

Development of a Graphical Approach to Software Requirements Analysis

by

Xinhui Chen

B.E., M.E., Engineering Physics (1989, 1991)

Tsinghua University, Beijing

M.S., Nuclear Engineering (1996)

Massachusetts Institute of Technology

Submitted to the Department of Nuclear Engineering
in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

at the

Massachusetts Institute of Technology

May 1998

© Massachusetts Institute of Technology, 1998. All rights reserved.

Signature of Author.....

Department of Nuclear Engineering

May 1, 1998

Certified by

Prof. George E. Apostolakis, Nuclear Engineering

Thesis Supervisor

Certified by

Prof. Michael W. Golay, Nuclear Engineering

Thesis Reader

Accepted by

Prof. Lawrence M. Lidsky

Chairman, Department Committee on Graduate Students

AUG 18 1998 Science

Development of a Graphical Approach to Software Requirements Analysis

by

Xinhui Chen

Submitted to the Department of Nuclear Engineering
on May 1, 1998 in Partial Fulfillment of the
Requirements for the Degree of
Doctor of Philosophy

Abstract

A user-friendly graphical approach has been developed in the requirements analysis of mission-critical digital systems, especially in assuring the consistency of software requirements. This approach contains three distinct steps. Step One is software requirements collection which collects all the requirements from the customer. Step Two is requirements grouping which divides the set of requirements collected from Step One into smaller groups (or subsets), the union of which is the original set of requirements. Requirements on a specific aspect of software functionality are grouped together so that only logical operations exist within each group. The requirements between the groups (or subsets) may be linked by physical relations. Step Three is the actual analysis procedure for achieving unambiguous, consistent (and more complete) requirements. The procedure contains two sub-step analyses: the within-group analysis and the between-group analysis. A Tree-Based Graphical Approach (TBGA) is developed for the within-group analysis to ensure that the requirements are consistent within individual groups. Compared with the Purely Analytical Approach, the Tree-Based Graphical Approach is more mechanical since the analytical approach is sequential or one-dimensional while the graphical approach works in parallel or two-dimensional. As a byproduct, it can also be used to prove logic identity. In addition, two other approaches are identified for the within-group analysis in requirements: the Truth-Table Approach and the Dynamic Flowgraph Methodology (DFM). These methods can be used independently as well as used as a double check of the Tree-Based Graphical Approach. The between-group analysis is performed using dependency diagram in which each group of the requirements is represented by a node. If the groups are related physically, they are linked by an arc. The between-group analysis makes use of physical laws to check consistency in requirements between the groups as well as helps make individual group of requirements more complete. The between-group analysis indicates that, in requirements analysis, physics laws should be taken into account in addition to pure logic operations. If the within-group analysis shows that the requirements are consistent within each group and the between-group analysis show that the they are consistent between all linked groups, the customer's original set of requirements is consistent. Otherwise, it is inconsistent. In either case, the result of the analysis should be feedback to and shared with the customer.

Thesis Supervisor: George E. Apostolakis
Title: Professor of Nuclear Engineering

Acknowledgments

I wish to thank Professor George Apostolakis for his consistent support and regular discussions during the course of my dissertation. His quick grasp of both technical and personal issues has made my dissertation possible.

I acknowledge and thank Dr. John Bernard and Professor Michael Golay for serving on my Doctoral Thesis Committee in the past two years. The case study provided by Dr. Bernard is especially invaluable in testing the usability and effectiveness of the proposed methodology. I thank them for their continuous involvement and suggestions. I appreciate Christopher Garrett for many technical discussions, as well as his help of English in Chapters 4 and 5 of the dissertation.

My appreciation goes to Idaho National Engineering and Environmental Laboratory (INEEL) for sponsoring the cutting-edge research. I especially acknowledge Steven Novack for valuable discussions during the course of the project.

Deepest of all, I thank my wife Shuying for her consistent support and encouragement (without which I would not have made it) before and while I fulfilled my dream of a Ph.D. in the world's most prestigious institution. I thank my son Kevin for his understanding of my balance as a father and a Ph.D. candidate, and the joy he has brought to my entire family. I feel owing a debt to my parents, parents-in-law for taking care of Kevin as well as supporting my wife and myself. I thank my only brother, Xinjun, for his filial support to my parents during the years I was absent and his continuous encouragement to me. The sacrifices of my family are gratefully acknowledged.

Table of Contents

Abstract.....	3
Acknowledgments	5
Table of Contents	7
List of Figures.....	15
List of Tables	17
List of Acronyms	19
Chapter 1 Introduction	21
1.1 A Broader Picture	21
1.1.1 Moving towards Risk-Informed, Performance-Based Regulation.....	23
1.1.1.1 Deterministic-Based Regulation	23
1.1.1.2 Risk-Informed, Deterministic-Based Regulation.....	23
1.1.1.3 Performance-Based Regulation.....	24
1.1.1.4 Risk-Informed, Performance-Based Regulation	25
1.1.2 Importance of RIPBR	25
1.2 Statement of the Problem.....	27
1.3 The Waterfall Model and Software Requirements Analysis	29
1.3.1 The Waterfall Model.....	29
1.3.2 Requirements Analysis	29
1.3.3 Importance of Requirements Analysis.....	31
1.3.4 Inadequacy of Natural Languages in Documenting Software Requirements	31
1.3.5 Attributes of Software Requirements Specification.....	33
Chapter 2 Current State of the Art and Practice	35

2.1 Formal Methods	35
2.1.1 Definition	35
2.1.2 Categorization of Formal Methods	36
2.1.3 Taxonomy of Degrees of Rigor in Formal Methods.....	37
2.1.4 Formal Methods and the Industry	41
2.1.4.1 Advantages of Formal Methods.....	41
2.1.4.2 Disadvantages of Formal Methods	41
2.2 Finite State Machine	43
2.3 An Overview of Methodologies on Software Requirements Analysis	47
2.3.1 Software Cost Reduction (SCR)	47
2.3.1.1 SCR Tabular Notation.....	47
2.3.1.2 An Example in SCR Notation.....	47
2.3.1.3 Tool Support for SCR Notation	51
2.3.1.4 A Counterexample for SCR Notation.....	51
2.3.2 Requirements State Machine Language (RSML)	54
2.3.2.1 RSML and Its Tool Support.....	54
2.3.2.2 RSML and SCR	56
2.3.3 PVS Verification System	58
2.3.4 Quasi-Classical (QC) Logic	59
2.3.5 Fuzzy Logic and Utility Function	59
2.3.6 Structured Analysis and Design Technique (SADT).....	59
2.3.7 Systematic Activity Modeling Method (SAMM).....	60
2.3.8 Higher Order Software (HOS)	60
2.3.9 Requirements Statement Language (RSL).....	60
2.4 Summary	61
 Chapter 3 Fundamentals of Discrete Mathematics	 63
3.1 Definitions and Terminology	63
3.2 Propositional Calculus	66
3.2.1 Basic Propositional Operations.....	66

3.2.1.1 Negation.....	66
3.2.1.2 Conjunction.....	66
3.2.1.3 Disjunction.....	67
3.2.1.4 Implication	67
3.2.1.5 Equivalence.....	69
3.2.2 Derived Propositional Operations.....	69
3.2.2.1 Exclusive OR	69
3.2.2.2 NAND	69
3.2.2.3 NOR	70
3.2.3 Important Logic Laws.....	70
3.2.4 Important Inference Rules.....	70
3.2.5 Some Important Theorems.....	70
3.3 Predicate Calculus.....	76
Chapter 4 The Proposed Methodology	79
4.1 Introduction.....	79
4.2 Graphical Representation of Logic Operations.....	81
4.3 Procedure in the Proposed Graphical Approach.....	83
4.4 The Proposed Graphical Approach and An Example	85
4.4.1 Requirements Grouping and Its Benefits.....	85
4.4.2 Within-Group Analysis.....	88
4.4.2.1 Inspection.....	89
4.4.2.2 Variable Definitions.....	89
4.4.2.3 Translation of Requirements from English to Logic Expressions	90
4.4.2.4 Tree-Based Graphical Approach.....	90
4.4.2.5 Tree-Based Graphical Approach vs. Fault Tree Analysis.....	93
4.4.3 Between-Group Analysis	96
4.5 Four Approaches Applicable to Within-Group Analysis.....	99
4.5.1 A Simple Example	99
4.5.2 The Purely Analytical Approach.....	101

4.5.3 The Truth-Table Approach	102
4.5.4 The Dynamic Flowgraph Methodology (DFM) Approach.....	105
4.5.4.1 Features of DFM.....	105
4.5.4.2 Automated Tool	106
4.5.4.3 Previous Applications of DFM	109
4.5.4.4 Use of DFM to Find Prime Implicants of Logic Expressions	109
4.5.4.5 Application of DFM to Requirements Analysis	110
4.5.5 The Tree-Based Graphical Approach.....	115
4.5.6 Summary	115
 Chapter 5 Application of the Proposed Approach to a Case Study.....	 119
5.1 Description of the Case Study.....	119
5.1.1 MIT-SNL Period-Generated Minimum Time Control Laws.....	119
5.1.2 Annular Core Research Reactor.....	122
5.2 Requirements Collection.....	124
5.3 Requirements Grouping	130
5.3.1 General Information.....	130
5.3.2 Power Multiplication Group	130
5.3.3 Operational Mode Group	131
5.3.4 Fuel Temperature Group.....	131
5.3.5 Reactor Period Group	132
5.3.6 Reactivity Group.....	132
5.3.7 Motor Frequency Group	133
5.3.8 Rod Velocity Group.....	134
5.3.9 Hardware and Other non-Software Group	135
5.4 Within-Group Analysis.....	137
5.4.1 General Information.....	137
5.4.2 Power Multiplication Group	137
5.4.2.1 Analysis by Inspection.....	138
5.4.2.2 Lessons Learned.....	139

5.4.3 Operational Mode Group	139
5.4.3.1 The Analytical Approach	143
5.4.3.2 The Truth-Table Approach	144
5.4.3.3 The DFM Approach	149
5.4.3.4 The Tree-Based Graphical Approach.....	155
5.4.3.5 Order Analysis in the Tree-Based Graphical Approach	159
5.4.3.6 Lessons Learned.....	160
5.4.4 Fuel Temperature Group	167
5.4.4.1 Analysis by Inspection.....	167
5.4.4.2 Lessons Learned.....	167
5.4.5 Reactor Period Group	167
5.4.6 Reactivity Group.....	168
5.4.6.1 Analysis by Inspection.....	169
5.4.6.2 Analysis by Tree-Based Graphical Approach.....	169
5.4.6.3 Lessons Learned.....	172
5.4.7 Motor Frequency Group	172
5.4.7.1 Analysis by Inspection.....	173
5.4.8 Rod Velocity Group.....	174
5.4.9 Hardware and Other non-Software Group	175
5.4.10 Revised Requirements after Within-Group Analysis.....	176
5.5 Between-Group Analysis	180
5.5.1 Power Multiplication–Reactor Period.....	180
5.5.2 Power Multiplication–Operatioal Mode	186
5.5.3 Reactor Period–Rod Velocity	190
5.5.3.1 Reactor Period–Rate of Reactivity.....	190
5.5.3.2 Rate of Reactivity–Rod Velocity	191
5.5.4 Rod Velocity–Motor Frequency	193
5.5.5 Operational Mode–Fuel Temperature.....	195
5.5.6 Reactor Period–Reactivty	195
5.5.7 Revised Requirements after the Between-Group Analysis.....	196

Chapter 6 Conclusions and Future Work.....	201
6.1 Conclusions and Findings.....	201
6.2 Future Work.....	203
References	205
Appendix 1 Differences between Digital Systems and Analog/Mechanical Systems	227
A1.1 Different Complexities.....	227
A1.1.1 Software Complexity Measures.....	227
A1.1.2 Software Maintainability Index	228
A1.2 Different Behaviors and Mathematical Modeling.....	229
A1.3 Different Data Manipulation Processes	230
A1.4 Different Production Processes.....	230
A1.5 Different Failure Modes.....	232
A1.6 Different Failure Manifestation Patterns.....	232
A1.7 Different Consequences	233
A1.8 Different Weightings in Cost Structure	233
A1.9 Different Destinies	234
Appendix 2 A Survey on Software Development Cycle Models.....	237
A2.1 The Code-and-Fix Model.....	237
A2.2 The Waterfall Model.....	238
A2.2.1 Planning Activities.....	238
A2.2.2 Software Requirements Analysis	239
A2.2.3 Software Design.....	239
A2.2.4 Software Implementation.....	239
A2.2.5 Software Testing	239
A2.2.6 Software Installation	240
A2.2.7 Operation and Maintenance	240

A2.3 The Evolutionary Development Model.....	240
A2.4 The Transform Model	241
A2.5 The Spiral Model	242
A2.5.1 A Typical Cycle of the Spiral	242
A2.5.2 Initiating and Terminating the Spiral	242
A2.5.3 Advantages and Disadvantages of the Spiral Model.....	243
Appendix 3 Recommended Practices to Improved Software Quality.....	245
A3.1 Testing.....	245
A3.1.1 Black Box Testing and White Box Testing	245
A3.1.2 Unit Testing, Integration Testing, System Testing and Acceptance Testing	247
A3.1.3 Boundary Condition Testing.....	247
A3.1.4 Drawbacks of Testing	248
A3.2 Design Diversity	250
A3.3 Commercial Off-the-Shelf (COTS) Software	251
A3.4 Reusable Software Modules.....	252
A3.4.1 Benefits from Reusable Modules.....	252
A3.4.2 Guidelines on Software Modularization	253
A3.5 Object-Oriented Programming (OOP).....	255
A3.5.1 Characteristics of Object-Oriented Paradigm	255
A3.5.2 Benefits of OOP.....	256
A3.6 Fault Avoidance	257
Appendix 4 A Survey on Formal Methods	259
A4.1 Vienna Development Method (VDM)	259
A4.1.1 Components of a VDM Specification.....	259
A4.1.2 The Weakest Pre-Condition	260
A4.1.3 An Example in VDM	262
A4.1.4 Recent Developments in VDM	263

A4.2 Z	265
A4.2.1 The Z Notation	265
A4.2.2 Z and VDM	266
A4.2.3 An Example in Z	266
A4.3 Development Before the Fact (DBTF) and the OO1 System	270
A4.4 Symbolic Model Verifier (SMV)	273
A4.4.1 The SMV Notation	273
A4.4.2 An Example in SMV	274
Appendix 5 Software Safety Analysis Methodologies	277
A5.1 Software Fault Tree Analysis (SFTA)	277
A5.1.1 An Upper Bound of the Probability of Top Event	279
A5.1.2 Applications of SFTA and Tool Support	282
A5.2 The Dynamic Flowgraph Methodology (DFM)	284
A5.3 HAZards and OPerability Analysis (HAZOP)	287
Appendix 6 Application of TBGA to Prove Logic Identity	291
Appendix 7 Another Graphical Method Developed to Document Software Requirements Specification	295
A7.1 An <i>M</i> -ary Tree-Based Graphical Method and An Example	296
A7.2 Advantages of the Graphical Method	298
A7.3 Findings	299
Appendix 8 Application of Flowchart in Code Analysis	303
A8.1 The Flowchart	303
A8.2 Application of Flowchart	303
A8.3 Code Listing of Function REACTR(P)	309
Appendix 9 Proofs of Theorems in Section 3.2.5	313
Appendix 10 A Between-Group Analysis: Reactor Period–Rod Velocity	323

List of Figures

Figure 1.3-1	The Waterfall Model.....	30
Figure 2.1-1	Formal Methods and the Hierarchical Components	38
Figure 2.2-1	A Graphical FSM Equivalent to Table 2.2-1	46
Figure 2.3-1	Code Segment from [Viol96].....	49
Figure 2.3-2	The <i>M</i> -ary Tree-Based Graphical Method	55
Figure 2.3-3	RSML Specification of the Example in Table 2.3-1.....	57
Figure 4.3-1	Schematic of the Proposed Graphical Approach	84
Figure 4.4-1	Within-Group Analysis for Group “Driving Force” in the Example.....	95
Figure 4.4-2	Between-Group Analysis for the Example	98
Figure 4.5-1	DFM Model for the Example.....	114
Figure 4.5-2	DFM Result for the Example (Screen Dump)	114
Figure 4.5-3	Tree-Based Graphical Approach for the Example	116
Figure 4.5-4	Order-Independence of the Tree-Based Graphical Approach.....	116
Figure 5.1-1	Isometric View of Annual Core Research Reactor	123
Figure 5.2-1	Transient Rod Worth.....	133
Figure 5.4-1	DFM Model for the Case Study.....	150
Figure 5.4-2	DFM Results for the Case Study (Screen Dump).....	154
Figure 5.4-3	Within-Group Analysis Using the Tree-Based Graphical Approach (Operational Mode Group) (A) SO= true (B) PO= true.....	157
Figure 5.4-4	Order Analysis for the Within-Group Analysis (Operational Mode Group, PO = true, 1/6).....	161
Figure 5.4-5	Order Analysis for the Within-Group Analysis (Operational Mode Group, PO = true, 2/6).....	162
Figure 5.4-6	Order Analysis for the Within-Group Analysis (Operational Mode Group, PO = true, 3/6).....	163
Figure 5.4-7	Order Analysis for the Within-Group Analysis (Operational Mode Group, PO = true, 4/6).....	164
Figure 5.4-8	Order Analysis for the Within-Group Analysis (Operational Mode Group, PO = true, 5/6).....	165
Figure 5.4-9	Order Analysis for the Within-Group Analysis	

	(Operational Mode Group, PO = true, 6/6).....	166
Figure 5.4-10	Within-Group Analysis Using the Tree-Based Graphical Approach (Reactivity Group)	171
Figure 5.4-11	Transient Rod Worth (same as Figure 5.2-1)	179
Figure 5.5-1	Dependency Diagram for Between-Group Analysis of the Case Study	181
Figure 5.5-2	Possible Power Trajectories.....	183
Figure 5.5-3	Possible Reactor Period Trajectory.....	183
Figure 5.5-4	A New Requirement on the Initial Power Levels	189
Figure 5.5-5	Rate of Change of Reactivity (in \$/sec)	192
Figure 5.5-6	Dynamic Rod Velocity with Initial Rod Position at 4062 Units	192
Figure 5.5-7	Maximum rod Velocity vs. Initial Rod Position.....	194
Figure 5.5-8	A New Requirement on the Initial Power Levels (same as Figure 5.5-4).....	198
Figure 5.5-9	Transient Rod Worth (same as Figure 5.2-1)	200
Figure A1-1	Relationship between SMI and Maintainability	231
Figure A1-2	Failure Rates in Analog and Digital Systems (A) An Analog System (B) A Digital System	235
Figure A4-1	Implicit Function Specification.....	261
Figure A4-2	An Example of Specification in Z	269
Figure A4-3	SMV Specification for a Thermostat	275
Figure A5-1	Union of Sets A and B—Cardinality	280
Figure A5-2	An Overview of the SDA Procedure.....	289
Figure A6-1	Application of the Proposed Graphical Approach to Example A6-1.....	293
Figure A6-2	Application of the Proposed Graphical Approach to Example A6-2.....	293
Figure A7-1	M-ary Tree Developed at Step 3 of the Example.....	300
Figure A7-2	M-ary Tree Developed at Step 4 of the Example.....	301
Figure A7-3	M-ary Tree Developed at Step 5 of the Example.....	302
Figure A8-1	At “check” Point, x is confined to $[a, b]$ if $a < b$	306
Figure A8-2	Code Analysis Using Flowchart (1/2).....	307
Figure A8-2	Code Analysis Using Flowchart (2/2)	308

List of Tables

Table 2.2-1	An Example of Mealy Machine	45
Table 2.3-1	The SCR Notation of a Trip Meter (A) Tabular Notation for the Last Section in Figure 2.3-1 (B) An Equivalent of (A)	50
Table 2.3-2	(A) Corresponding Super-Table of the Code Segment (B) An Equivalent of (A)	53
Table 3.2-1	Truth Table for Negation ($\neg P$)	71
Table 3.2-2	Truth Table for Conjunction ($P \wedge Q$)	71
Table 3.2-3	Truth Table for Disjunction ($P \vee Q$)	71
Table 3.2-4	Truth Table for Implication ($P \Rightarrow Q$)	71
Table 3.2-5	Truth Table for Equivalence ($P \Leftrightarrow Q$)	72
Table 3.2-6	Truth Table for Exclusive OR ($P \vee_e Q$)	72
Table 3.2-7	Truth Table for NAND ($P Q$)	72
Table 3.3-8	Truth Table for NOR ($P \downarrow Q$)	72
Table 3.2-9	Important Logic Laws	73
Table 3.2-10	Important Inference Rules	74
Table 3.2-11	Some Important Theorems in Propositional Calculus	75
Table 3.3-1	Universal Quantification and Existential Quantification	77
Table 3.3-2	Some Important Theorems in Predicate Calculus	77
Table 4.2-1	Graphical Representation of Basic Propositional Operations	82
Table 4.4-1	Example: Requirements on an Auto Controller	86
Table 4.4-2	Example: Mathematical Representation of Requirements for an Auto Controller	91
Table 4.4-3	Example: Simplified but Equivalent Representation of Requirements Using the Tree-Based Graphical Approach	94
Table 4.5-1	Simple Example: Requirements for a Computer System	100
Table 4.5-2	Simple Example: the Purely Analytical Approach	100
Table 4.5-3	Simple Example: the Truth Table Approach	104
Table 4.5-4	Building Blocks of a DFM Model	107

Table 4.5-5	Prime Implicants Generated by DFM for Four Test Cases.....	111
Table 4.5-6	Simple Example: Decision Table for Transfer Box 1.....	113
Table 4.5-7	Simple Example: Decision Table for Transfer Box 2.....	113
Table 4.5-8	Simple Example: Decision Table for Transfer Box 3.....	113
Table 4.5-9	Simple Example: Decision Table for Transfer Box 4.....	113
Table 4.5-10	Comparison of the Four Approaches in Within-Group Analysis	117
Table 5.4-1	Case Study: the Purely Analytical Approach (Operational Mode Group)	
	(A) Case 1: SO = true.....	145
	(B) Case 2: PO = true.....	146
Table 5.4-2	Case Study: the Truth-Table Approach (Operational Mode Group)	147
Table 5.4-3	Case Study: Intermediate Nodes and Logic Expressions in Figure 5.4-1	151
Table 5.4-4	Case Study: Decision Table in the DFM Model.....	152
Table A3-1	Cost of Software Testing	249
Table A3-2	Coupling and Software Maintenance.....	254
Table A4-1	A Full List of Symbols Used to Distinguish Types of Relations in Z	267

List of Acronyms

ABB-CE	Asea Brown Boveri/Combustion Engineering
ABWR	Advanced Boiling Water Reactor
ACRR	Annular Core Research Reactor
AECB	Atomic Energy Control Board (of Canada)
ANS	American Nuclear Society
CANDU	CANada Deuterium Uranium (Reactor)
CFR	Code of Federal Regulations
CNF	Conjunctive Normal Form
COTS	Commercial-Off-The-Shelf
DBTF	Design Before The Fact
DFM	Dynamic Flowgraph Methodology
DNF	Disjunctive Normal Form
DOD	Department of Defense
ETA	Event Tree Analysis
FSM	Finite State Machine
FTA	Fault Tree Analysis
GE	General Electric
HAZOP	HAZard and OPerability (Analysis)
IAEA	International Atomic Energy Agency
I&C	Instrumentation and Control
IEC	International Electrotechnical Commission
IEEE	Institute of Electrical and Electronics Engineers
INEEL/INEL	Idaho National Engineering and Environmental Laboratory
IPE	Individual Plant Examination
IPEEE	Individual Plant Examination–External Events
LOC	Lines of Code
MCS	Minimum Cut Set

MIT	Massachusetts Institute of Technology
MITR	Massachusetts Institute of Technology Research Reactor
MTTF	Mean-Time-To-Failure
NPP	Nuclear Power Plant
NRC	National Research Council
OOD	Object-Oriented Design
OOP	Object-Oriented Programming
PRA	Probabilistic Risk Assessment
QA	Quality Assurance
RIPBR	Risk-Informed Performance-Based Regulations
RSML	Requirements State Machine Language
RML	Requirements Statement Language
SBWR	Simplified Boiling Water Reactor
SCR	Software Cost Reduction
SDD	Software Design Document
SNL	Sandia National Laboratories
SRA	Software Requirements Analysis
SRS	Software Requirements Specification
TMI	Three Mile Island
TBGA	Tree-Based Graphical Approach
USNRC	United States Nuclear Regulatory Commission
V&V	Verification and Validation
VDM	Vienna Development Method

Chapter 1

Introduction

1.1 A Broader Picture

Since the early 1970s, the United States Nuclear Regulatory Commission (USNRC) spent significant amount of resources in the development and application of Probabilistic Risk Assessment (PRA) technology including the ground-breaking work of the Reactor Safety Study—WASH-1400 in 1975. In 1979, the USNRC issued a policy statement entitled “(US) NRC Statement of Risk Assessment and the Reactor Safety Study Report (WASH-1400) in Light of the Risk Assessment Review Group Report” (Risk Assessment Review Group Report, NUREG/CR-0400). In addition to addressing specific criticisms of WASH-1400, the 1979 policy statement articulated limitations in the use of PRA in the regulatory arena. Many of these limitations have been addressed; however, some still remain pertinent today. Primary among these limitations is the characterization of uncertainties associated with calculated probabilities of reactor accidents. PRA methodologies have, however, provided a better means for identifying and characterizing the range of uncertainty.

The Three Mile Island (TMI) accident in 1979 substantially changed the character of the analysis of severe accidents worldwide. It led to a substantial research program on severe accident phenomenology. In addition, two major investigations of the accident (the Kemeny and Rogovin studies) recommended that PRA techniques be used more widely to augment the traditional deterministic methods of analyzing nuclear plant safety. In 1984, the USNRC completed a study (NUREG-1050) that addressed the state-of-the-art risk analysis techniques. In early 1991, the USNRC published NUREG-1150, “Severe Accident Risks: An Assessment for Five U.S. Nuclear Power Plants”. In NUREG-1150, the USNRC used improved PRA techniques to assess the risk associated with five nuclear power plants. This study was a significant turning point in the use of risk concepts in the regulatory process and

enabled USNRC to greatly improve its methods for assessing containment performance given core damage initiation and subsequent accident progression. The methods developed for, and results from, these studies provided a valuable foundation in quantitative risk techniques.

PRA methods have been applied successfully in several regulatory activities and have proved to be a valuable complement to traditional deterministic engineering approaches. This application of PRA represents an extension and enhancement of traditional regulation rather than a separate and different technology. Several recent USNRC policies or regulations have been based, in part, on PRA methods and insights. These include the Backfit Rule (10 CFR 50.109, “Backfitting”), the Policy Statement on “Safety Goals for the Operation of Nuclear Power Plants” (51 FR 30028; August 21, 1986), the USNRC’s “Policy Statement on Severe Reactor Accidents Regarding Future Designs and Existing Plants” (50 FR 32138; August 8, 1985), and the USNRC’s “Final Policy Statement on Technical Specifications Improvement for Nuclear Power Reactors” (58 FR 39132; July 22, 1993). PRA methods also were used effectively during the anticipated transient without scram (ATWS) and station blackout (SBO) rulemakings, and have been used extensively in the generic issue prioritization and resolution process. Additional benefits have been found in the use of “Risk-Based Inspection Guides” to focus USNRC reactor inspector efforts and make more efficient use of USNRC inspection resources. Probabilistic analyses were extensively used in the development of the recently proposed rule change to reactor siting criteria in 10 CFR Part 100 (59 FR 52255; October 17, 1994), especially in the area of estimating the Safe Shutdown Earthquake ground motion for a nuclear reactor site.

Currently, the USNRC is using PRA techniques to assess the safety importance of operating reactor events and as an integral part of the design certification review process for advanced reactor designs. In addition, the Individual Plant Examination (IPE) program and the Individual Plant Examination - External Events (IPEEE) program have resulted in commercial reactor licensees using risk-assessment methods to identify any vulnerabilities.

1.1.1 Moving towards Risk-Informed, Performance-Based Regulation (RIPBR)

The purpose of applying risk-informed, performance-based approaches is to deal with uncertainties in regulatory decision-making, and to strategically consider how to ensure regulatory coherence during the transition from deterministic-based regulations to RIPBR.

1.1.1.1 Deterministic-Based Regulation

The USNRC has generally regulated the use of nuclear facilities (including nuclear materials and reactors) based on deterministic approaches. Deterministic approaches to regulation consider a set of challenges to safety and specify how those challenges should be mitigated. In other words, the deterministic approach establishes requirements for use of nuclear facilities and for engineering margin and quality assurance in design, manufacture, construction, and operation of nuclear facilities.

USNRC established its regulatory requirements to ensure that a facility is designed, constructed, and licensed to operate without undue risk to the health and safety of the public. These requirements are largely based on deterministic engineering criteria. In addition, this approach assumes that adverse conditions can exist (e.g., equipment failures and human errors) and establishes a set of design basis events. It then requires that the licensed facility design include safety systems capable of preventing and/or mitigating the consequences of those design basis events to protect the public health and safety. As a matter, even the deterministic approach contains implied elements of probability. For example, reactor vessel rupture is considered too improbable to be included as an accident to be analyzed. However, the likelihood that a single emergency core cooling system or system train would not function was considered so high that safety train redundancy and protection against single failure were required.

1.1.1.2 Risk-Informed, Deterministic-Based Regulation

A risk-informed, deterministic approach to regulation enhances and extends this traditional, deterministic approach, by

- allowing consideration of a broader set of potential challenges to safety;
- providing a logical means for prioritizing these challenges based on likelihood and risk significance, and
- allowing consideration of a broader set of resources to defend against these challenges.

A risk-informed approach can be used to focus deterministic regulations by considering risk in a more coherent and comprehensive manner. By considering risk insights, operating experience, and engineering judgment, the USNRC and its licensees can focus regulatory approaches and licensee activities on those items most important to public health and safety. Where appropriate, a risk-informed regulatory approach can be used to reduce unnecessary conservatism in deterministic approaches or can be used to identify areas with insufficient conservatism and provide the bases for additional requirements. Deterministic-based regulations have been successful in protecting the public health and safety and risk insights are most valuable when they serve to focus the deterministic-based regulations and support the defense-in-depth philosophy.

1.1.1.3 Performance-Based Regulation

A performance-based regulatory approach requires at least four key elements:

- There are measurable parameters to monitor acceptable plant and licensee performance;
- Objective performance criteria are established to assess performance;
- There is licensee flexibility to determine how to meet established performance criteria;
- Failure to meet a performance criterion must not result in unacceptable consequences.

In theory, a performance-based approach can be implemented without the use of risk insights. This type of performance-based approach would require that objective performance criteria be based on deterministic analysis and performance history. This approach would provide

additional flexibility to the licensee to determine how to meet performance criteria. However, the net impact on public health and safety would be difficult to determine.

1.1.1.4 Risk-Informed, Performance-Based Regulation (RIPBR)

RIPBR uses risk insights, together with deterministic analyses and performance history, to develop measurable parameters for monitoring plant and licensee performance, as well as for developing criteria for performance assessment, and focuses on the results as the primary means of regulatory oversight. Similar to a risk-informed, deterministic-based approach, an approach based on RIPBR can be used to reduce unnecessary conservatism in deterministic approaches or can be used to support additional regulatory requirements. In addition, the approach can further focus performance-based approaches by defining the goal or purpose of the approach in terms of performance characteristics and safety significance and permitting the licensee additional flexibility in meeting the regulation. Performance-based initiatives can be considered for activities where objective performance criteria can be established for performance monitoring. Additional evaluation of performance-based approaches may result in a determination that a number of functional areas are not amenable to performance-based treatment.

1.1.2 Importance of RIPBR

The treatment of uncertainties is an important issue for regulatory decisions. Uncertainties exist in any regulatory approach and these uncertainties are the direct result of limitations in knowledge. These uncertainties and limitations existed during the development of deterministic regulations and attempts were made to accommodate these limitations by imposing prescriptive and overly conservative regulatory requirements. A probabilistic approach has exposed some of these limitations and provided a framework to assess their significance and assist in developing a strategy to accommodate them in the regulatory process.

RIPBR is widely recognized as an important step towards the elimination of the burden of the current deterministic-based, prescriptive regulatory structure and the restoration of the economic competitiveness of the nuclear power industry. Moving to a RIPBR approach will help develop more efficient and effective regulatory measures that focus directly on public safety and will provide a basis for optimizing the regulations. RIPBR will play an especially important role in the upcoming utility deregulation.

1.2 Statement of the Problem

Nuclear Power Plants (NPPs) rely on Instrumentation and Control (I&C) systems to support plant display, monitoring, control, and protection. Traditional I&C systems are analog systems based on hard-wired electromechanical and solid-state technology. Many of these analog systems are still in operation and proven to be effective. However, analog systems experience aging, e.g., drift in readings, random mechanical failures, environmental degradation. These issues inevitably increase the workload on plant surveillance and maintenance, and will lead to decreasing the overall availability of the NPPs, which, in return, will affect the competitiveness of the NPPs. On the other hand, as other industries are shifting towards digital systems, analog replacement parts are becoming more and more difficult to obtain in the market.

The new digital I&C systems have many advantages over the existing analog ones. Digital devices are essentially free of drift that afflicts analog devices, digital devices maintain their calibration better, thus reduce calibration and maintenance. Digital systems have improved system performance in terms of accuracy and computational capabilities, and they have higher data manipulation and storage capacities. As a result, the plant operational conditions will be better measured, monitored, and displayed. Digital systems can be easier to use and upgrade. A detailed comparison between digital and analog systems is provided in Appendix 1. The Canadians have extensive experiences in digital I&C in NPPs. About twenty-five years ago, their first digital system was implemented because of its better provision of on-line control of the CANDU reactors, specifically in controlling power level and xenon oscillations. In Europe, the N4 plant at the Chooz-B site in France and the Sizewell B plant in the United Kingdom have employed integrated digital I&C systems. The U.S. electric industry is approaching deregulation which allows the customers to have the option of selecting their utilities freely as they are doing in the telecommunication business. As a result, the energy utilities (including nuclear power industry) are facing fierce competition. It is under these circumstances that the U.S. nuclear power industry expects that analog I&C systems will be replaced by digital ones both in the existing NPPs and the advanced designs

which are currently under review by the USNRC. These new designs, including GE's Advanced Boiling Water Reactor (ABWR) and Simplified Boiling Water Reactor (SBWR), ABB-CE's System 80+, and Westinghouse's AP600, embody fully integrated digital I&C systems^[Chap95].

However, with the advent of the advanced digital technology, USNRC is very concerned about the potential new failure modes inherent in software. Among them are software design errors, common-cause errors, and other possible failure modes different from the hardware counterpart ([Thad93], [NRC97], [NRC95]).

Funded by Idaho National Engineering and Environmental Laboratory, this study is one prong of the three-pronged approach to the development of a methodology that would allow a regulator to use RIPBR in the licensing of software used in safety-critical nuclear power plant applications. The three throngs are

1. Identification of Issues and Selection of Case Studies;
2. Formal Method Development and Integration;
3. Formulation of Guidelines for Developing Performance Requirements.

The author's research was focused on software requirements analysis since, at this stage of the software development cycle, that is where

- the largest amount of error in software is found,
- these errors are the most expensive ones to correct, and
- few methodologies exist.

1.3 The Waterfall Model and Software Requirements Analysis

1.3.1 The Waterfall Model

As the most widely used model in software standards, the Waterfall Model of software development process is comprised of seven distinct stages/phases: planning activities, software requirements analysis, software design, software implementation, software verification and validation (e.g., testing), software installation, and operation and maintenance (Figure 1.3-1). Feedback exists between stages. Once errors are detected stemming from a previous stage, it must be fed back to that stage such that modifications can be made. A detailed description of the Waterfall Model is provided in Appendix 2.

1.3.2 Requirements Analysis

Requirements Analysis is “a process of discovery and refinement” of customer requirements^[Pres87], of “removing errors in the statement of requirements”^[Ince88]. The input to the analysis is the customer statement of requirements and the end product is a Software Requirements Specification (SRS) containing both functional and non-functional requirements. Functional requirements are statements of what a software system is supposed to do. Non-functional requirements are statements which are concerned with practical constraints upon the software developer (e.g., programming languages). The analysis requires both the developer and the customer take active roles. The customer attempts to reformulate his nebulous concept of software functions and performance into concrete details while the developer acts as interrogator, consultant, and problem solver.

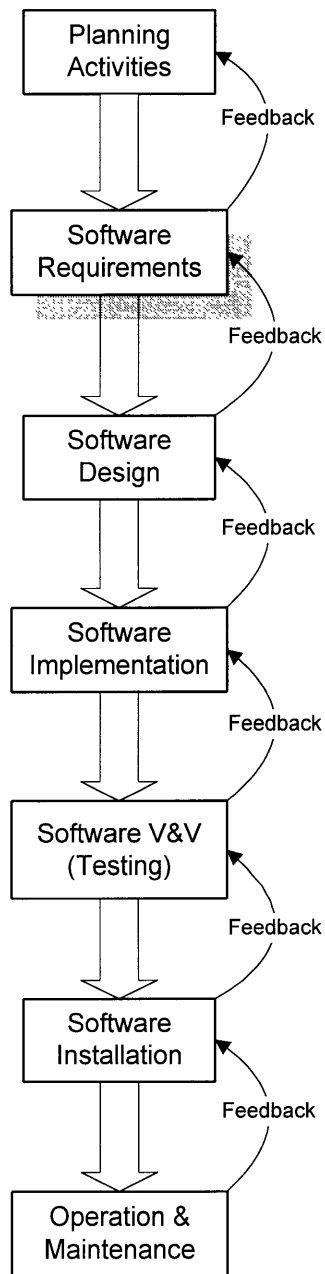


Figure 1.3-1 The Waterfall Model (refined from [Wing93], [Boeh88], [Pres87])

1.3.3 Importance of Requirements Analysis

Studies have shown that about 50% of software faults can be traced back to faulty requirements or specification. Rushby reports that^[Rush96], of 197 critical faults found during integration and system testing of the Voyager and Galileo spacecraft, only 3 were coding errors ($\approx 1.5\%$), the majority were in the requirements ($\approx 49.2\%$, mostly of which is omission in the requirements), interfaces ($\approx 24.6\%$), and design ($\approx 24.6\%$). While working on project to develop a safety-critical system, Sayet found that, of all faults detected, 46% were in the specification phase, 46% in coding, and 8% during integration test^[Saye90]. Basili reports that 48% of the errors were due to incorrect or misinterpreted functional specification or requirements^[Basi84]. Viola indicated that, in order to allow greater resolution for smaller error bars (e.g., representing number of coding errors), histogram of error diagram has to be rescaled by removing 160 errors identified from Software Requirements Specification review process^[Viol95].

More severely, these errors, if not detected and corrected early, can survive until later in the software development cycle when correcting them would be up to two orders of magnitude more expensive, or lead to project cancellation eventually. According to data presented by Fairley ([Fair85, pp.48-50]), it is 5 times more expensively to correct a requirement fault at the design phase than at requirements analysis phase, 10 times more expensively to correct at implementation phase, 20 to 50 times more expensive to correct at acceptance testing, 100 to 200 times more expensive to correct once the system is operating. According to [Zave95], requirements engineering is probably the most informal and subjective of all the areas in computer science.

1.3.4 Inadequacy of Natural Languages in Documenting Software Requirements

Some of the reason as to why more errors tend to occur in the requirements or specification phase are as follows. First, the customer usually can not describe precisely what the software is supposed to do. Second, inadequate communication between the customer and the software

developer may result in a misunderstanding or misinterpretation of the requirements. Third, the customer's requirements are constantly evolving as the project progresses. Fourth, the software requirements written in natural language by the customer may be ambiguous, inconsistent, and/or incomplete. For example, Parnas presents four different interpretations of a requirement expressed in English concerning water level^[Parn91]. The original requirement is

“Shut off the pumps if the water level remains above 100m for more than 4s.”

Four interpretations of this requirement are

- “Shut off the pumps if the *mean* water level remains above 100m for more than 4s.”
- “Shut off the pumps if the *median* water level remains above 100m for more than 4s.”
- “Shut off the pumps if the *root-mean-square* water level remains above 100m for more than 4s.”
- “Shut off the pumps if the *minimum* water level remains above 100m for more than 4s.”

The second example describes the requirements of software to store, retrieve, add and delete telephone numbers from the telephone directory of a city. Citing this example, Butler shows the inadequacy of natural languages as a medium of recording formal specification^[Butl93] in terms of addressing completeness and consistency issues. He concludes that “the process of translating the requirements into mathematics has forced us to enumerate many things that are usually left out of English specifications” and “the formal process exposes ambiguity and deficiencies in the requirements” of natural languages.

These examples and other experiences have demonstrated the inadequacy of natural languages in documenting precise requirements specification (due to their inherent ambiguity), especially in safety- and mission- critical applications such as nuclear reactor control and protection systems, airplane flight controller, crucial financial system.

1.3.5 Attributes of Software Requirements Specification

As the output of software requirements analysis, software requirements specification (SRS) is the highest-level specification of a software component in a system. An SRS specifies what a software component must do and what it must not do (e.g., completeness). Seven attributes are identified in [USNRC95] for the SRS and are summarized here.

- *Consistency*—the requirements do *not conflict* with each other within an SRS;
- *Completeness*—an SRS should include *all* necessary requirements, define software response to all realizable classes of input data in all realizable class of situations, specify response to both valid and invalid input values;
- *Unambiguity*—the requirements are interpreted the same way by all readers, that is, every requirement has one and only *one interpretation*;
- *Verifiability*—a practical method or process exists (inspection, demonstration, analysis, or testing) to verify that an implementation fulfills the requirements;
- *Readability*—a reader can easily read and understand all the requirements, i.e., formality should not compromise readability and understandability;
- *Modifiability*—the requirements are easy to modify correctly, and
- *Traceability*—the origin of each of its requirements is clear and it facilitates the referencing of each requirement in future development or enhancement documentation. Both forward and backward traceability should be provided.

The work in this dissertation will be focused on requirements analysis, especially on consistency, ambiguity and completeness of software requirements. The research leads to the development of a Tree-Based Graphical Approach in software requirements analysis.

Chapter 2

Current State of the Art and Practice

2.1 Formal Methods

2.1.1 Definition

The function of mathematical logic in digital systems is analogous to that of calculus in the traditional, continuous systems: as an analytical tool calculating and predicating the behavior of the systems as well as being a notation describing the systems. Rushby defines *Formal Methods* as “methods that use ideas and techniques from mathematical or formal logic to *specify* and *reason about* computation systems (both hardware and software)”^[Rush95]. The term “formal” in the terminology refers to the idea that a proof can be known to be valid based on its “form” of reasoning, that is, the validity of a proof can be established by simply checking the syntax of an argument regardless of its semantics^[Butt93, Holl95]. By contrast to fault removal and fault tolerance, formal methods are fault avoidance techniques that will increase dependability of software by avoiding errors throughout the software development cycle.

In the 1970s, formal methods were focused on proofs of program correctness^[Hoar69, Hoar71, Lond75], which turned out to be not only an expensive exercise, but a very difficult one. Since the 1980s, emphasis of formal methods has been shifting from program correctness to the use of formalism in specifications. It is realized that the majority of errors in the software can be traced back to the early phases of software development process, and the conventional techniques such as code review and testing are very effective removing coding errors^[Rush93b]. More recently, with the development of efficient techniques for model checking, attention of formal methods is being paid to large-scale mechanized model checking. However, due to limited expressiveness of temporal logic, model checking is prevented from fully characterizing the functionality required of a system. By contrast, emphasis is focused on the

important properties that the system should possess. Due to its excellent diagnostic ability in the form of counterexamples, model checking is usually focused on incorrectness of a program, e.g., finding errors, rather than proving its correctness.

Since no single tool or method is universal enough to support all the diverse applications of formal methods to software, a spectrum of them have been developed (see Section 2.3 and Appendix 4).

2.1.2 Categorization of Formal Methods

The definition of formal methods implies that the two primary components of formal methods are *Formal Specification* (e.g., “to specify”) and *Formal Verification* (e.g., to “reason about”). Miner defines formal specification as “use of notations derived from logic to describe assumptions about the world in which a system will operate, requirements that the system is to achieve, and the intended behavior of the system”^[Mine95]. He also defines formal verification as “use of formal logic to analyze specifications for certain forms of consistency, completeness, to prove that specified behavior will satisfy the requirements given the assumptions, to prove that a more detailed design implements a more abstract one.”

Formal specification languages can be divided into four categories (refer to Figure 2.1-1). The first two categories are suitable for the sequential systems while the next two categories are proper for the concurrent systems.

- **Model-Oriented Specification Language** The category models a system’s desired behavior in terms of abstract mathematical objects such as sets, sequences (or lists), functions. Specification languages VDM and Z belongs to this category;
- **Property-Oriented Specification Language** The category can further be subdivided into axiomatic methods which use first-order predicate logic to express pre- and post-conditions of operations over abstract data types, and

algebraic methods based on multi-sorted algebra and relate properties of the system to equations over the entities of the algebra;

- **Process Algebra Specification Language** This category is applicable to the concurrent system through describing their algebra of communication processes. According to Rushby, this is the model-oriented counterpart for the concurrent system^[Rush93b];
- **Temporal Logic Specification Language** This category makes use of logic designed to reason about time^[Ladk87, Whit90, Alle83, Bouz95, Kaut91]. According to Rushby, this is the property-oriented counterpart for the concurrent system^[Rush93b].

2.1.3 Taxonomy of Degrees of Rigor in Formal Methods

Rushby classifies formal methods into four levels of different rigor¹ according to how and what formal methods are used^[Rush93b]. This taxonomy is cited by [USNRC95].

- **Level 0 — No Use of Formal Methods**

Level 0 corresponds to the mainstream of the current industrial practice in which verification is a manual process of review and inspection applied to documents written in natural languages, pseudocodes, or programming languages. Validation is based on testing driven either by software requirements specification or the structure of the program.

¹ Butler seems to have a different taxonomy. In [Butl93], he defines

- Level 1 as “Formal Specification of All or Part of the System”,
- Level 2 as “Paper and Pencil Proof of Correctness” (which is consistent with Miner’s definition, “specification using a formal specification language with manual proofs”^[Mine95]),
- Level 3 as “Formal Proof Checked by Mechanical Theorem Prover”.

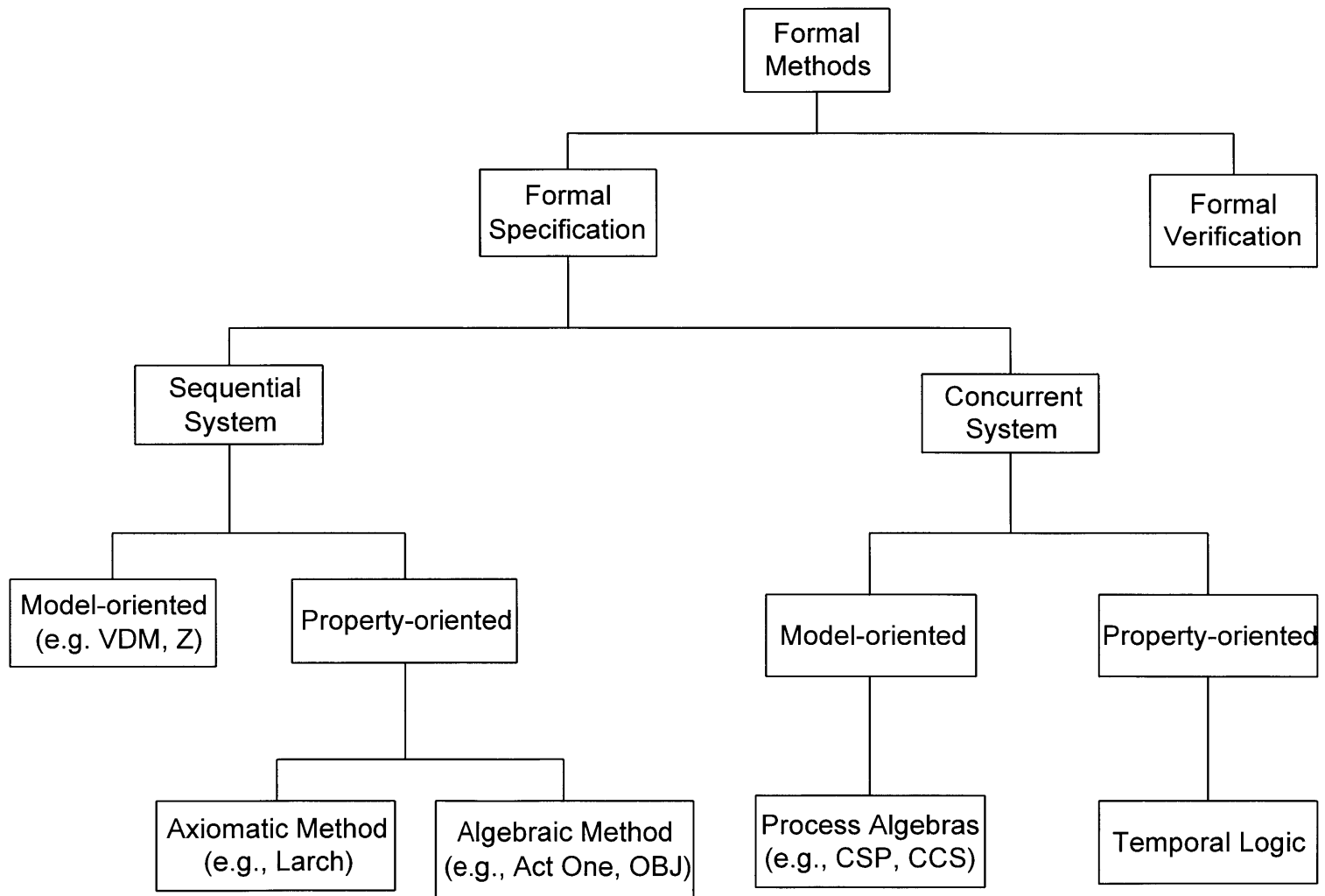


Figure 2.1-1 Formal Methods and the Hierarchical Components (based on [Hinc93], page 29 of [Rush93b])

- Level 1 — Use of Concepts and Notation from Discrete Mathematics

The idea of Level 1 rigor of formal methods is to replace some of the natural language used in the software requirements specification with notations from logic and discrete mathematics. This level of rigor matches the way mathematics is used in most other engineering disciplines. Hence, this level of rigor is most liable to be accepted by the practitioners. Use of discrete mathematics at this level provides several benefits. First, the versatile collection of building blocks introduced in discrete mathematics can help in the development of clear, precise, and systematic statements of requirements, specifications, and designs. Second, discrete mathematics provides a compact set of notations that allows descriptions of requirements, specification, and design to be recorded and communicated with much less ambiguity than natural language such that the recorder and the reader can share the unique understanding of the notations. Third, axioms, theorems, and inference rules can systematically guide the elaboration of the specification or design from one level to another. Software Cost Reduction (SCR) introduced in Section 2.3.1 is an example of Level 1 formal method. Level 1 formal method is what this dissertation is intended to address. The purpose choosing this level of rigor is two fold. First, the proposed methodology should rely on minimal knowledge of formal logic and notations. Second, such a process helps analyst gain more insight than running a commercial package as a black box. If software requirements are proved inconsistent by a commercial package, a message will be displayed. However, no hint will be given as where the inconsistency is, nor will suggestions be given as how to remove the inconsistency through modifying the requirements.

- Level 2 — Use of Formalized Specification Languages with Some Mechanized

- Support Tools

In addition to retaining the benefits obtained from Level 1 applications of formal methods, Level 2 applications of formal methods may contain additional benefits. First, specification languages not only provide a standard

set of notations from discrete mathematics, but also address software engineering concerns and allow specifications to be structured into units (e.g., modules, abstract data types) with explicitly specified interface. Second, mechanized tools such as syntax checkers or type checkers allow more efficient detection of certain types of faults or make it easier to be the “living” documentation. Third, it is possible to generate a prototype implementation from a specification to explore certain properties of the specification.

However, Level 2 specification languages may be optimized for a specific domain of applications and cumbersome or inapplicable in other domains. As a result, they may lead to implementation bias as means of specification.

- Level 3 — Use of Fully Formal Specification Languages with Comprehensive Support Environments, Including Mechanized Theorem Prover or Proof Checker

Using a specification language with formal proof techniques, this level of rigor has the highest level of rigor. Once the model is completely formalized, i.e., reduced to symbol manipulation, it is possible to mechanize proofs in terms of proof checker², theorem prover³ or, most commonly, something in between^[Gro94]. The advantages of this approach are that requirements, specification, and designs can be subjected to searching examination, and that mechanization eliminates faulty reasoning with higher certainty than the paper-and-pencil approach. But the fully formal specifications and automatic proofs are expensive to develop, usually with rather impoverished notation based on restricted logic for the sake of mechanized support. Therefore, one of the barriers of Level 3 methods is the readability and understandability associated with level of rigor. In general, the higher the level of rigor, the higher the cost. Depending on the benefits desired from the formal methods, the criticality of the application, and the resources available, higher levels of rigor of formal methods are not necessarily superior to lower level ones.

² Proof checker is a computer program that checks the steps of a proof proposed by a human being.

³ Theorem prover is a computer program that attempts to discover proofs without the help of human being.

2.1.4 Formal Methods and the Industry

Most of the formal methods have been developed by computer scientists and are aimed at computer scientists or others with very strong background in discrete mathematics. As a result, when formal methods are developed by computer science community, they are usually difficult to be accepted by the practicing engineers due to the poor readability inherent in the methods. There is still a long way before formal methods are universally accepted by the industry.

2.1.4.1 Advantages of Formal Methods

- Experience gained from projects such as IBM's CICS shows that formal methods do not necessarily increase the development cost, nor do they necessarily cause delay^[Hinc93, Bowe95]. It is true that cost does increase in the initial phases of the development process. However, benefit pays off later as coding and maintenance costs are reduced significantly. As a result, the overall development of software costs will be decreased.
- Even though complete formal development (i.e., including proof and refinement) requires a strong mathematical background, understanding and writing formal specification requires relatively less knowledge of mathematics.
- Formal methods are not only beneficial to the safety-critical systems. In fact, formal specifications help with any system^[Hall90].

2.1.4.2 Disadvantages of Formal Methods

- Obstacles exist to getting started with formal methods. These obstacles include
 - lack of readability and understandability and insufficient education;
 - immaturity of tools and difficulty in choosing the proper tool. Many tools in existence are domain specific, research prototypes not mature enough for industry application;

- Formal methods can not guarantee error-free software. Although formal methods enable us to check rigorously for consistency in requirements and reason about the properties of the requirement, human-generated specifications or proofs are error-prone and formal methods do not check if requirements themselves are correct^[Hinc93, Hall90].
- Formal methods do not replace traditional software engineering methods such as testing^[Hinc93, Bowe95].
- Errors may exist in the formal verification tools themselves.

For more information, interested readers in this topic should refer to [Hall90], [Bowe95], [Saie96], [Rose96], [Knig97], [Webe93], [Crai93], [Crai95].

2.2 Finite State Machine (FSM)

Used in many branches of computer science, a Finite-State Machine (FSM)^[Rose95] is formally defined as a six-tuple⁴: $M = (S, I, O, f, g, s_0)$ comprised of a finite set of states S , a finite input set I , a finite output set O , a transition function f that assigns a new state to each state and input pair, an output function g that assigns an output to each state and input pair, and an initial state s_0 . Two basic types of FSM are the *Mealy machine* and the *Moore machine*. For a Mealy machine, output is determined by both current inputs and current state. For a Moore machine, on the other hand, output is solely determined by the current state. The behavior of such a machine can be specified by giving rules of transitions between states and the set of output values corresponding to each state.

FSM can be expressed using *State Table* to represent the transition function and output function for all pairs of states and input. FSM can also be represented by *State Diagram*, i.e., directed graph with edges labeled as (input, output) pairs and arrows indicating respective state transition. The advantage of the State Diagram is that the state transition and output functions are expressed clearly on the graph, but the clarity vanishes when the FSM has many states (e.g., 10 or more) since the state transition curves will inevitably overlap with each other, making it difficult to keep track of. For a deterministic FSM with n states, there could be as many as n^2 state transition edges. On the other hand, the State Table is advantageous when there are many states for an FSM. For each additional state, we can simply increase the size of the table (e.g., number of rows or columns). The disadvantage of the State Table is that, it is not as informative as the State Diagram in terms of state transition and output functions. Hence, the State Table and State Diagram are complementary of each other.

It should be noted that the State Diagram looks very similar to the diagram of discrete Markov model^[Rama93] in stochastic analysis, e.g., probability and reliability theory. But the

⁴ In [Lync96], a State Machine is defined to be a triple (Q, Q_0, δ) , where

- Q is non-empty (finite or infinite) set of *states*,
- Q_0 is a non-empty subset of the states known as *start states*,
- $\delta \subseteq Q \times Q$ is a binary relation over Q known as the *transition relation*.

two diagrams have fundamental differences. In Markov model, edges are labeled with the probability of transitions from one state to another. The probability (or rate) is usually not unity, so transition from one state to others is random. Markov model usually doesn't touch upon input and output functions. In State Diagram, however, edges are labeled with ordered (*input, output*) pair, indicating that the transition occurs from one state to another with the *input* and generates the *output*. For a deterministic FSM, the transition will occur with probability of unity.

An example of Mealy machine with $S = \{s_0, s_1, s_2, s_3, s_4\}$, $I = \{0,1\}$ and $O = \{0, 1\}$ is shown in Table 2.2-1 (State Table of the FSM). To read the table, the first row means that, if the FSM is in state s_0 with input 0 at time t_n , the FSM will be in state s_1 with output $g=1$ at time t_{n+1} ; if the FSM is in state s_0 with input 1 at time t_n , the FSM will be in state s_4 with output $g=0$ at time t_{n+1} . Figure 2.2-1 shows an equivalent FSM expressed in State Diagram. In the diagram, states are labeled inside the circles with (input, output) pairs labeled on the edges whose directions represent state transitions.

Table 2.2-1 An Example of Mealy Machine

State	Input			
	0		1	
	f	g	f	g
s_0	s_1	1	s_4	0
s_1	s_1	1	s_2	1
s_2	s_4	0	s_3	0
s_3	s_4	0	s_3	0
s_4	s_1	0	s_0	0

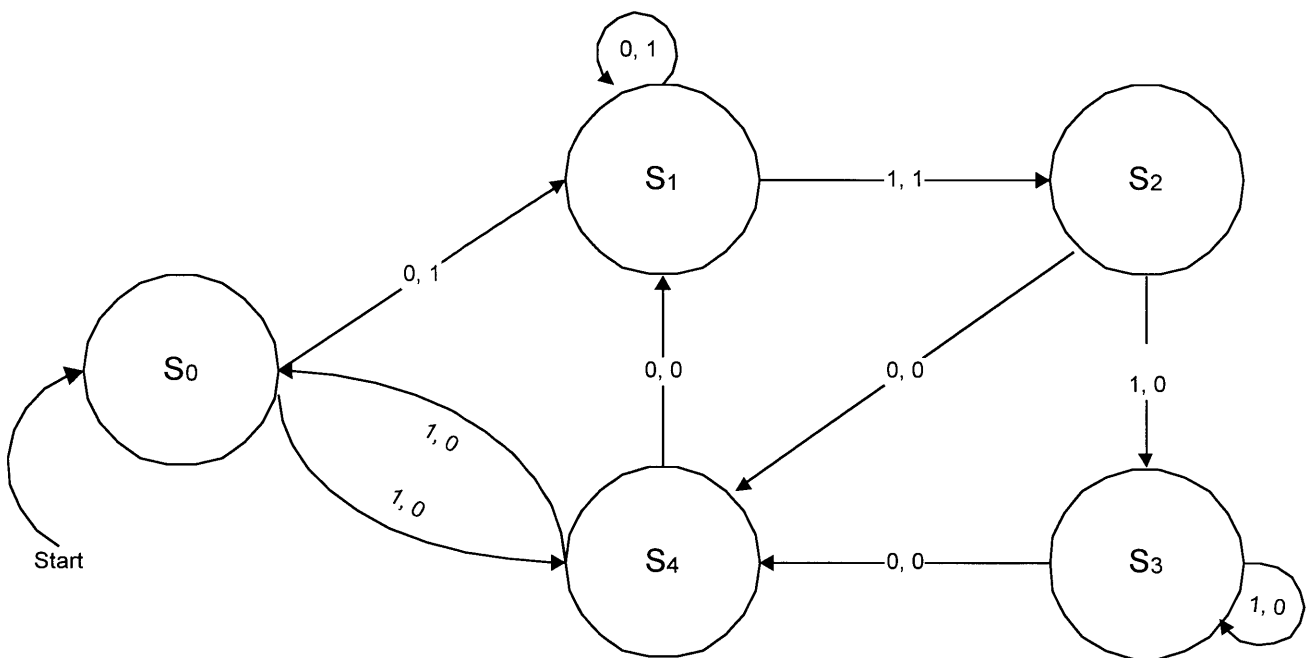


Figure 2.2-1 A Graphical FSM Equivalent to Table 2.2-1

2.3 An Overview of Methodologies on Software Requirements Analysis

2.3.1 Software Cost Reduction (SCR)

2.3.1.1 SCR Tabular Notation

SCR is the abbreviation for Software Cost Reduction, a formal notation developed by Parnas and others at the US Naval Research Laboratory between the late 1980s and the early 1990s^[Parn90a, Parn91, Parn95]. The concept of the methodology can be traced back to as early as the 1970s^[Taus77]. The technique was tested and improved in a project involving an experimental redesign of the Onboard Operational Flight Program for the United States Navy's A-7E aircraft^[Will94].

Realizing that the conventional formal methods were mathematically too sophisticated, Parnas and his colleagues decided to seek a “middle road” to bridge the gap between the theorists and the practitioners. The SCR specification language they developed makes use of tabular notation based on finite-state machine model, e.g., via modes (states), condition statements, and action statements. It classifies all data items and identifiers according to type (e.g., input variable, mode, mode class) and uses special bracketing symbols to denote the type associated with an identifier although one does not have to use the brackets as in [McDo]. Besides, SCR can represent concurrency and temporal characteristics of a system^[Will94].

2.3.1.2 An Example in SCR Notation

SCR has been successfully employed in a variety of practical systems, including avionics systems (e.g., a Shuttle CR for the “Heading Alignment Cylinder”, in which SCR reveals that several rows in one table have inconsistent or incomplete conditions^[Rush97]), the A-7 Operational Flight Program, a submarine communications system^[Heit83], and safety-critical systems of two Nuclear Power Plants: the Darlington plant in Canada^[Joan, Joan92, Viol95, Viol96] and one in Belgium^[Cour93]. More recently, a version of the SCR method called CoRE was used to document the requirements of Lockheed's C-130J Operational Flight Program^[Heit95a].

Equipped with Canada's first computerized emergency reactor shutdown systems, Darlington Nuclear Power Generation Station is a 4×880 MWe CANDU type^[Petr91] nuclear power plant located east of Toronto. Each of the reactor has two independent shutdown systems: SDS1 control rod system and SDS2 liquid boron system. In addition to 6,000 lines of assembly code in SDS1 and SDS2 combined, SDS1 has 7,000 lines of Fortran and SDS2 has 13,000 lines of Pascal^[Crai94]. Application of SCR notation to Darlington include three main steps:

- formalizing informal requirements by generating specification tables;
- use of existing code to develop program-function tables for it;
- demonstrating that the code is consistent with the specifications by comparing the two tables developed above.

An example of a trip meter logic is given here in SCR notation. The code segment in Figure 2.3-1 is taken from [Viol96]. Software Design Document (SDD) is generated in SCR notation by reverse-engineering the code segment. The reverse-engineered SDD is shown in Table 2.3-1. Completeness of the requirements is achieved by developing a table with mutually exclusive, collectively exhaustive columns and rows. When the conditions defined in the rows and columns are satisfied, the corresponding actions described in the requirements will be taken. Table 2.3-1 (A) specifies the new state of a reactor trip meter (*trip_state'*) with respect to its current state (*trip_state*), the sensors reading (*signal*), trip setpoint (*setpoint*), and its deadband (*db*). To specify the requirements in English, if the measured *signal* is below *setpoint-db*, then, no matter what the current trip state is, the trip meter will be in the *not_open* status (i.e., no trip initiated); if the *signal* is between *setpoint-db* and *setpoint*, then, the new meter state will remain in its current state; if the *signal* is above the *setpoint*, the trip meter will be in *open* state (i.e., trip initiated) regardless its current state. Table 2.3-1 (B) is an equivalent mutation of Table 2.3-1 (A). In the table, completeness is achieved by exhaustively enumerating the mutually exclusive columns named 1, 2, 3, and 4. Note that columns 1 and 4 each represent two columns hidden under the *don't care* notation “–”. When the conditions defined in the “Condition Statements” are satisfied (i.e., columns), the actions will be taken in the corresponding “Action Statements”.

```

void determine_trip_state (void)

Declarations ...

sequence_check(p_trip);

signal = get_signal();
setpoint = get_setpt();

IF ((signal < sig_min) || (sig_max < signal))
    fatal_error (90);
If ((setpoint < setpt_min) || (setpt_max < setpoint))
    fatal_error (91);

new_trip = tripped;
IF ((signal <= (setpt-db)) || ((signal < setpoint) && (trip_state ==
not_tripped))) new_trip = not_tripped;
IF (signal >= setpoint)
    new_trip = tripped;
trip_state = new_trip;

```

Figure 2.3-1 Code Segment from [Voil96]

Table 2.3-1 The SCR Notation for a Trip Meter

(A) Tabular Notation for the Last Section in Figure 2.3-1

<i>trip_state</i> '	$signal \leq \text{setpoint-db}$	$\text{setpoint-db} < signal < \text{setpoint}$	$signal \geq \text{setpoint}$
<i>trip_state</i> =open	not_open	open	open
<i>trip_state</i> =not_open	not_open	not_open	open

(B) An Equivalent of (A)

Condition Statements	1	2	3	4
a: $signal \leq (\text{setpoint-db})$ b: $(\text{setpoint-db}) < signal < \text{setpoint}$ c: $signal \geq \text{setpoint}$	a	b	b	c
<i>trip_state</i> = open	—	T	F	—
Action Statements				
<i>trip_state</i> ' = open		x		x
<i>trip_state</i> ' = not_open	x		x	

2.3.1.3 Tool Support for SCR Notation

In the Darlington case, the SCR notation was used to convince the Atomic Energy Control Board of Canada (AECB), the Canadian nuclear regulatory agency, that the software in the shutdown systems was of acceptable quality and in accordance with its specification. Even though the control software of the two shutdown systems at Darlington is not big, application of formal method turned out to constitute a substantial effort due to lack of tool at that time. All the verification activities were done manually and some 30 engineers were involved in different aspects of verification. The products include the documentation of specification, code, and proofs as thick as “25 three-inch binders” (≈ 1.90 m) for each system. As observed by Parnas at the certification of the plant that the “reviewers spent too much time of their time and energy checking for simple, application-independent properties”, rather than “more difficult safety-relevant issues”^[Heit95b], the cost-effectiveness of SCR was hindered by lack of tool support.

