AN INTEGRATED FORMAL APPROACH FOR DEVELOPING RELIABLE SOFTWARE OF SAFETY-CRITICAL SYSTEM

by

MENG OUYANG

B.S., SHANGHAI JIAO TONG UNIVERSITY, CHINA (July 1982)
M.S., MASSACHUSETTS INSTITUTE OF TECHNOLOGY, U.S.A (February 1992)

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

September, 1995

© Massachusetts Institute of Technology, 1995.
All rights reserved

Signature of Author

Department of Nuclear Engineering
August, 1995

Certified by:

Professor Michael W. Golay
Thesis Supervisor

Certified by:

Professor George E. Apostolakis
Thesis Reader

Accepted by:

Professor Jeffrey P. Freidberg
Chairman, Department Committee on Graduate Students

MASSACHUSETTS INSTITUTE
OF TECHNOLOGY

APR 2 2 1996
ARCHIVES
AN INTEGRATED FORMAL APPROACH FOR DEVELOPING RELIABLE SOFTWARE OF SAFETY-CRITICAL SYSTEM

by

MENG OUYANG

Submitted to the Department of Nuclear Engineering on August 11, 1995 in partial fulfillment of the requirements for the Degree of Doctor of Philosophy in Nuclear Engineering

ABSTRACT

This thesis presents the results of a study which devises an Integrated Formal Approach (IFA) for improving specifications of the designs of computer programs used in safety-critical systems. In this IFA, the formal specification techniques of a formal method -- Development Before The Fact (DBTF) and its supporting tool -- the OO1 Tool Suite, are used systematically to identify and remove various kinds of defects in software specifications.

Defects usually exist in most computer programs developed using ad-hoc processes in which mathematical formality is not enforced in the program development effort. Five classes of defects are identified from program studies. The IFA here is designed in order to reduce the number of these defects more efficiently.

This IFA is then applied in two cases studies. One case is that for specifying the small and functionally simple Reactor Protection System (RPS) program. The other case is that for specifying a larger sized, more complex program named the Signal Validation Algorithm (SVA) used in actual nuclear power plant safety system. The results of the applications show that the IFA can quickly identify and remove any ambiguities and inconsistencies in using words and terms, and incompleteness in defining functions and operations in the specifications. The results also show that for a small program like the RPS, functional correctness can be achieved with very high confidence. For a larger program like the SVA, the IFA could efficiently help the system designers to identify the places where improvements of design in functional completeness and correctness should be made. In all, using this approach requires much less work force while produces larger benefits in obtaining a very reliable specification of the program.

Thesis Supervisor: Michael W. Golay
Title: Professor of Nuclear Engineering
ACKNOWLEDGMENTS

Here, I wish to express my gratitude to my thesis supervisor, Professor Michael W. Golay, for his never-waning encouragement and patience, personal and financial support and particularly his visionary guidance throughout the entire research of this thesis. Without all these gifts from him, this achievement would not be possible.

Thanks also go to ABB-Combustion Engineering for their financial support to this research project. Special thanks go to Michael Novak, the primary contact-person within ABB-CE, for his comments, clear focus of requirements upon our research and his insightful review of the results of the research. Thanks also go to Richard M. Manazir, Jr., Raymond R. Senechal, Stephen J. Wilkosz and Robert L. Rescort within Instrumentation and Controls Engineering division of ABB-CE, for their valuable contributions in the discussions of the progress of the research.

I also want to thank the Hamilton Technology Inc. for its supplying us with OO1 system for use in the project. Thanks especially go to Margaret Hamilton for her generous offer of the OO1 evaluation system package; to Ron Hackler for his technical guidance in using the wonderful features of the OO1 system; to Nils for his on-line supports for using OO1 system. I would also like to express my gratitude to Dr. Leonardo Topa for his administrative effort in installing and maintaining OO1 system in the Whitaker College Image Laboratory.

My gratitude is also directed to Professor George Apostolakis for being the reader of the thesis. His timely comments prove to be of great value to perfecting the results my research.

For the time spent in that little office of MIT, I enjoyed working with my colleague Jon Lungalhofer, and we had some stimulating and interesting talks over all the topics. Special thanks also go to Paula Cornelio for all her helps in editing part of my thesis, and to Rachel Morton for her support on using the computer hardware and software facilities of the Institute.

Finally, my great thank goes to my wife, Xiaoxia, for her tremendous encouragement and constant love to me through all these years. My thanks also go to my dear son, Jason, who is the endless source of my inspiration. Thanks also go to my parent-in-law Kee-Tong Wong and Hin-Ming Lau for their love and supports. Mom and Dad, you are great parents. You are always with me no matter how far I go, and we always share our feelings together, no matter how I feel. Thanks for you constant love.
DEDICATED TO

my dear wife Xiaoxia,

my parents Ouyang Da-Song and Cao Li-Nan
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABSTRACT</td>
<td>2</td>
</tr>
<tr>
<td>ACKNOWLEDGMENTS</td>
<td>3</td>
</tr>
<tr>
<td>TABLE OF CONTENTS</td>
<td>5</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>9</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td>11</td>
</tr>
<tr>
<td>CHAPTER 1. INTRODUCTION</td>
<td>12</td>
</tr>
<tr>
<td>1.1 Problems, Opportunities, and Scope of The Study</td>
<td>12</td>
</tr>
<tr>
<td>1.2 Contribution of This Study</td>
<td>17</td>
</tr>
<tr>
<td>1.3 Organization of This Thesis</td>
<td>20</td>
</tr>
<tr>
<td>CHAPTER 2. STUDY OF FORMAL METHODS</td>
<td>22</td>
</tr>
<tr>
<td>2.1 Introduction</td>
<td>22</td>
</tr>
<tr>
<td>2.2 A Software Formal Development Process</td>
<td>22</td>
</tr>
<tr>
<td>2.2.1 Elements of the Formal Development Process</td>
<td>24</td>
</tr>
<tr>
<td>2.2.2 Where in the Process Formal Techniques Apply</td>
<td>27</td>
</tr>
<tr>
<td>2.3 Established Formal Methods and Approaches</td>
<td>29</td>
</tr>
<tr>
<td>2.3.1 Issues of Research and Development</td>
<td>30</td>
</tr>
<tr>
<td>2.3.2 Established Methods and Approaches</td>
<td>31</td>
</tr>
<tr>
<td>2.3.2.1 VDM</td>
<td>31</td>
</tr>
<tr>
<td>2.3.2.1.1 The VDM Specification Language</td>
<td>32</td>
</tr>
<tr>
<td>2.3.2.1.2 The VDM Design Procedure and Tool</td>
<td>35</td>
</tr>
<tr>
<td>2.3.2.1.3 The VDM Applications</td>
<td>36</td>
</tr>
<tr>
<td>2.3.2.2 Z</td>
<td>37</td>
</tr>
<tr>
<td>2.3.2.2.1 Z Specification Language</td>
<td>37</td>
</tr>
<tr>
<td>2.3.2.2.2 Z Design Procedure and Tool</td>
<td>37</td>
</tr>
<tr>
<td>2.3.2.2.3 Z Applications</td>
<td>38</td>
</tr>
<tr>
<td>2.3.2.3 Gypsy</td>
<td>38</td>
</tr>
<tr>
<td>2.3.2.3.1 Gypsy Specification Language</td>
<td>39</td>
</tr>
<tr>
<td>2.3.2.3.2 Gypsy Design Procedure and Tool</td>
<td>39</td>
</tr>
<tr>
<td>2.3.2.4 HDM</td>
<td>39</td>
</tr>
<tr>
<td>2.3.2.4.1 HDM Specification Language</td>
<td>40</td>
</tr>
<tr>
<td>2.3.2.4.2 HDM Design Procedure and Tool</td>
<td>40</td>
</tr>
</tbody>
</table>
2.3.2.5 HOS
   2.3.2.5.1 HOS Specification Language 40
   2.3.2.5.2 HOS Design Procedure and Tool 41
2.3.2.6 IBM Cleanroom Approach 42
   2.3.2.6.1 Cleanroom Specification Language 42
   2.3.2.6.2 Cleanroom Design Procedure and Tool 43
2.3.2.7 SCR 44
   2.3.2.7.1 SCR Specification Language 44
   2.3.2.7.2 SCR Development Procedure and Tool 45
2.3.2.8 DBTF 46
   2.3.2.8.1 DBTF Specification Language 46
   2.3.2.8.2 DBTF Design Procedure and Tool 47
   2.3.2.8.3 Applications 49
2.4 Selection of A Formal Method For Use in This Study 50
   2.4.1 Criteria of the Selection 50
   2.4.2 Result of the Selection 52
2.5 Major Features of the DBTF Method and the OO1 Tool 52
   2.5.1 The OO1 Tool Suite 53
   2.5.2 Modeling Environment 53
   2.5.3 The OO1 Primitive Control Structures 56
   2.5.4 Reusability with the OO1-Defined Structures 57
   2.5.5 Definition of Objects with Parameterized Data Types 59
   2.5.6 Constraint Specification and Analysis 60
2.6 Summary 61

