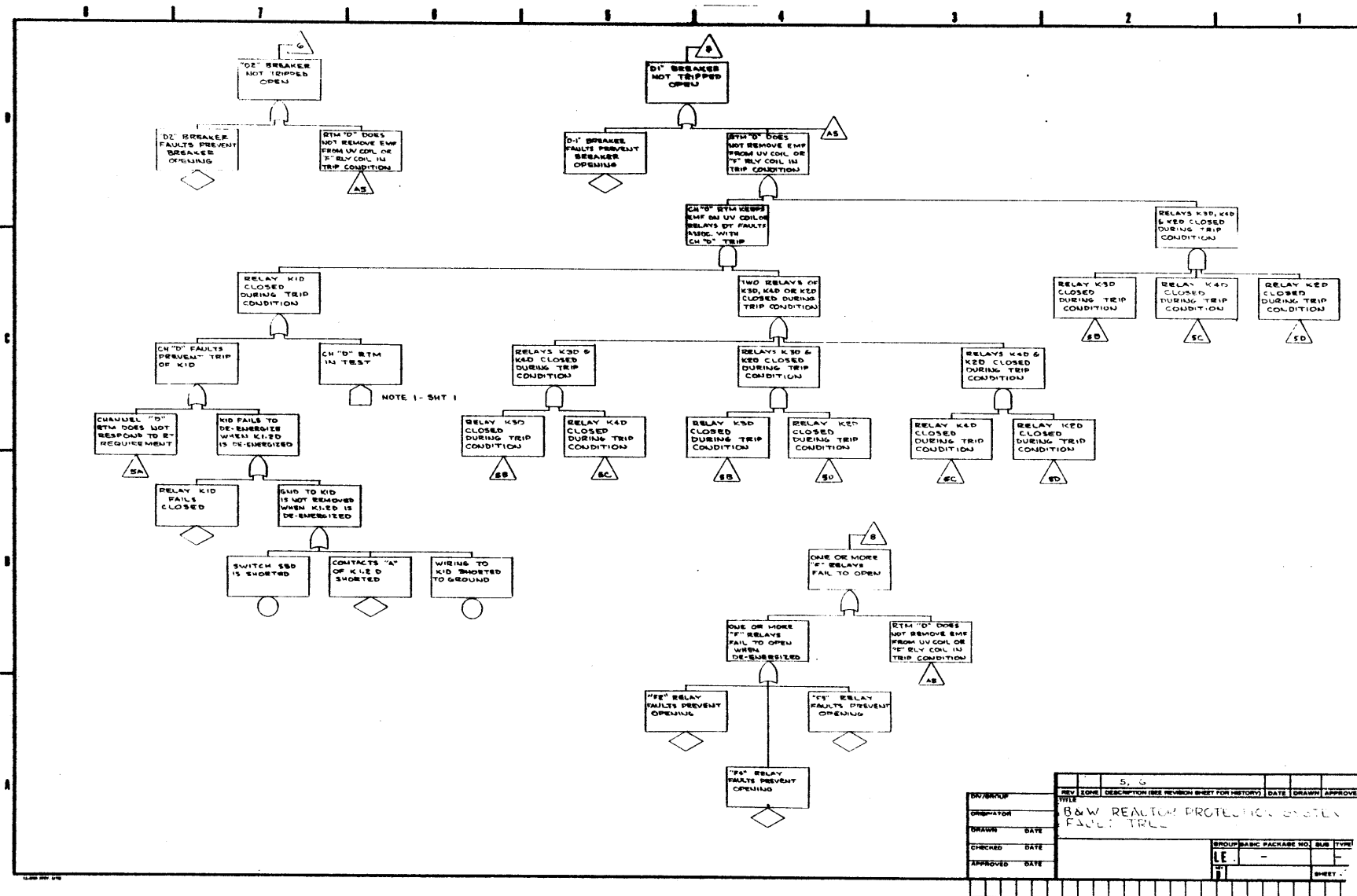
(continued)

Figure (8.3.7) (continued)



(continued)

Figure (8.3.7) (continued)



are performed by using the PL-MODMC code.

Due to the fact that in the code PL-MODMC, all of the fault tree information is stored in the main memory of the computer, and because the fault tree for the Reactor protection system is very large (the fault tree consists of 170 basic events and 157 gates), extra core storage was required in the first attempt to evaluate the entire fault tree by the PL-MODMC Code. Therefore, it has been decided to evaluate the fault tree in two steps. First, top event point unavailability and unavailability distribution is found for all of the independent replicated gates (i.e., replicated gates having no component in their domain that appears elsewhere in the tree, for example, transfer gates 1A, 2A, 3A, 5A, ..., etc.). Second, the median and error factors which are calculated for the independent replicated gates (i.e. in the first step) are input to the PL-MODMC. However, in this step, each independent replicated gate is removed by an equivalent replicated component. Unavailability distribution for the independent replicated gates is therefore, approximated to be log-normal.

Failure data for this fault tree are given in Reference [9], and they are primarily taken from WASH-1400. Some of the data are also taken from MIL217B [10] or GIDEP[11]. Failure rates for this fault tree are shown in Table (8.3.5). The fault tree consists of the Class 1 and 2 components. For the Monte-Carlo analysis all unavailabilities and failure rates with uncertainty are assigned a 90% error factor of 10, which is comparable to the values used in WASH-1400. However, in some cases the error factors are expected to be lower than 10 and, therefore, the Monte-Carlo calculation is expected to give conservative results.

Table 8.3.5
GO COMPONENTS AND FAILURE DATA

359

Type	Kind	Description	Failure Mode	Failure Rate Per 10 ⁶ hr.	λ , hours	Demand Failure	Unavailability (DT or demand)
1	103	Temp. Element	low	1	4000		4×10^{-3}
1	104	Press Xmtz	low	1	4000		4×10^{-3}
1	105	Press Xmtz Pwr Supply	low	3	360		1×10^{-3}
1	106	Press Switch	open			1×10^{-4}	1×10^{-4}
1	107	Transformer & Relay	open	1	360		4×10^{-4}
1	108	Ion Chambers	low	1	360		4×10^{-4}
1	109	Aux. Pwr. Supply (Pump mon.)	low	3	360		1×10^{-3}
1	110	Detect. Pwr. Supply	low	3	360		1×10^{-3}
3	501	RB Press Relay Module	open	0.3	360		1×10^{-4}
3	502	Aux. Trip Relay Coil	short to gnd open	0.3 0.3	360 360		1×10^{-4} 1×10^{-4}
3	503	BKR Actuators; C&D	fail to oper. premature	1	360	1×10^{-3}	1×10^{-3} 4×10^{-4}
3	504	BKR Actuators; A&B	fail to oper. premature	1	360	1×10^{-3}	1×10^{-3} 4×10^{-4}
3	505	Holding PS Rectifiers	open	3	360		1×10^{-3}
3	506	Relay Coils E&F	open	0.3	360		1×10^{-4}
3	507	Prog. Gate Driver Circuits	output short open	1.4 3	360 360		5×10^{-4} 1×10^{-3}
3	508	Bypass Relay	open	0.3	360		1×10^{-4}
3	611	Chan. Trip Relay Coil	open	0.3	360		1×10^{-4}
3	615	Inductrol & Delta Star Transf.	open	2	360		7×10^{-4}
3	616	Prog. Lamps	open	3	360		1×10^{-3}
3	617	Prog. Motor	fail to oper.			3×10^{-4}	3×10^{-4}
3	618	Transfer Relay	open	0.3	360		1×10^{-4}
3	619	CRD Stator Winding	open	0.3	360		1×10^{-4}
6	201	Perfect Switch (logic)					0
6	202	Test Unit					
6	203	Bridge & Signal Conv					
6	204	B/S Trip Unit					
6	207	Buffer Amp	low	1	360		4×10^{-4}
6	208	Linear Amp	high	0.3	360		1×10^{-4}
6	209	Sum Amp					
6	210	Diff Amp					
6	211	Punc Gen					
6	212	Pump Control Monitor Amp					
7	205	Perfect Switch (logic)					0
9	402	Test Inhibit (logic)					0
16	301	Aux Trip Relay Contacts	short	0.01	360		4×10^{-6}
16	302	Manual Trip Switch Contacts	fail to open			1×10^{-5}	1×10^{-5}
16	303	Interlock, Bistable Trip, & KA Relay Contacts	short	0.01	360		4×10^{-6}
16	304	Bypass Relay Contacts	short	0.01	30		3×10^{-7}
16	305	Reset Switch Contacts	fail to open			1×10^{-5}	1×10^{-5}
16	306	Breaker Contacts (A&B)	(perfect - all failures included in actuator)				
16	307	Breaker Contacts (C&D)	(" " " " " ")				
16	308	Relay Contacts (E&F)	short	0.01	360		4×10^{-6}
16	311	Silicon Controlled Rectifiers (SCR)	open	2	360		7×10^{-4}
			short	0.6	360		2×10^{-4}
16	312	Photo-optical programmer	open	1	360		4×10^{-4}
16	313	Test Switch Contacts	short to gnd.	0.01	360		4×10^{-6}
16	315	Logical Switch (perfect)					
5	11	Aux Relay Coil	Short to gnd.	0.01	360		4×10^{-6}
5	10	Connector Pins	short	0.3	360		1×10^{-5}
5	13	Bypass Switch	short to gnd.	0.01	30		3×10^{-7}
5	40	Maintenance (6 hrs out of 720)					8×10^{-3}

The Monte-Carlo calculation is performed by using 1200 trials, and a Central Limit Theorem Approximation of $n = 12$. The results obtained from the PL-MODMC code are shown in Table (8.3.6).

The average unavailability calculated by the PL-MODMC is 2.99×10^{-5} for the top event compared to the results of both the Go code and the Fault Tree analysis code which are about 3×10^{-5} [9]. It should be noticed, however, that the result given in Reference [9] is a single point value for the top event unavailability, whereas, in PL-MODMC both a single point unavailability and a Monte-Carlo simulation are calculated. A total of 10.95 CPU seconds was spent by the PL-MODMC code for finding an empirical distribution of the top event for the whole fault tree. In addition to the Monte-Carlo simulation, a point average unavailability is calculated for the top event (i.e., no Monte-Carlo simulation) and a total of 2.74 seconds was used by the code to calculate this single value for the top event average unavailability. The calculation of this value is also performed in two steps by setting the variable MONT = 1 in the input to PL-MODMC and by removing the error factors for the input parameters (see Appendix C).

The Go code running time was about 30 seconds on a CDC-6400 computer. It should be noticed that the PL-MODMC calculations are performed by using an IBM 370/168 computer. However, the computation capacity of this computer is smaller than that of CDC-6400 by a factor of roughly 3 to 6 depending on the computer program being analyzed. The Go model contained 538 operators without the instrumentation channels. Running time for this model was about 50 seconds. However, the time was reduced to about 30 seconds when components which do not appear in the fault tree were made perfect (see Appendix A for description of the Go methodology).

Table (8.3.6) PL-MODMC Output for the Fault Tree in Figure (8.3.7) and
Date in Table (8.3.5)

MEAN PROB=	8.154909E-05	STANDARD DEVIATION=	2.267788E-04	MEDIAN PROB=	2.990065E-05
ERROR FACTOR(5Z)=	7.24407	ERROR FACTOR(95Z)=	8.98032		
CONFIDENCE LEVEL	UNAVAILABILITY	INTVL PROBABILITY	MAX ERROR		
0.50000	1.408673E-06	9.999995E-01	1.855380E-02		
1.00000	1.968094E-06	9.996197E-01	9.110769E-03		
2.50000	2.897806E-06	9.764599E-01	6.371949E-03		
5.00000	4.266671E-06	8.952425E-01	8.733936E-03		
10.00000	6.442290E-06	7.614073E-01	1.027986E-02		
15.00000	8.576671E-06	6.778980E-01	1.026612E-02		
20.00000	1.090645E-05	6.232409E-01	9.981245E-03		
25.00000	1.342608E-05	5.857837E-01	9.682640E-03		
30.00000	1.598327E-05	5.595992E-01	9.428132E-03		
40.00000	2.198067E-05	5.295134E-01	9.084430E-03		
50.00000	2.990065E-05	5.204999E-01	8.947823E-03		
60.00000	4.148361E-05	5.295134E-01	9.084430E-03		
70.00000	5.974923E-05	5.595992E-01	9.428132E-03		
75.00000	7.188643E-05	5.857837E-01	9.682640E-03		
80.00000	9.058492E-05	6.232409E-01	9.981245E-03		
85.00000	1.158023E-04	6.778980E-01	1.026612E-02		
90.00000	1.661868E-04	7.614073E-01	1.027986E-02		
95.00000	2.908874E-04	8.952425E-01	8.733936E-03		
97.50000	4.756341E-04	9.764599E-01	6.371949E-03		
99.00000	9.489986E-04	9.996197E-01	9.118769E-03		
99.50000	1.241333E-03	9.999995E-01	1.855380E-02		

THE END

The Monte-Carlo calculation of the Go fault tree in this section proved the accuracy of the results obtained by using the PL-MODMC code. The efficiency and effectiveness of the PL-MODMC code is also demonstrated in this example. However, for a more effective evaluation of a large fault tree by the PL-MODMC code incorporation of an out-of-core storage facility is inevitable.

8.4 Examples for the Code MODCUT

Four examples are discussed in this section as follows:

- (1) The fault tree shown in Figure (5.3.1), which has been used in Chapter 5 to explain the procedures employed by the MODCUT Code to generate the cutsets is solved by the MODCUT code. Two runs have been made, the first to evaluate the number of cutsets in each gate, and the second to evaluate the cutsets themselves. Also, by setting the DEBUG variable equal to one in the second run, all modules and minimal cutsets are printed out.
- (2) Minimal cutsets of an arbitrary tree consisting of simple, replicated and ON-OFF components, and K-out-of-N gates are determined in this example. This fault tree had also been used in reference [1].
- (3) The cutsets of a fault tree to identify potential sequences of events by which solidified high-level waste could be released from containment during storage at the Radioactive Surface Facility are generated by the MODCUT.

- (4) The fault tree for the new B&W Reactor Protection System which is discussed in Section 8.3 of this chapter is also analyzed by the MODCUT code to determine its low order cutsets, (i.e., cutsets of size 1, 2 and 3).

8.4.1 Generating Cutsets of the Fault Tree Shown in Figure (5.3.1)

By Using the MODCUT Code

The fault tree shown in Figure (5.3.1) is an arbitrarily designed fault tree used in Chapter 5 of this thesis to explain the procedures used to generate minimal cutsets by the MODCUT Code. Two computer runs have been performed by using the MODCUT code. First, to calculate the number of minimal cutsets by setting the variable CUTNO equal to one and second to generate the cutsets of the top event with the DEBUG variable equal to one in order to print in addition to the minimal cutsets, modular information and modular minimal cutsets. This would further verify the steps discussed in Chapter 5 to generate minimal cutsets.

The input to MODCUT for the first run is listed in Table (8.4.1) and the output is shown in Table (8.4.2). The total number of non-minimal cutsets for the top event is 20. The input for the second run is exactly similar to the first run with the exception that the CUTNO variable is set to zero and the DEBUG variable is set equal to one (see Appendix D for the MODCUT users manual). The output from MODCUT is shown in Table (8.4.3). The running time for this example was about 1.35 seconds for the step which generated minimal cutsets.

Table (8.4.1) Input to MODCUT with
the CUTNO Variable Equal to One

```

      EXAMPLE
0 0 0 0 1 0
'TOP' 'AND' 3 'G1' 'G2' 'G3' 0
'G1' '2/3' 1 'G4' 2 'C1' 'C2'
'G2' 'AND' 1 'G5' 1 'C5'
'G3' 'AND' 1 'G6' 1 'C8'
'G4' 'OR' 1 'G7' 1 'C6'
'G5' 'OR' 1 'G8' 1 'C2'
'G6' 'OR' 1 'G8' 2 'C7' 'C3'
'G7' 'AND' 0 2 'C3' 'C4'
'G8' 'AND' 1 'G9' 2 'C5' 'C9'
'G9' 'OR' 0 2 'C10' 'C11'
'END'
8 3
2 'G8' 'TOP'

```

Table (8.4.2) Number of Cutsets in Each Module Calculated
by MODCUT with CUTNO = 1

GATE NAME: G7		
NO. OF NONMINIMAL CUTSETS=	1	
GATE NAME: G9		
NO. OF NONMINIMAL CUTSETS=	2	
GATE NAME: G4		
NO. OF NONMINIMAL CUTSETS=	2	
GATE NAME: G8		
NO. OF NONMINIMAL CUTSETS=	2	
GATE NAME: G1		
NO. OF NONMINIMAL CUTSETS=		5
GATE NAME: TOP		
NO. OF NONMINIMAL CUTSETS=	5	
GATE NAME: G5		
NO. OF NONMINIMAL CUTSETS=	1	
GATE NAME: G6		
NO. OF NONMINIMAL CUTSETS=	2	
GATE NAME: TOP		
NO. OF NONMINIMAL CUTSETS=		20

Table (8.4.3) Cutsets Calculated for Fault tree in Figure (5.3.1)
with CUTNO = 0 and DEBUG = 1

TREE ANALYSIS BY MODULES

DATE: 10/10/79
TIME: 11:18.17.000
RUN TITLE: EXAMPLE

ESTIMATED MAX TOTAL GATES: 100
ESTIMATED MAX TOTAL COMPONENTS: 200
ESTIMATED MAX GATES INPUT TO A GATE: 5
ESTIMATED MAX COMPONENTS INPUT TO A GATE: 10

FAULT TREE INPUT

GATE NAME	TYPE	INPUTS
TOP	AND	G1 G2 G3
G1	2/3	G4 C1 C2
G2	AND	G5 C5
G3	AND	G6 C8
G4	OR	G7 C6
G5	OR	G8 C2
G6	OR	G8 C7 C3
G7	AND	C3 C4
G8	AND	G9 C5 C9
G9	OR	C10 C11

(continued)

Table (8.4.3) (continued)

NUMBER OF GATES= 10
 NUMBER OF COMPONENTS= 11
 MAX SIZE OF THE CUT_SETS= 8
 REPLICATED GATES: G8
 ON_OFF REPLICATED COMPONENTS: NONE

GATE	GATE INS	FREE LEAVES	DEP LEAVES	
TOP		3	0	0
G1		1	2	0
G2		1	1	0
G3		1	1	0
G4		1	1	0
G5		0	1	1
G6		0	2	1
G7		0	2	0
G8		1	2	0
G9		0	2	0

REP GATE 9 LEAF=29001

REP COMP=29001 APPEARANCES= 2

GATE NAME	MODULAR MINIMAL CUT SETS	
NESTID= 6	G5	
NESTID= 7	G4	

FREE G7 C3
 C4

FREE G9 C10 C11

FREE G4 C6
 G7

(continued)

CUT_SETS FOR GATE G8

Table (8.4.3) (continued)

MCS=	1	C5	C9	C10
MCS=	2	C5	C9	C11
FREE G8		C5 C9 G9		
SYMM G1		C1 C2 C1	G4 G4 C2	
NUM REPLICATED LEAVES IN MODULE= 2				
BOOLEAN HAS BEEN CALLED				
*TOP		C5 C8 G1		
NEST G5		C2		
NEST G6		C7	C3	

(continued)

Table (8.4.3) (continued)

RICS '1101'B		COMP= COMP=		'0111'B '1100'B		COMP= COMP=		'1110'B	
PROP	TOP	*TOP G8	G5 *TOP	G6					
CUT_SETS FOR GATE TOP									
MCS=	1	C5	C8	C2	C6	C7			
MCS=	2	C5	C8	C2	C6	C3			
MCS=	3	C5	C8	C2	C3	C4			
MCS=	4	C5	C8	C1	C2	C7			
MCS=	5	C5	C8	C1	C2	C3			
MCS=	6	C5	C9	C10	C8	C1	C6		
MCS=	7	C5	C9	C10	C8	C2	C6		
MCS=	8	C5	C9	C10	C8	C1	C2		
MCS=	9	C5	C9	C11	C8	C1	C6		
MCS=	10	C5	C9	C11	C8	C2	C6		
MCS=	11	C5	C9	C11	C8	C1	C2		
MCS=	12	C5	C9	C10	C8	C1	C3	C4	
MCS=	13	C5	C9	C11	C8	C1	C3	C4	

THE END

8.4.2 A Fault Tree with Different Types of Components and Gates

The fault tree in Figure (8.4.1)[1] is an arbitrary tree consisting of simple, replicated and ON-OFF complementary components, K-out-of-N, AND, OR, and replicated gates. The cutsets of the top event and the replicated gate cutsets are determined for the case that the SIZE variable is nine and SIZE1 is 4 (see Appendix D). A test run (CUTNO = 1) showed that 2772 non-minimal cutsets exist in this fault tree. The output consists of a check on the input as well as cutsets of the gates G1 and TOP. Table (8.4.4) shows the output of the MODCUT for the fault tree in Figure (8.4.1). The execution time for this run is 19.8 seconds in an IBM 370/168 computer for generating minimal cutsets of gate G1 and the TOP event.