More recent work on SCR is emphasized on developing supporting tools^[Heit95a, Heit95b, Shen96, Sree96]. Heitmeyer^[Heit95a, Heit95b] is developing toolset that includes a *specification editor* for creating and editing formal requirements specification, a *consistency checker* testing the specification for selected properties, a *simulator* for symbolically executing the specification, and a *verifier* for checking that the specification satisfies selected application properties; Shen focuses on developing tools to transform (normal) tables into inverted tables and vice versa^[Shen96]; Sreemani attempts to transform an SCR specification into a format that Symbolic Model Verifier (SMV) can process, and claims that, if properly implemented, SMV could save time significantly, e.g., more than two weeks computer running can be reduced to 15 CPU seconds^[Sree96]. For more information on SCR tools, readers should refer to the proceedings of a 1996 workshop on tabular notations^[Work96].

2.3.1.4 A Counterexample for SCR Notation

SCR Tabular Notation makes it more user-friendly to analyze consistency and completeness of software requirements specification or software design document. However, one should

bear in mind that tabular notation is good only in situations in which less than three variables are involved. The following example is a segment of code written in the high level programming language C or C++. Using tabular notation, our task is to back out the design document from the code so that it can be checked against the initial design document.

```
{
    ...
    if (a==3) result =11;
        else if (b==8) result =12;
            else if (c<9) result =13;
                else if (d>7) result =14;
                    else result =15;
    ...
}
```

Since tables are usually two-dimensional, they are most suitable in handling two independent variables. When more variables are involved (e.g., four in this case: a, b, c, d), we have to embed one table into another or create a super-table. Table 2.3-2 (A) is such a super-table, which lists all the 16 ($=2^4$) possible combinations. The disadvantage of the table is that all the combinations leading to the same result must be listed explicitly, e.g., eight combinations listed lead to one single result (*result=11*). Table 2.3-2 (B) is somehow an improvement over Table (A) in that only combinations that lead to different results are listed as columns of Table (B). Its disadvantage is that each condition at each column must be marked individually.

Table 2.3-2 (A) Corresponding Super-Table of the Code Segment

		$c < 9$		$c \geq 9$	
		$d > 7$	$d \leq 7$	$d > 7$	$d \leq 7$
$a = 3$	$b = 8$	$result = 11$	$result = 11$	$result = 11$	$result = 11$
	$b \neq 8$	$result = 11$	$result = 11$	$result = 11$	$result = 11$
$a \neq 3$	$b = 8$	$result = 12$	$result = 12$	$result = 12$	$result = 12$
	$b \neq 8$	$result = 13$	$result = 13$	$result = 14$	$result = 15$

(B) An Equivalent of (A)

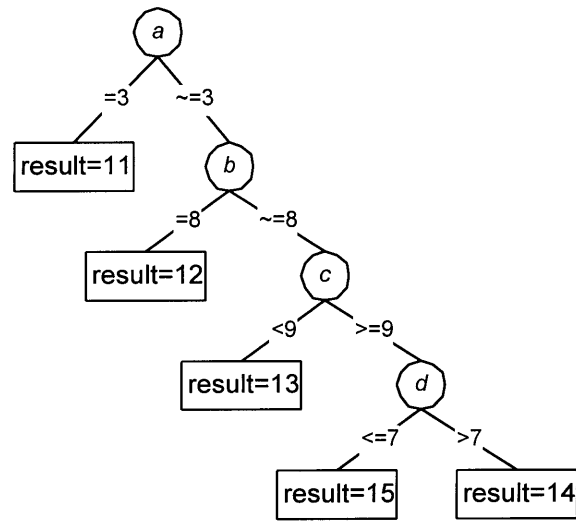
Condition Statements						
Combination		1	2	3	4	5
a	$=3$	x				
	$\neq 3$		x	x	x	x
b	$=8$	—	x			
	$\neq 8$			x	x	x
c	<9	—	—	x		
	≥ 9				x	x
d	>7	—	—	—	x	
	≤ 7					x
Action Statements						
$result =$		11	12	13	14	15
Caveat		“—” represents “don’t-care” case, “x” represents case that condition is satisfied.				

A better notation to document specification of this segment of code is the a Tree-Based Graphical Method developed in Appendix 7. In the tree developed in Figure 2.3-2, root and the internal nodes represent independent variables (in circle), the edges at a level represent a partition of the domain of an variable. The partition is labeled on the edges, and each leaf (in rectangle) represents the action to be taken if the conjunction of the conditions from the root down to the parent of that leaf is satisfied. For example, the leaf “**result=13**” is initiated if $a \neq 3$ and $b \neq 8$ and $c < 9$ are satisfied. By comparing Table 2.3-2 with Figure 2.3-2, it is obvious that the graphical method is logically more intuitive and informative than the tabular notation.

2.3.2 Requirements State Machine Language (RSML)

2.3.2.1 RSML and Its Tool Support

Modugno, Leveson and others^[Modu96] have developed a state-based specification language called Requirements State Machine Language (RSML). RSML is based on the finite state Mealy machine with outputs on transitions between states (refer to Section 2.2). Each RSML transition consists of a source state, a destination state, a triggering event, a guarding condition based on the triggering event, and a possible output action. A transition will take place and may generate an output action if the triggering event occurs and the guarding condition is evaluated to be *true*. The guarding condition is expressed in a tabular representation of disjunctive normal form named *AND/OR* table (e.g., “sum of product”), whose leftmost column lists the logical expressions. Each of the other columns in the *AND/OR* table represents a conjunction (AND) of the rows in leftmost column, the disjunction (OR) of the columns other than the leftmost represents the guarding condition (see Figure 2.3-3). RSML has been used in an aircraft Traffic Collision Avoidance System (TCAS II)^[Leve94a]. According to [Leve97a], a toolkit called SpecTRM is under development.



"~" represents logical "not".

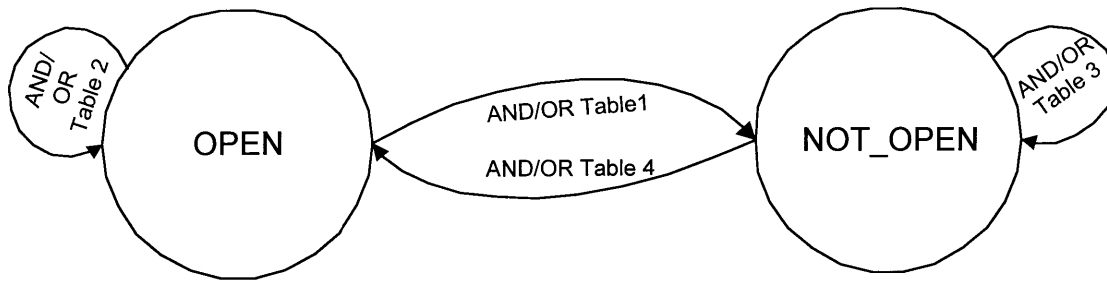
Figure 2.3-2 The *M*-ary Tree-Based Graphical Method

2.3.2.2 RSML and SCR

Using RSML notation, the same example as in Section 2.3.1.2 is expressed in Figure 2.3-3. In the figure, four transitions are identified, namely OPEN to NOT_OPEN, OPEN to OPEN, NOT_OPEN to NOT_OPEN, and NOT_OPEN to OPEN. The guarding conditions to these transitions are represented in AND/OR Tables 1 through 4, respectively.

- AND/OR Table 1 specifies that the transition of `trip_state` from OPEN to NOT_OPEN will be initiated if `signal` is less than or equal to `setpoint-db`;
- Table 2 specifies that the transition of `trip_state` from OPEN to OPEN will be initiated if either `signal` is greater than `setpoint-db` and less than `setpoint` or `signal` is greater than or equal to `setpoint`. This guarding condition can be simplified to “`signal` is greater than `setpoint-db`”;
- Table 3 specifies that the transition of `trip_state` from NOT_OPEN to NOT_OPEN will be initiated if either `signal` is less than or equal to `setpoint-db` or `signal` is greater than `setpoint-db` and less than or equal to `setpoint-db`. This guarding condition can be simplified to “`signal` is less than `setpoint`”;
- Table 4 specifies that the transition of `trip_state` from NOT_OPEN to OPEN will be initiated if either `signal` is greater than or equal to `setpoint`.

It can be shown that Figure 2.3-3 is logically equivalent to Table 2.3-1. However, AND/OR tables in RSML emphasize on state transitions (which are referred to as Action Statements in SCR notation) while tables in SCR emphasize on the partitions of variables or “pre-conditions” (the Guarding Conditions in RSML notation). AND/OR tables become complex in the case when more than two logical variables are involved in guarding conditions (similar to SCR notation) or more states are involved in state transitions.



1) Transition (trip_state): OPEN → NOT_OPEN

Triggering Event: signal

Condition: AND/OR Table 1

signal ≤setpoint-db	OR
setpoint-db< signal <setpoint	T
setpoint≤ signal	F
	F

2) Transition (trip_state): OPEN → OPEN

Triggering Event: signal

Condition: AND/OR Table 2

	OR	
signal ≤setpoint-db	F	F
setpoint-db< signal <setpoint	T	F
setpoint≤ signal	F	T

3) Transition (trip_state): NOT_OPEN → NOT_OPEN

Triggering Event: signal

Condition: AND/OR Table 3

	OR	
signal ≤setpoint-db	T	F
setpoint-db< signal <setpoint	F	T
setpoint≤ signal	F	F

4) Transition (trip_state): NOT_OPEN → OPEN

Triggering Event: signal

Condition: AND/OR Table 4

signal ≤setpoint-db	OR
setpoint-db< signal <setpoint	F
setpoint≤ signal	F
	T

Figure 2.3-3 RSML Specification of the Example in Table 2.3-1

2.3.3 PVS Verification System

Based on years of experience at SRI International in building and using tools to support formal methods, PVS provides mechanized support for formal verification as well as formal specification.

The specification language of PVS is based on classical, typed higher-order logic. The basic types include the built-in types (boolean, integers, and real, etc.), uninterpreted types introduced by the user. The type-constructors include function, set, tuple, record, enumeration, and recursive abstract data types (binary trees, lists, etc.). PVS specifications are organized into parameterized theories that may contain assumptions, definitions, axioms, and theorems. PVS expressions provide the usual arithmetic and logical operators, function application, lambda abstraction, and quantifiers. Tabular specifications such as SCR are supported with automated checks for disjointness and coverage of conditions^{[Owre96], [Rush93a]}.

The PVS theorem prover provides a collection of powerful primitive inference procedures applied interactively under user guidance with a sequent calculus framework. The inferences include propositional and quantifier rules, induction, rewriting, and decision procedure for linear arithmetic. User-defined procedures can combine these primitive ones to generate large proofs and yield higher-level proof strategies. PVS includes a decision procedure for the relational μ -calculus.

PVS is mainly intended for the formalization of requirements and design-level specifications, and for the analysis of intricate problems. It has been applied to algorithms and architecture for fault-tolerant flight control systems, and to problems in hardware and real-time system design. Alpha-release of the current version of PVS is available through internet^[SRI97], but documentation is not well organized, nor is it fully up to date. Unfortunately, specific training is required to learn PVS notations.

2.3.4 Quasi-Classical (QC) Logic

To deal with inconsistent requirements, Besnard^[Besn95] has developed a Quasi-Classical (QC) logic which is essentially first-order propositional logic (see Chapter 3) that allows the derivation of non-trivializable classical inferences. The inference rules of QC-logic are a subset of the classical inference rules. Hunter^[Hunt97] later improved the QC-logic (and named it “labeled QC-logic”) by prefixing labels to logic formulae so that it was possible to pinpoint or track inconsistency in the requirements. However, QC-logic does not reduce the dependency on knowledge of discrete mathematics.

2.3.5 Fuzzy Logic and Utility Function

Yen and others introduced an interesting concept into requirements analysis by developing a quantitative measure analyzing the tradeoffs between conflicting requirements using techniques from decision sciences (e.g., fuzzy logic and utility functions). They argue that each conflicting requirement can be satisfied to *some degree*, rather than resolving the conflict. This idea may be appealing in non-safety-critical applications (e.g., developing word processing software), so that priorities can be assigned to requirements, assuming certain features are more important than others and features with lower priorities can be ignored if deadline approaches. However, according to a survey of 23 projects in 10 companies by Lubars^[Luba93], no company knew “how to assign and modify priorities or how to communicate those priorities effectively to project members”. For safety-critical applications, such as I&C in NPPs, such a prioritization of requirements is not applicable.

2.3.6 Structured Analysis and Design Technique (SADT)

Developed SofTech Inc., Structured Analysis and Design Technique (SADT)^[Dick78, Ross77a, Ross77b] is made up of a hierarchical diagram, each of which is a network of boxes representing activities. The arrows at four sides of a box represent input, output, control and mechanism for the activity involved. However, SADT was only designed for manual application (e.g., with no more than 40 primitive constructs), and it does not provide rules to check consistency and completeness of an SRS.

2.3.7 Systematic Activity Modeling Method (SAMM)

Systematic Activity Modeling Method (SAMM)^[Lamb78] was developed by Boeing Computer Service Co.. Expressed in hierarchy, a typical SAMM specification is comprised of a context tree (i.e., labeled trees), activity diagrams (i.e., directed graphs) and condition charts (e.g., similar to decision tables).

2.3.8 Higher Order Software (HOS)

Higher Order Software (HOS)^[Ham76] is a requirements specification language developed by Higher Order Software Inc. to support entire systems development process. A system is represented by a binary tree, each of whose node is a mathematical function with input labeled to the right-hand side of the node and output to its left. According to [Ouya95], HOS is no longer commercially available today and is replaced by Development Before The Fact (DBTF) which is introduced in Appendix 4.

2.3.9 Requirements Statement Language (RSL)

Developed by TRW Defense and Space Systems Group, a Requirements Statement Language (RSL)^[Alfo77, Alfo85] specification specifies software requirements in terms of processing paths. The paths are represented graphically by requirements nets (or R-nets) which are essentially flow diagrams with logical connectives. Although developed hierarchically, the final SRS is a single, flat diagram by eliminating the intermediate steps of refinement and decomposition. As a result, the final SRS for a sizable problem will be very huge.

2.4 Summary

Most of the formal methods have been developed by computer scientists and are aimed at professionals with strong background in discrete mathematics, rather than engineers. As a result, when they are developed, they are usually difficult to be accepted and used by engineers due to its poor readability or intensive training required. Parnas criticizes this situation by saying that it “does no good to prove that a piece of software satisfies a specification, if that specification cannot be read, understood, and criticized by potential users of their representatives”^[Pam93b]. Therefore, formal methods must have two attributes: they must be formal, and they require minimal training.

None of the methods described above indicates the importance of conformance to physics laws in requirements analysis, which the author has discovered to be very essential in safety-critical systems such as nuclear reactor control. The author’s approach to attack the problem is to develop a user-friendly methodology from a practicing engineer’s point of view.

Chapter 3

Fundamentals of Mathematical Logic

Discrete mathematics is a branch of mathematics devoted to the study of discrete objects. It covers a wide range of topics, including mathematical logic, boolean algebra, mathematical reasoning, counting techniques, graph theory, data structure, etc. A key reason that discrete mathematics is of increasing importance is that information is stored and manipulated by computers in a discrete fashion. Discrete mathematics also provides the mathematical foundations for formal languages.

Formal methods are grounded in mathematical logic and reasoning, which provide precise and unambiguous meaning to a mathematical statement. A logical proof system consists of: (1) language for stating propositions, (2) axioms accepted as true, (3) inference rules, and (4) theorems.

3.1 Definitions and Terminology

Use of the standard definitions and terminology eases communication, reduces misunderstanding or ambiguity. The following definitions are selected from many others since they will be used throughout the context. They are summarized from [Rose95], [Rush93b], [Leig97], and [Karg97].

Definition 3.1-1: A *proposition* is a statement that is either true or false, but not both. A *compound proposition* is a proposition formed from existing propositions using logical operators. A *predicate* is an expression containing variables such that if the variables are replaced by values, the expression is a proposition. A predicate is also called a *propositional function*.

Definition 3.1-2: An *axiom* is a proposition that is assumed to be true. Therefore, there is no proof that an axiom is true. A *theorem* is a proposition that can be derived by applying inference rules to the set of axioms. A *lemma* (plural *lemmas* or *lemmata*) is a simple theorem used in the proof of other theorems. The distinction between a lemma and a theorem is pretty arbitrary. Complicated proofs are usually easier to understand when they are proved using a series of lemmas, each of which can be proved individually. A *corollary* is a true proposition that can be established directly from a theorem that has been proved.

Definition 3.1-3: A set of axioms is *consistent* if no proposition can be proved to be both true and false. Consistency is an absolute must for a proof system. A set of axioms is *complete* if it can be used to prove or disprove any proposition. A proof system is *sound* if, starting with axioms (which are valid), each subsequent formula derived with the inference rules is also valid.

Definition 3.1-4: The *rules of inference* are the means used to derive new conclusions from existing assertions.

Definition 3.1-5: A mathematical *proof* is a verification of a proposition by a chain of logical deductions from a set of axioms. It has three distinct steps:

- to begin with true propositions, e.g., axioms;
- to use inference rules;
- to derive new true propositions.

Definition 3.1-6: A proof system is *decidable* if there exists an algorithm (i.e., a computer program that is guaranteed to terminate) that can determine whether or not any given statement is a theorem. A proof system is *semidecidable* if there exists an algorithm that can recognize the theorem (i.e., the algorithm is guaranteed to terminate with the answer “yes” if a theorem is given, but that need not terminate if a non-theorem is given—though if it does halt, it must give the answer “no”). A proof system is *undecidable* if it is neither decidable nor semidecidable.

Definition 3.1-7: A proof system is *sound* if every provable fact is true. A proof system is *complete* if every true fact is provable. Inconsistent system can not be sound since an inconsistent system contains both a proposition and its negation as theorems, which can be used to prove any proposition, true or false. Only sound systems are of use in formal methods. It would be nice if they are also complete and even nicer if they are decidable.

Definition 3.1-8: A *tautology* is a compound proposition that is always true, no matter what truth values are assigned to the propositions that occur in it. A *contradiction* is a compound proposition that is always false. A *contingency* is proposition that is neither a tautology nor a contradiction.

3.2 Propositional Calculus

3.2.1 Basic Propositional Operations

The propositional calculus is sound (i.e., all valid theorems are provable), complete (i.e., all provables are valid theorems), and decidable (i.e., by means of truth table). There are five propositional operations: *negation*, *conjunction*, *disjunction*, *implication*, and *equivalence* in descending order of precedence of operation. Other operations such as *Exclusive OR*, *NAND*, *NOR* can be derived from these fundamental logic operations. For example, Exclusive OR is the negation of equivalence, NAND is the negation of conjunction, and NOR is the negation of disjunction.

3.2.1.1 Negation (\neg or \sim)

Definition 3.2-1: Let P be a proposition, the statement “It is not the case that P ” is defined to be the *negation* of P , denoted as $\neg P$ or $\sim P$. $\neg P$ is *true* when P is *false* and *false* otherwise.

The negation of a proposition can also be considered as the result of the negation operator (\neg or \sim) operated on the proposition. The negation operator creates a new proposition from a single proposition. As a result, the negation operator is called a monadic operator. The truth table for $\neg P$ is shown in Table 3.2-1. In the table, T stands for truth value *true*, F stands for truth value *false*. The size of the truth table (i.e., number of rows) is 2^n , where n is the number of independent propositions. Therefore, if n is large, the size of the table will be explosive exponentially. In the negation operation, $n=1$. Hence the truth table has two rows.

3.2.1.2 Conjunction (\wedge)

Definition 3.2-2: Let P and Q be two propositions, the *conjunction* of P and Q (denoted $P \wedge Q$) is defined to be the compound proposition that is *true* only when both P and Q are *true* and is *false* otherwise.

The conjunction operator (\wedge) creates a new proposition by operating on two propositions. Hence, it is called a dyadic operator. The truth table of conjunction is shown in Table 3.2-2. (To be consistent with boolean algebra, T is customarily denoted as 1 while F denoted as 0 in the truth tables.)

3.2.1.3 Disjunction (\vee)

Definition 3.2-3: Let P and Q be two propositions, the *disjunction* of P and Q (also known as *inclusive or*, denoted $P \vee Q$) is defined to be the proposition that is *false* only when both P and Q are *false* and is *true* otherwise.

The disjunction operator (\vee) creates a new proposition by operating on two propositions. Hence, it is a dyadic operator. The truth table of disjunction is shown in Table 3.2-3.

3.2.1.4 Implication (\Rightarrow or \supset or \rightarrow)

Definition 3.2-4: Let P and Q be two propositions, the implication $P \Rightarrow Q$ is the proposition that is *false* only when P is *true* and Q is *false*, and is *true* otherwise. In the implication, P is called the *hypothesis* (or *antecedent* or *premise* or *pre-condition*), Q is called the *conclusion* (or *consequence* or *post-condition*).

The implication operator (\Rightarrow) creates a new proposition by operating on two propositions. Hence, it is a dyadic operator. It is most frequently used in deriving new propositions from the known propositions in the format of “if P , then Q ”, “ P implies Q ”, “ P only if Q ”, “ P is sufficient for Q ”, “ Q if P ”, “ Q whenever P ”, “ Q is necessary for P ”, etc. The truth table of implication is shown in Table 3.2-4. This operation deserves more discussion with an example. Let

P be the proposition “a polygon is a square”, and

Q be the proposition “each internal angle of the polygon is 90° ”. Therefore,

$P \Rightarrow Q$ represents the sentence (or compound proposition) “*If a polygon is a square, then each internal angle of the polygon is 90°* ”.

To ease our discussion, let us start with the last row in Table 3.2-4. This row represents that a true conclusion (i.e., Q is 1 or it is true that *each internal angle of the polygon is 90°*) can be derived *validly* (i.e., the truth value of $P \Rightarrow Q$ is 1) from a true premise (i.e., P is 1 or it is true that the *polygon is indeed a square*). This case is rather straightforward. So is the third row, which says that the reasoning is false (i.e., the truth value of $P \Rightarrow Q$ is 0) if we get a false conclusion (e.g., *each internal angle of the polygon is 90°*) when we start with a true premise (e.g., *the polygon is indeed a square*).

The tricky cases are the first two rows, which indicates that if the premise is false, then no matter what conclusion we end up with, it is a valid reasoning. Put it in another way, we can get any conclusion (i.e., Q is either 1 or 0) from a false premise (i.e., P is 0). In other words, we don't care what conclusion we get if premise we start with is false. In this example, if a polygon is *not* a square, then we do not know whether each internal angle of the polygon is 90° from the implication: it may be (e.g., if the polygon is a rectangle) or may be not (e.g., if the polygon is a trapezoid). The cases can be used as the basis for a vacuous proof, e.g., in establishing base case in mathematical induction. The first row has applications in detecting inconsistency (or contradiction) in the premise (i.e., we will conclude that P is 0) if we end up with a false conclusion (i.e., Q is 0) and our reasoning process is valid (i.e., $P \Rightarrow Q$ is 1). This property can be used to detect inconsistency in Software Requirements Analysis (SRA). By comparing columns 2 and 4 in Table 3.2-4, we get a very important property of implication, that is, $P \Rightarrow Q$ is logically identical to $\neg P \vee Q$. This equality can also be obtained by applying De Morgan's Law and the Law of Double Negation to the third row, i.e.,

$$\neg(P \Rightarrow Q) = P \wedge \neg Q \quad (\text{row 3})$$

Therefore,

$$\begin{aligned} P \Rightarrow Q &= \neg(P \wedge \neg Q) \\ &= \neg P \vee \neg(\neg Q) && (\text{De Morgan's Law in Table 3.2-11}) \\ &= \neg P \vee Q && (\text{Double Negation Law in Table 3.2-9}) \end{aligned}$$

3.2.1.5 Equivalence or Equality (\Leftrightarrow or \equiv)

Definition 3.2-5: Let P and Q be two propositions, the *equivalence* of P and Q (denoted $P \Leftrightarrow Q$) is defined to be the proposition that is *true* only when P and Q have the same truth value and is *false* otherwise.

As a dyadic operator, the equivalence operator is similar to equality of algebra (we might interchangeably use \Leftrightarrow and $=$ in the text). In logic, it is used to represent English phrases “exactly when”, “only when”, “if and only if (iff)”, “if... then..., and vice versa”, “if...then, and conversely”, “sufficient and necessary for”, etc. The truth table of equivalence is shown in Table 3.2-5. From the truth table, it is concluded the $P \Leftrightarrow Q$ is the same as $P \Rightarrow Q \wedge Q \Rightarrow P$.

3.2.2 Derived Propositional Operations

3.2.2.1 Exclusive Or (\vee_e or \oplus)

Definition 3.2-6: Let P and Q be two propositions, the *exclusive or* of P and Q (denoted $P \vee_e Q$ or $P \oplus Q$) is defined to be the proposition that is *true* only when exactly one of P and Q is true and is *false* otherwise.

The exclusive or (\vee_e) creates a new proposition by operating on two propositions. Hence, it is a dyadic operator. The truth table of exclusive or is shown in Table 3.2-6.

- By comparing columns 3 and 6, it is concluded that $P \oplus Q \Leftrightarrow (\neg P \wedge Q) \vee (P \wedge \neg Q)$;
- By comparing the table with Table 3.2-5, it is concluded that exclusive or is the negation of equivalence.

3.2.2.2 NAND (\downarrow)

Definition 3.2-7: Let P and Q be two propositions, the proposition P *NAND* Q (denoted $P \downarrow Q$) is defined to be *false* when both P and Q are true and is *true* otherwise.

As a dyadic operator, the *NAND* operation is the negation of conjunction. The truth table for *NAND* is shown in Table 3.2-7.

3.2.2.3 NOR (\downarrow)

Definition 3.2-8: Let P and Q be two propositions, the proposition $P \text{ NOR } Q$ (denoted $P \downarrow Q$) is defined to be *true* when both P and Q are false and is *false* otherwise.

As a dyadic operator, the *NOR* operation is the negation of disjunction. The truth table for *NOR* is shown in Table 3.2-8.

3.2.3 Important Logic Laws

Table 3.2-9 contains some important identities (or logic laws) found in standard formal logic textbooks such as [Rose95]. In the table, P , Q , and R are propositions, T stands for *true* and F for *false*. Familiarity with these laws is essential to simplify logic expressions and to prove new logical identities.

3.2.4 Important Inference Rules

Based on tautology, inference rules are means of drawing conclusions for known facts or assertions. Some important inference rules are listed in Table 3.2-10.

3.2.5 Some Important Theorems

All the theorems listed in Table 3.2-11 are either the author's original contribution or from other literature. Literally, the majority of propositional logic theorems can be proved by using the truth tables. However, the truth tables are not always the best proof method, the method becomes tedious when three or more logical variables are involved in a theorem. The author provides all the proofs in Appendix 9.

Table 3.2-1 Truth Table for Negation ($\neg P$)

P	$\neg P$
T	F
F	T

Table 3.2-2 Truth Table for Conjunction ($P \wedge Q$)

P	Q	$P \wedge Q$
F (0)	F (0)	F (0)
F (0)	T (1)	F (0)
T (1)	F (0)	F (0)
T (1)	T (1)	T (1)

Table 3.2-3 Truth Table for Disjunction ($P \vee Q$)

P	Q	$P \vee Q$
0	0	0
0	1	1
1	0	1
1	1	1

Table 3.2-4 Truth Table for Implication ($P \Rightarrow Q$)

P	Q	$P \Rightarrow Q$	$\neg P$	$\neg P \vee Q$
0	0	1	1	1
0	1	1	1	1
1	0	0	0	0
1	1	1	0	1

Table 3.2-5 Truth Table for Equivalence ($P \Leftrightarrow Q$)

P	Q	$P \Leftrightarrow Q$	$Q \Rightarrow P$	$P \Rightarrow Q$	$P \Rightarrow Q \wedge Q \Rightarrow P$
0	0	1	1	1	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	1	1	1

Table 3.2-6 Truth Table for Exclusive OR ($P \oplus Q$)

P	Q	$P \oplus Q$	$\neg P \wedge Q$	$P \wedge \neg Q$	$\neg P \wedge Q \vee P \wedge \neg Q$
0	0	0	0	0	0
0	1	1	1	0	1
1	0	1	0	1	1
1	1	0	0	0	0

Table 3.2-7 Truth Table for NAND ($P \mid Q$)

P	Q	$P \mid Q$	$P \wedge Q$	$\neg(P \wedge Q)$
0	0	1	0	1
0	1	1	0	1
1	0	1	0	1
1	1	0	1	0

Table 3.2-8 Truth Table for NOR ($P \downarrow Q$)

P	Q	$P \downarrow Q$	$P \vee Q$	$\neg(P \vee Q)$
0	0	1	0	1
0	1	0	1	0
1	0	0	1	0
1	1	0	1	0

Table 3.2-9 Important Logic Laws

<i>Name of the Laws</i>	<i>Logic Laws</i>
Identity Laws	$P \wedge T \Leftrightarrow P$ $P \vee F \Leftrightarrow P$
Domination Laws	$P \vee T \Leftrightarrow T$ $P \wedge F \Leftrightarrow F$
Completeness Law ⁵	$P \vee \neg P \Leftrightarrow T$
Inconsistency Law ⁶	$P \wedge \neg P \Leftrightarrow F$
Idempotent Laws	$P \vee P \Leftrightarrow P$ $P \wedge P \Leftrightarrow P$
Double Negation Law	$\neg(\neg P) \Leftrightarrow P$
Commutative Laws	$P \vee Q \Leftrightarrow Q \vee P$ $P \wedge Q \Leftrightarrow Q \wedge P$
Associative Laws	$(P \vee Q) \vee R \Leftrightarrow P \vee (Q \vee R)$ $(P \wedge Q) \wedge R \Leftrightarrow P \wedge (Q \wedge R)$
Distributive Laws	$P \vee (Q \wedge R) \Leftrightarrow (P \vee Q) \wedge (P \vee R)$ $P \wedge (Q \vee R) \Leftrightarrow (P \wedge Q) \vee (P \wedge R)$
De Morgan's Laws	$\neg(P \wedge Q) \Leftrightarrow \neg P \vee \neg Q$ $\neg(P \vee Q) \Leftrightarrow \neg P \wedge \neg Q$

⁵ This name is given by the author. There might be other names.

⁶ This name is given by the author. There might be other names.

Table 3.2-10 Important Inference Rules

<i>Name</i>	<i>Rules of Inference</i>	<i>Corresponding Tautology</i>
Addition	$\frac{P}{\therefore P \vee Q}$	$P \Rightarrow (P \vee Q)$
Simplification	$\frac{P \wedge Q}{\therefore P}$	$(P \wedge Q) \Rightarrow P$
Modus Ponens	$\frac{P \quad P \Rightarrow Q}{\therefore Q}$	$[P \wedge (P \Rightarrow Q)] \Rightarrow Q$
Modus Tollens	$\frac{\neg Q \quad P \Rightarrow Q}{\therefore \neg P}$	$[\neg Q \wedge (P \Rightarrow Q)] \Rightarrow \neg P$
Hypothetical Syllogism	$\frac{P \Rightarrow Q \quad Q \Rightarrow R}{\therefore P \Rightarrow R}$	$(P \Rightarrow Q \wedge Q \Rightarrow R) \Rightarrow (P \Rightarrow R)$
Disjunctive Syllogism	$\frac{P \vee Q \quad \neg P}{\therefore Q}$	$(P \vee Q) \wedge \neg P \Rightarrow Q$

Table 3.2-11 Some Important Theorems in Propositional Calculus

<i>Theorem Number</i>	<i>Theorem</i>
Theorem 3.2-1A	$P \Rightarrow P \vee Q$ is a tautology.
Theorem 3.2-1B	$P \wedge Q \Rightarrow P$ is a tautology.
Corollary 3.2-1	$P \wedge Q \Rightarrow P \vee Q$ is a tautology.
Theorem 3.2-2A	If $P \Rightarrow Q$, then $P \wedge R \Rightarrow Q \wedge R$.
Theorem 3.2-2B	If $P \Rightarrow Q$, then $P \vee R \Rightarrow Q \vee R$.
Theorem 3.2-3	If $P \Rightarrow Q$ and $Q \Rightarrow R$, then $P \Rightarrow R$.
Corollary 3.2-2A	If $P_1 \Rightarrow P_2 \wedge P_2 \Rightarrow P_3 \wedge P_3 \Rightarrow P_4 \wedge \dots \wedge P_{n-1} \Rightarrow P_n$, then $P_1 \Rightarrow P_n$.
Corollary 3.2-2B	If $P \Leftrightarrow Q \wedge Q \Leftrightarrow R$, then $P \Leftrightarrow R$.
Theorem 3.2-4A	If $P \Rightarrow Q$ and $S \Rightarrow T$, then $P \wedge S \Rightarrow Q \wedge T$.
Theorem 3.2-4B	If $P \Rightarrow Q$ and $S \Rightarrow T$, then $P \vee S \Rightarrow Q \vee T$.
Theorem 3.2-5	$P \Rightarrow P \wedge Q$ is equivalent to $P \Rightarrow Q$. Mathematically, $(P \Rightarrow Q) \Leftrightarrow (P \Rightarrow P \wedge Q)$.
Theorem 3.2-6A	$P \wedge (P \vee Q) \Leftrightarrow P$
Theorem 3.2-6B	$P \vee (P \wedge Q) \Leftrightarrow P$
Theorem 3.2-7A	$(P \Rightarrow Q \wedge R) \Leftrightarrow (P \Rightarrow Q \wedge P \Rightarrow R)$
Theorem 3.2-7B	$(P \Rightarrow Q \vee R) \Leftrightarrow (P \Rightarrow Q \vee P \Rightarrow R)$
Theorem 3.2-7C	$(P \wedge Q \Rightarrow R) \Leftrightarrow (P \Rightarrow R \vee Q \Rightarrow R)$
Theorem 3.2-7D	$(P \vee Q \Rightarrow R) \Leftrightarrow (P \Rightarrow R \wedge Q \Rightarrow R)$
Theorem 3.2-8	$(P \Rightarrow R \wedge Q \Rightarrow R) \Rightarrow (P \wedge Q \Rightarrow R)$
Theorem 3.2-9	$(P \wedge Q) \vee (\neg P \wedge R) \vee (Q \wedge R) \Leftrightarrow (P \wedge Q) \vee (\neg P \wedge R)$
Theorem 3.2-10	$P \wedge Q \Rightarrow (P \Rightarrow Q)$
Corollary 3.2-3	$P \wedge Q \Rightarrow (P \Leftrightarrow Q)$
Lemma 3.2-1	$(P \Rightarrow (\text{true} \Rightarrow Q)) \Leftrightarrow (P \Rightarrow Q)$
Theorem 3.2-11	$(P \Rightarrow Q) \wedge (\neg P \Rightarrow R) \Rightarrow (Q \vee R)$
Theorem 3.2-12	$(P \Leftrightarrow Q) \Leftrightarrow (\neg P \Leftrightarrow \neg Q)$
Theorem 3.2-13	$(P \Rightarrow Q \wedge Q \Rightarrow R \wedge R \Rightarrow P) \Leftrightarrow (P \Leftrightarrow Q \wedge Q \Leftrightarrow R)$
Corollary 3.2-4	$(P_1 \Rightarrow P_2 \wedge P_2 \Rightarrow P_3 \wedge P_3 \Rightarrow P_4 \wedge \dots \wedge P_{n-1} \Rightarrow P_n \wedge P_n \Rightarrow P_1) \Leftrightarrow$ $(P_1 \Leftrightarrow P_2 \wedge P_2 \Leftrightarrow P_3 \wedge \dots \wedge P_{n-1} \Leftrightarrow P_n)$
Theorem 3.2-14	If $P \Rightarrow Q_1 \wedge Q_2$ and $\neg Q_1$, then $\neg P$.

3.3 Predicate Calculus

Like propositional calculus, predicate calculus is sound and complete. However, unlike propositional calculus, predicate calculus is semi-decidable.

When each variable in a predicate is assigned to a value, the predicate is reduced to a proposition with a truth value. Another kind of mechanism converting the predicate into a proposition is called quantification. In general, two types of quantification are used, namely, the *universal quantification* and the *existential quantification*. The universal quantification of predicate $P(x)$ is the proposition “ $P(x)$ is *true* for *all* values of x in the universe of discourse X ”, denoted as $\forall x \in X. P(x)$. The existential quantification of predicate $P(x)$ is the proposition “There *exist* an element of x in the universe of discourse X such that $P(x)$ is *true*”, denoted as $\exists x \in X. P(x)$ (see Table 3.3-1).

Some of the important theorems in predicate calculus are listed in Table 3.3-2.

Table 3.3-1 Universal Quantification and Existential Quantification

<i>Quantification</i>	<i>When true?</i>	<i>When false?</i>
$\forall x \in X. P(x)$	$P(x)$ is <i>true</i> for every x in X .	There is an x in X for which $P(x)$ is <i>false</i> .
$\exists x \in X. P(x)$	There is an x in X for which $P(x)$ is <i>true</i> .	$P(x)$ is <i>false</i> for every x in X .

Table 3.3-2 Some Important Theorems in Predicate Calculus

<i>Theorem #</i>	<i>The Theorem</i>
3.4-1A	$\forall x \in X. P(x) \Leftrightarrow \neg \exists x \in X. \neg P(x)$
3.4-1B	$\neg \forall x \in X. \neg P(x) \Leftrightarrow \exists x \in X. P(x)$
3.4-2A	$\forall x \in X. \forall y \in Y. P(x, y) \Leftrightarrow \forall y \in Y. \forall x \in X. P(x, y)$
3.4-2B	$\exists x \in X. \exists y \in Y. P(x, y) \Leftrightarrow \exists y \in Y. \exists x \in X. P(x, y)$
3.4-2C	$\forall x \in X. \exists y \in Y. P(x, y) \neq \exists y \in Y. \forall x \in X. P(x, y)$
3.4-3A	$(\forall x \in X. P(x) \wedge Q(x)) \Leftrightarrow (\forall x \in X. P(x)) \wedge (\forall x \in X. Q(x))$
3.4-3B	$(\exists x \in X. P(x) \vee Q(x)) \Leftrightarrow (\exists x \in X. P(x)) \vee (\exists x \in X. Q(x))$
3.4-3C	$(\forall x \in X. P(x) \vee Q(x)) \neq (\forall x \in X. P(x)) \vee (\forall x \in X. Q(x))$
3.4-3D	$(\exists x \in X. P(x) \wedge Q(x)) \neq (\exists x \in X. P(x)) \wedge (\exists x \in X. Q(x))$
3.4-4A	$(\forall x \in X. P(x)) \wedge R \Leftrightarrow (\forall x \in X. P(x) \wedge R)$
3.4-4B	$(\forall x \in X. P(x)) \vee R \Leftrightarrow (\forall x \in X. P(x) \vee R)$
3.4-4C	$(\exists x \in X. P(x)) \wedge R \Leftrightarrow (\exists x \in X. P(x) \wedge R)$
3.4-4D	$(\exists x \in X. P(x)) \vee R \Leftrightarrow (\exists x \in X. P(x) \vee R)$
3.4-5A	$\forall x \in \Phi. P(x) \Leftrightarrow \text{true}$
3.4-5B	$\exists x \in \Phi. P(x) \Leftrightarrow \text{false}$
Note	A is a proposition without any quantifiers.

Chapter 4

The Proposed Methodology

4.1 Introduction

A set of propositional expressions (e.g., those translated from requirements) is *consistent*, if and only if, there exists at least one assignment of truth values for the variables in each propositional expression such that all the expressions simultaneously receive the *true* value or, equivalently, the conjunction of the expressions receives the *true* value^[Ince88]. *Inconsistency* is simply the negation of *consistency*. A set of logic expressions is inconsistent, if and only if, for every assignment of truth values to the variables making up each propositional expression, there is at least one expression receiving the *false* value. Consistency requires the set of requirements be consistent physically and logically, i.e., no contradictory requirements should exist in the set of requirements. To achieve consistency, we need to undertake mathematical analysis, physical modeling, and/or logical reasoning. For completeness, we want the set of requirements to describe all required properties or behaviors of the software. This is the basis against which QA personnel check to see whether the final product is really what has been expected.

This analysis is extremely important since the work required to correct a fault infused at an early stage of software development process can cost up to two orders of magnitude more if not identified and corrected immediately. With a set of complete and consistent requirements, work during the remaining stages can be undertaken more efficiently and more cost-effectively by minimizing the amount of rework.


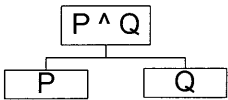
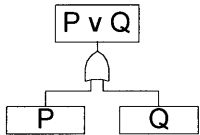
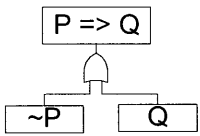
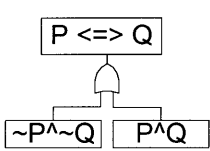
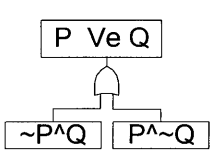
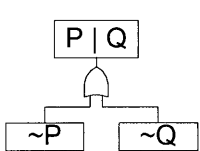
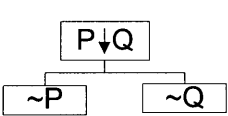
Based on the concept of “divide-and-conquer”, the graphical approach will be focused on consistency checks. The task of Software Requirements Analysis (SRA) is divided into

groups of smaller sub-problems. Using tree-based graphical methods, we first analyze the consistency of requirements in each group individually. In addition to the graphical approach developed in the dissertation, the author identified three other approaches applicable to the analysis: the Analytical Approach, the Truth-Table Approach, and the Dynamic Flowgraph Methodology (DFM). By combining the results of the sub-problems, we analyze the requirements of the original problem using dependency diagrams. The details of the process will be illustrated by an example in Section 4.4.

4.2 Graphical Representation of the Logic Operations

As defined in Section 3.1, a proposition is a statement that is either *true* or *false* (not both), e.g., “It is raining now” is a proposition, which is either *true* or *false*. There are five basic logic operations on propositional logic: negation, conjunction, disjunction, implication, and equivalence. Three derived operations are exclusive or, NAND, and NOR. These operations are graphically expressed in Table 4.2-1, where P and Q are propositions. From the top to the bottom rows of the table, the precedence of the operations is decreasing. The first row represents the negation operation, which is *true* if P is *false*, and *false* otherwise. The second row represents the conjunctive operation which is *true* if and only if its constituents P , Q are *true* simultaneously. The third row represents the disjunctive operation which is *false* if and only if its constituents P , Q are *false* simultaneously. The fourth row is rather tricky. The implication is *true* if and only if either both of the constituents, P (the premise or pre-condition) and Q (the consequence or post-condition), are *true* simultaneously or P is *false*. The former case means that the truth of the consequence can validly follow from the truth of the premise. The latter case simply means, if the premise is false, then any consequence can follow from the premise. In other words, if the premise is not satisfied, one does not care about the consequence at all. Logically, the implication operation ($P \Rightarrow Q$) is identical to $\neg P \vee Q$. Equivalence is a two-way implication, that is, $P \Leftrightarrow Q$ is the same as $P \Rightarrow Q$ and $Q \Rightarrow P$. To simplify the graphics, the conjunction gate (\wedge) itself is not plotted on the graph for simplicity and in conformance with the organization charts as well as to distinguish the operation from the disjunction gate (\vee).

Table 4.2-1 Graphical Representation of Propositional Operations

<i>Propositional Operations</i>		<i>Equivalent Graphical Representations</i>	<i>Priority (1 = highest)</i>
Negation	$\neg P$ (or $\sim P$)	 (or $\neg P$)	1
Conjunction	$P \wedge Q$		2
Disjunction	$P \vee Q$		3
Implication	$P \Rightarrow Q$		4
Equivalence	$P \Leftrightarrow Q$		5
Exclusive OR	$P \vee_e Q$		—
NAND	$P Q$		—
NOR	$P \downarrow Q$		—

4.3 Procedure in the Proposed Graphical Approach

The purpose of requirements analysis is to resolve inconsistency and ambiguity in the requirements, to remove all irrelevant information, and to achieve completeness. The proposed methodology is comprised of three major steps listed below and also shown schematically in Figure 4-3.1.

Step I Collection of software requirements.

In general, the requirements from the customer are given piece-by-piece and evolve with time. If the customer has provided a complete set of requirements, this step can be bypassed. In most cases, however, this will not be the case.

Step II Grouping of software requirements.

This step divides the original set of requirements into groups whose union is the original set.

Step III Analyze software requirements for unambiguity, consistency (and completeness) using the proposed graphical approach.

Consistency of requirements is achieved by a process containing two sub-steps known as the “within-group consistency” analysis followed by the “between-group consistency” analysis.

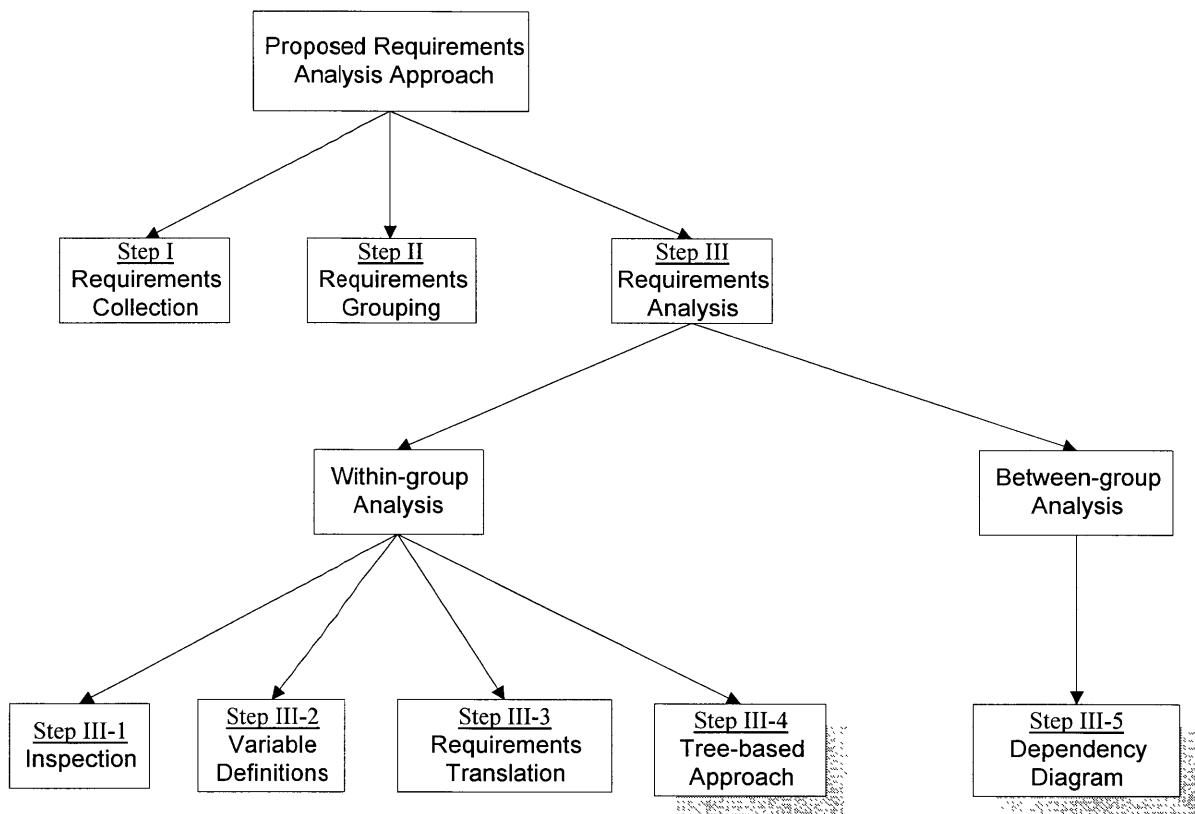


Figure 4.3-1 Schematic of the Proposed Graphical Approach

4.4 The Proposed Graphical Approach and An Example

The procedure described in Section 4.3 will be demonstrated through an example of an automobile controller. The purpose of the controller is to control the modes of operation of the automobile. The set of requirements is elicited in Table 4.4-1 (assuming that they have been provided by the customer).

4.4.1 Requirements Grouping and Its Benefits

Software requirements are typically informal documents from the customer that are usually evolving. If it is a brand new project with which the customer does not have much experience or does not understand well, then it is usually the case that the requirements are ambiguous, inconsistent, incomplete, or even ill-structured. On the other hand, if the customer wants to upgrade an existing system that he understands pretty well, it is likely that the relationship between the existing system and its successor is not defined clearly and correctly in his requirements. Therefore, in either case, ambiguity, inconsistency, or incompleteness may exist in the requirements.

Before grouping the requirements, it is crucial to identify keywords to serve as the titles of groups. These keywords are usually the major tasks that the system is to accomplish and/or constraints that the system is required to satisfy. Grouping is based on interrelations between the requirements, e.g., requirements of a specific topic are grouped together so that completeness and consistency analyses on that topic can be done fairly easily because they are fewer in number and are logically related. Within each group, only logical operations exist between requirements. If two requirements are related by physical relations, then they belong to two different groups. Between these groups, we analyze their inter-consistency based on their mathematical or physical relationships.

Table 4.4-1 Example: Requirements on an Auto Controller

1.	The auto controller has six modes of operation: standstill, constant acceleration, varying acceleration, constant speed cruise, constant deceleration, varying deceleration.
2.	If the auto controller is in standstill, then the auto speed is zero.
3.	If the auto controller is in constant speed mode, then the auto speed is constant at 100 miles/hour or 44.4m/s.
4.	If the auto controller is in constant acceleration mode, then the auto must maintain the constant acceleration of 10 m/s ² .
5.	If the auto controller is in constant deceleration mode, then the auto must maintain the constant deceleration of -10 m/s ² .
6.	The auto controller will not be in varying acceleration or varying deceleration modes.
7.	Combustion of gas is the source of power that accelerates the auto. The nozzle opens at a constant rate of 20% of its entire opening per second to regulate the intake of gas, e.g., its opening is proportional to time once the constant acceleration mode is entered.
8.	Constant gas speed is assumed at the nozzle. All losses are negligible and all energy generated by gas combustion is converted into kinetic energy of the car (In reality, an energy conversion factor, η , is assumed. For simplicity of the analysis, this factor is assumed to be 1.0 here).

Since grouping combines similar requirements together, any changes made to the requirements are confined locally. For example, if a requirement is changed in one group, then changes in the requirements analysis are limited to this group or its related groups. One does not need to worry about other groups. The same is true with the addition of requirements. If new requirements are added to a specific group or an entire new group of requirements is added, only that group or its related groups will be affected. Hence, requirements analyses should only be conducted locally, i.e., within that group or between the related groups. No other groups need to be worried about. This property of locality is very important in software requirements analysis, in that software requirements are evolving constantly in practice.

It should be noted that grouping is an important process, but it is not crucial. Grouping collects all logically relevant requirements into one group and each requirement is assigned into some group (i.e., no requirement is neglected). When grouping requirements, it is common for an original requirement to contain information belonging to more than one group. If this is the case, we can duplicate the requirement for as many times as the number of groups to which the requirement correlates. Mathematically, this duplication does not alter the truth value of the logical expressions, i.e., requirements do not become overly-restrictive (see Theorem 4.4-1). Therefore, in no sense will the requirements grouping lose or gain anything. However, in practice, a good grouping will facilitate requirements analysis. In the example, requirements 1 through 6 for the car controller are collected into one group entitled “Operational Mode” since these requirements refer to how the race car is operated, while requirements 7 and 8 are grouped as “Driving Force” since they refer to the car’s power system.

Theorem 4.4-1

Let $P_1, P_2, P_3, \dots, P_i, \dots, P_n$ be propositions, then for any $m \geq 1$ and $1 \leq i \leq n$,

$$P_1 \wedge P_2 \wedge P_3 \wedge \dots \wedge P_i \wedge \dots \wedge P_n \Leftrightarrow P_1 \wedge P_2 \wedge P_3 \wedge \dots \wedge P_i \wedge P_i \wedge P_i \wedge \dots \wedge P_i \wedge \dots \wedge P_n$$

$$|\longleftarrow m \longrightarrow|$$

Lemma 4.4-1

If Q is a proposition and $m > 1$, then

$$Q \Leftrightarrow Q \wedge Q \wedge Q \wedge \dots \wedge Q.$$

$$|\longleftarrow m \longrightarrow|$$

Proof of Corollary using Induction.

1. Base case with $m = 2$ is true directly from the Idempotent Law in Table 3.2-6.
2. Induction: Assume the lemma is true for m , that is,

$$Q \Leftrightarrow Q \wedge Q \wedge Q \wedge \dots \wedge Q.$$

$$|\longleftarrow m \longrightarrow|$$

Starting with induction assumption

$$Q \Leftrightarrow Q \wedge Q \wedge Q \wedge \dots \wedge Q$$

$$|\longleftarrow m \longrightarrow|$$

and substituting the Idempotent Law (i.e., $Q \Leftrightarrow Q \wedge Q$ for the last proposition Q in the induction assumption yields

$$Q \Leftrightarrow Q \wedge Q \wedge Q \wedge \dots \wedge (Q \wedge Q)$$

$$|\longleftarrow m \longrightarrow|$$

which is equivalent to

$$Q \Leftrightarrow Q \wedge Q \wedge Q \wedge \dots \wedge (Q \wedge Q).$$

$$|\longleftarrow m + 1 \longrightarrow|$$

Therefore if the lemma is true for case m , then it is true for case $m+1$. By induction, it is concluded that the lemma itself is true. ■

The correctness of the theorem can be justified by using the corollary with $Q = P_i$.

4.4.2 Within-Group Analysis

Within each group, we use inspection and the tree-based graphical representation of propositional logic. A typical within-group analysis has four steps, as identified in Figure 4.3-1. First, inspection is used to remove any obvious ambiguities and inconsistencies that

may occur in different sections of the requirements document. For unambiguity, we need to look for qualitative words that do not precisely specify what is required, e.g., “some”, “several”, “many”, “(quite) a few”, “significant”, “good criteria”, “acceptable safety margin”, etc. These words are a source of ambiguity.

The next step defines logical variables. This step is the most challenging, as it requires the judgment of the analyst. However, once this is done, the remaining work will be mechanical and relatively straightforward, i.e., it will involve the manipulation of variables according to the defined logical rules. In defining the events, one must always bear in mind that an event and its negation shall *never* be defined as two different variables. Step Three translates natural language requirements into equivalent propositional expressions using the definitions in Step Two. In Step Four, the Tree-Based Graphical Approach (TBGA) is used to check for inconsistencies hidden in the document.

4.4.2.1 Inspection

By inspecting the requirements, one might realize that requirements for two operational modes of the controller are not specified, i.e., those for varying acceleration and varying deceleration. This important discovery should be reported to the customer for clarification or addition. In this example, however, these requirements turn out to be irrelevant information (see Section 4.4.2.4: Tree-based Graphical Approach).

4.4.2.2 Variable Definitions

Let

```
SS = auto controller's standstill operational mode
CA = auto controller's constant acceleration operational mode
VA = auto controller's varying acceleration operational mode
CS = auto controller's constant speed operational mode
VS = auto controller's varying speed operational mode
CD = auto controller's constant deceleration operational mode
VD = auto controller's varying deceleration operational mode
```

a = auto's acceleration (positive means acceleration,
negative means deceleration)
 A = gas nozzle entire area of opening, i.e. maximum of $A(t)$
 t = time after a mode of operation is entered
 $A(t) = k \cdot t$ opening of nozzle at time t
 v = auto speed
 V_F = the variable representing gas speed at the nozzle
 v_F = the constant value of gas speed at the nozzle

Subscripts SS , CA , VA , CS , VS , CD , VD refer to the parameters under auto controller's corresponding modes of operation.

4.4.2.3 Translation of Requirements from English to Logic Expressions

Once variables are defined, the English requirements for the auto controller in Table 4.4-1 can be translated into corresponding mathematical expressions. The result of the translation is shown in Table 4.4-2.

It should be noted that, when translating requirement 7 into mathematics, one finds that at no time should the opening of the gas nozzle be greater than the maximum possible opening (i.e., $A(t) \leq A$). However, this is missing from the original set of requirements. Therefore, translating requirements from natural language into mathematics does help us re-think about the requirements and makes them more complete.

4.4.2.4 Tree-Based Graphical Approach

In constructing the tree model, the logic expressions are translated into the equivalent tree based on Table 4.2-1 which includes the graphical representation of five logic operations. The graphical approach is more mechanical and less mathematics-oriented than the purely analytical approach. For any one familiar with fault tree analysis (FTA), the graphical method is very straightforward. For those unfamiliar with FTA, only slight training is necessary. Therefore, it is expected to be very helpful for practicing engineers.

Table 4.4-2 Example: Mathematical Representation of Requirements
for an Auto Controller

<i>Requirements #</i>	<i>Group “Operational Mode”</i>
1	$SS \vee_e CA \vee_e VA \vee_e CS \vee_e CD \vee_e VD$
2	$SS \Rightarrow V_{ss}=0$
3	$CS \Rightarrow V_{cs}=44.4m/s$
4	$CA \Rightarrow a_{ca}=10m/s^2$
5	$CD \Rightarrow a_{cd}=-10m/s^2$
6	$\neg VA \wedge \neg VD.$
	<i>Group “Driving Force”</i>
7	$\forall t, A(t)=k \cdot t \wedge A(t) \leq A$
8	$V_F=v_F$ (where v_F is constant gas speed at the nozzle).

Tree-Expansion Algorithm The idea behind the graphical method is straightforward. To check whether the requirements are consistent is equivalent to checking whether the corresponding logic expressions can hold the truth value of *true* simultaneously, i.e., the conjunction of the logic expressions can be *true*. Each of these expressions is transformed into its equivalent graphical form, i.e., the transformation is an identity. The rule of tree expansion is to start with the simplest branch followed by the ones containing the same basic events or their negation as in the already expanded one. It should be pointed out that the order in which the next branch/sub-expressions is picked in the analysis does not affect the result of the analysis. However, it does affect the structural complexity of the tree in the process of the analysis (see Sections 4.5.5 and 5.3.2.5 for more detail). Hierarchically, these equivalent forms are transformed into more basic forms until we end up with the definitions of events (or variables). Since each transformation is an identity, the consistency check of the original logical expression is equivalent to the consistency check of the leaf level. In other words, the conjunction of (the disjunctions of) the leaf level events is the same as the top event which represents the conjunction of the requirements to be analyzed.

Tree-Pruning Algorithms In the course of performing the transformation, it is always a good practice to simplify the tree based on logic constraints, i.e., prune the branches that contradict any known *true* branch. As a matter of fact, if a leaf contradicts any known *true* branches/leaves, then the tree can be pruned upward (i.e., towards the root) until the first *OR* gate is encountered. This is because, if a leaf is *false*, the conjunction of this leaf with any other leaves or subtrees must also be *false*. Therefore, they must be pruned from the tree. Conversely, if a leaf is *true*, the disjunction of this leaf with any other leaves or subtrees must be *true*. As a result, they also should be pruned from the tree to simplify the analysis. As a special case, if a *true* leaf appears later in the analysis, this later leaf should be pruned from the tree.

The tree-based graphical approach for within-group analysis of group “Operational Mode” is shown in Figure 4.4-1. In the figure, the numbers in the two highest levels of nodes represent

the logical expression numbers in Table 4.4-2. Satisfying the set of requirements at level 1 is equivalent to satisfying each individual requirement simultaneously (level 2). Then the level 2 nodes are expanded further until the basic definitions are reached. For example, according to the rule of tree expansion, box 6 is expanded, followed by box 1. During the expansion, any leaves/branches that contradict any known *true* leaves/branches must be pruned from the tree. For example, when expanding box 1, it is realized that $\neg VA$ is *true* from the previously expanded box 6, so VA is not *true*. As a result, VA should be pruned from the tree. Similarly, VD is pruned (dashed boxes in Figure 4.4-1). No other branches can be pruned in the example. The analysis shows that the set of requirements for group “Operational Mode” is consistent and can be simplified as in Table 4.4-3.

The within-group analysis for group “Driving Force” was not identified as necessary. If an inconsistency is detected during the within-group analysis of any group, it must be reported to the customer. Then the analyst and the customer must find a resolution and the requirements must be revised before any further effort is invested in the between-group analysis of requirements.

4.4.2.5 Tree-Based Graphical Approach (TBGA) vs. Fault Tree Analysis (FTA)

Despite the fact that the Tree-Based Graphical Approach (TBGA) looks similar to Fault Tree Analysis (FTA), there are fundamental differences between the two. First, FTA does not have logic operations such as implication (\Rightarrow) and equivalence (\Leftrightarrow). These two operations play significant roles in TBGA. As a result, no logical derivation from known facts to unknown facts exists in FTA. Second, the purpose of FTA is to identify, through back tracking, all the minimal cut sets (MCS) or prime implicants that lead to the top event^{[Yau95], [Yau98]}. In TBGA, the purpose is to check whether the top event *could happen* (i.e., the requirements are consistent) through logical reasoning. Third, FTA is used to identify the weaknesses of a system, so that measures can be taken to reduce the probability of the top event, while the graphical method is simply another way to simplify logic expressions and has nothing to do with probabilistic or numerical analyses at all.

Table 4.4-3 Example: Simplified but Equivalent Representation of Requirements
Using the Tree-Based Graphical Approach (TBGA)

	<i>TBGA-Simplified Requirements (Operational Mode)</i>
1	$(SS \vee_e CA \vee_e CS \vee_e CD) \wedge \neg VA \wedge \neg VD$
2	$SS \Rightarrow V_{ss} = 0$
3	$CS \Rightarrow V_{cs} = 44.4 m/s$
4	$CA \Rightarrow a_{ca} = 10 m/s^2$
5	$CD \Rightarrow a_{cd} = -10 m/s^2$

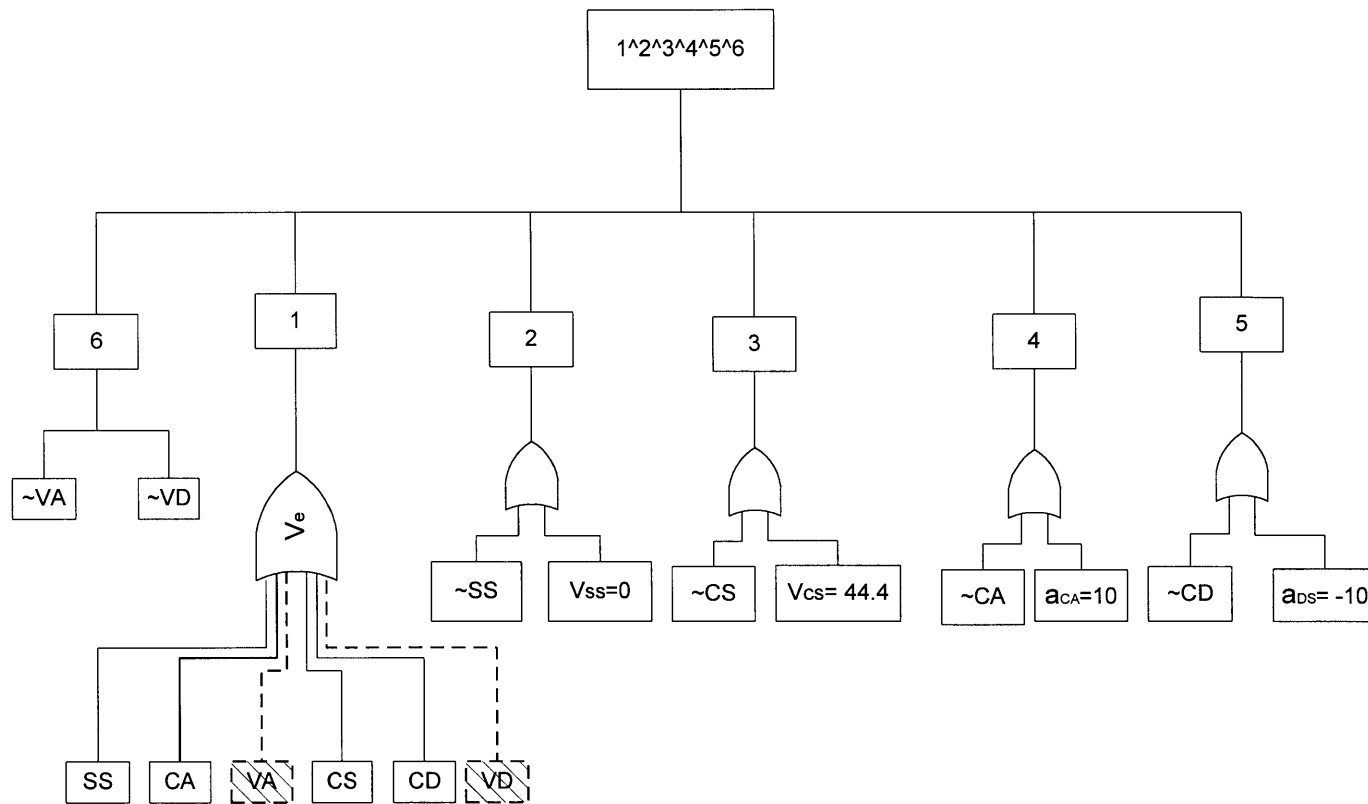


Figure 4.4-1 Within-Group Analysis for Group “Driving Force” in the Example

4.4.3 Between-Group Analysis

Through within-group analysis, we conclude that, the requirements are consistent within each group. However, the analysis does not rule out the possibility of inconsistency between the related groups.

To check consistency between the groups, we use dependency diagrams. These dependence diagrams indicate physically relationships between groups of the requirements. Therefore, requirements in one group may constrain those in other groups. In the dependency diagram, each group is represented by a solid circle. An arc connecting two groups indicates that the two groups are directly dependent, e.g., by physical laws. The analyst must identify all the arcs such that consistency can be completely checked. Between-group consistency is assured by analyzing all the arcs in the dependency diagram. Represented by dashed circles, intermediate groups may be needed to bridge two groups not directly related. In this example, the two groups are related by the law of energy conservation (Figure 4.4-2).