CHAPTER 3. THE INTEGRATED FORMAL APPROACH 63
3.1 Introduction 63
3.2 Systematic Ways For the Reduction of Defects 64
   3.2.1 Classification and Distribution of Defects 66
   3.2.2 The Way to Reduce the Frequency of Defects 69
3.3 A Simple System Design Documentation 72
3.4 The Proposed Integrated Formal Approach 74
   3.4.1 Rules for Transforming Information 76
   3.4.2 Procedure of Constructing Specifications 80
      3.4.2.1 The Reason for Step I and the RPS TMap 83
      3.4.2.2 The Reason for Step II and the RPS Data Type Invariance 84
3.4.2.3 The Reasons for Step III and IV
3.4.2.4 The Reasons for Step V and VI, and the RPS Results
3.4.2.5 Experience From Applying the Seven-Steps Procedure
3.4.2.6 Data Summary of the Developed RPS Specification
3.4.3 Strategy for Testing Specifications
3.4.3.1 Literature Survey on Functional Testing Techniques
3.4.3.2 Selecting Program Functions for Testing
3.4.3.3 How to Select Testing Cases for A Module
    3.4.3.3.1 Identifying the Basic Testing Cases
    3.4.3.3.2 Selecting A Criterion for Complete Testing
    3.4.3.3.3 Defining the Equivalence Classes of Paths
    3.4.3.3.4 Deriving A Complete Set of Test Cases
    3.4.3.3.5 Effectiveness of the Test Case Selection Method
        3.4.3.3.5.1 Effectiveness in Building Structural Graphs
            from Specifications
        3.4.3.3.5.2 Effectiveness on Detecting Defects of
            External Semantics
    3.4.3.4 Integrated Testing on the Entire Specification
3.4.3.5 Quality of the Tested Specifications: What Do We Know?
    3.4.3.5.1 Before the Selected T Tests are Performed
    3.4.3.5.2 In the Process of Performing the Selected T Tests
    3.4.3.5.3 After the T Tests
3.4.4 The Automatic Target Code Generation
3.5 Summary on Treating Defects by the IFA
3.6 Summary of the Chapter

CHAPTER 4. A SYSTEM DESIGN CASE STUDY USING
THE INTEGRATED FORMAL APPROACH

4.1 Introduction
4.2 Signal Validation Algorithm (SVA)
    4.2.1 Signal Validation Features
    4.2.2 Original Design Documents of the SVA
    4.2.3 SVA Formal Specifications Using the IFA
        4.2.3.1 Deriving the Set of Valid States From the TMap
        4.2.3.2 the SVA's TMap
        4.2.3.3 the SVA's FMaps

Page 7
4.2.3.4 Results from Applying the Rules and the Procedure 160
  4.2.3.4.1 Category 1: Ambiguities in Words and Terms 161
  4.2.3.4.2 Category 2: Inconsistencies in Words and Terms 161
  4.2.3.4.3 Category 3: Ambiguities in Operations and Functions 162
  4.2.3.4.4 Category 4: Inconsistencies in Operations and Functions 162
  4.2.3.4.5 Category 5: Incomplete Logic Design of Functions and Operation 163
  4.2.3.4.6 Category 6: Incomplete Output State Definitions 163
  4.2.3.5 Results from Testing the Specifications 164

4.3 Summary and Conclusions 165

CHAPTER 5. RELIABILITY OF THE GENERATED CODE 168

CHAPTER 6. CONCLUSIONS 174
5.1 Results of This Study 174
5.2 Suggested Future Work 176

REFERENCES 177

APPENDIX A. THE RPS PROGRAM COLLECT FILE AND C CODE 184
LIST OF FIGURES

2.1 Process in Developing Software Embedded Engineering System 23
2.2 Alternative Software Development Process Schematic Diagram 24
2.2a WaterFall Model 24
2.2b Formal Development Model 24
2.3 An example of Marriage Bureau Database in VDM Specification 33
2.4 The OO1 Tool Suite Structure 55
2.5 Reliable Building Blocks 55
2.6 The Three Primitive Control Structures 58
2.7 Co-Include Structure Definition 60
2.8 Examples of Uses of Three Parameterized Types 60

3.1 Relationships Among Errors, Faults and Failures 66
3.2 Distribution of Defect Classes 68
3.3 Simple Reactor Protection Signal System Data Flow Diagram 74
3.4 Schematic Diagram of The Integrated Formal Approach 75
3.5 TMap of the RPS Program 84
3.5a CarColor Example TMap 86
3.5b CarColor Example FMap 87
3.5c FMap of Module "Do_Something_To_Car" in CarColor Example 87
3.6 The Reactor Protection System Program FMap 88
3.7 FMap Specification of the Module "Update_Indi" 90
3.8 FMap of the Defined Structure "SetIndiUpdate" 91
3.9 FMap Specification of the Module "CheckGLOK" 91
3.10 FMap Specification of the Module "Update_GL" 92
3.11 Relationship Among Five Coverage Criteria 99
3.12 A Portion of the OO1 RPS Specification 112
3.13 Structural Diagram of the Portion of the OO1 Specification Shown also in Figure 3.12 113
3.14 Expansion Tree of Specification of Module "Update_Indi" As Shown in Figures 3.7 and 3.8 116
3.15 Extended Structure of Specification of Module "Update_Indi" As Derived From the Structure of Figure 3.14 117
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.16</td>
<td>Simplified Structure of the Specification of Module &quot;Update_Indi&quot; As Derived From Figure 3.15</td>
<td>118</td>
</tr>
<tr>
<td>3.17</td>
<td>Expansion Tree of Operation &quot;Update_GL&quot; As Shown in Figure 3.10</td>
<td>129</td>
</tr>
<tr>
<td>3.18</td>
<td>Simplified Structure of Operation &quot;Update_GL&quot; As Derived From Figure 3.17</td>
<td>129</td>
</tr>
<tr>
<td>3.19</td>
<td>Road Map of the Reactor Protection System Program</td>
<td>140</td>
</tr>
<tr>
<td>3.20</td>
<td>Structural Graph of the RPS Main Operation</td>
<td>141</td>
</tr>
<tr>
<td>Table Number</td>
<td>Table Title</td>
<td>Page</td>
</tr>
<tr>
<td>-------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>1.1</td>
<td>The Major Contributions of This Study</td>
<td>19</td>
</tr>
<tr>
<td>2.1</td>
<td>Criteria for Acceptance of a Formal Method for Use in Industrial Software Development</td>
<td>52</td>
</tr>
<tr>
<td>2.2</td>
<td>Formal Methods Comparison Matrix</td>
<td>54</td>
</tr>
<tr>
<td>3.1</td>
<td>The RPS Specification Data Summary - I: TMap</td>
<td>93</td>
</tr>
<tr>
<td>3.2</td>
<td>The RPS Specification Data Summary - II: FMaps</td>
<td>93</td>
</tr>
<tr>
<td>3.3</td>
<td>Summary of Selected Set of Test Cases and Testing Results for the Module #2 &quot;Update_Indi&quot;</td>
<td>128</td>
</tr>
<tr>
<td>3.4</td>
<td>Summary of Selected Set of Test Cases and Testing Results for the Module #5 &quot;Update_GL&quot;</td>
<td>133</td>
</tr>
<tr>
<td>3.5</td>
<td>Results of the Two Tests on the New Specification</td>
<td>139</td>
</tr>
<tr>
<td>3.6</td>
<td>Summary of the Set of Test Cases and Testing Results for the Path A-B-C of the Main Operation</td>
<td>143</td>
</tr>
<tr>
<td>3.7</td>
<td>Summary of the Set of Test Cases and Testing Results for the Path A-B-D of the Main Operation</td>
<td>144</td>
</tr>
<tr>
<td>4.1</td>
<td>Definitions of Object Values of the Element 'ProcessRecord'</td>
<td>157</td>
</tr>
<tr>
<td>4.2</td>
<td>The Set of Valid States of Element 'ProcessRecord'</td>
<td>158</td>
</tr>
<tr>
<td>4.3</td>
<td>Definitions of Object Values in the Element 'SensorRecord'</td>
<td>158</td>
</tr>
<tr>
<td>4.4</td>
<td>The Set of Seven Valid States of the SVA Program</td>
<td>159</td>
</tr>
<tr>
<td>4.5</td>
<td>Improvements in the Original Design in Developing the SVA Specifications</td>
<td>160</td>
</tr>
<tr>
<td>4.6</td>
<td>Numbers of the Five Types of Functional Statements Found in the SVA Specification</td>
<td>164</td>
</tr>
<tr>
<td>4.7</td>
<td>Summary of the Results of Developing the SVA Original Design and O01 Formal Specifications</td>
<td>167</td>
</tr>
</tbody>
</table>
CHAPTER 1