8.4.3 A Fault Tree to Identify Solidified High-Level Waste Release from the Containment in the RSSF

The fault tree in Figure (8.4.2) was constructed to identify potential sequences of events by which solidified high-level waste could be released from containment during storage at the Radioactive Surface Storage Facility [13].

The input to the code MODCUT for calculating the cutsets is similar to the previous examples and is described in detail in Appendix D. The output from MODCUT is shown in Table (8.4.5). Minimal cutsets were calculated for the top event with variables SIZE = SIZE1 = 8. A total of 83 minimal cutsets were generated in 9.8 CPU seconds. The MOCUS and MICSUP codes have also been used for comparison with the MODCUT code. Output from these codes for the fault tree shown in Figure (8.4.2) is compared to that in Table (8.4.5). The result of this comparison

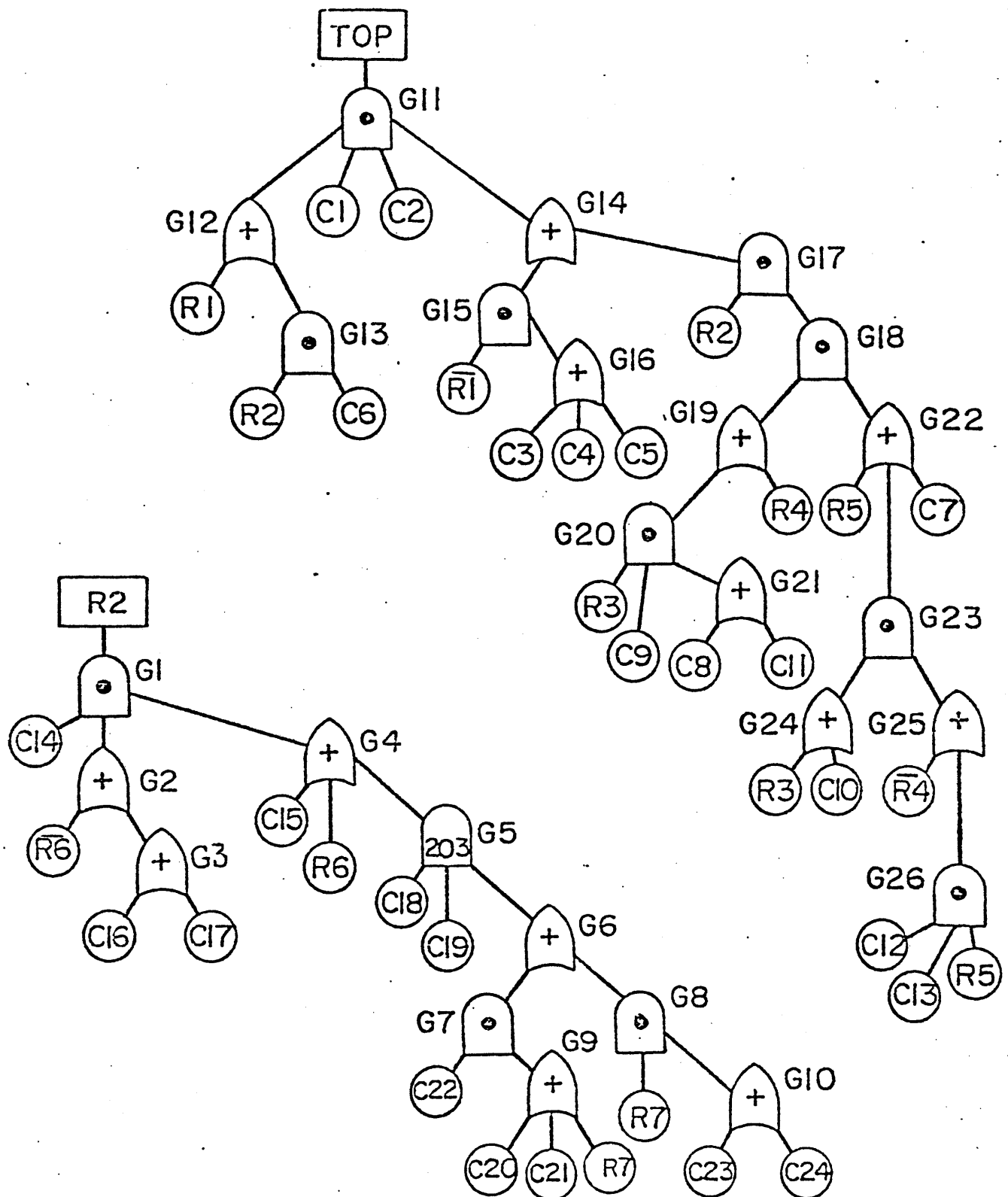


Figure (8.4.1)

SAMPLE FAULT TREE

Table (8.4.4) MODCUT Output for the Fault Tree in Figure (8.4.1)

 TREE ANALYSIS BY MODULES

DATE: 09/29/77

TIME: 17:15.33.000

RUN TITLE: PLMOD TEST RUN

ESTIMATED MAX TOTAL GATES:	100
ESTIMATED MAX TOTAL COMPONENTS:	200
ESTIMATED MAX GATES INPUT TO A GATE:	5
ESTIMATED MAX COMPONENTS INPUT TO A GATE:	10

(continued)

Table (8.4.4) continued

373

FAULT TREE INPUT				
GATE NAME	TYPE	INPUTS		
G1	AND	G2 C14	G4	
G2	OR	G3 R6		
G3	OR	C16	C17	
G4	OR	G5 C15	R6	
G5	2/3	G6 C18	C19	
G6	OR	G7	G8	
G7	AND	G9 C22		
G8	AND	G10 R7		
G9	OR	C20	C21	R7
G10	OR	C23	C24	
TOP	AND	G12 C1	G14 C2	
G12	OR	G13 R1		
G13	AND	G1 C6		
G14	OR	G15	G17	
G15	AND	G16 R1		
G16	OR	C3	C4	C5
G17	AND	G18	G1	
G18	AND	G19	G22	
G19	OR	G20 R4		
G20	AND	G21 C9	R3	
G21	OR	C8	C11	
G22	OR	G23 C7	R5	
G23	AND	G24	G25	
...	..	...		

(continued)

Table (8.4.4) continued

G24 UN C10 R3
 G25 OR G26
 ^R4
 G26 AND C12 C13 R5
 NUMBER OF GATES= 26
 NUMBER OF COMPONENTS= 30
 MAX SIZE OF THE CUT-SETS= 9
 REPLICATED GATES: 01
 ON/OFF REPLICATED COMPONENTS: R6 R1 R4

CUT-SETS FOR GATE 01

MCS=	1	C14	C16	C15	
MCS=	2	C14	C17	C15	
MCS=	3	R6	C14	C16	
MCS=	4	R6	C14	C17	
MCS=	5	^R6	C14	C15	
MCS=	6	C14	C16	C18	C19
MCS=	7	C14	C17	C18	C19
MCS=	8	^R6	C14	C18	C19

CUT-SETS FOR GATE TOP

MCS=	1	^R1	C14	C16	C15	C1	C2	C6	C3
MCS=	2	^R1	C14	C16	C15	C1	C2	C6	C4
MCS=	3	^R1	C14	C16	C15	C1	C2	C6	C5
MCS=	4	^R1	C14	C17	C15	C1	C2	C6	C3
MCS=	5	^R1	C14	C17	C15	C1	C2	C6	C4
MCS=	6	^R1	C14	C17	C15	C1	C2	C6	C5
MCS=	7	^R1	R6	C14	C16	C1	C2	C6	C3
MCS=	8	^R1	R6	C14	C16	C1	C2	C6	C4
MCS=	9	^R1	R6	C14	C16	C1	C2	C6	C5
MCS=	10	^R1	R6	C14	C17	C1	C2	C6	C3
MCS=	11	^R1	R6	C14	C17	C1	C2	C6	C4
MCS=	12	^R1	R6	C14	C17	C1	C2	C6	C5

(continued)

Table (8.4.4) continued

MCS=	13	R1	R6	C14	C15	C1	C2	C6	C3	
MCS=	14	R1	R6	C14	C15	C1	C2	C6	C4	
MCS=	15	R1	R6	C14	C15	C1	C2	C6	C5	
MCS=	16	R1	C14	C16	C15	R4	C1	C2	C7	
MCS=	17	R1	C14	C16	C15	R4	C1	C2	R5	
MCS=	18	R1	C14	C17	C15	R4	C1	C2	C7	
MCS=	19	R1	C14	C17	C15	R4	C1	C2	R5	
MCS=	20	R1	R6	C14	C16	R4	C1	C2	C7	
MCS=	21	R1	R6	C14	C16	R4	C1	C2	R5	
MCS=	22	R1	R6	C14	C17	R4	C1	C2	C7	
MCS=	23	R1	R6	C14	C17	R4	C1	C2	R5	
MCS=	24	R1	R6	C14	C15	R4	C1	C2	C7	
MCS=	25	R1	R6	C14	C15	R4	C1	C2	R5	
MCS=	26	C14	C16	C15	R4	C1	C2	C6	C7	
MCS=	27	C14	C16	C15	R4	C1	C2	C6	R5	
MCS=	28	C14	C17	C15	R4	C1	C2	C6	C7	
MCS=	29	C14	C17	C15	R4	C1	C2	C6	R5	
MCS=	30	R6	C14	C16	R4	C1	C2	C6	C7	
MCS=	31	R6	C14	C16	R4	C1	C2	C6	R5	
MCS=	32	R6	C14	C17	R4	C1	C2	C6	C7	
MCS=	33	R6	C14	C17	R4	C1	C2	C6	R5	
MCS=	34	R6	C14	C15	R4	C1	C2	C6	C7	
MCS=	35	R6	C14	C15	R4	C1	C2	C6	R5	
MCS=	36	R1	C14	C16	C18	C19	C1	C2	C6	C3
MCS=	37	R1	C14	C16	C18	C19	C1	C2	C6	C4
MCS=	38	R1	C14	C16	C18	C19	C1	C2	C6	C5
MCS=	39	R1	C14	C17	C18	C19	C1	C2	C6	C3
MCS=	40	R1	C14	C17	C18	C19	C1	C2	C6	C4
MCS=	41	R1	C14	C17	C18	C19	C1	C2	C6	C5
MCS=	42	R1	R6	C14	C18	C19	C1	C2	C6	C3

(continued)

Table (8.4.4) continued

MCS= 43	R1	R6	C14	C18	C19	C1	C2	C6	C4
MCS= 44	R1	R6	C14	C18	C19	C1	C2	C6	C5
MCS= 45	R1	C14	C16	C10	C17	R4	C1	C2	C7
MCS= 46	R1	C14	C16	C10	C17	R4	C1	C2	R5
MCS= 47	R1	C14	C17	C10	C19	R4	C1	C2	C7
MCS= 48	R1	C14	C17	C10	C19	R4	C1	C2	R5
MCS= 49	R1	R6	C14	C18	C19	R4	C1	C2	C7
MCS= 50	R1	R6	C14	C18	C19	R4	C1	C2	R5
MCS= 51	C14	C16	C10	C19	R4	C1	C2	C6	C7
MCS= 52	C14	C16	C10	C19	R4	C1	C2	C6	R5
MCS= 53	C14	C17	C10	C19	R4	C1	C2	C6	C7
MCS= 54	C14	C17	C18	C19	R4	C1	C2	C6	R5
MCS= 55	R6	C14	C10	C19	R4	C1	C2	C6	C7
MCS= 56	R6	C14	C18	C19	R4	C1	C2	C6	R5

THE END



377

Table (8.4.5) Outputs for the Fault Tree Shown in Figure 8.4.2

378

CUT SETS FOR GATE G2

HCS=	1	X6		
HCS=	2	X3		
HCS=	3	X8	X7	
HCS=	4	X9	X20	
HCS=	5	X14	X13	
HCS=	6	X16	X15	
HCS=	7	X17	X18	
HCS=	8	X21	X12	
HCS=	9	X22	X14	
HCS=	10	X23	X12	
HCS=	11	X32	X19	
HCS=	12	X33	X19	
HCS=	13	X36	X19	
HCS=	14	X38	X19	
HCS=	15	X23	X19	
HCS=	16	X24	X19	
HCS=	17	X42	X47	
HCS=	18	X29	X28	X12
HCS=	19	X40	X39	X19
HCS=	20	X20	X42	X48
HCS=	21	X13	X42	X50
HCS=	22	X7	X43	X47
HCS=	23	X15	X44	X47
HCS=	24	X17	X45	X47
HCS=	25	X32	X47	X46
HCS=	26	X33	X47	X46
HCS=	27	X36	X47	X46
HCS=	28	X38	X47	X46
HCS=	29	X23	X47	X46
HCS=	30	X24	X47	X46
HCS=	31	X21	X42	X49
HCS=	32	X22	X42	X49
HCS=	33	X23	X42	X49
HCS=	34	X23	X46	X49
HCS=	35	X7	X20	X43
HCS=	36	X7	X13	X43
HCS=	37	X20	X15	X44
HCS=	38	X13	X15	X44
HCS=	39	X20	X17	X45
HCS=	40	X13	X17	X45
HCS=	41	X12	X20	X43
HCS=	42	X12	X13	X43

(continued)

Table (8.4.5) (continued)

379

NCS=	43	X11	X29	X44	X46	
NCS=	44	X11	X13	X50	X46	
NCS=	45	X36	X29	X48	X46	
NCS=	46	X36	X11	X50	X46	
NCS=	47	X34	X20	X44	X46	
NCS=	48	X38	X13	X50	X46	
NCS=	49	X23	X20	X48	X46	
NCS=	50	X23	X13	X50	X46	
NCS=	51	X24	X20	X48	X46	
NCS=	52	X24	X13	X50	X46	
NCS=	53	X40	X39	X47	X46	
NCS=	54	X21	X7	X43	X49	
NCS=	55	X21	X15	X44	X49	
NCS=	56	X21	X17	X45	X49	
NCS=	57	X22	X7	X43	X49	
NCS=	58	X22	X15	X44	X49	
NCS=	59	X22	X17	X45	X49	
NCS=	60	X23	X7	X43	X49	
NCS=	61	X23	X15	X44	X49	
NCS=	62	X23	X17	X45	X49	
NCS=	63	X29	X28	X42	X49	
NCS=	64	X21	X32	X46	X49	
NCS=	65	X21	X33	X46	X49	
NCS=	66	X21	X36	X46	X49	
NCS=	67	X21	X38	X46	X49	
NCS=	68	X21	X24	X46	X49	
NCS=	69	X22	X32	X46	X49	
NCS=	70	X22	X33	X46	X49	
NCS=	71	X22	X36	X46	X49	
NCS=	72	X22	X38	X46	X49	
NCS=	73	X22	X24	X46	X49	
NCS=	74	X40	X39	X20	X48	X46
NCS=	75	X40	X39	X13	X50	X46
NCS=	76	X29	X28	X7	X43	X49
NCS=	77	X29	X28	X15	X44	X49
NCS=	78	X29	X20	X17	X45	X49
NCS=	79	X21	X40	X39	X46	X49
NCS=	80	X22	X40	X19	X46	X49
NCS=	81	X29	X29	X12	X46	X49
NCS=	82	X29	X24	X13	X46	X49
NCS=	83	X29	X29	X36	X46	X49
NCS=	84	X29	X28	X18	X46	X49
NCS=	85	X29	X29	X29	X46	X49
NCS=	86	X29	X24	X11	X19	X46

showed that all these codes provide the same set of cutsets for the fault tree in this example.

8.4.4 The Fault Tree for the New B&W Reactor Protection System

The system and its fault tree have been fully discussed in Section 8.3 of this chapter. The fault tree is input to the MODCUT code in order to determine cutsets of order 3 and less. In Table (8.4.6) all cutsets of order two and some of the important cutsets of order 3 are listed. The important cutset is referred to those cutsets with a high probability of occurrence.

The total CPU time for calculating over one thousand cutsets of up to order 3 is about 68 seconds. It should be emphasized here that by using failure data given in Table (8.3.8) for this fault tree it can be seen that the cutsets shown in Table (8.4.6) constitute about 98% of the total system unavailability. This value is obtained by using the top event unavailability calculated for this fault tree in Section 8.3.3 by the PL-MODMC code, and by using the unavailability contribution of the cutsets listed in Table (8.4.6) which can be found through a simple hand calculation. Also, the cutsets of order 2 represent about 80% of the total system unavailability, and the four cutsets that are of order 3 are, therefore, contributing about 18% of the total system unavailability. Table (8.4.7) shows the corresponding names in the fault tree for the names given in the MODCUT output for this example.