According to requirement 8, all energy generated by gas is converted into kinetic energy of the auto during the constant acceleration mode, i.e., $\eta=1.0$. This can be described by the differential equation.

$$v_F \cdot A(t) \cdot \rho \cdot q \cdot dt \cdot \eta = \frac{1}{2} m(v + dv)^2 - \frac{1}{2} mv^2 \quad (4.4-1)$$

where

ρ is the average density of gas at the nozzle,

m is the total mass of the auto, including the driver,

q is the energy generated by combustion of unit mass of gas.

The right hand side of Equation 4.4-1 represents the increment in the cars kinetic energy during time t and $t+dt$ while the left hand side of the equation represents the energy generated by the combustion of gas during the same time period. Equation 4.4-1 can be simplified as

$$\frac{v_F \rho q}{m} \cdot A(t) \cdot dt = v \cdot dv \quad (4.4-2)$$

Integrating this differential equation yields

$$v = \sqrt{\frac{2v_F \rho q}{m} \cdot \int_0^t A(t) dt} \quad (4.4-3)$$

From requirement 8 in group “Driving Force”, opening of the nozzle is proportional to time

$$A(t) = kt \quad (4.4-4)$$

Equation 4.4-3 can be simplified as

$$v = \sqrt{\frac{v_F \rho q k}{m}} \cdot t \quad (4.4-5)$$

Therefore, the acceleration that can be provided by the engine, a_{Engine} , which satisfies group “Driving Force” is

$$a_{\text{Engine}} = \frac{dv}{dt} = \sqrt{\frac{v_F \rho q k}{m}} \quad (4.4-6)$$

To satisfy the requirements in group “Operational Mode”, it is required that

$$a_{\text{Engine}} \geq a_{\text{CA}} \quad (4.4-7)$$

In other words, to satisfy the requirements in both groups, it is required that the maximum acceleration generated by the engine (a_{Engine}) exceed that specified by the requirements (a_{CA}). By substituting values for the parameters, the validity of the inequality can be checked. If the inequality is satisfied, the requirements between the two groups are consistent. Otherwise, they are not. It should be noted that Equation (4.4-6) is valid only when requirement 7 is satisfied ($A(t) = kt \leq A$ or $t \leq A/k$). In addition, through the between-group analysis, we can also make the requirements more complete, e.g., by discussing with the customer the addition of requirements for the values of ρ , q , m , etc.. Also omitted from this analysis are the requirements for constant deceleration that will surely impose additional requirements on the brakes.

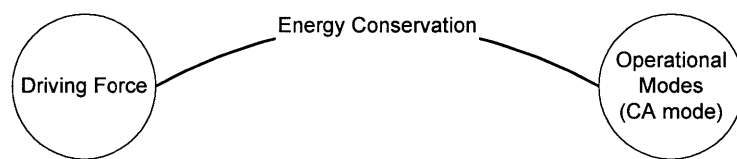


Figure 4.4-2 Between-Group Analysis for the Example

4.5 Four Approaches Applicable to Within-Group Analysis

In Section 4.4.2, a Tree-Based Graphical Approach (TBGA) is developed to perform within-group analysis of software requirements. The graphical approach is very user-friendly and can be learned by a novice very quickly. In this section, three other approaches are identified and shown to be applicable to the within-group analysis: the Purely Analytical Approach, the Truth-Table Approach, and the Dynamic Flowgraph Methodology (DFM). Please note that

- the first approach is taken directly from [Ince88], [Hoar69], [Lond75];
- the second approach is taken from a standard electrical engineering approach for hardware design of digital circuits^{[Ward90], [Shaw93]}. However, the application of the methodology here for software requirements analysis is entirely original;
- DFM is usually applied to solve multi-valued logic problems under time-dependent conditions. The author originally applies DFM to automate the within-group portion of the software requirements analysis.

4.5.1 A Simple Example

Application of the three approaches, as well as the Tree-Based Graphical Approach, will be illustrated by a simple example whose requirements are provided in column 2 of Table 4.5-1.

Three logical variables are identified in this simple example and are defined as

A = The computer system will be in state A

B = The computer system is in state B

C = The computer system is required to have property C .

Table 4.5-1 Simple Example: Requirements for a Computer System

	<i>Requirements</i>	<i>Equivalent Expressions</i>
1	A computer system can only be in two states, either A or B ;	$A \vee B$
2	If the computer system is in state B , then it is required to have property C ;	$B \Rightarrow C$
3	The computer system is not in state B .	$\neg B$

Table 4.5-2 Simple Example: the Purely Analytical Approach

<i>Expression #</i>	<i>Manipulation/Derivation</i>	<i>Derived from</i>
1.	$(A \vee B) \wedge (B \Rightarrow C) \wedge \neg B$	Requirements 1, 2, 3
2.	$\neg B$	(1)
3.	$A \vee B$	(1)
4.	A	(2, 3)
5.	$B \Rightarrow C$	(1)
6.	$\neg B \vee C$	(2 or 5)
7.	$A \wedge \neg B$	(2, 4)

Then, the requirements written in English are translated into formal logic respectively as shown in column 3 of Table 4.5-1. Mathematically, it requires $(A \vee B) \wedge (B \Rightarrow C) \wedge \neg B$ be *true*. Four approaches will be used to show the analysis.

4.5.2 The Purely Analytical Approach

The Purely Analytical Approach is discussed in [Ince88], [Hoar69], [Hoar71], and [Lond75]. This method is usually expressed in tabular form with three columns: the first column records the sequential order of the analysis, the second column represents the results of the analysis derived from the expressions listed in the third column. As a result, the expression numbers in the third column must be strictly less than those in the first column of the same row. In Table 4.5-2, row 1 means it is derived from the original requirements. Row 2 means $\neg B$ is derived from expression 1. In other words, the fact that $\neg B$ is *true* is derived from the fact that $(A \vee B) \wedge (B \Rightarrow C) \wedge \neg B$ is required to be *true* (refer to Theorem 3.2-1B in Chapter 3). Similarly, each of rows 3 and 5 is derived from expression 1. Row 4 is derived from expressions 2 and 3, that is, A must be *true* when both $\neg B$ and $A \vee B$ are *true* simultaneously. As a rule, if the numbers in the third column are separated by a comma, the expression in the row is derived from the expressions in the rows with corresponding numbers in column 1. Row 6 is *true*, either from rows 2 (i.e., if $\neg B$ is *true*, $\neg B \vee C$ is *true*) or 5 (i.e., $B \Rightarrow C$ is equivalent to $\neg B \vee C$ by definition). Finally, row 7 takes the conjunction of all the basic events in the second column of the table. Since no contradictions are found in the analysis, it means that the requirements are consistent. Mathematically, this means

$$(A \vee B) \wedge (B \Rightarrow C) \wedge \neg B = A \wedge \neg B \quad (4.5-1)$$

Now we can translate Equation (4.5-1) back into English with A and B defined in Section 4.5.1, and the requirements in Table 4.5-1 can be simplified as

“The computer system can only be in state A, not state B”.

Comparing with Table 4.5-1, it is noticed that requirement 2 has been removed from the set of requirements because it is irrelevant.

4.5.3 The Truth-Table Approach

The Truth-Table Approach is a standard electrical engineering method in designing and simplifying digital circuits. This approach checks for all possible permutations of truth values assigned to all the literals (i.e., logical variables) involved in the logical expression. For a logic expression containing n independent variables, there are 2^n rows in the truth table since each variable has two truth values, *true*(1) or *false* (0). Each row in the table corresponds to a distinct permutation of truth value assignments to the variables. For each permutation, first determine the truth values of the components in the logical expression, e.g., $A \vee B$, $B \Rightarrow C$, and $\neg B$ in this example, as shown in columns 4, 5, and 6 of Table 4.5-3. Then, determine the corresponding truth values of the logical expression, $TOP = (A \vee B) \wedge (B \Rightarrow C) \wedge \neg B$. Finally, check to see if there is any row in the truth table that satisfies the logical expression (i.e., making it be *true*). If this is the case, the logical expression is satisfiable. In other words, the corresponding requirements are consistent. Otherwise, the logical expression is not satisfiable or the corresponding requirements are inconsistent.

When the logical expression is satisfiable, there are two ways to find its simplified and equivalent expression based on the truth table. Picking all the rows that evaluate the logical expression to be *true*, we

- take the disjunction of the conjunctions of the variables (also known as the sum of products) and simplify the disjunction analytically. Each of the conjunction corresponds to a row that evaluates the logical expression as *true*. In the conjunction, if a logical variable is *true* (or 1) in that row of the table, the variable itself is used, otherwise its negation is used;
- take the disjunction of the conjunctions of the variables and simplify the disjunction using Karnaugh maps. This graphical method is recommended if six or fewer logical variables are involved in the logical expression. Readers interested in the method should refer to [Shaw93], [Rose95], [Ward90] for a more detailed discussion.

The advantage of the truth-table approach is that it is a mechanical process. The disadvantages are

- the approach requires exponential time, i.e., for an expression involving n independent variables, the truth table contains 2^n rows. This is unsatisfactory for $n > 3$;
- most of the rows in the truth table are useless under many circumstances, i.e., they can not satisfy the logical expression. Therefore, the effort to create these rows is futile. However, they can not be avoided since one does not know whether they are futile until after they have been developed;
- with 2^n rows in a truth table, it can be error-prone if evaluated manually.

In this example, when the triple $(A, B, C) = (1, 0, 0)$ or $(1, 0, 1)$, the top expression is *true*. Taking the disjunction of the conjunctions, the logical expression is simplified as

$$(A \vee B) \wedge (B \Rightarrow C) \wedge \neg B = (A \wedge \neg B \wedge \neg C) \vee (A \wedge \neg B \wedge C) \quad (4.5-2)$$

or

$$(A \vee B) \wedge (B \Rightarrow C) \wedge \neg B = A \wedge \neg B \quad (4.5-3)$$

Not surprisingly, Equation (4.5-3) is exactly the same as Equation (4.5-1).

Table 4.5-3 Simple Example: the Truth-Table Approach

A	B	C	$\neg B$	$A \vee B$	$B \Rightarrow C$	$TOP = (A \vee B) \wedge (B \Rightarrow C) \wedge \neg B$
0	0	0	1	0	1	0
0	0	1	1	0	1	0
0	1	0	0	1	0	0
0	1	1	0	1	1	0
1	0	0	1	1	1	1
1	0	1	1	1	1	1
1	1	0	0	1	0	0
1	1	1	0	1	1	0

4.5.4 The Dynamic Flowgraph Methodology (DFM) Approach

4.5.4.1 Features of DFM

Based on the Logic Flowgraph Methodology (LFM)^[Guar84, 88, 90], a method for analyzing processes with feedback and feedforward control loops in steady state, the Dynamic Flowgraph Methodology (DFM) aims to extend LFM to systematically model and analyze the hardware and software components of time-dependent embedded systems. According to [Yau95], two features were added to LFM to address the issues relevant to an embedded system:

- the ability to deal with the dynamic aspect of the system, since time transitions are often present in the software and hardware of embedded system;
- the ability to identify and represent the continuous physical and the discontinuous logic influences present in the system, i.e., use of multi-valued logic.

Normally, DFM works in a deductive fashion, i.e., it uses reverse causality backtracking of a top event to find the *prime implicants*. Corresponding to the minimum cut sets (MCS) of a binary fault tree, the prime implicants of a multi-valued logic tree, are the minimum combinations of basic events whose simultaneous occurrence will ensure the occurrence of the top event. The prime implicants are the unique failure modes of non-coherent systems, i.e., systems *NOT* gates as opposed to simply *AND-OR* logic^[Hen92].






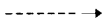
Through use of directed graphs, with relations of causality and conditional switching actions represented by edges that connect nodes and operators in the diagram, a DFM model integrates three types of networks. The *Time-Transition Network* describes the sequence in which software modules (e.g., subroutines) are executed, the *Causality Network* shows the functional relationships among key hardware and software parameters, and the *Conditioning Network* models discrete software behavior caused by conditional branching or discontinuous hardware performance due to component failure. The building blocks of these networks

include *Process Variable Nodes*, *Condition Nodes*, *Causality Edges*, *Condition Edges*, *Transfer Boxes*, *Transition Boxes*, and their associated decision tables (Table 4.5-4). Process Variable Nodes are used to represent essential physical or hardware variables of the digital system. Like process variable nodes, Condition Nodes represent physical or software parameters that identify component failures, changes of process operation regimes and modes, or software conditional branching. Condition Nodes can be associated with transition for this purpose. Causality Edges are employed to connect Process Variable Nodes which have a cause-and-effect relationship between the variables. Condition Edges are used to model discrete behavior of the system. They link parameter nodes to transfer boxes, showing the possibility of using a different transfer function to map input variable states to output variable states. A Transfer Box models a transfer function between Process Variable Nodes. In spite of indicating a cause-and-effect relationship similar to that of a Transfer Box, a Transition Box differs from a Transfer Box in that a time delay for the transition is assumed to exist between the time when the input variable states become true and the time when the corresponding output variable states are reached. As an extension of a truth table, a decision table allows each variable to be represented by any number of states, instead of being limited to the binary logic of 0 (*false*) and 1 (*true*). The decision tables are constructed from empirical knowledge, physical equations, logic relations, software, or pseudo code.

4.5.4.2 Automated Tool

The directed graph approach of DFM is implemented by a toolset which allows the system model to be constructed and analyzed automatically^[ASCA].

Table 4.5-4 Building Blocks of a DFM Model

<i>Name of Building Block</i>	<i>DFM Graphical Representation</i>
Process Variable Node	
Condition Node	
Transfer Box	
Transition Box	
Causality Edge	
Condition Edge	

The toolset consists of a model editor and an analysis engine. The model editor provides a graphical environment for the user to construct a model representing the system of interest. A DFM model discretizes key process parameters, represented by nodes, into states. The domain of each node is a set of consecutive, non-negative integers usually starting with zero. These nodes are linked together via connectors (e.g., a transfer box) to represent, in network form, the causal and temporal relationships of the system being analyzed. The details of the underlying system parametric functions are expressed in terms of multi-valued logic decision tables associated with connectors. Once the model is set up, one can use the DFM analysis engine to automatically search for prime implicants of a given top event. A typical execution of a DFM analysis is a two-step process. The two basic steps are:

- Step 1 Build a model of the system for which a safety analysis is required. The model consists of both the controlling software and the entities being controlled. Multi-valued logic relations are used to discretize continuous domain quantities, including time;
- Step 2 Use the model developed in Step 1 to systematically search for the prime implicants of a top event, desirable or undesirable, depending on the objectives of the analysis.

One mode of a DFM analysis is conducted by back tracking through the model to identify, in “reverse causality”, the combinations of causes that bring about system level top events. Such an analysis is based on a series of intermediate decision tables. An intermediate decision table is an equivalent to a truth table in the binary logic system, which is modeled as a set of logic gates in conventional fault tree analysis. As Yau emphasizes ([Yau98]), “a single DFM model can be used to analyze an unlimited number of system top events” since the logical relations “in the decision tables are sufficient to generate all the associated prime implicants”.

The advantages of DFM over Software Fault Tree Analysis (SFTA)^[Leve83] include its capability to deal with multi-state logic (as well as binary logic) and dynamics. In other words, DFM provides a multi-valued, timed fault tree which can be viewed as a series of

snapshots of conventional fault trees. In the conventional fault tree, only the probabilities of occurrence of events may be time-dependent, while in the DFM model, in addition to time-dependent probabilities of occurrence of events, the events themselves are also time-dependent. And DFM is applicable to non-coherent situations in which *NOT* gates (or their equivalents) are present, while conventional FTA does not apply to these situations.

4.5.4.3 Previous Applications of DFM

DFM was initiated and designed to solve multi-valued logic and temporal problems, e.g., finding coding errors in software. The applicability of DFM has been demonstrated in analyzing software-controlled systems in nuclear applications^[Guar96] and aerospace applications^[Yau95], as well as a simple embedded system^[Garr95]. It should be noted that all the applications of DFM in software have occurred at late stages of the software development process, i.e., after the software has been developed.

4.5.4.4 Use of DFM to Find Prime Implicants of Logic Expressions

A new application of DFM is proposed in this dissertation, that is, it is applied in software requirements analysis where only static, binary logic is needed. The logic expressions in this section can be viewed as mathematical translations of the software requirements.

Four test cases, to which we know the answers, have been designed to cover a variety of situations. These test cases are to find the prime implicants

- of top event $TOP = \neg A \wedge A$ (Test Case 1);
- of top event $TOP = \neg A \vee A$ (Test Case 2);
- of top event $TOP = (A \wedge B) \vee \neg A$ (Test Case 3);
- of top event $TOP = (A \vee B) \wedge \neg A$ (Test Case 4).

For test case 1, no prime implicants exist. Two prime implicants exist for each of test cases 2 and 3. Only one prime implicant exists for test case 4. The known answers can be used to

check with those obtained from applying the DFM toolset. As summarized in Table 4.5-5, the examples indicate that DFM can find all the prime implicants of the logic expressions.

4.5.4.5 Application of DFM to Requirements Analysis

In this dissertation, the author will apply DFM to Software Requirements Analysis (SRA), i.e., to check whether software requirements are consistent or not. The purpose of the application of DFM in software requirements analysis is two-fold. First, it provides a mechanical tool for requirements analysis, minimizing reliance on knowledge of formal logic to simply understanding logic gates (i.e., *NOT*, *AND*, *OR*). Second, it provides an independent double check of the Tree-Based Graphical Approach (TBGA) introduced in Section 4.4.2. The output of the DFM tool is a set of prime implicants. If the set is an empty set, it implies that the software requirements are inconsistent because an empty set of prime implicants means that no combinations of the basic events will generate the top event. In the case of SRA, the top event is the conjunction of all requirements. Conversely, if the set is not empty, the requirements are consistent. Therefore, unlike previous applications of DFM, the SRA application of DFM uses only binary logic and does not involve temporal effects at all.

The analysis will be discussed in detail with the simple example of top event $TOP = (A \vee B) \wedge (B \Rightarrow C) \wedge \neg B$. As a rule in this thesis, a DFM component (e.g., node, transfer box) is typed in `courier` font and a logical variable is typed in *italic* font.

Table 4.5-5 Prime Implicants Generated by DFM for Four Test Cases

Test Case Number	Logical Expressions	Prime Implicants Found by DFM	Theoretical Prime Implicants
1	$\neg A \wedge A$	no	not exist
2	$\neg A \vee A$	exist ⁷	$A, \neg A$
3	$(A \wedge B) \vee \neg A$	$\neg A, B$	$\neg A, B$
4	$(A \vee B) \wedge \neg A$	$\neg A \wedge B$	$\neg A \wedge B$

⁷ The current version of DFM can not find the prime implicants of a tautology though it can detect the existence of prime implicants.

To start the DFM analysis, we first build nodes A, B, and C (bottom circles in Figure 4.5-1) representing logical variables A , B , and C respectively, whose truth values can either be 0 (*false*) or 1 (*true*). Then, we build intermediate nodes NOT_B, R1, and R2 representing logical expressions $\neg B$, $A \vee B$ and $\neg B \vee C$ (or $B \Rightarrow C$), respectively. The truth values of these nodes are either 0 (*false*) or 1 (*true*). It should be noted that the nodes NOT_B and B can not take truth values arbitrarily. In fact, the truth value of NOT_B must be exactly the opposite of that of B, and vice versa. This logical negation is reflected by transfer box 1 (marked with \sim). The input to the transfer box is node B and the output is node NOT_B. The input-output relation is determined by the decision table of the transfer box (Table 4.5-6). In Figure 4.5-1, the logical expressions of $A \vee B$ and $\neg B \vee C$ (or $B \Rightarrow C$) are represented by transfer boxes 2 and 3, whose decision tables are shown in Tables 4.5-7 and 4.5-8, respectively. Finally, we build node TOP, whose truth values are 0 (*false*) or 1 (*true*), representing the top event. The logical conjunction is modeled using transfer box 4 (marked with \wedge). The inputs to the transfer box are nodes R1, R2, and NOT_B. The decision table of transfer box 4 is shown in Table 4.5-9.

Once the graphical network of the DFM model is set up, the analysis engine of DFM can be initiated. In the example, the top event is chosen to be TOP=1, which stands for requiring the top event to be *true*. As expected, the analysis using DFM correctly finds the prime implicant to be $A \wedge \neg B$. The screen dump of the DFM result is illustrated in Figure 4.5-2.

Table 4.5-6 Simple Example: Decision Table for Transfer Box 1

B	NOT_B
0	1
1	0

Table 4.5-7 Simple Example: Decision Table for Transfer Box 2

A	B	R1
0	0	0
0	1	1
1	0	1
1	1	1

Table 4.5-8 Simple Example: Decision Table for Transfer Box 3

C	NOT_B	R2
0	0	0
0	1	1
1	0	1
1	1	1

Table 4.5-9 Simple Example: Decision Table for Transfer Box 4

R1	NOT_B	R2	TOP
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

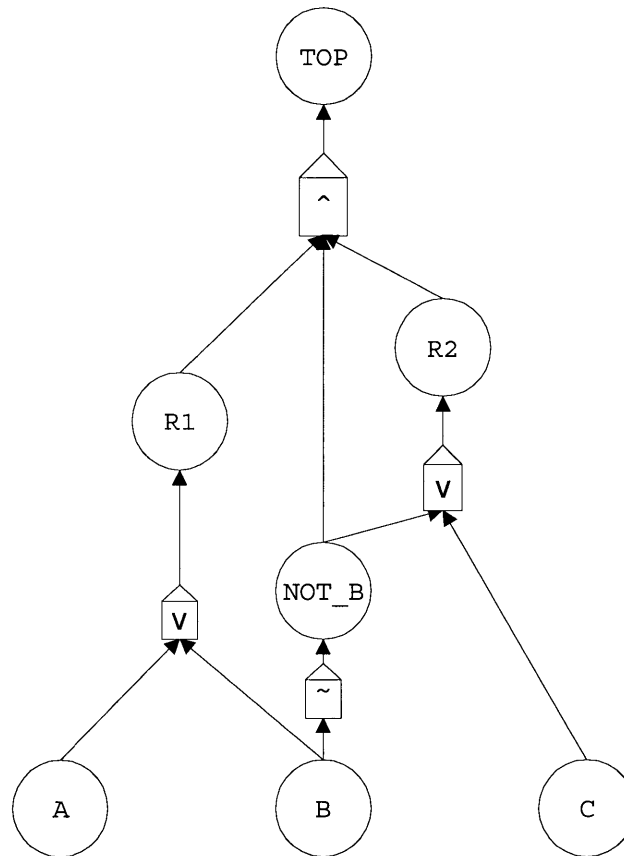


Figure 4.5-1 DFM Model for the Example

```

Prime - Notepad
File Edit Search Help
For the top event:

At time 0 , TOP = T (True)

There are 1 prime implicants

Prime Implicant #1
  At time 0 , A = 1      AND
  At time 0 , B = 0
  
```

Figure 4.5-2 DFM Result for the Example (Screen Dump)

4.5.5 Tree-Based Graphical Approach

The application of the Tree-Based Graphical Approach (TBGA) to this example is shown in Figure 4.5-3. After expanding the tree, leaf B in the middle branch is pruned (shaded with lines) since it contradicts the upper left leaf $\neg B$, which is required to be *true*. The fact that the upper left leaf $\neg B$ is *true* leads to pruning leaves $\neg B$ and C , which are connected via a disjunctive operation (shaded with dots). As a result, their disjunction is *true*. Therefore, by taking the conjunction of the leaves, the original expression can be equivalently simplified to $A \wedge \neg B$.

Figure 4.5-4 shows the same analysis following a different order of branch selection at subsequent steps, i.e., $B \Rightarrow C$ is chosen before $A \vee B$. Two facts are observed from Figures 4.5-3 and 4.5-4.

- As expected, the final result is independent of the order of branch selection, e.g., the original expression is simplified to $A \wedge \neg B$ in Figure 4.5-4, which is identical to the result from Figure 4.5-3;
- The order to branch selection does determine the order in which the branches get pruned, e.g., leaves $\neg B$ and C are pruned before leaf B in Figure 4.5-4 while they are pruned after leaf B in Figure 4.5-3. As a result, the order will affect the complexity of the tree structure in the course of the analysis.

4.5.6 Summary

The tree-Based Graphical Approach (TBGA) has been developed for the within-group analysis of software requirements analysis. The approach is based on graphical representation of formal logic (Table 4.2-1). Three other approaches are identified and applied for the same purpose: the Analytical Approach, the Truth-Table Approach, and the DFM Approach. These approaches can act as an independent double check of the graphical approach. The features of the four approaches are summarized in Table 4.5-10.

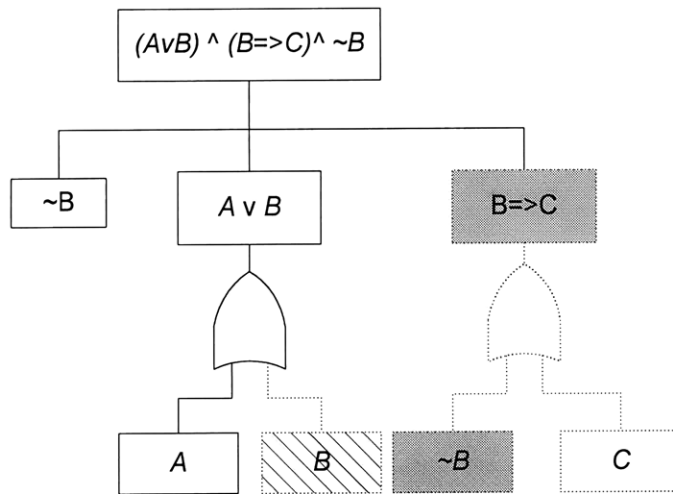


Figure 4.5-3 Tree-Based Graphical Approach for the Example

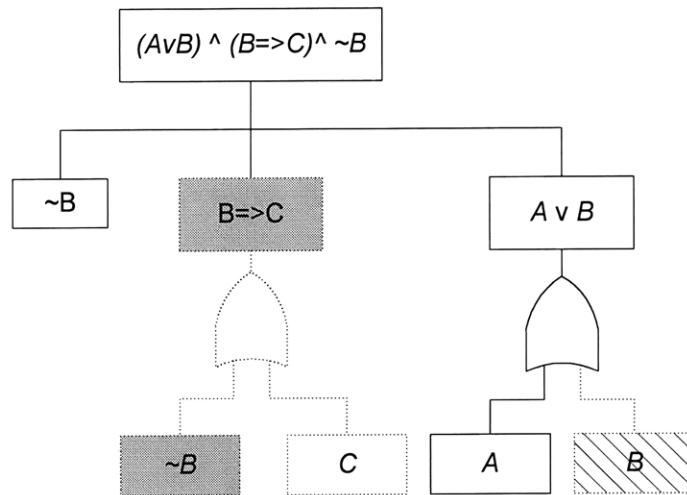


Figure 4.5-4 Order-Independence of the Tree-Based Graphical Approach

Table 4.5-10 Comparison of the Four Approaches in Within-Group Analysis

<i>Approaches</i>	<i>Features</i>
TBGA	straightforward, polynomial time
DFM Approach	automatic tool, but exponential time
Truth-Table Approach	exponential time
Analytical Approach	mathematically-sophisticated

Chapter 5

Application of the Proposed Approach to a Case Study

5.1 Description of the Case Study

The case study is the MIT-SNL Period-Generated Minimum Time Control Laws taken from [Bern89]. The ultimate goal of the control laws was to rapidly increase the neutronic power of a normally dormant space reactor by many orders of magnitude in a few seconds and to conform to a specified power trajectory. The control laws were extensively and successfully experimented with on both the MITR-II reactor at MIT and the Annular Core Research Reactor (ACRR) at Sandia National Laboratories (SNL). The experiments at MITR-II were for the initial testing of the control laws. ACRR was used as a prototype reactor to demonstrate that the control laws were indeed feasible to satisfy its ultimate goal.

5.1.1 MIT-SNL Period-Generated Minimum Time Control Laws

A good model describing the neutronic behavior for small to medium size nuclear reactor is the point kinetics model as shown in Equations (5.1-1) and (5.1-2).

$$\frac{dT(t)}{dt} = \frac{\rho(t) - \bar{\beta}}{l^*} \cdot T(t) + \sum_{i=1}^N \lambda_i \cdot C_i(t) \quad (5.1-1)$$

$$\frac{dC_i(t)}{dt} = \frac{\bar{\beta}_i}{l^*} \cdot T(t) - \lambda_i \cdot C_i(t) \quad \text{for } i=1, 2, \dots, N \quad (5.1-2)$$

where

$T(t)$ is the amplitude function, a weighted integral of all neutrons in the reactor

and roughly proportional to reactor power,

$\rho(t)$ is the reactivity,

$\bar{\beta}$ is the effective delayed neutron fraction,

$\bar{\beta}_i$ is the effective fractional yield of the i^{th} group of delayed neutrons,

λ_i is the decay constant for the i^{th} precursor group,

$C_i(t)$ is the concentration of the i^{th} precursor group,

l^* is the prompt neutron lifetime,

N is the number of groups of delayed neutrons, including photoneutrons.

Usually, N is set to 6.

With the instantaneous reactor period ($\tau=1/\omega$) defined in Equation (5.1-3) and several pages of algebra, Bernard derived the dynamic period equation^[Bern89]. The equation is reproduced as Equation (5.1-4) below.

$$\dot{T}(t) = \omega(t) \cdot T(t) \quad (5.1-3)$$

$$\tau(t) = \frac{\bar{\beta} - \rho(t) + l^* \left[\frac{\dot{\omega}(t)}{\omega(t)} + \omega(t) + \lambda_e(t) - \frac{\dot{\lambda}_e(t)}{\lambda_e(t)} \right]}{\dot{\rho}(t) + \lambda_e(t)\rho(t) + \frac{\dot{\lambda}_e(t)}{\lambda_e(t)}(\bar{\beta} - \rho(t))} \quad (5.1-4)$$

where

$\omega(t)$ is the inverse of the specified reactor period,

$\dot{\omega}(t)$ is the rate of change of the inverse of the specified reactor period,

$\lambda_e(t)$ is the standard, effective, multi-group decay constant as defined in

Equation (5.1-5).

$$\lambda_e(t) \equiv \frac{\sum_{i=1}^N \lambda_i \cdot C_i(t)}{\sum_{i=1}^N C_i(t)} \quad (5.1-5)$$

Rewriting the dynamic period equation, i.e., solving for $\dot{\rho}(t)$ in Equation (5.1-4), yields the MIT-SNL Period-Generated Minimum Time Control Laws. The standard form of the laws is given as Equation (5.1-6). The Laws also have an alternate form which is mathematically equivalent to the standard form. Readers interested in the topic should refer to [Bern89].

$$\begin{aligned} \dot{\rho}_c(t) = & (\bar{\beta} - \rho(t)) \cdot \omega(t) - \lambda_e(t) \rho(t) - \frac{\dot{\lambda}_e(t)}{\lambda_e(t)} (\bar{\beta} - \rho(t)) - \dot{\rho}_f(t) \\ & + I^* \dot{\omega}(t) + I^* [\omega^2(t) + \lambda_e(t) \omega(t) - \frac{\dot{\lambda}_e(t)}{\lambda_e(t)} \omega(t)] \end{aligned} \quad (5.1-6)$$

where]

$\dot{\rho}(t) = \dot{\rho}_c(t) + \dot{\rho}_f(t)$, is the sum of the rate of change of reactivity from control devices ($\dot{\rho}_c(t)$) and feedback ($\dot{\rho}_f(t)$) such as fuel Doppler effect.

The MIT-SNL Control Laws are unique in two aspects. First, compared with other reactor control laws that use reactivity as the control signal, the MIT-SNL laws use the rate of change of reactivity as the control signal. This is advantageous when a rapid transient is desirable, since reactivity is a function of control rod position, which can not be changed instantaneously, while the rate of reactivity insertion corresponds to the control rod speed, which can be changed on demand. As a result, controllers designed in terms of the rate of change of reactivity are very responsive and achieve any desired power profile. Second, they are closed-form expressions for the time-optimal control of power of reactors which subject to a limitation on the allowed period.

5.1.2 Annular Core Research Reactor

The ACRR is a modified TRIGA nuclear reactor that uses $\text{UO}_2\text{-BeO}$ fuel elements with a U-235 enrichment of 35%. Its annular-shaped core is formed by 236 fuel elements arranged in a hexagonal pattern around the 23 cm diameter irradiation cavity (Figure 5.1-1). Noted for its exceptionally large negative reactivity coefficient for the fuel, the reactor is controlled by two safety rods, six control rods, and three poison transient rods. The safety rods and the control rods are made of B_4C followed by fuel. The transient rods are also made of B_4C but followed by air (a hollow aluminum tube with both ends sealed). Operated as a bank and driven by variable speed stepping motors, the transient rods were used to conduct the experiments.

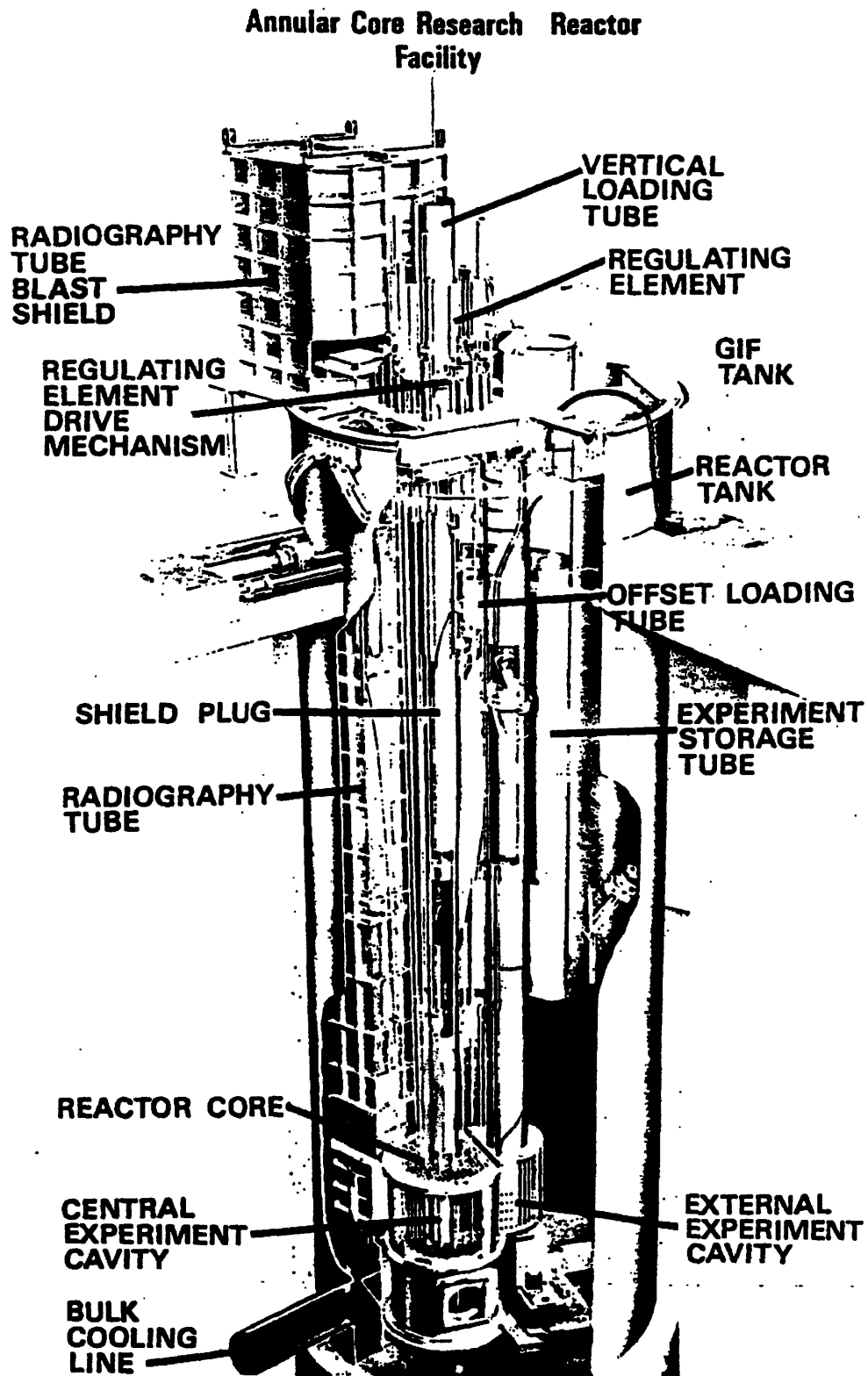


Figure 5.1-1 Isometric View of Annular Core Research Reactor

5.2 Requirements Collection

The requirements for the MIT-SNL control software are scattered throughout the documentation. This is because, in this case study, the “customer” and the developer of the software are the same person and the software itself was developed in an informal way, e.g., the developer started writing the code without an explicit set of requirements or specification. The developer claimed the correctness of the code by (informal) inspection and extensive testing. As a result, the documents do not contain an explicit list of all the requirements. After careful reading, extensive searching and frequent discussion with the developer, the author collected all the requirements in the report (five pages), including two additional requirements that are not listed in [Bern89] explicitly, but used by default by the developer. Listed below is the collection of requirements. The first column represents the page number in the report of the corresponding requirements in the second column.

Page Requirements

- | | |
|----|---|
| 4 | Principal objective was to develop a robust control technique that would permit a neutronic power to be raised by five to seven orders of magnitude both in a few seconds and without significant overshoot. |
| 9 | The reactor operates in either a steady-state or a pulsed mode. For steady-state operation, the maximum allowed power level is 2 MWt. For pulsed operation, there is no restriction of the power. Rather there is a limit of 500 MJ total energy per pulse and one of 1800°C on the fuel temperature. |
| 11 | The limiting condition for the ACRR is not a specific power level but rather the total energy produced during the transient. |

126 The transient rods are normally operated as a bank (gang) and, as mentioned , can be moved either pneumatically or via stepper motors. Relative to the first method, nitrogen gas is used to pressurize each rod's piston chamber and thereby eject the three rods in times as short as 55 milliseconds. When operated in this manner, the maximum allowed reactivity insertion is 3β ...

(127)The alternative means of moving the transient rods is the "Transient Rod Withdrawal" or TRW method. Using this approach, the transient rods are raised by fork-shaped devices which are each connected to variable speed stepper motors via an electromagnet and a rack and pinion gear. The maximum speed of withdrawing the transient rods without any chance of magnet separation is 750 pulses per second. ... If moved from the "full-in" to "full-out" position, the total reactivity associated with the transient rods is 4.25β . Transient rod location on the ACRR is given in "position units" with "full-in" being 2031 units and "full-out" being 7464 units. Every pulse delivered to the stepper motors moves the rods 2.661 units.

127 The scram circuit will cause an automatic shutdown to prevent either the energy release from exceeding 500 MJ or the fuel temperature from rising above 1800°C.

130 Relative to the actual conduct of experiments involving the MIT-SNL minimum time laws, an upper limit on the allowable frequency of 1100 Hz was selected. This figure was chosen because its is the value listed as acceptable for the TRW mode of operation in the ACRR's Safety Analysis Report and because the range of frequencies needed for the testing of the MIT-SNL minimum time laws was approximately 400-600 Hz.

130-2 Testing of the MIT-SNL Period-Generated Minimum Time Control Laws on Sandia's ACRR was approved subject to the following:

- 1) The instrumentation that formed the ACRR's safety system would not also be used for the controller. The safety system was to be maintained as a separate, independent entity.
- 2) Signals from the digital computer would replace those from the existing eight-step programmable controller. The hardware and circuitry previously used for control of the ACRR would be retained.
- 3) An independent, hard-wired over-speed circuit would be installed to preclude withdrawal for the transient rod bank at frequencies in excess of 1100 Hz.
- 4) A "Loss of Computer Enable" circuit which would result in an insertion signal to the transient bank should there be a malfunction of the computer or the associated hardware would be installed.
- 5) The following restrictions would be incorporated in the software:
 - Absolute limit on the net reactivity, maximum neutronic power level, minimum observed reactor period, and fuel temperature.
 - Use of the standard, sufficient reactivity constraint in a supervisory role.
 - Use of the absolute, sufficient energy constraint in a supervisory role.
- 6) Specification of a transient duration limit following which control would revert to manual.
- 7) A requirement to test both the over-speed trip and the "loss of enable" circuit prior to each set of runs and periodically during the runs.

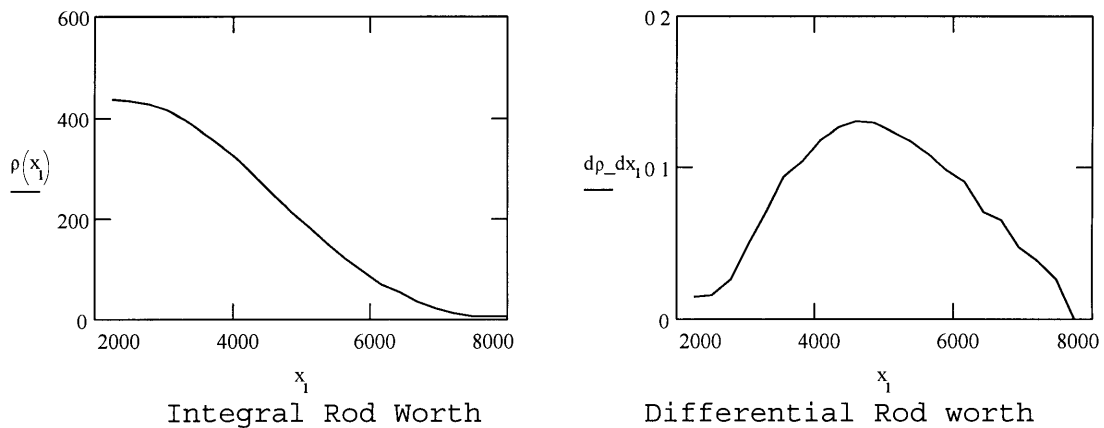
- 8) The pneumatic mode of transient rod bank withdrawal would be disabled and officially tagged out so as to preclude its inadvertent use.
- 9) Control would never be transferred from manual to automatic with reactor already on a positive period. The purpose of this restriction was to limit the excess reactivity to that inserted by the transient rod bank.
- 10) Reactor operation would at all times be monitored by licensed personnel.
- 140 (ACRR) The design specifications imposed on the interface controller board were that it be possible to move the stepper motors over the frequency range of -1000 Hz to +1000 Hz, that the deviation of the output frequency from the required frequency not exceed 5% and that the over frequency trip be set at 1100 Hz. (Note: There was a deadband over the range of ± 15.26 Hz ...)
- 141 ... However, despite its having been designed to accommodate demanded frequencies of as much as 1000 Hz, the MIT-SNL experiments were restricted to 400 Hz.
- 178 ... the program's ultimate goal ... was to raise a reactor's neutronic power by five-six orders of magnitude both on a period of several tenths of a second and without overshoot ...
- 186 ... there was a limit of 0.20 second on the minimum allowed period ... applied to ... the MIT-SNL laws.
- 234 The objective of the MIT-SNL Laws is to determine the rate of change of reactivity ($\dot{\rho}_c$) that will cause a reactor's neutronic power to vary on a specified period. The decisions

of these laws are most readily implemented if the differential reactivity worth ($\frac{d\rho}{dx}$) of the associated control device is at a maximum.

$$\dot{\rho}_c = \frac{d\rho}{dt} = \frac{d\rho}{dx} \cdot \frac{dx}{dt}$$

Clearly, if the control device's differential reactivity worth, $\frac{d\rho}{dx}$, is large, then the speed at which the control device must be moved, $\frac{dx}{dt}$, can be minimized. So doing is desirable because it will reduce wear on the variable speed stepper motors. In contrast, if the differential worth is low, then the stepper motor must be operated at high frequency. Not only will this accelerate wear but it also risks the possibility of causing the motor to stall.

236 (Transient Rod Worth, in cents)



247 ... The power peaked ... , well within the permitted 5% band.

266 It was used as a supervisory routine during all of the ACRR control experiments to ensure that the reactor's energy output would not exceed the limiting value of 350 MJ.

Further discussion with the customer (also the developer, in this case) added more requirements/restrictions to the set of requirements collected from [Bern89].

- MIT-SNL Period-Generated Minimum Time Control Laws should be used in the control software.
- To control reactor, no super-prompt criticality is allowed, e.g., $\rho < \beta$. (This requirement was later removed by the customer.)
- To satisfy Technical Specifications of the ACRR, the reactor period can not fall below 0.3 second

5.3 Requirements Grouping

Nine groups were identified, including one group for general information and one for non-software. Seven other groups were Power Multiplication, Operational Mode, Fuel Temperature, Reactor Period, Reactivity, Motor Frequency, and Rod Velocity.

5.3.1 General Information

Page Requirements

130-2

- 5) The following restrictions would be incorporated in the software:
 - Use of the standard, sufficient reactivity constraint in a supervisory role.
 - Use of the absolute, sufficient energy constraint in a supervisory role.
- MIT-SNL Period-Generated Minimum Time Control Laws should be used in the control software.

5.3.2 Power Multiplication Group

- 4 Principal objective was to develop a robust control technique that would permit a neutronic power to be raised by five to seven orders of magnitude⁸ both in a few seconds and without significant overshoot;

130-2

- 5) (There will be an) absolute limit on ... the maximum neutronic power level.

⁸From an engineer's point of view, this ambiguous, descriptive statement can be tolerated, but from the requirements' point of view, this ambiguity is unacceptable.

178 ... the program's ultimate goal ... was to raise a reactor's
neutronic power by five-six orders of magnitude both on a
period of several tenths of a second and without overshoot ...

247 ... The power peaked ... , well within the permitted 5% band.

5.3.3 Operational Mode Group

9 The reactor operates in either a steady-state or a pulsed
mode. For steady-state operation, the maximum allowed power
level is 2 MWt. For pulsed operation, there is no restriction
of the power. Rather, there is a limit of 500 MJ total energy
per pulse ...

11 The limiting condition for the ACRR is not a specific power
level but rather the total energy produced during the
transient.

127 The scram circuit will cause an automatic shutdown to prevent
... energy release from exceeding 500 MJ ...

266 It was used as a supervisory routine during all of the ACRR
control experiments to ensure that the reactor's energy output
would not exceed the limiting value of 350 MJ.

5.3.4 Fuel Temperature Group

9 For pulsed operation, there is a limit ... of 1800°C on the fuel
temperature.

127 The scram circuit will cause an automatic shutdown to prevent
... the fuel temperature from rising above 1800°C.

130-2

5) (There will be an) absolute limit on ... fuel temperature.

5.3.5 Reactor Period Group

130-2

5) (There will be an) absolute limit on ... the minimum observed reactor period.

186 ... there was a limit of 0.20 second on the minimum allowed period ... applied to ... the MIT-SNL laws.

- To satisfy Technical Specifications of the ACRR, the reactor period can not fall below 0.3 second.

5.3.6 Reactivity Group

126 The transient rods are normally operated as a bank (gang) and, as mentioned, can be moved either pneumatically or via stepper motors. Relative to the first method, nitrogen gas is used to pressurize each rod's piston chamber and thereby eject the three rods in times as short as 55 milliseconds. when operated in this manner, the maximum allowed reactivity insertion is 3β ...

(127) The alternative means of moving the transient rods is the "Transient Rod Withdrawal" or TRW method. Using this approach, the transient rods are raised by fork-shaped devices which are each connected to variable speed stepper motors via an electromagnet and a rack and pinion gear. The maximum speed of withdrawing the transient rods without any chance of magnet separation is 750 pulses per second. ... If moved from the "full-in" to "full-out" position, the total reactivity associated with the transient rods is 4.25β .

130-2

- 5) (There will be an) absolute limit on the net reactivity.
- 8) The pneumatic mode of transient rod band withdrawal would be disabled and officially tagged out so as to preclude its inadvertent use.

236 (Transient Rod Worth, in cents)

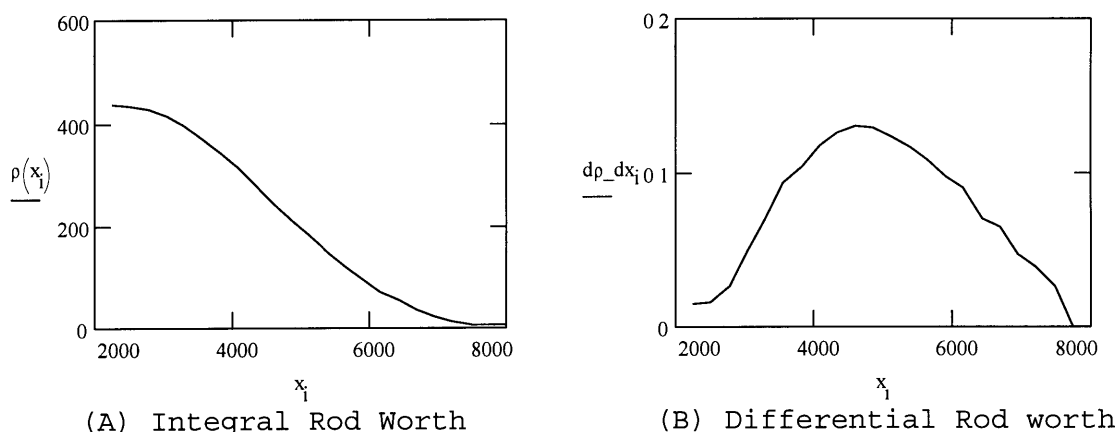


Figure 5.2-1 Transient Rod Worth

5.3.7 Motor Frequency Group

(127) The alternative means of moving the transient rods is the "Transient Rod Withdrawal" or TRW method. Using this approach, the transient rods are raised by fork-shaped devices which are each connected to variable speed stepper motors via an electromagnet and a rack and pinion gear. The maximum speed of withdrawing the transient rods without any chance of magnet separation is 750 pulses per second. ... If moved from the "full-in" to "full-out" position, the total reactivity associated with the transient rods is 4.25β . Transient rod location on the ACRR is given in "position units" with "full-in" being 2031 units and "full-out" being 7464 units. Every

pulse delivered to the stepper motors moves the rods 2.661 units.

130 Relative to the actual conduct of experiments involving the MIT-SNL minimum time laws, an upper limit on the allowable frequency of 1100 Hz was selected. This figure was chosen because its is the value listed as acceptable for the TRW mode of operation in the ACRR's Safety Analysis Report and because the range of frequencies needed for the testing of the MIT-SNL minimum time laws was approximately 400-600 Hz.

140 (ACRR) The design specifications imposed on the interface controller board were that it be possible to move the stepper motors over the frequency range of -1000 Hz to +1000 Hz, that the deviation of the output frequency form the required frequency not exceed 5% and that the over frequency trip be set at 1100 Hz. (Note: There was a deadband over the range of ± 15.26 Hz ...)

141 ... However, despite its having been designed to accommodate demanded frequencies of as much as 1000 Hz, the MIT-SNL experiments were restricted to 400 Hz.

5.3.8 Rod Velocity Group

127 Transient rod location on the ACRR is given in "position units" with "full-in" being 2031 units and "full-out" being 7464 units. Every pulse delivered to the stepper motors moves the rods 2.661 units.

234 The objective of the MIT-SNL Laws is to determine the rate of change of reactivity ($\dot{\rho}_c$) that will cause a reactor's neutronic power to vary on a specified period. The decisions

of these laws are most readily implemented if the differential reactivity worth ($\frac{d\rho}{dx}$) of the associated control device is at a maximum.

$$\dot{\rho}_c = \frac{d\rho}{dt} = \frac{d\rho}{dx} \cdot \frac{dx}{dt}$$

Clearly, if the control device's differential reactivity worth, $\frac{d\rho}{dx}$, is large, then the speed at which the control device must be moved, $\frac{dx}{dt}$, can be minimized. So doing is desirable because it will reduce wear on the variable speed stepper motors. In contrast, if the differential worth is low, then the stepper motor must be operated at high frequency. Not only will this accelerate wear but it also risks the possibility of causing the motor to stall.

5.3.9 Hardware and Other non-Software Group

130-2 Testing of the MIT-SNL Period-Generated Minimum Time Control Laws on Sandia's ACRR was approved subject to the following:

- 1) The instrumentation that formed the ACRR's safety system would not also be used for the controller. The safety system was to be maintained as a separate, independent entity.
- 2) Signals from the digital computer would replace those from the existing eight-step programmable controller. The hardware and circuitry previously used for control of the ACRR would be retained.
- 3) An independent, hard-wired over-speed circuit would be installed to preclude withdrawal for the transient rod bank at frequencies in excess of 1100 Hz.
- 4) A "Loss of Computer Enable" circuit which would result in an insertion signal to the transient bank should there be a

malfunction of the computer or the associated hardware would be installed.

- 5) (This requirement is moved to groups of General Information, Power Multiplication, Fuel Temperature, Reactor Period, and Reactivity.)
- 6) Specification of a transient duration limit following which control would revert to manual.
- 7) A requirement to test both the over-speed trip and the "loss of enable" circuit prior to each set of runs and periodically during the runs.
- 8) (This requirement is moved Reactivity group.)
- 9) Control would never be transferred from manual to automatic with reactor already on a positive period. The purpose of this restriction was to limit the excess reactivity to that inserted by the transient rod bank.
- 10) Reactor operation would at all times be monitored by licensed personnel.

5.4 Within-group Analysis

The purpose of the Within-Group Analysis is to determine whether software requirements are consistent within each individual group. The ambiguity, inconsistency, or incompleteness identified by the analysis will be underlined.

5.4.1 General Information

130-2

- 5) The following restrictions would be incorporated in the software
 - Use of the standard, sufficient reactivity constraint in a supervisory role.
 - Use of the absolute, sufficient energy constraint in a supervisory role
- MIT-SNL Period-Generated Minimum Time Control Law should be used in the control software.

In this group, no problem is identified as long as the analyst understands such terminology as sufficient reactivity constraint and sufficient energy constraint.

5.4.2 Power Multiplication Group

- 4 Principal objective was to develop a robust control technique that would permit a neutronic power to be raised by five to seven orders of magnitude both in a few seconds and without significant overshoot;

130-2

- 5) (There will be an) absolute limit on the maximum neutronic power level.

178 ... the program's ultimate goal ... was to raise a reactor's
neutronic power by five-six orders of magnitude both on a
period of several tenths of a second and without overshoot ...

247 ... The power peaked ... , well within the permitted 5% band.

5.4.2.1 Analysis by Inspection

The following ambiguities were identified. Column one refers to the page number of the requirement in [Bern89] and column two refers to the corresponding ambiguity.

4 "Principal objective"

4 "a few seconds", "five to seven orders of magnitude", "without
significant overshoot";

178 "five-six orders of magnitude", "a period of several tenths of a
second" "without overshoot";

247 "the permitted 5% band".

The following inconsistencies were identified. Numbers in the parentheses refer to the page number in [Bern89].

"five to seven orders of magnitude"(4) vs. "five-six orders of
magnitude" (178);

"without significant overshoot" (4), "without overshoot" (178) vs.
"the permitted 5% band" (247).

Whenever ambiguity or inconsistency arises, it must be fed back to the customer immediately. Discussion with the customer helps clarify the requirements and remove ambiguities or inconsistencies.

In this case study, several major sources of ambiguity or inconsistency are identified. The first source is that different individuals wrote different parts of the report/system requirements. People with different backgrounds tend to have a different emphasis when addressing a problem and assume that others will have the same understanding, thus leaving out relevant information. Second, to meet the deadline of delivery, managers and engineers tend to be more interested in tangible products, e.g., hardware set up, lines of code developed daily, than intangible products such as requirements, design documents, users' manuals, etc. They usually work on the intangibles because they are required by contract, rather than they realize these documents are as important as the tangibles. The third source is that, sometimes, no sound set of requirements is developed in advance, or no good house-keeping record is developed and well maintained. This negligence results in writing final documentation based on a blurred recollection of the project in the end.

After resolving ambiguities and inconsistencies, the requirements in the Power Multiplication group are revised as follows.

The objective was to develop a robust control technique that would permit a neutronic power to be raised by up to six orders of magnitude in 5 seconds and without overshooting by 5% of nominal target power level.

5.4.2.2 Lessons Learned

When documenting software requirements, avoid using inherently vague, ambiguous words such as “a few”, “significant”. Use exact wording whenever possible.

5.4.3 Operational Mode Group

Page Requirements

9 The reactor operates in either a steady-state or a pulsed mode. For steady-state operation, the maximum

allowed power level is 2 MWt. For pulsed operation, there is no restriction of the power. Rather, there is a limit of 500 MJ total energy per pulse ...

11 The limiting condition for the ACRR is not a specific power level but rather the total energy produced during the transient.

127 The scram circuit will cause an automatic shutdown to prevent the energy release from exceeding 500 MJ.

266 It was used as a supervisory routine during all of the ACRR control experiments to ensure that the reactor's energy output would not exceed the limiting value of 350 MJ.

We start the within-group analysis with inspection. The underlined words indicate ambiguity or inconsistency identified by inspection. It is ambiguous whether “the transient” is the same as “pulse”, “pulsed mode”, or “pulsed operation”; whether “maximum allowed power level” is the same as “restriction of the power”, and “a limit of 500 MJ” is not consistent with “the limiting value of 350 MJ”. Discussions with the developer clarify that the limit is 500 MJ, “maximum allowed power level” is the same as “restriction of the power”, and “transient” is better rephrased as “pulse”. As a result, requirements on pages 127 and 266 are incorporated into that on page 9, and the requirements in the group are rephrased into two requirements.

Requirement # Requirements

- A. The reactor operates in either a steady-state⁹ or a pulsed mode. For steady-state mode, the limiting power level is 2 MWt. For pulsed mode, there is no limit of the maximum allowed power level. Rather,

⁹ Later consultation (May 1998) with the Dr. John Bernard, the developer, indicated that steady-state operation should be better phrased as low power operation.

there is a limit of 500 MJ total energy per pulse.

- B. The limiting condition for the reactor is not the power level but rather the total energy produced during the pulse.

Requirements A can be split into three requirements named 1, 2, and 3. This split does not affect the result of the analysis. This is because the requirements specified in A indicate satisfying the conjunction of the three, which can be explicitly expressed as three requirements. Requirement B is renamed requirements 4.

Requirement # Requirements

1. The reactor operates in either a steady-state or a pulsed mode.
2. For steady-state mode, the limiting power level is 2 MWt.
3. For pulsed mode, there is no limit of the maximum allowed power level. Rather, there is a limit of 500 MJ total energy per pulse.
4. The limiting condition for the reactor is not the power level but rather the total energy produced during the pulse.

The next step is to define logical variables in the requirements.

- PO* = the reactor operates in pulsed mode
SO = the reactor operates in steady-state mode
P = limiting condition is the power level
P2 = the (power) limit is 2 MWt
E = limiting condition is the total energy produced
E500 = the (energy) limit is 500 MJ per pulse

Once variables are defined above, the natural language requirements can be transformed into the equivalent propositional expressions. It should be noted that

- the reactor operates in either steady-state mode or pulsed mode, but *not both*. Therefore, the logical operation between the two variables should be exclusive OR, rather than disjunction;
- there are at least three interpretations of requirements 4 as shown below.

This example, as well as many others, indicates the inherent ambiguity of natural language and its inadequacy in documenting software requirements. After consultation with the customer, it turned out that interpretation (i) is the only correct interpretation of the requirement. In this specific situation, interpretations (ii) and (iii) will generate the same result in this case.

<i>Requirement #</i>	<i>Expression #</i>	<i>Propositional Expressions.</i>
1	1	$SO \vee_e PO$
2	2	$SO \Rightarrow P \wedge P2$
3	3	$PO \Rightarrow \neg P \wedge E \wedge E500$
4	4	(i) $PO \Rightarrow \neg P \wedge E$ (ii) $PO \wedge \neg P \wedge E$ (iii) $\neg P \wedge E$

The final step is to analyze the logical expressions for consistency. As a comparison, four approaches will be used to illustrate that the *Tree-Based Graphical Approach (TBGA)* is more user-friendly. Through this analysis, it is seen that, in the *Analytical Approach*, proofs are performed in a sequential fashion (one-dimensional) and one must be skillful enough to determine which axioms, theorems, or previously proved identities should be used during the process of reasoning. Therefore, it requires much more mathematical intuition. The TBGA Approach, on the other hand, is more mechanical and proofs are performed in a parallel fashion (two-dimensional), i.e., pushing proofs as far as possible until the basic definitions

are reached. In the graphical approach, it is preferable to start (and continue) the analysis with the expression that is the easiest to reason, e.g., the shortest expression, although the order in which expressions are selected does not affect the final result of the analysis at all (see Sections 4.5.5 and 5.4.3.5). The *Truth-Table Approach* is the most mechanical method, but the size of the table grows exponentially with the number of independent variables. The *DFM Approach* uses an automatic tool. However, since it is based on decision table, exponential time is needed to construct the decision tables.

5.4.3.1 The Analytical Approach

The Analytical Approach is discussed in [Ince88], [Hoar69], [Hoar71], [Lond75]. As indicated in Table 5.4-1, the proofs are expressed sequentially in a tabular form with three columns. The first column orders the expressions according to the sequence of the proof, the second column represents the result obtained from the expressions listed in the third column. Therefore, the expression numbers contained in the third column of each row must be strictly less than that in the first column.

Two cases were identified to be necessary in the analysis: a case with $SO = true$ and a case with $PO = true$. The result using the Analytical Approach for case 1 shows that the original set of requirements under steady-state operational mode is consistent (row 14 in Table 5.4-1 (A)) and is equivalent to Expression (5.4-1). The result for case 2 show that the original set of requirements under pulsed operational mode is consistent (row 18 in Table 5.4-1 (B)) and is equivalent to Expression (5.4-2).

$$SO \wedge \neg PO \wedge P \wedge P2 \quad (5.4-1)$$

$$\neg SO \wedge PO \wedge \neg P \wedge E \wedge E500 \quad (5.4-2)$$

The fact that the reactor will not be operated in both the steady-state and the pulsed modes simultaneously (i.e., explicitly expressed as $SO \vee PO$) is equivalently expressed by both expressions, i.e., $SO \wedge \neg PO$ in Expression (5.4-1) and $\neg SO \wedge PO$ in Expression (5.4-2).

5.4.3.2 The Truth-Table Approach

As defined in Section 5.4.3, six independent variables are defined within this group (i.e., SO , PO , P , $P2$, E , $E500$). Therefore, to exhaustively enumerate all permutations of the truth value assigned to these variables, the table contains 2^6 (=64) rows (see Table 5.4-2.).

In the truth table, only two rows, i.e., six six-tuples (0, 1, 0, 0, 1, 1), (0, 1, 0, 1, 1, 1), (1, 0, 1, 1, 0, 0), (1, 0, 1, 1, 0, 1), (1, 0, 1, 1, 1, 0) or (1, 0, 1, 1, 1, 1) for (SO , PO , P , $P2$, E , $E500$), satisfy the logical expression. Therefore, the expression is equivalent to the disjunction of the corresponding rows.

$$\begin{aligned}
 & (SO \vee_e PO) \wedge (SO \Rightarrow P \wedge P2) \wedge (PO \Rightarrow \neg P \wedge E \wedge E500) \wedge (PO \Rightarrow \neg P \wedge E) \\
 &= [(\neg SO \wedge PO \wedge \neg P \wedge \neg P2 \wedge E \wedge E500) \vee (\neg SO \wedge PO \wedge \neg P \wedge P2 \wedge E \wedge E500)] \\
 &\quad \vee [(SO \wedge \neg PO \wedge P \wedge P2 \wedge \neg E \wedge \neg E500) \vee (SO \wedge \neg PO \wedge P \wedge P2 \wedge \neg E \wedge E500)] \\
 &\quad \vee [(SO \wedge \neg PO \wedge P \wedge P2 \wedge E \wedge \neg E500) \vee (SO \wedge \neg PO \wedge P \wedge P2 \wedge E \wedge E500)] \\
 &= [(\neg SO \wedge PO \wedge \neg P \wedge E \wedge E500) \wedge (\neg P2 \vee P2)] \\
 &\quad \vee [(SO \wedge \neg PO \wedge P \wedge P2 \wedge \neg E) \wedge (\neg E500 \vee E500)] \\
 &\quad \vee [(SO \wedge \neg PO \wedge P \wedge P2 \wedge E) \wedge (\neg E500 \vee E500)] \\
 &= (\neg SO \wedge PO \wedge \neg P \wedge E \wedge E500) \vee (SO \wedge \neg PO \wedge P \wedge P2 \wedge \neg E) \\
 &\quad \vee (SO \wedge \neg PO \wedge P \wedge P2 \wedge E) \\
 &= (\neg SO \wedge PO \wedge \neg P \wedge E \wedge E500) \vee [(SO \wedge \neg PO \wedge P \wedge P2 \wedge (\neg E \vee E)] \\
 &= (\neg SO \wedge PO \wedge \neg P \wedge E \wedge E500) \vee (SO \wedge \neg PO \wedge P \wedge P2) \tag{5.4-3}
 \end{aligned}$$

Expression (5.4-3) is exactly the same as the disjunction of Expressions (5.4-1) and (5.4-2).