Introduction

1.1 PROBLEMS, OPPORTUNITIES, AND SCOPE OF THE STUDY

In safety critical systems, use of computers provides many potential advantages. These advantages include more sophisticated safety algorithms, improved availability, easier maintenance, reduced installation costs, ease of modification and potential for reuse. However, because of the critical nature of the application, these advantages must be weighed against the problems of ensuring that the computer system can be trusted and adequately assured. The problem of design errors in complex, safety-critical system software is of particular concern.

In the nuclear industry, nuclear safety improvement has been retarded due to the delayed introduction of safety-related digital techniques. Current nuclear technology is thus obsolescent. In trying to remove this obsolete, many nuclear utilities are installing or upgrading their plants' digital related systems and computer systems to improve their economical and safety capabilities. For example, in some nuclear power plants, instrumentation and control systems are out of date, because digital techniques have not much been used, and because verification and validation in the designs of these control systems remains a problem. This is the area where some valuable advancements could be made from this study.

The following phenomenon is often observed in some nuclear industrial software development projects. First, the system designers finish the entire system design documented in their engineering languages. This document could be system
block diagrams, or logic diagrams, or data flow diagram, or just a natural language
description of the objects and related actions, or any combination of them. This
document is then passed to the software developer as system design specifications
in order for them manually to write the corresponding software. However, a large
variation exists in their styles and procedures of programming, a variation in their
skills in developing a program, and perhaps variations in their knowledge of the
engineering system to be designed. The result is that programs implemented by
different programmers perform differently in terms of completeness of
functionality, correctness of functionality and reliability.

This result of the variations in the performance of different programs is
mostly due to the following two factors in the development: (a) lack of
unambiguous specifications of the system designs that should serve as common
bases of communications between system designers and software developers. In
reading designers' specifications that are inherently ambiguous and inconsistent
when they are expressed in natural language, programmers usually interpret
definitions of terms, definitions of functions and interrelated activities between
objects differently (If these differences are not resolved among them, then surely
the products of the design will be different); (b) lack of a systematic formal
approach that enforces unique development of specifications from original design
documents and consistent use of this unique specification for program realization,
testing, quality assessments and program maintenance.

Currently, there is no generally applicable software design process and
software failure model that is sufficiently accurate to enable reliability predictions
to be made for safety-critical systems. Those systems' safety critical functions
usually have stringent demands for high reliability. For instance, the FAA requires
a failure rate of $10^{-9}$ /flight-hour for automatic flight control systems.
As indicated by P. Rook (Rook90), conventionally, the best developed methods for assessment and improvement of software reliability are based upon observation of failures at the stage of testing and operation of the software products with appropriate statistical analyses following those observations. Those methods are studied and used to measure the reliability of the software products, because they are based upon the direct observation of the product's behavior. For our concerns in improving the reliability of the software product, there are two important shortcomings in the currently available methods.

(1) These methods are concerned simply with product reliability measurement at the stage of testing and operation. Information concerning the methods used to develop the product is usually ignored. It would be valuable if this extra information could be used to provide greater confidence in the estimates of reliability of the products.

(2) These methods are only used when the final software product is put into testing and actual running (as reflected in their failure data). Thus, these methods can be seen as a fixing-wrong-thing-up-after-fact (Hami91) treatment. This is a costly process. According to some studies on the classification of faults that cause failures at the testing stage (Beiz90, Mart85, Schu90), it is seen that nearly 85% of the faults identified at the testing stage are related to design errors, specification errors, data flow errors and decision logic errors in the program. If these faults are corrected at earlier steps of the development process in validating and verifying specification against requirements, the cost and labor time of treating them at a later coding stage could be saved and high quality software products would be obtained.

Until nearly a decade ago, there was a lack of emphasis on validation (V) and verification (V), or in other words V&V activities, in the earlier stages of software development, so consequently heavy reliance was placed upon testing.
Today, however, there tends to be more emphasis upon introducing V&V activities earlier and throughout the software development process, thus decreasing both the dependence that the developer places upon testing, and the data that the reliability assessor can expect to use at the testing stages. With the emergence of formal methods for software development, the role of testing may diminish, but will never vanish entirely.

Advances in the theory of programming and computation over last two decades have enabled the introduction of an increasing amount of mathematical formality into software and hardware engineering. Formal method techniques are the terms used to describe the applications of mathematical formality in the development of engineering systems. Their aim is to improve the quality of software in two related ways:

- By providing a clear, complete, unambiguous and easy to validate mathematical specification of statements of the required software behavior;
- By making verification during the software production process more effective and easier to audit.

With the use of a supporting tool developed for the method, the above aim could be realized more efficiently.

According to the studies of the actual experience of industrial usage of formal methods (Bowe93, Crai93), there are three cases in which applications of various formal methods and approaches for system designs are used. They are regulatory-related cases, commercially-related cases, and exploratory cases. This examination concludes that most of the industrial usage of formal methods either is still on very small pilot projects, or is more academically oriented. There is no single formal method that could be perfect for every practical industrial application. With this observation, we see an opportunity to advance the study of
formal method usage in actual industrial safety-critical system software. That is the goal of this work.