Table (8.4.6) Low Order Cutsets Obtained by the MODCUT Code for the Fault Tree in Figure (8.3.7)

CUT_SETS FOR GATE TOP

MCS=	1	BBFFBO	C1BFFBO	
MCS=	2	BBFFBO	C2BFFBO	
MCS=	3	ABFFBO	D1BFFBO	
MCS=	4	ABFFBO	D2BFFBO	
MCS=	5	ABFFBO	SCRSHSG5	
MCS=	6	ADFFBO	SCRSHSG6	
MCS=	7	ABFFBO	SCRSHSG7	
MCS=	8	ABFFBO	FOOLGD5A	
MCS=	9	ABFFBO	FOOLGD6A	
MCS=	10	ABFFBO	FOOLGD7A	
MCS=	11	ABFFBO	E2RFP0	
MCS=	12	ABFFBO	E3RFP0	
MCS=	13	ABFFBO	E4RFP0	
MCS=	14	BBFFBO	SCRSSSG5	
MCS=	15	BBFFBO	SCRSSSG6	
MCS=	16	BBFFBO	SCRSSSG7	
MCS=	17	BBFFBO	FOOLGD5B	
MCS=	18	BBFFBO	FOOLGD6B	
MCS=	19	BBFFBO	FOOLGD7B	
MCS=	20	BBFFBO	F2RFP0	
MCS=	21	BBFFBO	F3RFP0	
MCS=	22	BBFFBO	F4RFP0	
MCS=	23	CHARTMT	CHCRTMT	CHDRMT
MCS=	24	CHARTMT	CHCRTMT	CHDRMT
MCS=	25	CHARTMT	CHCRTMT	CHDRMT
MCS=	26	CHARTMT	CHCRTMT	CHDRMT

THE END

TABLE 8.4.7

CORRESPONDING NAMES FOR THE COMPONENTS LISTED IN THE MINIMAL CUTSETS

ABFPBO, BBFPBO	"A", "B", "C ₁ ", "C ₂ ", "D ₁ ", "D ₂ " Breaker
C1BFPBO, C2BFPBO	Faults Prevent Breaker
D1BFPBO, D2BFPBO	Opening
SCRSMBSG5, SCRSMBSG6	One SCR shorted on Main Bus Side
SCRSMBSG7	of Group "5", "6", "7" Regulatory Power Supply
FOOLDG5A, FOOLDG6A	Fault OUTPUT on one Leg of Gate
FOOLDG7A, FOOLDG5B	Driver "5A", "6A", "7A", "5B", "6B", "7B"
FOOLDG6B, FOOLDG7B	
E2RFPO, E3RFPO	"E ₂ ", "E ₃ ", "E ₄ ", "F ₂ ", "F ₃ ", "F ₄ "
E4RFPO, F2RFPO	Relay Faults Prevent Opening
F3RFPO, F4RFPO	
SCRSSSG5, SCRSSSG6	One SCR shorted on secondary side of
SCRSSSG7	Group "5", "6", "7" Regulatory Power Supply
CHARTMT, CHBRTMT	Channel "A", "B", "C", "D" RTM in Test
CHCRTMT, CHDRTMT	

REFERENCES

- [1] Olmos, J. and L. Wolf, "A Modular Approach to Fault Tree and Reliability Analysis", Dept. of Nuclear Engineering, MIT, MITNE-209, August, 1977
- [2] Caldorola, L., "Unavailability and Failure Intensity of Components", Nucl. Eng. Design, 44, 147 (1977)
- [3] "Reactor Safety Study", U. S. Atomic Energy Commission, Washington D. C., Appendix II
- [4] Vesely, W. E. and F. F. Goldberg, "FRANTIC: A Computer Code for Time-Dependent Unavailability Analysis", NUREG-0193, October, 1977
- [5] Lambert, H. E., "Fault Trees for Decision Making in System Analysis", UTRL-51829, October 1975
- [6] Vesely, W. E., "A Time-Dependent Methodology for Fault Tree Evaluation", Nuclear Engineering 13 (1970), 337-360
- [7] Modarres, M. and L. Wolf, "PL-MODT: An Extended Version of the PL-MOD Code for the Modular, Transient Fault Tree Analysis and Evaluation", Dept. of Nuclear Engineering, MIT, Internal Report
- [8] Modarres, M. and L. Wolf, "PL-MODT: A Modular Fault Tree Analysis and Transient Evaluation Code", Trans. Am. Nucl. Soc. 28, 510 (1978)
- [9] Wood, D. E. and N. J. Becar, "A Comparison of Results from the GO Methodology and Fault Tree Analysis", Kaman Sciences Corporation, K77-38U(R), August, 1977
- [10] "Reliability Prediction Electronic Equipment", MIL-HDBK-2178, September, 1974
- [11] "Summaries of Failure Rate Data", GIDEP Operations Center, Corona, California, August, 1975
- [12] Modarres, M. and L. Wolf, "PL-MODT and PL-MODMC: Two Codes for Reliability and Availability Analysis of Complex Technical Systems using the Fault Tree Modularization Technique", MITNE-222, Dept. of Nucl. Eng., MIT, November, 1978

CHAPTER 9

CONCLUSIONS AND RECOMMENDATIONS

9.1 Summary and Conclusions

The method of modular decomposition of a fault tree that has been used in the PL-MOD code is extended to include several calculational features, and to remove some of the restrictions of the original PL-MOD Code. The methodology to analyze a fault tree in terms of its modular structure had been previously developed [1]. This methodology consists of piecewise collapsing and modularizing portions of the tree, until eventually the fault tree structure is described as a set of modular equations recursively relating the top tree event to its basic component inputs. For fault trees with replicated events a Boolean vector representation was chosen to express the family of minimal cutsets corresponding to a higher order gate. It has been demonstrated [1] that quantitative calculations such as the top event probability and importance parameters may be efficiently evaluated by using the modular structure of the fault tree. In particular, the PL-MOD Code demonstrated the ability to evaluate large fault trees in an economic, efficient, and accurate manner. The PL-MOD Code is written in PL/I language in order to use the dynamic core utilization techniques available in this language. Accordingly, the extensions of the code presented in this thesis have all been performed by using the PL/I language, too.

In this thesis the method of modular decomposition is developed to include time-dependent fault tree evaluation, Monte-Carlo simulation of fault tree failure parameters, fault tree reduction based on the Vesely-Fussell

importance measures, and generation of minimal cutsets from the modular minimal cutsets. Also, an extensive review of the common cause failure analysis methodologies is given and the potential use of the modularization technique in common cause failure analysis is studied.

Time-dependent models were developed to include the effect of different classes of time-dependent components in a fault tree, on the top event unavailability at different instants of time. Mathematical developments showed that proper modules can also be approximated as time-dependent components and still maintain the same top event unavailability. These models are incorporated into the PL-MOD code in order to obtain the code PL-MODT. It is proved, through extensive examination and comparison with other codes, such as FRANTIC and KITT, that PL-MODT is sufficiently accurate and fast to be used for time-dependent evaluation of large fault trees.

One of the features that is incorporated into the PL-MODT is the automatic reduction of large fault trees. In this procedure a CUTOFF value is defined for the V-F importance measures in order to eliminate those components and modules of the fault tree whose importance are less than this value. The elimination of "non-important" components and modules in the fault tree provides an objective basis with which to reduce large fault trees. This procedure is performed in the Code PL-MODT very efficiently and takes only a negligible fraction of the total fault tree evaluation time consumed by the computer for modularizing the tree and calculating the probabilities.

For the PL-MODMC Code a Monte-Carlo simulation package is incorporated into the PL-MOD Code for the Monte-Carlo simulation of those parameters of the fault tree for which uncertainties can be assigned. The log-normal distribution is chosen to represent the unavailability spread for the parameters of the fault tree. Four classes of components are chosen to represent different types of components in a fault tree. Failure rates, repair duration, average downtime, and unavailability per demand are the parameters in these four classes of components to which log-normal distributions can be assigned. A PL/I random number generator is designed to be used in the procedure to select samples from the log-normal distribution. The accuracy of this random number generator is tested by comparing it to other random number generators (e.g. the one in the LIMITS code [2]), and by using the Chi-square test. A fast sorting routine has been employed similar to that used in the LIMITS Code [2]. Some mathematical models were developed in order to approximate unavailability distributions for evaluating equivalent log-normal distribution for the modules of the fault tree. These approximations have resulted in an enormous reduction in the computational time for evaluating the top event unavailability distribution.

In PL-MODMC, the restriction of not allowing a replicated component to be repeated within all replicated domains and the top event of the fault tree is removed. Thus, in the new form of the Code a replicated component can appear in all domains of the fault tree without any restriction. A preprocessor is designed to change the input and output to PL-MOD in order to be able to input alphanumeric names for components and gates of the fault tree. Also, many unnecessary input values to PL-MOD that can be calculated by the code from the logical definition of the

tree are omitted and procedures are employed to calculate these values directly by the code itself (e.g. number of appearances of a replicated component, number of gates, ..., etc.). These provisions help the user. The code is also tested for its accuracy by comparing it to other codes. Thereby it is found that PL-MODMC is especially suited for large fault trees. In particular, the comparison with codes such as LIMITS and SAMPLE [6] has proved the accuracy and efficiency of PL-MODMC.

An algorithm is developed based on deriving all simple minimal cutsets from the modular cutsets. This algorithm allows the determination of cutsets of all proper modules presented in the set of modular cutsets to be easily calculated. Then by assigning these cutsets to the corresponding modular minimal cutsets, a complete set of simple cutsets for the top event of the tree can be attained. This algorithm is incorporated into the PL-MOD Code in order to obtain the MODCUT Code. The advantage of this code is its capability to eliminate non-essential simple cutsets in the replicated and higher order modules before they are expanded. Also, the use of modular minimal cutsets to generate simple cutsets has been shown to be beneficial, because modular cutsets contain all simple cutsets, and generation of cutsets of the modules is very simple and straightforward. Intermediate elimination of the cutsets that have no significance on the number and order of the top event cutsets has proven to be a useful tool in the MODCUT Code and reduces computation time. For example, those cutsets which exceed the prespecified size of the cutsets of the top event can be easily eliminated; or, in a common cause analysis, those cutsets which are non-common-cause contributors can be removed. Therefore, MODCUT is essentially a discriminator code

based on any defined discrimination criterion (e.g. common cause failure, cutset probability, ..., etc.). A very high level of computational efficiency might be obtained by MODCUT because of these discrimination procedures. In MODCUT the same preprocessor which is described for the PL-MODMC code has been used to input only alphanumeric names, and similarly the restriction with regard to the appearance of replicated components is also removed.

Common cause failure analysis is an increasingly important segment of a complete system reliability analysis. It was believed that the modularization technique provided an easy tool with which to analyze the common cause failures in a complex technical system. Qualitative and quantitative aspects of common cause analysis have been studied. In particular, a quantitative model has been developed to evaluate system unavailability due to the presence of the common cause failures from the minimal cutsets of the fault tree which have been identified as common cause candidates. The results obtained for system unavailability from this model are in close agreement with other available models. Also, the codes MOCUS [3], and BACFIRE [4], and MODCUT and BACFIRE are coupled to obtain convenient packages (i.e. MOCUS-BACFIRE, and MODCUT-BACFIRE) for identifying common cause candidates from fault trees of safety systems. The results obtained through the examination of the common cause failure methods showed that the code MODCUT can be easily used to identify common cause candidates by incorporating a common cause discrimination criterion into this code. However, direct common cause discrimination capability is not yet incorporated into the MODCUT code, but if this code is coupled with the BACFIRE code it provides an alternative to MOCUS-BACFIRE for the analysis of complex fault trees.

Several examples for evaluating fault trees of different sizes and properties are provided in Chapter 8. These fault trees are analyzed by using the PL-MODT, PL-MODMC and MODCUT codes. The results obtained from these examples all support the aforementioned features of the PL-MODT, PL-MODMC and MODCUT codes.

Their application in future studies is recommended.

9.2 Recommendations

The structural and logical limitations described in the PL-MOD code are exactly the same for the PL-MODT code. Therefore, the same preprocessor and procedure to remove the restriction concerning the replicated components that are used in the PL-MODMC should also be added to the PL-MODT code.

Methodologies should be developed in order to automatically select proper time step sizes during different periods of system operation, e.g., to select finer mesh time steps during the inspection time of the periodically tested components in the fault tree. It would be highly beneficial to implement an option which allows the use of a Calcomp plot to display the time-dependent unavailability functions after they are calculated by the PL-MODT code. This plotter option should possess the same flexibility as that originally provided in the FRANTIC code by the NRC.

The reduction capabilities incorporated into the PL-MODT provide a basis for objective reduction of complex fault tree branches and leaves which are not significantly contributing to the failure of the top event. Automatic reduction of a fault tree prior to the time-dependent calculations performed by PL-MODT is highly recommended.

PL-MODT calculates the Vesely-Fussell importance measures for each time step. However, as has been discussed in section 8.2.6 of this thesis, this seemingly provides less information than a unique value of importance given for the entire period of operation. Lambert [5] uses the following expression for the importance of repairable components (monitored components):

$$I_i = \frac{[g(1_i, A) - g(0_i, A)] / (\mu_i + \tau_i)}{\sum_{j=1}^n [g(1_j, A) - g(0_j, A)] / (\mu_j + \tau_j)}$$

where μ_i : Mean time to repair (MTTR) for component i ;

τ : Mean time to fail (MTTF) for component i and;

$g(1_i, A) = (A_1, \dots, A_i = 1, \dots, A_n)$ conditional structure function

It is suggested that this expression also be employed in PL-MODT so as to obtain a unique value of the importance for a system consisting of Class 3 components. For the importance measure of periodically tested components (Class 4), it is suggested that the total period of component unavailability function be divided into the following three regions:

- (a) test region
- (b) repair region
- (c) operational region

The idea behind this division is to avoid abrupt change in the unavailability within transitions between the different periods of system operation. In this manner, it is believed that a unique constant value for the importance within each specific region should be generated. For each region the importance of the component should be appropriately

defined. For instance, for the test region it can be the probability that the periodically tested component i causes system failure under the assumption that the system and component unavailabilities are constant during test and repair periods.

Implementation of other types of probability distribution into the PL-MODMC is recommended. In particular, normal, uniform, exponential gamma, beta, binomial, Poisson, Weibull, and empirical distributions (Johnson's method). Also, the possibility of having a combination of these distributions in a fault tree would provide a reliable way of more realistically evaluating a fault tree. Attempts must be made to fit other types of distributions to the OR module distribution, especially when the number of components input to the OR module is statistically significant. The use of Johnson's distribution is highly recommended as an approximate distribution for the OR module distribution.

It would be beneficial if the fault trees were reduced prior to the Monte-Carlo simulation by the PL-MODMC code, particularly if an automated method could be implemented so as to automatically exclude portions of the fault tree that are not contributing to the failure of the top event. Especially, the fault tree reduction based on the Vesely-Fussell importance measure is suitable for this purpose, but it should be noted that the CUTOFF value of a component must be reduced by a factor F , where F is the error factor associated with that component. This is because of the fact that the sampling routine might select an unavailability of an order F greater than the median unavailability and therefore, the effect of high unavailability values must be accounted for. It should be noticed that the Vesely-Fussell reduction option presently exists in the PL-MODT but has not been automated.

The Code MODCUT is constructed for cutset discriminations based on any defined criterion. At the present time only cutset size discrimination is performed by the MODCUT. It is recommended, however, that other important discrimination criteria be included such as common cause failure, cutset probability, component types in a cutset, and a combination of these criteria. In particular, common cause failure analysis can be easily implemented, and a great deal of advantage would be obtained. In this case all of the cutsets that are common cause contributors can be economically obtained. It is recommended that criteria for identifying a common cause failure candidate be revised so as to include, in the list of common cause candidates, those cutsets that have two or more components susceptible to a common cause. This would provide a complete set of common cause candidates for quantification purposes. It should be noted that the susceptibility criterion used in the BACFIRE code to identify common cause candidates is based on the assumption that a cutset is a common cause candidate if all of its components are susceptible to the same cause and they are located in the same physical domain. This would, however, eliminate from the list of common cause candidates those cutsets in which only some, but not all, of the components are susceptible to a common cause and which are located in the same physical domain. Elimination of these cutsets would affect the quantification procedure, and the unavailabilities would be underestimated, although not significantly most of the time.

The quantification model developed in Chapter 6 of this thesis can be easily incorporated into the MODCUT code, in order to provide a complete list of common cause candidates for the top event unavailability

calculations including the common cause failure effect. Obviously, if the system contains monitored components (Class 3), one should use the Markovian Model discussed in Figure (6.3.1).

The codes PL-MODMC and MODCUT can not handle fault trees in which a replicated gate appears in the domains of two or more other replicated gates (i.e., similar to the PL-MOD Code restriction of not allowing a replicated component to appear in all domains of the fault tree, which was removed in the research presented in this thesis during the development of MODCUT and PL-MODMC codes).

It is recommended that the method of modular decomposition be extended in order to handle EXCLUSIVE OR, NAND, NOR, and SPECIAL gates represented by any valid Boolean expression defined by the user. This would enable the user to evaluate fault trees that are non-coherent and, if implemented in the MODCUT code, one would be able to generate "prime implicants" of a fault tree without any change in the procedures added to the PL-MOD code in order to develop the MODCUT code (i.e., CUT-SET and TOP-SET procedures).

During the evaluation of the new B&W Reactor Protection System described in Section (8.3.3) of this thesis it became apparent that for complex fault trees additional core storage is required because that provided by the main memory of the computer is not sufficient. The reason is that the PL-MODT, PL-MODMC and MODCUT codes store all of the fault tree informations in the main memory of the computer and apparently, for complex fault trees, the use of an external core storage facility becomes inevitable. The addition of a Random Access Temporary storage device would greatly increase the potential use of the code to process large fault trees. In this case the bit string elements

associated with the minimal or modular minimal cutsets will be stored in the out-of-core storage facility.

Fault tree diagrams for certain complex safety systems often contain series of sub-trees of identical structure. This simply reflects the various sub-system redundancies which the safety system contains. It is recommended that advantage be taken of this structural feature in MODCUT by handling this set of subtrees as single "Vector" structures. As a result, the safety system design corresponding to a fault tree will become immediately discernible and more compact and an efficient analysis of the fault tree will be possible.

REFERENCES

- [1] Olmos, J., "A Modular Approach to Fault Tree and Reliability Analysis", Ph.D. Thesis, MIT Dept. of Nucl. Eng., August, 1977
- [2] Lee, Y. T., and S. L. Salem, "Probability Intervals for the Reliability of Complex Systems using Monte-Carlo Simulation", UCLA-ENG-7758, December, 1977
- [3] Fussell, J. B., et. al., "MOCUS - A Computer Program to Obtain Minimal Sets from Fault Trees", Aerojet Nuclear Co., ANCR-1156 (August, 1974)
- [4] Cate, C. L. and J. B. Fussell, "BACFIRE - A Computer Code for Common Cause Failure Analysis", University of Tennessee Knoxville, Tennessee, May, 1977
- [5] Lambert, H., "Fault Trees for Decision Making in System Analysis", UCRL-51829, October, 1975
- [6] "Reactor Safety Study", WASH-1400, U. S. Nuclear Regulatory Commission (NUREG-75/014), 1975

APPENDIX A

FAULT TREE EVALUATION CODES

A.1 Introduction

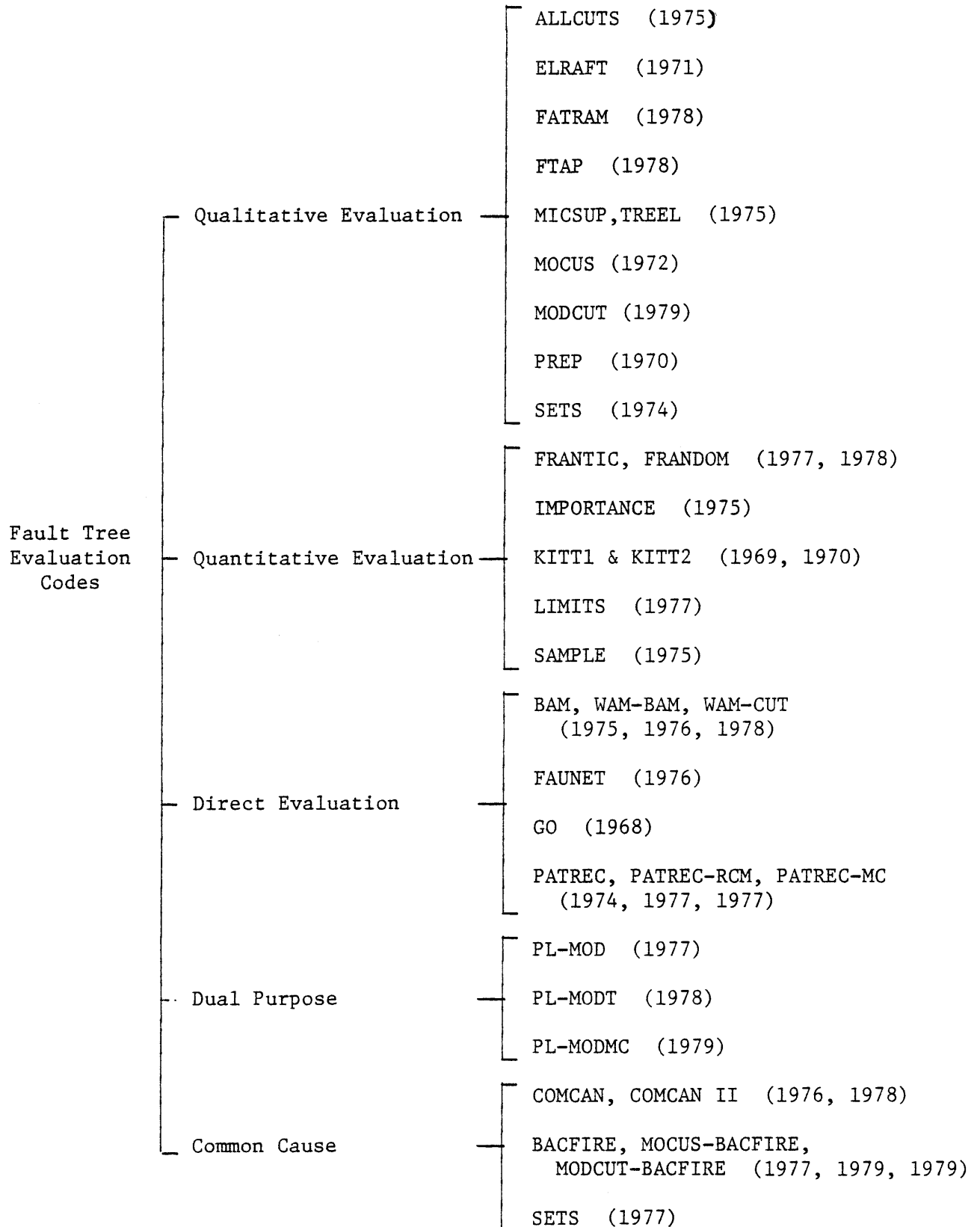
Fault tree evaluation codes that are currently available can be divided into five groups [1], namely;

1. qualitative evaluation codes
2. quantitative evaluation codes
3. direct evaluation codes
4. dual purpose codes
5. common cause analysis

This Appendix contains a brief description of some of the well-known codes that are commonly used. Figure A.1 illustrates these five groups and their associated codes.