Table 5.4-1 Case Study: the Purely Analytical Approach (Operational Mode Group)(A) Case 1: $SO = true$

<i>Expression #</i>	<i>Manipulation/Derivation</i>	<i>Derived from</i>
1	$SO \vee_e PO$	requirement 1
2	$SO \Rightarrow P \wedge P2$	requirement 2
3	$PO \Rightarrow \neg P \wedge E \wedge E500$	requirement 3
4	$PO \Rightarrow \neg P \wedge E$	requirement 4
5	SO	case 1
6	$(SO \wedge \neg PO) \vee (\neg SO \wedge PO)$	(1)
7	$\neg PO$	(5, 6)
8	$\neg PO \vee (\neg P \wedge E \wedge E500)$	(3 or 7)
9	$\neg PO \vee (\neg P \wedge E)$	(4 or 7)
10	$\neg SO \vee (P \wedge P2)$	(2)
11	$P \wedge P2$	(5, 10)
12	P	(11)
13	$P2$	(11)
14	$SO \wedge \neg PO \wedge P \wedge P2$	(5, 7, 12, 13)

Table 5.4-1 Case Study: the Purely Analytical Approach (Operational Mode Group)

(B) Case 2: $PO = true$

<i>Expression #</i>	<i>Manipulation/Derivation</i>	<i>Derived from</i>
1	$SO \vee_e PO$	requirement 1
2	$SO \Rightarrow P \wedge P2$	requirement 2
3	$PO \Rightarrow \neg P \wedge E \wedge E500$	requirement 3
4	$PO \Rightarrow \neg P \wedge E$	requirement 4
5	PO	case 2
6	$(SO \wedge \neg PO) \vee (\neg SO \wedge PO)$	(1)
7	$\neg SO$	(5, 6)
8	$\neg SO \vee (P \wedge P2)$	(2 or 7)
9	$\neg PO \vee (\neg P \wedge E \wedge E500)$	(3)
10	$\neg P \wedge E \wedge E500$	(5, 9)
11	$\neg P$	(10)
12	E	(10)
13	$E500$	(10)
14	$\neg PO \vee (\neg P \wedge E)$	(4)
15	$\neg P \wedge E$	(5, 14)
16	$\neg P$	(15)
17	E	(15)
18	$\neg SO \wedge PO \wedge \neg P \wedge E \wedge E500$	(5, 7, 11, 12, 13, 16, 17)

Table 5.4-2 Case Study: Truth-Table Approach (Operational Mode Group)

<i>SO</i>	<i>PO</i>	<i>P</i>	<i>P2</i>	<i>E</i>	<i>E500</i>	X_1	X_2	X_3	X_4	$\bigwedge_{i=1}^4 X_i$
0	0	0	0	0	0	0	1	1	1	0
0	0	0	0	0	1	0	1	1	1	0
0	0	0	0	1	0	0	1	1	1	0
0	0	0	0	1	1	0	1	1	1	0
0	0	0	1	0	0	0	1	1	1	0
0	0	0	1	0	1	0	1	1	1	0
0	0	0	1	1	0	0	1	1	1	0
0	0	0	1	1	1	0	1	1	1	0
0	0	1	0	0	0	0	1	1	1	0
0	0	1	0	0	1	0	1	1	1	0
0	0	1	0	1	0	0	1	1	1	0
0	0	1	0	1	1	0	1	1	1	0
0	0	1	1	0	0	0	1	1	1	0
0	0	1	1	0	1	0	1	1	1	0
0	0	1	1	1	0	0	1	1	1	0
0	0	1	1	1	1	0	1	1	1	0
0	1	0	0	0	0	1	1	0	0	0
0	1	0	0	0	1	1	1	0	0	0
0	1	0	0	1	0	1	1	0	1	0
0	1	0	0	1	1	1	1	1	1	1
0	1	0	1	0	0	1	1	0	0	0
0	1	0	1	0	1	1	1	0	0	0
0	1	0	1	1	0	1	1	0	1	0
0	1	0	1	1	1	1	1	1	1	1
0	1	1	0	0	0	1	1	0	0	0
0	1	1	0	0	1	1	1	0	0	0
0	1	1	0	1	0	1	1	0	0	0
0	1	1	0	1	1	1	1	0	0	0
0	1	1	1	0	0	1	1	0	0	0
0	1	1	1	0	1	1	1	0	0	0
0	1	1	1	1	0	1	1	0	0	0
0	1	1	1	1	1	1	1	0	0	0

Table 5.4-2 Case Study: Truth-Table Approach (Operational Mode Group)
(continued)

SO	PO	P	$P2$	E	$E500$	X_1	X_2	X_3	X_4	$\bigwedge_{i=1}^4 X_i$
1	0	0	0	0	0	1	0	1	1	0
1	0	0	0	0	1	1	0	1	1	0
1	0	0	0	1	0	1	0	1	1	0
1	0	0	0	1	1	1	0	1	1	0
1	0	0	1	0	0	1	0	1	1	0
1	0	0	1	0	1	1	0	1	1	0
1	0	0	1	1	0	1	0	1	1	0
1	0	0	1	1	1	1	0	1	1	0
1	0	1	0	0	0	1	0	1	1	0
1	0	1	0	0	1	1	0	1	1	0
1	0	1	0	1	0	1	0	1	1	0
1	0	1	0	1	1	1	0	1	1	0
1	0	1	1	0	0	1	1	1	1	1
1	0	1	1	0	1	1	1	1	1	1
1	0	1	1	1	0	1	1	1	1	1
1	0	1	1	1	1	1	1	1	1	1
1	1	0	0	0	0	0	0	0	0	0
1	1	0	0	0	1	0	0	0	0	0
1	1	0	0	1	0	0	0	0	1	0
1	1	0	0	1	1	0	0	1	1	0
1	1	0	1	0	0	0	0	0	0	0
1	1	0	1	0	1	0	0	0	0	0
1	1	0	1	1	0	0	0	0	1	0
1	1	0	1	1	1	0	0	1	1	0
1	1	1	0	0	0	0	0	0	0	0
1	1	1	0	0	1	0	0	0	0	0
1	1	1	0	1	0	0	0	0	0	0
1	1	1	0	1	1	0	0	0	0	0
1	1	1	1	0	0	0	1	0	0	0
1	1	1	1	0	1	0	1	0	0	0
1	1	1	1	1	0	0	1	0	0	0
1	1	1	1	1	1	0	1	0	0	0
Note	$X_1 = (SO \vee_e PO)$ $X_2 = (SO \Rightarrow P \wedge P2)$					$X_3 = (PO \Rightarrow \neg P \wedge E \wedge E500)$ $X_4 = (PO \Rightarrow \neg P \wedge E)$				

5.4.3.3 The DFM Approach

The requirements of the above group are expressed formally as

$$(SO \vee_e PO) \wedge (SO \Rightarrow P \wedge P2) \wedge (PO \Rightarrow \neg P \wedge E \wedge E500) \wedge (PO \Rightarrow \neg P \wedge E) \quad (5.4-4)$$

The task of finding whether the requirements are consistent is equivalent to finding at least one set of truth value assignments for the variables in Expression (5.4-4) such that the expression is evaluated as *true*. In other words, if this expression is viewed as a top event, the task is to find all of its prime implicants using the DFM tool. Following the same steps as described in Section 4.5.4, we construct a DFM network model of the case study as in Figure 5.4-1. Since six logical variables are defined in Section 5.4.3, there are six corresponding independent variable nodes, as shown at the bottom in Figure 5.4-1, namely, P, P2, SO, PO, E, and E500. The tool requires intermediate nodes be constructed to express logical operations. In Figure 5.4-2, eleven intermediate nodes are needed. For example, R1 represents the logical expression $SO \vee_e PO$, R3 represents the logical expression $SO \Rightarrow PO \wedge P2$, R5 represents the logical expression $PO \Rightarrow \neg P \wedge E \wedge E500$, R6 represents the logical expression $(SO \vee PO) \wedge (SO \Rightarrow P \wedge P2) \wedge (PO \Rightarrow \neg P \wedge E \wedge E500)$ or requirements 1, 2 and 3, and R8 represents the logical expression $PO \Rightarrow \neg P \wedge E$ or requirement 4. These intermediate nodes are summarized in Table 5.4-3. Decision tables, as shown in Table 5.4-4, are used to model the logical expressions.

The result of running DFM model in Figure 5.4-1 is shown in Figure 5.4-2. The result indicates that there are two prime implicants for the top event, “SO=0 AND E500=1 AND P=0 AND E=1 AND PO=1” and “P2=1 AND SO=1 AND P=1 AND PO=0”. These two prime implicants are logically the same as Expression (5.4-3). As explained in Section 4.5.4.5, the set of requirements is consistent.

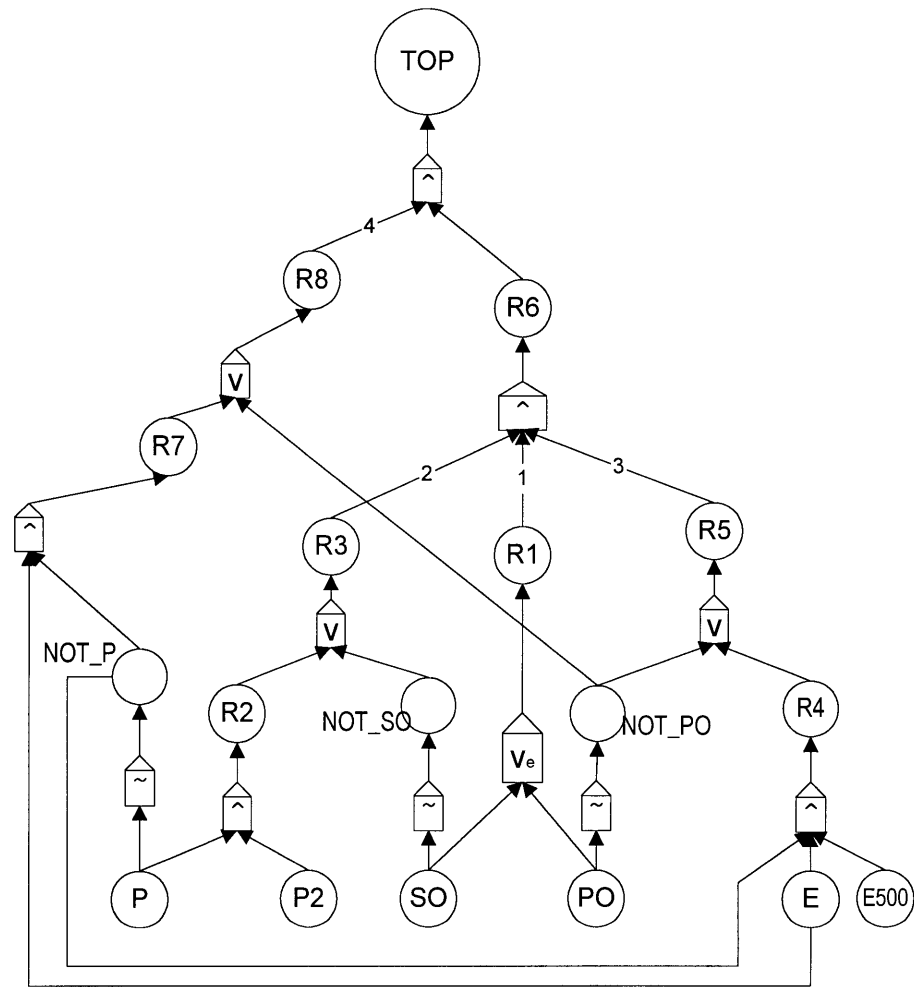


Figure 5.4-1 DFM Model for the Case Study

Table 5.4-3 Case Study: Intermediate Nodes and the Logical Expressions in Figure 5.4-1

<i>Intermediate Nodes</i>	<i>Corresponding Logical Expressions</i>
R1	$SO \vee_e PO$
R2	$P \wedge P2$
R3	$SO \Rightarrow P \wedge P2$ (or $\neg SO \vee P \wedge P2$)
R4	$\neg P \wedge E \wedge E500$
R5	$PO \Rightarrow \neg P \wedge E \wedge E500$ (or $\neg PO \vee (\neg P \wedge E \wedge E500)$)
R6	$(SO \vee PO) \wedge (SO \Rightarrow P \wedge P2) \wedge (PO \Rightarrow \neg P \wedge E \wedge E500)$
R7	$\neg P \wedge E$
R8	$PO \Rightarrow \neg P \wedge E$ (or $\neg PO \vee (\neg P \wedge E)$)
NOT_P	$\neg P$
NOT_SO	$\neg SO$
NOT_PO	$\neg PO$

Table 5.4-4 Case Study: Decision Table in the DFM Model (Operational Mode Group)

```

12
0.000000
2 1 1
      SO      NOT_SO
      0        1
      1        0

0.000000
4 2 1
      P        P2        R2
      0        0        0
      0        1        0
      1        0        0
      1        1        1

0.000000
4 2 1
      R2      NOT_SO      R3
      0        0        0
      0        1        1
      1        0        1
      1        1        1

0.000000
8 3 1
      E      E500      NOT_P      R4
      0        0        0        0
      0        0        1        0
      0        1        0        0
      0        1        1        0
      1        0        0        0
      1        0        1        0
      1        1        0        0
      1        1        1        1

0.000000
2 1 1
      P      NOT_P
      0        1
      1        0

0.000000
4 2 1
      NOT_P      E      R7
      0        0        0
      0        1        0
      1        0        0
      1        1        1

```

Table 5.4-4 Case Study: Decision Table in the DFM Model (Operational Mode Group)
(continued)

0.000000				
8 3 1				
	R3	R1	R5	R6
	0	0	0	0
	0	0	1	0
	0	1	0	0
	0	1	1	0
	1	0	0	0
	1	0	1	0
	1	1	0	0
	1	1	1	1
0.000000				
4 2 1				
	R6	R8	TOP	
	0	0	0	
	0	1	0	
	1	0	0	
	1	1	1	
0.000000				
2 1 1				
	PO	NOT_PO		
	0	1		
	1	0		
0.000000				
4 2 1				
	SO	PO	R1	
	0	0	0	
	0	1	1	
	1	0	1	
	1	1	0	
0.000000				
4 2 1				
	R7	NOT_PO	R8	
	0	0	0	
	0	1	1	
	1	0	1	
	1	1	1	
0.000000				
4 2 1				
	NOT_PO	R4	R5	
	0	0	0	
	0	1	1	
	1	0	1	
	1	1	1	

```
Prime - Notepad
File Edit Search Help

For the top event:

At time 0 , TOP = T (True)

There are 2 prime implicants

Prime Implicant #1
  At time 0 , S0 = 0 AND
  At time 0 , E500 = 1 AND
  At time 0 , P = 0 AND
  At time 0 , E = 1 AND
  At time 0 , P0 = 1

Prime Implicant #2
  At time 0 , P2 = 1 AND
  At time 0 , S0 = 1 AND
  At time 0 , P = 1 AND
  At time 0 , P0 = 0 |
```

Figure 5.4-2 DFM Results for the Case Study (Screen Dump)

Originally, DFM was developed to analyze problems involving multi-valued logic and temporal features. A new application of DFM to software requirements analysis was developed. Results from the four test cases in Chapter 4 and the real case study demonstrate that DFM can be used in software requirements analysis. This application is a part of the development of the proposed user-friendly methodology in software requirements analysis. Minimal training is needed to create truth tables (or decision tables). DFM can be used as an independent automatic tool in the within-group analysis as well as a double check of the Tree-Based Graphical Approach (TBGA).

5.4.3.4 The Tree-Based Graphical Approach

The detailed analyses are shown in Figure 5.4-3 (A) and (B) where 1, 2, 3 and 4 represent Expressions 1, 2, 3, and 4 respectively. The numbers on the edges of the tree indicate the order of operations.

At the top level, it is required that the conjunction of 1, 2, 3 and 4 be *true*, which is equivalent to 1 being *true*, 2 being *true*, 3 being *true* and 4 being *true* simultaneously. According to the rules, we start with the simplest one—1, in this example. Two cases need to be discussed separately.

Case 1: $SO = \text{true}$ Since $SO = \text{true}$, $\neg SO$ is false and $\neg SO \wedge PO$ is false. As a result, they are pruned from the tree (shaded with lines in Figure 5.4-3 (A)). Since Expression 1 is required to be true, $\neg PO$ must be true. The process is continued by selecting Expression 2 next since it contains variable SO which is contained and is true in Expression 1. Expand this branch and prune leaf $\neg SO$ since SO is true in Expression 1. But this branch is required to be true, P and $P2$ must be true (otherwise, the entire branch is false). For the same reason, Expression 4 is the next to be selected and expanded. $\neg P$ is pruned since it contradicts true leaf P in Expression 2. As a result, $\neg P \wedge E$ is pruned. Note that, $\neg PO$ is pruned (shaded with dots) not because it contradicts any known true branches or leaves, but $\neg PO$ is true in

Expression 1. Therefore, the entire branch of Expression 4 can be pruned. Similarly, Expression 3 is expanded and entirely pruned. Finally, take the conjunction of the leaves left in the tree. The conjunction of 1, 2, 3 and 4 is equivalent to the conjunction of the leaves or

$$1 \wedge 2 \wedge 3 \wedge 4 = SO \wedge \neg PO \wedge P \wedge P2 \quad (5.4-5)$$

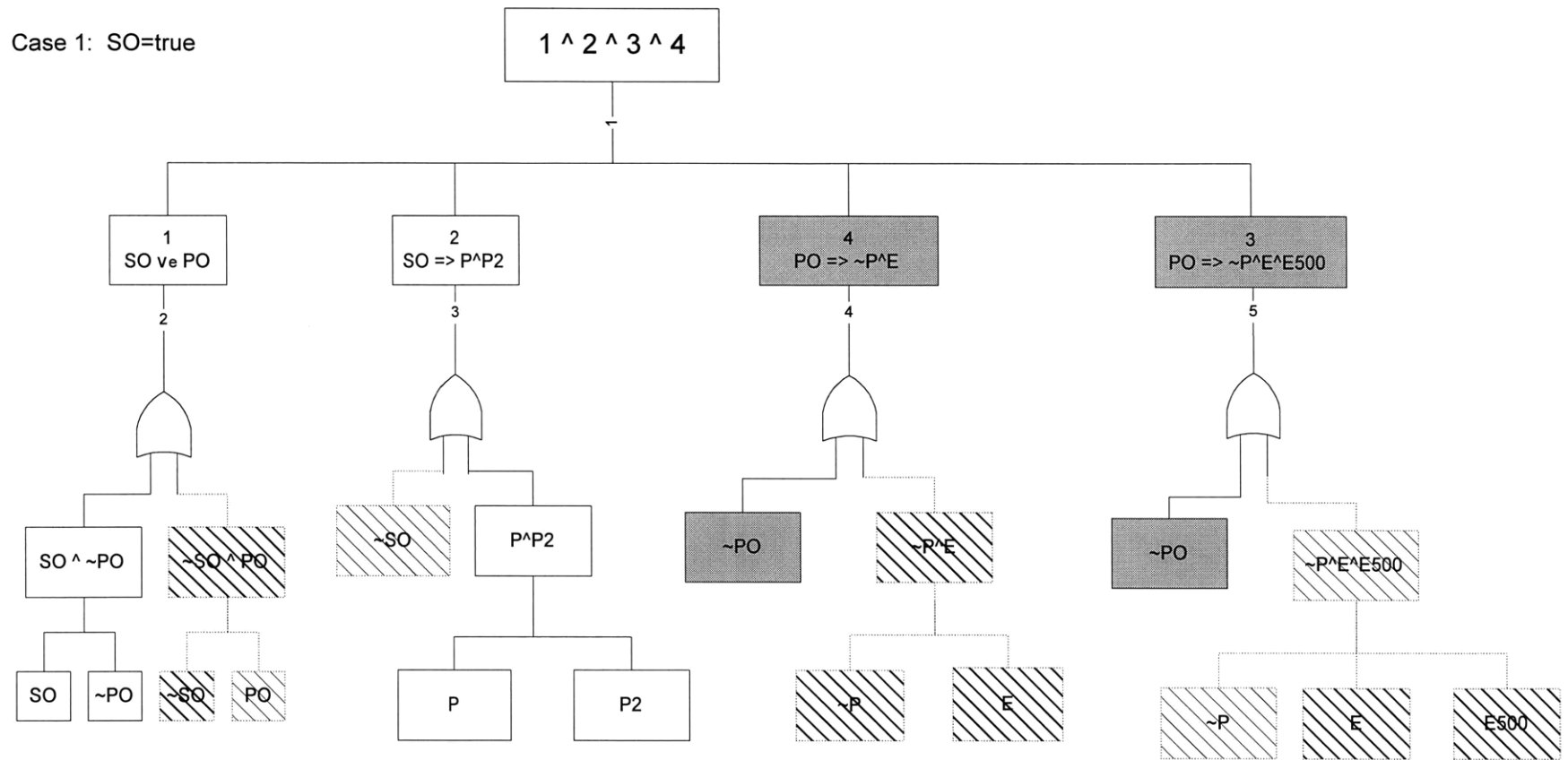
Expression (5.4-5) is the same as the second prime implicant in Section 5.4.3.3.

Case 2: $PO = \text{true}$ Since $PO = \text{true}$, $\neg PO$ is false and $SO \wedge \neg PO$ is false. Therefore, they are pruned from the tree (shaded with lines in Figure 5.4-3 (B)). Since Expression 1 is required to be true, $\neg SO$ must be true. The process is continued by selecting Expression 4 next since it contains variable PO which is contained and is true in Expression 1. Expand this branch and prune leaf $\neg PO$ since PO is true in Expression 1. But this branch is required to be true, $\neg P$ and E must be true. For the same reason, Expression 2 is selected and expanded next. P is pruned since it contradicts true leaf $\neg P$ in Expression 4. As a result, $P \wedge P2$ is pruned. Note that, $\neg SO$ is pruned (shaded with dots) not because it contradicts any known true branches or leaves, but $\neg SO$ is true in Expression 1. Therefore, the entire branch of Expression 2 is pruned. Similarly, Expression 3 is expanded and pruned. Finally, take the conjunction of the leaves left in the tree. The conjunction of 1, 2, 3 and 4 is equivalent to the conjunction of the leaves or

$$1 \wedge 2 \wedge 3 \wedge 4 = \neg SO \wedge PO \wedge \neg P \wedge E \wedge E500 \quad (5.4-6)$$

Expression (5.4-6) is the same as the first prime implicant in Section 5.4.3.3.

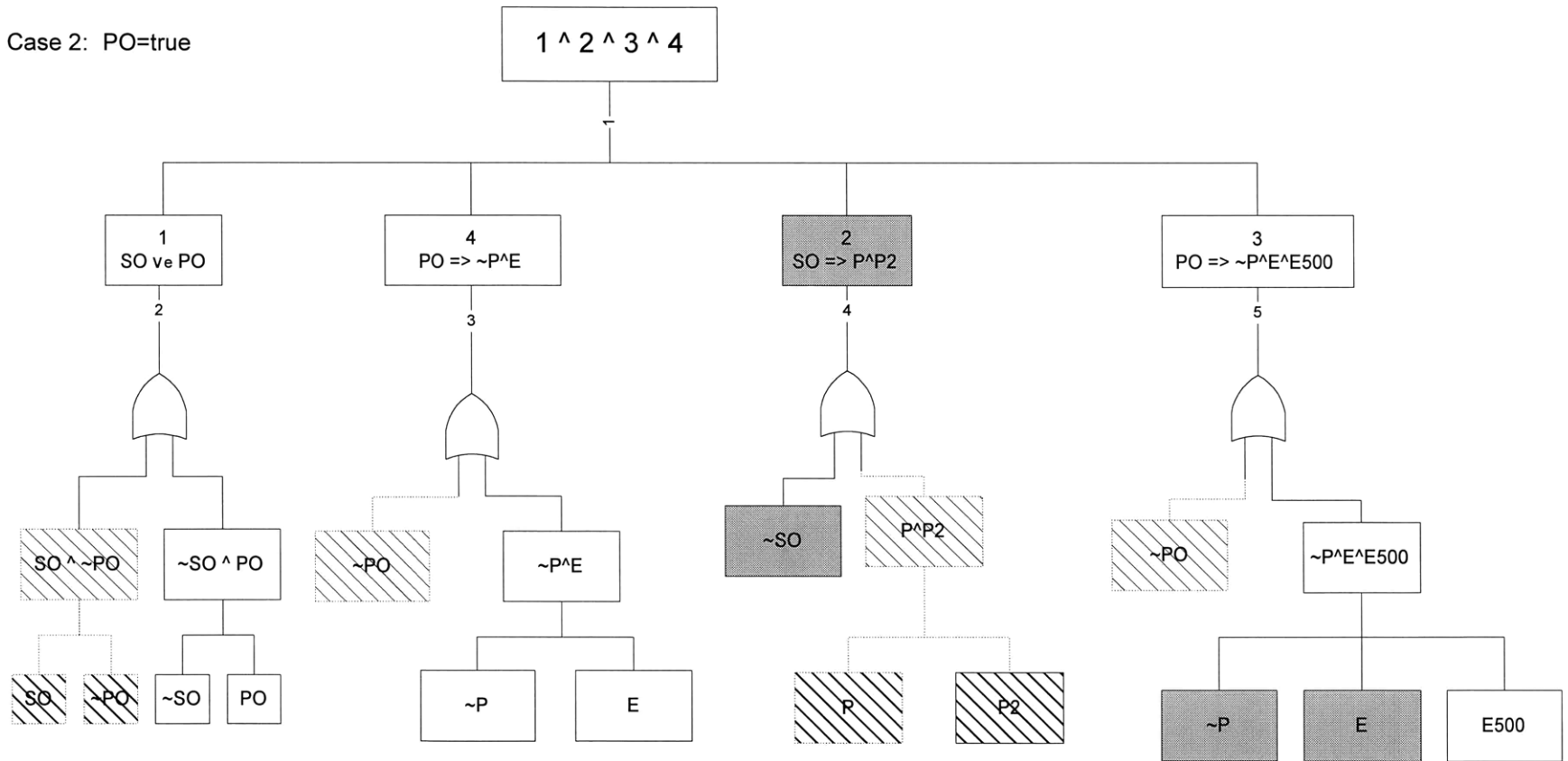
As expected, the graphical approach generates the same result as the other three approaches. Translating the above expression back into English, we have a more precise, unambiguous, and consistent requirement, since irrelevant information has been removed.



(A) $SO = \text{true}$

Figure 5.4-3 Within-Group Analysis Using the Tree-based Graphical Approach

Case 2: PO=true



(B) $PO = true$

Figure 5.4-3 Within-Group Analysis Using the Tree-based Graphical Approach

The reactor operates in steady-state mode, rather than pulsed mode, and the limiting power level is 2 MW_t; or the reactor operates in pulsed mode, rather than steady state mode, and there will be a limit of 500 MJ in total energy generated per pulse rather than power level¹⁰.

Note that the shaded boxes in Figure 5.4-3 represent leaves pruned during the analysis and should be removed from the tree. They are left in this tree simply for the purpose of illustrating the reasoning process. As a matter of fact, all parts of the tree between the line-shaded boxes and the next logic *OR* gate above the boxes (i.e., towards the root) should be pruned; all parts of the tree between the dot-shaded boxes and the next logic *AND* gate above the boxes (i.e., towards the root) should be pruned. As experience is gathered, many intermediate steps in the tree can be bypassed to simplify the process.

5.4.3.5 Order Analysis in the Tree-Based Graphical Approach

Based on formal logic, the within-group requirements analysis using the graphical approach is a very mechanical process with well defined rules. Starting with the simplest sub-expression, proceeding next with the sub-expression that contains the same logical variables (or literals) as in the previously worked sub-expressions, continuing the tree downward towards the leaves, and pruning any branch that contradicts known true leaves (or branches) identified earlier in the process or pruning any branch that is the same as a true leaf (or branch) identified earlier. In practice, it may not be always straightforward as to which sub-expression is the simplest. In other words, the simplest sub-expression may not be unique, e.g., two or more sub-expressions might be the simplest ones simultaneously. If this situation occurs, one can arbitrarily pick one of the sub-expressions to start (and/or continue) the analysis. The random selection of expressions will not affect the results of the analysis. In other words, the order on which the reasoning is based does not matter at all. To show this, we should revisit the above example by exhaustively analyzing all twenty-four *permutations*

¹⁰In this case study, only pulsed mode was used in the experiment.

(4!=24) of in the order of selecting expressions in Case 2. The following discussions will be focused on cases in which $PO = true$ and Expression 1 is the first to be selected, except for Figure 5.4-9 in which Expression 2 is selected first (Figures 5.4-4 to 5.4-9 as well as Figure 5.4-3 (B)).

The difference between Figures 5.4-4 and 5.4-3 (B) is the order in which sub-expressions $SO \Rightarrow P \wedge P2$ and $PO \Rightarrow \neg P \wedge E \wedge E500$ are evaluated. Figure 5.4-3 strictly abides by our pre-defined selection rules. The difference between Figures 5.4-5 and 5.4-6 is the order in which the sub-expressions $PO \Rightarrow \neg P \wedge E \wedge E500$ and $PO \Rightarrow \neg P \wedge E$ are expanded. The difference between Figures 5.4-7 and 5.4-8 is the order in which sub-expressions $PO \Rightarrow \neg P \wedge E$ and $SO \Rightarrow P \wedge P2$ are expanded. Figure 5.4-9 represents a scenario in which Expression 2 is selected first. $P \wedge P2$ in Expression 2 is not pruned until Expression 4 is expanded and pruned. $\neg SO$ in Expression 2 is not pruned until Expressions 3 and 1 are expanded and pruned.

Not surprisingly, it turns out that all the cases end up with the same result.

From this order analysis, it is concluded that the order in which the sub-expressions/branches are expanded does not affect the final result of the analysis. In other words, the analysis is order-independent. Therefore, in theory, starting with any sub-expression will guarantee the same final result. However, in practice, a good sub-expression selection order will simplify the tree in the process of the analysis. For large expressions, this could be a tremendous advantage.

5.4.3.6 Lessons Learned

When describing or documenting software requirements, one should use the same phrases and notation to represent the same events or requirements. Otherwise, confusion, ambiguity, or inconsistency will arise.

Case 2: $PO = \text{true}$

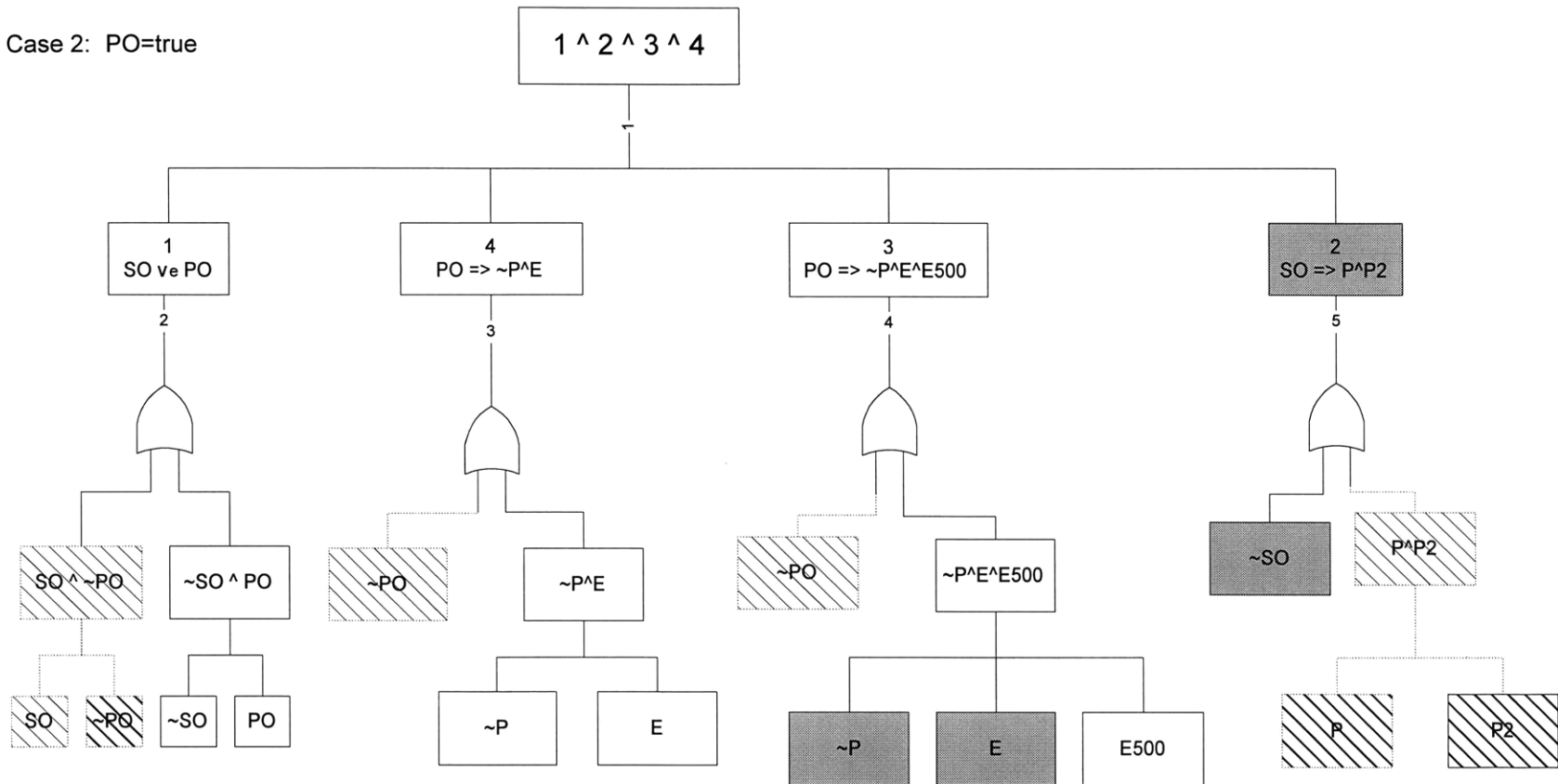


Figure 5.4-4 Order Analysis for the Within-Group Analysis ($PO = \text{true}$, 1/6)

Case 2: $PO = \text{true}$

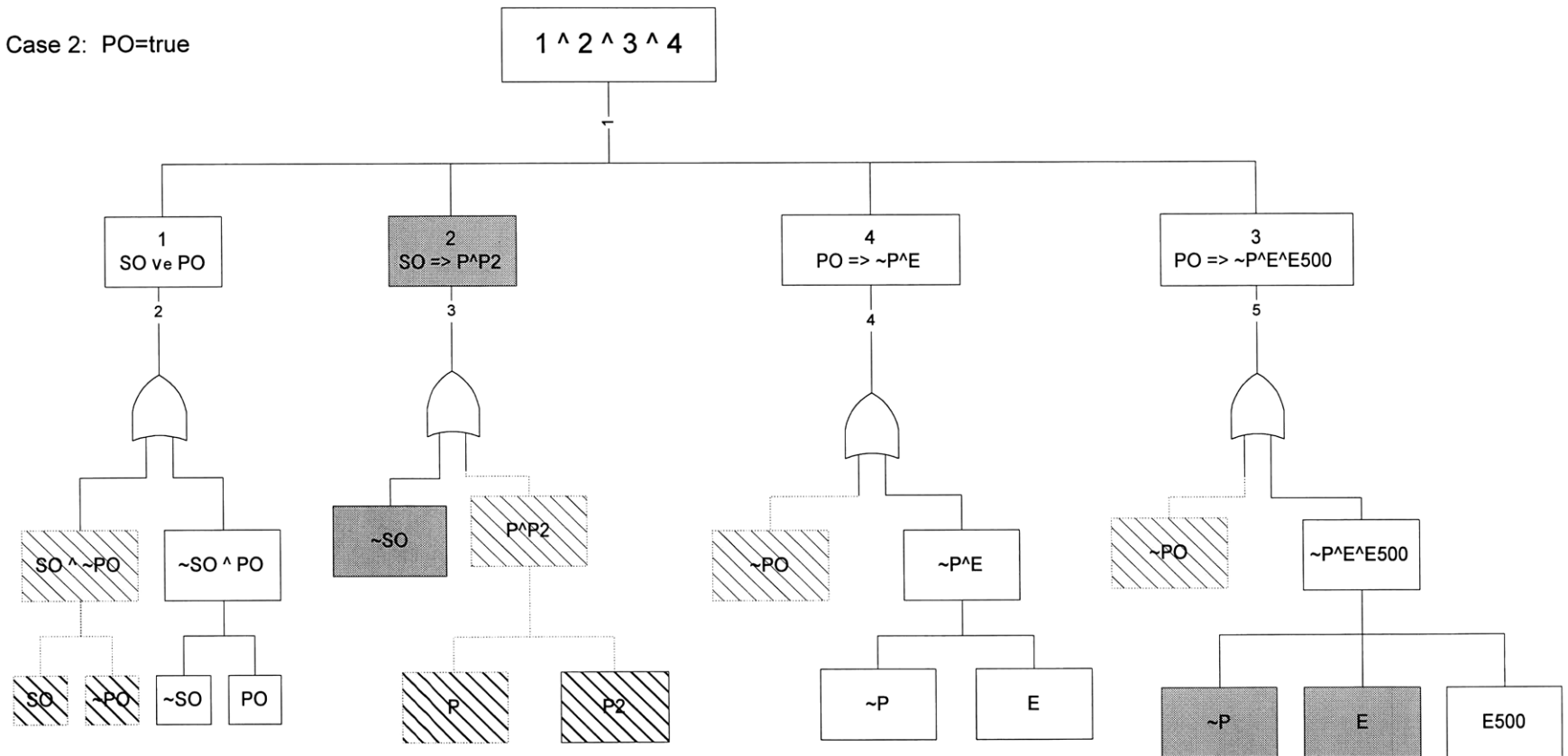


Figure 5.4-5 Order Analysis for the Within-Group Analysis ($PO = \text{true}, 2/6$)

Case 2: PO=true

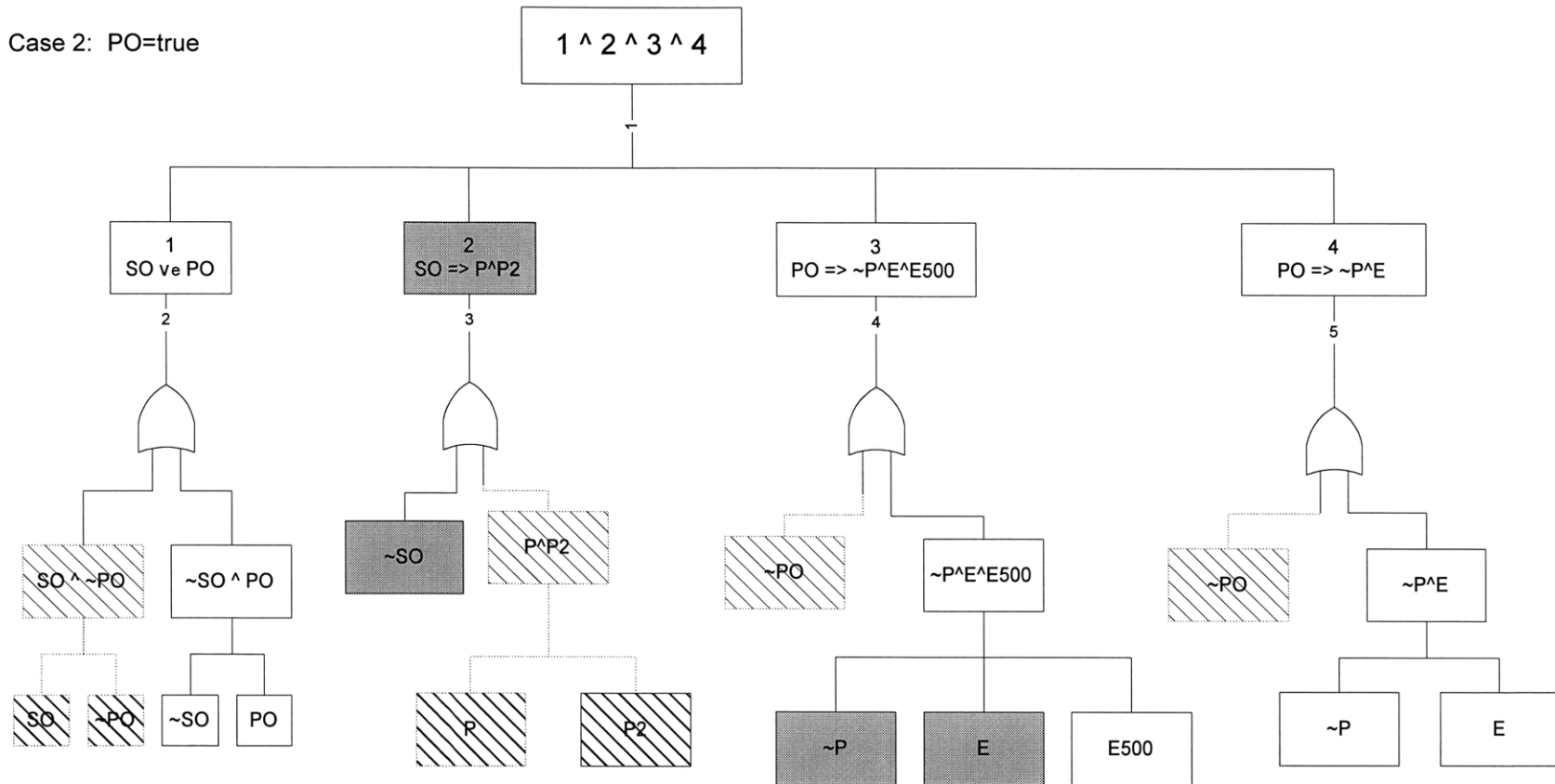


Figure 5.4-6 Order Analysis for the Within-Group Analysis ($PO = true, 3/6$)

Case 2: PO=true

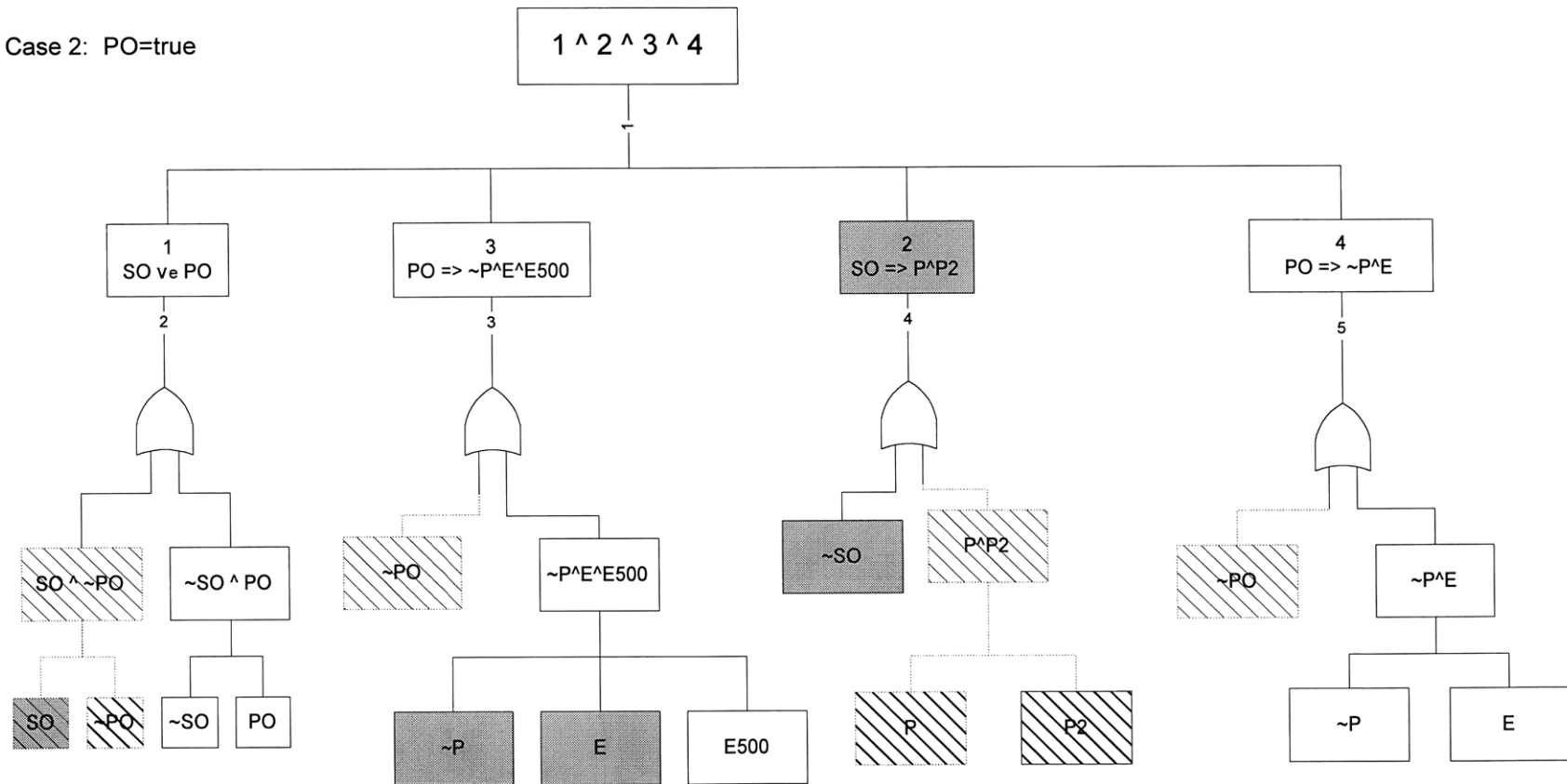


Figure 5.4-7 Order Analysis for the Within-Group Analysis ($PO = true$, 4/6)

Case 2: PO=true

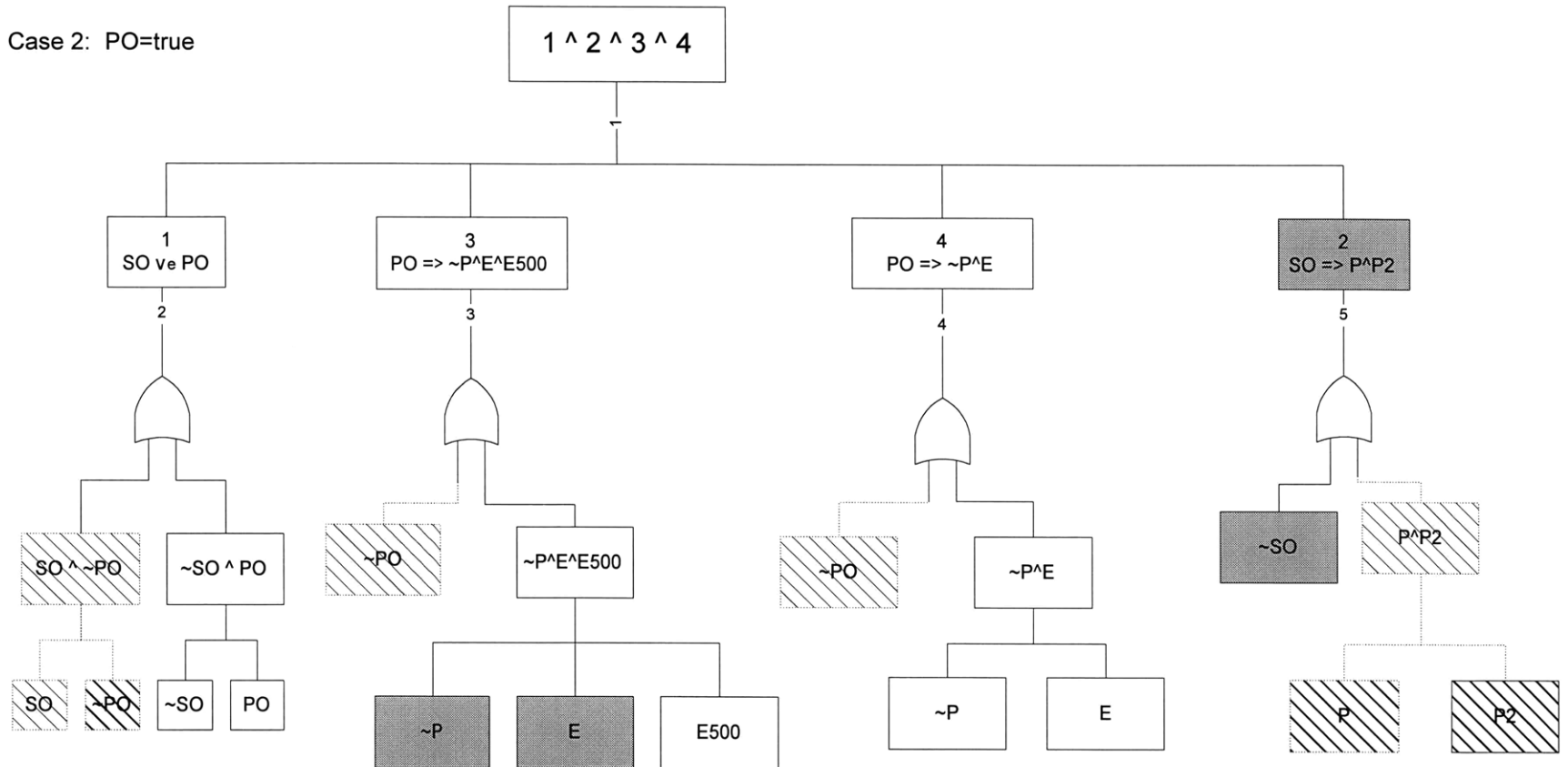


Figure 5.4-8 Order Analysis for the Within-Group Analysis ($PO = true$, 5/6)

Case 2: $PO = \text{true}$

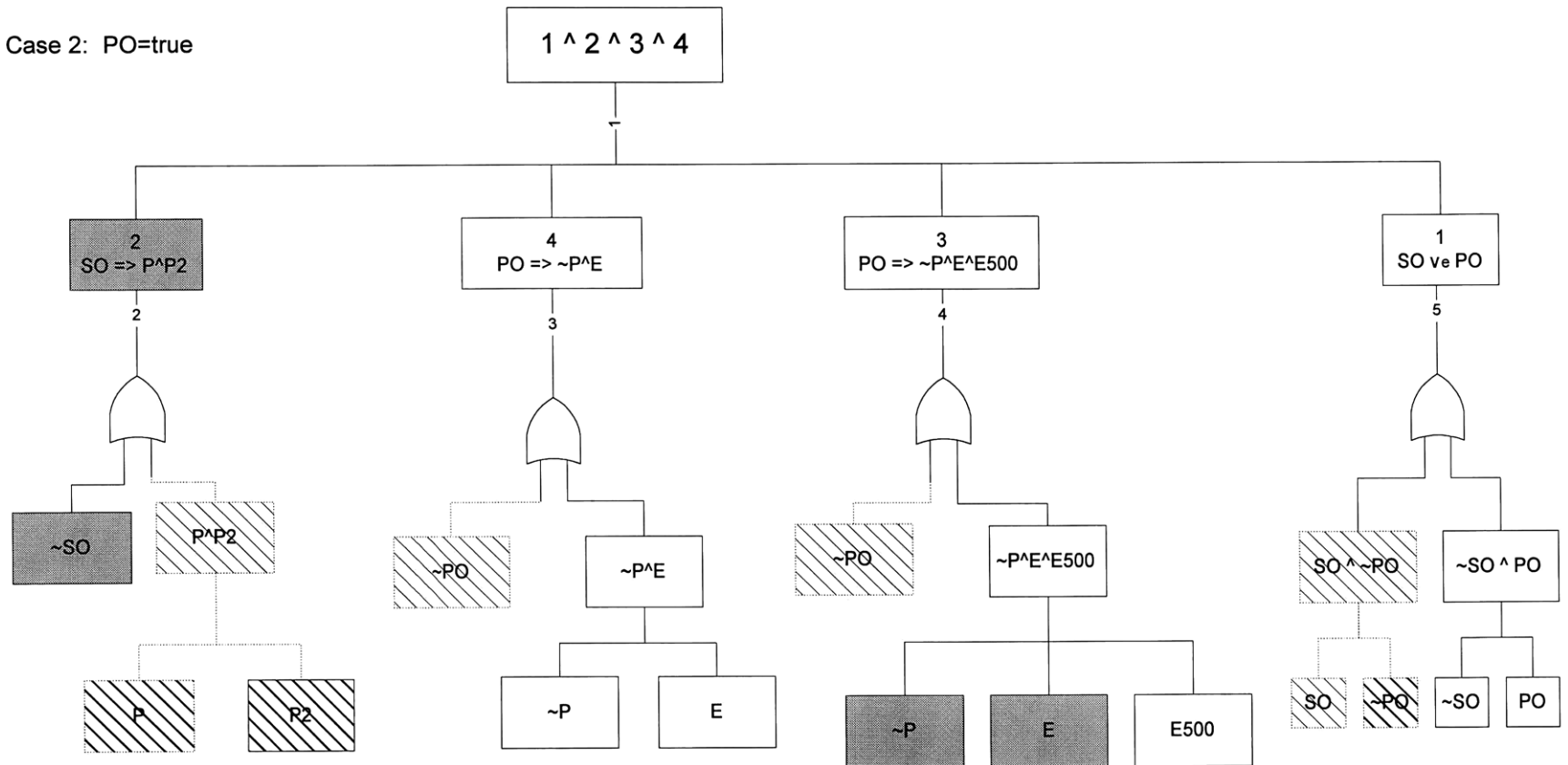


Figure 5.4-9 Order Analysis for the Within-Group Analysis ($PO = \text{true}$, 6/6)

5.4.4 Fuel Temperature Group

9 For pulsed operation, there is a limit ... of 1800°C on the fuel temperature.

127 The scram circuit will cause an automatic shutdown to prevent ... the fuel temperature from rising above 1800°C.

130-2

5) (There will be an) absolute limit on ... fuel temperature.

5.4.4.1 Analysis by Inspection

The following incompleteness or ambiguity was identified. It is not stated whether there is a limit on fuel temperature for steady state operation. After communication with the developer, this requirement was revised as follows:

For both steady state and transient operation, there is a limit of 1800°C on the fuel temperature.

5.4.4.2 Lessons Learned

When documenting software requirements, one should always specify the behaviors of the software under all situations, i.e., the requirements should be complete. Researches indicate that the majority of errors in requirements are omissions or incompleteness issues.

5.4.5 Reactor Period Group

130-2

5) (There will be an) absolute limit on ... the minimum observed reactor period.

179 ... there was a limit of 0.20 second on the minimum allowed period ... applied to the MIT-SNL laws.

- To satisfy Technical Specifications of the ACRR, the reactor period can not fall below 0.3 second

The following ambiguity or inconsistency was identified by inspection: “a limit of 0.20 second on the minimum allowed period” (179) vs. “the reactor period can not fall below 0.3 second”.

After communication with the developer, the ambiguity/inconsistency was resolved, resulting in requirement below:

To satisfy Technical Specifications of the ACRR, the reactor period can not fall below 0.3 second.

5.4.6 Reactivity Group

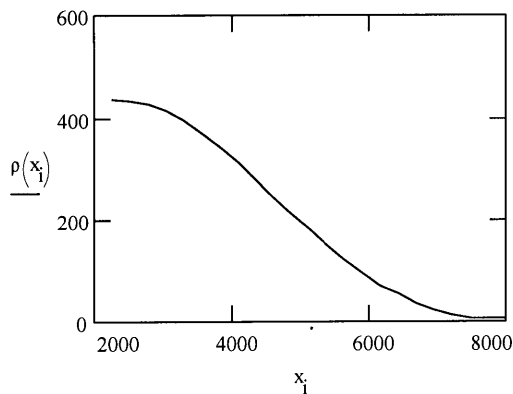
126 The transient rods are normally operated as a bank (gang) and, as mentioned , can be moved either pneumatically or via stepper motors. Relative to the first method, nitrogen gas is used to pressurize each rod's piston chamber and thereby eject the three rods in times as short as 55 milliseconds. When operated in this manner, the maximum allowed reactivity insertion is 3β ...

(127) The alternative means of moving the transient rods is the "Transient Rod Withdrawal" or TRW method. Using this approach, the transient rods are raised by fork-shaped devices which are each connected to variable speed stepper motors via an electromagnet and a rack and pinion gear... If moved from the "full-in" to "full-out" position, the total reactivity associated with the transient rods is 4.25β .

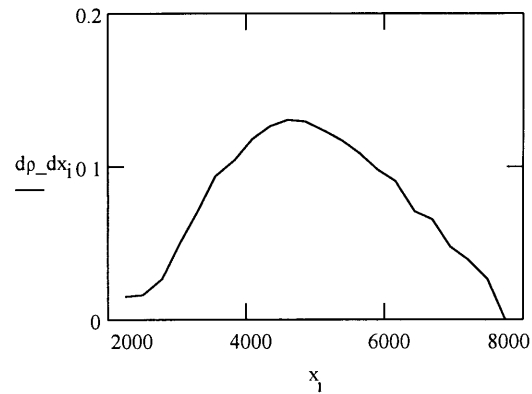
130-2

- 5) (There will be an) absolute limit on the net reactivity
- 8) The pneumatic mode of transient rod band withdrawal would be disabled and officially tagged out so as to preclude its inadvertent use.

236 (Transient Rod Worth, in cents)



(A) Integral Rod Worth



(B) Differential Rod worth

5.4.6.1 Analysis by Inspection

Through inspection, one inconsistency was found. On page 127 of [Bern89] it is claimed that reactivity associated with rod withdrawal “from the full-in to full-out position” is 4.25β , while the reactivity is 4.40β , or 440 cents, according to Figure 11.5.1-1 on page 236 of the report. Discussion with the developer indicates that 4.40β is the correct value.

5.4.6.2 Analysis by Tree-Based Graphical Approach

Let

- Bank = Transient rods are normally operated as a bank (gang)
 - Pneu = the transient rods moved pneumatically
 - Stepper = the transient rods moved via stepper motor
 - PneuCons = nitrogen gas is used to pressurize each rod’s piston chamber and thereby eject the three rods in times as short as 55 milliseconds.
- When operated in this manner, the maximum allowed reactivity

insertion is 3β ...

StepperCons = the transient rods are raised by fork-shaped devices which are each
ns connected to variable speed stepper motors via an electromagnet
and a rack and pinion gear. The maximum speed of withdrawing
the transient rods without any chance of magnet separation is 750
pulses per second. ... If moved from the “full-in” to “full-out”
position, the total reactivity associated with the transient rods is
 4.40β .

The requirements can be translated into the following propositional expressions.

<i>Page</i>	<i>Expression #</i>	<i>Propositional Expressions</i>
126-7	1	$Bank \wedge (Pneu \vee_e Stepper) \wedge (Pneu \Rightarrow PneuCons) \wedge (Stepper \Rightarrow StepperCons)$
130-2	2	$\neg Pneu$

Consistency requires the conjunction of expressions 1 and 2 be *true*, that is

$$Bank \wedge (Pneu \vee_e Stepper) \wedge (Pneu \Rightarrow PneuCons) \wedge (Stepper \Rightarrow StepperCons) \wedge \neg Pneu = true \quad (5.4-7)$$

Expression (5.4-7) will be analyzed using the Tree-Based Graphical Approach (TBGA) as shown in Figure 5.4-10.

Consistency requires the conjunction be *true*, which is equivalent to requiring all the terms in the conjunction be *true*, i.e., expressions at level 2 in the tree be *true*. We resolve the graph in further detail, pruning any branch that contradicts any known *true* branch or any branch that leads to a tautology conditioned on any *true* branch (e.g., a branch that leads to a *true* input to an OR gate). Using the graphical approach, the conjunction is equivalent to

$$\begin{aligned} & Bank \wedge (Pneu \vee_e Stepper) \wedge (Pneu \Rightarrow PneuCons) \wedge (Stepper \Rightarrow StepperCons) \wedge \neg Pneu \\ & = Bank \wedge \neg Pneu \wedge Stepper \wedge StepperCons \end{aligned}$$

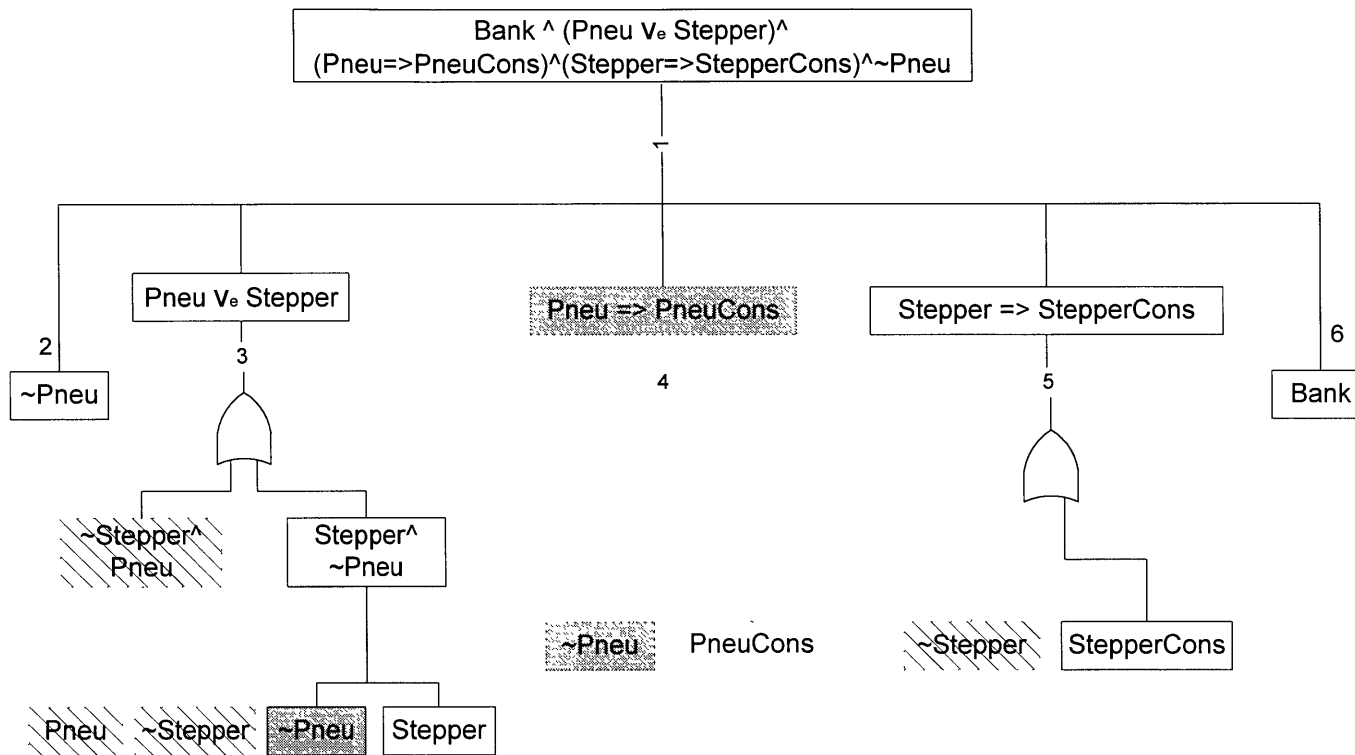


Figure 5.4-10 Within-Group Analysis Using the Tree-Based Graphical Approach (Reactivity Group)

Translating the expression back into plain English requirements, we have

The transient rods are normally operated as a bank (gang) and are moved via stepper motors, rather than pneumatically. The transient rods are raised by fork-shaped devices which are each connected to variable speed stepper motors via an electromagnet and a rack and pinion gear. If moved from the "full-in" to "full-out" position, the total reactivity associated with the transient rods is 4.40β . The rod worth is shown in Figure 5.2-1.

5.4.6.3 Lessons Learned

When documenting requirements, one should bear in mind that no part of the requirements should contradict other parts.

5.4.7 Motor Frequency Group

- (127) The maximum speed of withdrawing the transient rods without any chance of magnet separation is 750 pulses per second.
- 130 Relative to the actual conduct of experiments involving the MIT-SNL minimum time laws, an upper limit on the allowable frequency of 1100 Hz was selected. This figure was chosen because its is the value listed as acceptable for the TRW mode of operation in the ACRR's Safety Analysis Report and because the range of frequencies needed for the testing of the MIT-SNL minimum time laws was approximately 400-600 Hz.
- 140 (ACRR) The design specifications imposed on the interface controller board were that it be possible to move the stepper motors over the frequency range of -1000 Hz to +1000 Hz, that

the deviation of the output frequency from the required frequency not exceed 5% and that the over frequency trip be set at 1100 Hz. (Note: There was a deadband over the range of ± 15.26 Hz ...)

141 ... However, despite its having been designed to accommodate demanded frequencies of as much as 1000 Hz, the MIT-SNL experiments were restricted to 400 Hz.

5.4.7.1 Analysis by Inspection

By inspecting the requirements in this group, the following ambiguities/inconsistency were identified. Numbers in the parentheses represent the page numbers in the original report.

Ambiguities:

- It is not clear if “pulses per second” (127) is the synonym of “Hz” (130);
- It is not clear whether the maximum speed of the transient rods without any chance of magnet separation, “750 pulses per second”, is applicable to rod insertion as well as rod withdrawal (127);
- It is not clear what negative frequency means, e.g., “frequency range of -1000 Hz to +1000 Hz” (140);
- It is not clear what the “deadband over the range of ± 15.26 Hz” (140) is, e.g., relative deadband around a certain nominal frequency or absolute deadband centered at zero.

It turns out after discussion with the developer, that, negative frequency such as “-1000Hz” refers to the fact that the motor rotates in the opposite direction. A “pulse per second” is identical to Hz, and the deadband is a band in the absolute (not relative) sense, within which the motor doesn’t respond to signals.

Inconsistency:

- “testing of the MIT-SNL minimum time laws was approximately 400-600 Hz” (130). vs. “the MIT-SNL experiments were restricted to 400 Hz” (141).

Again, after discussion with the developer, it turns out that the MIT-SNL experiments were restricted to 400 Hz. So the requirement can be revised as follows.

The maximum frequency of moving the transient rods is ± 400 pulses per second and there is a deadband over the range of $[-15.26, 15.26]$ pulses per second within which the motor doesn't respond to signals. Negative frequency represents that motor rotates in the opposite direction.

5.4.8 Rod Velocity Group

(127) Transient rod location on the ACRR is given in "position units" with "full-in" being 2031 units and "full-out" being 7464 units. Every pulse delivered to the stepper motors moves the rods 2.661 units.

234 The objective of the MIT-SNL Laws is to determine the rate of change of reactivity (ρ_c) that will cause a reactor's neutronic power to vary on a specified period. The decision of these laws are most readily implemented if the differential reactivity worth ($\frac{d\rho}{dx}$) of the associated control device is at a maximum.

$$\dot{\rho}_c = \frac{d\rho}{dx} \cdot \frac{dx}{dt} = \frac{d\rho}{dx} \cdot v(t)$$

Clearly, if the control device's differential reactivity worth is large, then the speed at which the control device must be

moved, can be minimized. So doing is desirable because it will reduce wear on the variable speed stepper motors. In contrast, if the differential worth is low, then the stepper motor must be operated at high frequency. Not only will this accelerate wear but it also risks the possibility of causing the motor to stall.

Ambiguity:

- It is not specified whether the rod positions (in *units*) between the “full-in” (2031 units) and “full-out” (7464 units) positions can be linearly interpolated to obtain its actual position.

After discussion with the developer, the requirement on Rod Velocity as follows.

Transient rod location on the ACRR is given in "position units" with "full-in" being 2031 units and "full-out" being 7464 units. Rod positions in between can be interpolated to get its actual position. Every pulse (Hz) delivered to the stepper motor moves the rod 2.661 units.