This project is motivated by concerns gained in the industrial experience of developing safety-critical systems, such as nuclear reactor engineering systems (Ives94). One concern is with the possible ways to ensure the correctness of the functional design of the program through formal specification. Another concern is with how to obtain an error-free specification and code implementation for a relatively small- or medium-sized system program throughout the development process. The main objective in this study is to study the available logically consistent, or formal techniques to address these concerns, and to develop a comprehensive method for using these techniques to produce high quality software. Thus, the scope of this research covers the following three aspects: (1) an examination of currently available formal methods and approaches, with the aim of identifying their strengths and weaknesses according to industrial application requirements as expressed in document [Abbc94a]; (2) formulation of an Integrated Formal Approach that effectively uses the selected formal method and its supporting tool at each stage of software development process (These stages include capturing the system requirements, documenting the specification of the design, formulating the testing strategy for the program, implementing the code, and helping the effective maintenance of the program and specifications); and (3) assessing the effectiveness of the formulated Approach in improving software functional design, software specifications and code implementation, by applying it to case studies and comparing the results with those from ad-hoc approaches.
1.2 CONTRIBUTIONS OF THIS STUDY

The contributions of this study include three aspects as summarized in Table 1.1. In all, this work has advanced the study of applying formal method in nuclear safety-critical system software design. All the issues in the stated scope of this research have been studied. The overall purpose of this study is to formulate a systematic approach of using identified formal method and its tool to improve the system software design process so that the resulting software would have high quality in completeness of functional design and specification, and error-free code implementation from the developed specification. The purpose is achieved by the successes in the following three aspects.

First, an extensive study has been performed concerning currently available formal methods, design approaches, and available supporting tools (although some methods do not have supporting tools). By following the selection criteria that are based upon actual industrial application requirements, the formal method named Development Before The Fact (DBTF) is selected for use in this study. This method has sound mathematical formality and a very powerful supporting tool: the OO1 Tool Suite.

Second, an Integrated Formal Approach is constructed in order to use effectively the features of the DBTF formal method and its Computer-Aided Software Engineering (CASE) support tool: the OO1 Tool Suite. The approach is constructed to achieve the primary objective of the study: to improve and produce correct system software design and to develop an error-free program. It involves restructuring the traditional systems/software design process. The capabilities of the DBTF formal method and the OO1 Tool Suite are used to their maximum limit. In this approach, the strategy of the test case generation from specifications is formulated, with the definition of Equivalence Class of Paths being given. This
definition is based upon the characteristics of the formal specification in DBTF formal language.

Third, the case studies made using this Integrated Formal Approach are performed for purposes of improving the specifications of two safety-critical system software programs examined. The results show that the approach is effective in identifying many categories of design errors in the original design documents of English text, semi-structured flow diagrams, and pseudo program structures. The testing strategy in this approach helps the system developer gain greater confidence on the reliability of the specifications. Using the information from testing, a conceptual process is discussed for updating one's knowledge of the quality of the tested specification. However, in this process, two important unjustified assumptions are used which should be further studied in an independent work.

The most valuable result from this research is that the selected DBTF formal method, its supporting tool -- The OO1 Tool Suite, and the proposed Integrated Formal Approach prove to be useful in a practical industrial project right now for improving the current development of safety-critical system software.

However, there are also some limitations in this study. Although the devised Integrated Formal Approach includes automatic code generation in the target language code, the reliability of the code generator remains unexamined. The assessments of this code generator of the OO1 CASE Tool has not been formally published by its vendor. This question remains for all CASE tools which provide code generation capability. However, the base of industrial applications of the OO1 CASE Tool is expected to be very wide, so that statistical information concerning the OO1 Tool's code generator will be potentially available from those applications.
Table 1.1 The Major Contributions of This Study

<table>
<thead>
<tr>
<th></th>
<th>Details of the Contributions</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Advanced the study of applying formal methods and supporting tools for use in nuclear engineering system design; Identified formal method and supporting tool which has the potential to satisfy the nuclear industrial practices.</td>
</tr>
<tr>
<td>2</td>
<td>Devised an Integrated Formal Approach (IFA) which uses the identified formal method and its tool to systematically develop formal specification and program code; Proposed a conceptual process for updating the knowledge of the quality of the tested specification by using some unjustified assumptions which could be addressed in an independent study.</td>
</tr>
<tr>
<td>3</td>
<td>From case studies of the Integrated Formal Approach, we found that for small-sized and functionally simple program, error-free specification of the design is achievable with very high confidence. We also found that the IFA can help to improve original program design by identifying classes of possible improvements. The IFA is considered to be valuable for immediate use in actual engineering project.</td>
</tr>
</tbody>
</table>

Second, we did not examine the issue of the quality of compilers for producing object code which is to be used in the target machine. Since compilers are also software products, the improvement of their design could also be achieved, by following the Approach devised in this study.

It should be noted here that the fundamental philosophy of developing highly reliable software is to start by producing small programs with error-free (or high quality) character, and then to produce larger programs based upon those small, high quality programs. This is the philosophy employed by the DBTF method and its CASE tool, and this is the treatment which we have examined in this study.

Finally, although we found from this study that formal methods, through systematic application, could help to improve the current designs of system software by adding some missing functions (for example, with the DBTF method,
the Data Type Map of the system could help to identify some missing functional logic), they could not help designers of the systems to completely re-configure the engineering features of the whole system. From our studies of formal methods, this conclusion is true for all the currently available formal methods.

1.3 ORGANIZATION OF THE THESIS

Chapter 2 presents a survey of currently available literature of formal methods and their supporting tools. It also discusses the current situation of industrial acceptance of formal methods and relevant trends. At last, one formal method and its CASE supporting tool are selected for this study.

In Chapter 3, the elements of a proposed Integrated Formal Approach are discussed. First, from comparison with the conventional development processes, possible ways of improving software reliability using the major features of the selected formal method are discussed. Next, detailed discussions are presented for each element of the approach. These elements include a set of Rules for transforming information from informal design documents into elements of formal specifications; a Procedure for constructing complete and correct formal specifications; a Strategy of performing complete functional testing based on the specifications. Based upon the information from Integrated Formal Approach, a conceptual process of updating one's knowledge of the quality of the tested specification is proposed. In this Chapter, a simple Reactor Protection System design case is used to explain the design and application of the Integrated Formal Approach.

In Chapter 4, a case study of applying the Integrated Formal Approach to an actual safety-critical system program is performed. The case concerns the design of a Signal Validation Algorithm used in the reactor coolant system for validating
the values of safety parameters. The effectiveness of the approach is discussed and results are assessed in detail.

Chapter 5 presents a discussion of assessing the reliability of generated program code. The ways of using the information from specification development process for assessing the program reliability are discussed.

Chapter 6 serves as the summary of this thesis research. In it, the major findings are summarized and conclusions are drawn. Recommendations for future work are presented.
CHAPTER 2

Study of Formal Methods

2.1 INTRODUCTION

The primary purpose of this Chapter is to present the results of the study of the established formal methods and their applications in industrial projects, and to explain the selection of the most appropriate methods for use in this project. First, a formal software development process is introduced with the definitions of each element of the process being given; Then the places are identified in the process where the established formal techniques could be used effectively to improve the quality of software. This is followed by discussions of various formal methods and approaches that have been researched academically, established and applied in some industrial projects. After one method is selected for further use according to the selection criteria, its major features are discussed in detail.

2.2 A SOFTWARE FORMAL DEVELOPMENT PROCESS

The development of a safety-critical system and its control software is a complex process. It usually involves the collaboration of many system and software engineers, each with his own preferred procedures and modes of expression. The process itself must be adequately managed when changes, progress and quality must continually be monitored. Consequently, a variety of tools, both manual and computational, are required. For software and system development to be advanced from their current 'craft' status into the status of being
true engineering disciplines, it is necessary for scientifically based formal
techniques to replace the current ad hoc approaches used in most of development
efforts. However, we also must realize that such a change can occur only in an
evolutionary way. Most of time, it will be necessary to introduce formal
techniques gradually, alongside the less formal approaches. This is true both for
reasons of economy and to avoid adopting error-prone new products.

In developing the software for an embedded digital-based engineering
system, it is important to understand the interfaces and information flows between
the hardware and software parts of the system. The Figure 2.1 illustrates those
general design process interfaces and their relationships.

Figure 2.1 Process in Developing Software Embedded Engineering System
2.2.1 Elements of the Formal Development Process

From Figure 2.1, we understand the relationship between the development of software and the development of the engineering system that embeds the software. Now from the 'software components' side of that figure, we look into the formal procedure of constructing a program.