FIGURE A.1

FAULT TREE EVALUATION CODES



A.2 Qualitative Analysis Codes

(a) ALLCUTS:

ALLCUTS is a computer code [2] written in FORTRAN IV for the CDC 6600 computers to generate minimal cutsets. It uses a top-down algorithm similar to that of MOCUS. Auxiliary programs are provided to check the input data and plot the fault tree diagram based on the input description and conversational plotting instructions. It finds cutsets of up to size ten. ALLCUTS can sort cutsets based on their probabilities. Up to 175 basic events and 425 gate events can be handled by this code. Probability calculations are also performed by the code upon request, and the calculation includes the modeling of repairable and non-repairable components, standby and on-line components. By using the rare event approximation, the probability of the top event is also calculated.

(b) ELRAFT:

The Efficient Logic Reduction Analysis of Fault Trees (ELRAFT) [3] is a computer code to determine minimal cutsets of fault trees using the unique factorization property of natural numbers. Every integer greater than one can be expressed as a product of powers of prime factors, except for the order in which the factors appear. Because of the unique factorization property, the cutsets for higher level gate events are determined through a bottom-up algorithm as a product of the numbers associated with each of the input events. Therefore, cutsets indicated by, say 30, are logically contained in the cutsets by 15, and the cutsets with the larger number are non-minimal and can

be eliminated. The major drawback of ELRAFT is that it requires large amounts of storage capacity. It is written in FORTRAN IV for the CDC 6600 and is capable of finding cutsets of up to 6 basic events.

(c) FATRAM

FATRAM (Fault Tree Reduction Algorithm) [4] is a top-down algorithm similar to that of MOCUS. Beginning with the top event, FATRAM resolves the gates. If the top event is an AND gate, all inputs are listed as one set; and if it is an OR gate the inputs are listed as separate sets. Iteration to resolve gates continues until all OR gates with gate inputs and all AND gates are resolved and replaced by their input components. At this point non-minimal cutsets are removed. FATRAM will process any replicated events remaining in the unresolved OR gates, by replacing them with all unresolved gates of which this replicated event is an input. These new sets are added to the collection and replicated events are disconnected from the gate, and minimal cutsets are found. Finally, all remaining OR gates are resolved.

FATRAM handles only AND and OR gates. It's computation time is highly dependent upon the logical configuration of the fault tree. FATRAM is written in FORTRAN for the CDC Cyber 76 computers.

(d) FTAP

FTAP is a computer program that finds prime implicant families of a fault tree [5]. This program is written in FORTRAN language for the CDC 6600/7600 computer series. Some of the operations in the code which cannot be handled by FORTRAN are written in Assembler language.

A pre-processor checks the consistency and the correctness of the input data.

The code is based on a semi-modularization method in which only portions of the tree which are considered as "independent branches" are modularized (i.e. simple modules are created). Three methods of analysis are available and the user can choose one of them along with some options to specify any special procedure that the method should perform. The methods are NELSON, MSUP and MSDOWN. These methods are fully described in Reference [5]. FTAP is available in two distinct versions that differ only in the internal storage format for representing implicant sets. FTAP1 stores implicants as a variable number of consecutive computer words in the main memory. FTAP2 uses a bit of the computer to store an implicant, and therefore, implicants will be stored as a fixed group of consecutive words. FTAP2 is more suitable to be used for large fault trees and is limited to implicants with small number of events.

(e) MICSUP and TREEL

MICSUP and TREEL [7] is a computer code that generates cutsets, and employs an upward algorithm, starting with the basic events at the bottom of the tree and working upward to the top event. TREEL reads the input representation of the tree to check the tree for errors, and determines in advance, number and size of the BICS. The MICSUP and TREEL programs are coded in FORTRAN for the CDC 6400 Computer. The program output includes the minimal cutsets for the top event and for any requested intermediate event. Most of the features of these codes are similar to MOCUS, except MOCUS's algorithm is downward.

(f) MOCUS

MOCUS [8,9] is a computer program which determines the minimal cut and path sets of a fault tree. It is written in FORTRAN IV for the IBM-360/370 computer series. MOCUS is designed to replace the PREP code and can be used as the cutset generator of the KITT-1 and KITT-2 codes. MOCUS utilizes a top-down algorithm and performs a successive substitution of gate equations, to move down the tree until only basic events remain. The program handles only AND and OR gates, and all minimal cut and path sets of a fault tree of up to a size 20 can be handled. All types of events including house events are handled by this code. MOCUS can determine cutsets of all intermediate gates upon request of the user. Compared to the PREP code, MOCUS is much faster and more efficient. An input preprocessor extensively checks the input for errors.

(g) MODCUT

MODCUT code is written in PL/I language and is part of the research presented in this thesis. For a complete mathematical and technical description of MODCUT the reader is referred to Chapters 5 and 7. A few examples are also provided in Chapter 8. The MODCUT algorithm is based on the modularization technique employed in PL-MOD and developed by Olmos and Wolf [10, 11, 12] .

In the MODCUT algorithm, cutsets are generated through modular cutsets of a fault tree which is a compact representation of all simple cutsets of that tree. Hence, by generating cutsets of the modules of a tree (i.e. proper and improper modules) and by assigning them to the

modular cutsets, all simple cutsets can be obtained. The advantage associated with this code is its capability to eliminate non-essential cutsets of modules before assigning them to the modular cutsets. At the stage where cutsets of modules are generated, it is possible to discriminate among cutsets of an intermediate gate's module, based on size, probability, common cause failure, ..., etc. This substantially reduces the computation time, since simple cutsets do not expand significantly. Currently, elimination of the cutsets are based only on the size (order) of the cutsets. However, the code is constructed such that other methods of discrimination can be easily incorporated. There is no limit on the size of the cutsets or the size of the fault tree. Therefore, if desired all of the cutsets of a tree can be calculated, provided that sufficient core storage is available. A special procedure is incorporated to calculate the number of cutsets that exist for each gate event, and the top event prior to the calculation of the cut sets.

The cutset generation in MODCUT is numerically efficient because of an extensive use of list processing techniques, and bit manipulation available in the PL/I language. The code can also handle K-out-of-N gates and components with a corresponding complement in the fault tree.

(h) PREP

The PREP program [13] is designed to determine minimal cutsets. PREP consists of two parts: PREP-TREBIL and PREP-MINSET. TREBIL determines the state of a system given the state of all basic events. It produces a subroutine that MINSET uses to find minimal cutsets.

PREP-MINSET has two options for cutset generation: COMBO and FATE. The COMBO option determines minimal cutsets by a process of deterministic testing. COMBO systematically postulates that basic events have occurred one at a time, then two at a time, ..., etc. For each combination, the logic of the fault tree is tested to determine if any combination of the primary events causes the top event, and, if so, they will be identified as a cutset. Finally, nonminimal cutsets will be removed. FATE incorporates quantitative data on the component's reliability to find only "most likely" minimal cutsets. This is performed by Monte-Carlo trials. Obviously, PREP requires prohibitive computer time for large trees, while FATE is not guaranteed to find all of the minimal cutsets of a specific order. PREP is written in FORTRAN IV for IBM360/370 machines, and is limited to a maximum cutset size of 10.

(i) SETS

The Set Equation Transformation System (SETS) [14, 15] is a program which manipulates Boolean Equations. By applying Boolean identities the equation can be transformed into a more desirable form. By writing a SETS user program, the SETS code is able to dynamically manipulate the tree through the SETS user program. SETS is coded in FORTRAN for CDC6600/ 7600 machines. EOR gates and SPECIAL gates represented by a valid Boolean expression can be handled by the SETS code. SETS uses a bit level storage scheme and auxiliary storage for processing of large trees.

Four methods for processing a fault tree are available for the user. The first method is a simple step equation reduction which is very time-consuming, particularly for large fault trees. The second method

consists of separate processing of independent subtrees and, therefore, independent portions of the tree must be identified by the user, with an independent portion of the tree being the portion whose domain has no replicated event. The third method is processing the fault tree in stages, which include deriving the minimal cutset equation for intermediate steps in a series of steps, and by applying the Boolean identities the equations will be reduced at the intermediate stages. The fourth method is a combination of the second and third methods. The advantage associated with methods 2 through 4 is basically that they save computing time. The disadvantage is that the identification of independent subtrees and the section of the stages require an experienced user and extensive familiarity with fault tree evaluation strategies. The SETS algorithm for finding the prime implicants of a fault tree with complement events can be very time-consuming for even medium sized fault trees.

A.3 Quantitative Analysis Codes

(a) FRANTIC

The FRANTIC Computer Code [16, 17] (Formal Reliability Analysis including Normal Testing, Inspection and Checking) has been developed by the NRC to compute the average and time-dependent unavailability of any general system model such as a fault tree or event tree. A special set of equations is needed to determine, from the fault tree, the logical relations between the components of the system. This equation is called the System Unavailability Equation and should be determined by the user prior to the FRANTIC run. Four types of components can

be handled by FRANTIC; they are:

- i) constant unavailability (time-independent)
- ii) non-repairable (time-dependent)
- iii) repairable (monitored)
- iv) repairable (periodically tested)

The program can be used to determine the effects of test downtime, repair time, test inefficiency, test override probability and test-caused failures on the total system unavailability. Human error and common cause contributors can also be modeled by the code. Similar to the LIMITS and SAMPLE codes, the system unavailability equation which models the system must be inputted. Cal Comp Plots of the time-dependent system unavailability are also provided by FRANTIC.

The Monte-Carlo version of the FRANTIC Code, FRANDOM [17], is being made available. In this version probability distributions can be used as inputs for component failure rates. FRANDOM produces a probability distribution for the total system unavailability contribution due to random failure of components contribution due to test, and contribution due to repair. In FRANDOM, normal, lognormal, and uniform distributions can be assigned to define the uncertainty in the failure rates.

(b) IMPORTANCE

The Computer Code IMPORTANCE [18,19] has been developed at the U. C. Berkeley to rank basic events and cutsets according to various available importance measures. IMPORTANCE is written in FORTRAN IV for CDC 7600 computers.

The IMPORTANCE Code is capable of handling fault trees with time-dependent components under the assumption that basic components are statistically independent, and that their failure and repair distributions are exponential in time. The importance measures which are included in the IMPORTANCE Code are:

- i) Birnbaum [42]
- ii) Criticality [18]
- iii) Upgrading Function [18]
- iv) Vesely-Fussell [43]
- v) Barlow-Proschan [44]
- vi) Steady state Barlow-Proschan [44]
- vii) Sequential contributory [18]

Input to the IMPORTANCE Code is a list of cutsets along with basic component failure data. The output consists of the top event probability, importance, and ranking of basic components and cutset's based on one or more of the above-mentioned measures.

(c) KITT-1 and KITT-2

KITT-1 and KITT-2 [13,20] are two computer codes to perform fault tree quantification calculations based on the cutsets of a fault tree. They are written in FORTRAN IV for IBM 360/75 computers. Any of the cutset generating codes described before can be used to generate cutsets for use with the KITT codes. However, PREP and MOCUS generate the cutsets in a form which is directly usable as the KITT input. The KITT codes compute the following five probabilities based on the

exponential distribution of the input components (repairable or non-repairable) for the top event, basic event, and cutsets of the fault tree:

- i) the probability of failure at time t ;
- ii) the probability that failures do not occur up to time t ;
- iii) the expected number of failures occurring up to time t ;
- iv) the failure rate (hazard rate);
- v) the failure intensity per hour.

In addition to the capabilities of the KITT-1 code, KITT-2 is capable of performing the above calculations for dynamic failure data changes, such as for instance a fault tree which represents a system with different phases of operation at different periods of time. Based on the equations (1.3.4) and (1.3.5) presented in this thesis, the KITT codes can find upper and lower bounds for the probabilities which they calculate.

(d) LIMITS

LIMITS [21] is a Monte-Carlo simulation code similar to SAMPLE. The code assesses the uncertainty associated with the top event unavailability of a fault tree, through a Monte-Carlo Simulation. LIMITS requires the system unavailability equation as input to the code. Therefore, similar to SAMPLE and FRANTIC, it is limited to small systems for which this equation can be found. LIMITS handles only log-normal distributions for the failure rates of basic components. The LIMITS

Code is significantly faster than the SAMPLE code due to an optimized sorting routine [22] and a special random-number generator. The output of the LIMITS code gives the top event unavailability distribution along with probability measures to determine the accuracy of the top event unavailability calculated for each confidence interval. LIMITS is programmed in FORTRAN IV for the IBM 360/370 series.

(e) SAMPLE

SAMPLE [23] is a Monte Carlo simulation code written in FORTRAN IV for the IBM 360/370 series. SAMPLE was used in the WASH-1400 study to find upper and lower bounds of system failure probabilities for Nuclear Reactor Safety systems. A system unavailability function representing the logical configuration of the system components must be supplied as an input to SAMPLE. The Monte Carlo simulation is therefore based on the system unavailability equation. The Monte-Carlo simulation is performed by sampling component failure rates from their input probability distributions and finding the system unavailability corresponding to this "trial". After several trials the system unavailabilities are sorted, and unavailabilities corresponding to various confidence levels are obtained. SAMPLE can use primary failure rate data with either normal, lognormal, or uniform distribution. Once selected, the same form of distribution is used for all components throughout the problem. After 200 trials, results are ordered and the accuracy is tested. The program continues for another 200 trials and this process is repeated until the number of specified trials are completed or a maximum of 4800 trials is reached. SAMPLE is very inefficient with respect to its sorting procedure

of the unavailabilities. Also, the provision of the System Unavailability function is a non-trivial task for large fault trees. An extension of the SAMPLE code is MOCARS [24] which is similar to SAMPLE, but it can handle additional distributions including gamma, beta, binomial, poisson, Weibull and any other empirical distribution.

A.4 Direct Evaluation Codes

(a) WAM-BAM

The WAM-BAM Codes [25, 26, 27] were developed at Science Applications, Inc. and are written in FORTRAN for CDC 6600 computers. The WAM-BAM package consists of four codes: WAM, WAMTAP, BAM, and WAMCUT. WAM and WAMTAP are input preprocessors to generate a set of corresponding numeric descriptions of the components and gates of the fault tree instead of the input alphanumeric names for use in the evaluation code BAM (Boolean Algorithm Model). In addition to WAM, WAMTAP prepares a numerical description of the tree, such that a sensitivity analysis can also be performed by the BAM Code. WAMCUT is a cutset generator code, and evaluates mean and variance of the probability of the occurrence of any gate. The BAM Code is similar to the concept of the GO methodology but applies only AND and OR operations that are used in a fault tree diagram. BAM calculates the top event probability via a truth table approach. As in GO, primary techniques must be used; otherwise the truth table may soon exceed the available storage. Each line of the truth table represents a product term (P term) event disjoint from all other P terms. The product of the probabilities of the events in each P term gives the probability of P term,

and the union of the applicable P terms gives the probability of the top event.

(b) FAUNET

FAUNET program [28, 29] is a program to evaluate fault trees and networks based on a modularization algorithm. This code is written in FORTRAN for the PDP8 mini computers with 8K or 16K under OS/8. FAUNET contains a procedure to check the fault tree input data for logical consistency and provides a graphical representation of the tree. A special modularization algorithm is used to modularize the fault tree prior to finding minimal cutsets. The fault tree only operates on AND and OR gates, and handles replicated events. Components must be indexed from 1 to 999 and gates from 1000 up.

The Modularization algorithm starts by collapsing identical gate types. Then a "factorization" algorithm performs a bottom-up analysis to identify pairs of basic events always occurring together in the same types of gates (replicated events). Such a pair is replaced by a dummy event that is called a "complex event". The tree gates are further collapsed and the search continues for more identical complex and basic events, until no more pairs are found. A result file stores all information about the complex events. From the minimal cutsets of the modularized tree, the top event time-dependent unavailability and Murchland's failure intensity will be found. The obvious limitation of this code is that the core storage capacity might exceed that available in a mini computer if a large fault tree is to be run and alphanumeric names cannot be used as inputs for identification of components and gates of the tree.

(c) GO

The CO Methodology [30,31] is a probabilistic combinatorial analysis procedure which was developed by Kamman Science Corporation in the mid-1960's. Component probabilities and logical interactions are combined to produce the probabilities of several output events in several operating states. The methodology includes:

- i) A set of standardized functional operators which are used to model physical component events with mathematical entities which are easily identified as components.
- ii) A modeling technique whose result is the GO chart. This chart co-responds closely to the physical layout, diagram, or schematic. For the modeling procedure, 16 GO operators are used. Some of these operators are similar to fault tree gates, but in addition to logic functions, time delays and switches can be modeled as well as complementary event logic and mutually exclusive states. A special method is used to model replicated components which are referred to as "super type". The development of the GO chart consists of selecting operators to represent each component, and connecting them with arrows to represent the flow of information. The user must specify the probabilities associated with the possible operational modes of each component, which is analogous

to supplying failure probabilities for the components of a fault tree.

- iii) The GO computer code would finally perform the logical connections, Boolean Algebra, probability calculations and the generation of min cutsets of up to order 4. Probabilities are traced by the model, using a Markov Chain approach. This means that a change in component probability requires a complete reevaluation, even though the system structure remains unchanged.

Because of the diversity and detail of the GO operator and the necessity of including all system components, the modeling process is quite complex. However, the use of the GO chart can be useful for design and system engineering. The GO code is written in FORTRAN for CDC 7600 computers.

(d) PATREC (PATREC-RCM and PATREC-MC)

PATREC [32, 33, 34] is a computer program to evaluate the top event probabilities of a fault tree by using a "pattern recognition algorithm". PATREC is the first fault tree analysis code which is written in PL/I and uses the list processing technique for efficient use of the core memory. Although the PL-MOD code uses a completely different algorithm for evaluating a fault tree, it has a somewhat similar programming organization to that of PATREC. The PATREC algorithm consists of identifying subtree patterns. Each subtree pattern is later assigned as a super-component (leaf) which represents the subtree for the rest of the evaluation. The probability of each subtree pattern is

calculated through its input components, and is assigned to the corresponding supercomponent. By continuing the search for subtrees with or without supercomponents, and assigning new supercomponents, the tree is finally represented by only one supercomponent. It should be noticed that patterns are stored in a library, and thus, the approach is limited to a limited amount of patterns.