5.4.9 Hardware and Other non-Software Group

130-2 Testing of the MIT-SNL Period-Generated Minimum Time Control Laws on Sandia's ACRR was approved subject to the following:

- 1) The instrumentation that formed the ACRR's safety system would not also be used for the controller. The safety system was to be maintained as a separate, independent entity.
- 2) Signals from the digital computer would replace those from the existing eight-step programmable controller. The hardware and circuitry previously used for control of the ACRR would be retained.

- 3) An independent, hard-wired over-speed circuit would be installed to preclude withdrawal for the transient rod bank at frequencies in excess of 1100 Hz.
- 4) A "Loss of Computer Enable" circuit which would result in an insertion signal to the transient bank should there be a malfunction of the computer or the associated hardware would be installed.
- 5) (This requirement is moved to groups of General Information, Power Multiplication, Fuel Temperature, Reactor Period, and Reactivity.)
- 6) Specification of a transient duration limit following which control would revert to manual.
- 7) A requirement to test both the over-speed trip and the "loss of enable" circuit prior to each set of runs and periodically during the runs.
- 8) (This requirement is moved Reactivity group.)
- 9) Control would never be transferred from manual to automatic with reactor already on a positive period. The purpose of this restriction was to limit the excess reactivity to that inserted by the transient rod bank.
- 10) Reactor operation would at all times be monitored by licensed personnel.

Since the interest of the analysis lies in requirements analysis of software, the analysis for non-software requirements is not presented here, assuming that this part of the analysis has been performed by other people.

5.4.10 Revised Requirements after Within-Group Analysis

After the analysis, the software related system requirements were revised as follows.

General Information

130-2

- 5) The following restrictions would be incorporated in the software
- Use of the standard, sufficient reactivity constraint in a supervisory role.
 - Use of the absolute, sufficient energy constraint in a supervisory role
- MIT-SNL Period-Generated Minimum Time Control Law should be used in the control software.

Power Multiplication

The objective was to develop a robust control technique that would permit a neutronic power to be raised by up to six orders of magnitude in 5 seconds and without overshooting by 5% of nominal target power level.

Operational Mode

The reactor will operate in transient mode, rather than the steady state mode, and there is a limit of 500 MJ in total energy rather than power level generated per transient.

Fuel Temperature

For both steady state and transient operation, there is a limit of 1800°C on the fuel temperature.

Reactor Period

To satisfy Technical Specifications of the ACRR, the reactor period can not fall below 0.3 second.

Reactivity

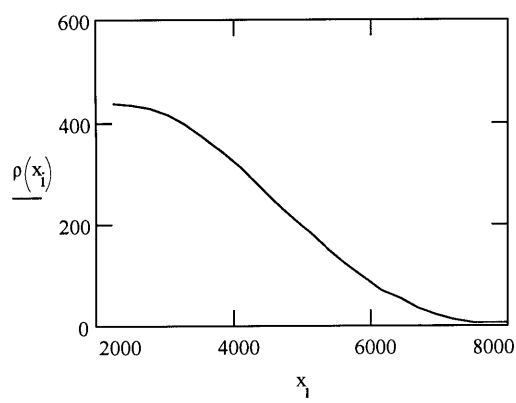
The transient rods are normally operated as a bank (gang) and are moved via stepper motors rather than pneumatically. The transient rods are raised by fork-shaped devices which are each connected to variable speed stepper motors via an electromagnet and a rack and pinion gear. If moved from the "full-in" to "full-out" position, the total reactivity associated with the transient rods is 4.40β . The rod worth is as shown in Figure 5.4-11.

Motor Frequency

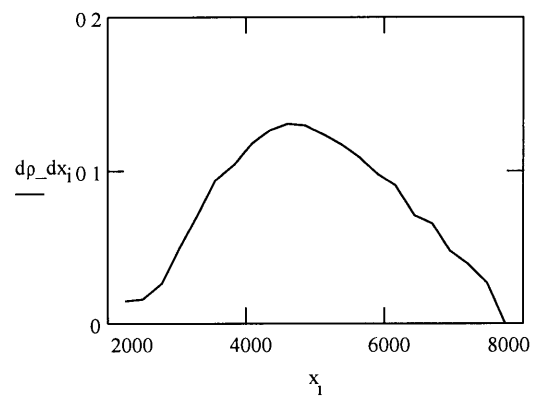
The maximum frequency of moving the transient rods is ± 400 pulses per second and there is a deadband over the range of $[-15.26, 15.26]$ pulses per second within which motor doesn't response to signals. Negative frequency represents that motor rotates in the opposite direction.

Rod Velocity

Transient rod location on the ACRR is given in "position units" with "full-in" being 2031 units and "full-out" being 7464 units. Rod positions in between are linearly interpolated to get its actual position. Every pulse delivered to the stepper motor moves the rods 2.661 units.



(A) Integral Rod Worth



(B) Differential Rod worth

Figure 5.4-11 Control Rod Worth (same as Figure 5.2-1)

5.5 Between-Group Analysis

Through within-group analysis, we conclude that the requirements are consistent within each group. However, the analysis does not rule out the possibility of between-group inconsistencies.

Between-group analysis involves the physics of the system. A *Dependency Diagram* of the case study is shown in Figure 5.5-1. In the dependency diagram, seven nodes are identified, each corresponding to one of the groups of requirements, namely, Power Multiplication, Energy (Operational Mode), Fuel Temperature, Reactor Period, Reactivity, Rod Velocity, and Motor Frequency. An intermediate state, Rate of Reactivity, is also included to connect Reactor Period with Rod Velocity, and Period with Reactivity. We have analyzed all the arcs in the between-group analyses except Power Multiplication–Fuel Temperature since this one requires a detailed thermal-hydraulic model, which is unavailable to us.

5.5.1 Power Multiplication–Reactor Period

After the within-group analysis, the requirements for group “Power Multiplication” are

Power Multiplication

The objective was to develop a robust control technique that would permit a neutronic power to be raised by up to six orders of magnitude in 5 seconds and without overshooting by 5% of nominal target power level.

The requirement for group “Reactor Period” is

Reactor Period

To satisfy Technical Specifications of the ACRR, the reactor period can not fall below 0.3 second.

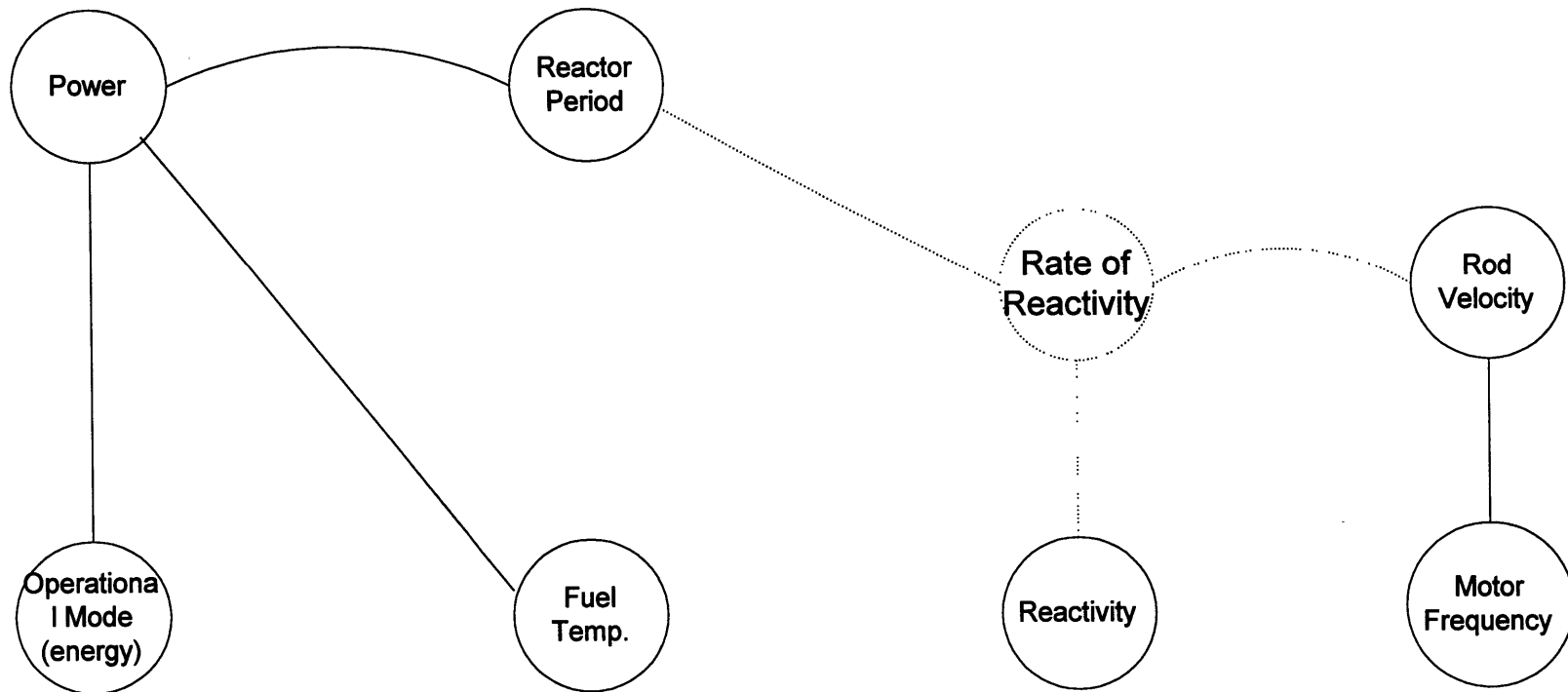


Figure 5.5-1 Dependency Diagram for the Between-Group Analysis of the Case Study

Reactor Period is defined as

$$\tau(t) = \frac{p(t)}{\dot{p}(t)} = \frac{p(t)}{\frac{dp(t)}{dt}} \quad (5.5-1)$$

Requirements in the “Power Multiplication” group specify two points on the power plot (Figure 5.5-2). However, the shape of the power trajectory is not specified. Three types of the power profile shapes will be taken into account, as identified in Figure 5.5-2. The relationship between the slopes of the three curves at $t=0^+$ is that the slope of curve III (concave) is less than that of curve I (linear), which is less than that of curve II (convex). Starting with the simplest, we will analyze all the three trajectories.

Trajectory I: Linear power shape between two points in the power plot.

For a linear power trace, it should be noted that the rate of change in power (i.e., slope of the trajectory) is constant at any time. According to Equation (5.5-1), the reactor period at time 0^+ is

$$\begin{aligned} \tau_I(0^+) &= \frac{p(0^+)}{\dot{p}(t)|_{t=0^+}} \\ &= \frac{p(0)}{\frac{10^6 \cdot p(0) \cdot 1.05 - p(0)}{5 - 0}} \\ &\approx 5 \times 10^{-6} \ll 0.30 \text{ sec} \end{aligned} \quad (5.5-2)$$

Therefore, Trajectory I can not satisfy both requirements in “Power Multiplication” and “Reactor Period”. In other words, if Trajectory I had been the power shape, the between-group analysis would indicate inconsistency in the requirements of the two groups, i.e., one or more of the requirements in the two groups would have been violated. (The allowed overshooting is taken into account by the factor of 1.05.)

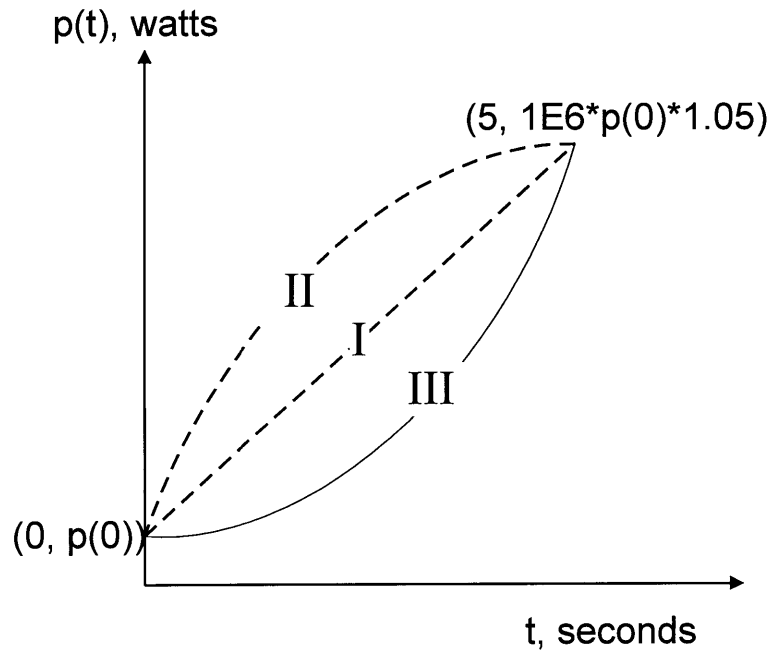


Figure 5.5-2 Possible Power Trajectories

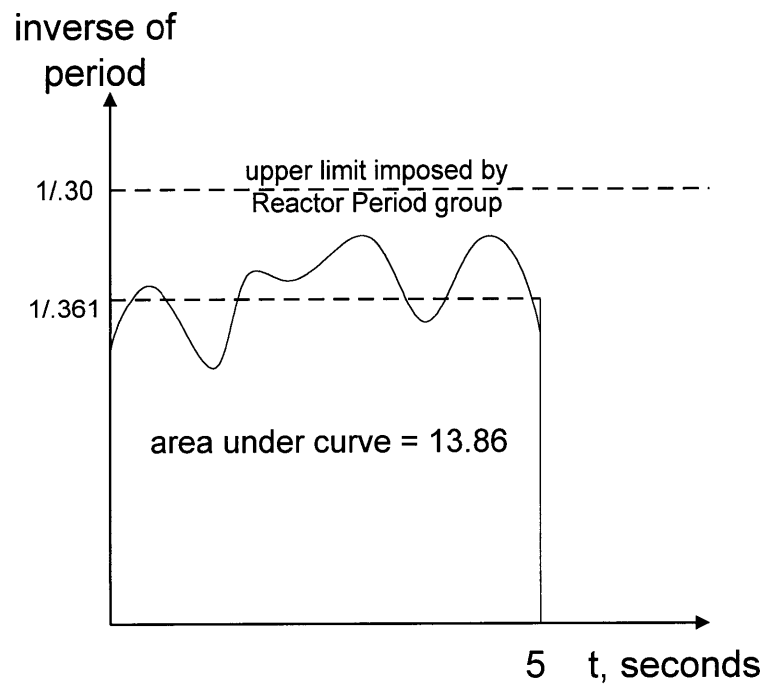


Figure 5.5-3 Possible Reactor Period Trajectory

Trajectory II: Convex power trace

For this power trace, the rate of change in power is continually decreasing and eventually approaching zero at 5 seconds, which seems very appealing. Using Equation (5.5-1), we compute the reactor period at time 0^+ .

$$\begin{aligned}\tau_{II}(0^+) &= \frac{p(0^+)}{\dot{p}(t)|_{II, t=0^+}} \\ &< \frac{p(0^+)}{\dot{p}(t)|_{I, t=0^+}} \\ &= \frac{p(0)}{10^6 \cdot p(0) \cdot 1.05 - p(0)} \\ &\quad \frac{5 - 0}{5 - 0} \\ &\approx 5 \times 10^{-6} \ll 0.30 \text{ sec}\end{aligned}\tag{5.5-3}$$

Therefore, just like Trajectory I, Trajectory II can not satisfy the between-group analysis of power–period requirements.

Trajectory III: Concave power trace

Since Trajectories I and II can not satisfy the requirements of both groups, the only possible power profile shape is Trajectory III. Integrating the Equation

(5.5-1), $\tau(t) = \frac{p(t)}{\dot{p}(t)}$, yields

$$p(t) = p(0) \cdot e^{\int_0^t \frac{1}{\tau(t)} dt}\tag{5.5-4}$$

or

$$\int_0^t \frac{1}{\tau(t)} dt = \ln\left(\frac{p(t)}{p(0)}\right)\tag{5.5-5}$$

Substituting the requirements of group Power Multiplication into Equation (5.5-5) yields

$$\int_0^5 \frac{1}{\tau(t)} dt = \ln \left(\frac{10^6 p(0) \times 1.05}{p(0)} \right) = 13.86 \quad (5.5-6)$$

Equation (5.5-7) expresses the requirements in group Power Multiplication. To satisfy the requirement in group Reactor Period, $\tau(t)$ is lower bounded by 0.30 second or $1/\tau(t)$ is upper bounded by 3.33 second⁻¹.

$$\tau(t) \geq 0.30 \quad (5.5-7)$$

In theory, $1/\tau(t)$, can be of any shape as long as it satisfies both Equations (5.5-6) and (5.5-7), i.e., the curve in Figure 5.5-3. In practice, to simplify the process, it is determined to choose an exponential power shape, a shape with constant period τ_0 (i.e., the horizontal line in Figure 5.5-3). Solving Equation (5.5-6) for constant τ_0 , we have $\tau_0=0.361$, which surely satisfies Equation (5.5-7). Therefore, the requirements are consistent between groups “Power Multiplication” and “Reactor Period” if the power shape is exponential. With the missing requirement underlined, the requirements can be rephrased as

Power Multiplication

The objective was to develop a robust control technique that would permit a neutronic power to be raised exponentially by up to six orders of magnitude in 5 seconds and without overshooting by 5% of nominal target power level.

Reactor Period

To satisfy Technical Specifications of the ACRR, the reactor period can not fall below 0.3 second.

From the analysis, it is indicated that the between-group analysis will not only check whether the requirements of the two groups under consideration are consistent, but it will also make the requirements more complete.

5.5.2 Power Multiplication-Operational Mode

The requirements for the group “Power Multiplication” are

The objective was to develop a robust control technique that would permit a neutronic power to be raised exponentially by up to six orders of magnitude in 5 seconds and without overshooting by 5% of nominal target power level.

Also, the requirements for the group “Energy (Operational Mode)” include

The reactor will operate in pulsed mode, rather than the steady state mode, and there will be a limit of 500 MJ in total energy generated per pulse.

This specifies that the total energy generated in the transient should not exceed 500 MJ. Mathematically, the two requirements lead to

$$\int_0^{t_{\text{end}}} p(t') \cdot dt' \times 1.05 \leq E_{\text{max}} = 500 \text{ MJ} \quad (5.5-8)$$

where

t_{end} is the time when the reactor is shutdown and its power is negligibly small thereafter;

1.05 is a factor reflecting the fact that a power overshoot of 5% is allowed.

Now, we realize that some important information is missing in the first group when we try to integrate the left-hand side of Equation (5.5-8). Namely, the time during which the power is maintained at the target level (t_2) and the time at which the reactor is shutdown and its power becomes negligible (t_{end}). Rewriting Equation (5.5-8) yields

$$\left(\int_0^5 p(0) \cdot e^{t'/\tau} \cdot dt' + \int_5^{t_2} p(0) \cdot 10^6 \cdot dt' + \int_{t_2}^{t_{end}} p(t') \cdot dt' \right) \times 1.05 \leq E_{max} = 500 \text{ MJ} \quad (5.5-9)$$

The first term on the left-hand side of the equation represents the energy generated during the first 5 seconds of the transient when the reactor power increases exponentially. The second term represents the energy generated when the power is maintained at the target level. The third term refers to the energy generated (i.e., decay heat) after the reactor is shut down. Neglecting the third term¹¹ and solving for an upper bound of the initial power level, Equation (5.5-9) can be rewritten as

$$p(0) = \frac{500}{(t_2 - 5 + \tau) \times 1.05} \quad (5.5-10)$$

where

τ is the reactor period (=0.361 second) during the first five seconds of the transient

Thus, the between-group analysis leads to a new constraint mathematically described by Equation (5.5-10) and graphically shown in Figure 5.5-4. Not specified in the initial set of requirements, the new requirement is the direct result of the between-group analysis. Since the third term in Equation (5.5-9) is neglected, the actual constraint on the initial power level should be slightly smaller than the corresponding value in Equation (5.5-10) or Figure 5.5-4. The figure indicates that the longer the power is maintained at the target level, the lower the

¹¹ Personal communication with Dr. John Bernard (1998) indicated that the term is usually negligibly small.

initial power level is allowed. The figure sets an upper limit for the initial power. For example, if the target power is maintained for 5 seconds after reaching the target power (i.e., $t_2=10$ seconds), the maximum allowable initial power is 88.8 watts. Therefore, the between-group analysis has revealed omissions in the requirements of one group, i.e., the constraint on the initial power level should be added.

After the between-group analysis, the requirements for each group can be revised as

Power Multiplication

The objective was to develop a robust control technique that would permit a neutronic power to be raised exponentially by up to six orders of magnitude in 5 seconds and without overshooting by 5% of nominal target power level. And for power to be maintained at the target power level until t_2 seconds (i.e., $t_2>5$), the initial reactor power level is upper bounded by the corresponding value in Figure 5.5-4.

Operational Mode

The reactor will operate in transient mode, rather than steady state mode, and there will be a limit of 500 MJ in total energy, rather than power generated per transient.

Through the between group analysis, an important goal is achieved. Namely, the between-group analysis has revealed incompleteness or omissions in the requirements of one group, i.e., it identified a constraint on the initial power level that needed to be added. Omission is the most common cause of errors in software requirements, as identified by other independent studies^[Rush96].

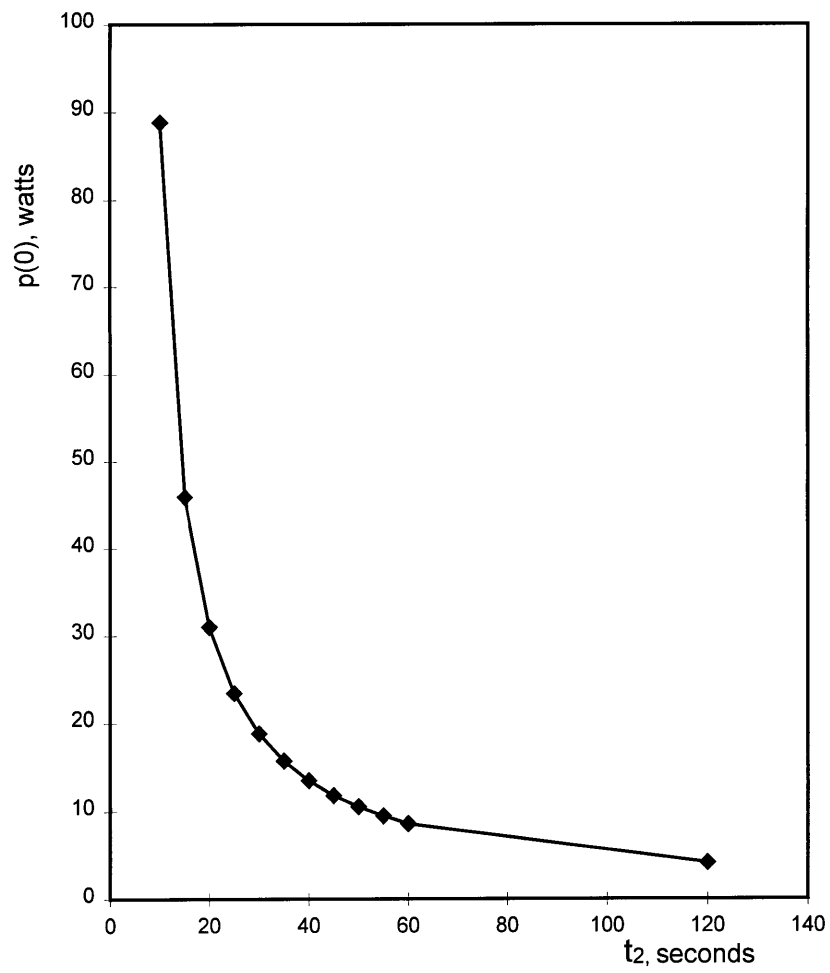


Figure 5.5-4 A New Requirement on the Initial Power Levels

5.5.3 Reactor Period–Rod Velocity

The requirements for group “Reactor Period” is

To satisfy Technical Specifications of the ACRR, the reactor period can not fall below 0.3 second.

The requirements for group “Rod Velocity” is

Transient rod location on the ACRR is given in "position units" with "full-in" being 2031 units and "full-out" being 7464 units. Rod positions in between are linearly interpolated to get its actual position. Every pulse delivered to the stepper motor moves the rods 2.661 units.

The two groups are not related with each other directly. However, they are related by an intermediate group, “Rate of Reactivity”, which is not a member of the original set of groups. Therefore, the analysis is split into two between-group analyses, *Reactor Period–Rate of Reactivity* and *Rate of Reactivity–Rod Velocity*.

5.5.3.1 Reactor Period–Rate of Reactivity

The between-group analysis makes use of group “General Information”, which requires use of the MIT-SNL Period-Generated Minimum Time Control Laws. The control laws physically relate Reactor Period and Rod Velocity. The control laws differ from other nuclear reactor control theories in that the control laws use the rate of change of reactivity as the means of reactor control, rather than reactivity itself. To maintain a reactor period of 0.361 second, the resultant rate of change of reactivity, calculated from the MIT-SNL Minimum Time Control Laws, is shown in Figure 5.5-5. A detailed analysis is provided in Appendix 10. Note that, for simplicity, the Doppler effect in the fuel is not taken into account.

5.5.3.2 Rate of Reactivity–Rod Velocity

The required rate of reactivity is achieved by the movement of the transient rods, which are implemented by signaling the electric motor to rotate. The two quantities are related by the differential rod worth, $\frac{d\rho}{dx}$, which is a function of transient rod position, x .

$$\frac{d\rho}{dt} = \frac{d\rho}{dx} \cdot \frac{dx}{dt} = \frac{d\rho}{dx} \cdot v(t) \quad (5.5-1)$$

or

$$v(t) = \frac{d\rho}{dt} \bigg|_t \bigg/ \frac{d\rho}{dx} \bigg|_x \quad (5.5-2)$$

Substituting into this equation the differential rod worth (Figure 5.4-11), the initial rod position x_0 , and the rate of change of reactivity in Figure 5.5-5, we calculate the transient rod velocity as a function of time. The result of the calculation is shown in Figure 5.5-6. Details of the analysis are given in Appendix 10.

Figure 5.5-5 shows that the rate of reactivity addition is largest at the onset of the transient. Correspondingly, the rod velocity is at its maximum at the onset of the transient (Figure 5.5-6). This maximum rod velocity is orders of magnitude larger than the rod velocity a few seconds afterwards. Minimizing the maximum rod velocity will, in return, minimize the torque required from the motor, thus minimizing the chance of motor stall and mechanical wear-out. Since the differential rod worth is maximized at the middle of the reactor core (see Figure 5.4-11), minimizing the maximum rod velocity can be achieved with an initial rod position between 4,000 and 5,000 units.

This relationship between the maximum rod velocity and the initial transient rod position is shown in Figure 5.5-7. Each dot in the figure corresponds to one run in part 4 of Appendix 10. Each run generates a figure similar to Figure 5.5-6 whose maximum speed is represented by a dot in Figure 5.5-7. From Figure 5.5-7, we get the minimum value of the maximum rod speeds in order to achieve the required rate of change of reactivity. The minimum value is 2133.7 units/sec when the control rod is initially positioned at 4562 units.

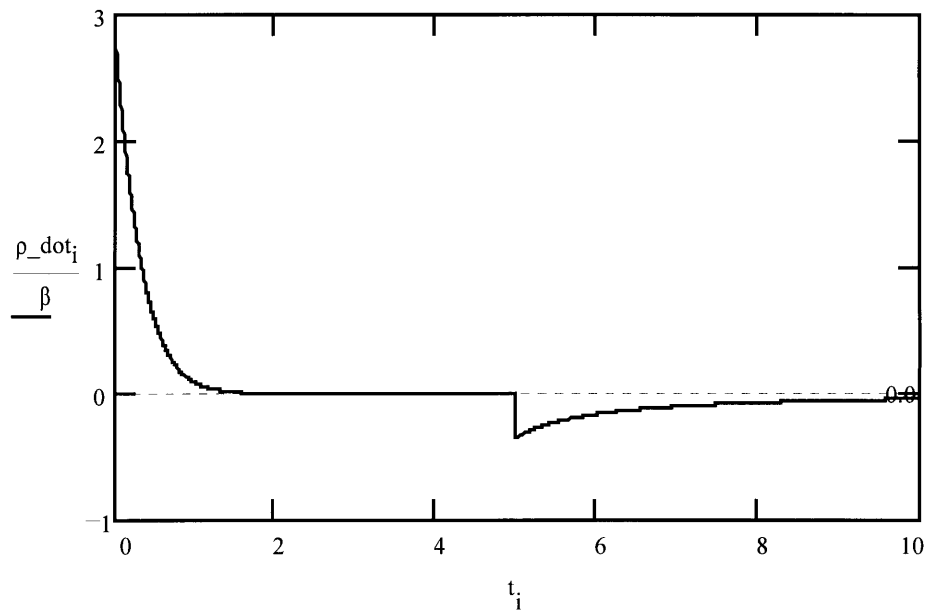


Figure 5.5-5 Rate of Change of Reactivity (in \$/sec)

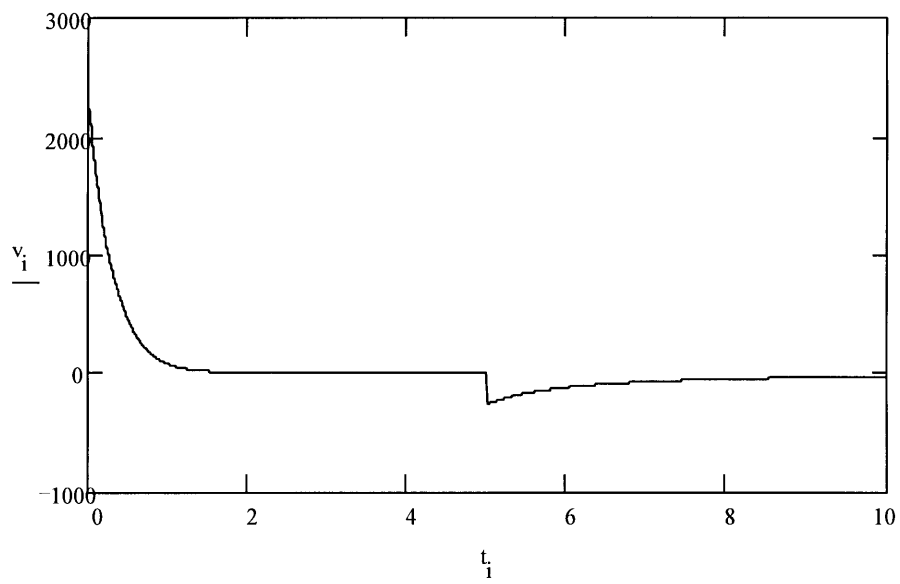


Figure 5.5-6 Dynamic Rod Velocity with Initial Rod Position at 4062 Units

5.5.4 Rod velocity–Motor Frequency

Motor Frequency

The maximum frequency of moving the transient rods is ± 400 pulses per second and there is a deadband over the range of $[-15.26, 15.26]$ pulses per second within which motor doesn't response to signals. Negative frequency represents that motor rotates in the opposite direction.

Velocity of Rod

Transient rod location on the ACRR is given in "position units" with "full-in" being 2031 units and "full-out" being 7464 units. Rod positions in between are linearly interpolated to get its actual position. Every pulse delivered to the stepper motor moves the rods 2.661 units.

The maximum allowed control rod speed is

$$2.661 \text{ units / pulse} \times (\pm 400 \text{ pulses / sec}) = 1064.4 \text{ units / sec} < 2133.7 \text{ units / sec}$$

This value is only half of the minimum value required. This fact implies that the requirements are not consistent. The minimum required maximum frequency of moving the transient rods is

$$2133.7 \text{ units / sec} \div 2.661 \text{ units / pulse} = 802 \text{ pulses / sec}$$

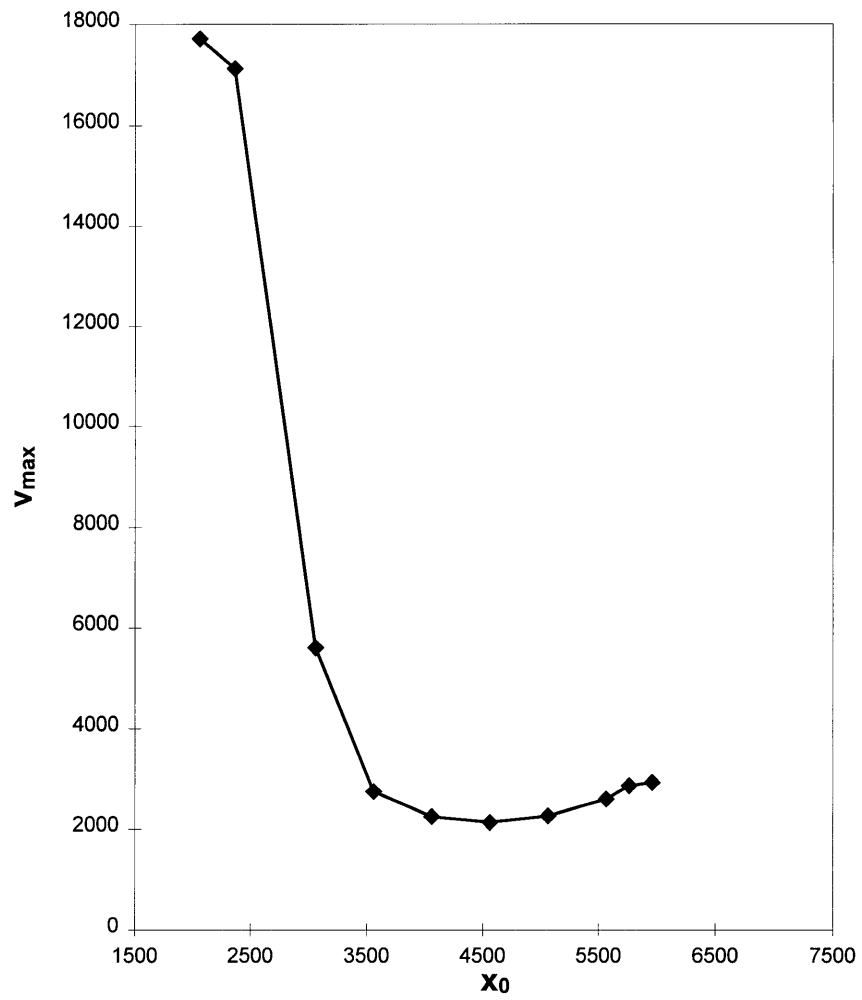


Figure 5.5-7 Maximum Rod Velocity vs. Initial Rod Position

The revised requirements should be rephrased as

Motor Frequency

The maximum frequency of moving the transient rods is ± 802 pulses per second and there is a deadband over the range of $[-15.26, 15.26]$ pulses per second within which motor doesn't response to signals. Negative frequency represents that motor rotates in the opposite direction.

Velocity of Rod

Transient rod location on the ACRR is given in "position units" with "full-in" being 2031 units and "full-out" being 7464 units. Rod positions in between are linearly interpolated to get its actual position. Every pulse delivered to the stepper motor moves the rods 2.661 units.

5.5.5 Operational Mode–Fuel Temperature

The analysis between the groups requires a detailed thermal-hydraulic analysis. A thermal-hydraulic tool, with the detailed thermal-hydraulical, geometrical, and physical data was unavailable, so the analysis is not further discussed here.

5.5.6 Reactor Period–Reactivity

Although constraint on reactivity was removed (i.e., $\rho < \beta$ was not required), this analysis is still necessary. This is because, there should be enough reserved reactivity during the transient to compensate for the needed reactivity. In fact, this is indeed the case, i.e., ρ_{needed} (figures in parts 3 and 5 of Appendix 10) $< \rho_{\text{reserved}}$ (Figure 5.2-1). Therefore, the between-group analysis is consistent.

5.5.7 Revised Requirements after the Between-Group Analysis

Through the analysis in this chapter, it has been demonstrated that Requirements Analysis will detect inconsistencies (both logical and physical), remove irrelevant information, and help achieve a more complete set of requirements. The application of the graphical requirements analysis approach developed to the case study demonstrates not only its effectiveness as a requirements analysis tool, but a user-friendly one. The initial set of requirements in the case study was reduced by half after the two-step analysis. The final set of requirements is listed as below.

General Information

130-2

- 5) The following restrictions would be incorporated in the software
 - Use of the standard, sufficient reactivity constraint in a supervisory role.
 - Use of the absolute, sufficient energy constraint in a supervisory role
- MIT-SNL Period-Generated Minimum Time Control Law should be used in the control software.

Power Multiplication

The objective was to develop a robust control technique that would permit a neutronic power to be raised exponentially by up to six orders of magnitude in 5 seconds and without overshooting by 5% of nominal target power level. And for power to be maintained at the target power level until t_2 seconds (i.e., $t_2 > 5$), the initial

reactor power level is upper bounded by the corresponding value in Figure 5.5-8.

Operational Mode/Energy

The reactor will operate in transient mode, rather than steady state mode, and there will be a limit of 500 MJ in total energy, rather than power generated per transient.

Fuel Temperature

There is a limit of 1800°C on the fuel temperature.

Reactor Period

To satisfy Technical Specifications of the ACRR, the reactor period can not fall below 0.3 second.

Reactivity

The transient rods are normally operated as a bank (gang) and are moved via stepper motors rather than pneumatically. The transient rods are raised by fork-shaped devices which are each connected to variable speed stepper motors via an electromagnet and a rack and pinion gear. If moved from the "full-in" to "full-out" position, the total reactivity associated with the transient rods is 4.40β . The control rod worth is Figure 5.5-9.

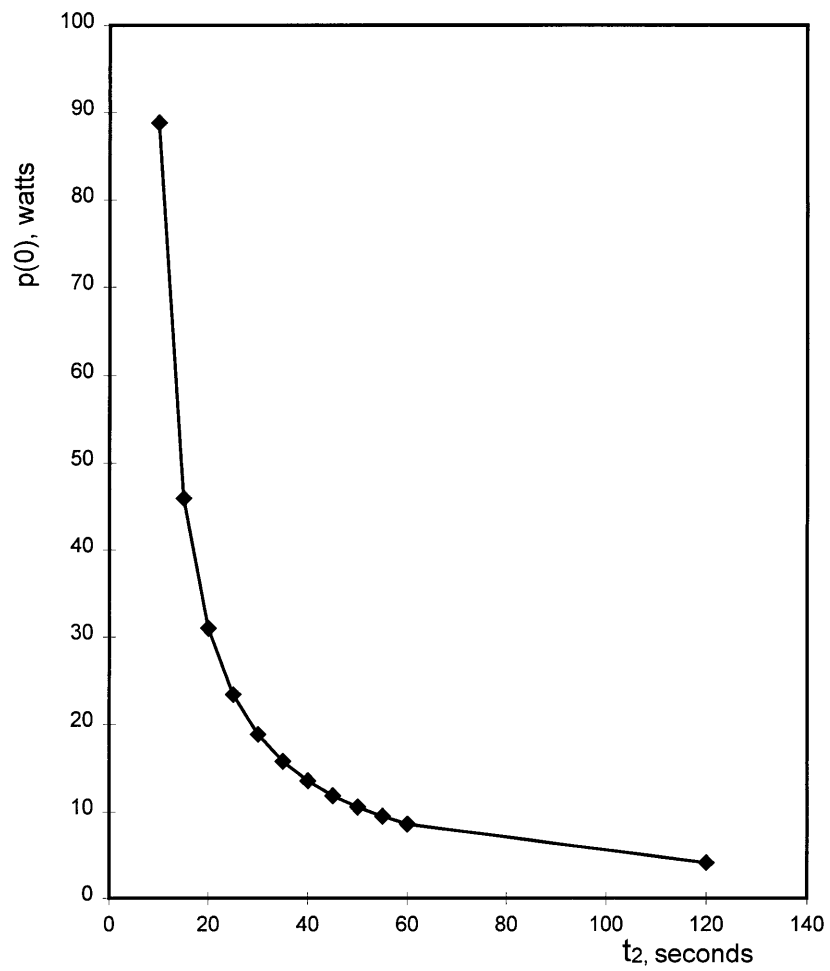


Figure 5.5-8 A New Requirement on the Initial Power Levels
(same as Figure 5.5-4)

Motor Frequency

The maximum frequency of moving the transient rods is ± 802 pulses per second and there is a deadband over the range of $[-15.26, 15.26]$ pulses per second within which motor doesn't response to signals. Negative frequency represents that motor rotates in the opposite direction.

Velocity of Rod

Transient rod location on the ACRR is given in "position units" with "full-in" being 2031 units and "full-out" being 7464 units. Rod positions in between are linearly interpolated to get its actual position. Every pulse delivered to the stepper motor moves the rods 2.661 units.

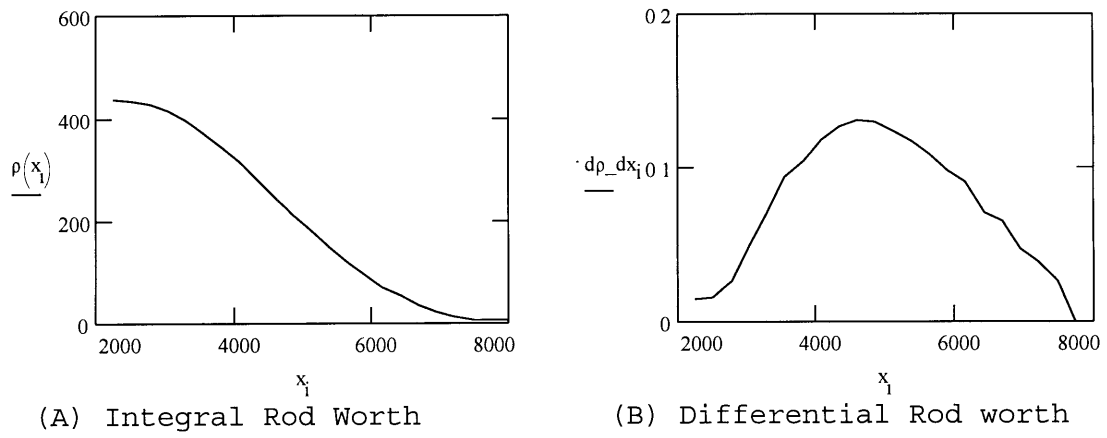


Figure 5.5-9 Control Rod Worth (same as Figure 5.2-1)

Chapter 6

Conclusions and Future Work

6.1 Conclusions and Findings

A new graphical approach has been developed for software requirements analysis. The approach consists of three distinct steps: requirements collection, requirements grouping, the within-group analysis and the between-group analysis. The purpose of this analysis is to detect inconsistencies, to remove all irrelevant information, and to achieve completeness in the requirements.

The tree-based graphical method for the within-group requirements analysis is syntactically very similar to fault tree analysis. This is important, since fault tree analysis is a familiar technique for system engineers. Based on discrete mathematics, the graphical method is a more mechanical means of performing logical analyses. Compared with the pure analytical method, which works sequentially, the graphical method works in parallel, is more informative, and easier to understand for engineers. As a by-product, this method can be applied to prove logical identities (Appendix 6).

Two additional approaches were identified for the within-group analysis: the truth-table approach and the DFM approach. The advantage of the truth-table approach is that it is mechanical and one can simply enumerate all of the permutations of the truth value assignments to the variables in the logical expressions. Unfortunately, for a logical expression involving n independent variables, there are 2^n rows in the truth table. The DFM approach is a computerized tool, allowing the search for prime implicants to be done automatically by computer. However, decision tables have to be manually supplied. The decision tables are essentially truth tables. Therefore, the method is also an exponential

approach. Nevertheless, the two approaches can act as independent checks of the graphical approach.

Through the within-group analysis, it is concluded whether the requirements within each individual group are consistent and unambiguous. However, the analysis doesn't rule out the possibility of inconsistency between groups. The between-group analysis is performed using dependency diagrams.

Several benefits were identified in the case study. First, the concept of "divide-and-conquer" helps solve a complex problem by splitting it into sub-problems. Solving these smaller problems collectively solves the complex problem. Second, the within-group analysis removes ambiguity and inconsistency within a group. Third, the between-group analysis removes inconsistency between groups, and reduces ambiguity and incompleteness. It was demonstrated that logic analysis alone is not sufficient in software requirements analysis, other analyses involving knowledge of the system, e.g., physical laws, are necessary.

Application of the graphical approach to the case study involving the MIT-SNL Period-Generated, Minimum Time Control Laws reveals several ambiguities, one inconsistency, and two omissions in the requirements collected. The case study illustrates that consistency requires that each entity has exactly one name. Otherwise, ambiguity will arise.

6.2 Future Work

Currently, the tree-based graphical method for the within-group analysis is a manual process. Although the manual analysis may give analyst insight into the source of inconsistencies and how to resolve them, the process can be error-prone because humans may make mistakes. Therefore, future work should be focused on either developing a specific tool to automate the within-group analysis or using existing automated fault tree analysis tools which have the capability to handle non-coherent trees.

Experience with the application of the DFM tool to the within-group analysis indicates that DFM can be such a candidate. However, several problems with the current version of the tool need to be solved so as to make it user-friendly. The problems include the following

- All user files (e.g., model representation) must be in the /DFM directory. This makes the directory structure confusing and unwieldy;
- A label can not be placed on a transfer box. This makes the model less informative and communicative. Even more severe, if two transfer boxes happen to have the same name, the structure of the model will become unpredictable;
- DFM is not robust enough to handle errors. For example, deleting an unselected connection will crash the program;
- DFM does not support model modification well. For example, when building a new model based on an existing one, the original logic relations do not get overwritten in the table file (i.e., file postfixed with `.tbl`) even if the decision tables are changed appropriately;
- On the “File” menu bar, the “Save” command does not seem to work properly. As a result, “Save as” is used to save changes committed to the model. On the “Edit” menu bar, the “Cut” command works only for “Hotspot” and not for “Connection”, and the opposite is true for “Delete” command.

The application of another graphical method, developed in Appendix 7, is recommended. Section 2.3.1.4 shows its advantages over the tabular notation.

Another area of future work is the selection of “good” programming language for mission-critical application such as nuclear reactor control. A presently (and perhaps permanently) unfulfilled dream in computer science is the development a universal programming language that is suited to all programs written by all programmers, working on all platforms. In past years, people have tried to create such a language. Algol-60 was one of these attempts. Ironically, its greatest success was resulting in dozens of new languages. Later attempts included PL/1, backed by IBM, and Ada, sponsored by U.S. Department of Defense. None of them has made the dream come true, however. Nowadays, there exist many programming languages, among which are Ada, C, C++, Fortran, and Pascal. More recently, research^[INEL95] has indicated that programming languages do strongly affect the readability and maintainability of software. Although no programming language will solve the problems caused by a poor design, the implementation of a good system design can be hampered by a programming language with limited expressive power^[Bell87]. Therefore, the selection of a good programming language can be crucial in developing safety-critical software. In a contract with the U.S. Air Force to upgrade its Improved Many-On-Many (IMOM) electronic combat model, INEL([INEL95]) studied the role of programming languages in developing and supporting a well-structured software implementation. It was concluded that the Ada programming languages is superior to Fortran and C in terms of abstraction, information hiding, encapsulation, modularization, and time needed to complete the project. Ada also provides the benefits of better object-oriented programming design techniques. Methods such as the Analytic Hierarchy Process might be applicable to the selection of a “good” programming language^{[Paul],[Zio]}.

References

A

- [Abri81] Abrial J-R, "A Course on System Specification", Technical Report, Oxford University Computing Laboratory, Programming Research Group, Oxford, UK, 1981.
- [Alag78] Alagic Suad, Arbib Michael A., "The Design of Well-Structured and Correct Programs", Springer-Verlag, 1978.
- [Alfo77] Alford M.W., "A Requirements Engineering Methodology for Real Time Processing Requirements", *IEEE Transactions on Software Engineering*, SE-3, Vol.1, pp.60-69, 1977.
- [Alfo85] Alford M.W., "SREM at the Age of Eight: the Distributed Computing Design System", *IEEE Computer*, Vol.18, No.4, pp.36-46, 1985.
- [Alle83] Allen James F., "Maintaining Knowledge about Temporal Intervals", *Communications of the ACM*, Vol.26, No.11, pp.832-843, November 1983.
- [Andr86] Andriole Stephen J. (editor), "Software Validation, Verification, Testing, and Documentation", Petrocelli Books, 1986.
- [Apos94] Apostolakis George E. (Editor-in-chief), *Reliability Engineering & System Safety*, Vol.43, Specific Issue on Software Safety, 1994.
- [Arch90] Archinoff R.H., Hohendorf R. J., Wassying A., Quigley B., Borsch M.R., "Verification of the Shutdown System Software at the Darlington Nuclear Generating Station", *Proceedings of the International Conference on Control and Instrumentation in Nuclear Installation*, the Institute of Nuclear Engineers, Glasgow, UK, No.4.3, 23pp., May 1990.
- [ASCA] ASCA, Inc., "DFM Software Tool Demo", undated.

B

- [Back86] Backhouse Roland C., "Program Construction and Verification", Prentice Hall International Series in Computer Science edited by C.A.R. Hoare, 1986.
- [Basi84] Basili V.R., Perricone B.T., "Software Errors and Complexity: An Empirical Investigation", *Communications of the ACM*, Vol.27, No.1, 1984.
- [Bast96] Bastani Farakh, Cukic Bojan, Hilford Victoria, Jamoussi Anouar, "Toward Dependable Safety-Critical Software", *IEEE* ?, pp.86-92, 1996.

- [Bech92] Bechta Dugan, Baruso S.J., Boyd M.A., "Dynamic Fault Tree Models for Fault Tolerant Computer Systems", *IEEE Transactions on Reliability*, Vol.41, No.3, pp.363-377, 1992.
- [Beck93] Beckjord Eric S., Cunningham Mark A., Murphy Joseph A., "Probabilistic Safety Assessment Development in the United States 1972-1990", *Reliability Engineering and System Safety*, Vol.39, pp.159-170, 1993.
- [Beiz90] Beizer Boris, "Software Testing Techniques", Van Nostrand Reinhold, New York, 2nd edition, 1990.
- [Bell87] Bell Doug, Morrey Ian, Pugh John, "Software Engineering — A Programming Approach", Prentice-Hall International (UK), 1987.
- [Bern89] Bernard John A, "Formulation and Experimental Evaluation of Closed-Form Control Laws for the Rapid Maneuvering of Reactor Neutronic Power", MIT Nuclear Reactor Laboratory, Report No. MITNRL-030, September 1989.
- [Bern90] Bernard John A., "Startup and Control of Nuclear Reactors Characterized by Space-Independent Kinetics", MIT Nuclear Reactor Laboratory, Report No. MITNRL-039, May 1990.
- [Besn95] Besnard Philippe, Hunter Anthony, "Quasi-Classical Logic: Non-trivializable Classical Reasoning from Inconsistent Information", *Proceedings of European Conference ECSQARU'95*, (Lecture Notes in Artificial Intelligence No.946), pp.44-51, Fribourg, Switzerland, July 3-5, 1995
- [BeuA93] Beu-Ari M., "Mathematical Logic for Computer Science", Prentice Hall International Series in Computer Science edited by C.A.R. Hoare, 1993.
- [Bjør95] Bjørlo Thorbjørn J., Berg Øivind, Pehrsen Morten, Dahll Gustav, Sivertsen Terje, "Lessons Learned from Experience with Development and Quality Assurance of Software Systems at the Halden Project", HWR-418, OECD Halden Reactor Project, Institutt for Energiteknikk, Norway, August 1995.
- [Bjør82] Bjørner Dines, Jones Cliff B., "Formal Specification and Software Development", Prentice Hall International Series in Computer Science edited by C.A.R. Hoare, 1982.
- [Blac96] Blackburn Mark R., Busser Robert D., "T-VEC: A Tool for Developing Critical Systems", *COMPASS'96, Proceedings of the Eleventh Annual Conference on Computer Assurance*, pp.237-249, Gaithersburg, Maryland, USA, June 17-21, 1996.
- [Bloo86] Bloomfield Robin E., Froome Peter K.D., "The Application of Formal Methods to the Assessment of High-Integrity Software", *IEEE Transactions on Software Engineering*, Vol.SE-12, No.9, pp.988-993, September 1986.
- [Boeh88] Boehm Barry W., "A Spiral Model of Software Development and Enhancement", *IEEE Computer*, Vol.21, No.5, pp.61-72, May 1988.

- [Bouz95] Bouzid Maroua, Ladkin Peter B., "Simple Reasoning with Time-Dependent Propositions", 1995 (?, undated).
- [Bowe93] Bowen Jonathan, Stavridou Victioria, "Safety-Critical System, Formal Methods and Standards", *Software Engineering Journal*, Vol.8, No.4, pp.189-209, July 1993.
- [Bowe95] Bowen Jonathan P., Hinchey Michael G., "Seven More Myths of Formal Methods", *IEEE Software*, Vol.12, Issue 4, pp.34-41, July 1995.
- [BowJ] Bowen J. P., Stavridou V., "Formal Methods and Software Safety", undated.
- [BowW] Bowen William C., Archinoff Glenn H., Raina Vijay M., Tremaine David R., Levenson Nancy G., "An Application of Fault Tree Analysis to Safety Critical Software at Ontario Hydro", undated.
- [Broo96] Broomfield E.J., Chang P.W.H., "Safety Assessment and the Software Requirements Specification", *Reliability Engineering & System Safety*, Vol.51, 1996.
- [Broo87] Brooks Frederick P. Jr., "No Silver Bullet: Essence and Accidents of Software Engineering", *IEEE Computers*, Vol.20, No.4, pp.10-19, April 1987.
- [Brya86] Bryant Randal E., "Graph-Based Algorithms for Boolean Function Manipulation", *IEEE Transactions on Computers*, Vol.C-35, No.8, August 1986.
- [Burc90] Burch J.R., Clarke E.M., McMillan K.L., Dill D.L., Hwang L.J., "Symbolic Model Checking: 10^{20} States and Beyond", *Proceedings of the 5th Annual IEEE Symposium on Logic in Computer Science*, Philadelphia, PA, June 4-7, 1990.
- [Butl93] Butler Ricky W., Johnson Sally C., "Formal Methods for Life-Critical Software", pp.319-329, *AIAA in Aerospace 9 Conference*, San Diego, Ca., October 19-21, 1993.
- [Butl95] Butler Richy W., "An Elementary Tutorial on Formal Specification and Verification Using PVS 2", NASA Technical Memorandum 108991, revised June 1995.

C

- [Cham90] Champine G.A., Geer D.E. Jr., Ruh W.N., "Project Athena as a Distributed Computer System", *IEEE Computer*, pp.40-50, September 1990.
- [Chan73] Chang Chin-Liang, Lee Richard Char-Tung, "Symbolic Logic and Mechanical Theorem Proving", Academic Press, 1973.
- [Chan85] Chang Chin-Liang, "Introduction to Artificial Intelligence Techniques", JMA Press Inc., 1985.

- [Chap95] Chapin Douglas M., “The Nuclear Industry in a Digital World”, interview with *Nuclear News*, pp.23-25, December 1995.
- [Chen98] Chen Xinhui, Apostolakis George, “A Graphical Approach to Software Requirements Analysis”, accepted for presentation at *the International Topical Meeting on Safety of Operating Reactors*, American Nuclear Society, Cathedral Hill Hotel, San Francisco, October 11-14, 1998.
- [Chor84] Chorafas Dimitris N., “The Software Handbook”, 1984.
- [Clar86] Clarke E., Emerson E., Sistla A., “Automatic Verification of Finite Concurrent Systems Using Temporal Logic Specifications”, *ACM Transactions on Programming Languages and Systems*, Vol.8, N0.2, pp.244-263, April 1986.
- [Clar96] Clarke Edmund M., Kurshan Robert P., “Computer-Aided Verification”, *IEEE Spectrum*, June 1996.
- [Cohe90] Cohen Edward, “Programming in the 1990s—An Introduction to the Calculation of Programs”, Springer-Verlag, 1990.
- [Cole92] Coleman D., “1992 Software Engineering Productivity Conference Proceedings”, Hewlett-Packard Company, 1992.
- [Cong97] Congress, “Computers: Worry Over Year 2000 Problem Has Hill Pressing for Action”, *Congressional Quarterly*, pp.3124-3125, December 1997.
- [Corm90] Corman Thomas H., Leiserson Charles E., Rivest Ronald L., “Introduction to Algorithms”, the MIT Press, 1990.
- [Cour93] Courtois P.-J., Parnas D.L., “Documentation for Safety Critical Software”, *Proceedings of the 15th International Conference on Software Engineering*, Baltimore, 1993.
- [Crai93] Craigen Dan, Gerhart Susan, Ralston Ted, “Formal Methods Reality Check: Industrial Usage”, *FME'93 Industrial-Strength Formal Methods, Proceedings of First International Symposium of Formal Methods Europe*, Odense, Denmark, pp.250-267, April 1993.
- [Crai94] Craigen Dan, Gerhart Susan, Ralston Ricis Ted, “Case Study: Darlington Nuclear Generating Station”, *IEEE Software*, pp.30-31, January 1994.
- [Crai95] Craigen Dan, Gerhart Susan, Ralston Ted, “Formal Methods Reality Check: Industrial Usage”, *IEEE Transactions on Software Engineering*, Vol.21, No.2, pp.90-98, February, 1995.
- [Cuki97a] Cukier Michel, Arlat Jean, Powell David, “Frequentest and Bayesian Coverage Estimations for Stratified Fault-Injection”, LAAS-CNRS Research Report 96336, France, Revised on January 2, 1997.
- [Cuki97b] Cukier Michel, Powell David, Arlat Jean, “Coverage Estimation Methods for Stratified Fault-Injection”, LAAS-CNRS Research Report 97001, France, January 12, 1997.

D

- [Dahl94a] Dahll Gustav, Sivertsen Terje, “A Lesson Learned Report on Software Dependability—Part I: Survey & Conclusions & Recommendations”, HWR-374, OECD Halden Reactor Project, Institutt for Energiteknikk, Norway, June 1994.
- [Dahl94b] Dahll Gustav, Sivertsen Terje, “A Lesson Learned Report on Software Dependability—Part II: Technical Basis”, HWR-375, OECD Halden Reactor Project, Institutt for Energiteknikk, Norway, June 1994.
- [Davi93] Davis S., “C++ Programmer’s Companion”, Addison-Wesley, 1993.
- [Deli90] Delisle Norman, Garlan David, “A Formal Specification of an Oscilloscope”, *IEEE Software*, pp.29-36, September 1990.
- [DeMa91] De Martinis A, Pasquini, “A Computer System Application to Improve Nuclear Plant Availability”, Safety of Computer Control Systems 1991, *Proceedings of the /IFAC/IFIP/EWICS/SRE Symposium* (edited by J.F. Lindeberg), Trondheim, Norway, 30 October-1 November 1991, pp.41-45, Pergamon Press, 1991.
- [Denv86] Denvir Tim, “Introduction to Discrete Mathematics for Software Engineering”, Macmillan Education, 1986.
- [Deut82] Deutsch Michael S., “Software Verification and Validation — Realistic Project Approaches”, Prentice-Hall Inc., 1982.
- [Dick91] Dick Jeremy, Loubersac Lérôme, “The Visual Presentation of VDM Specification”, *VDM’91 Formal Software Development Methods*, Lecture Notes in Computer Science 551 edited by S. Prehn and W.J. Toetenel, Springer-Verlag, 1991.
- [Dick78] Dickover M.E., McGowan C, Ross D.T., “Software Design Using SADT”, *Structured Analysis and Design*, Vol.2, pp.99-114, Maidenhead, England, 1978.
- [Dijk68] Dijkstra Edsger.W., “Go To Statement Considered Harmful”, *Communications of the ACM*, Vol.11, No.3, pp.147-148, March 1968.
- [Dunn84] Dunn Robert H., “Software Defect Removal”, McGraw-Hill Book Company, 1984.

E

F

- [Fair85] Fairley Richard E., "Software Engineering Concepts", McGraw-Hill, New York, 1985.
- [Feij92] Feijs L.M.G., Jonbers H.B.M., "Formal Specification and Design", Cambridge Tracts in Theoretical Computer Science 35, 1992.
- [Fink94] Finkelstein A.C.W., Gabbay D., Hunter A., Kramer J., Nuseibeh B., "Inconsistency Handling in Multiperspective Specifications", *IEEE Transactions on Software Engineering*, Vol.20, No.8, pp.569-578, August 1994.
- [Fran97] France Robert B., Bruel Jean-Michel, "Taming the Octopus: Using Formal Models to Integrate the Octopus Object Oriented Analysis Models", *Proceedings of the Second IEEE High-Assurance Systems Engineering Workshop—HASE'97* (Working Draft), Bethesda, Maryland, August 11-12, 1997.
- [Fran83] Frank Werner L., "The History of Myth No.1", *Datamation*, pp.253-256, May 1983.
- [Fran90] Franklin Gene F., Powell J. David, Workman Michael L., "Digital Control of Dynamic Systems", Addison-Wesley Publishing Company, 2nd edition, 1990.
- [Fras91] Fraser Martin D., Kumar Kuldeep, Vaishnavi Vijay K., "Informal and Formal Requirements Specification Languages: Bridging the Gap", *IEEE Transactions on Software Engineering*, Vol.17, No.5, pp.454-466, May 1991.
- [Froo88] Froome Peter, Monahan Brian, "The Role of Mathematically Formal Methods in the Development and Assessment of Safety-Critical Systems", *Microprocessors and Microsystems*, Butterworth & Co. Ltd., Vol.12, No.10, pp.539-546, December 1988.

G

- [Gann94] Gannon John D., Purtilo James M., Zelkowitz Marvin V., "Software Specification—A Comparison of Formal Methods", Ablex Publishing Company, New Jersey, 1994.
- [Garr95] Garrett Chris J., Guarro Sergio B., Apostolakis George E., "The Dynamic Flowgraph Methodology for Assessing the Dependability of Embedded Software Systems", *IEEE Transactions on Systems, Man, and Cybernetics*, Vol.25, No.5, May 1995.
- [Glas80] Glass Robert L., "Software vs. Hardware Errors", *IEEE Computer*, December 1980.
- [Grim] Grimm K., "An Effective Strategy and Automation Concepts for Systematic Testing of Safety Related Software", undated.

- [Grov94] Groven Arne Kristian, Sivertsen Terje, “Formal Software Development — A Case Study on the Development of A Reactor Safety System”, HWR-397, OECD Halden Reactor Project, Institutt for Energiteknikk, Norway, October 1994.
- [Guar84] Guarro S.B., Okrent D., “The Logic Flowgraph: A New Approach to Process Failure Modeling and Diagnosis for Distrubance Analysis Applications”, *Nuclear Technology*, 67, pp.348-359, 1984.
- [Guar88] Guarro S.B., “A Logic Flowgraph Based Concept for Decision Support and Management of Nuclear Plant Operation”, *Reliability Engineering and System Safety*, Vol.22, 1988.
- [Guar90] Guarro S.B., “Diagnostic Model for Engineering Process Management: A Critical Review of Objectives, Constraints and Applicable Tools”, *Reliability Engineering and System Safety*, Vol.30, 1990.
- [Guar96] Guarro S., Yau M., Motamed M., “Development of Tools for Safety Analysis of Control Software in Advanced Reactors”, NUREG/CR-6465, U.S. Nuclear Regulatory Commission, April 1996.
- [G ü sg95] G ü sgen Hans-Werner, Ladkin Peter B., “An Algebraic Approach to General Boolean Constraint Problems”, revised from Technical Report TR-90-008, International Computer Science Institute, Berkeley, CA, April 1995.

H

- [Hall90] Hall Anthony, “Seven Myths of Formal Methods”, *IEEE Software*, pp.11-19, September 1990.
- [Hals77] Halstead M.H., “Elements of Software Science”, Elsevier North-Holland, 1977.
- [Hami76] Hamilton M., Zeldin S., “Higher Order Software: A Methodology for Defining Software”, *IEEE Transactions of Software Engineering*, SE-2, Vol.1, pp.9-36, 1976.
- [Harr96] Harry Andrew, “Formal Methods Fact File — VDM and Z”, John Wiley & Sons, 1996.
- [Haye97a] Hayes Brian, “Can’t Get No Satisfaction”, *American Scientist*, Vol.85, pp.108-112, March-April 1997.
- [Haye97b] Hayes Brian, “Cafebare”, *American Scientist*, Vol.85, pp.304-308, July-August 1997.
- [Hech96] Hecht Myron, Tang Dong, Hecht Herbert, Brill Robert W., “Quantitative Reliability and Availability Assessment for Critical Systems Including Software”, 1996 (?, undated).

- [Heim96] Heimdahl Mats P.E., Leveson Nancy G., “Completeness and Consistency in Hierarchical State-Based Requirements”, *IEEE Transactions on Software Engineering*, May 1996.
- [Heit83] Heitmeyer C.L., McLean J., “Abstract Requirements Specifications: A New Approach and Its Application”, *IEEE Transactions on Software Engineering*, SE-9(5), September 1983.
- [Heit95a] Heitmeyer Constance, Bull Alan, Gasarch Carolyn, Labaw Bruce, “SCR*: A Toolset for Specifying and Analyzing Requirements”, COMPASS’95, *Proceedings of the Tenth Annual Conference on Computer Assurance*, pp.109-122, Gaithersburg, Maryland, June 25-29, 1995.
- [Heit95b] Heitmeyer Constance, Labaw Bruce, Kiskis Daniel, “Consistency Checking of SCR-Style Requirements Specifications”, *Proceedings of the 2nd IEEE International Symposium on Requirements Engineering*, pp.56-63, York, England, March 27-29, 1995.
- [Heit97] Heitmeyer Constance, “Formal Methods: A Panacea or Academic Poppycock?”, *Proceedings of the 10th International Conference of Z Users*, pp.3-9, Reading, UK, April 3-4, 1997.
- [Heni80] Heninger Kathryn L., “Specifying Software Requirements for Complete Systems: New Techniques and Their Application”, *IEEE Transactions on Software Engineering*, Vol.SE-6, No.1, pp.2-13, January 1980.
- [Henl92] Henley Earnest J., Kumanoto Hiromitsu, “Probabilistic Risk Assessment—Reliability Engineering, Design, and Analysis”, IEEE Press, New York, 1992.
- [Henr75] Henry Allen F., “Nuclear Reactor Analysis”, the MIT Press, 1975.
- [Hinc93] Hinchey Michael G., “Formal Methods for System Specification”, *IEEE Potentials*, Vol.12, Issue 5, pp.50-52, 1993.
- [Hoar69] Hoare C.A.R., “An Axiomatic Basis for Computer Programming”, *Communications of the ACM*, Vol.12, No.10, pp.576-583, October 1969.
- [Hoar71] Hoare C.A.R., “Proof of a Program: FIND”, *Communications of the ACM*, Vol.13, No.1, pp.39-45, January 1971.
- [Hoar75] Hoare C.A.R., “Data Reliability”, *SIGPLAN Notices*, Vol.20, No.6, pp.396-404, June 1975.
- [Holl95] Holloway C. Michael, Miner Paul S., Butler Ricky W., “Informal Introduction to Formal Methods”, (slides) *Proceedings of the 3rd NASA LaRC Formal Methods Workshop*, May 10-12, 1995.
- [Hugh95] Hughes David, Dornheim Michael A., “Accidents Direct Focus on Cockpit Automation”, *Aviation Week & Space Technology*, pp.52-54, January 30, 1995.