It is shown by Wing (Wing93) that conventional software development process can be described in terms of a 'waterfall' model as shown in Figure 2.2a. Summarized from several studies (Berg82, Cohe86, Gerh90, Jone90, Mart82), it is generally agreed that a formal software development process should include five elements as shown in Figure 2.2b, incorporating strong feedback between the different elements of the design process,

![Diagram showing the alternative software development process with four phases: Requirements Analysis, Design, Implementation, and Validation.]

Figure 2.2 Alternative Software Development Process Schematic Diagram
Requirements Engineering:

Requirements engineering identifies the major functions and constraints of the program to be constructed. This results in two classes of information about the proposed program. (1) The functional requirements informally describe the required behavior of the program. (2) The attributes (such as the program's set of initial values, initial state, valid system states, security, robustness, and viability).

Specifications:

The Specifications is a document that prescribes, in an unambiguous, complete, precise, verifiable and consistent manner, the desired requirements, design, behavior, or other characteristics of a system or system components. Its only concern is to have engineering requirements interpreted and recorded correctly. A specification should be modifiable, traceable and usable during program operation, modification and the maintenance.

A specification has its intrinsic and extrinsic properties. The intrinsic properties include:

--- Consistency
--- Completeness
--- Conformance to Methodology.

Those intrinsic properties are to be verified using the formal mathematics of the specification language during the development of the specification. Its extrinsic properties include Correctness, which requires the resulting software to be validated with respect to the original engineering requirements.

Verification:

Verification is the process of determining whether the product of a given phase of the software development cycle fulfills the requirements established
during the previous phase. It is a task of showing that a program achieves its intended purposes. The early computers were devoted to mathematical computation, and their programs could be verified by a simple process of manually duplicating the computation for some subset of the data. As computer applications have expanded, the tasks of verifying programs have become more difficult. This results in the development of elaborate verification technology based upon testing (Beiz90).

Testing strategies are inherently limited, because the number of test cases required to exercise even a small program completely, as in complete path testing, is prohibitively large such that complete testing is usually impossible to achieve in practice. Therefore, only a subset of the possible test cases can be tried. Successful completion of a test subset is a necessary but not sufficient condition for proving that a program is correct. Despite these limits, programmers have almost exclusively used testing as the main verification method. However, with the help of some formal techniques used to prove the correctness of the specifications of a program, defining a strategically selected subset of cases might be achievable, and can provide assurance that a program's structure is correct. This point is addressed in Chapter 4.

**Validation:**

Validation is the process of evaluating software at the end of the software development process to ensure its compliance with software requirements.

A concern of this study, as expressed in Chapter 1, is to identify ways to make error-free specification of a software. It is true to say that we are dealing with the software problem using a 'zero-defect' approach, where defect include both error in the original designs of functional requirements and fault in the specification resulting from the previous errors. This approach is one of two
general approaches used to improve the quality of software of safety-critical system. Before moving onto the next Section to discuss the uses of the formal techniques to achieve zero-defect software, we note an other approach, discussed and used by some researchers (Leve86, Schu90): the fault-tolerant approach.

In the fault tolerance approach, the concern is more with safety than with the reliability of the software. Usually, reliability requirements are concerned with making a system failure free, whereas safety requirements are concerned with making it mishap free, or say accident free. Software fault tolerance is a system's ability to provide continuous operation in the presence of software faults, as a result of the redundancy of functional processes. Notice that phrase "in the presence of software faults". This means that faults have the chance to be in the software. Thus this method does not produce zero-defect software, but rather software that will tolerate some faults. Usually a system is designed with up to 30% excess software in order to render the systems fault-tolerant as a means of safeguarding the software against defects. Use of the concept of fault tolerance then, philosophically, replaces the concept of zero defects in the software. This is the approach that we do not address in this study.

2.2.2 Where in the Process Formal Techniques Apply

The academic research and industrial practice during the last decade have greatly advanced software engineering to a more disciplined level where increasing amounts of mathematical formality are introduced. The term, formal method, is used to describe such a cluster of mathematically formal techniques.

Formal methods aim to improve the quality of software in two related ways (Bloo91, Wing90):
• By providing a specification that is clear, complete, and unambiguous, using easy to validate mathematical statements of the required software behavior;

• By making verification during the software production process more effective and easier to audit.

In a formal development process, the specification provides a mathematical description of the concept that a program should implement. This description is then reified (made more concrete) by means of a number of development steps that elevate the level of mathematical abstraction underlying the operations and data structures that are available on the target machine. Although formal development of a program remains an intuitive, human activity, verification and validation have striking differences from an informal development.

By comparing above points with the software development process shown in Figure 2.2b, it can be seen that formal techniques in formal methods are used to address activities in three major blocks: Specification, Validation, and Verification. The objectives of involving these three blocks are explained in more detail below.

The production of a formal specification involves a considerable shift of emphasis toward specification, and a consequent increase in costs incurred early in production. It is justified by the observation (Beiz90, Bloo91, Schu90) that most of the faults discovered late in the production life cycle tend to originate from the specification. These faults are consequently the most expensive to correct. Thus, the cost and time spent on improving specifications can reduce the cost and time required to correct faults found at the later stages of use. Since a formal specification is a mathematical model of software, it is possible to validate it by deriving properties from it. This process is known as specification animation, or specification prototyping, and it may be carried out directly by mathematical
manipulation, or by production of a rapid prototype to investigate certain features of the specification.

Formal methods have greatly increased the effectiveness of formal verification during the software production process. They enable verification to be carried out as an integral part of software production, which is both less expensive and more effective than the usual situation where verification is only attempted by testing software at a late stage in development.

So What is A Formal Method? In summary, it is a method that includes following two elements: (1) Formal Specification, and (2) Verified Design. Those are the formal techniques used in the development process.

2.3 ESTABLISHED FORMAL METHODS AND APPROACHES

The terms, formal methods, formal specification, and formal verification are widely used and have acquired a variety of meanings to mathematicians, engineers, computer scientist and logicians. Internationally, especially between Europe and the USA, there is a considerable degree of difference in the use of formal methods. However, European use is much greater than in the USA in both developing formal methods and applying them to industrial projects. Formal methods have now been developed to the extent that they are increasingly being applied industrially to the production of both software and hardware that must be highly dependable. As indicated by some researchers (Gerh90, Hall90), Europeans and Americans have historically emphasized different approaches to formal methods. Europeans tend to work without tools and to focus on specifications while Americans tend to work with tools and to focus on proofs.
2.3.1 Issues of Research and Development

In both parts of the world, however, governments have taken an active role concerning improving software reliability, both as users and funders. In the USA, national-security organizations have funded these activities for decades, but have adopted a set of standards for trusted computer systems, which is known as the DoD(Department of Defense) Orange Book (Depa85). It is more focused on US software verification for industrial use. Other countries have followed suit in the area of security. The UK has also led with a safety initiative that now provides an interim standard (Ukmi89). This standard can better identify methods and tool combinations with credibility, prescribe how they should be used and impose requirements that companies can respond to. Coming out of this effort, in Europe, two formal methods, the Vienna Development Method (VDM) and Z (named for Zermelo, a developer of modern axiomatic set theory), emerged as the leading candidates to become standard methods. There are other variants of formal methods that originate from these two methods. While in the US, some tool-heavy, security-driven approaches have been established based upon research in taking the challenge of modeling security and building secure systems. The most prominent and widely used approach, is the IBM's Cleanroom treatment. All of these, with other formal methods and approaches are discussed in detail below.

Most of the mature, established formal methods and approaches are used for the description of sequential and algorithmic properties of the system. Theoretical methods and approaches handling concurrency and temporal properties of systems are still the subjects of considerable research activity. A number of promising concurrent formal methods and approaches have been identified (Bloo91, Cohe86) which might eventually form the basis of industrial development methods, including:
- Petri-nets, such as predicate transition nets, using marked graphs with specific interpretations (Nets are marked with 'tokens' which represent loci of control within a concurrent system);

- Communicating Sequential Processes, which model a system as a network in independent processes that communicate via channels - It is model-oriented method that embodies a notion of equivalence in testing.