PATREC can evaluate trees consisting of components with a corresponding complement elsewhere in the tree as well as K-out-of-N gates. Cutsets can be generated by using an algorithm similar to that of MOCUS. PATREC cannot handle replicated events and therefore a new version of PATREC (PATREC-RCM) was developed to remove this restriction. The limitations of the PATREC-RCM code is its way of handling replicated events. Since the pattern recognition algorithm only yields exact probabilities, the PATREC-RCM replaces a fault tree with n replicated events to 2^n fault trees (2^n is the total possible states that combinations of n replicated components can have). Then by multiplying the probability of the existence of each state to that of the top event of the fault tree, and adding the resulting values for all 2^n fault trees, the top event probability is found. However, an optional procedure has been added so that only those states with significant probability of existences are taken into account. However, identification of the states with low probabilities is somewhat difficult and results in a subjective TOP event probability.

PATREC-MC is a Monte-CARLO version of the PATREC-RCM. PATREC-MC's principle is similar to SAMPLE, except that the system function is not required. A sampling method is incorporated into the PATREC-MC so as to take sample probabilities from only significant contributing events.

PATREC-MC allows the propagation of failure rates with only log-normal distribution. PATREC-MC's output consists of the top event's distribution with 5% and 95% confidence limits and the median value.

There is no limit on the size of the fault tree, but the number of replicated events plays a key role in limiting the use of the PATREC code series. For more information on the mathematics and methodology of the PATREC code, the reader is referred to section 1.4.3.2 of this thesis.

A.5 Dual Purpose Codes (the PL-MOD Series)

(a) PL-MOD

PL-MOD is a computer code [10] developed at MIT, and is based on a particular modularization algorithm that finds modules directly through the description of the fault tree diagram. For more information about the methods of modularization employed in PL-MOD the reader is referred to section 1.4.2 of this thesis. The modularization process used by PL-MOD is unique in that it is applied not only to the cutsets, but directly to a description of the fault tree diagram, using the list processing features of the PL/I language.

Advantages associated with the PL-MOD approach are as follows:

- i) The structure of the tree is more easily comprehensible from the modular description, because the modular cutset of a fault tree is a condensed representation of all simple minimal cutsets of the same tree. Modular cutsets represent all the necessary information for further qualitative and quantitative analysis of the tree. These modular cutsets

are obtained without using the time-consuming cutset enumeration process.

- ii) PL-MOD is able to handle complement events, direct field input, and dynamic storage allocation or release in order to efficiently use computer storage capacity. There are no constraints on the total number of basic events or gates in the tree, or the number of inputs to any particular gate. The only constraint is on the total amount of available core storage because all of the fault tree information is stored in the main memory of the computer. The output from PL-MOD includes the modular minimal cutsets for the top event, replicated gates, and K-out-of-N gates along with their input gates.

A disadvantage of PL-MOD is its machine dependence (PL/I is available on IBM, CDC, Honeywell, Univac). The present version of PL-MOD does not allow symmetric gates whose inputs are replicated elsewhere in the tree. A replicated component may only appear within the domain of a replicated gate of the original fault tree. Therefore, a replicated component cannot exist both in the domain of a replicated gate, say G1, and the domain of the fault tree with top event, say TOP. However, most of these restrictions have been removed in the new version of the code (PL-MODMC and MODCUT) as a result of this thesis research.

The quantitative calculations in the code PL-MOD include the computation of occurrence probabilities and Vesely-Fussell importance measures for all modules, including the top event. Two advantages of the

modular approach for quantitative calculations are (1) it is faster than evaluating probabilities from simple cutsets and (2) it results in tighter bounds on the probability of occurrence of the top event.

(b) PL-MODT

The original PL-MOD code can only handle class one components (i.e. constant failure, time-independent components), and thus it is only able to evaluate time-independent components. The PL-MODT code is constructed to incorporate into the PL-MOD the capability of handling time-independent components. The PL-MODT code is constructed to incorporate into the PL-MOD the capability of handling time-dependent components [12, 35]. There are three time-dependent classes of components (except the Class 1 which is a time-independent class) that are used in PL-MODT as follows:

- (1) non-repairable
- (2) repairable monitored
- (3) periodically tested.

The mathematical descriptions of these three time-dependent classes of components are extensively discussed in Chapter 2 of this thesis. PL-MODT has all the features of the PL-MOD code and therefore, the calculations are based on the modular cutsets which are very efficient for the fault tree quantitative evaluations. In addition to time-dependent handling of a fault tree, a reduction strategy based on the Vesely-Fussell importance measures has been incorporated into the code PL-MODT [36]. Therefore, by inputting a CUTOFF value for the importance measures, those modules and components with V-F importance less than the CUTOFF value

are automatically eliminated from the tree. The fault tree resulting from this process still has all the features of the original tree from the quantification point of view. The selection of a proper value for the CUTOFF value is very important. In Chapter 3 of this thesis examples for a proper selection of this value is presented.

Restrictions to the PL-MODT code are exactly similar to those of PL-MOD. One of the PL-MODT features is its capability to calculate unavailabilities effectively and economically for sizable fault trees. Also, because of the four classes of components that are modeled by single continuous equations in the code PL-MODT, evaluation of unavailabilities is easier, due to the elimination of the necessary procedures to select the proper equation for each particular interval of the component operation. PL-MODT also calculates average unavailabilities over a period of time. The capability of using different time step sizes during different periods of the system operations enables the user to carefully study system behavior during high unavailability periods, such as during test and maintenance intervals. For a complete description of technical and mathematical concepts of the PL-MODT code the reader is referred to chapters 2, 3 and 7 of this thesis.

(c) PL-MODMC

The code PL-MODMC [12] is a time-dependent Monte-Carlo simulation code, based on the modularization technique. In this code, most of the restrictions contained in the PL-MOD and PL-MODT codes are removed. PL-MODMC is capable of performing Monte-Carlo simulations of complex fault trees with time-dependent components.

The calculations are based on average unavailabilities over the period of operations. The code PL-MODMC provides the following capabilities for quantification:

- (1) The ability to compute a point estimate for the unavailability value of the top event, and/or
- (2) The ability to propagate error or uncertainty bounds on several failure parameters, including failure rate, average downtime, average repair time, and average test duration.
- (3) The ability to simulate large fault trees.

PL-MODMC assumes that the log-normal distribution holds for all failure data and their average unavailabilities. PL-MODMC has all features of the PL-MOD Code, but the restriction of not allowing replicated components to appear in different replicated gate domains has been removed. Also, a preprocessor has been added to the code, so that inputs to PL-MODMC can be alphanumeric names and the preprocessor would change them automatically to the proper numeric values for use in the code. The input to the code has been changed and made very simple and convenient. A PL/I random number generator is designed for use in the PL-MODMC, which is based on the Central Limit Theorem to produce normal random numbers from uniform numbers that the computer generates. This random number generator has proven to be very fast and accurate. Shell's sorting method [22] is used to sort the top event unavailabilities. For a complete description of the code PL-MODMC and some examples the reader is referred to Chapters 4, 7 and 8 of this thesis.

A.6 Common Cause Analysis Codes

(a) BACFIRE, MOCUS-BACFIRE, MODCUT-BACFIRE

The BACFIRE Code [37,38] is very similar to COMCAN (for complete information see part (b) of this section). The input is almost the same. In addition to COMCAN, BACFIRE permits a component to appear in more than one physical location (domain). This is useful for piping and wiring which may pass through different domain barriers. Chapter 6 of this thesis discusses the method used in COMCAN and BACFIRE to identify cutsets that are common cause candidates.

Similar to COMCAN, BACFIRE requires cutsets as input to the code. For that reason the code MOCUS has been coupled with BACFIRE during this thesis research to provide an easy package similar to COMCAN-II (see part (b) of this section), so that the cutsets are to be found from the fault tree diagram. MOCUS-BACFIRE is capable of identifying common cause candidates of the top events and, if required of all other gates in a fault tree. For more information with regard to coupling MOCUS and BACFIRE codes the reader is referred to Chapter 6 and 7 of this thesis.

Since MODCUT is a modular cutset generator code, it has been used for generating cutsets for the BACFIRE Code for test purposes. This is done by the IBM conversational Monitor System (CMS), by providing a temporary output file which is automatically read by the BACFIRE code. MODCUT-BACFIRE is capable of analyzing the common cause problems where the fault tree is large and not manageable by MOCUS because of its restrictions. The disadvantage of using MODCUT-BACFIRE is that MODCUT is written in PL/I and BACFIRE is in FORTRAN. Thus, it may be operable

with other machines directly. For more information on MODCUT-BACFIRE the reader is referred to Chapters 6 and 7 of this thesis.

(b) COMCAN and COMCAN II

COMCAN and COMCAN II[39,40] are codes for common cause failure analysis. COMCAN is the first program to perform common cause failure analysis. COMCAN uses fault tree minimal cutsets as input for the analysis. However, for large fault trees, time and monetary considerations usually require that codes designed for obtaining fault tree minimal cutsets have their outputs restricted to minimal cutsets containing small number of events. Therefore, prohibitive amounts of computer time must be sent to evaluate all cutsets, and then input them into the COMCAN Code. COMCAN II has been designed to circumvent COMCAN's limitations of dependence upon minimal cutsets obtained by other codes. COMCAN II utilizes the fault tree cutset code FATRAM (that has been previously discussed in this Appendix) to generate cutsets of the fault trees. Except for the minimal cutsets, input to the COMCAN and COMCAN II codes is similar, and describes the common susceptibility data for each basic event. The output is a list of the minimal cutsets that are common cause candidates. The COMCAN and COMCAN II input allows three generic cause categories (mechanical or thermal, electrical or radiation, and chemical or miscellaneous) and a common link category. Under each category a number of causes (or links) may be specified, (e.g. under the mechanical or thermal cause category one could specify the generic causes -- impact, vibration, pressure, grit, stress and temperature). Optional inputs are the component manufacturer, location, domain definitions for each generic cause, the location of each component implied by the basic events, and

ranks of component susceptibility for each common cause. The output includes the ability to permit only common cause candidates with ranks greater than an order N, and to include similar type components as one of the criteria for common cause candidates. COMCAN II is written in FORTRAN IV for the CDC-CYBER-76 computer, and COMCAN is written in FORTRAN IV for the IBM 360/370 series.

(c) SETS

The SETS code which is previously described in this Appendix can also perform common cause analysis [41]. The analysis is conducted similarly to COMCAN by inputting generic cause susceptibilities for each basic event. A variable transformation incorporates the common cause susceptibilities into the Boolean equation for the top or any intermediate gates of the fault tree, and a few simple manipulations allow the user to display the cutsets which are the common cause candidates.

REFERENCES

- [1] U. S. Nuclear Regulatory Commission "Fault Tree Hand-Book", 1978
- [2] Van Slyke, W. J. and D. E. Griffing, "ALLCUTS, A Fast, Comprehensive Fault Tree Analysis Code", Atlantic Richfield Hanford Company, Richland, Washington, ARH-ST-112, July, 1975
- [3] Semanderes, S. N., "ELRAFT, A Computer Code for Efficient Logic Reduction Analysis of Fault Trees", IEEE Trans. on Nucl. Sci., Vol. NS-18, No. 1, p. 262, 1972
- [4] Rasmussen, D. M. and N. H. Marshall, "FATRAM, A Core Efficient Cutsets Algorithm", IEEE Trans. on Reliability, Vol. R-27, No. 4, p. 250, October, 1978
- [5] Willie, Randal R., "Computer Aided Fault Tree Analysis", U. C., Berkeley, ORC 78-14, August, 1978
- [6] Chatterjee, P., "Fault Tree Analysis: Reliability Theory And System Safety Analysis", U. C., Berkeley, AD/A-004 209, November, 1974
- [7] Velself, W. E., "Analysis of Fault Trees by Kinetic Tree Theory", Idaho Nuclear Corporation, Idaho Falls, Idaho, IN-1330, October, 1969
- [8] Fussell, J. B. and W. E. Vesely, "A New Methodology for Obtaining Cutsets for Fault Trees", Trans. ANS, 15, p. 262, 1972
- [9] Fussell, J. B., et. al., "MOCUS - A Computer Program to Obtain Minimal Sets from Fault Trees", Aerojet Nuclear Co., ANCR-1156 (August, 1974)
- [10] Olmos, J., "A Modular Approach to Fault Tree and Reliability Analysis", Ph.D. Thesis, MIT Dep. of Nucl. Eng., August, 1977
- [11] Olmos, J. and L. Wolf, "A Modular Representation and Analysis of Fault Trees", Nucl. Eng. and Design, Vol. 48, Nos. 203, pp. 531-561, August, 1978
- [12] Modarres, M. and L. Wolf, "PL-MODT and PL-MODMC Two Codes for Reliability and Availability Analysis of Complex Technical Systems using the Fault Tree Modularization Technique", Dept. of Nucl. Eng., MITNE-222, November, 1978
- [13] Vesely, W. E. and R. E. Narum, "PREP and KITT Computer Codes for the Automatic Evaluation of a Fault Tree", Idaho Nuclear Corporation, Idaho Falls, Idaho, IN-1349, 1970
- [14] Worrell, R. B., "Using the Set Equation Transformation System in Fault Tree Analysis", SAND 74-0240, Sandia Laboratories, Albuquerque, New Mexico, 1974

- [15] Wornel, R. B. and D. W. Stack, "A SETS User's Manual for the Fault Tree Analyst", NUREG/CR-0405, SAN 77-2051, November 1978
- [16] Vesely, W. E. and F. F. Goldberg, "FRANTIC - A Computer Code for Time Dependent Unavailability Analysis", U. S. Nuclear Regulatory Commission, NUREG-0193, October 1977
- [17] Vesely, W. E. and F. F. Goldberg, "Time Dependent Availability Analysis of Nuclear Safety Systems Using Randomly Sampled Failure Rates", U. S. Nuclear Regulatory Commission, NRC 4/4
- [18] Lambert, H. E., "Fault Trees for Decision Making in Systems Analysis", UCRL-51829 (October, 1975)
- [19] Lambert, H. B., "Measures of Importance of Events and Cutsets in Fault Trees", Reliability and Fault Tree Analysis SIAM (1975), pp. 72-100
- [20] Vesely, W. E., "Analysis of Fault Trees by Kinetic Tree Theory", Idaho Nuclear Corporation, Idaho Falls, Idaho, IN-1330, October, 1969
- [21] Lee, Y. T. and S. L. Salem, "Probability Intervals for the Reliability of Complex Systems Using Monte Carlo Simulation", UCLA-ENG-7758, December, 1977
- [22] Shell, D. L., "A High-Speed Sorting Procedure", Communications of the Association for Computing Machinery, Vol. 3, No. 3, pp. 134-166, July, 1956
- [23] U. S. Nuclear Regulatory Commission, "Reactor Safety Study", WASH-1400 (NUREG-75/014), Appendix II, pp. 64-76, October, 1975
- [24] Matthews, S. D., "MOCARS: A Monte Carlo Simulation Code for Determining the Distribution and Simulation Limits", TREE-1138, July, 1977
- [25] Rumble, E. T., F. L. Leverenz, Jr. and R. C. Erdman, "Generalized Fault Tree Analysis for Reactor Safety", Electric Power Research Institute, Palo Alto, California, EPRI 217-2-2, Interim Report, June, 197
- [26] Leverenz, F. L. and H. Kirch, "User's Guide for the WAM-BAM Computer Code", Science Application, Inc., EPRI 217-2-5, January, 1976
- [27] Erdman, R. C., F. L. Leverenz and H. Kirch, "WAMCUT, A Computer Code for Fault Tree Evaluation", Science Applications, Inc., EPRI NP-803, June, 1978
- [28] Platz, O. and J. V. Olsen, "FAUNET: A Program Package for Evaluation of Fault Trees and Network", RISØ Report No. 348, Copenhagen, Denmark, September, 1976

- [29] Platz, O. and J. V. Olsen, "FAUNET: A Program for Fault Tree and Network Calculations", Topical Meeting on Probabilistic Analysis of Nuclear Reactor Safety, Los Angeles, California, May 1975
- [30] Gateley, W. Y., D. W. Stoddard and R. L. Williams, "GO, A Computer Program for Reliability Analysis of Complex Systems", Kaman Science Corporation, Colorado Springs, Colorado, KN-67-704(R), April, 1968
- [31] Wood, D. E. and N. J. Becar, "A Comparison of Results from the GO Methodology and Fault Tree Analysis", Kaman Science Corporation, Colorado Springs, Colorado, K77-38U(R)
- [32] Koen, B. V. and A. Carnino, "Reliability Calculations with a List Processing Technique", IEEE Trans. on Reliability, Vol. 13-23, No. 1, April, 1974
- [33] Koen, B. V., A. Carnino, et. al., "The State of the Art of PATREC: A Computer Code for Evaluation of Reliability and Availability of Complex Systems", presented at the National Reliability Conference, Nottingham, England, September, 1977
- [34] Koen, B., A. Carnino, et. al., "A Computer Code for Fault Tree Calculations: PATREC", presented at Probabilistic Analysis of Nuclear Safety Topical Meeting, Los Angeles, May, 1978
- [35] Modarres, M. and L. Wolf, "PL-MODT: A Modular Fault Tree Analysis and Transient Evaluation Code", Transaction of the Am. Nucl. Society, 29, 510, June, 1978
- [36] Modarres, M. and L. Wolf, "Automatic Reduction of Complex Fault Trees by PL-MODT", Tran. of the Am. Nucl. Society, 30, 362, November, 1978
- [37] Cate, C. L. and J. B. Fussell, "BACFIRE - A Computer Code for Common Cause Failure Analysis", University of Tennessee, Knoxville, Tennessee, May, 1977
- [38] Wagner, D. P., "A Procedure for Qualitative Common Cause Failure Analysis of Complex Systems", Master's Thesis, University of Tennessee, August, 1972
- [39] Burdick, G. R., N. H. Marshall and J. R. Wilson, "COMCAN - A Computer Program for Common Cause Analysis", Aerojet Nuclear Company, ANCR-1314 May, 1976
- [40] Rasmuson, D. M., N. H. Marshall, J. R. Wilson and G. R. Burdick, "COMCANII - A Computer Program for Common Cause Failure Analysis", TREE-1289, September, 1978
- [41] Worrell, R. B. and D. W. Stack, "Common Cause Analysis Using SETS" SAND77-1832, Sandia Laboratories, Albuquerque, New Mexico, 1977

- [42] Birnbaum, Z. W., "On the Importance of Different Components in a Multicomponent System in Multivariate Analysis II," P. R. Krishnaiah (ed.), Academic Press, New York, 1969
- [43] Fussell, J. B., "How to Hand-Calculate System Reliability Characteristics", IEEE Transaction on Reliability, Vol. R-24, No. 3, August, 1975
- [44] Barlow, R. E. and F. Proschan, "Importance of System Components and Fault Tree Events", Stochastic Processes and Their Applications, 3 (1975), pp. 1-21

APPENDIX B

DESCRIPTION OF INPUT DATA FOR PL-MODT

The input data for the PL-MODT code is FORMAT free. The only requirement is that the data, when punched on the same card or typed on the same line, be separated by at least one blank space or a comma.

To more easily comprehend the information contained in this manual, some sample problems are presented at the end of this appendix, showing different inputs for PL-MODT.

B.1 Description of the Code PL-MODT

1. Card groups I through VIII describe the fault tree logic following these card groups. Any one of the following card group sets described in Parts 3 through 5 plus the card group in Part 2 could be used.
2. Card groups IX and X are control cards for the type of time-dependent analysis to be used.
3. Card groups X through XIII are devoted to the analysis of Class 1 components (time-independent PL-MOD case).
4. Card groups XIV through XXIII are devoted to the analysis of Class 2 components.
5. Card groups XIX through XXIII are devoted to the analysis of Class 3 components.

6. Card groups XXIV through XXVIII are devoted to the analysis of Class 4 components, or a combination of the three time-dependent classes of components.
7. Card group XXIX is the fault tree reduction option card.