- [Hung93] Hung Sheung-Lun, Kwok Lam-For, and Chan Raymond, "Automatic Programming Assessment", *Computer and Education*, Vol.20, No.2, pp.183-190, 1993.
- [Hunt97] Hunter Anthony, Nuseibeh Bashar, "Analyzing Inconsistent Specifications", *Proceedings of Third IEEE International Symposium on Requirements Engineering*, Annapolis, Maryland, USA, pp.78-86, January 6-10, 1997.

I

- [IAEA94] IAEA, "Software Important to Safety in Nuclear Power Plants", Technical Reports Series No.367, International Atomic Energy Agency, Vienna, 1994.
- [IAEA95] IAEA, "Verification and Verification of Software Related to Nuclear Power Plant Control and Instrumentation", Technical Reports Series, Final Draft, Vienna, 1995.
- [IFAD97] IFDK, <http://www.ifad.dk/products/venus.html>, date of visit September 8, 1997.
- [Ince88] Ince D.C., "An Introduction to Discrete Mathematics and Formal System Specification", Clarendon Press, 1988.
- [INEL95] INEL, "Electronic Combat Model Reengineering", AJPO-MON-95-01, March 1995.

J

- [Jani95] Janicki R., Parnas D.L., Zucker J., "Tabular Representations in Relational Documents", Telecommunication Research Institute of Ontario, McMaster University, CRL Report No.313, November 1995.
- [Java97] -, "The Java™ Language: An Overview", from the internet: http://www.cs.huji.ac.il/course/java/java_overview/index.html, date of visit: summer 1997.
- [Jens81] Jensen R.W., "Tutorial Series-6: Structured Programming", *Computer*, Vol.14, No.3, pp.31-48, March 1981.
- [Joan] Joannou Paul K., "Experiences from Application of Digital Systems in Nuclear Power Plants", Ontario Hydrio, undated.
- [Joan92] Joannou Paul K., Tremaine D.R., Jankowski E., "Software Requirements Specification for Critical Applications", Ontario Hydrio, 1992(?), undated).
- [Joan94] Joannou Paul K., Harauz J., Tremaine D.R., "The Ontario Hydro/AECL Approach to Real-time Software Engineering Standards", *Reliability Engineering and System Safety*, Vol.43, pp.143-150, 1994.

- [John95] Johnsonbaugh Richard, Kalin Martin, “Object Oriented Programming in C++”, Prentice-Hall, 1995.
- [Jone96] Jones Capers, “Software Defect-Removal Efficiency”, *IEEE Computer*, Vol.29, No.4, pp.94-95, April 1996.
- [Jone80] Jones Cliff B., “Software Development — A Rigorous Approach”, Prentice Hall International Series in Computer Science edited by C.A.R. Hoare, 1980.
- [Jone90a] Jones Cliff B., “Systematic Software Development using VDM”, second edition, Prentice Hall International Ltd., 1990.
- [Jone90b] Jones Cliff B., Shaw R.C., “Case Studies in Systematic Software Development”, Prentice Hall International Series in Computer Science edited by C.A.R. Hoare, 1990.

K

- [Karg97] Karger David, Lecture Notes 1 and 2 for Course 6.042J/18.062J, Department of Electrical Engineering and Computer Science, MIT, Spring 1997.
- [Kaut91] Kautz Herry A., Ladkin Peter B., “Integrating Metric and Qualitative Temporal Reasoning”, *Proceedings of AAAI-91*, Anaheim, California, 1991.
- [Kece87] Kececioglu John D., Kececioglu Dimitri B., “Quantified Software Reliability Techniques”, *IASTED International Symposium: Quality Control and Reliability*, June 1-3, 1987.
- [Knig85] Knight John C., Ammann Paul E., “An Experimental Evaluation of Simple Methods for Seeding Program Errors”, *Proceedings of 8th International Conference on Software Engineering*, Imperial College, London, UK, August 28-30, 1985.
- [Knig86] Knight John C., Leveson Nancy G., “An Experimental Evaluation of the Assumptions of Independence in Multiversion Programming”, *IEEE Transactions on Software Engineering*, Vol.SE-12, No.1, pp.96-109, January 1986.
- [Knig97] Knight John C., DeJong Colleen L., Gibbe Matthew S., “Why Are Formal Methods Not Used More Widely? — An Extended Abstract”, submitted to NASA Formal Methods Workshop at NASA Langley Research Center, 1997.
- [Kram88] Kramer Jeff, Ng Keng, Potts Colin, Whitehead Ken, “Tool Support for Requirements Analysis”, *IEE Software Engineering Journal*, No.3, pp.86-96, May 1988.
- [Kuwa93] Kuwana Eiji, Herbsleb James D., “Representing Knowledge in Requirements Engineering: An Empirical Study of What Software Engineers Need to Know”, *Proceedings of the IEEE International Symposium on Requirements Engineering*, pp.273-276, San Diego, CA, January 4-6, 1993.

L

- [Ladk87] Ladkin Peter B., “The Logic of Time Representation”, Ph.D. Dissertation, the University of California at Berkeley, November 1987.
- [Ladk94] Ladkin Peter B., Maddus Roger D., “On Binary Constraint Problems”, *Journal of the ACM*, May 1994.
- [Ladk95] Ladkin Peter B., Reinefeld Alexander, “Fast Algebraic Methods for Interval Constraint Problems”, 1995 (?, undated).
- [Ladk95a] Ladkin Peter B., “Correctness in System Engineering”, April 1995.
- [Ladk95b] Ladkin Peter B., Leue Stefan, “Interpreting Message Flow Graphs”, to appear in *Formal Aspects of Computing*, 1995.
- [Lamb78] Lamb S.S. et al, “SAMM: A Modeling Tool for Requirements and Design Specification”, *Proceedings of 2nd IEEE International Computer Software and Applications Conference (Compsac’78)*, pp.48-53, Silver Spring, Maryland, 1978.
- [Lano90] Lano K., “Z++, An Object-Orientated Extension To Z”, Z User Workshop, *Proceedings of the 5th Annual Z User Meeting*, pp.151-172, Oxford, UK, December 17-18, 1990.
- [Lano94] Lano Kevin, Haughton Howard, “Reverse Engineering and Software Maintenance — A Practical Approach”, McGraw-Hill Book Company, London, 1994.
- [Lano97] Lano K., Goldsack S., Bicarregui J., Kent S., “Integrating VDM++ and Real-Time System Design”, *Proceedings of the 10th International Conference of Z Users*, pp.188-219, Reading, UK, April 3-4, 1997.
- [Lars96] Larsen Peter Gorm (editor), “The VDM Bibliography”, (internet version) IFAD, the Institute of Applied Computer Science, Denmark, April 16, 1996.
- [Lecl92] Leclercq Philippe R., “A Software-Reliability Assessment Model”, *Proceedings of Annual Reliability and Maintainability Symposium*, pp.294-298, 1992.
- [Ledg81] Ledgard Henry, “ADA: An Introduction”, Springer-Verlag, 1981.
- [Leig97] Leighton Tom, “What is a Proof?”, Lecture Note 1 for Course 6.042J/18.062J, Department of Electrical Engineering and Computer Science, MIT, Fall 1997.
- [Leve83] Leveson Nancy G., Harvey Peter R., “Analyzing Software Safety”, *IEEE Transactions on Software Engineering*, Vol.SE-9, No.5, September 1983.
- [Leve87] Leveson Nancy G., Stolzy Janice L., “Safety Analysis Using Petri Nets”, *IEEE Transactions on Software Engineering*, Vol.SE-13, No.3, March 1987.

- [Leve94a] Leveson Nancy G., Heimdahl Mats Per Erik, Hildreth Holly, Reese Jon Damon, "Requirements Specification for Process-Control Systems", *IEEE Transactions on Software Engineering*, Vol.20, No.9, pp.684-707, September 1994.
- [Leve94b] Leveson Nancy G., "High-Pressure Steam Engines and Computer Software", *IEEE Computer*, October 1994.
- [Leve95] Leveson Nancy G., "Safeware—System Safety and Computers", Addison-Wesley Publishing Company, 1995.
- [Leve97a] Leveson Nancy G., Pinnel L. Denise, Sandys Sean David, Koga Shuichi, Reese Jon Damon, "Analyzing Software Specifications for Mode Confusion Potential", *Human Error Workshop*, Glasgow, March 1997.
- [Leve97b] Leveson Nancy G., Palmer Ererett, "Designing Automation to Reduce Operator Errors", *Proceedings of Systems, Man, and Cybernetics*, October 1997.
- [Leve98] Leveson Nancy G., "Intent Specifications: An Approach to Building Human-Centered Specificaitons", *Requirements Engineering Conference*, April 1998 (expected date).
- [Lisk86] Liskov Barbara, Guttag John, "Abstraction and Specification in Program Development", the MIT Press, 1986.
- [Lipp95] Lippman Stanley, "C++ Primer", Addison-Wesley Publishing Company, 2nd Edition, 1995.
- [Litt95] Littlewood Bev, Neil Martin, Ostrolerk Gary, "The Role of Models in Managing the Uncertainty of Software-Intensive System", Elsevier Science Limited, pp.87-95, 1995.
- [Liu93] Liu Shaoying, "A Formal Requirements Specification Method Based on Data Flow Analysis", *J. Systems Software*, 21:141-149, 1993.
- [Loec87] Loeckx Jacques, Sieber Kurt, "The Foundations of Program Verification", second edition, Wiley-Teubner Series in Computer Science, John Wiley & Sons Ltd., 1987.
- [Lond75] London Ralph L., "A View of Program Verification", *SIGPLAN Notices*, Vol.10, No.6, pp.534-545, June 1975.
- [Lori97] Internet: <http://www.loria.fr/~molli>, data of visit: September 8, 1997.
- [Luba93] Lubars Mitch, Potts Colin, Richter Charles, "A Review of the State of the Practice in Requirements Modeling", *Proceedings of the IEEE International Symposium on Requirements Engineering*, pp.2-14, San Diego, CA, January 4-6, 1993.
- [Luka90] Lukaszewicz Witold, "Non-Monotonic Reasoning — Formalization of Commonsense Reasoning", Ellis Horwood Limited, 1990.

- [Lync89] Lynch Nancy A., Goldman Kenneth J., "Distributed Algorithms", MIT/LCS/RSS 5, May 1989.
- [Lync91] Lynch Nancy, Vaandrager Frits, "Forward and Backward Simulations for Timing-Based System", MIT/LCS/TM-458, November 1991.
- [Lync96] Lynch Nancy, Leighton Tom, "State Machines", Lecture Notes Handout 40 for Course 6.042J/18.062J, Department of Electrical Engineering and Computer Science, MIT, Fall 1996.
- [Lyu93] Lyu Michael R., "A Coverage Analysis Tool for the Effectiveness of Software Testing", *IEEE* ?, pp.25-34, 1993.

M

- [Mars96] Marshall A.C., "HPF Programming (5 Day Course) Course Notes (FORTRAN 90)", the University of Liverpool, version of November 18, 1996.
- [May95] May J.H.R., Lunn A.D., "A Model of Code Sharing for Estimating Software Failure on Demand Probabilities", *IEEE Transactions on Software Engineering*, Vol.21, No.5, pp.747-753, September 1995.
- [McCa76] McCabe T.J., "A Complexity Measure", *IEEE Transactions on Software Engineering*, 2:308-320, 1976.
- [McDo] McDougall J., Viola M., Moum G., "Tabular Representation of Mathematical Functions for the Specification and Verification of Safety Critical Software", undated.
- [Mill97] Miller Steve, "How can Requirements Engineering Research Become Requirements Engineering Practice?", *Proceedings of Third IEEE International Symposium on Requirements Engineering*, Annapolis, Maryland, USA, p.260, January 6-10, 1997.
- [Mine95] Miner Paul S., "Application of Logic to Digital System Design", (slides) *Proceedings of the 3rd NASA LaRC Formal Methods Workshop*, May 10-12, 1995.
- [Modu96] Modugno Francesmary, Leveson Nancy G., Reese Jon D., Partridge Kurt, Sandys Sean, "Creating and Analyzing Requirement Specifications of Joint Human-Computer Controller for Safety-Critical Systems", *Symposium on Human Interaction with Complex Systems (HICS'96)*, Dayton, OH., August 25-28, 1996.
- [Modu97] Modugno Francesmary, Leveson Nancy G., Reese Jon D., Partridge Kurt, Sandys Sean, "Integrated Safety Analysis of Requirement Specifications", *Proceedings of Third IEEE International Symposium on Requirements Engineering*, Annapolis, Maryland, USA, pp.148-159, January 6-10, 1997.

- [More82] Moret Bernard M.E., “Decision Trees and Diagrams”, *ACM Computing Surveys*, Vol.14, No.4, pp.593-623, December 1982.
- [Musa87] Musa John D., Iannino Anthony, Okumoto Kazuhira, “Software Reliability — Measurement, Prediction, Application”, McGraw-Hill Book Company, 1987.

N

- [Nara90] Narayana K.T., Dharap S., “Formal Specification of a Look Manager”, *IEEE Transactions on Software Engineering*, Vol.16, No.9, pp.1089-1103, September 1990.
- [Nels90] Nelson Victor P., “Fault-Tolerant Computing: Fundamental Concepts”, *IEEE Computer*, pp.19-25, July 1990.
- [Neud91] Neuder David L., “A Test Verification Tool for C and C++ Programs”, *Hewlett-Packard Journal*, Vol.42, No.2, pp.83-92, April 1991.
- [Neum96] Neumann Peter G., “Using Formal Methods to Reduce Risks”, *Communications of the ACM*, Vol.39, No.7, July 1996.
- [Ng96] Ng W.S., Tan C.K., “On Safety Enhancement for Medical Robots”, School of Mechanical and Production Engineering, Nanyang Technological University, June 1996.
- [NRC95] National Research Council, “Digital Instrumentation and Control Systems in Nuclear Power Plants: Safety and Reliability Issues”, by Committee on Application of Digital Instrumentation and Control Systems to Nuclear Power Plant Operations and Safety, National Research Council, National Academic Press, 1995.
- [NRC97] National Research Council, “Digital Instrumentation and Control Systems in Nuclear Power Plants: Safety and Reliability Issues”, Advanced Copy of Final Report by Committee on Application of Digital Instrumentation and Control Systems to Nuclear Power Plant Operations and Safety, National Research Council, National Academic Press, January 1997.

O

- [Ober96] Oberweis Andreas, Sander Peter, “Information System Behavior Specification by High Level Petris Nets”, *ACM Transactions on Information Systems*, Vol.14, No.4, pp.381-420, October 1996.
- [Ogun81] Ogunbiyi E.I., Henley E.J., “Irredundant Forms and Prime Implicants of a Function with Multistate Variables”, *IEEE Transactions on Reliability*, Vol. R-30, No.1, pp.39-42, April 1981.

- [Oman92] Oman P, "Construction and Validation of a Polynomials for Predicting Software Maintainability", University of Idaho, 1992.
- [ORNL95] ORNL, "Fortran 90 and Computational Science", version of 1995 from the internet: <http://csep1.phy.ornl.gov/csep.html>.
- [Ouya95] Ouyang Meng, Golay Michael W., "An Integrated Formal Approach for Developing High Quality Software of Safety-Critical Systems", Report No. MIT-ANP-TR-035, Department of Nuclear Engineering, MIT, September 1995.
- [Ouya] Ouyang Meng, Golay Michael W., Novak Michael S., "An Integrated Formal Approach for Developing High Quality Software for Safety-Critical Systems", undated.
- [Ovst91] Ovstedal E.O., "Using Fault Tree Analysis in Developing Reliable Software", Safety of Computer Control Systems 1991, *Proceedings of the /IFAC/IFIP/EWICS/SRE Symposium* (edited by J.F. Lindeberg), Trondheim, Norway, 30 October-1 November 1991, pp.77-82, Pergamon Press, 1991.
- [Owre96] Owre Sam, Rushby John, Shankar Natarajan, "Analyzing Tabular and State-Transition Requirements Specifications in PVS", Technical Report CSL-95-12, Computer Science Laboratory, SRI International, June 1995 (revised April 1996)

P

- [Parn72a] Parnas D.L., "A Technique for Software Module Specification with Examples", *Communications of the ACM*, Vol. 15, No. 5, May 1972.
- [Parn72b] Parnas D.L., "On the Criteria To Be Used in Decomposing Systems into Modules", *Communications of the ACM*, Vol.15, No.12, 1972.
- [Parn85] Parnas D.L., Clements P.C., Weiss D.M., "The Modular Structure of Complex Systems", *IEEE Transactions on Software Engineering*, Vol.SE-11, No.3, pp.259-266, March 1985.
- [Parn90a] Parnas D.L., Madey J., "Functional Documentation for Computer Systems Engineering", Telecommunications Research Institute of Ontario, Queen's University, Technical Report 90-287, ISSN 0836-0227, September 1990.
- [Parn90b] Parnas D.L., Van Schouwen A.J., Kwan S.P., "Evaluation of Safety-Critical Software", *Communications of the ACM*, Vol.33, No.6, pp.636-648, 1990.
- [Parn91] Parnas D.L., Asmisn G.J., Madey J., "Assessment of Safety-Critical Software in Nuclear Power Plants", *Nuclear Safety*, Vol. 32, No.2, pp.189-198, April-June 1991.

- [Parn92] Parnas D.L., "Tabular Representation of Relations", Telecommunication Research Institute of Ontario, McMaster University, CRL Report No.260, October 1992.
- [Parn93a] Parnas D.L., "Predicate Logic for Software Engineering", *IEEE Transactions on Software Engineering*, Vol.19, No.9, pp.856-862, September 1993.
- [Parn93b] Parnas D.L., "Some Theorems We Should Prove", Higher Order Logic Theorem Proving and Its Applications (*Proceedings of the 6th International Workshop, HUG'93*), No.780 Lecture Notes in Computer Science, pp.155-162, Vancouver, Canada, August 11-13, 1993.
- [Parn94a] Parnas D.L., "Software Aging", (invited plenary talk) *Proceedings of the 16th International Conference on Software Engineering*, pp.279-287, Italy, May 16-21, 1994.
- [Parn94b] Parnas D.L., Madey J., Iglewski M., "Precise Documentation of Well-Structured Programs", *IEEE Transactions on Software Engineering*, Vol.20, No.12, pp.948-976, December 1994.
- [Parn95] Parnas D.L., Madey J., "Functional Documents for Computer Systems", *Science of Computer Programming*, Vol.25, No.1, pp.41-61, October 1995.
- [Paul93] Paula H.M., Roberts M.W., Battle R.E., "Operational Failure Experience of Fault-Tolerant Digital Control System", *Reliability Engineering and System Safety*, Vol.39, pp.273-289, 1993.
- [Paul] Paulos Todd, Apostolakis George, "A Methodology to Select a Wire Insulation for Use in Habitable Spacecraft", undated.
- [Pete81] Peterson J.L., "Petri Net Theory and the Modeling of Systems", Prentice-Hall, Englewood, N.J., 1981.
- [Petr91] Petrella S., Michael P., Bowman W.C., Lim S.T., "Random Testing of Reactor Shutdown System Software", *Probabilistic Safety Assessment and Measurement* edited by G.E. Apostolakis, Elsevier Science Publishing Co. Inc., pp.681-686, 1991.
- [Pohm81] Pohm A.V., Smay T.A., "Top-Down System Design", *Computer*, Vol.14, No.6, pp.65-67, June 1981.
- [Port96] Porter A., Siy H.P., Votta L.G., "A Review of Software Inspections", *Software, Progress, Advances in Computer 42* edited by M.V. Zelkowitz, Academic Press, San Diego, 1996.
- [Pott96] Potts Anthony, Friedel David H Jr., "Java—Programming Language Handbook", Internet Programming Series, Coriolis Group Books, 1996.
- [Powe96a] Powell David, Michel Cukier, Arlat Jean, "On Stratified Sampling for High Coverage Estimations", LAAS-CNRS Research Report 96059, France, Revised on July 23, 1996.

- [Powe96b] Powell David, Michel Cukier, Arlat Jean, Yves Crouzert, “Estimation of Time-Dependent Coverage”, LAAS-CNRS Research Report 96466, France, December 4, 1996.
- [Prec93] Preckshot G.G., “Data Communications”, NUREG/CR-6082, August 1993.
- [Pres87] Pressman Roger S., “Software Engineering — A Practitioner’s Approach”, McGraw-Hill Book Company, 2nd edition, 1987.

Q

- [Quin52] Quine W.V., “The Problem of Simplifying Truth Functions”, *American Mathematical Monthly*, 59, pp.521-531, 1952.
- [Quin55] Quine W.V., “A Way to Simplify Truth Functions”, *American Mathematical Monthly*, 62, pp.627-631, 1955.

R

- [Rama96] Ramachandran Muthu, “Requirements-Driven Software Test: A Process-Oriented Approach”, *ACM SIGSOFT, Software Engineering Notes*, Vol.21, No.4, pp.66-70, July 1996.
- [Rama93] Ramakumar R., “Engineering Reliability: Fundamentals and Applications”, Prentice-Hall Inc., 1993.
- [Rand90] Randell Gill, “Data Flow Diagrams and Z”, Z User Workshop, *Proceedings of the 5th Annual Z User Meeting*, pp.216-227, Oxford, UK, December 17-18, 1990.
- [Rata96] Ratan Vivek, Patridge Kurt, Reese Jon, Leveson Nancy, “Safety Analysis Tools for Requirements Specifications”, *COMPASS'96, Proceedings of the Eleventh Annual Conference on Computer Assurance*, pp.149-160, June 17-21, 1996, Gaithersburg, Maryland, USA.
- [Redm96] Redmill F., Chudleigh M.F., Catmur J.R., “Principle Underlying a Guideline for Applying HAZOP to Programmable Electronic Systems”, 1996 (?, undated).
- [Rees96] Reese J.D., “Software Deviation Analysis”, Ph.D. Dissertation, University of California, Irvine, January 1996.
- [Rees97] Reese Jon Damon, Leveson Nancy G., “Software Deviation Analysis: A ‘Safeware’ Technique”, accepted to *the International Conference on Software Engineering*, Boston, May 1997.
- [Rose95] Rosen Kenneth H., “Discrete Mathematics and Its Application”, third edition, McGraw-Hill Inc., 1995.

- [Rose96] Rosenblum David S., “Formal Methods and Testing: Why the State-of-the Art is Not the State-of-the Practice”, *ACM SIGSOFT, Software Engineering Notes*, Vol.21, No.4, pp.64-66, July 1996.
- [Ross77a] Ross D.T., Schoman K.E., “Structured Analysis for Requirements Definition”, *IEEE Transactions on Software Engineering*, Vol.SE-3, No.1, pp.6-15, January 1977.
- [Ross77b] Ross Douglas T., “Structured Analysis (SA): A Language for Communicating Ideas”, *IEEE Transactions on Software Engineering*, Vol.SE-3, No.1, pp.16-34, January 1977.
- [Rush88] Rushby John, “Quality Measures and Assurance for AI Software”, Technical Report CSL-88-7R, Computer Science Laboratory, SRI international, Menlo Park, CA, September 1988.
- [Rush91] Rushby John, “Measures and Techniques for Software Quality Assurance”, Computer Science Laboratory, SRI international, Menlo Park, CA, September 1991.
- [Rush92] Rushby John, “Formal Methods for Dependable Real-Time Systems”, invited paper at *International Symposium on Real-Time Embedded Processing for Space Applications*, France, November 3-6, 1992.
- [Rush93a] Rushby John, Srivas Mandayam, “Using PVS to Prove Some Theorems of David Parnas”, Higher Order Logic Theorem Proving and Its Applications (*Proceedings of the 6th International Workshop, HUG’93*), No.780 Lecture Notes in Computer Science, pp.163-173, Vancouver, Canada, August 11-13, 1993.
- [Rush93b] Rushby John, “Formal Methods and the Certification of Critical Systems”, Technical Report CSL-93-7, Computer Science Laboratory, SRI international, Menlo Park, CA, December 1993.
- [Rush95] Rushby John, “Formal Methods and Their Role in the Certification of Critical Systems”, Technical Report CSL-95-1, Computer Science Laboratory, SRI international, Menlo Park, CA, March 1995.
- [Rush96] Rushby John, “Mechanized Formal Methods: Progress and Prospects”, *the 16th Conference on the Foundations of Software Technology and Theoretical Computer Science*, Hyderabad, India, December 1996, Springer-Verlag Lecture Notes in Computer Science, Vol.1180, pp.43-51.
- [Rush97] Rushby John, “Calculating with Requirements — Extended Abstract”, *Proceedings of the 3rd IEEE International Symposium on Requirements Engineering*, pp.144-146, IEEE Computer Society Press, Annapolis MD, January 1997.
- [Ryan93] Ryan Kevin, “The Role of Natural Language in Requirements Engineering”, *Proceedings of the IEEE International Symposium on Requirements Engineering*, pp. 240-242, San Diego, CA, January 4-6, 1993.

S

- [Sahn81] Sahni Sartaj, "Concepts in Discrete Mathematics", the Camelot Publishing Company, 1981.
- [Saie96] Saiedian Hossein et al, "An Invitation to Formal Methods", *IEEE Computer*, Vol.29, No.4, pp.16-30, April 1996.
- [Saye90] Sayet C., Pilaud E., "An Experience of a Critical Software Development", *IEEE Conference Proceeding, Fault tolerate computing: 20th International Symposium*, pp.36-45, 1990.
- [Schu67] Schultz Donald G., Melsa James L., "State Functions and Linear Control Systems", McGraw-Hill Book Company, 1967.
- [Shaw93] Shaw Alan W., "Logic Circuit Design", Saunders College Publishing, 1993.
- [Shen96] Shen H., Zucker J., Parnas D.L., "Table Transformation Tools: Why and How", *COMPASS'96, Proceedings of the Eleventh Annual Conference on Computer Assurance*, pp.3-11, June 17-21, 1996, Gaithersburg, Maryland, USA.
- [Sing95] Singpurwalla Nozer D., "The Failure Rate of Software: Does it exist ?", *IEEE Transactions on Reliability*, No.44, Vol.3, pp.463-469, September 1995.
- [Somm82] Sommerville I., "Software Engineering", Addison-Wesley Publishing Company, 1982.
- [Spiv88] Spivey J.M., "Understanding Z — A Specification Language and Its Formal Semantics", Cambridge University Press, 1988.
- [Sree96] Sreemani Tirumale, Atlee Joanne M., "Feasibility of Model Checking Software Requirements: A Case Study", *COMPASS'96, Proceedings of the Eleventh Annual Conference on Computer Assurance*, pp.77-88, June 17-21, 1996, Gaithersburg, Maryland, USA.
- [SRI97] <http://www.csl.sri.com/PVS/overview.html>, date of visit: November 26, 1997.
- [Stan77] Stanat Donard R., McAllister David F., "Discrete Mathematics in Computer Science", Prentice Hall Inc., 1977.
- [Stew97] Stewart James C., "An NRC Perspective on the Use of Commercial Off-the-Shelf Software", *Nuclear News*, pp.43-46, September 1997.

T

- [Taka85] Takahashi Muneo, Kamayachi Yuji, "An Empirical Study of A Model for Program Error Predication", *Proceedings of the 8th International Conference*

on Software Engineering, Imperial College, London, UK, August 20-30, 1985.

- [Tan94] Tan Yang Meng, “Formal Specification Techniques for Promoting Software Modularity, Enhancing Documentation, and Testing Specifications”, Ph.D. Dissertation, Dept. of Electric Engineering and Computer Science, M.I.T., 1994.
- [Taus77] Tausworthe Robert C., “Standardized Development of Computer Software”, Prentice-Hall, Inc., 1977.
- [Thad93] Thadani A.C., Perch R.L., “Consideration of Important Technical Issues for Advanced Light Water Reactors”, Proceedings of the 2nd ASME/JSME Joint Conference, San Francisco, CA, 1993.
- [Thor97] Thorn Tommy, “Programming Languages for Mobile Code”, *ACM Computing Surveys*, Vol.29, No.3, pp.215-239, September 1997.
- [Tse91] Tse T.H., Pong L., “An Examination of Requirements Specification Languages”, *Computer Journal*, Vol.34, No.2, pp.143-152, 1991.

U

- [UKMoD91] UK Ministry of Defense Interim Standard (DEF-STAN-0055), Issue 1: “The Procurement of Safety-Critical Software in Defense Equipment — Requirement and Guidance”, April 1991.
- [Urba84] Urban Joseph E., “Computer Languages”, pp.184-200 in [Vick84].
- [USNRC75] U.S. Nuclear Regulatory Commission, “Reactor Safety Study — An Assessment of Accident Risks in U.S. Commercial Nuclear Power Plants”, WASH-1400 (NUREG-75/014), Washington DC, 1975.
- [USNRC95] U.S. Nuclear Regulatory Commission, “High Integrity Software for Nuclear Power Plants—Candidate Guidelines, Technical Basis and Research Needs”, NUREG/CR-6263 (MTR 94W0000114), Main Report prepared by the MITRE Corporation, Washington DC, June 1995.
- [USNRC97] U.S. Nuclear Regulatory Commission, “Review Guidelines on Software Languages for Use in Nuclear Power Plant Safety Systems”, NUREG/CR-6463, Revision 1 from the internet: <http://www.nrc.gov/NRC/NUREGS/CR6463/>, date of visit: December 29, 1997.

V

- [VanS93] Van Schouwen A. John, Parnas David Lorge, Madey Jan, “Documentation of Requirements for Computer Systems”, *Proceedings of the IEEE International*

Symposium on Requirements Engineering, pp. 198-207, San Diego, CA, January 4-6, 1993.

- [Vick84] Vick C.R., Ramamoorthy C.V. (editors), "Handbook of Software Engineering", Van Nostrand Reinhold Company, 1984.
- [Viol95] Viola Mike, "Ontario Hydro's Experience with New Methods for Engineering Safety Critical Software", *14th International Conference on Computer Safety, Reliability and Security*, pp.283-298, Belgirate, Italy, October 1995.
- [Viol96] Viola Mike, "Ontario Hydro's Experience with New Methods for Engineering Safety Critical Software", slides from *COMPASS'96*.

W

- [Ward90] Ward Stephen A., Halstead Robert H. Jr., "Computation Structures", the MIT Press, 1990.
- [Webe93] Weber-Wulff Debora, "Selling Formal Methods to Industry", *FME'93 Industrial-Strength Formal Methods, Proceedings of First International Symposium of Formal Methods Europe* Odense, Denmark, pp.671-678, April 1993.
- [Wein96] Weinberg Henri B., "Correctness of Vehicle Control Systems: A Case Study", Thesis for Master of Science, Dept. of Electrical Engineering and Computer Science, M.I.T., February 1996.
- [Whit89] White Bebo, "The Comparison and Selection of Programming Languages for High Energy Physics Applications", *Computer Physics Communications*, 57 (1989), 538-542, North-Holland.
- [Whit90] White George M., "A Formal Method for Specifying Temporal Properties of the Multi-User Interface", *Multi-User Interfaces and Applications* edited by S. Gibbs and A.A. Verrijn-Stuart, pp.49-59, 1990.
- [Will94] Williams L, "Assessment of Safety-Critical Specifications", *IEEE Software*, pp.51-67, January 1994.
- [Wing90] Wing Jeannette M., "A Specifier's Introduction to Formal Methods", *IEEE Computer*, pp.8-24, September 1990.
- [Wing93] Wing Jeannette, Zaremski Amy Moorman, "Unintrusive Ways to Integrate Formal Specifications in Practice", *FME'93 Industrial-Strength Formal Methods, Proceedings of the First International Symposium of Formal Methods Europe* Odense, Denmark, pp.545-569, April 1993.
- [Work96] –, "Workshop on Tools for Tabular Notations", Communications Research Laboratory, McMaster University, Hamilton, Ontario, December 11-12, 1996.

X

- [Xu95] Xu Qiwen, He Weidong, "Hierarchical Design of a Chemical Concentration Control System", UNU/IIST Report No.41, International Institute for Software Technology, the United Nations University, October 1995.

Y

- [Yau95] Yau M., Guarro S., Apostolakis G., "Demonstration of Dynamic Flowgraph Methodology using the Titan II Space Launch Vehicle Digital Flight Control System", *Reliability Engineering and System Safety*, Vol.49, pp.335-353, 1995.
- [Yau97] Yau M., Guarro S., "Quantifying Software Reliability with the Dynamic Flow-Graph Methodology", *Transactions of the American Nuclear Society*, Vol.77, pp.261-262, Albuquerque, New Mexico, November 16-24, 1997.
- [Yau98] Yau M., Apostolakis G., Guarro S., "The Use of Prime Implicants in Dependability Analysis of Software Controlled Systems", accepted for publication in *Reliability Engineering and System Safety*, 1998.
- [Yen97] Yen John, Tiao W.A., "A Systematic Tradeoff Analysis for Conflicting Imprecise Requirements", *Proceedings of Third IEEE International Symposium on Requirements Engineering*, Annapolis, Maryland, USA, pp.87-96, January 6-10, 1997.
- [Yovi93] Yovits Marshall C. (Editor), "Advances in COMPUTERS", Vol.36, Academic Press Inc., 1993.
- [Yu97] Yu Eric S.K., "Towards Modelling and Reasoning Support for Early-Phase Requirements Engineering", *Proceedings of Third IEEE International Symposium on Requirements Engineering*, Annapolis, Maryland, USA, pp.226-235, January 6-10, 1997.

Z

- [Zave95] Zave Pamela, "Classification of Research Efforts in Requirements Engineering", *Proceedings of the 2nd IEEE International Symposium on Requirements Engineering*, pp.214-216, York, England, March 27-29, 1995.
- [Zio] Zio E., Apostolakis G., "The Use of the Analytic Hierarchy Process for the Assessment of Expert Judgments Credibility", undated.

Appendix 1

Differences between Digital Systems and Analog/Mechanical Systems

With the introduction of digital technology to I&C of nuclear power plants, new concerns arise. Digital systems are essentially different from analog counterparts in a number of ways. Here is a list of the major discrepancies between them.

A1.1 Different Complexities

Compared with digital systems, analog/mechanical systems are relatively simple, well studied, and well understood. As a result, “exhaustive output versus input functional testing and cycle-to-failure testing are commonly used” in mechanical systems^[NRC95]. However, such exhaustive testing for a piece of application size software is usually impractical, if not impossible. Davis presents an estimation of the possible number of paths of an unstructured piece of software, and it turns out that a program as small as 100 lines of code can have up to 10^{158} possible paths^[Davi93].

A1.1.1 Software Complexity Measures

Two of the most commonly used complexity measures of software are the Halstead’s Metric and the McCabe Metric. First introduced in [Hals77], Halstead’s program length (called Halstead Length in [Beiz90]) is defined as

$$H = n_1 \cdot \log_2 n_1 + n_2 \cdot \log_2 n_2 \quad (\text{A1-1})$$

where

H is the Halstead’s metric or Halstead Length, not lines of code (LOC),

n_1 is the number of distinct operators (e.g., keywords) in the program. Paired operators (e.g., IF...THEN...ELSE, BEGIN...END, DO... UNTIL) should be counted as one operator. An n -branch case statement should be counted as ($n-1$), and

n_2 is the number of distinct operands (e.g., variables, data objects) in the program.

In the 1970s, software companies had difficulty determining how to modularize the software they were developing. Many of them had modularized their software based on the length-of-code metric, which was irrational since some programs are logically more complex than others of the same or even larger length. Based on graph theory, McCabe defined an alternative complexity measure^[McCa76] of a graph $G(V, E)$ as

$$V(G) = e - n + 2p \quad (\text{A1-2})$$

where

$V(G)$ is the McCabe's Cyclomatic Complexity Measure of a graph $G(V, E)$,
 n is the number of vertices in $G(V, E)$,
 e is the number of edges in $G(V, E)$, and
 p is the number of connected components in $G(V, E)$.

For a strongly connected graph, McCabe's Cyclomatic Complexity Measure equals to the maximum number of *linearly independent* paths (e.g., paths not obtainable by linear combination of other paths) in the graph.

A1.1.2 Software Maintainability Index

Software Maintainability Index (SMI)^[Oman92] reflects the degree of how easy a program is maintained and is defined as follows.

$$SMI = 171 - 3.42 \times \ln(\text{ave}E) - 0.23 \times \text{ave}V(G) - 16.2 \times \ln(\text{ave}LOC) \quad (\text{A1-3})$$

where

$aveE$ is the average Halstead effort/module,

$aveV(G)$ is the average extended McCabe cyclomatic complexity/module (a measure of the number of paths through source code) and

$aveLOC$ is the average LOC/module.

Upper bounded by 171, the SMI of a piece of software will decrease as the software becomes more and more complex, e.g., $aveE$, $aveV(G)$, $aveLOC$ increasing. Therefore, the smaller the SMI of software, the more difficult its maintenance. Based on experience from Hewlett-Packard ^[Cole92], any software with an SMI less 65 is considered difficult to maintain.

INEL ^[INEL95] and many other studies show the negative correlation between software functionality and SMI, e.g., as more and more functionality is added to the existing software to meet the growing functional requirements under new application, software maintainability becomes worse and worse. Figure A1-1 shows, as more and more functionality added, an accelerated decline in maintainability index is observed in software written in FORTRAN and C while similar deterioration is not noticed in software programmed in Ada.

A1.2 Different Behaviors and Mathematical Modeling

Most traditional, physical systems can be described using differential equations under reasonable and defensible assumptions. The outputs are usually continuous functions of their input parameters, and state transitions are usually smooth, e.g., the enthalpy of liquid water at a specified pressure is a continuous function of its temperature, hence we can interpolate (in some cases, even extrapolate) the unknown enthalpy at a certain temperature from the known enthalpies of water at other temperatures. By contrast, the outputs from the digital systems are discretized functions of inputs and transitions are usually abrupt. As a result, we can not expect the properties of an unknown case, e.g., output of a code or command from a digital

control system, from the known cases by interpolation or extrapolation. Hence, instead of differential equations, discrete mathematics is used to describe the digital systems.

A1.3 Different Data Manipulation Processes

For an analog system, data are collected continuously and each measurement, function, or command is transmitted continuously over dedicated metal wire. By contrast, data are collected either periodically or via event-driven discrete sampling, and multiple measurements, functions, or commands are transmitted over shared metal wire or optical fiber for a digital system.

A1.4 Different Production Processes

When mechanical device is built, human creativity is ultimately transformed into a physical entity. Modern manufacturing is in the form of mass production. This process generates enough samples for applying well developed statistical methods to understand, describe, predict, and update the properties (e.g., mean lifetime, standard deviation, failure rate) of the products using Bayes Theorem.

Software, on the other hand, is a logical and mental product, rather than a physical entity produced by the developer. Compared with mechanical hardware which is usually assembled from standard industry parts, no such standard part warehouse exists for software. Each software is a unique, (usually) manually developed intellectual property rather than from the mass production lines unless an exact copy of some existing software is made. Hence, the statistical methods do not work properly.

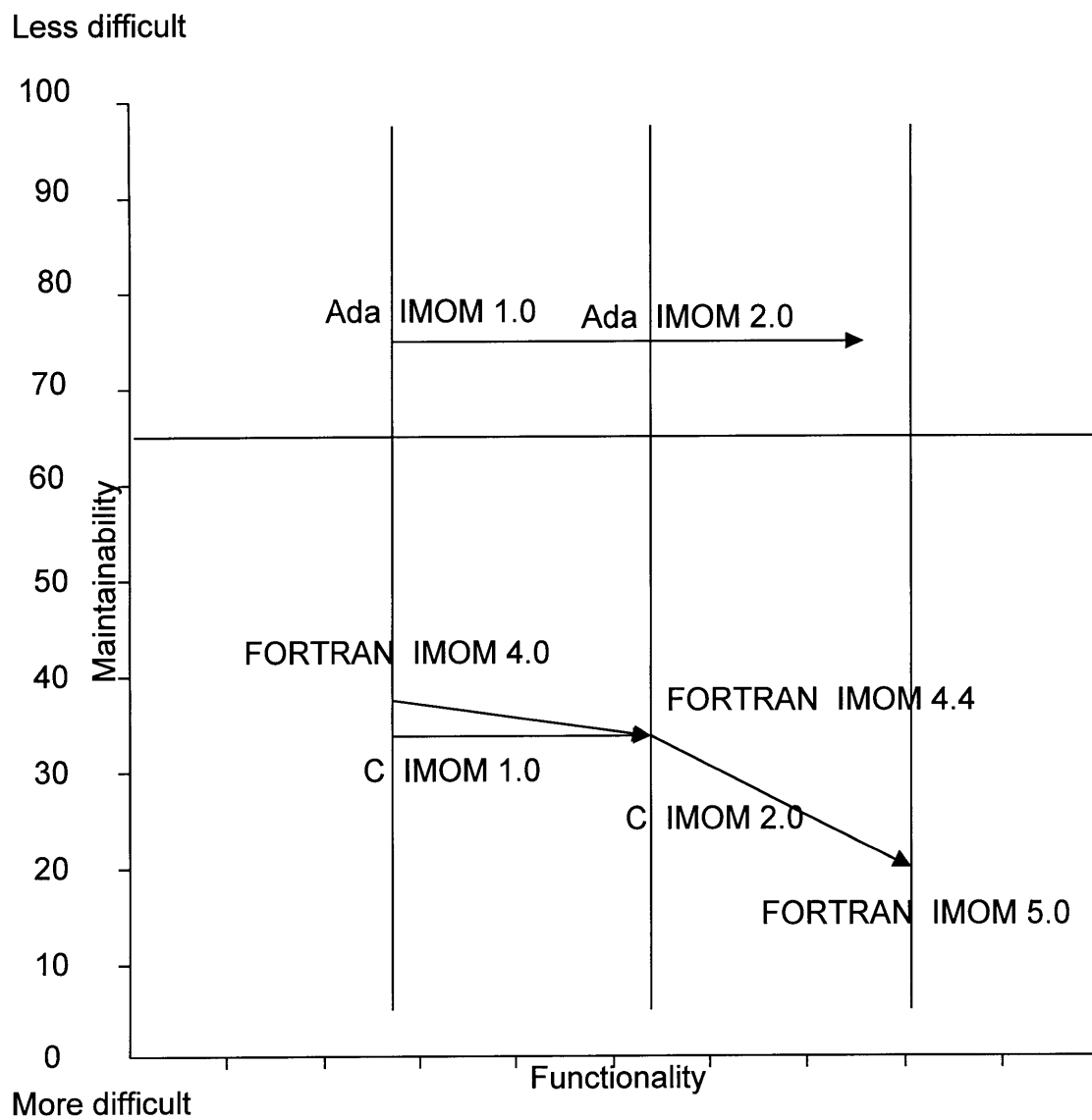


Figure A1-1 Relationship between SMI and Maintainability (from [INEL95])

A1.5 Different Failure Modes

Analog systems unavoidably experience aging and failure mode is usually random failure. Analog systems are affected by physical environment, e.g., temperature, pressure, moisture, smoke, radiation. During their life time, analog components experience “bathtub-shaped” failure curve with high infancy failure rate due to manufacturing flaws and high “end-of-life” failure rate due to aging (Figure A1-2 (A)). Traditional statistical method works well in estimating random failure rate, e.g., through measuring mean-time-to-failure (MTTF) analysis, fault tree analysis and/or event tree analysis. On the other hand, software does not experience aging¹² at all. Software failures are fundamentally human-made design errors. Software does not exhibit such bathtub behavior, although many researchers tend to agree that the initial failure rate is higher for software, then the failure rate decreases as the system is debugged (Figure A1-2 (B)). The spikes in the actual curve represent the fact that new faults are introduced when software is modified. As a rule of thumb, on average, a new error is introduced while correcting every six errors. Unfortunately, there are no universally accepted methods to quantify the failure rate. Even worse, it is still controversial within the software engineering community whether software fails randomly, whether its failure rate exists.^{[NRC97], [Sing95]} Many researchers tend to believe that software is either correct (i.e., satisfying its specification which could be inappropriate or incomplete) or not correct. The point is that we do not know, in advance, whether the software is correct or not with 100% confidence.

The software system crashes by the presence of malicious virus while the mechanical system does not have such failure mode.

A1.6 Different Failure Manifestation Patterns

Failures in mechanical system can be easily detected, e.g., failure of a pump can be detected immediately if redundant sensors experience sudden reduction in flowrate simultaneously.

¹² In [Parn94a], Parnas discusses software “aging” under different context, i.e., software fails to meet its changing environment, or obsolescence.

Meanwhile, software error does not manifest itself until the very portion of the software containing the error is executed. Therefore, software errors are more difficult to be detected or revealed.

A1.7 Different Consequences

Mechanical failures is usually local with limited consequences^[Glas80]. However, failure of a software system for control purpose can cripple the entire system. Bell reports that, during a U.S. space program in which an unmanned vehicle was sent to study the planet Venus, the vehicle turned to a wrong course and was lost forever simply due to a tiny coding error in a Fortran-“do” statement^[Bell87]: DO 3 I = 1.3 should have been DO 3 I = 1,3.

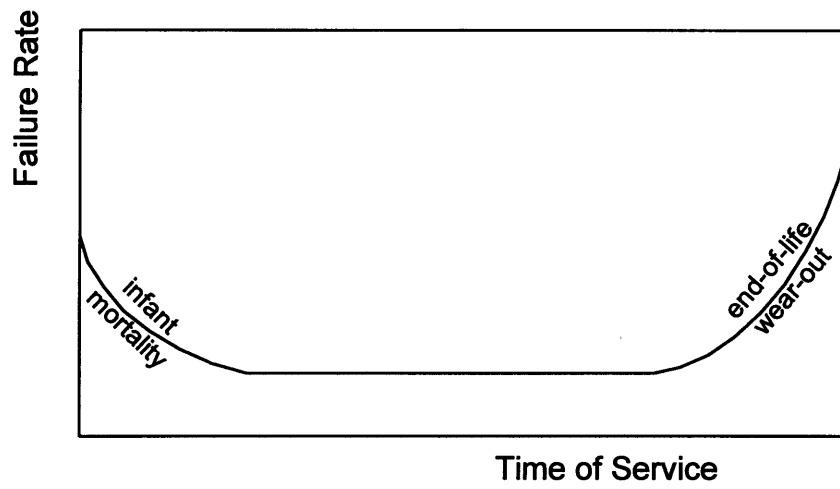
Leveson reports that the software errors caused Therac-25 accident^[Leve95] which overdosed six people through the computer-controlled radiation therapy between June 1985 and January 1987. Software errors in the telephone switch systems disabled half of the continental U.S. telecommunication network for hours. As a result, software errors are more costly to correct. For an example, the notorious Year 2000 problem will cost \$600 billion, half of which will be spent by the U.S. government and private companies^[Cong97]. (Year 2000 problem is a software bug that can not distinguish Year 2000 from Year 1900 and will therefore cause software chaos in Year 2000.)

A1.8 Different Weightings in Cost Structure

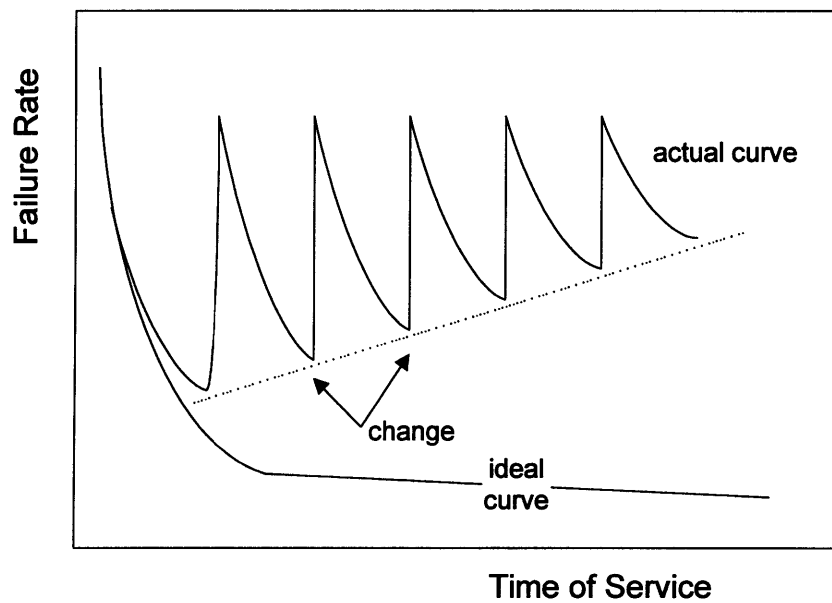
In the 50’s and early 60’s, software accounted for 10-20% of the total budget (i.e., 90-80% for hardware). In the late 80’s, The cost structure reversed with software accounting for about 80-90% of the budget^[Fran83]. The trend of increasing software cost in budget is believed to continue.

A1.9 Different Destinies

When an analog device fails, it is replaced by a spare part that satisfies the same technical specification. For software, however, there are no such corresponding “spare parts”. Although some efforts had been tried to develop such inventory, results turned out to be disappointing. If software is found to be faulty, it is usually debugged and corrected rather than abandoned.



(A) An Analog System



(B) A Digital System

Figure A1-2 Failure Rates in Analog and Digital Systems (after [Pres87])

Appendix 2

A Survey on Software Development Cycle Models

The primary functions of a software process model are to determine the order of the stages (or phases) in software development and to establish the transition criteria for progressing from one stage to the next. These include completion criteria for the current stage as well as choice criteria and entrance criteria for the next stage.

A2.1 The Code-and-Fix Model

The Code-and-Fix Model was used in the earliest days of software development and it contained two basic steps:

- write a code;
- fix the problems in the code.

The order of work in this model is coding first, followed by thinking about the requirements, specification, design, testing, and maintenance. There are several problems with this naïve model:

- This model does not reflect the fact that coding is usually preceded by activities such as requirements, specification, design;
- After a number of fixes, the code usually becomes so ill-structured that subsequent modifications are very difficult as well as expensive, implying the need for design prior to coding;
- Very often, even well-designed software deviates what the customer is expecting, implying a requirements/specification phase prior to design;
- Code is expensive to fix due to poor preparation for testing and modification;

- Acceptance testing is difficult without explicit requirements documentation.

A2.2 The Waterfall Model

In [Boeh88], Boehm reports that, as early as 1956, experience on large scale software project such as Semi-Automated Ground Environment (SAGE) led to the development of a stagewise model, which divides software development project into several successive stages, e.g., operational plan, operational specification, coding specifications, coding, parameter testing, assembly testing, system evaluation.

As a refinement model of the stagewise model in the 1970s, the Waterfall Model provided two primary enhancements over its predecessor.

- Recognition and addition of feedback between stages/phases, and a guideline to confine the feedback to minimize the costly rework involved in feedback across many stages;
- An initial incorporation of prototyping in the software life cycle, via a “build it twice” step running in parallel with requirements analysis and design.

A typical Waterfall Model of software development process consists of seven distinct stages: planning activities, software requirements analysis, software design, software implementation, software verification and validation (V&V, e.g., testing), software installation, and operation and maintenance (Figure 1.3-1). It should be noted that feedback exists between stages. Once errors are detected originating from a previous stage, it must be fed back to that stage such that modifications can be made.

A2.2.1 Planning Activities

Work begins with establishing requirements at the system level and allocating some subset of these requirements to software. The managerial decision and system view is essential when

software must interface with other elements of the system, i.e., hardware, humanware, and databases.

A2.2.2 Software Requirements Analysis

The requirements are collected specifically for the software. The purpose of this stage is to analyze customer's requirements defined for the software such that the requirements are consistent, complete, and unambiguous. The product of this stage is software requirements specification (SRS). The SRS is the basis for software V&V and should be reviewed by the customer.

A2.2.3 Software Design

Software design is a multi-step process, e.g., the preliminary design and the detailed design. The purpose of this stage is to translate SRS into a representation of the software that can be assessed for quality before actual coding begins and to divide the complex problem into functional less complex modules. Like the SRS, the design is documented and becomes part of the software configuration.

A2.2.4. Software Implementation (or Coding)

This stage translates design documents into computer readable code in programming languages. The code is a prime product of the project. If design is performed in a detailed manner, coding can be accomplished automatically using tools. However, the majority of coding currently implemented are generated by human programmer.

A2.2.5 Software Testing

As the most widely used software V&V method in industry, software testing is a “process of executing a program on a set of test cases and comparing the actual results with the expected results” defined by software requirements specification ([Lisk86]). The purpose of testing is to detect or reveal the existence of errors. Usually two kinds of testing are performed, unit

testing to reveal any errors at module level and integration testing to reveal errors of the software as a whole.

A2.2.6 Software Installation

Installation is a process to mount the software developed onto the target hardware frame. At this stage, an acceptance testing is usually required by the customer.

A2.2.7 Operation and Maintenance

Operation of software will usually detect errors not revealed by the testing and software will undoubtedly undergo changes after it is delivered to the customer from the developer. The changes may be due to changes in the external environment (e.g., context) or the customer's new needs on functional or performance enhancement.

As a document-driven approach, the Waterfall Model has become the basis for many software acquisition standards, and is continuing to be the most widely used model in software engineering. The Waterfall Model emphasizes on fully elaborated documents as completion of criteria for early requirements and design phases. Complaints from the cons of this model include that the model does not work well for many classes of software, e.g., interactive end-user applications and, in areas supported by fourth-generation languages (spreadsheet or small business application), it is clearly not necessary to write elaborate specification for one's application before implementing it^[Boeh88]. But it is strongly believed that, for safety- and mission- critical application, the Waterfall Model is the most effective model to proceed.

A2.3 The Evolutionary Development Model

In the Evolutionary Development Model, software development stages consist of expanding increments of an operational software product, with the directions of evolution being determined by operational experience. This model is most suitable for fourth-generation language application. The problems with this model are obvious.

- This model has the flavor of the Code-and-Fix Model, e.g., code-driven;
- It is based on the unrealistic assumption that the user's operational system will be flexible enough to accommodate unplanned changes.

A2.4 The Transform Model

The Transform Model assumes automatic transforming a formal specification into a software product satisfying the specification. The steps of this model include

- a formal specification developed to reflect the best initial understanding of the desired product;
- automatic transformation of the formal specification into code;
- when necessary, an iterative loop is developed to improve the performance of the code through optimizing the transformation;
- exercise the software product;
- an outer iterative loop to adjust the specification based on operational experience, and to reoptimize, exercise the software product.

Through modifying and optimizing the specification, this model bypasses the dilemma of making the code unstructured and eliminates additional rework in the intermediate stages such as design. The downsides of the model are

- a formal specification must be in existence at the beginning of applying the model;
- a formal specification also implies extra training in formal language is required;
- automatic transformation capabilities (e.g., automatic tools) must be available to transform specification into code. In practice, such tools are not always satisfactory;
- like the Evolutionary Development Model, customer's operational system must be flexible enough to support unplanned modification.

A2.5 The Spiral Model

As a risk-driven approach, the Spiral Model is more universal and can accommodate most previous models as special cases^[Boeh88, p.69]. Pictorially, the Spiral Model resembles a polar coordinate system, whose radial dimension represents the cumulative cost in accomplishing the steps to date, whose angular dimension represents the progress made in completing each cycle of the spiral. As its name implies, the Spiral Model is comprised of several cycles, reflecting the underlying assumption that each cycle involves a progression of same sequence of steps.

A2.5.1 A Typical Cycle of the Spiral

A typical cycle of the spiral begins with the identification of the objectives of the product being elaborated, the alternative means of implementing the objective (e.g., buy or reuse), and constraints imposed on the alternatives.

The next step is to evaluate the alternatives relative to the objectives and constraints by means of identifying uncertainty/risk, formulating a cost-effective strategy for resolving the risk.

When the risks are evaluated, the relative remaining risks determine the following step, which can be the Evolutionary Development Model if user-interface dominates, or the Waterfall Model if program development or interface-control risks dominate.

Similar to most other models, each cycle in the Spiral Model is completed by a review involving the people or organizations interested in the product.

A2.5.2 Initiating and Terminating the Spiral

In both software development and maintenance, the spiral is triggered by a hypothesis that a particular mission can be improved by software. Then, the spiral process starts a test of the

hypothesis. If the test fails (e.g., a superior commercial product available), the spiral terminates. Otherwise, it terminates with the installation of new or modified software. Usually, experience with the operational mission leads to further hypothesis about software improvements, a new hypothesis test to form iterations.

A2.5.3 Advantages and Disadvantages of the Spiral Model

The advantages of the Spiral Model includes the following

- it pays early attention to the options involving evaluation of alternatives, e.g., reuse of existing software;
- it emphasizes on eliminating errors and unattractive alternatives early;
- it takes into account life cycle evolution, changes of software product;
- it incorporates software development and software maintenance into one approach;

The Spiral Model is not a perfect model yet in software engineering. The problems identifies with the model include

- many issues must be addressed and resolved before it becomes a mature, universal model;
- the model currently works well on internal software development, further work is required to make it match the world of contract software acquisition;
- unlike conventional, document-driven approach, the Spiral Model requires risk-assessment expertise;
- the Spiral Model is not yet as fully elaborated as other models.

Appendix 3

Recommended Practices to Improve Software Quality

A3.1 Testing

As the most widely used software verification and validation (V&V) technique in industry, *testing* is defined as a dynamic process of executing a program on a set of test cases and comparing the actual results with the expected results. Testing is used to show that the functional requirements of the code has been satisfied and is used to detect the errors in the code (but unlike *debugging*, testing does not need to pinpoint or correct the errors). To convince an independent party by testing alone, we must show a software works on the set of inputs exhaustively by running each input once. However, the *exhaustive testing* works effectively only if the input space is small, e.g., small and simple codes. For large and complex software, this approach is extremely expensive and sometimes impossible.

A3.1.1 Black Box Testing and White Box Testing

According to how test cases are generated, there are two types of testing techniques: *Black Box Testing* and *White Box Testing*. In Black Box Testing (which is sometimes referred to as *Functional Testing*), test cases are generated by simply studying the *specification alone* regardless of the internal structure of implementation of the software being tested. The general procedure of Black Box Testing is studying the specification, preparing the test data and desired output, running the software, and comparing software output with desired one. The advantages of this testing strategy include:

- Black Box Testing can be done and test results can be interpreted by people unfamiliar with the internal structures of the software being tested;

- Black Box Testing is robust with regard to changes in the implementation of the software since it does not care about the implementation detail in the software.

However, Block Box Testing can not tell how much coverage of paths in the software one will get from a set of blackly-generated test cases. As a supplemental means to test a program, *White Box Testing* (also known as *Glass Box Testing* or *Structural Testing*) is designed to test different paths in the program by simply studying the internal structure of the *source program*. The purpose of this type of testing is to generate a path-complete test set that covers each path at least once to ensure that all paths are implemented correctly. However,

- path-completeness in White Box Testing is not a sufficient condition to detect all the errors. As a counterexample, consider the following code in written in programming language C.

```
/*
the function is supposed to find the minimum
of three integers
*/
int minimum_of_three (int x, int y, int z)
{
    if (x>y) return z;
    if (x<=y) return z;
}
```

If the test cases happen to be the triples of (3, 2, 1) and (2, 3, 1) for (x, y, z), which are path-complete for the function, use of the test data will mislead us to believing that the function is correct when it is actually wrong. The incorrectness will be identified if an alternative triple of (1, 2, 3) is used for (x, y, z);

- White Box Testing focuses on the internal structure of a software, e.g., what is implemented in the software or what the “software is actually implemented to do”, rather than what the “software is supposed to do”;
- based on path-complete testing strategy, White Box Testing might need an exponential number of test cases even for a small program (see Section 1.2.1).

A3.1.2 Unit Testing, Integration Testing, System Testing and Acceptance Testing

According to the phases that tests are done during software development cycle, there are four kinds of tests: *Unit Testing*, *Integration Testing* and *Acceptance Testing* (or *Installation Testing*).

Unit Testing is to execute a module and is used to detect errors at the module level. The purpose of Unit Testing is to ensure that the modules behave as expected individually. Integration Testing is used to detect errors in the program as a whole when all modules are integrated. Unit Testing is usually much easier than Integration Testing since there usually exists well-defined specification for the modules. System Testing is carried out once Integration Testing is passed. The purpose of System Testing is to detect discrepancies in the interaction between hardware and software by exercising the entire system against the SRS and the system specification. It is the first time the entire system is tested against the software requirements. Acceptance Testing is a form of demonstrative testing by mounting the software onto the customer’s facility. The purpose of the testing is to ensure the system can be installed, properly configured, and executed in the customer’s environment. The testing includes user documentation such as user guides, reference manuals.

A3.1.3 Boundary Condition Testing

Boundary Condition Testing (or *Stress Testing*) chooses test data lying along the “boundaries” or extreme values of input domain, e.g., maximum or minimum values of the variables, upper and lower indices of an array, trivial cases of data structures such as empty

tree or linked list with zero or one element. The purpose of Boundary Condition Testing is to ensure that even out-of-range data do not result in unintended behaviors.

A3.1.4 Drawbacks of Testing

Even though testing is currently the most widely used software V&V method, it is by no means a perfect method since

- testing requires execution of the software, so it can only occur at the late stage of software development cycle;
- testing can only reveal the errors in the software being tested, it can not claim that the software is error-free without exhaustive testing (which is not always feasible);
- one can not extrapolate from a test case to another, e.g., tester can only claim that the software is correct with regard to its specification under the tested cases, he can not make any assertion to the untested cases;
- testing can never reveal errors such as extraneous segment of code (see Appendix 8);
- testing is usually very expensive. As a rule of thumb, testing takes as high as 50% of software budget (Table A3-1).

Table A3-1 Cost of Software Testing (% of software budget unless otherwise stated)

Project Name	Cost of Testing
Sage ^[Deut82]	47%
Naval Tactical Data System ^[Deut82]	50%
Gemini ^[Deut82]	47%
Saturn I ^[Deut82]	44%
Apollo Project ^[Deut82]	~80% of project cost
Darlington Nuclear Generation Station ^[Viol95]	20%

A3.2 Design Diversity

Design Diversity refers to use of separate design/implementation teams to produce multiple versions of software from the same specification. Via threshold voters (rather than exact-match voters), it is expected to mask out the effect of design errors in one of the versions while tolerating minor variations among different versions. The underlying assumption of the method is that different teams will deliver errors independently so that the probability of delivering same error by different teams at the same time is extremely small. Unfortunately, the independence assumption has been rejected at 99% confidence level in several experiments^[Knig86]. Studies find that developers are inclined to make similar mistakes. Hence, *Design Diversity* tends to create a deceptive “illusion” of ultrahigh reliability without actually achieving it. Typical design diversity techniques are software fault-tolerance, N-version programming.

A3.3 Commercial Off-the-Shelf (COTS) Software

Current policy in the US Department of Defense (DOD) requires the use of commercial off-the-shelf (COTS) software products that can satisfy DOD needs([INEL95]). Generally speaking, COTS software products are more cost-effective, easier to maintain and upgrade, less machine-dependent, and not installed on a “non-open” hardware platform. As the maintainability of the software packages increases, their support cost decreases, and the reliability of the system improves. However, as Stewart points out, one of the downsides in using COTS software is that software vendors will not sell their source code to the utility, making software essentially a black box to the utility. Another downside is that the volatility in the software industry due to fierce competition will inevitably push some commercial vendors out of business, making it difficult to trace back to the vendors if software problems are detected sometime in the future^[Stew97].

A3.4 Reusable Software Modules

One concept associated with the structured programming is to deal with complexity of large software systems in terms of modularity. The complexity of a complex software system can be controlled if the system is decomposed into small, understandable, and manageable modules, each of which accomplishes a particular task. As recommended in [Urba84], the lengths of these modules should be limited to one or two pages of high-level language code.

A3.4.1 Benefits from Reusable Modules

The benefits of modularity is three-fold. First, it reduces the complexity through “divide-and-conquer” technique, that is, splitting a complex problem into functionally related, smaller, and simpler subproblems. Secondly, the “divide-and-conquer” technique makes it possible to reuse existing modules. Reusable software modules tend to be smaller and less complex since each of these modules accomplishes a specific, well-defined task. Hence, the implementation is easy to verify, e.g., through formal verification or possibly exhaustive unit testing. Once they are verified, they can be invoked by other modules wherever their specifications are satisfied. The process reduces the effort of integration testing and eliminates the possibility of introducing potential new errors in developing new software. Together with other techniques, reusable software modules save the baseline Improved Many-On-Many (IMOM) code by 36%^[INEL95]. Third, reusable software modules will help reduce “time to market”, reduce software costs, increase productivity, and improve software reliability. INEL claims an saving of 10 times of manpower using Reusable Software Modules in their IMOM reengineering project. They spent 5-6 man months to implement each additional IMOM model using Ada. In contrast, it is estimated that more than 70 man months would have been needed if each model had been reengineered from the scratch. However, one should bear in mind that, “if the existing software is a commercial product, the system developer is not likely to be able to obtain a guarantee of vendor support, over a long period of time, for a particular version” of the software^[IAEA94].

A3.4.2 Guidelines on Software Modularization

As measures of maintainability and modifiability, *coupling* and *cohesion* provide means of measuring the degree of mutual dependence of modules in software. Coupling is a measure to indicate the degree of interaction between modules while cohesion measures the interactions within a module. The essence of modularization is to divide program in such a way that there is a minimum of interaction between modules (low coupling) and a high degree of interaction within a module (high cohesion). The former has a direct impact on maintenance since more coupling implies more dependency between modules. Therefore, the higher the degree of coupling, the more difficult it is to modify a module without the knowledge of or making corresponding changes to others. Table A3-2 indicates the relationship between the types of couplings and their measures. The larger the value, the stronger the coupling. The scope of cohesiveness ranges from accidental (low end), logical, temporal, communicative, sequential, to functional (high end). For example, functional cohesiveness represents that all the components of a module are required to perform the single function of that module.