- Synchronous Calculus Communicating Systems, which is being developed at Edinburgh University and is an algebraic method; it describes systems in terms of independent agents that communicate by means of ports, which are capable of inputs and outputs known as actions.

### 2.3.2 Established Methods and Approaches

The list below presents briefly a number of software engineering formal methods and approaches developed and applied to real industrial applications. These include VDM, Z, Gypsy, HDM, HOS, IBM Cleanroom approach, SCR, and DBTF. These are not, of course, the only formal methods and approaches established and applied to industrial scale problems, but they constitute a large portion of those methods and approaches.

#### 2.3.2.1 VDM

The term VDM stands for Vienna Development Method. It was developed at the IBM Vienna Research Laboratory during 1970. This method is a rigorous method of specification and design (Jone90). The VDM is based upon the approach to programming language theory known as denotational semantics. The VDM was originally used for programming language definition. It has subsequently been advanced, however, into a general-purpose software
development method and applied to a variety of applications. It is now standardized by the British Institute of Standard and is a method mandated to be used in safety-critical system software development projects (Ukmi89).

The VDM is a model-based approach in that descriptions of systems (both specification and design) are given as models. The constituents of models are data objects representing inputs, outputs and internal 'states' of the system, and operations and functions that manipulate the data. Class of data objects can be defined as data types (sometimes called domains as in VDM literature). The VDM is most suitable for specifying sequential information processing systems.

2.3.2.1.1 The VDM Specification Language

The specification language of VDM is a first order prepositional calculus; some literature calls it the Meta-IV language. A VDM specification consists of two major parts: Data objects and Operations. The data objects are specified using abstract mathematically oriented data types, such as sets and mappings; Operations and functions that manipulate those data are specified either implicitly using Pre- and Post-conditions or constructively with recursive functions. Operations and functions are usually specified using predicates in the form of the first order prepositional calculus. Complex predicates can be constructed by means of standard operators of predicate logic, such as the prepositional connectives and universal and existential quantifiers. For understanding how VDM formal specification is applied, let us look at example of VDM specification of a marriage bureau database, as shown in Figure 2.3 (example from Ref. (Cohe86))

The state of the system is defined using two abstract data structures. UNMARRIED represents the set of unmarried people in the database, while MARRIED represents the set of successfully matched clients. The values of both these variables will be the sets of people, so their type is Person-set. The type
Person is not yet defined in this specification, since it is dependent upon the detailed requirements of the marriage bureau (i.e. precisely what information they wish to store for each client) and is not relevant to the description of the operations which follow.

| State :: UNMARRIED  Person-set  
|        MARRIED       Person-set  
| Person = /* some suitable representation */  

REGISTER (Person)

ext UNMARRIED : wr Person-set
MARRIED : rd Person-set

pre p /∈ unmarried ^ p /∈ married
post unmarried' = unmarried ∪ {p}

MARRY (M:Person, W: Person)

ext UNMARRIED : wr Person-set
MARRIED : wr Person-set

pre m /∈ unmarried ^ w /∈ unmarried
post let couple = {m, w}
    married' = married ∪ couple
    unmarried' = unmarried - couple

INIT ()

ext UNMARRIED : wr Person-set
MARRIED : wr Person-set

post unmarried' = {} ^ married' = {}  

Figure 2.3 An Example of Marriage Bureau Database in VDM Specification

The remainder of the specification describes the operations which may be carried out by the marriage bureau clerks. The specification of each operation consists of four parts:
1. The name of the operation and any input or output parameters that it takes (The names and types of the input parameters are given in the functional arguments (parentheses) which determine the values of the output parameters.

2. A clause which indicates which part of the state the operation needs to access (This clause begins with the keyword ext (for external), and for each state component accessed, read only or read/write access is specified by the keywords rd and wr respectively).

3. A precondition which is a predicate over the values of the input parameters and the initial state, and which indicates the conditions for which the operation is defined to have an effect (If a precondition is given which might be FALSE, the specification of the operation is only partial, in that not all possible behaviors of the system are defined - For example, in the REGISTER operation, the precondition requires that the person to be legally registered must not be in both the UNMARRIED set and MARRIED set of the bureau's database).

4. A postcondition, which shows how the values of the variables in the state are affected by the operation, and which also defines how the values of the output parameters are to be generated (This is a predicate preceded by the keyword post; The postcondition is valid only if the precondition of the operation holds true; In the postcondition it is necessary to refer to values of variables in the state before the operation and in the state after the operation. In order to distinguish the two, variables in the post-state are suffixed with a prime. For consistency, the values of output parameters are similarly marked. For example, the postcondition clause of the REGISTER operation means that once the person is legally registered, the UNMARRIED set should be updated by adding this person in as an element in that set. The new UNMARRIED set is marked with a prime).
In the VDM specifications, invariant conditions of the system state are usually specified. These are the predicates that define additional constraints on the values that variables may assume which (usually) cannot be given through the type definition mechanisms. For example, in the marriage bureau database system, it should be clear that no individual person should be presented as being in both the set of unmarried clients and the set of married clients. This invariant condition is documented by adding to the specification the following predicate:

\[ \text{inv-State} = \text{unmarried} \cap \text{married} = \{\} \],

which says that the invariant state system requires the zero intersecting set of the sets \text{unmarried} and \text{married} to be null.

2.3.2.1.2 The VDM Design Procedure and Tool

Besides the formal notation of the specification language, VDM also provides rules and procedures to be followed in various stages of system development. Development of a program goes on either by data reification or by operational decomposition, by gradually including design, algorithmic and implementation detail.

In data reification (refinement), a new state that is closer to the implementation is defined and the operations are redefined on this state. A retrieve function relates the new, more concrete specification to the more abstract specification by showing how, given a state of the representation, the corresponding abstract specification can be achieved. At each reification stage, it is important to construct proofs that show why the reification adequately models the previous stage. In operational decomposition, however, the state remains unchanged and the operations are redefined as combinations of simpler operations.
using control structures such as sequence, selection and iteration. As with the reficitation process, a number of proof obligations arise; at least one for each of the control structures which is used within the decomposition process. Currently, no tool has been formally developed to support the use of VDM in applications, completely from specifications through design. However, in the research project Genesis (Crai89) funded by ESPRJT, the immature Genesis Meta-tool has been created to support the use of VDM, such as the LVF proof system. Due to its poor interface performance, it is still undergoing improvement Another tool called Adelard's SpecBox (Blo9o91) is created for the syntax and semantics analysis of VDM specifications. It is also still in the process of improvement and has not been widely used.

2.3.2.1.3 The VDM Applications

The VDM has been applied in variety of situations (Jone9o9b) such as language definition and data base and operating system development. Since its development, its applications (Cohe86) have been continued in the Dansk Datamatik Center, Copenhagen, the Universities of Manchester and Oxford, and the Standard Telecommunication Laboratories. A variety of applications have been carried out such as the definition of the CHILL and Ada programming languages, specification of the Kernel Ada Programming Support Environment, development of a formal model of system R and the development of B-trees. Other areas, such as office automation systems, have also been addressed. At the Dansk Datamatik Center, VDM has been applied to the development of a production-quality Ada compiler system, a large application involving the production of approximately 230000 lines of source code, and Rolls Royce and Associates have used VDM to specify and verify a nuclear reactor protection system (Blo9o91).
2.3.2.2 Z

The Z is an approach to the formal specification of systems originated by Abrial but under continuing development in the Programming Research Group of the University of Oxford.

2.3.2.2.1 Z Specification Language

As a model-based approach, Z uses set theory and first-order predicate logic as a specification language. It is conceptually similar to the pre- and post-condition style of VDM in that system operations are described as relations on a pair of states, using logical properties.