INPUT DESCRIPTIONTitle CardCARD GROUP I: TITLE CARD

<u>No. of Variable</u>	<u>Variable</u>	<u>Entry</u>
1	TITLE	A set of characters enclosed by a single quotation mark

NOTE: Number of characters must be equal to or less than 71.

Options for the Types of Calculations

CARD GROUP II: RELIABILITY PARAMETER OPTION

<u>No. of Variable</u>	<u>Variable</u>	<u>Entry</u>
1	DEL	Number of items to be computed if DEL=1 only reliabilities or unavailabilities are calculated. If DEL = 2 reliabilities and importances are calculated

NOTE: When periodically tested or repairable (revealed fault) components are considered, it is recommended to set DEL = 1 because the importance measure built into the code seemingly loses its meaning under these conditions.

When automatic reduction of the tree is desired, the value of DEL must be equal to 2 (i.e., DEL = 2).

Fault Tree CharacteristicsCARD GROUP III: GATES

<u>No. of Variable</u>	<u>Variable</u>	<u>Entry</u>
1	GUM	Total number of fault tree gates

NOTE: GUM includes all AND, OR, and k-out-of-N gates but excludes replicated gates (modules). The replicated gates in the original tree will be considered as a replicated component or module.

CARD GROUP IV: REPLICATED MODULES

<u>No. of Variables</u>	<u>Variable</u>	<u>Entry</u>
1	RMOD	Total number of replicated modules; = 0 for no replication RMOD=0

NOTE: RMOD does not include replicated components. However,
replicated components may be represented in a replicated
module, i.e., component is replicated within this module.

CARD GROUP V: TREE STRUCTURE

<u>No. of Variable</u>	<u>Variable</u>	<u>Entry</u>
1	I	Gate number
2	AGIN(I)	Total number of gate inputs to gate number I
3	ALIL(I)	Total number of leafs which are input to gate number I
4	ALIR(I)	Total number of replicated leafs which are input to gate number I = 0 no replicated leaf

NOTE: I = 1, 2, ..., GUM

I includes all gates and associated modules, i.e., replicated
or non-replicated modules in the tree.

CARD GROUP VI: REPLICATED MODULES

<u>No. of Variable</u>	<u>Variable</u>	<u>Entry</u>
1	TRIM(I)	Name of replicated leaf associated with a module
2	TRIN(I)	Number of replicated gate

NOTE: $I = 1, 2, \dots, RMOD$

If $RMOD = 0$ on card group IV, then card group VI is to be skipped. If replicated gates exist then $TRIM(I) = A9BCD$, where

A : Total number of occurrences of the specific replicated module

9 : Represents the replication of gate or module

BCD : Number of replicated components associated with this module in the fault tree

CARD GROUP VII: REPLICATED LEAVES

<u>No. of Variable</u>	<u>Variable</u>	<u>Entry</u>
1	NOR	Total number of replicated leaf inputs = 0 for replicated leaf inputs

CARD GROUP VIII: GATE STRUCTURE

<u>No. of Variable</u>	<u>Variable</u>	<u>Entry</u>
1	NAME	Gate Number
2	VALUE	= 1 for AND gate = 2 for OR gate = KON for K-out-of-N gate
3	GIN	Total number of gate inputs to this gate
4	PIT(I) I = 1,2,...,GIN	I-th gate input For GIN = 0, then PIT(I) = 0
5	LIL	Total number of free leaf inputs
6	TIL(I) I = 1,2,...,LIL	I-th free leaf input For LIL = 0, then TIT(I) = 0
7	LIR	Total number of replicated leaf inputs
8	TIR(I) I = 1,2,...,LIR	I-th replicated leaf input For LIR = 0, then TIR(I) = 0

CARD GROUP VIII (continued)

NOTE: $LIR(I) = \overline{A0BCD}$

where A: total number of occurrences of this component

$\overline{BCD}$: number of replicated component

For dual replicated components:

$LIR(I) = \overline{A1BCD}$ when ON

$= \overline{A2BCD}$ when OFF

Numerical EvaluationsCARD GROUP IX: EVALUATION OPTION

<u>No. of Variable</u>	<u>Variable</u>	<u>Entry</u>
1	FOX	= 0 numerical evaluation is not desired
		= 1 numerical evaluation is desired

NOTE: If FOX = 0, then all of the following card groups can be deleted.

If the former version, PL-MOD is to be run, only card groups XI,
XII and XIII should be used after this card group.

If PL-MODT is to be run, card group X must be included and the
procedure thereafter.

CARD GROUP X: CALCULATION OPTION

<u>No. of Variable</u>	<u>Variable</u>	<u>Entry</u>
1	ESF	= 1 for steady state calculation = 2 for time-dependent calculation of non-repairable components = 3 for time-dependent calculation of repairable components = 4 for time-dependent calculation of periodically tested components, or the combination of periodically tested components and time-dependent non-repairable or repairable nontested components.

NOTE: For ESF = 1, only card groups XI, XII, XIII and XXIX are needed in what follows.

For ESF = 2, only card groups XIV, XV, XVI, XVII, XVIII and XXIX are needed.

For ESF = 3, only card groups XIX, XX, XXI, XXII, XXIII, and

CARD GROUP X (continued)

XXIX are needed.

For $ESF = 4$, only card groups XXIV, XXV, XXVI, XXVII, XXVIII and XXIX are needed.

If none of these options is desired, any number other than 1, 2, 3, 4 suffices, and all of the following card groups should be deleted.

CARD GROUP XI: LEAF INPUT

<u>No. of Variable</u>	<u>Variable</u>	<u>Entry</u>
1	FUN	Total number of free leaf inputs
2	DUN	Total number of replicated leaf inputs = 0, no replicated components or modules

CARD GROUP XII: COMPONENT RELIABILITY

<u>No. of Variable</u>	<u>Variable</u>	<u>Entry</u>
1	I I = 1,2,...,FUN	Number of free component
2	STATE(1,I)	Probability of occurrence of the I-th free input

CARD GROUP XIII: REPLICATED COMPONENT RELIABILITY

<u>No. of Variable</u>	<u>Variable</u>	<u>Entry</u>
1	I	Number of replicated component
2	STATD(1,I)	Probability associated with the I-th replicated component = 0 if I-th component is associated with a replicated module

CARD GROUP XIV: TIME STEP CHANGES

<u>No. of Variable</u>	<u>Variable</u>	<u>Entry</u>
1	MOH	Number of regions where time step size changes = 1 for no changes in time step size

NOTE: This card group must be supplied for ESF = 2 (see card group X).

CARD GROUP XV: TIME STEP INPUT

<u>No. of Variable</u>	<u>Variable</u>	<u>Entry</u>
1	DEL9T(1,I)	Time step associated with the I-th region
2	AUN(1,I) I = 1,2,...,MOH	Time interval for which time step size is applied

NOTE: This card group must be supplied for ESF = 2 (see card group X)

CARD GROUP XVI: LEAF INPUT

<u>No. of Variable</u>	<u>Variable</u>	<u>Entry</u>
1	FUN	Total number of free leaf inputs
2	DUN	Total number of replicated leaf inputs = 0 no replicated components or modules

NOTE: This card group must be supplied for ESF = 2 (see card group X).

CARD GROUP XVII: FREE COMPONENT FAILURE RATE

<u>No. of Variable</u>	<u>Variable</u>	<u>Entry</u>
1	I	Number of free component
2	STATT(1,I)	Failure rate $\lambda(\text{hr}^{-1})$, associated with the I-th free component

NOTE: This card group must be supplied for ESF = 2 (see card group X)

CARD GROUP XVIII: REPLICATED COMPONENT FAILURE RATE

<u>No. of Variable</u>	<u>Variable</u>	<u>Entry</u>
1	I	Number of replicated component
2	STATS(1,I)	Failure rate $\lambda(\text{hr}^{-1})$ associated with the I-th replicated component = 0 if I is a replicated component associated with a replicated module

NOTE: This card group must be supplied for ESF = 2 (see card group X).

CARD GROUP XIX: TIME STEP CHANGES

<u>No. of Variable</u>	<u>Variable</u>	<u>Entry</u>
1	MOH	Number of regions where time step size changes. = 1 for no changes in time step size

NOTE: This card group must be supplied for ESF = 3 (see card group X).

CARD GROUP XX: TIME STEP INPUT

<u>No. of Variable</u>	<u>Variable</u>	<u>Entry</u>
1	DEL9T(1,I)	Time step associated with the Ith region
2	AUN(1,I) I = 1,2,...,MOH	Time step interval for which time step size is applied

NOTE: This card group must be supplied for ESF = 3
(see card group X)

CARD GROUP XXI: LEAF INPUT

<u>No. of Variable</u>	<u>Variable</u>	<u>Entry</u>
1	FUN	Total number of free leaf inputs
2	DUN	Total number of replicated leaf inputs = 0 no replicated components or modules

NOTE: This card group must be supplied for ESF = 3 (see card group X)

CARD GROUP XXII: FREE COMPONENT FAILURE AND REPAIR RATES

<u>No. of Variable</u>	<u>Variable</u>	<u>Entry</u>
1	I	Number of free components
2	STATT(1,I)	Failure rate $\lambda(\text{hr}^{-1})$ associated with the Ith free component
3	STATTE(1,I)	Repair rate $\mu(\text{hr}^{-1})$ associated with the Ith free component

NOTE: This card group must be supplied for ESF = 3 (see card group X)

CARD GROUP XXIII: REPLICATED COMPONENT FAILURE AND REPAIR RATES

<u>No. of Variable</u>	<u>Variable</u>	<u>Entry</u>
1	I	Number of replicated component
2	STATS(1,I)	Failure rate $\lambda(\text{hr}^{-1})$ associated with the I-th replicated component = 0 if I is a replicated component associated with a module
3	STATED(1,I)	Repair rate $\mu(\text{hr}^{-1})$ associated with the I-th replicated component

NOTE: If STATS(1,I) = 0 (i.e., the component I is associated with a replicated module), then STATED(1,I) = 1. This card group must be supplied for ESF = 3 (see card group X).

CARD GROUP XXIV: TIME STEP CHANGES

<u>No. of Variable</u>	<u>Variable</u>	<u>Entry</u>
1	MOH	Number of regions where time step size changes = 1 for no changes in time step size

NOTE: This card group must be supplied for ESF = 4 (see card group X)

CARD GROUP XXV TIME STEP INPUT

<u>No. of Variable</u>	<u>Variable</u>	<u>Entry</u>
1	DEL9T(1,I)	Time step associated with the I-th region
2	AUN(1,I) I = 1,2,...MOH	Time interval for which time step size is applied

NOTE: This card group must be supplied for ESF = 4 (see card group X)

CARD GROUP XXVI: LEAF INPUT

<u>No. of Variable</u>	<u>Variable</u>	<u>Entry</u>
1	FUN	Total number of free leaf inputs
2	DUN	Total number of replicated leaf inputs = 0 no replicated components or modules

NOTE: This card group must be supplied for ESF = 4 (see card group X)

CARD GROUP XXVII: FREE COMPONENT TEST DATA

<u>No. of Variables</u>	<u>Variables</u>	<u>Entry</u>
1	I	Number of free components
2	START(1,I)	Failure rate $\lambda(\text{hr}^{-1})$ associated with I-th free component
3	ETTA(1,I)	Time between inspections (hrs)
4	TTETA(1,I)	Inspection time (hrs)
5	TAVV(1,I)	Repair time (hrs)
6	FIRTM(1,I)	Time of first inspection (hrs)
7	QUZR(1,I)	Override probability

NOTE: If the component is not inspected, then $ETTA(1,I) = 0$,
 $TTETA(1,I) = 0$, $TAVV(1,I) = 0$, $FIRTM(1,I) = 0$ and $QUZR(1,I) = 0$.
 In this case the component will be considered as being a
nonrepairable one. If the component is repairable but not
 tested (revealed fault), then $ETTA(1,I) = 0$, $TTETA(1,I) = \text{repair}$
 $\text{rate}(\text{hr}^{-1})$, $TAVV(1,I) = 0$, $FIRTM(1,I) = 0$ and $QUZR(1,I) = 0$

CARD GROUP XXVIII: REPLICATED COMPONENT TEST DATA

<u>No. of Variable</u>	<u>Variable</u>	<u>Entry</u>
1	I	Number of replicated component
2	STATS(1,I)	Failure rate (hr^{-1}) associated with this component
3	ETTAD(1,I)	Time between inspections (hrs)
4	TTETAD	Inspection time (hrs)
5	TAVVD(1,I)	Repair time (hrs)
6	FIRTMD(1,I)	Time of first inspection (hrs)
7	QUZRD(1,I)	Override probability

NOTE: If the component is not inspected then $\text{ETTAD}(1,I) = 0$, $\text{TTETAD}(1,I) = 0$, $\text{TAVVD}(1,I) = 0$, $\text{FIRTMD}(1,I) = 0$, and $\text{QUZRD}(1,I) = 0$.

If the component is associated with a replicated module then $\text{STATS}(1,I) = 0$, $\text{ETTAD}(1,I) = 0$, $\text{TTETAD}(1,I) = 0$, $\text{TAVVD}(1,I) = 0$, $\text{FIRTMD}(1,I) = 0$, and $\text{QUZRD}(1,I) = 0$. In this case the component is considered as being a nonrepairable one. If the component is repairable but not tested (revealed fault), then $\text{ETTAD}(1,I) = 0$, $\text{TTETAD}(1,I) = \text{repair rate } (\text{hr}^{-1})$, $\text{TTETAD}(1,I) = 0$, $\text{TAVVD}(1,I) = 0$, $\text{FIRTMD}(1,I) = 0$, and $\text{QUZRD}(1,I) = 0$.

CARD GROUP XXIX: REDUCTION OPTION CARD

<u>No. of Variable</u>	<u>Variable</u>	<u>Entry</u>
1	CUT-OFF	The cut-off value. Any component having an importance less than this value will be cancelled from the fault tree.

NOTE: If DEL = 1, then this card group is not needed since no importance calculation is performed.

If DEL = 2 and CUT-OFF = 0, then no reduction process will be performed. Only the importance of modules and components in the fault tree will be calculated.

If DEL = 2 and CUT-OFF > 0, then the reduction procedure will be followed.

B.2 Input Data Example

In this section the input to PL-MODT for a time-dependent analysis of the fault tree shown in Figure (8.4.1) of this thesis is discussed. The data set needed to describe the logic of the fault tree is listed in Table (B.1). Following Table (B.1) any one of the data sets shown Tables (B.2), (B.3), (B.4) or (B.5) may be used.

Table (B.2) is used if all of the components in the tree consist of only Class 1 components (i.e. the original PL-MOD code). Table (B.3) is used for calculating unavailabilities of the fault tree, if all of the components are of Class 2. Table B.4 is used if time-dependent calculations are performed for only Class 3 components. Finally, Table (B.5) can be used when the fault tree is composed of only Class 4 components. The particular set of data shown in Table (B.5) represents a combination of four classes of components in the fault tree shown in Figure (8.4.1)

TABLE B.1

GENERAL DATA SET REPRESENTING THE LOGIC OF THE TREE

'SAMPLE PROBLEM'

2	26	1		
1	2	1	0	
2	1	0	1	
3	0	2	0	
4	1	1	1	
5	1	2	0	
6	2	0	0	
7	1	1	0	
8	1	0	1	
9	0	2	1	
10	0	2	0	
11	2	2	0	
12	1	0	1	
13	0	1	1	
14	2	0	0	
15	1	0	1	
16	0	3	0	
17	1	0	1	
18	2	0	0	
19	1	0	1	
20	1	1	1	
21	0	2	0	
22	1	1	1	
23	2	0	0	

Table B.1 (continued)

24 0 1 1

25 1 0 1

26 0 2 1

29002 1

7

1 1 2 2 4 1 14 0 0

2 2 1 3 0 0 1 22006

3 2 0 0 2 16 17 0 0

4 2 1 5 1 15 1 21006

5 203 1 6 2 18 19 0 0

6 2 7 8 0 0 0 0

7 1 1 9 1 22 0 0

8 1 1 10 0 0 1 20007

9 2 0 0 2 20 21 1 20007

10 2 0 0 2 23 24 0 0

11 1 2 12 14 2 1 2 0 0

12 2 1 13 0 0 1 21001

13 1 0 0 1 6 1 29002

14 2 2 15 17 0 0 0 0

15 1 1 16 0 0 1 22001

16 2 0 0 3 3 4 5 0 0

17 1 1 18 0 0 1 29002

18 1 2 19 22 0 0 0 0

19 2 1 20 0 0 1 21004

Table B.1 (continued)

20	1	1	21	1	9	1	20003
21	2	0	0	2	8	11	0 0
22	2	1	23	1	7	1	20005
23	1	2	24	25	0	0	0 0
24	2	0	0	1	10	1	20003
25	2	1	26	0	0	1	22004
26	1	0	0	2	12	13	1 20005

1

TABLE B.2

DATA SET FOLLOWING TABLE A.1 FOR ONLY
CLASS 1 COMPONENTS (PL-MOD CASE)

1	
24	7
1	1.0E-01
2	1.0E-01
3	1.0E-02
4	1.0E-02
5	1.0E-02
6	1.0E-01
7	1.0E-03
8	0.5E-03
9	0.5E-03
10	2.0E-05
11	2.0E-04
12	2.0E-04
13	6.1E-04
14	6.1E-04
15	8.1E-04
16	.6E-02
17	.6E-02
18	3.0E-01
19	3.0E-01
20	2.5E-04

TABLE B.2 (continued)

21	2.5E-04
22	2.5E-04
23	2.5E-04
24	1.0E-02
1	1.0E-01
2	0
3	1.0E-02
4	1.5E-03
5	1.5E-03
6	1.2E-01
7	1.2E-01
	1.0E-03

TABLE B.3

DATA SET FOLLOWING TABLE A.1 FOR ONLY CLASS 2 COMPONENTS

2	
3	
18	180
24	240
36	720
24	7
1	1.0E-01
2	1.0E-01
3	1.0E-02
4	1.0E-02
5	1.0E-02
6	1.0E-01
7	1.0E-03
8	0.5E-03
9	0.5E-03
10	2.0E-05
11	2.0E-04
12	2.0E-04
13	6.1E-04
14	6.1E-04
15	8.1E-04
16	.6E-02
17	.6E-02

TABLE B.3 (continued)

18	3.0E-01
19	3.0E-01
20	2.5E-04
21	2.5E-04
22	2.5E-04
23	2.5E-04
24	1.0E-02
1	1.0E-01
2	0
3	1.0E-02
4	1.5E-03
5	1.5E-03
6	1.2E-01
7	1.2E-01
	1.0E-03

•

TABLE B.4

DATA SET FOLLOWING TABLE A.1 FOR ONLY CLASS 3 COMPONENTS

3		
2		
2	24	
24	720	
24	7	
1	1.0E-01	1.0E-01
2	1.0E-01	1.0E-01
3	1.0E-02	1.0E-01
4	1.0E-02	1.0E-01
5	1.0E-02	1.0E-01
6	1.0E-01	1.2E-01
7	1.0E-03	1.2E-01
8	0.5E-03	1.2E-01
9	0.5E-03	1.2E-01
10	2.0E-05	1.0E-02
11	2.0E-04	1.0E-02
12	2.0E-04	1.0E-02
13	6.1E-04	1.0E-02
14	6.1E-04	1.0E-02
15	8.1E-04	1.0E-01
16	.6E-02	1.0E-01
17	.6E-02	1.0E-01
18	3.0E-01	5.0E-01
19	3.0E-01	1.0E-01

TABLE B.4 (continued)

20	2.5E-04	1.0E-01
21	2.5E-04	1.0E-01
22	2.5E-04	1.0E-02
23	2.5E-04	1.0E-02
24	1.0E-02	1.0E-02
1	1.0E-01	1.0E-1
2	0	0
3	1.0E-2	1.0E-01
4	1.5E-3	1.0E-01
5	1.5E-3	1.0E-01
6	1.2E-1	1.0E-01
7	1.2E-1	1.0E-01
	1.0E-03	

TABLE B.5

DATA SET FOLLOWING TABLE A.1 FOR ONLY CLASS 4 COMPONENTS
OR COMBINATION OF ALL TIME DEPENDENT CLASSES OF COMPONENTS

4
5
12 240
1 5
24 360
2 8
26 1440
24 7
1 1.0E-01 360 2 15 360 1
2 1.0E-01 400 3 18 700 1
3 1.0E-02 720 1.5 19 1440 1
4 1.0E-02 150 1 10 150 1
5 1.0E-02 1200 5 25 1200 .5
6 1.0E-01 600 1 20 600 1
7 1.0E-03 720 1.5 19 800 0.5
8 0.5E-03 120 1 12 120 1
9 0.5E-03 450 3 12.5 600 1
10 2.0E-05 720 1 22 720 1
11 2.0E-04 300 2.5 20 300 0.1
12 2.0E-04 0 1.0E-01 0 0 0
13 6.1E-04 0 0 0 0 0
14 6.1E-04 0 1.5E-01 0 0 0
15 8.0E-04 720 1 12 720 1
16 .6E-02 400 2 15 400 1

TABLE B.5 (continued)

17	.6E-02	0	0	0	0	0
18	3.0E-01	120	1	12	120	1
19	3.0E-01	0	0	0	0	0
20	2.5E-04	840	2.6	20	1440	0.5
21	2.5E-04	140	1.5	19	280	1
22	2.5E-04	720	1.5	12	840	1
23	2.5E-04	0	1.0E-02	0	0	0
24	1.0E-02	720	1.5	19	1440	1
1	1.0E-01	0	0	0	0	0
2	0	0	0	0	0	0
3	1.0E-02	450	2	16	720	1
4	1.5E-03	0	0	0	0	0
5	1.5E-03	1440	3	28	1440	1
6	1.2E-01	720	1.5	19	720	1
7	1.2E-01	440	1.5	20	440	1
	1.0E-03					

APPENDIX C

DESCRIPTION OF THE INPUT FOR PL-MODMC

C.1 Input Data

The input to the PL-MODMC code is FORMAT free. The only requirement is that the data, when punched on one card, be separated by at least one blank space or a comma. Four input groups are required in PL-MODMC.