Table A3-2 Coupling and Software Maintenance (form [Lano94])

Coupling Types	Typical Values	Descriptions
Content	0.95	Direct reference from one module to the contents of another
Common	0.7	Reference to a shared global data structure (e.g., COMMON blocks in FORTRAN)
External	0.6	Reference to named external data items
Control	0.5	Use of parameters dedicating the control flow in the called module
Stamp	0.35	Reference to a shared data structure passed as a parameter
Data	0.2	All references are to local data or to individual data values passed as parameters

A3.5 Object-Oriented Programming (OOP)

Object-oriented paradigm was first introduced in the late 1960s with the language Simula67. In the 1970s, it was further developed with the language Smalltalk-80^[Davi93]. In contrast to the traditional procedure-oriented design which focuses on the decomposition of the software into various functional components (e.g., structured or procedural programming techniques which focus on functions rather than data), the object-oriented design (OOD) is currently a prevalent software design technique that places emphasis on organizing software into discrete objects or data-types (or data type hierarchies) that contain both the states (i.e., variables or parameters) and operations (i.e., method). Object-oriented programming (OOP) is the implementation of the object-oriented design.

A3.5.1 Characteristics of Object-Oriented Paradigm

The object-oriented paradigm is characterized by the properties of data abstraction, encapsulation, inheritance, and polymorphism.

Data Abstraction refers to the ability to build a data structure to define and use an object within a program without having to pay attention to its internal implementation details. Data abstraction simplifies the programmer's task by reducing the level of details to be dealt with at a given time. As a result, it facilitates team work and cooperation in software development. As a means of structuring software, *Data Encapsulation* (or *Information Hiding*) provides the enforcement necessary to make sure that only functions (or methods) immediately associated with the data structure are allowed to access its internal details. Therefore, it requires that, for each object, its data structure and operations (i.e., how to access and modify the structure) be part of a single module (usually called a class). Each class will only be allowed access to objects which are required to implement its function. Access to other objects, not needed by the class, are prohibited by using the scope rules to conceal these objects. In C++, for example, there are three types of data encapsulation: private, protected, and public members. From private to protected to public members, the level of security is decreasing. *Inheritance* allows the programmer to describe new objects (i.e., derived class or subclass) in terms of an

existing objects (known as base class or superclass). It is a means of sharing data and functions among classes based on a hierarchical relationship. This serves two purposes. First, it saves effort because it prevents repeating same task multiple times. Second, inheritance relationship between subclass and base class is inexpressible in purely structured languages. *Polymorphism* refers to a means same function may behave differently on different objects. Polymorphism works by binding methods or functions of objects at run time rather than at compile time.

A3.5.2 Benefits of OOP

The idea will achieve four main goals:

- **Changeability**

If design decisions are changed, consequent changes are limited locally, e.g., a few classes (preferably one).

- **Independent Development**

Industry-size software is being developed by a group of programmers, interface between the modules should be as simple as possible, and interface should be by means of calls on subroutines/functions rather than by means of shared data and file structure. Hence, there is no way that the hidden information may be corrupted by a module which is not supposed to deal with that information.

- **Comprehensibility**

To the ease of testing and maintenance, it facilitates understanding individual modules independently of others.

- **Code Sharing and Reusability**

OOP makes it possible and easy to share and reuse code between classes, e.g., via inheritance.

A3.6 Fault Avoidance

By contrast to fault removal, the idea of fault avoidance is to impose more stringent control on the software development process such that errors are not introduced into the product. One of the fault avoidance techniques is the use of formal methods. More detailed discussion on formal methods will be in Section 2.1 as well as Appendix 4.

Appendix 4

A Survey on Formal Methods

The purpose of presenting the survey on formal methods here, rather than in Chapter 2, is due to the fact that Chapter 2 is more focused on Software Requirements Specification (SRS).

A4.1 Vienna Development Method (VDM)

As one of the most widely used formal methods, Vienna Development Method (VDM) was developed by Dines Bjørner, Cliff Jones, and co-workers at the IBM Vienna Laboratory during the 1970s ([Blo86]). VDM is based on denotational semantics, in which each objects denotes, directly or indirectly, a mathematical object already understood. VDM makes use of discrete mathematics (e.g., set theory, predicate calculus) and has three components^[Harr96]:

- a notation to express software specification;
- an inference system to construct proofs of correctness;
- a methodological framework to develop software from a specification in a formally verifiable way through a process know as “reification”.

A4.1.1 Components of a VDM Specification

According to [Jone90b], the VDM specification of a program includes two major parts: *states* and *operations*. A state defines a class or set of valid states, e.g., data object. Operations are specified in the form of single-state *pre-condition* predicates (i.e., predicates on a single initial state) and two-state *post-condition* predicates (i.e., predicates on both the initial and final states). This type of specification is called implicit specification in that it intends to describe the properties required of a program without specifying the details how they are achieved. A typical format of Implicit Function Specification in VDM (Figure A4-1) has three sections: a *type clause*, a *pre-condition*, and a *post-condition*. The type clause defines

the data types of the domain variables and the range variables. The pre-condition restricts the domain of the arguments and indicates what must be true for the result of an operation to be defined before the operation is taken while the post-condition relates the domain values and the range values and asserts what must be true after the operation has completed. The post-condition holds only if the pre-condition is satisfied. Otherwise, the result of the operation is undefined. By convention, a `true` pre-condition is usually omitted. In Figure A4-1, D and R represent the data types of the domain and range (e.g., real, boolean, etc.) of a function respectively, d and r represent domain and range arguments respectively. For a complete set of syntax of VDM specification, readers are recommended to study Appendix F of [Jone90a] or Chapter 5 of [Harr96].

A4.1.2 The Weakest Pre-Condition

Working backwards, the general concept of the weakest pre-condition¹³ is to find the least restrictive pre-condition such that the post-condition holds true and the weakest pre-condition is the consequence of the pre-condition in the problem. The advantage of working backwards is that the reasoning is goal-directed. At each step, knowing that we wish to be true at the conclusion of some part of the program, we compute what exactly needs to be true prior to that part. When we reach the beginning of the program, we end up with the weakest pre-condition sufficient to imply the desired post-condition. Reasoning is completed by showing that the given pre-condition implies the weakest pre-condition. As a result, it is concluded that the given pre-condition implies the given post-condition. In other words, the program satisfies its specification. Interested readers should refer to [Cohe90] and [Gann94] for more profound discussions.

¹³ Pre-condition is the “weakest” in the sense that any other sufficient pre-condition implies it ([List86]). One can also work forward and reach the “strongest” post-condition which implies the given post-condition.

Three-part specification:

function: $D \rightarrow R$ (type clause)

pre-function $(d) \stackrel{\Delta}{=} \dots$ (pre-condition)

post-function $(d, r) \stackrel{\Delta}{=} \dots$ (post-condition)

where

pre-function: $D \rightarrow Bool$

post-function: $D, R \rightarrow Bool$

or equivalently

$d \in D \wedge \text{pre-function}(d) \Rightarrow \text{post-function}(d, \text{function}(d))$

Figure A4-1 Implicit Function Specification (after [Jone80])

A4.1.3 An Example in VDM

The specification in DVM notation for the average value of three real numbers is given, together with an implementation of the specification. The purpose of the example is to use the Weakest Pre-Condition Approach to prove or disprove that the implementation satisfies the specification.

$$\begin{aligned}\text{avg}: R \times R \times R &\rightarrow R \\ \text{pre-avg}(a, b, c) &\stackrel{\Delta}{=} \text{true} \\ \text{post-avg}(a, b, c, r) &\stackrel{\Delta}{=} 3*r = a + b + c\end{aligned}$$

Disprove that the following Program P satisfies the specification.

```
r: = a+1;
r: = r+b;
r: = r+c;
r: = r/3;
```

It is equivalent to prove the following logic is not true.

$$\begin{aligned}&\text{pre-avg}(a, b, c) \Rightarrow \text{post-avg}(a, b, c, r) \\ \text{where} \\ &r = \text{Program P} \\ \text{or} \\ &\text{true} \Rightarrow \text{post-avg}(a, b, c, P) \\ &\Leftrightarrow \text{post-avg}(a, b, c, (r:=a+1; r:=r+b; r:=r+c; r:=r/3;)) \\ &\Leftrightarrow [(r:=a+1; r:=r+b; r:=r+c; r:=r/3;), 3*r=a+b+c] \\ &\Leftrightarrow [(r:=a+1; r:=r+b; r:=r+c;), r=a+b+c] \\ &\Leftrightarrow [(r:=a+1; r:=r+b;), r=a+b] \\ &\Leftrightarrow [(r:=a+1;), r=a] \\ &\Leftrightarrow (a=a+1) \\ &\Leftrightarrow \text{false} \\ &\Leftrightarrow\end{aligned}$$

Since “ $\text{true} \Rightarrow \text{false}$ ” is logically false (Table 4.2-4). Therefore program P does not satisfy its specification.

From the formal proving, we not only conclude mathematically that the program P does not satisfy its specification, but locate where the error is in the program. After fixing the error, the program becomes

```
r: = a;  
r: = r+b;  
r: = r+c;  
r: = r/3;
```

and we name the new program P' to distinguish it from the original program. Now let us prove program P' satisfies the specification. It is equivalent to prove the following is true.

$\text{pre-avg}(a, b, c) \Rightarrow \text{post-avg}(a, b, c, r)$
where
r = Program P'

With similar step, we get “ $\text{true} \Rightarrow \text{true}$ ” which is logically true. Therefore, it is concluded that Program P' does satisfy its specification.

Interested readers should refer to [Bloo86] and [Jone90b]. [Bloo86] provides detailed case study using VDM to formally specify a prototype nuclear reactor protection system and Prolog to animate the formal specification. More case studies can be found in [Jone90b].

A4.1.4 Recent Developments in VDM

As a model-based method in which models are described in terms of a state and the operations performed on it, current version of VDM is most suited for specifying sequential processes, rather than concurrent or temporal ones. Although a substantial amount of *syntactic sugar* (i.e., elements unnecessary to the syntax that help make the language more

readable) has been added so as to enhance the readability of VDM specifications, it requires specific “mathematical training and experience” for non-experts, as Williams addresses^[Will94]. Another shortcoming of VDM is its lack of modularity.

Since conventional VDM makes use of mathematical notations that are not easily understandable by the non-specialist, a visual presentation named “VDM through Pictures” (or VtP) is introduced in VDM specification^[Dick91] to overcome this barrier to the industrialization of VDM. The VtP tool consists of a diagram editor, a VDM editor and type-checker, and a set of transformations between diagrams and VDM. Two sorts of diagram are employed:

- Type-Structure Diagrams—portraying VDM type definitions.
- Operation-State Diagrams—portraying VDM operation definitions in terms of their pre- and post- conditions.

According to [Will94], a standard VDM notation, known as VDM Specification Language (VDM-SL) is being developed under the auspices of the International Standards Organization and the British Standards Institution.

More recently, VDM++, an objected-oriented extension of VDM, has been developed. VDM++ provides a wide range of constructs so that the users can *formally* specify concurrent, real-time systems in an objected-oriented fashion^[IFAD97]. Tool has developed to transform VDM++ specification (constructs) into C++ by its code generator^[Lori97].

Readers interested in VDM and its descendants should refer to [Lars96], whose more than 600 entries represent perhaps the most comprehensive coverage of bibliography on the VDM family, ranging from newcomer level to tool support.

A4.2 Z

Z, a model-based specification language, was pioneered by Jean-Raymond Abrial^[Abri81] at Oxford University in the late 1970s and has been developed in the Programming Research Group at the university and elsewhere.

A4.2.1 The Z Notation

The basic building block of Z is a *schema*, which consists of a signature part and a predicate part. The schema is typically enclosed in box structure with its name typed on the top of the box and a short line delimiting the signature from the predicate. The signature (or declaration part) declares all the variables and their types, including those predefined ones. The relationships that must be maintained among the variables are declared as predicates in the predicate part. Inputs to a schema are expressed by terminating them with a question mark. If the definition of a schema is applicable to all types, then a T surrounded by a pair of square brackets, i.e., [T], is placed above the definition. For a complete set of the syntax of Z specification, readers should consult Chapter 6 and Appendices of [Harr96] or Part III and Appendix of [Ince88].

Z specifications deals with three types of entities, namely, states, observations, and events. A *state* is the mathematical structure which models a system. It is a collection of observations. An *event* is an occurrence which is of interest to the specifier. An *observation* is a set of theoretic variable whose pre- and post- event values can be examined. By convention, the post-event value of a variable is indicated by appending a prime (') to its pre-event counterpart.

A Z specification intends to reflect two kinds of properties: the static properties and the dynamic properties. The former are also known as invariants, or predicates that hold true over the course of time regardless of what event occurs. The name of the invariant schema is conventionally preceded by the symbol Δ . The latter characterize the names of observations made before an event occurs, those made after the event, and a predicate that relates them.

A4.2.2 Z and VDM

Z is a formal specification language that is based on the concepts and notations of typed first-order logic and set theory. Therefore, sets, relations, functions, and predicate logic are extensively used in the specification. But one should not be deceived by the similarity between the Z and VDM specifications. Z is better structured than VDM. Z was invented to model abstract systems in terms of mathematics whilst VDM could be better describe as an attempt to abstract programming, e.g., purports to derive programs from specifications. Hence, VDM contains many low-level as well as high-level constructs. Focused on describing the properties of a system rather than the algorithms used by the implementers to achieve these properties, Z specifically eliminated the low-level constructs. The effect of inclusion low-level constructs is two-fold. On one hand, they makes it easier to translate a specification into a program. On the other hand, they can be abused and result in operational bias where the specification is too prescriptive and restricts the implementers' freedom. According to [Harr96], Z is more popular in academic circles than VDM and is more powerful. Currently, both VDM and Z are used at "an equal amount in the commercial world, but this seems to be changing with Z gaining dominance". As VDM, Z is best suitable for stand-alone process, rather than protocol and distributed system since neither of them has explicit support for concurrency and time.

A4.2.3 An Example in Z

For readers interested in the application of Z specification language, [Deli90] and [Nara90] are recommended. The former presents a case study of the formal specification of an electronic instrument whilst the latter describes that of the look manager in dialog system, who deals with the presentation of visual aspects of objects and the editing of the visual aspects. Here, a simple example of a segment of a natural language specification for a file system is given as follows.

Table A4-1 A Full List of Symbols Used to Distinguish Types of Relations in Z^{Ince88}

Symbol	Relation to Represent
\leftrightarrow	a relation
$-\mapsto$	a partial function*
\rightarrow	a total function
$>-\mapsto$	a partial injection
\Rightarrow	a total injection
* This should not be confused with \mapsto , the ordered pair operator, e.g., $\{1 \mapsto 2\}$ is the same as $(1, 2)$.	

“... Of the file system, there will never be more than 500 users who currently own files. No user will be allowed to own more than 300 files and no file can will be allowed to be more than 2000 blocks...”

In order to develop its corresponding formal specification using Z notation, we need to make some definitions first. The relation *ownership* is used to define the fact that a user may have more than one file (hence a function is inappropriate) while the function *file_space* is defined to relate files and the disc space (expressed in natural numbers) they take. *File_space* is a function since a file will correspond to a unique natural number at a time (Figure A4-2).

Recent development in Z specification language includes extending Z to Z++, an object-oriented extension of Z. The extension makes explicit the known object-oriented features of Z and allows more complex forms of inheritance of specifications, as well as multilevel specifications^[Lano90], e.g., from user interface through the specification of high-level operations down to the specification of low-level operations. The benefit of the extension is that modification can be made to each of the levels independently of and invisibly to other levels. Readers interested in the most recently application of Z notation are recommended to [Fran97].

Like VDM, the Z notations are very mathematical and not feasibly understood by non-specialist. To illustrate the idea that formal languages are not necessarily harder to understand, Randell showed the bi-directional translation between graphical Data Flow Diagrams (DFD) and the Z specification through a simplified credit card management system^[Rand90]. He claims that it is “easier to show the diagrams to others to gain approval” and that “generating a diagram from a Z specification is a useful way of helping to validate the specification”. But he also admits that further work needs to be done to specify rules for generating new diagrams, e.g., diagrams other than DFD.

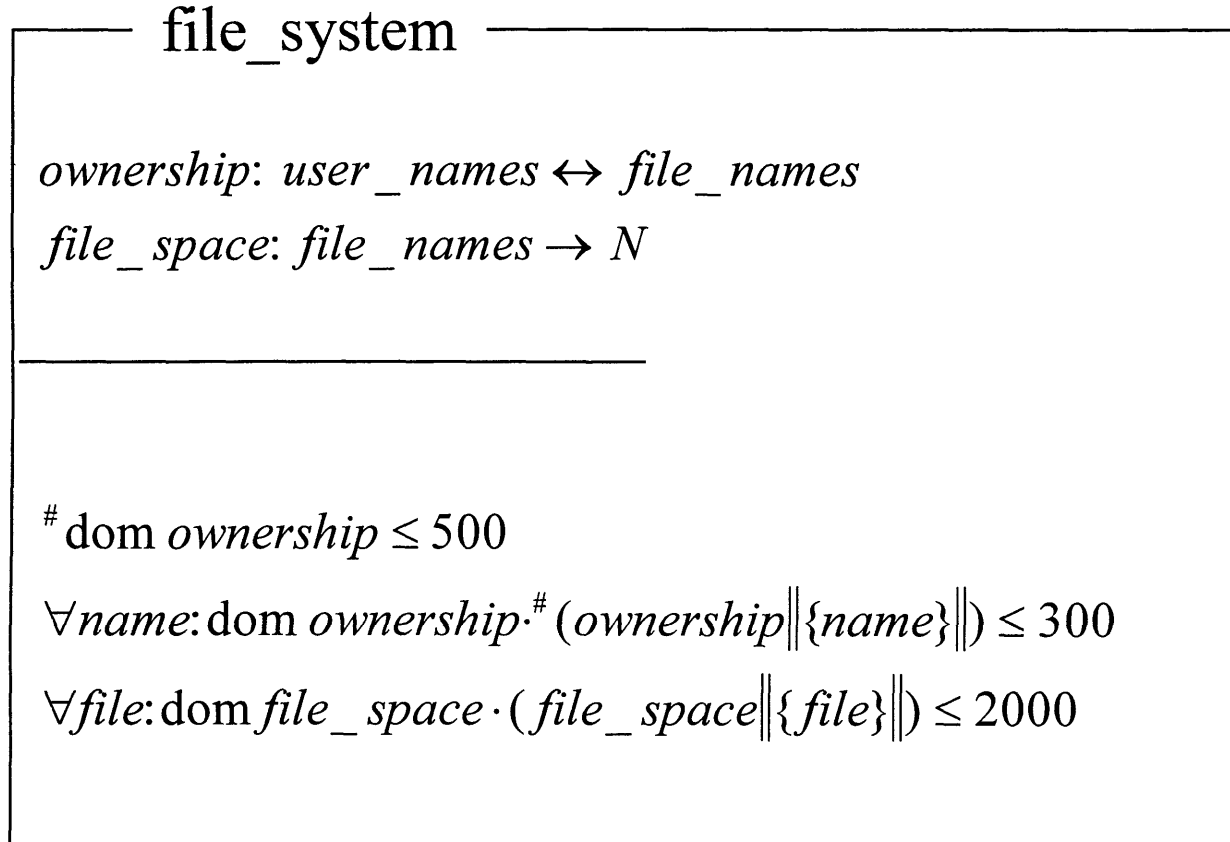


Figure A4-2 An Example of Specification in Z

A4.3 Development Before The Fact (DBTF) and the OO1 System

The Development Before The Fact methodology was developed and marketed by Hamilton Technology Inc.. According to [Ouya95], DBTF has been used in many industrial applications successfully. Currently, DBTF is supported by an integrated CASE Tool Suite called the OO1 System.

The OO1 AXES specification language has two, major components: Functional Maps (FMaps) and Data Type Maps (TMaps). The former specifies functional hierarchies that show interlocking and interconnecting functions at all levels of control while the latter defines all the primitive and abstract data types of the system objects. Objects which are elements of the TMaps are materialized as Objects Maps (OMaps) and an execution of a system that is an instance of a FMap is materialized as Execution Map (EMap). The OO1 Tool Suite is comprised of the following components:

- the *OO1 AXES Specifier*; an OMap Editor for defining FMaps and TMaps in either graphical form or textual form;
- an *Analyzer* to check the format of functions in the FMaps and data type specifications TMaps during the definition of a model to ensure the rules are followed correctly. The *Analyzer* acts as a compiler or an interpreter in a conventional programming environment;
- an *Executor* to execute or simulate (prototype) the behavior of a system before the programming to observe characteristics such as timing;
- a *Generator* to generate source code in programming language C automatically.

Despite the major advantages claimed by its advocates, the OO1 Tool Suite is by no means a perfect tool before the following problems are overcome.

- By virtually eliminating the design stage (Figures 2.2 and 3.4 of [Ouya95]), OO1 eliminates the possibility of uncovering specification errors before implementation. Hence, a perfect specification is required;

- “The DBTF technique and OO1 tool together provide four stages of verification in order to eliminate internal system defects, internal semantic defects, and external semantic defects.” The fourth stage of verification technique (Section 3.2.2 of [Ouya95]) doesn’t seem to be optimized since, as described in Section 3.4.3 [Ouya95], “it is at the fourth stage that defects of external semantics of the specifications are to be sought. This action is equivalent to the validation of the specification. It tries to answer this question: are we developing the correct specifications with respect to the requirements?”. Obviously, this stage should have been done in the first place since if the answer to the posed question is “No”, then we have wasted time and resources on solving a wrong problem before we even realize it at stage 4, and we have to spend extra resources to correct it;
- One of major advantages claimed by its developer is its ability to automatically generate C code based on its specification. But the code generated by the Suite may not be optimal without the active participation of human programmer. To understand this, consider the optimality of a code directly written in low level assembly language or machine language and a code in high level languages achieving the same function. The former is usually more optimal, more efficient in terms of time and space than the machine language code translated from the latter. This issue could be crucial in real-time control software where timing and code execution speed are essential to the control systems. Decreased participation of experienced programmer may also imply decreased understanding to the code;
- This tool system can only generate C code which can directly access memory through pointer arithmetic’s¹⁴, but C is a programming language that does not have the capability of automatic garbage collection. If a C program dynamically allocates memory to a data structure, it must remember to release or free the memory space once the structure is no longer used. Otherwise, the

¹⁴ Hoare argues^[Hoar75] that the pointer type of data structures is very analogous to the “goto” statement^[Dijk68] for control structures.

entire computer memory can be clogged with garbage. This kind of errors will not be caught at compilation time, but will be in effect at running time. Therefore, C may not be a good programming language in safety-critical application. In fact, as suggested in Chapter 6 of [Gro94], for the development of safety-critical software, it may be necessary to use a “safe subset” of a programming language, e.g., use only static data structure and avoid recursive technique. More explicitly in [UKMoD91], it is required that the implementation practice for safety-critical software avoid techniques which are difficult to analyze (e.g., excluding the use of dynamic memory allocation, recursion and floating point arithmetic) and implementation language be block and strongly typed. As illustrated in [INEL95], software developed in C becomes less and less maintainable when more functionality is added to the software;

- The code automatically generated is usually not commented. A non-commented code will hamper user’s understanding of the code and will make it difficult for the developers (or others) to upgrade or modify the code. Therefore, it is becoming a standard training and requirement for human programmers to comment their codes so as to facilitate understanding, communication, and peer review;
- The code generated is not well-indented. Indentation helps to modularize a code which helps user to use and understand the code;
- The Suite can not be used to reverse-engineer a code, e.g., can’t be used to study an existing code. From the practical experience of Darlington Nuclear Generation Station, it is sometimes necessary to generate software design document (SDD) from existing code to better understand, verify or upgrade the code;
- Using the Suite requires ad hoc training, e.g., training materials are tens of centimeters thick.

A4.4 Symbolic Model Verifier (SMV)

According to [Rush96], in the 1970s, formal methods were focused on proofs of program correctness which is not only a very expensive and difficult task, but also of little value since traditional methods of code review and testing are highly effective in detecting coding errors. In the 1980s, attention of formal methods shifted from program verification to the use of formalism in specifications while, in the 1990s, with the advent of efficient techniques for model checking, attention is paid to the important properties of a system rather than the full functionality of the system, model checking is generally focused on incorrectness, e.g., finding errors, rather than on trying to establish correctness.

According to [Sree96], model checking is an effective technique for verifying properties of a finite specification. A model checker accepts a specification and a property of the specification, then searches the reachable states to determine whether the property is a theorem of the specification or not. Model checking is a more thorough validation technique than testing executable specifications since it checks every reachable state of the specification.

A4.4.1 The SMV Notation

A typical specification written in Symbol Model Verifier (SMV) has four major sections: the VAR section, the ASSIGN section, the TRANS section, and the SPEC section.

Environment conditions (e.g., a set of predicates on environmental variables), mode classes and modes are defined as SMV variables declared in the VAR section. Most conditions are expressed as boolean variables while others are represented as variables of enumerated types, e.g., in terms of set, if these variables are related in such a way that exactly one condition is true at all times.

In the ASSIGN section, one expects to find the system's initial state in the *init()* statement and transition relation in the *next()* statement that specifies the next value of the mode class

variable. *Next()* statement usually employs a *case* expression whose final branch beginning with “1” indicates the *else* clause. “!” is the SMV *not* operator, “&” is the SMV *and* operator, “|” is the SMV *or* operator, and “->” is the SMV *implication* operator.

Environmental assumptions, e.g., conditions imposed either by laws of the nature or by other mode classes in the system, are invariant constraints that hold in all reachable states of the specification graph and are expressed in the form of DNF assertions declared in the TRANS section. DNF represents Disjunctive Normal Form, which is defined to be the disjunction of clauses, each of which is the conjunction of one or more literals, which are logical variables or their negations.

The properties to be verified with respect to an SMV specification are declared in the SPEC section. The SMV model checker verifies the properties expressed in the form of computational tree logic (CTL) branching-time temporal logic. The details of the syntax and semantics for CTL formulas can be found in [Clar86]. A summary is provided below.

1. Every propositional variable is a CTL formula,
2. If ϕ and φ are CTL formulas, so are: $!\phi, \phi \& \varphi, \phi | \varphi, AX\phi, EX\phi, EF\phi, AG\phi$, where
 - ! is the *not* operator, & is the *and* operator, | is the *or* operator,
 - X is the *next-state* operator, E is the operator for *existential quantifier*,
 - A is the operator for *universal quantifier*, e.g., $AX\phi$ is true in state s_i if and only if (iff) formula ϕ is true in every successor state of s_i in the reachability graph,
 - F is the *future* operator, e.g., $EF\phi$ is true in state s_i iff along some path from s_i there exists a future state in which formula ϕ is true,
 - G is the *global* operator, e.g., $AG\phi$ is true in state s_i iff ϕ holds in every state along every path stemming from s_i .

A4.4.2 An Example in SMV

Figure A4-3 is a specification of a thermostat that regulates the temperature of a room in SMV notation.

MODULE main

VAR

Thermostat: {Off, Inactive, Heat, AC};
Enum1: {TooCold, TempOk, TooHot};
SwitchIsOn: boolean;

ASSIGN

init(Thermostat):=Off;
init(SwitchIsOn):=0;

next(Thermostat):=case
 Thermostat =Off & !SwitchIsOn & next(SwitchIsOn) & Enum1=TooCold: Heat;
 Thermostat =Off & !SwitchIsOn & next(SwitchIsOn) & Enum1=TempOk: Inactive;
 Thermostat =Off & !SwitchIsOn & next(SwitchIsOn) & Enum1=TooHot: AC;
 Thermostat =Inactive & SwitchIsOn & next(!SwitchIsOn): Off;
 Thermostat =Inactive & SwitchIsOn & !Enum1 =TooCold & next(Enum1=TooCold): Heat;
 Thermostat =Inactive & SwitchIsOn & !Enum1 =TooHot & next(Enum1=TooHot): Heat;
 Thermostat =Heat & SwitchIsOn & next(!SwitchIsOn): Off;
 Thermostat =Heat & SwitchIsOn & !Enum1=TempOk & next(Enum1=TempOk): Inactive;
 Thermostat =AC & SwitchIsOn & next(!SwitchIsOn): Off;
 Thermostat =AC & SwitchIsOn & !Enum1=TempOk & next(Enum1=TempOk): Inactive;
1: Thermostat
esac;

TRANS

(Enum1=TempOk & !next(Enum1=TempOk) & !Enum1=TooHot & next(Enum1=TooHot)) |
(!Enum1=TempOk & next(Enum1=TempOk) & Enum1=TooHot & !next(Enum1=TooHot)) |
(Enum1=TooCold & !next(Enum1=TooCold) & !Enum1=TempOk & next(Enum1=TempOk)) |
(!Enum1=TooCold & next(Enum1=TooCold) & Enum1=TempOk & !next(Enum1=TempOk)) |
(Enum1= next(Enum1))

TRANS

(!(Enum1=next(Enum1)) & (SwitchIsOn =next(SwitchIsOn))) |
((Enum1=next(Enum1)) & !(SwitchIsOn =next(SwitchIsOn))) |
((Enum1=next(Enum1)) & (SwitchIsOn =next(SwitchIsOn)))

SPEC

AG((Thermostat =Off -> !SwitchIsOn)
AG((Thermostat =Inactive -> (SwitchIsOn & Enum1=TempOk))
AG((Thermostat =Heat -> SwitchIsOn & Enum1=TooCold))
AG((Thermostat =AC -> (SwitchIsOn & Enum1=TooHot))

Figure A4-3 The SMV Specification for a Thermostat (from [Sree96])

Appendix 5

Software Safety Analysis Methodologies

A5.1 Software Fault Tree Analysis (SFTA)

Originally developed in 1961 at Bell Labs to evaluate the Minuteman Launch Control System for the possibility of an inadvertent missile launch, Fault Tree Analysis (FTA) is a widely used safety analysis method in the aerospace, electronics, and nuclear industries [USNRC75]. With successful applications in hardware hazard analysis, FTA uses a backward analysis strategy, e.g., starts with a hazardous state of the system and uses Boolean logic to determine what the next level of conditions or events can lead to an upper level event, the process continues until we reach the basic events or minimum cut set, e.g., those we understand pretty well.

The idea in hardware fault tree analysis (HFTA) was borrowed by Leveson to analyze software errors [Leve83]. Software Fault Tree Analysis (SFTA) proceeds in a similar manner to HFTA and uses a subset of symbols in HFTA, e.g., *AND* gate and *OR* gate. The Software Fault Tree Analysis is a fault avoidance technique in that it can be used to identify where needs special attention. As discussed in [Leve95], FTA contains four basic steps.

(1) System Definition

This step determines the top events, initiating events, boundary conditions, existing events, and impossible events. This step is crucial in that FTA is not comprehensive until the fault trees are developed for all possible, significant top events.

(2) Fault Tree Construction

Once step 1 is finished, analyst can start constructing fault tree based on casual events related with the top events and logical relations between them.

(3) Qualitative Analysis

After fault tree is constructed, one can reduce the tree to an equivalent logic expression using only *AND*, *OR*, and/or *NOT* gates. By manipulating Boolean algebra, Minimum Cut Set (MCS) can be obtained from the logic expression. Therefore, analyst know qualitatively what combinations of basic events can lead to the top event. This analysis can help identify the weakness of a system, e.g., by identifying and modifying single event MCS.

(4) Quantitative Analysis

From Boolean manipulation in step 3, the top event can be expressed as the logic *OR* of all independent MCSs.

$$top\ event = \bigcup_{i=1}^n MCS_i$$

where

\cup is the notation for union (e.g., logic *OR*) operation,

n is the total number of MCSs.

If the probabilities of MCSs are known, we can compute probability of the top event. (Strictly speaking, we can only obtain an upper bound of the probability of top event.) Applying probability to the above equality yields

$$\Pr(top\ event) = \Pr\left(\bigcup_{i=1}^n MCS_i\right)$$

Applying Boole' Inequality and mathematical induction (Appendix 5.1.1), we get an upper bound for the probability of the top event.

$$\Pr(top\ event) = \Pr\left(\bigcup_{i=1}^n MCS_i\right) \leq \sum_{i=1}^n \Pr(MCS_i)$$

Information from the SFTA can be used to identify minimum cut sets (MCS) that lead to software failure or hazardous configuration, thus measures can be taken to avoid such MCSs from occurring, e.g., through modifying/fixing the software being analyzed.

A5.1.1 An Upper Bound for the Probability of Top Event

From Set Theory([Rose95]), it is obvious that, for any sets A and B (Figure A5-1), the following *inclusion-exclusion principle* holds.

Theorem A5.1

$$|A \cup B| = |A| + |B| - |A \cap B| \quad (\text{A5-1})$$

where

$|X|$ represents the cardinality of set X , i.e., number of elements in set X .

From probability theory, for any events A and B , there exists a similar theorem, i.e., dividing both sides of Theorem A5.1 by the cardinality of the sample space yields Theorem A5.2.

Theorem A5.2

$$\Pr(A \cup B) = \Pr(A) + \Pr(B) - \Pr(A \cap B) \quad (\text{A5-2})$$

where

$\Pr(X)$ represents the probability of event X .

Proof.

Let set S be the sample space.

Dividing both side of Equation (A5-1) by the cardinality of set S , $|S|$, yields

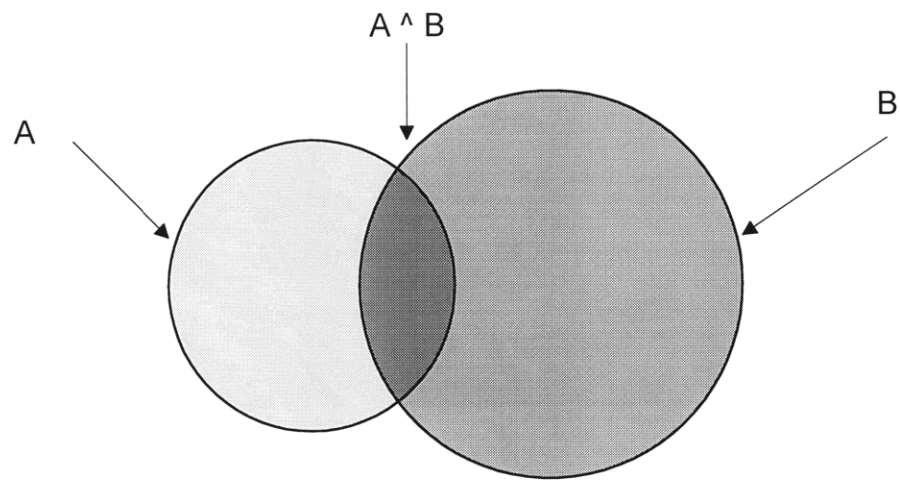
$$\frac{|A \cup B|}{|S|} = \frac{|A|}{|S|} + \frac{|B|}{|S|} - \frac{|A \cap B|}{|S|} \quad (\text{A5-3})$$

Note that, by definition of probability, we have

$$\Pr(A \cup B) = \frac{|A \cup B|}{|S|}, \Pr(A) = \frac{|A|}{|S|}, \Pr(B) = \frac{|B|}{|S|}, \text{ and } \Pr(A \cap B) = \frac{|A \cap B|}{|S|}.$$

Therefore, Equation (A5-2) holds.

■



$$|A \cup B| = |A| + |B| - |A \cap B|$$

Figure A5-1 Union of Sets A and B—Cardinality

Corollary A5.3 (Boole's Inequality)

$$\Pr(A \cup B) \leq \Pr(A) + \Pr(B) \quad (\text{A5-4})$$

Proof.

Since $\Pr(A \cap B) \geq 0$, Corollary is obvious from Theorem A5.2. ■

More interesting case is that Boole's Inequality can be extended to n events, which is given as Theorem A5.4.

Theorem A5.4

For all $n \geq 1$ and any events A_1, A_2, \dots, A_n , we have

$$\Pr\left(\bigcup_{i=1}^n A_i\right) \leq \sum_{i=1}^n \Pr(A_i) \quad (\text{A5-5})$$

Proof.

We prove it by induction number of events n .

Base case of $n = 1$, we have $\Pr(A_1) \leq \Pr(A_1)$ which is trivially true.

Induction step. Assume that the theorem holds for $n-1$ events, that is,

$$\Pr\left(\bigcup_{i=1}^{n-1} A_i\right) \leq \sum_{i=1}^{n-1} \Pr(A_i) \quad (\text{A5-6})$$

Now what we want to show is that the theorem holds for n events.

$$\Pr\left(\bigcup_{i=1}^n A_i\right) = \Pr\left[\left(\bigcup_{i=1}^{n-1} A_i\right) \cup A_n\right] \quad (\text{A5-7})$$

Applying Corollary A5.3 with $A = \bigcup_{i=1}^{n-1} A_i$, $B = A_n$ to the right hand side of the above equality, we get

$$\Pr\left[\left(\bigcup_{i=1}^{n-1} A_i\right) \cup A_n\right] \leq \Pr\left(\bigcup_{i=1}^{n-1} A_i\right) + \Pr(A_n) \quad (\text{A5-8})$$

Using induction hypothesis to the above inequality, we have

$$\Pr(\bigcup_{i=1}^n A_i) \leq \Pr(\bigcup_{i=1}^{n-1} A_i) + \Pr(A_n) \leq \sum_{i=1}^n \Pr(A_i) \quad (\text{A5-9})$$

Therefore, it is true for n events, and, by induction, Theorem A5.4 holds. ■

Theorem A5.4 gives an upper bound of the probability of the union of n events in terms of each individual n events. In the Fault Tree Analysis, the n events can be Minimum Cut Sets and the union of the n events can be the top event. By using Theorem A5.4, an upper bound of the top event can be evaluated.

A5.1.2 Applications of SFTA and Tool Support

Ovstedal applied SFTA technique to software development process at a high level^[Ovst91], rather than at code level as Leveson did ^[Leve83].

Tools of SFTA have been developed to analyze Software Requirements Specification [Rata96]. However, SFTA is not an panacea, its shortcomings are obvious.

- Fault Tree Analysis is primarily a means to analyze the causes of hazards, not to identify hazards. Hence, other methods must be used prior to FTA to identify the hazards;
- For SFTA to be useful, all possible software failure modes must be identified and analyzed to ensure complete coverage. This task is usually too demanding for real life software. Hence it implies that SFTA is practical only when there is a small number of critical failures exist;
- HFTA usually generates the probability of the top event being analyzed through calculating probabilities of the basic events, e.g., determine the failure

probability of a system through computing those of pumps and valves. However such probability may be meaningless in software since software has its distinguishable characteristics from hardware, e.g., hardware usually fails independently while software component failures may be correlated, hardware fails by aging while mean time to failure (MTTF) or mean time between failure (MTBF) may be meaningless in software. Academic people are still debating if software failure probability does exist ^[Sing95];

- SFTA is used for verification purpose, hence, code must have already been developed to generate fault tree for the code.

SFTA has been used in Darlington Nuclear Power Generation Station in Canada to examine hazardous failure modes^[BowW].

A5.2 Dynamic Flowgraph Methodology (DFM)

Combining discrete transition models with Logic Flowgraph Methodology (LFM) [Guar84], DFM is a new approach for embedded system safety analysis. This methodology integrates the modeling and analysis of the hardware and software components of the embedded system. Normally, DFM works in a deductive fashion, i.e., use reverse causality backtracking of a top event to find the *prime implicants*. Prime Implicants are the unique failure modes of non-coherent systems, which contain other than simple fault modes connected by *AND-OR* logic gates^[Henl92].

A DFM model integrates three types of networks: *Time-Transition network* to describe the sequence in which software modules (e.g., subroutines) are executed, *Casuality Network* showing the functional relationships among key hardware and software parameters, and *Conditioning Network* modeling discrete software behavior caused by conditional branching or discontinuous hardware performance due to component failure.

The building blocks of these networks include *Process Variable Nodes*, *Condition Nodes*, *Causality Edges*, *Condition Edges*, and *Transfer/Transition Boxes* and their associated decision tables. Process Variable Nodes are used to represent essential physical or hardware variables of the digital control system. Like process variable nodes, Condition Nodes represent physical or software parameters that identify component failure, changes of process operation regimes and modes, or software conditional branching. Condition Nodes can be associated with Transition for this purpose. Causality Edges are employed to connect process variable nodes which has a cause-and-effect relationship between the variables. Condition Edges are used to model true discrete behavior of the system. They link parameter nodes to transfer boxes, showing the possibility of using a different transfer function to map input variable in output variable states. A Transfer Box models a transfer function between process variable nodes. In spite of indicating cause-and-effect relationship similar to Transfer Box, a Transition Box is different from transfer box in that a time delay for the transition is assumed to exist between the time when the input variable states become true and the time when the

corresponding output variable states are reached. Decision As extension of Truth Tables, Decision Tables allow each variable to be represented by any number of states instead of just 0 (false) and 1 (true). They are constructed from empirical knowledge, physical equations, software, or pseudo code.

The advantages of DFM over Software Fault Tree Analysis (SFTA) include multi-state logic (instead of 0-1 logic) and time-dependent dynamic nature. In other words, DFM provides a timed fault tree which can be viewed as a series of snapshots of convention fault trees [Garr95]. In the conventional fault tree, only the probabilities of occurrence of events may be time-dependent while the events themselves are also time-dependent in the DFM model. However, current version of DFM does not solve all the problems. It needs to be improved in several areas.

- DFM works in a deductive fashion, i.e., reverse causality backtracking of a top event [Yau95]. So current version of DFM can not analyze whether a given set of initial condition will lead to a failure state;
- The effectiveness of DFM depends on the selection of top events, e.g., to enumerate and analyze all the possible top events. This completeness of top events is difficult to achieve for a complex software system;
- Since DFM depends on software modules to generate input/output relations, it can't be used to detect such software errors as logically inaccessible portion of a module;
- DFM lacks of degree of hierarchy, e.g., simulating hardware and software in one single model. Hence, even if DFM works well on simple examples, it will encounter difficulty when applied to real life problems;
- The prime idea embedded in DFM is to discretize a process variable which is usually continuous in range (e.g., pressure in a tank) into a finite number of states, the reason for the discretization is to simplify the description of the relations between different variables. However the “discretization” and

“simplification” could distort the (continuous) world in which we are interested (since there is no solid mathematical basis preventing the distortion from occurring). For example, a deviation of 2% from nominal flowrate of the continuous world may not imply an unacceptable state, but it could be unacceptable in the discretized model if the discretized threshold is set at 101% (or 99%) of nominal flowrate since the deviation would result in a different state.

A5.3 HAZards and OPerability Analysis (HAZOP)

Initially developed at Imperial Chemical Industries (ICI) in England in the early 1960s, HAZOP was improved and published by the Chemical Industries Association in London. Currently, about half of the chemical process industries are using HAZOP in their new facilities. According to [Broo96], a HAZAP process is comprised of six stages, HAZOP 1–HAZOP 6, spanning from the conceptual stage of a project to the commissioning stage.

As a qualitative method, HAZOP is used to identify all possible deviations from the design's expected operation and all hazards associated with these deviations. It can not only identify hazards in the existing design, but also elicit hazards in the new designs or hazards that have not been considered previously. The method requires creative thinking about all the possible ways in which hazards or operational problems may arise. To reduce the chance of missing anything, HAZOP is performed systematically, taking into account every process unit in the plant and every hazard in turn.

The differences between HAZOP and Failure Modes and Effects Analysis (FMEA) can be found in [Redm96] and is summarized here. First, a HAZOP is a team exercise to identify hazards by examining the interactions between components while an FMEA examines the possible failures of the components themselves and can be performed by an individual. Second, a HAZOP starts with searching for possible deviations from design. The study is bi-directional once a deviation is found: it can backtrack the possible causes of the deviation as well as deduce the likely hazardous consequences. An FMEA, however, is unidirectional: after identifying a possible component failure, it proceeds to investigate the likely consequences on the system as a whole. Problems with HAZOP include

- it is difficult for HAZOP to uncover all hazards completely;
- HAZOP only provide qualitative results, not quantitative analysis;
- HAZOP relies strongly on the judgment, experience and competence of the engineers performing the assessment;

- HAZOP is a time- and labor-intensive team work^[Redm96] requiring intensive cooperation between individuals in the team.

In addition to application in hardware equipment, Reese devised an automated variant of HAZOP, Software Deviation Analysis (SDA), to analyze Software Requirements Specification ([Rees96]). An overview^[Rees97] of the procedure is summarized as follows.

The SDA is a forward search procedure, i.e., it starts with a deviation in the input environment of a software and attempts to find paths that lead to hazardous software outputs. The analyst provides a formal software requirements specification, which will be automatically translated into *causality diagram*. Then SDA uses *deviation formulas*, which define how deviations are related. With this information incorporated into causality diagram, an *augmented causality diagram* is generated. Next, *qualitative mathematics* is used on the augmented causality diagram to evaluate deviations. (see Figure A5-2) The *forward and backward definite functions* are used to construct a chain of states representing the path which will definitely be stemmed from the initial state. The chain terminates either with a safety-critical deviation or no deviation at all. In parallel, SDA can continue the search by constraining the software state using the *forward assumptive function* with constraints added to the initial state or every state in the chain. Then forward and backward functions are used to the additional constraints to create another chain of states. In a breadth-first fashion, SDA creates a tree of state chains when further constraints are added to the states. The height of a leaf in the tree corresponds to the additional assumptions made to reach that leaf. The procedure stops when it either runs out of constraints that can be made or the height reaches some predefined limit. Problems with SDA include

- a formal software requirements specification must exist prior to the Software Deviation Analysis—which does not alleviate the burden of an analyst at all;
- For a novel analyst, it may not be an easy task to determine what assumption is appropriate in the forward assumptive function;
- To stop the analysis when a height reaches the predefined limit is likely to neglect hazardous state, e.g., the hazardous state could occur at the next stage.

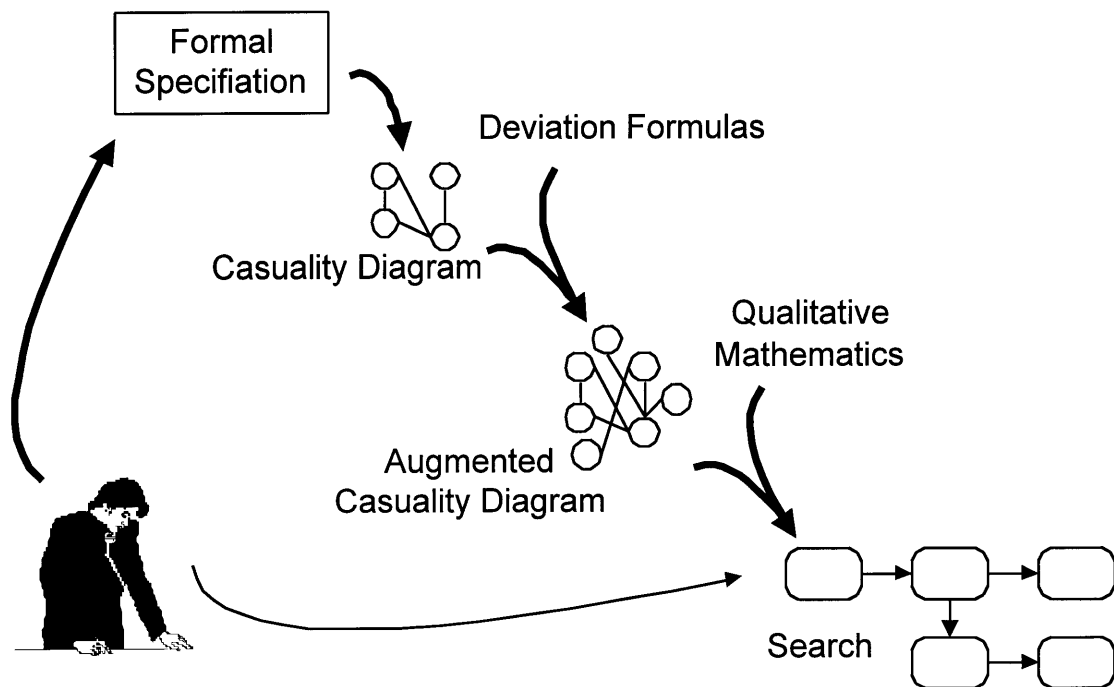


Figure A5-2 An Overview of the SDA Procedure (after [Rees97])

Appendix 6

Application of TBGA to Prove Logic Identity

The tree-based graphical approach developed in Section 4.4.2 can be used to prove identity of certain logical expressions. The benefit of the method is that an analyst is not required to have a strong background in logical operations, nor is it necessary for him to remember many formulas.

Example A6-1: Show that $A \wedge (\neg A \vee B)$ is identical to $A \wedge B$.

Method 1—Conventional approach

$$\begin{aligned} & A \wedge (\neg A \vee B) \\ &= (A \wedge \neg A) \vee (A \wedge B) \\ &= A \wedge B \end{aligned}$$

Method 2—Graphical approach

Taking the conjunction of the leaves in Figure A6-1 yields $A \wedge (\neg A \vee B) = A \wedge B$.

Example A6-2: Show that $(A \Rightarrow B) \wedge (B \Rightarrow C)$ is equivalent to $(\neg A \wedge B) \vee (B \wedge C)$.

Method 1—Conventional approach

$$\begin{aligned} & (A \Rightarrow B) \wedge (B \Rightarrow C) \\ &= (\neg A \vee B) \wedge (\neg B \vee C) \\ &= (\neg A \wedge \neg B) \vee (B \wedge \neg B) \vee (\neg A \wedge C) \vee (B \wedge C) \\ &= (\neg A \wedge \neg B) \vee (B \wedge C) \vee (\neg A \wedge C). \end{aligned} \tag{A6-1}$$

Method 2—Graphical approach (see Figure A6-2))

Use of Shannon's expansion theorem of logical expression

$$\begin{aligned} & \forall 1 \leq i \leq n, \\ & f(x_1, x_2, \dots, x_n) = (\neg x_i \wedge f|_{x_i = \text{false}}) \vee (x_i \wedge f|_{x_i = \text{true}}) \end{aligned} \tag{A6-2}$$

and expanding Figure A6-2 on B yields

$$f(A, B, C) = (\neg B \wedge \neg A) \vee (B \wedge C) \quad (\text{A6-3})$$

where

$$f|_{B=false} = \neg A \text{ and } f|_{B=true} = C \text{ are obtained from Figure A6-2.}$$

That expressions (A6-1) and (A6-3) are equivalent can be shown by the consensus method^[Quin55].

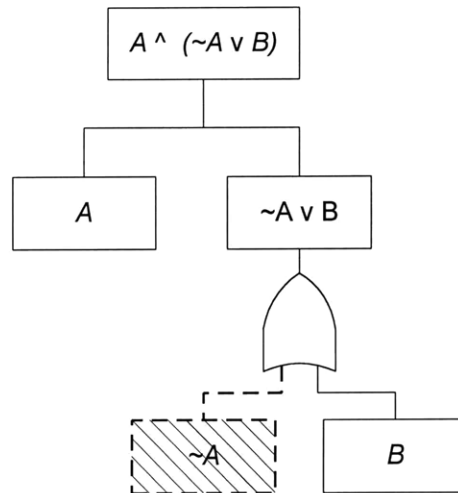


Figure A6-1 Application of the Graphical Method to Example A6-1

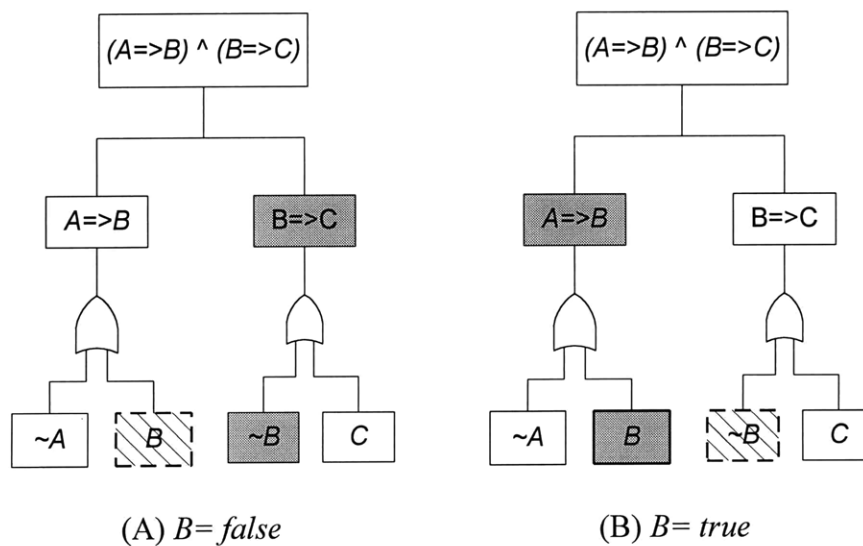


Figure A6-2 Application of the Graphical Method to Example A6-2

Appendix 7

Another Graphical Method Developed to Document Software Requirements Specification

Another graphical method has been developed to target practitioners. The method is also a tree-based approach. Used widely in computer science (e.g., in data structure, search, sorting, optimal file coding, file management), a *tree* is a connected, undirected graph with no simple circuit^[Rose95]. A tree is comprised of a root vertex (or root node), internal vertices, leaves (vertices without children), and edges (connecting vertices). The height of a tree is the maximum of the levels of vertices. An m -ary tree is a tree, each of whose internal vertex has at most m children. An m -ary tree with $m=2$ is called a binary tree.

In this graphical method, the internal vertices in the m -ary tree represent the logic variables of interest while the edges represent the logic expressions (e.g., expressions in predicate calculus). These logic expressions partition the domain of the logic variables and are overlaid on the edges of the tree. Each leaf represents action to be taken when the conditions on the path from root to the parent of the leaf are satisfied, e.g., conjunction of the predicates on the path is logically *true*. The height of the m -ary tree equals to the number of logic variables involved in the problem.

This method facilitates communication among not only theoretical computer scientists, but, more importantly, among practitioners such as domain experts whose interests lie in solving engineering problems with the help of computer.

A7.1 A *M*-ary Tree-Based Graphical Method and An Example

The graphical method is intended to be used in both the development and verification of the software. When the method is used to develop mission-critical software, a forward process is used, i.e., from customer's Software Requirements (SR), we develop SRS (the first formal document), from which we develop the Software Design Description (SDD), which is then used by the programmer to develop software. When the method is used to verify mission-critical software, a reverse-engineering process is used, i.e., from the software, we back out its SDD, from which we obtain the SRS which will be compared with the original SRS to check their consistency and completeness. The following reverse-engineering example reproduced in Figure 2.3-1 will show that the graphical method is superior to the tabular method.

Step1 Find all logic variables that determine the height of the tree

Find all logic variables, e.g., variables in guard part of the `if` statement, in the code of interest. Discovering all these variables is important, since they represent the height of the tree and the complexity of the tree structure. In this example, three variables were identified: `signal`, `setpoint`, and `trip_state`. Therefore, the tree to be developed is of height three.

Step2 Partition¹⁵ the input domains of the all logic variables found in Step 1

The way to partition the domain of a variable is usually based on the threshold values of the variable in the code. These values result in different control commands/actions to be taken. For example, the domains of `signal`, `setpoint`, and `trip_state` are partitioned respectively as follows:

¹⁵ A partition of a set *S* is a collection of pair-wise disjoint non-empty subsets whose union is set *S*.

signal $\in (-\infty, \text{sig_min}) \cup [\text{sig_min}, \text{setpoint_db}] \cup$
 $(\text{setpoint_db}, \text{setpoint}) \cup [\text{setpoint}, \text{sig_max}] \cup$
 $(\text{sig_max}, +\infty)$ ¹⁶

setpoint $\in (-\infty, \text{setpt_min}) \cup [\text{setpt_min}, \text{setpt_max}] \cup$
 $(\text{setpt_max}, +\infty)$

trip_state $\in \{\text{tripped}, \text{not_tripped}\}$

The partition corresponds to the edges of a node in the tree build in Step 3.

Step 3 Build the balanced m-ary tree top-down and left-right

The height of the m-ary tree represents the number of variables found in Step 1. Non-leaf nodes *at a level* represent a logic variable. Each leaf node represents the action or consequence when the conditions on the path from the root of the tree to the parent of the leaf are satisfied. For example, in Figure A7-1, the shadowed leaf represents the status (or action statement) of new `trip_state` if the `setpoint` is between `setpt_min` and `setpt_max` inclusive, `signal` is between `setpoint-db` and `setpoint` inclusive, and the current `trip_state` is `tripped`. The edges are organized in such a way that, if the partitioned sub-domain is a range of values, the sub-domains with smaller values are located from left to right. The edges immediately below an internal node represent the partition of the corresponding logic variable in Step 2. If an edge is not labeled with the partition, it is defaulted as the “otherwise” case, e.g., the sub-domain other than any of those labeled on the edges of the same node. As a result, to eliminate confusion, ambiguity, and inconsistency there is at most one unlabeled edge for each internal node. Each level of the internal nodes in the tree represents a logic variable while the leaves in the tree represent the

¹⁶ $-\infty$ and $+\infty$ are used to indicate completeness of the partition. They should be taken as the most negative and positive values expressible on a target machine.

results of code, corresponding to the “ACTION STATEMENTS” in tabular notation in Section 2.3.1. The internal nodes are represented in circles while the leaves in rectangles.

In this example, we can first assign “tripped” to all the rectangular boxes of the leaves with height 3, then, overwrite those boxes specified in the `if` statements in the code (Figure A7-1 with slight modifications).

Step 4 Transform the balanced tree into a simplified, unbalanced tree

Collapse paths from root (or its descendants other than the leaves) to leaves of the same “ACTION” as much as possible, e.g., by pass all logic variables along the paths that the action doesn’t depend on (e.g., the *don’t-care* variables) and replace the edges (along each path) originating from these internal nodes in the tree by dashed edges (Figure A7-2). Unlike other application of trees in computer science (e.g., search and sorting) in which a balanced tree is desirable, an unbalanced tree is preferred in the proposed method since it will simplify the tree.

Step 5 Further prune the edges of the unbalanced tree

When traversing the tree from root to leaves, if we encounter a dashed edge, we can eliminate the internal node from which the dashed edge originates. This is because, each dashed edge represent that fact the “ACTION STATEMENT” does not depend on the node. Therefore, it can be pruned. The simplified tree is shown in Figure A7-3. For example, the left most node `trip_state` is pruned from Figure A7-2.

A7.2 Advantages of the Graphical Method

By comparing Figure A7-3 with Table 2.3-1, it is concluded that:

- the tabular method is good when no more than two variables are involved because table is usually two-dimensional. However, the tree-based graphical method is not limited to two logic variables. For each additional logic variable, we simply increase the height of tree by one. Please be noted that Table 2.3-1 contains less information than Figure A7-3;
- the tree-based graphical method is similar to event tree in system reliability analysis if the tree is rotated anti-clockwise by 90°. However, there are fundamental differences. In this method, there is no temporal dependence of a node downstream (i.e., away from the root of the tree) on nodes upstream (i.e., towards the root). In other words, the tree is not unique. In event tree analysis, event downstream is usually dependent on the events upstream. Second, domain of an event is partitioned into at most two values, i.e., success or failure, in event tree analysis. In the graphical approach, the partition is a variable is not limited to two sub-domains;
- in developing mission-critical software, the graphical and the tabular methods can be used to double check each other for completeness and consistency of SRS and SDD.

A7.3 Findings

Application of the graphical method to the MIT-SNL Control Laws^[Bern89] reveals that, in function REACTR(P), a portion of the code is inaccessible. The error was first found using flow chart as explained in Appendix 8.

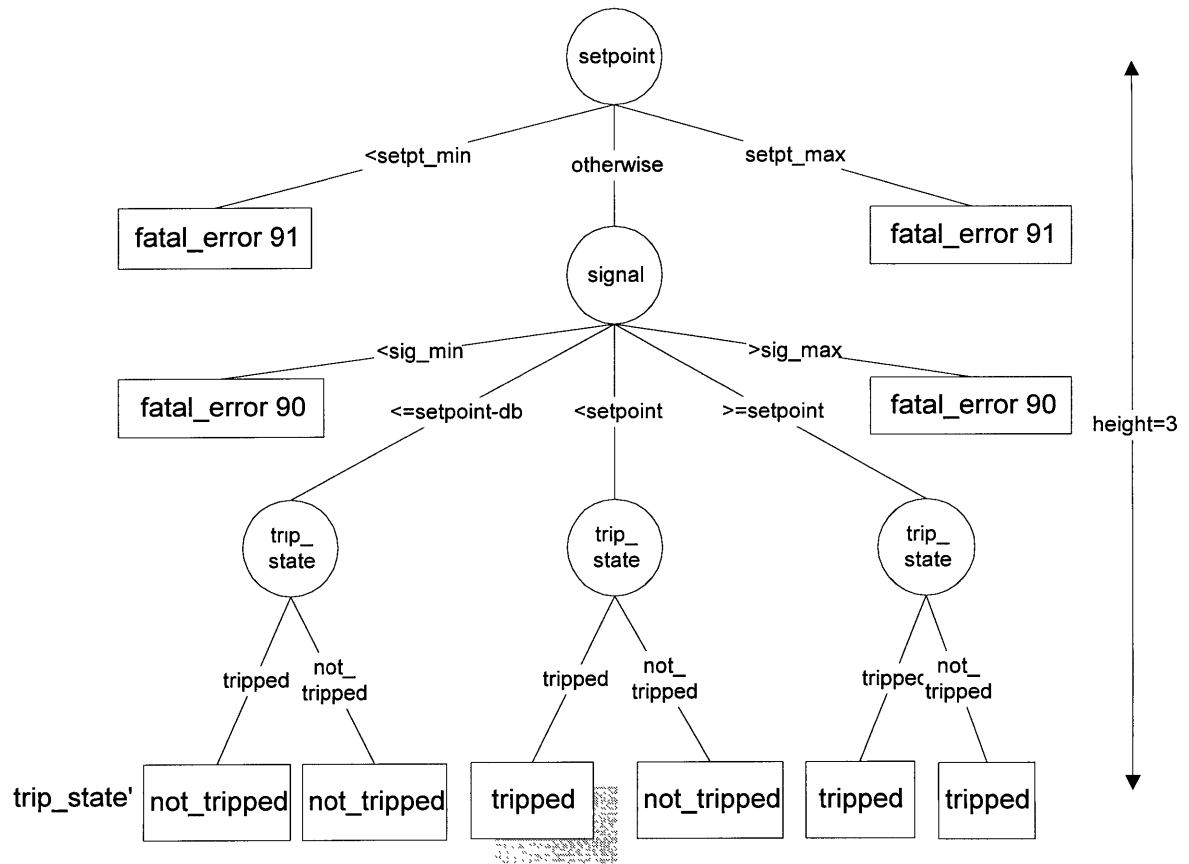


Figure A7-1 *M*-ary Tree Developed at Step 3 of the Example

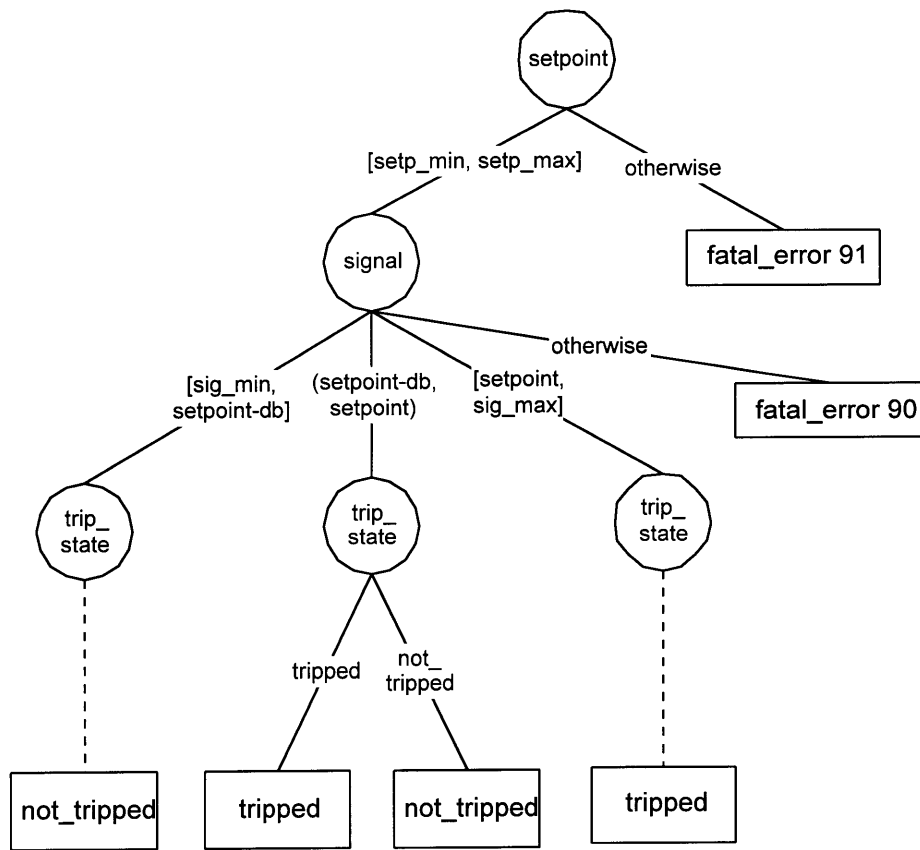


Figure A7-2 Simplified *M*-ary Tree Developed at Step 4 of the Example

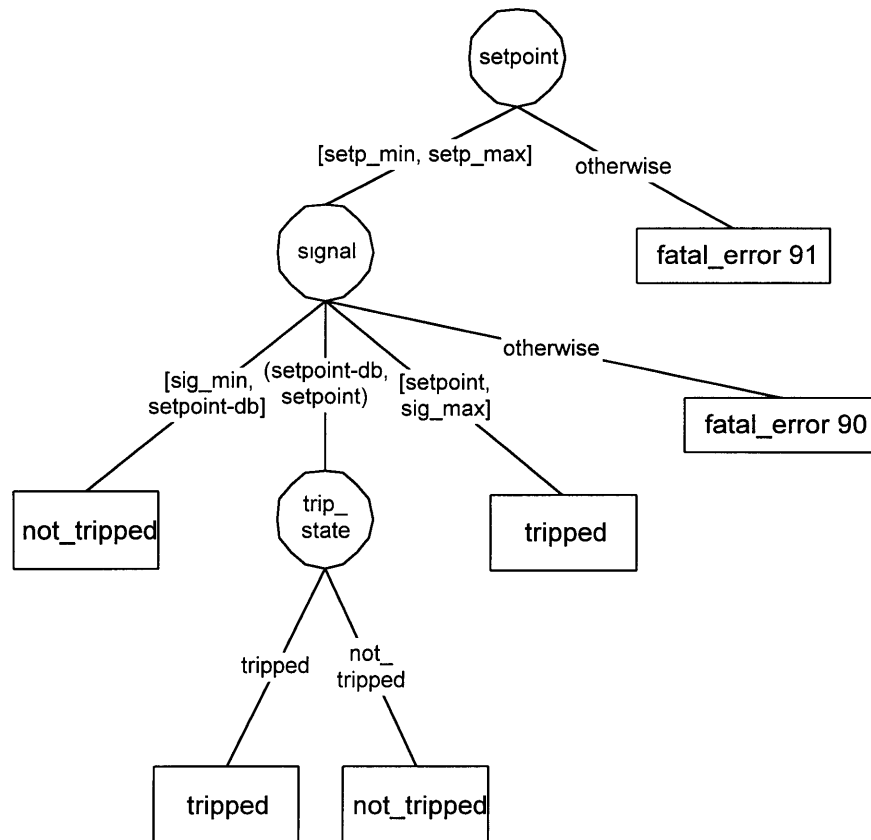


Figure A7-3 Final *M*-ary Tree Developed at Step 5 of the Example

Appendix 8

Application of Flowchart in Code Analysis

A8.1 The Flowchart

The flowchart can be used in code logic analysis. The advantages of this method become appreciate when it is applied to situation in which code review or “mental” execution of the code becomes difficult to track due to complex logic involved. For example, if many “if...then...else...” statements are in tandem, such mental execution can easily make reviewers strayed. This is one of the reasons that the complexity of codes can not be determined simply by the number of lines of code (LOC) in them. It is these statements, together with jump statements such as “go to” in the codes, that reduce the readability of the codes. However, the flowchart makes use of the fact that even though a series of such statements combined are entangling and misleading, each of them is very easy to handle since it has at most two branches, the “then” part and the optional “else” part.