2.3.2.2.2 Z Design Procedure and Tool

In Z, specifications are written as schema boxes, each of which contains a signature part and a predicate part. The schema calculus may then be used for combining specification. Great emphasis is given to the readability of specifications, and typically, formal specifications in Z are embedded in natural language documents, where the natural language commentary provides the background which is essential for the specification to be understood. Schemas can be manipulated by operations for extension, restriction, inclusion and composition. Generic schemas can also be written. Data reification or operational decomposition is also used for development of a program from specifications. So far, no Z tools have been formally developed to support the full scope of activities from specification to program development. One research (Crai89) has developed a syntactic support tool for Z to be used with proof system.
2.3.2.3 Z Applications

A number of non-trivial case studies have been published by the Oxford group (Haye87), such as the specification of a screen editor and of the UNIX file system. Z has also been applied to the CICS specification of an existing transaction control system, and in particular to the applications of programming interfaces (Bloo91). In several other projects, Z has been used to develop a formal specification of the safety-critical control system of a particular medical device (Jack93); It is used in combination with the Yourdon Structured Analysis notation (Semm90) to produce a requirement specification that is both structured and formal, for ensuring the data security of a computer system; In studies of using Z for hardware system design, one case (Spiv90) uses Z to construct the model of an embedded real-time kernel of a diagnostic X-Ray machine to prove the design, and it finally reveals some design flaws; In another case (Deli90), Delisle and Garlan use the Z method to gain insight into system architecture of electronic instrument design. For more information on Z, the reader is referred to (Bard91) and (Pola91).

2.3.2.3 Gypsy

Gypsy is a project aimed at producing a method, notation and environment for formal specification. It is being developed at the Institute for Computing Science and Computer Applications (ICSCA) at the University of Texas, Austin. It is based upon an axiomatic proof theory associated with a formally defined language. Gypsy has been largely developed for US military projects, so it is not generally available publicly.
2.3.2.3.1 Gypsy Specification Language

The specification language of Gypsy is also called Gypsy, which is a high-level programming language. This language has been designed to ease verification, to support modular program development and to describe system software. The language has also been structured so that verification can be performed incrementally during program implementation. The program description language has a Pascal-like syntax. A Gypsy program consists of a collection of small independently verifiable units that can be of type 'routine', 'macro', 'constant', or 'type'. Gypsy provides concurrency by means of the 'cobegin' statement. Fault detection, isolation and recovery are handled in Gypsy by means of 'conditions'.

2.3.2.3.2 Gypsy Design Procedure and Tool

The basic procedure in Gypsy is to write a specification in the Gypsy language and then to implement it by expanding the program units into an explicit program, again within Gypsy. The proof of specification can proceed in parallel with implementation by proving the implementation of a unit to be correct with respect to its specification, using only the external specifications of any units as references. Gypsy is supported by a sophisticated toolset that includes a syntax-directed editor, a parser, a verification condition generator, a theorem prover and a compiler, all coordinated by an executive that handles interactions with the user. Very few Gypsy based case studies are available for examination.

2.3.2.4 HDM

The term HDM stands for Hierarchical Design Methodology. It was developed at SRI International as an aid to developing reliable and maintainable software. HDM consists of a set of guidelines describing its view of stages of software development, a set of languages and a set of automated tools. It is based
upon an operational modeling approach. This approach has been, and is continually being, refined to address a wider range of problems with the aid of more sophisticated support tools.

2.3.2.4.1 HDM Specification Language

The major part of the HDM notation is the specification language, SPECIAL, which is used to describe the methods of a design. A subset of the semantics of SPECIAL has been formally defined using the Boyer-Moore theory. SPECIAL is a formal notation that allows one to define the types of arguments of and values returned from a procedure, and the effect of the procedure upon the internal system 'state'. Procedures are grouped in 'modules' and modules are grouped in levels of the design hierarchy.

2.3.2.4.2 HDM Design Procedure and Tool

The process of development is divided into seven stages that range from conceptualization of a problem to concrete implementation.

The most important tool in the HDM environment is STP, an interactive system supporting the specification and verification of theories. It provides a core theory whose primitive types are Boolean, Integer, Rational and Real, and mechanisms for constructing new theories. Few applications are available for examination. One study (Cohe86) provides four references of applications of the HDM method.

2.3.2.5 HOS

The term HOS stands for Higher Order Software, which is a development method marketed by Higher Order Software, Inc., of Cambridge, Massachusetts. It is important in that it is one of few formally based methods to have been used
successfully on large industrial projects and to be marketed commercially. However, HOS no longer exist in the market today and its advanced version, which is called Development Before The Fact -- DBTF, has replaced it. The DBTF methodology is discussed subsequently.

2.3.2.5.1 HOS Specification Language

The basic ideas found in HOS are a combination of abstract data types and functional decomposition. Types can be either defined axiomatically or derived from other types, or can be given 'operational' definitions as segments of programs. Three primitive 'abstract control structures' are provided (with their composition 'axioms') and more complex abstract control structures can be constructed from them.

Two language forms are provided. A graphical form is used for input and presentation, but the definition of new types must be represented in a classical algebraic notation called AXES, in which a full system can also be described.

2.3.2.5.2 HOS Design Procedure and Tool

A system (specification or design) is represented by a function and its hierarchical decomposition only using well defined control structures. These may include recursion and parallelism, which has in its sub-notes only operations on defined abstract data types. This can be read as a functional program. By providing realization for the referenced data types and functions, this functional program can be executed, thereby enabling 'rapid prototyping', or it can be translated to a language form (such as FORTRAN, Ada. Pascal) which can be compiled, raising the possibility of code optimization, both before and during compilation. The result is a form of automatic program generation.
Higher Order Software supplies, commercially, a comprehensive toolset, called USE-IT, to support their HOS methodology. This toolset contains a graphic editor, an analyzer for specification analysis, a resource-allocation program for generating target programming language and a simulator for prototyping the design. The applications are discussed in details in reference (Mart82).

2.3.2.6 IBM Cleanroom Approach

Most of the US approaches to software development are tools-heavy, security-driven approaches. IBM's Cleanroom approach is a major approach and is referred widely (Bloo91, Dema90, Dyer92). It is complex process model involving statistical testing, verification, functional specification, and reviews. Using it does require major organizational commitment to training and management processes.

Since the 1970s, the Federal System Division of IBM has been developing software using the 'Clean Room' concept. The purpose is to deny the entry of defects during the development of software. The focus of the method imposes discipline upon the development process by integrating formal methods for specification and design, a non-execution-based program development capability (apparently lacking a "prototyping" capability), and statistically-based independent testing. These components are intended to contribute to development of a software product that has a high probability of zero defects and consequently a high measure of operational reliability.

2.3.2.6.1 Cleanroom Specification Language

In order to support the life cycle of executable increments, Cleanroom developers use "structured specifications" to divide the product functionality into deeply nested subsets that can be developed incrementally. The mathematically based design methodology used in Cleanroom incorporates both structured
specifications and state machine models. A system engineer introduces the structured specifications to restate the system requirements precisely and to organize the complex problems into manageable parts.

The specifications determine the "system architecture" of the interconnections and groupings of capabilities to which state machine design practices can be applied. System implementation and test data formulation can then proceed from the structured specification independently.

2.3.2.6.2 Cleanroom Design Procedure and Tool

The programming methods used in Cleanroom: "to get it right the first time", embody the ideas of functionally based programming. The testing process is completely separated from the development process by not allowing developers to test and debug their programs. The developers focus upon the techniques of code reading by stepwise abstraction, code inspections, group walkthroughs, and formal verification to assert the correctness of their implementation. These nonexecution-based methods are called "off-line software review techniques". These constructive techniques apply throughout all phases of development, and condense the activities of defect detection and isolation into one operation. The intention with Cleanroom is to impose discipline upon software development so that system correctness results from a coherent, readable design rather than from reliance upon execution-based testing (prototyping).