Group 1 - Title and parameter default cards

Group 2 - Fault tree logic cards

Group 3 - Monte-Carlo options cards

Group 4 - Component failure data cards

Group 1TITLE CARD

<u>No. of Variable</u>	<u>Variable</u>	<u>Entry</u>
1	TITLE	A set of characters to be entered for the identification of the run

NOTE: Number of characters must be less than or equal to 71.

PARAMETER DEFAULT CARD

<u>No. of Variable</u>	<u>Variable</u>	<u>Entry</u>
1	EGATE	Estimated number of gates. The default is 100 for EGATE = 0
2	ECOMP	Estimated number of components. The default is 200 for ECOMP = 0
3	EGINO	Estimated maximum number of gates input to a gate. The default is 5 for EGINO = 0
4	ELILO	Estimated maximum number of components input to a gate. The default is 10 for ELILO = 0
5	DEBUG	If DEBUG = 1, then all of the information about the modules, number of replications, list of proper and free modules, and modular cutsets will be printed along with the Monte-Carlo output. Put DEBUG = 0 if no information except Monte-Carlo simulation and logical relations of the fault tree is required.

NOTE: Information printed under DEBUG = 1 is based on the numerical values that are assigned to input alphanumeric values.

Group 2FAULT TREE LOGIC CARD (Input one for each fault tree gate)

<u>No. of Variable</u>	<u>Variable</u>	<u>Entry</u>
1	GNAME	Name of the gate for which inputs will be given. A set of alphanumeric characters enclosed by single quotation marks
2	GTYPE	Gate GNAME's type. For AND gates, GTYPE should be input as 'AND' or 'A'; for OR gates, 'OR' or 'O'; for K-out-of-N gates, 'K/N'
3	NODEIN.GID	Number of gates input to the gate GNAME
4	INGATE(IG)	Name(s) of the gates input to GNAME. IG = 1, ... NODEIN.GID. Each input gate name should be alphanumeric values enclosed by single quotation marks
5	NODEIN.LILI	Number of components input to the gate GNAME

FAULT TREE LOGIC CARD (continued)

<u>No. of Variable</u>	<u>Variable</u>	<u>Entry</u>
6	INCOMP(IC)	Name(s) of the components input to the gate GNAME. IC = 1, ... NODEIN.LILI. Each input component name should be enclosed by single quotation marks. If the component is an OFF event, then put a ($\neg$) character in front of its alphanumeric name (e.g., ' $\neg$ MOV 12').

NOTES: (1) All alphanumeric names should have 12 or less characters.

(2) After the last gate's card, the user should input an 'END' card.

Group 3CONFIDENCE LEVEL DATA

<u>No. of Variable</u>	<u>Variable</u>	<u>Entry</u>
1	MONT	For Monte-Carlo analysis MONT = 1. For an average TOP event unavailability calculation only, set MONT = 0
2	ROM	Total number of confidence levels to be used

CONFIDENCE LEVEL DATA (continued)

<u>No. of Variable</u>	<u>Variable</u>	<u>Entry</u>
3	CONPNT(I)	Confidence level (e.g., CONPNT(I) = 40 means that the I-th confidence level is 40%). Not needed if ROM = 0.

NOTE: For ROM = 0 the following default confidence intervals are used:

CONPNT(1) = 0.5	CONPNT(2) = 1	CONPNT(3) = 2.5
CONPNT(4) = 5	CONPNT(5) = 10	CONPNT(6) = 15
CONPNT(7) = 20	CONPNT(8) = 25	CONPNT(9) = 30
CONPTN(10) = 40	CONPNT(11) = 50	CONPNT(12) = 60
CONPNT(13) = 70	CONPNT(14) = 75	CONPNT(15) = 80
CONPNT(16) = 85	CONPTN(17) = 90	CONPNT(18) = 95
CONPNT(19) = 97.5	CONPNT(20) = 99	CONPNT(21) = 99.5

In the Monte-Carlo calculations, the accuracy of the probabilities of the confidence level is equal to the lowest confidence level (e.g., for ROM = 0, with the lowest confidence level CONPNT(1) = 0.5, then accuracy = 0.5%).

SIMULATION CONTROL CARD

<u>No. of Variable</u>	<u>Variable</u>	<u>Entry</u>
1	IPP	An arbitrary number to start the random number generator procedure
2	NRAND	Total number of trials to be used in the simulation. Default value is 2000 for NRAND = 0
3	NTERM	Number of terms to be used in the Central Limit Theorem Approximation. Default value is 12 for NTERM = 0

Group 4FAILURE DATA CARD (Input one for each component)

<u>No. of Variable</u>	<u>Variable</u>	<u>Entry</u>
1	AAA	Name of the component. It should be a 12-character name and be enclosed by single quotation marks
2	OPTION	Class of the component, OPTION = 1 represents constant unavailability component (class one component). OPTION = 2 represents non-repairable component (class two component). OPTION = 3 represents repairable monitored component

FAILURE DATA CARD (continued)

<u>No. of Variable</u>	<u>Variable</u>	<u>Entry</u>
2	OPTION	(class three component). OPTION = 4 represents periodically tested component (class four component).

If OPTION = 1, then use the following variables

3	STATE	Unavailability per demand
4	FEN2	90% error factor associated with STATE variable. Ignored in MONT = 0

If OPTION = 2, then use the following variables

3	MED1	Failure rate in (hr^{-1})
4	FEN2	90% error factor associated with MED1 variable. Ignore if MONT = 0
4	MED2	Mission length (hr)

If OPTION = 3, then use the following variables

3	MED1	Failure rate in (hr^{-1})
4	FAC1	90% error factor associated with MED1 variable. Ignore if MONT = 0
5	MED2	Average down time (hr)

FAILURE DATA CARD (continued)

<u>No. of Variable</u>	<u>Variable</u>	<u>Entry</u>
6	FAC2	90% error factor associated with MED2. Ignore if MONT = 0

If OPTION = 4, then use the following variables

3	MED1	Failure rate (hr^{-1})
4	FAC1	90% error factor associated with MED1. Ignore if MONT = 0
5	MED2	Average repair time (hr)
6	FAC2	90% error factor associated with MED2. Ignore if MONT = 0
7	MED3	Average test duration (hr)
8	FAC3	90% error factor associated with MED3. Ignore if MONT = 0
9	MED4	Test interfal (hr)

NOTE: Repeat this failure data card for each component in the fault tree. For ON-OFF components, only unavailability for the ON components should be specified. Consider an ON component as a class one component.

C.2 Input Data Example

Input data for a Monte-Carlo analysis of the fault tree shown in Figure (8.3.1) are listed in Table (C.1). This fault tree consists of components of all four classes. All failure parameters have been assigned uncertainty values. The output from PL-MODMC for the set of input data is shown in Table (8.3.2).

TABLE C.1

Inputs to Fault Tree Presented in Figure 8.3.1

MONT-CARLO ANALYSIS BY THE PL-MODMC CODE

```

0      0      0      0      0      0

'TOP'  'A'  2  'G2'  'G3'  0
'G2'   'A'  1  'G4'  1  'C1'
'G3'   'O'  1  'G'   1  'C3'
'G4'   'O'  1  'G'   1  'C2'
'G5'   'O'  1  'G'   1  'C4'
'G6'   'A'  1  'G'   1  'C5'
'G7'   'A'  1  'G'   1  'C3'
'G8'   ' '  0  3  'C5'  'C6'  'C7'
'END

1 0

1331  1000  12

'C1'  2  2.0E-04  3  700
'C2'  1  1.3E-04  3
'C3'  3  1.47E-04  3  21  7
'C4'  1  2.0E-04  3
'C5'  4  2.0E-05  3  30  5  2.5  5  720
'C6'  2  2.0E-05  3  700
'C7'  1  1.0E-05  10  20  3  1.5  3  1440

```

APPENDIX D

DESCRIPTION OF INPUT TO MODCUT

D.1 Input Data

Input data to the MODCUT code are format-free. The only requirement is that the data, when punched on the same card or typed on the same line, be separated by at least one blank space or a comma.

Three input groups, 1 through 3, are described below.

Group 1 - Title and Parameter Default Cards

Group 2 - Fault Tree Logic Cards

Group 3 - Cutset Option Cards

Group 1TITLE CARD

<u>No. of Variable</u>	<u>Variable</u>	<u>Entry</u>
1	TITLE	A set of characters to be entered as the name of the run*

PARAMETER DEFAULT CARD

<u>No. of Variable</u>	<u>Variable</u>	<u>Entry</u>
1	EGATE	Estimated number of gates. The default is 100 for EGATE = 0
2	ECOMP	Estimated number of components. The default is 200 for ECOMP = 0

*Number of characters must be less than or equal to 71

PARAMETER DEFAULT CARD (continued)

<u>No. of Variable</u>	<u>Variable</u>	<u>Entry</u>
3	EGINO	Estimated maximum number of gates input to a gate. The default is 5 for EGINO = 0
4	ELILO	Estimated maximum number of components input to a gate. The default is 10 for ELILO = 0
5	CUTNO	If CUTNO = 1, then only the number of non-minimal cutsets will be calculated (only for test runs). If minimal cutsets are to be calculated, then CUTNO should be set to zero.
6	DEBUG	If DEBUG = 1, then all of the information about the modules, number of replications, list of proper and free modules, modular cutsets, and some other information will be printed along with the generated cutsets. Put DEBUG = 0 if no information except cutsets and logical relations of the tree are required.

- NOTES: (1) It is recommended that for very large trees, prior to cutset calculations a test run by putting CUTNO = 1 be performed in order to calculate the number of non-minimal cutsets that MODCUT will treat in cutset generation procedure.
- (2) Most of the information printed under DEBUG = 1 is based on the numerical values that are assigned instead of the input alphanumeric values.

GROUP 2FAULT TREE LOGIC CARD (Input one for each fault tree gate)

<u>No. of Variable</u>	<u>Variable</u>	<u>Entry</u>
1	GNAME	Name of the gate for which the inputs will be given. A set of alphanumeric characters enclosed by single quotation marks
2	GTYPE	Gate GNAME's type. For AND gates, GTYPE should be input as 'AND' or 'A'; for OR gates, 'OR' or 'O'; for K-out-of-N gates, 'K/N'
3	NODEIN.GID	Number of gates input to the gate GNAME
4	INGATE(IG)	Name(s) of the gates input to GNAME. IG = 1, ...NODEIN.GID. Each input gate name should be alphanumeric values enclosed by single quotation marks
5	NODEIN.LILI	Number of components input to the gate GNAME
6	INCOMP(IC)	Name(s) of the components input to the gate GNAME. IC = 1, ... NODEIN.LILI. Each input component name should be enclosed by single quotation marks. If the component is an OFF event, then put a ($\neg$) character in front of its alphanumeric name (e.g. ' $\neg$ MOVR')

NOTES: (1) All alphanumeric names should have 12 or less characters.
 (2) After the last gate's card, the user should input an 'END' card.

Group 3SIZE CARD

<u>No. of Variable</u>	<u>Variable</u>	<u>Entry</u>
1	SIZE	Maximum size of the cutsets to be generated for the TOP event.
2	SIZE1	Maximum size of the cutsets to be generated for all of the replicated modules (gates) in the tree. If no replicated modules, then put any arbitrary number.

Notes: (1) If SIZE is greater than SIZE1, make sure that no cutset of order between SIZE and SIZE1 will be lost

(2) For some complex trees there will be a high CPU time gain by putting SIZE1 smaller than SIZE

CUTSET REQUEST CARD

<u>No. of Variable</u>	<u>Variable</u>	<u>Entry</u>
1	NPRN	Number of gates for which the cutsets are to be generated
2	CPRINT	Name of the gates for which the cutsets are to be generated. Name(s) of these gates should be enclosed by single quotation marks.

NOTE: Cutsets of those gates that will be changed to nested or coalesced modules during the modularization process may not represent the exact cutsets of the gate. However, the correct cutsets are always generated for free modules, replicated gates, and the TOP event.

D.2 Input Data Example

Input data for generating the cutsets of the fault tree shown in Figure (8.4.1) are given in Table (D.1). This fault tree has been described in Chapter 8 of this thesis. This Fault Tree is composed of several types of components and gates. The listing from MODCUT for the Input Data shown in Table (D.1) is given in Table (8.4.4).

Table D.1

MODCUT Input for the Fault Tree Shown in Fig. 8.4.1

```

TEST RUN
0 0 0 0 0 0
'G1'  'AND'  2  'G2'  'G4'  1  'C14'
'G2'  'OR'   1  'G3'  1  '¬R6'
'G3'  'OR'   0  2  'C16'  'C17'
'G4'  'OR'   1  'G5'  2  'C15'  'R6'
'G5'  '2/3'  1  'G6'  2  'C18'  'C19'
'G6'  'OR'   2  'G7'  'G8'  0
'G7'  'AND'   1  'G9'  1  'C22'
'G8'  'AND'   1  'G10'  1  'R7'
'G9'  'OR'    0  3  'C20'  'C21'  'R7'
'G10' 'OR'    0  2  'C23'  'C24'
'TOP' 'AND'   2  'G12'  'G14'  2  'C1'  'C2'
'G12' 'CR'    1  'G13'  1  'R1'
'G13' 'AND'   1  'G1'  1  'C6'
'G14' 'OR'    2  'G15'  'G17'  0
'G15' 'AND'   1  'G16'  1  '¬R1'
'G16' 'CR'    0  3  'C3'  'C4'  'C5'
'G17' 'AND'   2  'G18'  'G1'  0
'G18' 'AND'   2  'G19'  'G22'  0
'G19' 'OR'    1  'G20'  1  'R4'
'G20' 'AND'   1  'G21'  2  'C9'  'R3'
'G21' 'CR'    0  2  'C8'  'C11'
'G22' 'CR'    1  'G23'  2  'C7'  'R5'
'G23' 'AND'   2  'G24'  'G25'  0
'G24' 'CR'    0  2  'C10'  'R3'
'G25' 'OR'    1  'G26'  1  '¬F4'
'G26' 'AND'   0  3  'C12'  'C13'  'R5'
'END'
9 4
2 'G1' 'TOP'

```

APPENDIX E

DESCRIPTION OF INPUT TO MOCUS-BACFIRE PACKAGE

The MOCUS-BACFIRE Package has been obtained by coupling the MOCUS[1] and BACFIRE [2] codes. The method by which the two codes are coupled together is discussed in Chapters 6 and 7 of this thesis. In this appendix the input to MOCUS-BACFIRE is discussed.

Input to MOCUS-BACFIRE is exactly similar to input of the MOCUS and the BACFIRE codes. No changes in the input format and the order in which data are input have been made. The input to the BACFIRE code is read in first by the package. However, instead of group 5 input data to BACFIRE which are the minimal cutsets, input data of the fault tree to MOCUS must be inserted. The input to BACFIRE is discussed in Reference [1] and the input to MOCUS is discussed in Reference [2]. Essentially, the input to BACFIRE as discussed in [1], consists of the following five data sets:

1. Event Definition
2. Event Location
3. Cause of Secondary Failure Library
4. Manufacturer Library
5. Hardware Minimal Cutset

Obviously, data set number 5 no longer exists in the MOCUS-BACFIRE package. Therefore, after the data set number 4, the input to MOCUS must be inserted. Input to MOCUS consists of the following data sets:

- (1) Control Information
- (2) Fault Tree Description
- (3) Failure and Repair Data for the Components
- (4) Gates for which Minimal Cutsets are to be found

Obviously, data set 3 of the MOCUS code is not necessary for the common-cause analysis and can be omitted from the input data set. It should be noted, however, that group (3) data set in the MOCUS code is optional, and has been used when the code is replaced by the PREP Code to create the MOCUS-KITT package.

The Job Control Language (JCL) Cards for the MOCUS-BACFIRE package are similar to those used for MOCUS and BACFIRE individually. An example of a set of input data and JCL cards to MOCUS-BACFIRE package is shown in Table (E.1) and its output list is given in Table (E.2). It should be added that for the MOCUS-BACFIRE package the H-compiler must be used. The H-compiler has additional features comparing to the G-compiler that are significantly applied in the MOCUS Code.