A8.2 Application of Flowchart

The flowchart has been applied to analyzing a function in the MIT-SNL Control Laws listed in Appendix A8.3. The flowchart of Function REACTR (P) is shown in Figure A8-2. Using flowchart, we easily claim and prove that the last branch (marked with *) of the function in Figure A8-2 is an extraneous one which will be inaccessible under any circumstances. To prove this claim, a generic case based on Figure A8-1 will be proved first.

Theorem A8-1:

If code execution reaches “*check*” in Figure A8-1, then variable $x \in [a, b]$ provided that $a < b$.

Proof:

Initially, there is no constraints on variable x . When it is input, it can any real number
or

$$x \in (-\infty, +\infty) = (-\infty, a) \cup [a, b] \cup (b, +\infty)$$

where

$a < b$ is assumed;

$-\infty$ should be considered to be the most negative real number expressible
by a specific computer, not mathematically negative infinite;

$+\infty$ should be considered to be the most positive real number expressible
by a specific computer, not mathematically positive infinite.

We will prove the theorem case by case:

Case 1: $x \in (-\infty, a)$

- $\Rightarrow C_1.$ Yes (“then” branching when $x < a$)
- $\Rightarrow x := a$ (assignment of x , read as “ a is assigned to variable x ”)
- $\Rightarrow x = a$ (valuation of x , read as “the value of variable x is a ”)
- $\Rightarrow C_2.$ No (“else” branching when $x = a (< b)$)
- $\Rightarrow \text{check. } x = a$ (value of x at check is a)

Case 2: $x \in [a, b]$

- $\Rightarrow C_1.$ No (“else” branching when $x \geq a$)
- $\Rightarrow C_2.$ No (“else” branching when $x \leq b$)
- $\Rightarrow \text{check. } x \in [a, b]$ (range of x at check since x is not changed)

Case 3: $x \in (b, +\infty)$

- $\Rightarrow C_1.$ No (“else” branching when $x > b (> a)$)
- $\Rightarrow C_2.$ Yes (“then” branching when $x > b$)

$\Rightarrow x := b$	(assignment of x)
$\Rightarrow x = b$	(valuation of x)
$\Rightarrow \text{check}.x = b$	(value of x at check)

Therefore, combining three cases, it is concluded that at *check*, $x \in [a, b]$ provided that $a < b$. ■

Applying Theorem A8-1 to Function REACTR(P) in Figure A8-2 (1/2) where $a = 2031$ and $b = 7406$, we have, just before the third branching (i.e., $P \leq 2700?$),

$$P \in [2031, 7406]$$

That is to say, unless otherwise changed later in the code (which is not the case here), the value of P will be limited within the above range. Therefore, the last “else” branching (i.e., $P > 7500$) in Figure A8-2 (2/2) will never be executed.

```

GOTO 705
...
...
705 REACTR = -0.00333 * P + 30.0
RETURN

```

This analysis shows the benefit of flowchart in code analysis especially in tracing code and detecting logical errors in segments containing branching.

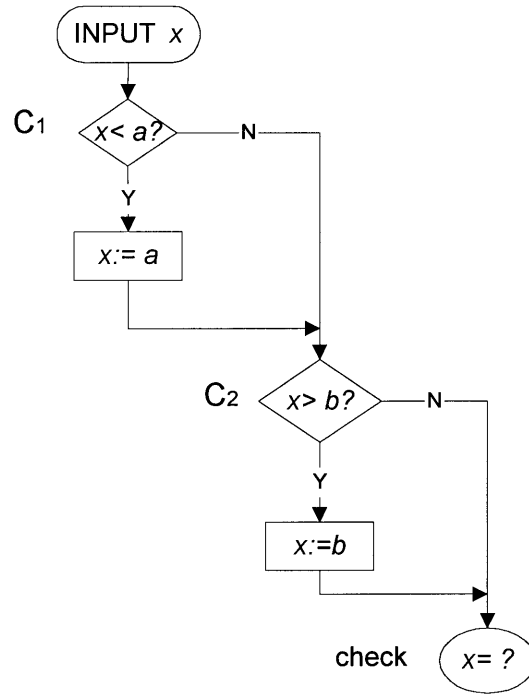


Figure A8-1 At “*check*” point, x is confined to $[a, b]$ if $a < b$.

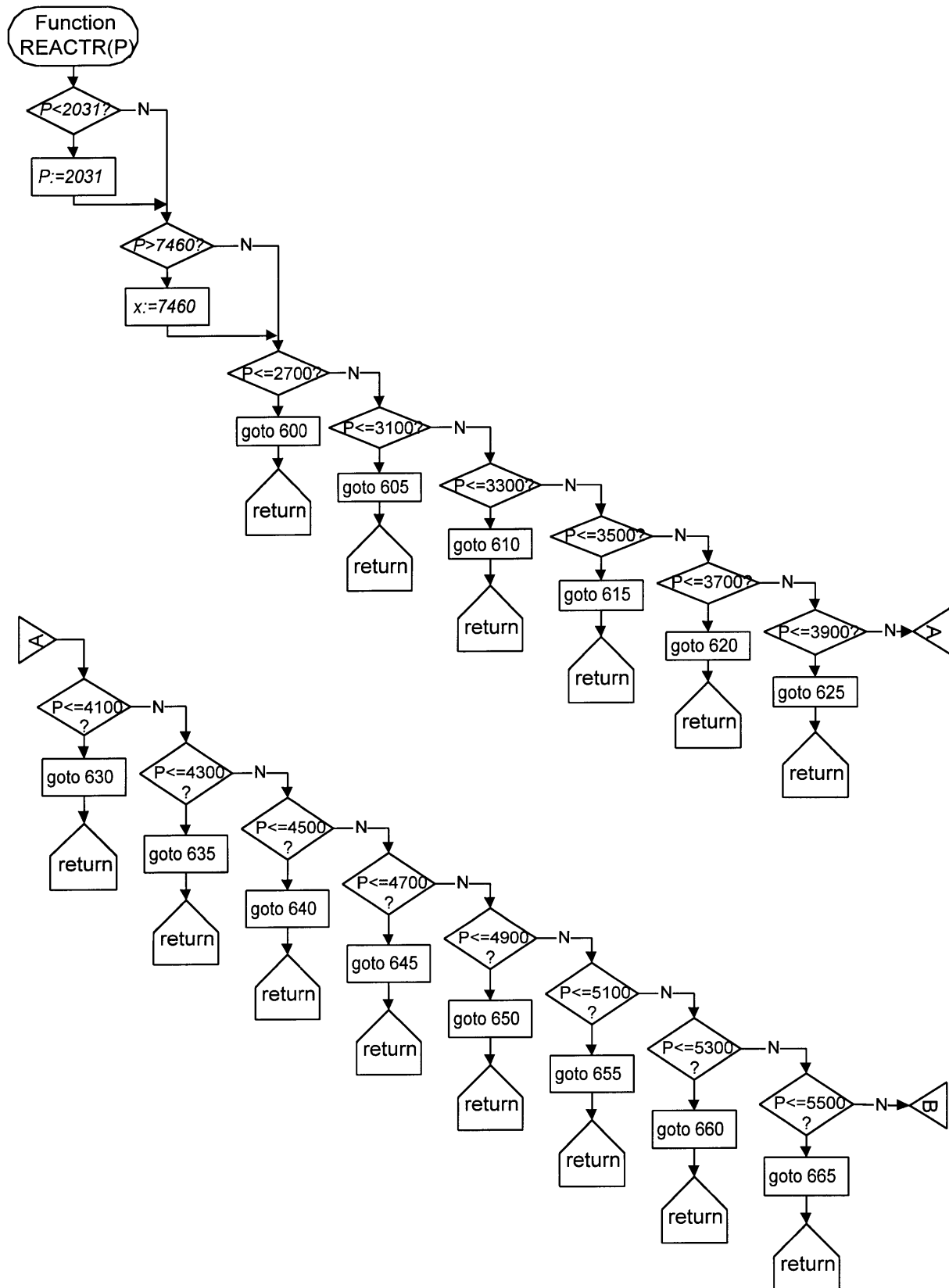
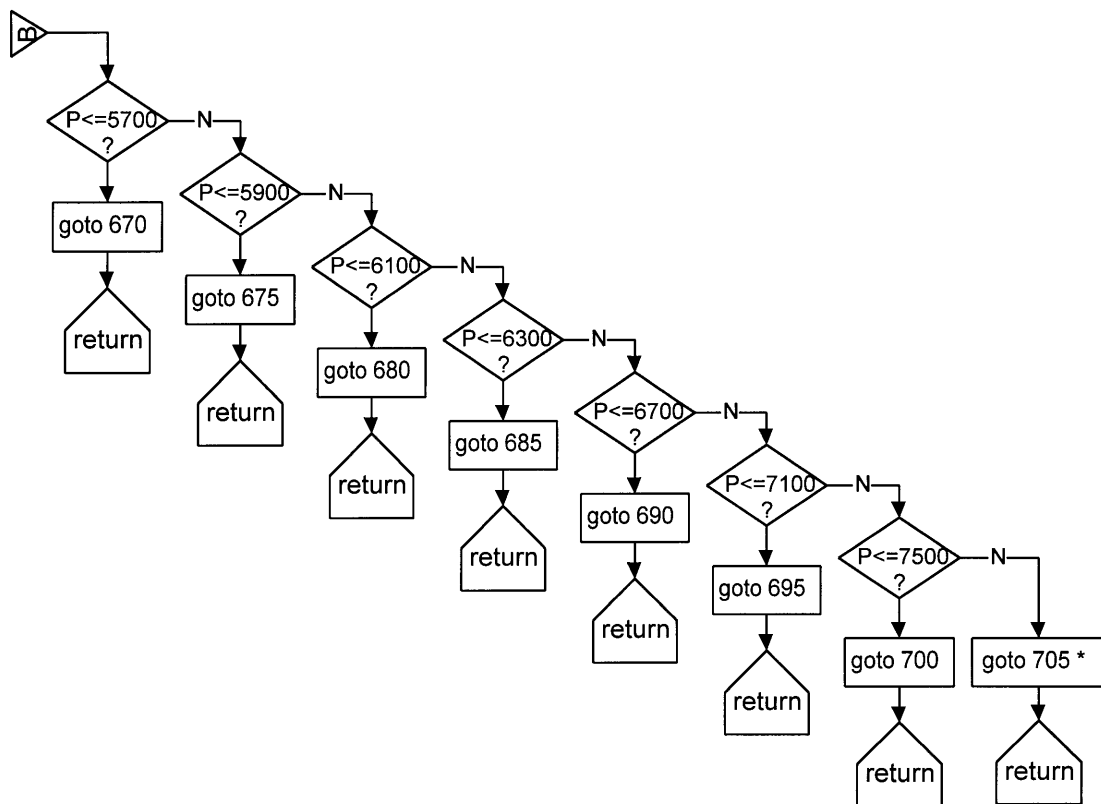


Figure A8.2 Code Analysis Using Flowchart (1/2)



- * *It can be mathematically proved that this branch will never be executed under any circumstances. Hence using flowchart, we find that the code contains an extraneous branch which will never be reached by code testing.*

Figure A8-2 Code Analysis Using Flowchart (2/2)

A8.3 Code Listing of Function REACTR(P)

```
FUNCTION REACTR(P)
C A FUNCTION WHICH RETURNS ACRR TRANSIENT ROD BANK WORTH IN CENTS
C GIVEN BANK POSITION IN POSITION UNITS.
C DATA FROM ACRR CURVES DATED 3 NOV.81
    IF(P .LT. 2031.) P=2031.
    IF(P .GT. 7460.) P=7460.
    IF(P .LE. 2700.) GO TO 600
    IF(P .LE. 3100.) GO TO 605
    IF(P .LE. 3300.) GO TO 610
    IF(P .LE. 3500.) GO TO 615
    IF(P .LE. 3700.) GO TO 620
    IF(P.LE.3900.)GOTO625
    IF(P.LE.4100.)GOTO630
    IF(P.LE.4300.)GOTO635
    IF(P.LE.4500.)GOTO640
    IF(P.LE.4700.)GOTO645
    IF(P.LE.4900.)GOTO650
    IF(P.LE.5100.)GOTO655
    IF(P.LE.5300.)GOTO660
    IF(P.LE.5500.)GOTO665
    IF(P.LE.5700.)GOTO670
    IF(P.LE.5900.)GOTO675
    IF(P.LE.6100.)GOTO680
    IF(P.LE.6300.)GOTO685
    IF(P.LE.6700.)GOTO690
    IF(P.LE.7100.)GOTO695
    IF(P.LE.7500.)GOTO700
    GOTO705
600    REACTR=-0.016143*P+472.985693
    RETURN
605    REACTR=-0.049250*P+562.375
```

```

        RETURN
610    REACTR=-0.0765*P+646.85
        RETURN
615    REACTR=-0.091*P+694.7
        RETURN
620    REACTR=-0.1005*P+727.9
        RETURN
625    REACTR=-0.1085*P+757.5
        RETURN
630    REACTR=-0.1225*P+812.15
        RETURN
635    REACTR=-0.125*P+822.4
        RETURN
640    REACTR=-0.1305*P+846.05
        RETURN
645    REACTR=-0.1295*P+841.55
        RETURN
650    REACTR=-0.1295*P+841.55
        RETURN
655    REACTR=-0.122*P+804.8
        RETURN
660    REACTR=-0.120*P+794.6
        RETURN
665    REACTR=-0.110*P+741.6
        RETURN
670    REACTR=-0.1065*P+722.35
        RETURN
675    REACTR=-0.0965*P+665.35
        RETURN
680    REACTR=-0.097*P+668.3
        RETURN
685    REACTR=-0.076*P+540.2

```

```
        RETURN
690    REACTR=-0.06575*P+475.625
        RETURN
695    REACTR=-0.04725*P+351.675
        RETURN
700    REACTR=-0.028*P+215.0
        RETURN
705    REACTR=-0.00333*P+30.0
        RETURN
        END
```


Appendix 9

Proofs of Theorems in Section 3.2.5

The majority of propositional logic theorems can be proved by using the truth tables. However, the truth tables are not always the best proof method. The method becomes tedious and may be error-prone when three or more logical variables are involved in a problem. The theorems listed here are either the author's original contribution or from other literature. All the proofs provided here are contributed by the author.

Theorem 3.2-1A:

$P \Rightarrow P \vee Q$ is a tautology.

Proof:

$P \Rightarrow P \vee Q$ is equivalent to $\neg P \vee (P \vee Q)$, which is equivalent to $(\neg P \vee P) \vee Q$ or *true*. Hence, $P \Rightarrow P \vee Q$ is a tautology.

■

Theorem 3.2-1B:

$P \wedge Q \Rightarrow P$ is a tautology.

Proof:

$P \wedge Q \Rightarrow P$ is equivalent to $\neg(P \wedge Q) \vee P$, which is equivalent to $(\neg P \vee \neg Q) \vee P$ or $\neg P \vee P \vee Q$ which is *true*. Hence, $P \wedge Q \Rightarrow P$ is a tautology.

■

Corollary 3.2-1:

$P \wedge Q \Rightarrow P \vee Q$ is a tautology.

Proof:

It follows from Theorems 3.2-1A and 3.2-1B using Theorem 3.2-3 which is proved independently by truth table. ■

The two theorems and the corollary indicates that $P \wedge Q$ is a sufficient condition for P (or Q), which is a sufficient condition for $P \vee Q$.

Theorem 3.2-2A:

If $P \Rightarrow Q$, then $P \wedge R \Rightarrow Q \wedge R$.

Proof:

$$P \Rightarrow Q \Leftrightarrow \neg P \vee Q \quad (\text{Definition of } \Rightarrow) \quad (3.2-1)$$

$$\begin{aligned} P \wedge R \Rightarrow Q \wedge R &\Leftrightarrow \neg(P \wedge R) \vee (Q \wedge R) && (\text{Definition of } \Rightarrow) \\ &\Leftrightarrow \neg P \vee \neg R \vee (Q \wedge R) && (\text{De Morgan's Law}) \\ &\Leftrightarrow \neg P \vee [\neg R \vee (Q \wedge R)] && (\text{Associative Law}) \\ &\Leftrightarrow \neg P \vee \neg R \vee Q && (\text{Distributive Law}) \end{aligned}$$

According to Corollary 3.2-2B,

$$P \wedge R \Rightarrow Q \wedge R \Leftrightarrow (\neg P \vee Q) \vee \neg R \quad (3.2-2)$$

Applying Theorem 3.2-1A to Equations (3.2-1) and (3.2-2), i.e., substituting $\neg P \vee Q$ for P and $\neg R$ for Q in the theorem respectively, yields $(3.2-1) \Rightarrow (3.2-2)$. Hence, Theorem 3.2-2A is true. ■

Theorem 3.2-2B:

If $P \Rightarrow Q$, then $P \vee R \Rightarrow Q \vee R$.

Proof:

$$P \Rightarrow Q \Leftrightarrow \neg P \vee Q \quad (\text{Definition of } \Rightarrow) \quad (3.2-3)$$

$$\begin{aligned}
P \vee R \Rightarrow Q \vee R &\Leftrightarrow \neg(P \vee R) \vee (Q \vee R) && \text{(Definition of } \Rightarrow \text{)} \\
&\Leftrightarrow (\neg P \wedge \neg R) \vee (Q \vee R) && \text{(De Morgan's Law)} \\
&\Leftrightarrow (\neg P \wedge \neg R) \vee R \vee Q && \text{(Commutative Law)} \\
&\Leftrightarrow \neg P \vee R \vee Q && \text{(Distributive Law)}
\end{aligned}$$

According to Corollary 3.2-2B

$$P \vee R \Rightarrow Q \vee R \Leftrightarrow (\neg P \vee Q) \vee R \quad (3.2-4)$$

Applying Theorem 3.2-1A to Equations (3.2-3) and (3.2-4), i.e., substituting $\neg P \vee Q$ for P and R for Q in the theorem respectively, yields $(3.2-3) \Rightarrow (3.2-4)$. Hence, Theorem 3.2-2B is true. ■

Theorem 3.2-3 (Syllogism):

If $P \Rightarrow Q$ and $Q \Rightarrow R$, then $P \Rightarrow R$.

Proof by truth table:

P	Q	R	$P \Rightarrow Q$	$Q \Rightarrow R$	$P \Rightarrow Q \wedge Q \Rightarrow R$	$P \Rightarrow R$	$(P \Rightarrow Q \wedge Q \Rightarrow R) \Rightarrow (P \Rightarrow R)$
T	T	T	T	T	T	T	T
T	T	F	T	F	F	F	T
T	F	T	F	T	F	T	T
T	F	F	F	T	F	F	T
F	T	T	T	T	T	T	T
F	T	F	T	F	F	T	T
F	F	T	T	T	T	T	T
F	F	F	T	T	T	T	T

Corollary 3.2-2A: (Theorem 3.2-3 can be extended to n propositions.)

If $P_1 \Rightarrow P_2 \wedge P_2 \Rightarrow P_3 \wedge P_3 \Rightarrow P_4 \wedge \dots \wedge P_{n-1} \Rightarrow P_n$, then $P_1 \Rightarrow P_n$.

Corollary 3.2-2B:

If $P \Leftrightarrow Q \wedge Q \Leftrightarrow R$, then $P \Leftrightarrow R$.

Proof:

$$\begin{aligned} (P \Leftrightarrow Q \wedge Q \Leftrightarrow R) &\Leftrightarrow [(P \Rightarrow Q \wedge Q \Rightarrow R) \wedge (R \Rightarrow Q \wedge Q \Rightarrow P)] && \text{(Definition of } \Leftrightarrow) \\ &\Rightarrow (P \Rightarrow R \wedge R \Rightarrow P) && \text{(Theorem 3.2 - 3)} \\ &\Rightarrow (P \Leftrightarrow R) && \text{(Definition of } \Leftrightarrow) \end{aligned}$$

■

Theorem 3.2-4A:

If $P \Rightarrow Q$ and $S \Rightarrow T$, then $P \wedge S \Rightarrow Q \wedge T$.

Proof:

Applying Theorem 3.2-2A to $P \Rightarrow Q$ and $S \Rightarrow T$ yields, respectively

$$P \wedge S \Rightarrow Q \wedge S \tag{3.2-5}$$

$$Q \wedge S \Rightarrow Q \wedge T \tag{3.2-6}$$

Applying Theorem 3.2-3 to Equations (3.2-5) and (3.2-6) yields

$$P \wedge S \Rightarrow Q \wedge T$$

■

Theorem 3.2-4B:

If $P \Rightarrow Q$ and $S \Rightarrow T$, then $P \vee S \Rightarrow Q \vee T$.

Proof:

Applying Theorem 3.2-2B to $P \Rightarrow Q$ and $S \Rightarrow T$ yields, respectively

$$P \vee S \Rightarrow Q \vee S \tag{3.2-7}$$

$$Q \vee S \Rightarrow Q \vee T \tag{3.2-8}$$

Applying Theorem 3.2-3 to Equations (3.2-7) and (3.2-8) yields

$$P \vee S \Rightarrow Q \vee T$$

■

Theorem 3.2-5:

$P \Rightarrow P \wedge Q$ is equivalent to $P \Rightarrow Q$. Mathematically,
 $(P \Rightarrow Q) \Leftrightarrow (P \Rightarrow P \wedge Q)$.

Proof:

$$\begin{aligned} (P \Rightarrow P \wedge Q) &\Leftrightarrow \neg P \vee (P \wedge Q) && \text{(Definition of } \Rightarrow \text{)} \\ &\Leftrightarrow (\neg P \vee P) \wedge (\neg P \vee Q) && \text{(Distributive Law)} \\ &\Leftrightarrow \text{true} \wedge (\neg P \vee Q) && \text{(Completeness Law)} \\ &\Leftrightarrow \neg P \vee Q && \text{(Identity Law)} \\ &\Leftrightarrow P \Rightarrow Q && \text{(Definition of } \Rightarrow \text{)} \end{aligned}$$

■

Theorem 3.2-6A:

$$P \wedge (P \vee Q) \Leftrightarrow P$$

Proof by truth table:

P	Q	$P \vee Q$	$P \wedge (P \vee Q)$	$P \wedge (P \vee Q) \Leftrightarrow P$
T	T	T	T	T
T	F	T	T	T
F	T	T	F	T
F	F	F	F	T

■

Theorem 3.2-6B:

$$P \vee (P \wedge Q) \Leftrightarrow P$$

Proof:

$$\begin{aligned} P \vee (P \wedge Q) &\Leftrightarrow (P \wedge \text{true}) \vee (P \wedge Q) && \text{(Identity Law)} \\ &\Leftrightarrow P \wedge (\text{true} \vee Q) && \text{(Distribution Law)} \end{aligned}$$

$$\Leftrightarrow P \wedge \text{true} \quad (\text{Domination Law})$$

$$\Leftrightarrow P \quad (\text{Identity Law})$$

■

Theorem 3.2-7A:

$$(P \Rightarrow Q \wedge R) \Leftrightarrow (P \Rightarrow Q \wedge P \Rightarrow R).$$

Proof:

$$(P \Rightarrow Q \wedge R) \Leftrightarrow \neg P \vee (Q \wedge R) \quad (\text{Definition of } \Rightarrow)$$

$$\Leftrightarrow (\neg P \vee Q) \wedge (\neg P \vee R) \quad (\text{Distributive Law})$$

$$\Leftrightarrow (P \Rightarrow Q) \wedge (P \Rightarrow R) \quad (\text{Definition of } \Rightarrow)$$

Applying Corollary 3.2-2B yields

$$(P \Rightarrow Q \wedge R) \Leftrightarrow (P \Rightarrow Q \wedge P \Rightarrow R).$$

■

Theorem 3.2-7B:

$$(P \Rightarrow Q \vee R) \Leftrightarrow (P \Rightarrow Q \vee P \Rightarrow R).$$

Proof:

$$(P \Rightarrow Q \vee R) \Leftrightarrow \neg P \vee (Q \vee R) \quad (\text{Definition of } \Rightarrow)$$

$$\Leftrightarrow \neg P \vee \neg P \vee Q \vee R \quad (\text{Idempotent Law})$$

$$\Leftrightarrow \neg P \vee Q \vee \neg P \vee R \quad (\text{Commutative Law})$$

$$\Leftrightarrow (\neg P \vee Q) \vee (\neg P \vee R) \quad (\text{Associative Law})$$

$$\Leftrightarrow (P \Rightarrow Q \vee P \Rightarrow R) \quad (\text{Definition of } \Rightarrow)$$

■

Theorem 3.2-7C:

$$(P \wedge Q \Rightarrow R) \Leftrightarrow (P \Rightarrow R \vee Q \Rightarrow R).$$

Proof:

$$\begin{aligned}
(P \wedge Q \Rightarrow R) &\Leftrightarrow \neg(P \wedge Q) \vee R && \text{(Definition of } \Rightarrow) \\
&\Leftrightarrow \neg P \vee \neg Q \vee R && \text{(De Morgan's Law)} \\
&\Leftrightarrow P \vee \neg Q \vee R \vee R && \text{(Idempotent Law)} \\
&\Leftrightarrow \neg P \vee R \vee \neg Q \vee R && \text{(Commutative Law)} \\
&\Leftrightarrow (\neg P \vee R) \vee (\neg Q \vee R) && \text{(Associative Law)} \\
&\Leftrightarrow (P \Rightarrow R \vee Q \Rightarrow R). && \text{(Definition of } \Rightarrow)
\end{aligned}$$

■

Theorem 3.2-7D:

$$(P \vee Q \Rightarrow R) \Leftrightarrow (P \Rightarrow R \wedge Q \Rightarrow R).$$

Proof:

$$\begin{aligned}
(P \vee Q \Rightarrow R) &\Leftrightarrow \neg(P \vee Q) \vee R && \text{(Definition of } \Rightarrow) \\
&\Leftrightarrow (\neg P \wedge \neg Q) \vee R && \text{(De Morgan's Law)} \\
&\Leftrightarrow (\neg P \vee R) \wedge (\neg Q \vee R) && \text{(Distributive Law)} \\
&\Leftrightarrow (P \Rightarrow R \wedge Q \Rightarrow R) && \text{(Definition of } \Rightarrow).
\end{aligned}$$

■

Theorem 3.2-8:

$$(P \Rightarrow R \wedge Q \Rightarrow R) \Rightarrow (P \wedge Q \Rightarrow R).$$

Proof:

It follows directly from Theorem 3.2-4A.

■

Caveat: The converse of Theorem 3.2-8 or $(P \wedge Q \Rightarrow R) \Rightarrow (P \Rightarrow R \wedge Q \Rightarrow R)$ is not true. As counterexamples, consider the triples $(P, Q, R) = (\text{true}, \text{false}, \text{false})$ or $(P, Q, R) = (\text{false}, \text{true}, \text{false})$.

Theorem 3.2-9: (Elimination of the Consensus¹⁷)

$$(P \wedge Q) \vee (\neg P \wedge R) \vee (Q \wedge R) \Leftrightarrow (P \wedge Q) \vee (\neg P \wedge R).$$

¹⁷ This theorem is taken from [Quin55] in which no proof is given. Here, a proof is provided by the author independently.

Proof:

$$\begin{aligned}
 & (P \wedge Q) \vee (\neg P \wedge R) \vee (Q \wedge R) \\
 \Leftrightarrow & (P \wedge Q) \vee (\neg P \wedge R) \vee [(Q \wedge R) \wedge \text{true}] && \text{(Identity Law)} \\
 \Leftrightarrow & (P \wedge Q) \vee (\neg P \wedge R) \vee [(Q \wedge R) \wedge (P \vee \neg P)] && \text{(Completeness Law)} \\
 \Leftrightarrow & (P \wedge Q) \vee (\neg P \wedge R) \vee [(Q \wedge R \wedge P) \vee (Q \wedge R \wedge \neg P)] && \text{(Distributive Law)} \\
 \Leftrightarrow & (P \wedge Q) \vee (P \wedge Q \wedge R) \vee (\neg P \wedge R) \vee (\neg P \wedge R \wedge Q) && \text{(Commutative Law)} \\
 \Leftrightarrow & [(P \wedge Q) \vee (P \wedge Q \wedge R)] \vee [(\neg P \wedge R) \vee (\neg P \wedge R \wedge Q)] && \text{(Associative Law)} \\
 \Leftrightarrow & (P \wedge Q) \vee (\neg P \wedge R). && \text{(Theorem 3.2 - 6B)}
 \end{aligned}$$

■

Theorem 3.2-10:

$$P \wedge Q \Rightarrow (P \Rightarrow Q).$$

Proof by truth table:

P	Q	$P \wedge Q$	$P \Rightarrow Q$	$P \wedge Q \Rightarrow (P \Rightarrow Q)$
F	F	F	T	T
F	T	F	T	T
T	F	F	F	T
T	T	T	T	T

(F = false, T = true)

■

Corollary 3.2-3:

$$P \wedge Q \Rightarrow (P \Leftrightarrow Q).$$

Proof:

$$\begin{aligned}
 P \wedge Q & \Rightarrow (P \Rightarrow Q) && \text{(Theorem 3.2-10)} \\
 P \wedge Q & \Rightarrow (Q \Rightarrow P) && \text{(Theorem 3.2-10)} \\
 P \wedge Q & \Rightarrow [(P \Rightarrow Q) \wedge (Q \Rightarrow P)] && \text{(Theorem 3.2-4A)} \\
 P \wedge Q & \Rightarrow (P \Leftrightarrow Q) && \text{(Definition of } \Leftrightarrow \text{)}
 \end{aligned}$$

■

Lemma 3.2-1:

$$P \Rightarrow (true \Rightarrow Q) \Leftrightarrow P \Rightarrow Q$$

Proof:

$$\begin{aligned} P \Rightarrow (true \Rightarrow Q) &\Leftrightarrow [P \Rightarrow (\neg true \vee Q)] && \text{(Definition of } \Rightarrow \text{)} \\ &\Leftrightarrow [P \Rightarrow Q] && \text{(Identity Law)} \end{aligned}$$

■

Theorem 3.2-11:

$$(P \Rightarrow Q) \wedge (\neg P \Rightarrow R) \Rightarrow (Q \vee R)$$

Proof:

$$[(P \Rightarrow Q) \wedge (\neg P \Rightarrow R)] \Rightarrow [(P \vee \neg P) \Rightarrow (Q \vee R)] \quad \text{(Theorem 3.2-4B)}$$

or

$$\begin{aligned} [(P \Rightarrow Q) \wedge (\neg P \Rightarrow R)] &\Rightarrow [true \Rightarrow (Q \vee R)] && \text{(Completeness Law)} \\ (P \Rightarrow Q) \wedge (\neg P \Rightarrow R) &\Rightarrow (Q \vee R) && \text{(Lemma 3.2-1)} \end{aligned}$$

■

Theorem 3.2-12:

$$(P \Leftrightarrow Q) \Leftrightarrow (\neg P \Leftrightarrow \neg Q)$$

Proof:

$$\begin{aligned} (P \Leftrightarrow Q) &\Leftrightarrow (P \Rightarrow Q \wedge Q \Rightarrow P) && \text{(Definition of } \Leftrightarrow \text{)} \\ &\Leftrightarrow (\neg Q \Rightarrow \neg P) \wedge (\neg P \Rightarrow \neg Q) && \text{(Contrapositive)} \\ &\Leftrightarrow (\neg P \Leftrightarrow \neg Q) && \text{(Definition of } \Leftrightarrow \text{)} \end{aligned}$$

■

Theorem 3.2-13:

$$(P \Rightarrow Q \wedge Q \Rightarrow R \wedge R \Rightarrow P) \Leftrightarrow (P \Leftrightarrow Q \wedge Q \Leftrightarrow R)$$

Proof:

“ \Rightarrow ”

$$\begin{aligned} &(P \Rightarrow Q \wedge Q \Rightarrow R \wedge R \Rightarrow P) \\ &\Leftrightarrow (P \Rightarrow Q \wedge Q \Rightarrow R \wedge R \Rightarrow P \wedge Q \Rightarrow R \wedge R \Rightarrow P \wedge P \Rightarrow Q) && \text{(Idempotent Law)} \end{aligned}$$

$$\Leftrightarrow P \Rightarrow Q \wedge (Q \Rightarrow R \wedge R \Rightarrow P) \wedge Q \Rightarrow R \wedge (R \Rightarrow P \wedge P \Rightarrow Q) \quad (\text{Associative Law})$$

$$\Leftrightarrow (P \Rightarrow Q \wedge Q \Rightarrow P) \wedge (Q \Rightarrow R \wedge R \Rightarrow Q) \quad (\text{Theorem 3.2-3})$$

$$\Rightarrow (P \Leftrightarrow Q \wedge Q \Leftrightarrow R) \quad (\text{Definition of } \Leftrightarrow)$$

“ \Leftarrow ”

$$(P \Leftrightarrow Q \wedge Q \Leftrightarrow R)$$

$$\Leftrightarrow (P \Rightarrow Q \wedge Q \Rightarrow P) \wedge (Q \Rightarrow R \wedge R \Rightarrow Q) \quad (\text{Definition of } \Leftrightarrow)$$

$$\Leftrightarrow (P \Rightarrow Q \wedge Q \Rightarrow R \wedge R \Rightarrow Q \wedge Q \Rightarrow P) \quad (\text{Commutative Law})$$

$$\Leftrightarrow P \Rightarrow Q \wedge Q \Rightarrow R \wedge (R \Rightarrow Q \wedge Q \Rightarrow P) \quad (\text{Associative Law})$$

$$\Leftrightarrow (P \Rightarrow Q \wedge Q \Rightarrow R \wedge R \Rightarrow P) \quad (\text{Theorem 3.2-3})$$

■

Theorem 3.2-13:

If $P \Rightarrow Q_1 \wedge Q_2$ and $\neg Q_1$, then $\neg P$.

Proof:

$$(P \Rightarrow Q_1 \wedge Q_2) \wedge \neg Q_1$$

$$\Leftrightarrow [\neg P \vee (Q_1 \wedge Q_2)] \wedge \neg Q_1 \quad (\text{Definition of } \Rightarrow)$$

$$\Leftrightarrow (\neg P \wedge \neg Q_1) \vee [(Q_1 \wedge Q_2) \wedge \neg Q_1] \quad (\text{Distributive Law})$$

$$\Leftrightarrow (\neg P \wedge \neg Q_1) \vee [(Q_1 \wedge \neg Q_1) \wedge Q_2] \quad (\text{Commutative Law})$$

$$\Leftrightarrow (\neg P \wedge \neg Q_1) \vee (false \wedge \neg Q_1) \quad (\text{Inconsistency Law})$$

$$\Leftrightarrow (\neg P \wedge \neg Q_1) \vee false \quad (\text{Identity Law})$$

$$\Leftrightarrow (\neg P \wedge \neg Q_1) \quad (\text{Identity Law})$$

$$\Rightarrow \neg P \quad (\text{Theorem 3.2-B})$$

■

Appendix 10

A Between-Group Analysis: Reactor Period–Rod Velocity

The following between-group analysis is the cut-and-paste result from running a program in MathCAD.

Transient Rod Worth (based on Appendix A8.3)

$$x = 2000, 2001 \dots 8000$$

$$\rho_0(x) = -0.016143 \cdot x + 472.985693$$

$$\rho_1(x) = \text{if}(x < 2031, \rho_0(2031), \rho_0(x))$$

$$\rho_2(x) = \text{if}(x < 2700, \rho_1(x), -0.049250 \cdot x + 562.375)$$

$$\rho_3(x) = \text{if}(x < 3100, \rho_2(x), -0.0765 \cdot x + 646.85)$$

$$\rho_4(x) = \text{if}(x < 3300, \rho_3(x), -0.091 \cdot x + 694.7)$$

$$\rho_5(x) = \text{if}(x < 3500, \rho_4(x), -0.1005 \cdot x + 727.9)$$

$$\rho_6(x) = \text{if}(x < 3700, \rho_5(x), -0.1085 \cdot x + 757.5)$$

$$\rho_7(x) = \text{if}(x < 3900, \rho_6(x), -0.1225 \cdot x + 812.15)$$

$$\rho_8(x) = \text{if}(x < 4100, \rho_7(x), -0.125 \cdot x + 822.4)$$

$$\rho_9(x) = \text{if}(x < 4300, \rho_8(x), -0.1305 \cdot x + 846.05)$$

$$\rho_{10}(x) = \text{if}(x < 4500, \rho_9(x), -0.1295 \cdot x + 841.55)$$

$$\rho_{11}(x) = \text{if}(x < 4700, \rho_{10}(x), -0.1295 \cdot x + 841.55)$$

$$\rho_{12}(x) = \text{if}(x < 4900, \rho_{11}(x), -0.122 \cdot x + 804.8)$$

$$\rho_{13}(x) = \text{if}(x < 5100, \rho_{12}(x), -0.120 \cdot x + 794.6)$$

$$\rho_{14}(x) = \text{if}(x < 5300, \rho_{13}(x), -0.110 \cdot x + 741.6)$$

$$\rho_{15}(x) = \text{if}(x < 5500, \rho_{14}(x), -0.1065 \cdot x + 722.35)$$

$$\rho_{16}(x) = \text{if}(x < 5700, \rho_{15}(x), -0.0965 \cdot x + 665.35)$$

$$\rho_{17}(x) = \text{if}(x < 5900, \rho_{16}(x), -0.097 \cdot x + 668.3)$$

$$\rho_{18}(x) = \text{if}(x < 6100, \rho_{17}(x), -0.076 \cdot x + 540.2)$$

$$\rho_{19}(x) = \text{if}(x < 6300, \rho_{18}(x), -0.06575 \cdot x + 475.625)$$

$$\rho_{20}(x) = \text{if}(x < 6700, \rho_{19}(x), -0.04725 \cdot x + 351.675)$$

$$\rho_{21}(x) = \text{if}(x < 7100, \rho_{20}(x), -0.028 \cdot x + 215.0)$$

$$\rho(x) = \text{if}(x < 7460, \rho_{21}(x), \rho_{21}(7460))$$

$$n = 23$$

$$x_{\min} = 2000$$

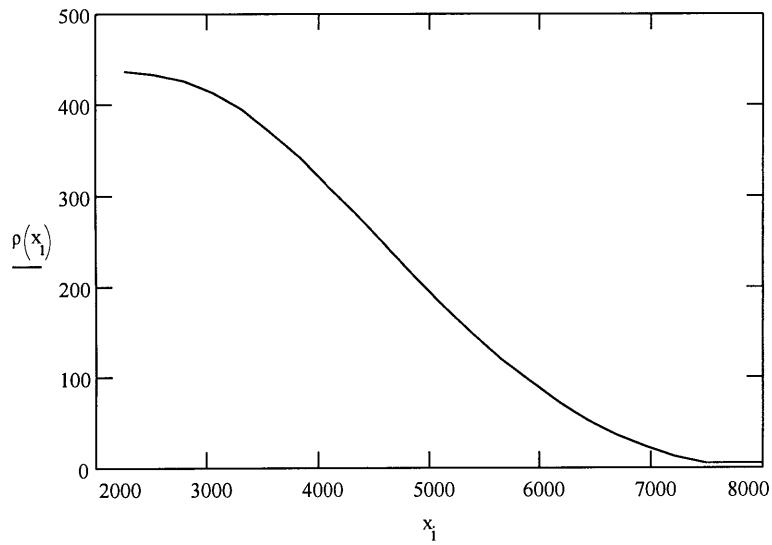
$$x_{\max} = 8000$$

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

$$dx = \frac{x_{\max} - x_{\min}}{n}$$

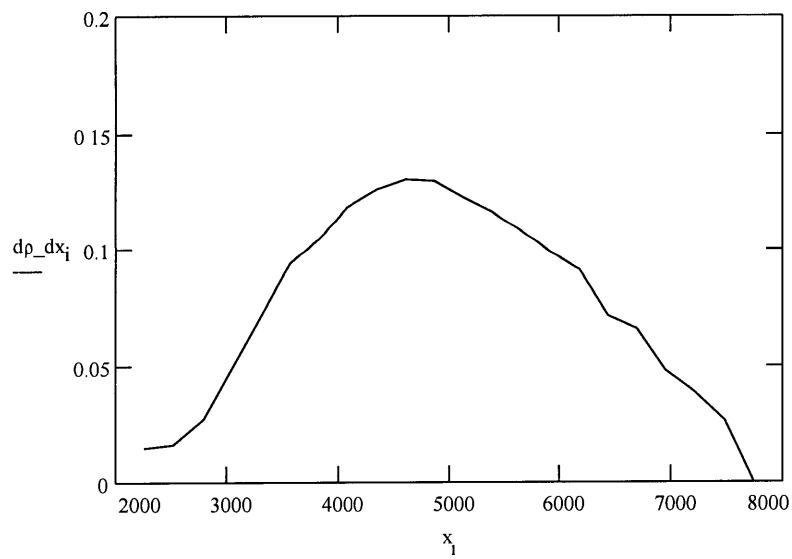
$$x_i = x_{\min} + i \cdot dx$$

Integral Rod Worth, in *cents*



Differential rod worth, in *cents/unit*

$$d\rho_{dx_i} = \frac{-(\rho(x_i) - \rho(x_{i-1}))}{dx}$$



1. Find asymptotic reactor period that raises reactor power by 10^6 in 5 seconds. (1.05 is a correcting factor dealing with overshoot)

$T_1 = 5.0$ seconds is the duration of power increase

$$T_1 = 5.0$$

$$\text{Power_ratio} = 10^6$$

$$f(\tau) = \left(\frac{T_1}{\tau} \right) - \ln(\text{Power_ratio})$$

Initial guess of period, *seconds*

$$\tau = .00250$$

Solve for specified (or required) period, *seconds*

$$\tau = \text{root}(f(\tau), \tau)$$

$$\omega_1 = \frac{1}{\tau}$$

$$\tau = 0.36$$

$$\omega_1 = 2.76$$

$T_2 = 10.0$ seconds is the end of the duration the power is maintained at target level

$$T_2 = 10.0$$

$$N = 2000$$

$$i = 0..N$$

$$\beta = 0.0078$$

$$\omega_2 = 0.0$$

$$\Delta t = \frac{T_2}{N}$$

2. Find maximum allowed initial power

Initial guess, *watts*

$$P_0 = 10.0$$

Solving for maximum allowed initial power, *watts*

$$P(P_0, t) = P_0 \cdot \text{if} \left[t \leq T_1, \exp(\omega_1 \cdot t), \exp(\omega_1 \cdot T_1) \cdot \exp[\omega_2 \cdot (t - T_1)] \right]$$

$$E(P_0) = \int_0^{T_2} P(P_0, t) \, dt$$

$$P_0 = \text{root}(E(P_0) \cdot 1.05 - 500 \cdot 10^6, P_0)$$

$$P_0 = 88.79$$

3. Solve for normalized reactivity, rate of change of reactivity, and inverse of reactor period (MIT-SNL Control Laws)

Initial conditions

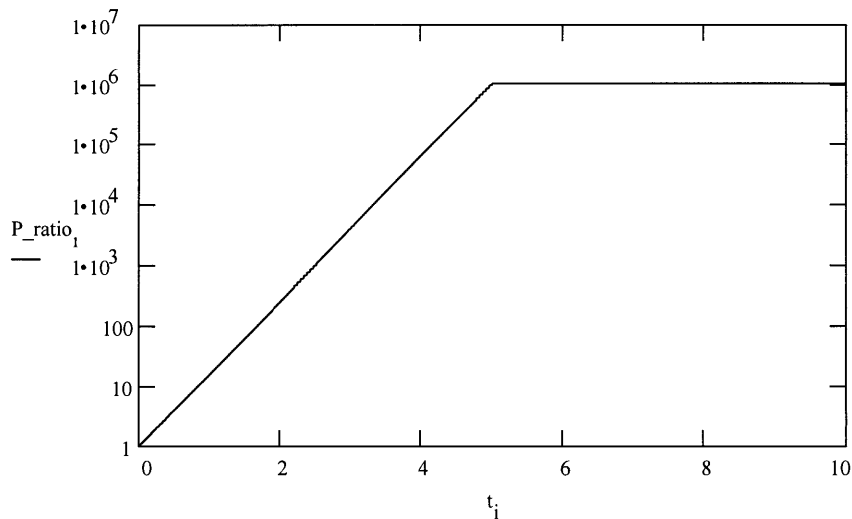
$$\begin{bmatrix} \lambda_0 \\ \rho_0 \\ t_0 \\ P_ratio_0 \end{bmatrix} = \begin{bmatrix} 0. \\ 0. \\ 0. \\ 1.0 \end{bmatrix}$$

Differential Equations

- Decay constant λ is obtained by curve fitting of Figure 5.3-1 of [Bern89]
- To simplify the computation, terms involving prompt neutron lifetime or rate of decay constant in Equation (5.1-4) of the dissertation are ignored.

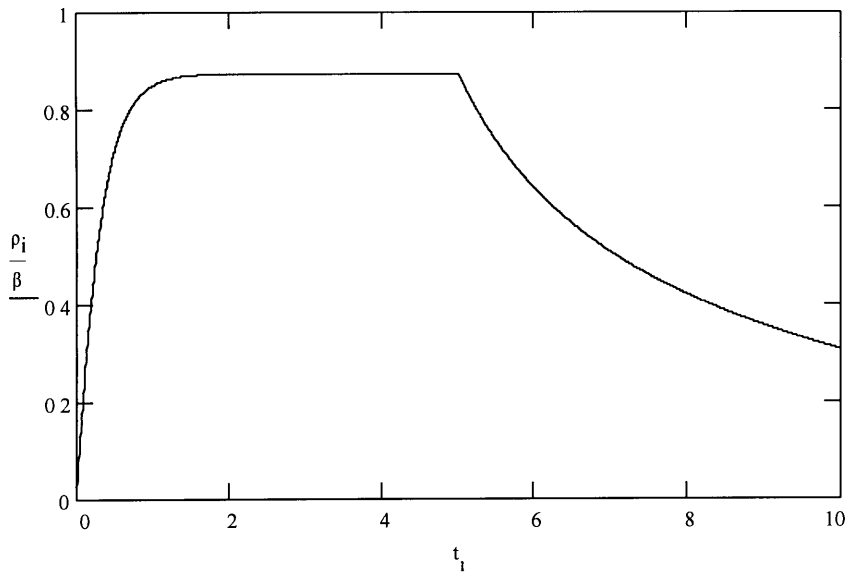
$$\begin{bmatrix} \lambda_{i+1} \\ \rho_{i+1} \\ t_{i+1} \\ P_ratio_i \end{bmatrix} = \begin{bmatrix} 0.079997 \cdot \exp\left(1.8334 \cdot \frac{\rho_i}{\beta}\right) \\ \rho_i + \Delta t \left[\left(\beta - \rho_i \right) \cdot \text{if} \left(i \leq \frac{N}{2}, \omega_1, \omega_2 \right) - \lambda_i \cdot \rho_i \right] \\ \Delta t \cdot (i + 1) \\ \text{if} \left[i \leq \frac{N}{2}, \exp(\omega_1 \cdot t_i), \exp\left(\omega_1 \cdot \frac{t_N}{2}\right) \cdot \exp\left[\omega_2 \cdot \left(t_i - \frac{t_N}{2}\right)\right] \right] \end{bmatrix}$$

Transient reactor power ratio



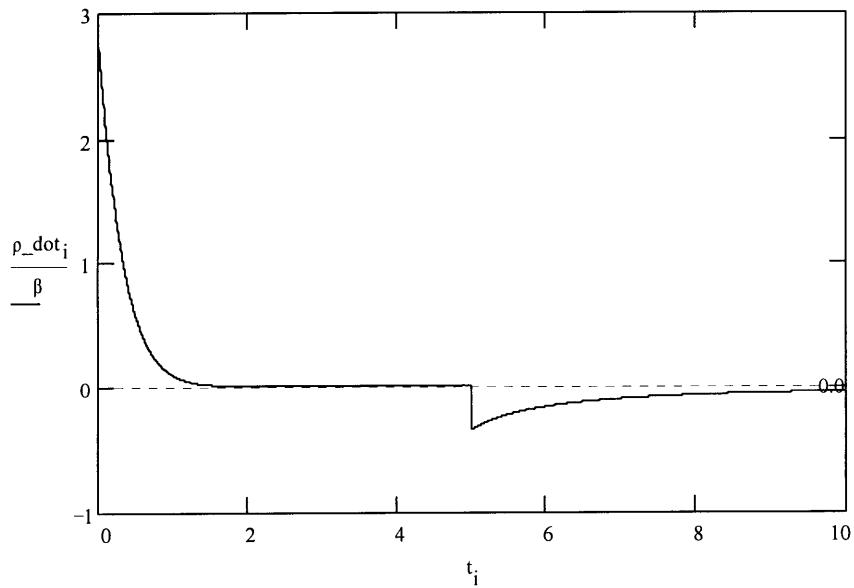
Normalized reactivity, in \$

(The transient rods have enough reactivity to compensate.)



Rate of change of reactivity, in \$/second

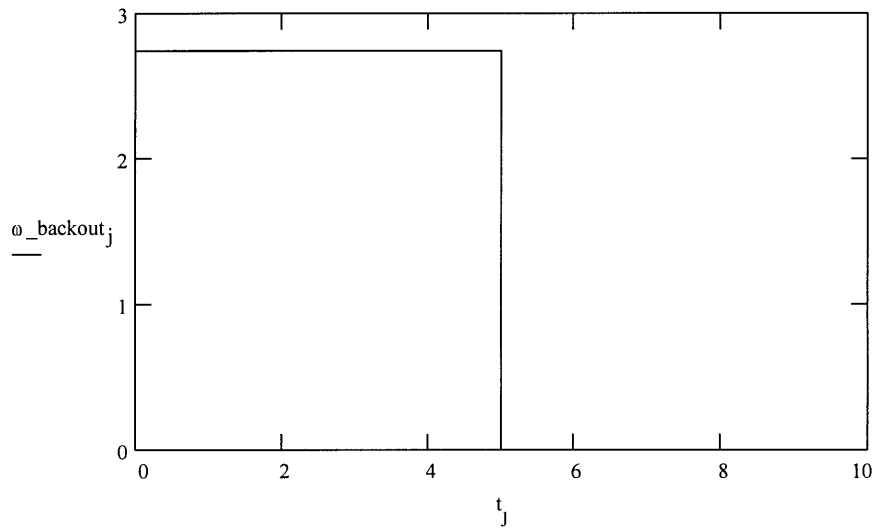
$$\rho_dot_{i+1} = \frac{\rho_{i+1} - \rho_i}{\Delta t}$$



Corresponding inverse of reactor period backed from the power profile, in *second*⁻¹

$$j = 1..N - 1$$

$$\omega_{\text{backout}_j} = \frac{1 - \frac{P_{\text{ratio}_{j-1}}}{P_{\text{ratio}_j}}}{\Delta t}$$



4. Find required transient rod velocity and position

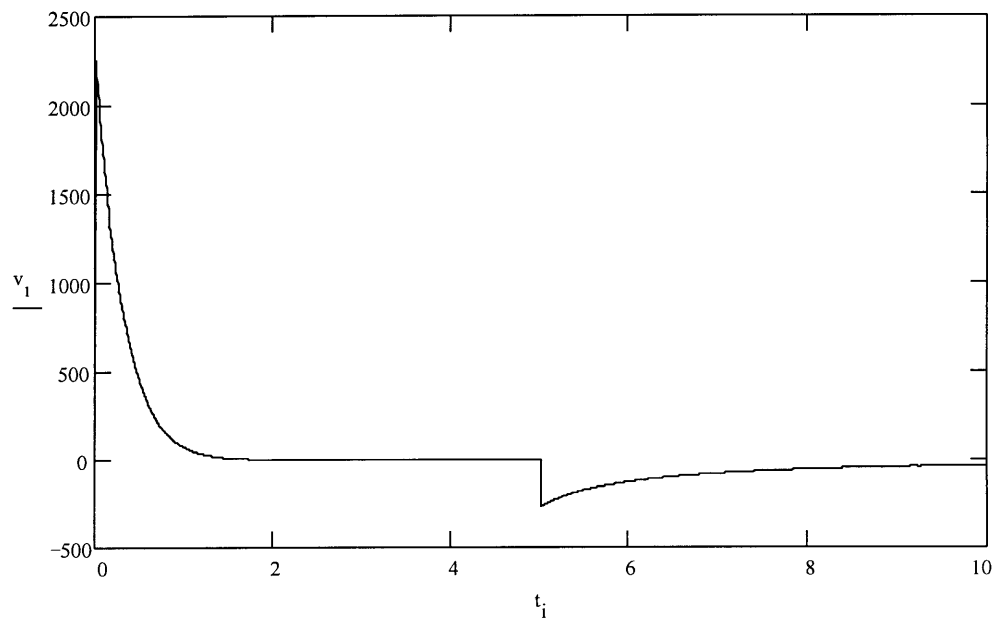
Initial condition: $x_{\text{init}}=4062$ units is a initial rod position

$x_{\text{init}} = 4062$

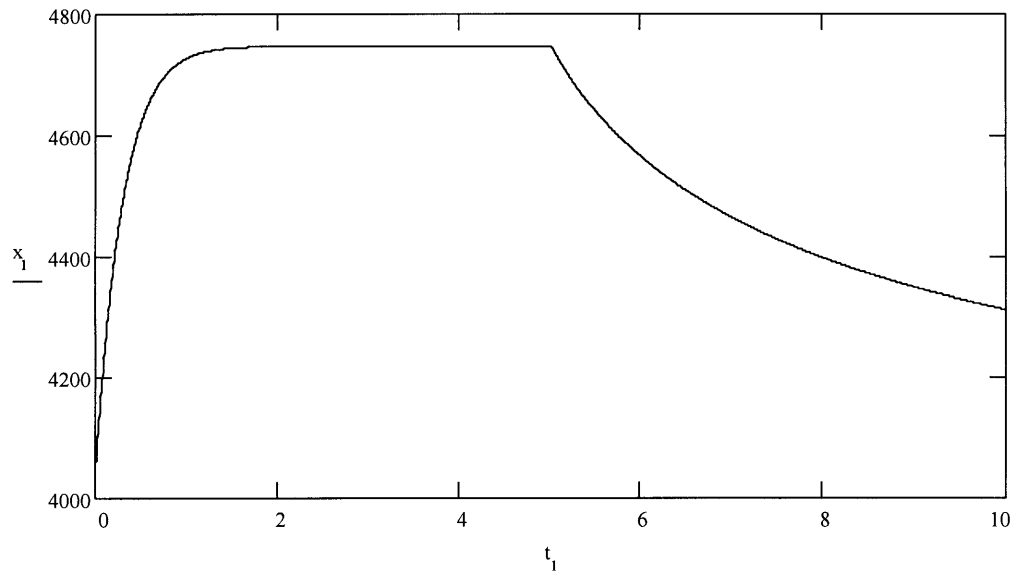
$$\begin{pmatrix} x_0 \\ v_0 \end{pmatrix} = \begin{pmatrix} x_{\text{init}} \\ 0 \end{pmatrix}$$

$$\begin{pmatrix} x_{i+1} \\ v_{i+1} \end{pmatrix} = \begin{bmatrix} x_i + v_i \cdot \Delta t \\ \frac{\rho_{\text{dot}_i} \cdot 100}{\beta} \\ \text{if}(x_i > 8000, .00001, d\rho_{\text{dx}} \left(\frac{x_i - 2031}{\text{ceil}\left(\frac{x_i - 2031}{dx}\right)}\right)) \end{bmatrix}$$

Dynamic rod velocity, in *units/second*



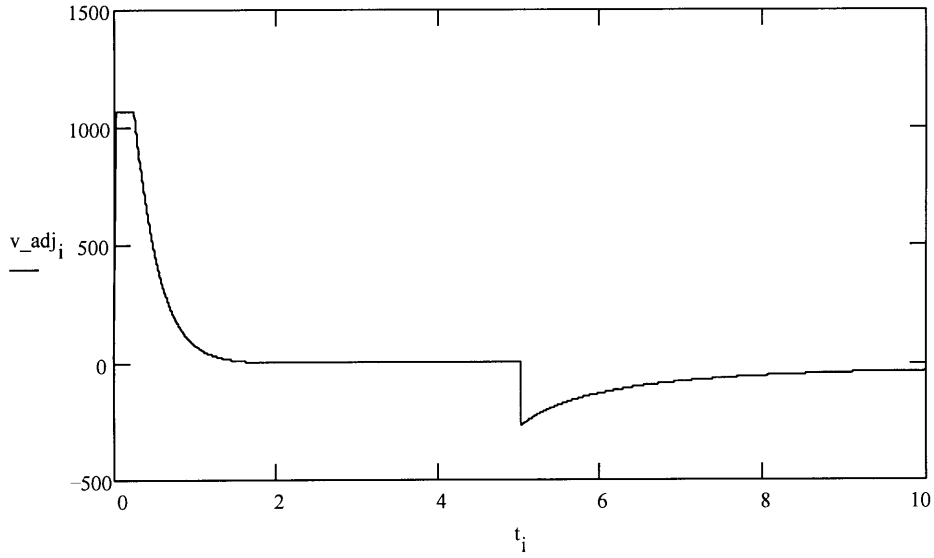
Dynamic transient rod position, in *units*



5. Find adjusted parameters with motor frequency limited to 400 Hz

Adjusted rod velocity (upper bounded by $400 \times 2.661 = 1064.4$ units/second)

$$v_adj_i := \text{if}(v_i > 1064.4, 1064.4, v_i)$$

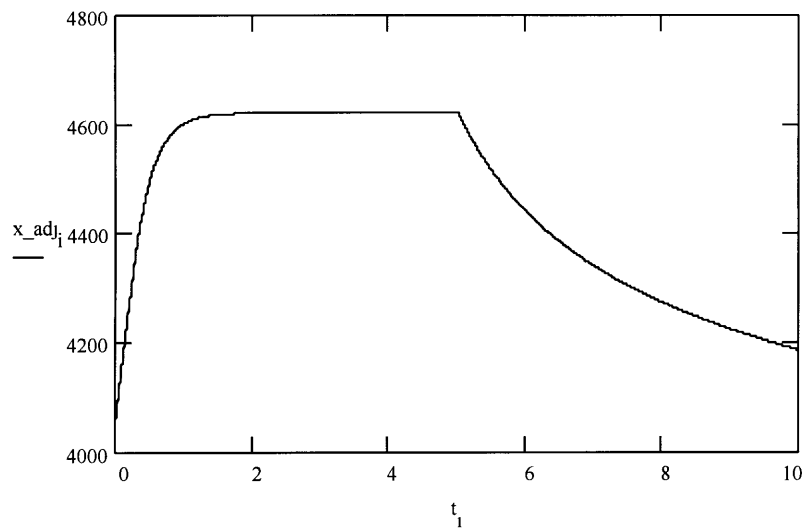


Adjusted rod position and rate of change of reactivity

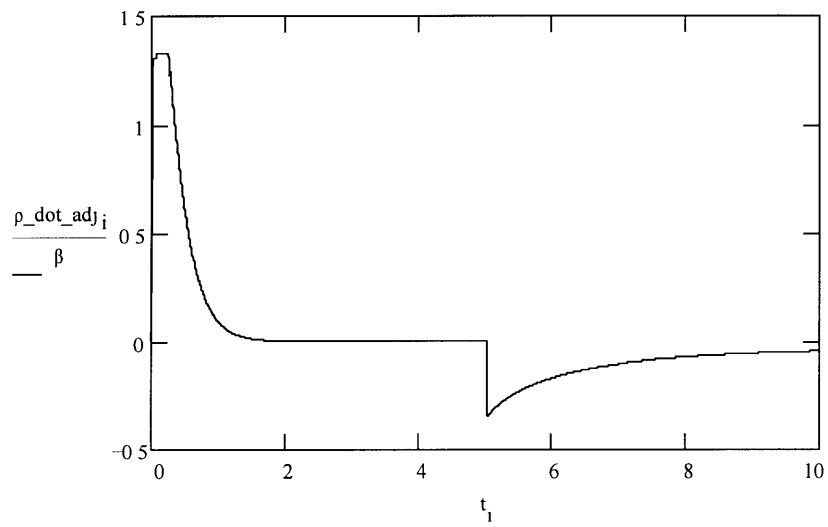
$$\begin{pmatrix} x_adj_0 \\ \rho_dot_adj_0 \end{pmatrix} = \begin{pmatrix} x_init \\ 0 \end{pmatrix}$$

$$\begin{pmatrix} x_adj_{i+1} \\ \rho_dot_adj_{i+1} \end{pmatrix} = \begin{bmatrix} x_adj_i + v_adj_i \cdot \Delta t \\ \frac{\beta}{100} \cdot d\rho_dx_{\text{ceil}\left(\frac{x_adj_i - 2031}{dx}\right)} \cdot v_adj_i \end{bmatrix}$$

Adjusted rod position ($x_{adj,i}$) is smaller than unadjusted rod position (x_i), in *units*



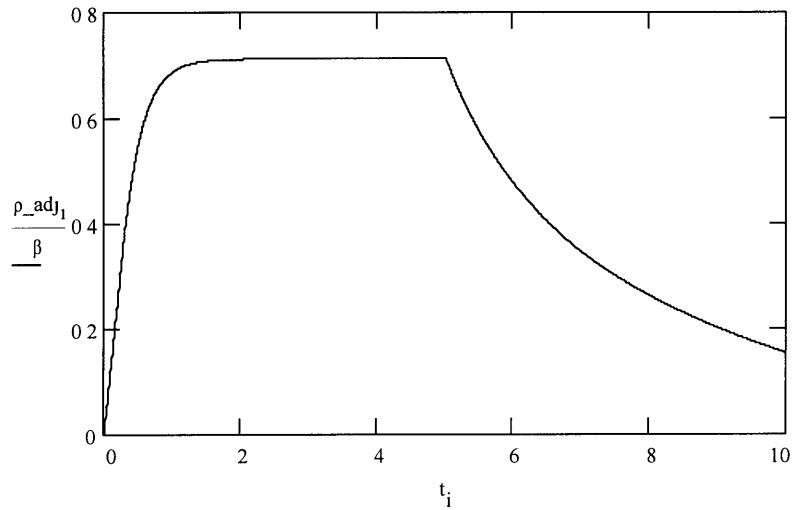
Adjusted rate of change of reactivity, in $\$/second$



Adjusted reactivity (ρ_{adj_i}) is smaller than corresponding required reactivity (ρ_i), in \$

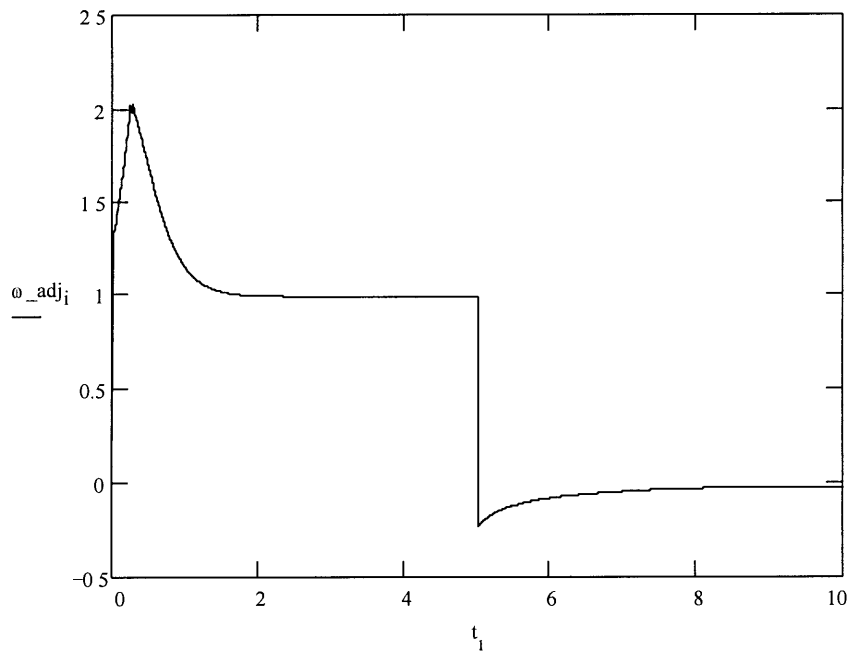
$$\rho_{adj_0} = 0$$

$$\rho_{adj_{i+1}} = \rho_{adj_i} + \rho_{dot_adj_i} \cdot \Delta t$$



Adjusted inverse of reactor period ($second^{-1}$)—it is impossible to maintain the required reactor period.

$$\omega_{adj_i} = \frac{\rho_{dot_adj_i} + \lambda_i \cdot \rho_{adj_i}}{\beta - \rho_{adj_i}}$$



4-72-35