Table (E.1) Input Example to the MOCUS-BACFIRE Package

```

//GO.FT10F001 DD DSN=88FL1,UNIT=SYSDA,DISP=(NEW,PASS),SPACE=(TRK,5),
// DCB=(RECFM=VB,LRECL=84,BLKSIZE=1634)
//GO.FT10F002 DD DSN=88FL2,UNIT=SYSDA,DISP=(NEW,PASS),SPACE=(TRK,5),
// DCB=(RECFM=VB,LRECL=84,BLKSIZE=1684)
//GO.FT10F003 DD DSN=88FL3,UNIT=SYSDA,DISP=(NEW,PASS),SPACE=(TRK,5),
// DCB=(RECFM=VB,LRECL=84,BLKSIZE=1684)
//GO.FT10F004 DD DSN=88FL4,UNIT=SYSDA,DISP=(NEW,PASS),SPACE=(TRK,5),
// DCB=(RECFM=VB,LRECL=84,BLKSIZE=1684)
//GO.FT14F001 DD UNIT=SYSDA,DISP=(NEW,DELETE),SPACE=(TRK,20),
// DCB=(RECFM=VBS,BLKSIZE=1600)
//GO.FT15F001 DD UNIT=SYSDA,DISP=(NEW,DELETE),SPACE=(TRK,20),
// DCB=(RECFM=VBS,BLKSIZE=1600)
//GO.FT16F001 DD UNIT=SYSDA,DISP=(NEW,DELETE),SPACE=(TRK,20),
// DCB=(RECFM=VBS,BLKSIZE=1600)
//GO.FT08F001 DD DUMMY
//GO.FT09F001 DD DUMMY
//GO.FT11F001 DD DSN=88WAG1,DCB=(RECFM=FB,LRECL=20,BLKSIZE=3120),
// UNIT=SYSDA,SPACE=(TRK,(100,100),ELSE),DISP=NEW
//GO.FT12F001 DD DSN=88WAG2,DCB=(RECFM=FB,LRECL=20,BLKSIZE=3120),
// UNIT=SYSDA,SPACE=(TRK,(100,100),ELSE),DISP=NEW
//GO.SYSIN DD *

```

2
2

PRINT

* EVENT,MANUFACTURER,LOCATION,SUSCEPTABILITIES, AND SPECIAL CONDITIONS

3

PIPE-1	N	1	IUPS	AO	N1
PIPE-1	N	A1	1234	56	7
PIPE-2	N	1	IUPS	AO	N2
PIPE-2	N	A1	1234	56	7
VALVE-4	G	1	GIS	AC	N2M1
VALVE-4	G	A1	123	45	607
PUMP-1	1	1	GIUPS	AO	N1M3E1
PUMP-1	1	A2	12345	67	080900
PUMP-2	1	1	GIUPS	AO	N2E2M4
PUMP-2	1	A2	12345	67	0809
HX-A	B	1	SUI	AO	N4
HX-A	B	A3	123	45	06
VALVE-A	G	1	GIS	AO	N4M2
VALVE-A	G	A3	123	45	0607
PIPE-4	E	1	IUPS	AO	N4
PIPE-4	E	A6	1234	56	07
VALVE-1	R	1	GIS	AC	N4M1
VALVE-1	R	A6	123	45	0607
PUMP-A	A	1	GIUPS	AO	N4E1M3
PUMP-A	A	A4	12345	67	08
VALVE-5	R	1	GIS	AO	N1M2
VALVE-5	R	A5	123	46	0507
PIPE-5	E		IUPS	AO	N5
PIPE-5	E	A7	1234	56	07
PIPE-6	E		IUPS	AO	N6
PIPE-6	E	A7	1234	56	07
VALVE-2	G		GISS	AO	N6M1

(continued)

Table E.1 (continued)

VALVE-2	G	A7	1233	56	0708
PIPE-3	N		IUPS	AC	N3
PIPE-3	N	A8	1234	56	07
VALVE-3	R		GISS	AO	N3M2
VALVE-3	R	A8	1233	45	0608
HX-B	B		SUI	AC	N6
HX-B	B	A9	123	45	06
VALVE-B	G		GIS	AC	N6M1
VALVE-B	G	A9	123	45	0608
PUMP-C	A	0	GIUPS	AO	N5E2M3
PUMP-C	A	A10	12345	67	009508
PUMP-B	A		GIUPS	AO	N4E4M4
PUMP-B	A	A10	12345	67	030609

END

* NO. OF MAPS LOCATION AND LOCATION ON EACH MAP

3

A1	B1	C1	D1
A2	B1	C2	D1
A3	B1	C3	D1
A4	B1	C4	D1
A5	B1	C5	D1
A6	B1	C6	D1
A7	B1	C7	D2
A8	B1	C8	D2
A9	B1	C9	D2
A10	B1	C10	D2

END

* SUSCEPTIBILITIES WITH MAP USED AND SPECIAL CONDITIONS

IMPACT	I	1	1
VIBRATION	U	1	1
STRESS	S	1	3
PRESSURE	P	1	3
GRIT (DUST OR GRIME)	G	1	3
CORROSION (ACID)	O	3	2
CORROSION (OXIDATION)	A	3	2
SAME HYDRAULIC FLOW LOCP	N1	4	
SAME HYDRAULIC FLOW LOCP	N2	4	
SAME HYDRAULIC FLOW LOCP	N3	4	
SAME HYDRAULIC FLOW LOCP	N4	4	
SAME HYDRAULIC FLOW LOCP	N5	4	
SAME HYDRAULIC FLOW LOCP	N6	4	
SAME POWER SOURCE	E1	4	
SAME POWER SOURCE	E2	4	
SAME MAINT. PROCEDURE	M1	4	
SAME MAINT. PROCEDURE	M1	4	
SAME MAINT. PROCEDURE	M2	4	
MAINT. - JOE'S FIXIT	M3	4	
SOUTHERN SERVICES MAINT.	M4	4	

END

* MANUFACTURER/ABBREVIATION TABLE

MANUFACTURER - 1	N
MANUFACTURER - 1	N
MANUFACTURER - 2	G
MANUFACTURER - 3	1
MANUFACTURER - 4	B

(continued)

Table E.1 (continued)

491

```

MANUFACTURER - 5      E
MANUFACTURER - 6      E
MANUFACTURER - 7      A
END
*   FAULT TREE INPUT
A TEST TO LINK MOCUS AND BACFIR
* DATA
    5      CUT      PRINT
END
FAULT TREE DESCRIPTION
* TREE
TOP      OR      7 0 TOP1      G1      G2      G3      G4      G5      TOP2
TOP1     OR      7 0 G8      G9      G10     G11     G12     G13     G14
TOP2     OR      3 0 G6      G7      G15
G1       AND     0 2 PUMP-A    PUMP-B
G2       AND     0 4 PIPE-6    PUMP-C    PIPE-1    PUMP-2
G3       AND     0 2 PUMP-A    HX-B
G4       AND     0 2 VALVE-A    PUMP-B
G5       AND     0 4 VALVE-2    PUMP-C    PIPE-1    PIPE-2
G6       AND     0 4 PIPE-6    PUMP-C    PUMP-1    PIPE-2
G7       AND     0 4 PIPE-4    PIPE-5    PIPE-1    PIPE-2
G8       AND     0 5 PUMP-B    PUMP-C    VALVE-3    PUMP-2    VALVE-4
G9       AND     0 3 PIPE-1    PIPE-2    PUMP-A
G10      AND     0 5 PUMP-A    PUMP-C    VALVE-3    PUMP-2    VALVE-4
G11      AND     0 4 PUMP-B    PUMP-C    PIPE-1    PUMP-2
G12      AND     0 4 PUMP-A    PUMP-C    PUMP-1    PUMP-2
G13      AND     0 4 PUMP-A    PUMP-C    PIPE-1    PUMP-2
G14      AND     0 4 VALVE-2    PUMP-C    PUMP-1    PUMP-2
G15      AND     0 4 PUMP-C    PUMP-E    PUMP-1    PUMP-2
END
* TESIS
TOP
END
/*

```

Table (E.2)
Output from MOCUS-BACFIRE Package

3				
PIPE-1	N	1	IUPS	AO M1
	A1	1	123400000056000	7 0 0 0 0
PIPE-2	N	1	IUPS	AO M2
	A1	1	123400000056000	7 0 0 0 0
VALVE-4	G	1	GIS	AO M2M1
	A1	1	123000000045000	6 7 0 0 0
PUMP-1	1	1	GIUPS	AO M1M2M1
	A2	1	123450000067000	8 9 0 0 0
PUMP-2	1	1	GIUPS	AO M2E2M4
	A2	1	123450000067000	8 9 0 0 0
HX-A	2	1	SUI	AO M4
	A3	1	123000000045000	6 0 0 0 0
VALVE-A	G	1	GIS	AO M4M2
	A3	1	123000000045000	6 7 0 0 0
PIPE-4	E	1	IUPS	AO M4
	A6	1	123400000056000	7 0 0 0 0
VALVE-1	R	1	GIS	AO M4M1
	A6	1	123000000045000	6 7 0 0 0
PUMP-4	A	1	GIUPS	AO M4E1M3
	A4	1	123450000067000	8 0 0 0 0
VALVE-5	R	1	GIS	AO M1M2
	A5	1	123000000046000	5 7 0 0 0
PIPE-5	E	1	IUPS	AO M5
	A7	1	123400000056000	7 0 0 0 0
PIPE-6	E	1	IUPS	AO M6
	A7	1	123400000056000	7 0 0 0 0
VALVE-2	G	1	GISS	AO M6M1
	A7	1	123300000056000	7 8 0 0 0
PIPE-3	N	1	IUPS	AO M3
	A8	1	123400000056000	7 0 0 0 0
VALVE-3	R	1	GISS	AO M3M2
	A8	1	123300000045000	6 8 0 0 0
HX-B	B	1	SUI	AO M6
	A9	1	123000000045000	6 0 0 0 0
VALVE-B	G	1	GIS	AO M6M1
	A9	1	123000000045000	6 8 0 0 0
PUMP-C	A	1	GIUPS	AO M5E2M3
	A10	1	123450000067000	0 5 8 0 0
PUMP-B	A	1	GIUPS	AO M4E4M4
	A10	1	123450000067000	3 6 9 0 0

3				
A1	B1	C1	D1	
A2	B1	C2	D1	

Table (E.2) continued

A2	B1	C2	D1
A3	B1	C3	D1
A4	B1	C4	D1
A5	B1	C5	D1
A6	B1	C6	D1
A7	B1	C7	D2
A8	B1	C8	D2
A9	B1	C9	D2
A10	B1	C10	D2

IMPACT	I	1	1
VIBRATION	U	1	1
STRESS	S	1	3
PRESSURE	P	1	3
GRIT (DUST OR GRINE)	G	1	3
CORROSION (ACID)	O	3	2
CORROSION (OXIDATION)	A	3	2
SAME HYDRAULIC FLOW LOOP	H1	4	0
SAME HYDRAULIC FLOW LOOP	H2	4	0
SAME HYDRAULIC FLOW LOOP	H3	4	0
SAME HYDRAULIC FLOW LOOP	H4	4	0
SAME HYDRAULIC FLOW LOOP	H5	4	0
SAME HYDRAULIC FLOW LOOP	H6	4	0
SAME POWER SOURCE	E1	4	0
SAME POWER SOURCE	E2	4	0
SAME MAINT. PROCEDURE	M1	4	0
SAME MAINT. PROCEDURE	M1	4	0
SAME MAINT. PROCEDURE	M2	4	0
MAINT. - JOHN'S FIXIT	M3	4	0
SOUTHERN SERVICES MAINT.	M4	4	0

MANUFACTURER - 1	N
MANUFACTURER - 1	N
MANUFACTURER - 2	G
MANUFACTURER - 3	1
MANUFACTURER - 4	D
MANUFACTURER - 5	E
MANUFACTURER - 6	E
MANUFACTURER - 7	A

Table (E.2) continued

PAGE 1
BACFIRE VERSION 3 JULY/76

BACFIRE --- A PROGRAM TO OBTAIN COMMON CAUSE CANDIDATES

NUMBER OF EVENTS WITH THEIR LOCATIONS AND GCS'S - - - -	20
NUMBER OF MAPS USED - - - - -	3
NUMBER OF LOCATIONS AND THEIR MAP LOCATIONS - - - - -	10
NUMBER OF GCS'S - - - - -	20
NUMBER OF MANUFACTURERS - - - - -	8
SENSITIVITY CUT OFF - - - - -	3

(continued)

Table (E.2) continued

MOCUS - - VERSION 3/74

```

*****
- MOCUS - - A COMPUTER PROGRAM TO OBTAIN MINIMAL SETS FROM FAULT TREES -
*****

```

MOCUS INPUT EDIT

A TEST TO LINK MOCUS AND DACFIB

* DATA

5 CUT PRINT

END

FAULT TREE DESCRIPTION

* TREE

TOP	OR	7	J	TOP1	G1	G2	G3	G4	G5	TOP2
TOP1	OR	7	0	G8	G9	G10	G11	G12	G13	G14
TOP2	OR	3	0	G6	G7	G15				
G1	AND	0	2	PUMP-A	PUMP-B					
G2	AND	0	4	PIPE-6	PUMP-C	PIPE-1	PUMP-2			
G3	AND	0	2	PUMP-A	HX-B					
G4	AND	3	2	VALVE-A	PUMP-B					
G5	AND	3	4	VALVE-2	PUMP-C	PIPE-1	PIPE-2			
G6	AND	0	4	PIPE-6	PUMP-C	PUMP-1	PIPE-2			
G7	AND	0	4	PIPE-4	PIPE-5	PIPE-1	PIPE-2			
G8	AND	0	5	PUMP-B	PUMP-C	VALVE-3	PUMP-2	VALVE-4		
G9	AND	0	3	PIPE-1	PIPE-2	PUMP-A				
G10	AND	0	5	PUMP-A	PUMP-C	VALVE-3	PUMP-2	VALVE-4		
G11	AND	3	4	PUMP-B	PUMP-C	PIPE-1	PUMP-2			
G12	AND	0	4	PUMP-A	PUMP-C	PUMP-1	PUMP-2			
G13	AND	0	4	PUMP-A	PUMP-C	PIPE-1	PUMP-2			
G14	AND	3	4	VALVE-2	PUMP-C	PUMP-1	PUMP-2			
G15	AND	0	4	PUMP-C	PUMP-D	PUMP-1	PUMP-2			

END

* TESTS

TOP

END

Table (E.2) continued

A TEST TO LINK MOCUS AND EACFIR

MOCUS - - VERSION 3/74

NUMBER OF GATES, NG----- 18

NUMBER OF COMPONENTS, NI----- 15

MAXIMUM SIZE OF SETS, ITHAX----- 5

CUT SET - DATA SET SWITCH, INDEX1-----CUT

PRINT - PUNCH SWITCH, INDEX2-----PRINT

Table (E.2) continued

A TEST TO LINK HOCUS AND BACPLR

HOCUS - - VERSION 3/74

NAME	TYPE	INPUTS----							
TOP	OR	7	0	TOP1	G1	G2	G3	G4	G5
TOP1	OR	7	0	G8	G9	G10	G11	G12	G13
TOP2	OR	3	0	G6	G7	G15			
G1	AND	3	2	PUMP-A	PUMP-B				
G2	AND	0	4	PIPE-6	PUMP-C	PIPE-1	PUMP-2		
G3	AND	3	2	PUMP-A	HX-B				
G4	AND	0	2	VALVE-A	PUMP-B				
G5	AND	3	4	VALVE-2	PUMP-C	PIPE-1	PIPE-2		
G6	AND	3	4	PIPE-6	PUMP-C	PUMP-1	PIPE-2		
G7	AND	0	4	PIPE-4	PIPE-5	PIPE-1	PIPE-2		
G8	AND	3	5	PUMP-B	PUMP-C	VALVE-3	PUMP-2	VALVE-4	
G9	AND	0	3	PIPE-1	PIPE-2	PUMP-A			
G10	AND	3	5	PUMP-A	PUMP-C	VALVE-3	PUMP-2	VALVE-4	
G11	AND	0	4	PUMP-B	PUMP-C	PIPE-1	PUMP-2		
G12	AND	0	4	PUMP-A	PUMP-C	PUMP-1	PUMP-2		
G13	AND	0	4	PUMP-A	PUMP-C	PIPE-1	PUMP-2		
G14	AND	0	4	VALVE-2	PUMP-C	PUMP-1	PUMP-2		
G15	AND	0	4	PUMP-C	PUMP-B	PUMP-1	PUMP-2		

(continued)

Table (E.2) continued

KOCUS - - VERSION 3/74

MINIMAL CUT SETS FOR GATE TOP

CUT SETS WITH 1 COMPONENTS

NONE EXIST.

CUT SETS WITH 2 COMPONENTS

2) 1) PUMP-A PUMP-B

1) PUMP-A PUMP-B

COMMON CONDITION -- FIRST CRITERION

SAME HYDRAULIC FLOW LOOP
MANUFACTURER - 7

RANK:	MIN	MAX	AVG	SUM
	3	8	5	11
	9	9	9	18

COMMON SUSCEPTIBILITY -- SECOND CRITERION

VIBRATION

RANK:	MIN	MAX	AVG	SUM
	3	3	3	6

2) 2) PUMP-A HX-B

2) 3) VALVE-A PUMP-B

2) VALVE-A PUMP-B

COMMON CONDITION -- FIRST CRITERION

SAME HYDRAULIC FLOW LOOP

RANK:	MIN	MAX	AVG	SUM
	3	6	4	9

CUT SETS WITH 3 COMPONENTS

3) 1) PIPE-1 PIPE-2 PUMP-A

3) PIPE-1 PIPE-2 PUMP-A

COMMON SUSCEPTIBILITY -- SECOND CRITERION

PRESSURE
STRESS

RANK:	MIN	MAX	AVG	SUM
	3	4	3	10
	4	5	4	13

(continued)

Table (E.2) continued

CUT SETS WITH 4 COMPONENTS

4	1)	PIPE-6	PUMP-C	PIPE-1	PUMP-2
4	2)	VALVE-2	PUMP-C	PIPE-1	PIPE-2
4	**WARNING** 5 FOUND REPEATED ON EVENT DATA CARD.				
4	3)	PIPE-6	PUMP-C	PUMP-1	PIPE-2
4	4)	PUMP-B	PUMP-C	PIPE-1	PUMP-2
4	5)	PUMP-A	PUMP-C	PUMP-1	PUMP-2
4	6)	PUMP-A	PUMP-C	PUMP-1	PUMP-2

COMMON SUSCEPTIBILITY -- SECOND CRITERION
VIBRATION

RANK:	MIN	MAX	AVG	SUM
	3	3	3	12

4	6)	PUMP-A	PUMP-C	PIPE-1	PUMP-2
4	7)	VALVE-2	PUMP-C	PUMP-1	PUMP-2
4	8)	PIPE-4	PIPE-5	PIPE-1	PIPE-2
4	9)	PUMP-C	PUMP-D	PUMP-1	PUMP-2
4	5)	PUMP-C	PUMP-B	PUMP-1	PUMP-2

COMMON SUSCEPTIBILITY -- SECOND CRITERION
VIBRATION

RANK:	MIN	MAX	AVG	SUM
	3	3	3	12

CUT SETS WITH 5 COMPONENTS

5	1)	PUMP-B	PUMP-C	VALVE-3	PUMP-2	VALVE-4
5	2)	PUMP-A	PUMP-C	VALVE-3	PUMP-2	VALVE-4

(continued)

Table (E.2) continued

QUALITATIVE FAILURE CHARACTERISTICS

IMPACT

VIBRATION

DOMAIN B1
U13 1 1
U13 4 9
U13 5 13

STRESS

DOMAIN D1
S14 3 4

PRESSURE

DOMAIN D1
P13 3 4

GRIT (DUST OR GRINE)

CORROSION (ACID)

CORROSION (OXIDATION)

SAME HYDRAULIC FLOW LOOP

SAME HYDRAULIC FLOW LOOP

SAME HYDRAULIC FLOW LOOP

SAME HYDRAULIC FLOW LOOP
N44 1 1
N44 2 3

SAME HYDRAULIC FLOW LOOP

SAME HYDRAULIC FLOW LOOP

Table (E.2) continued

SAME POWER SOURCE

SAME POWER SOURCE

SAME MAINT. PROCEDURE

SAME MAINT. PROCEDURE

SAME MAINT. PROCEDURE

MAINT. - JOE'S FIXIT

SOUTHERN SERVICES MAINT.

MANUFACTURER - 1

MANUFACTURER - 1

MANUFACTURER - 2

MANUFACTURER - 3

MANUFACTURER - 4

MANUFACTURER - 5

MANUFACTURER - 6

MANUFACTURER - 7
A 5 1 1

THIS CONCLUDES THE OUTPUT FROM BACFIN.

- [1] "BACFIRE - A Computer Program for Common Cause Failure Analysis", Nucl. Eng. Dep., University of Tennessee, NERS-77-02, May, 1977
- [2] Fussell, J. B., et. al., "MOCUS - A Computer Program to Obtain Minimal Sets from Fault Trees", Aerojet Nuclear Company, TIO-4500, August, 1974