However, after the design stage is passed, then statistically-based independent testing is conducted by independent testers. The system's performance is simulated with random testing. This testing process includes defining the frequency distribution of inputs to the system, the frequency distribution of different system states, and the expanding range of the developed system capabilities. It must be emphasized that the independent testing group
operationally tests the software product increments from a perspective of reliability assessment, rather than a perspective of error detection. The responsibility of the test group is, therefore, to certify the reliability of the final product rather than to help the development group in getting the product to an acceptable level of quality.

The Cleanroom method itself does not have any toolset to support applications. However, it encourages the use of any tools that will help to construct and manipulate the system design and source code, to ease the process of reviewing the system design and developed source code before submission for testing. The tools identified to be useful in Cleanroom development process are static analyzers, data flow analyzers, syntax checkers, data type checkers, formal specification checkers, concurrency analyzers, and modeling tools.

IBM report (IBM84) and Reference (Dema90) provides detailed discussions of the Cleanroom approaches and more instances where is has been applied.

2.3.2.7 SCR

The term SCR stands for Software Cost Reduction. The basic techniques of SCR were developed by David Parnas and others at the US Naval Research Laboratory. These techniques were tested and refined in a project involving the onboard operational flight program for the US Navy's A-7E aircraft (Will94). Since then, elements of the method have been applied to several industrial applications, including that of the reactor-shutdown system for the Darlington Nuclear Generating Station in Ontario, Canada (Crai94).

2.3.2.7.1 SCR Specification Language

A SCR specification is designed to contain everything one must know to produce an acceptable software system. It lets one specify the system's external
behavior without implying any particular implementation. The SCR specification language uses a tabular notation based on a mathematical language of modes (states), conditions, and events. It classifies all data items and identifiers according to type (input variable, mode, mode class, and so on). It also uses special bracketing symbols to denote the type associated with an identifier.

2.3.2.7.2 SCR Development Procedure and Tool

In the Darlington nuclear power station application case (Crai94), the formal method SCR was used to convince the Atomic Energy Control Board of Canada that the computer code of the station's reactor shutdown systems was of acceptable quality and following the specification. SCR, as applied at Darlington, had three main components in the development process:

- Formalized informal requirements, developed by generating specification tables,
- Use of the existing code to develop program-function tables for it, and
- Demonstration that the code is consistent with the specifications.

The specification and program-function tables consist of mathematical formulas that express the effects of the relevant routines. The proof of consistency consists of manually transforming and comparing these tables. In the Darlington example, three teams applied the SCR/Darlington method. For the most part, the developers felt that the process worked well, although they had no specialized formal method tools. Instead, the proving team performed the analysis manually. This process was identified to be labor-intensive. At one point, as many as 30 people were working on different aspects of verification.
2.3.2.8 DBTF

The term DBTF stands for Development Before The Fact. It is a software development methodology marketed by Hamilton Technology Inc., of Cambridge (Hami92a). It is an advanced method based on the HOS methodology developed by the same person: Margaret H. Hamilton of Hamilton Technology Inc.. DBTF has been successfully applied in many large industrial projects and is being marketed commercially.

With the Development Before The Fact methodology, every system is in an integrated, hierarchical, functional and object-oriented network based upon a unique concept of control. The method has its beginning in 1968 with an Apollo space mission when research was performed for developing software for man-rated missions. This led to the finding that interface errors accounted for approximately 75% of all errors found in the flight software during final testing. A theory and methodology were derived for defining a system such that this entire class of interface errors would be eliminated via automation. The first technology derived from this theory concentrated on defining and building reliable systems in terms of functional hierarchies (Hami86). Since then, this technology was further developed to design and build systems with the DBTF properties in terms of an integration of both functional and type hierarchies. Application of this methodology is supported by the integrated CASE Tool Suite called the OO1 System.

2.3.2.8.1 DBTF Specification Language

The OO1 AXES language is the specification language in this DBTF methodology. It includes two major components: FMaps and TMaps that are the OO1 hierarchical mapping tools that not only guide the designer in conceptual thinking at all levels of system definition and design but also provide the ability to specify the design. TMaps (read as 'T' maps which stand for Data Type Maps) are
used to specify data type hierarchies of the system. The TMap defines all of the primitives and abstract data types that the system objects could have. The FMaps (read as 'F' maps which stand for Functional Maps) are used to specify functional hierarchies that show interlocking and interconnecting functions at all levels of control. FMaps employ three primitive control structures and variants of primitive control structures in order to control the data flows among those functions in the hierarchies. Those control structures also control the decomposition of functions at each node of the hierarchies. This character is very similar to that of VDM where program development from specification could be achieved by functional decomposition using a set of control rules.

The same language is used to define functional architecture, resource architecture and the allocation of the functional architecture to resource architecture. The allocation definition defines how the elements of the resource architecture are applied to the functional architecture.

2.3.2.8.2 DBTF Design Procedure and Tool

The DBTF methodology is used throughout a life cycle of a system starting with requirements and continuing with functional analysis, simulation, specification, algorithm development, analysis, system architecture, configuration management, software implementation, testing, maintenance and reverse engineering.

A system development is the result from the integration of the FMaps and TMaps specified earlier. The way that they are integrated is as follows.

The TMap provides universal primitive operations inherited by all types (for example, Copy). The universal primitive operations are used for controlling objects and object states. These universal operations create, destroy, copy, reference, move, access a value, detect and recover from errors and access the type
of object. They provide an easy way to manipulate and permit one to think about different types of objects. With the universal primitive operations, building systems can be accomplished uniformly. TMaps and OMaps (a kind of Object Map that is an instance from a TMap) are also available as types. They make the system understand itself better, and let the system manipulate all objects in the same way.

TMap properties ensure data structure consistency and the proper use of objects in a FMap. A TMap has a corresponding set of control properties for controlling spatial relationships between objects. One cannot, for example, put an object into an object structure where an object already exists; conversely, one cannot remove an object from a structure where no object exists; A reference to the state of an object cannot be modified if there are other references to that state in the future; reject values exist in all types, allowing the FMap to recover from failures if they are encountered.

The "goodness" of a system design can be evaluated based upon the attributes of the particular FMaps and TMaps used to define it. The number of layers in a TMap, the degree of strong typing in a TMap, the number of inputs and outputs associated with each function in the FMaps, the sizes of the FMaps and TMaps, and the degrees of movement around a TMap needed to accomplish each functional task in a FMap, all come into consideration concerning the "goodness" of a system.

The DBTF design method is by its nature object-oriented from the beginning. The definition space is a set of real world objects, defined as FMaps and TMaps. Objects that are elements of the TMaps are materialized as OMaps (read as 'O' maps which stand for Object Maps). An execution of a system that is an instance of a FMap is materialized as EMap (Execution Map). Each system has only one EMap at a time. From the program's FMaps and TMap, complete target
system code or documentation is automatically generated and ready to execute. Such a technology changes the way that software is developed just as word processing has changed the way in which offices were managed. The building blocks are definitions, which are independent of particular object-oriented implementations. Properties of classical object-oriented systems such as inheritance, encapsulation, polymorphism and persistence are supported with the use of generalized functions on OMaps and TMaps.

The OO1 Tool Suite supports the automation of the methodology. It includes the following elements:

- The OO1 AXES Specifier; an OMap Editor for defining FMaps and TMaps in either graphical form (a very user-friendly feature) or in textual form;
- An Analyzer for analyzing the specifications of FMaps and TMaps at any point during the definition of a model in order to ensure that the rules for using the mechanism are followed correctly;
- An Executor that allows the user to execute or simulate the behavior of a system before the program's implementation in order to observe characteristics such as timing, cost and risk of failures.
- A Generator for generating source code and all specification documents.
- A Requirements Traceability RT(x) tool which generates metrics and allows the user to enter a requirement document into the system and trace between requirements and corresponding FMaps and TMaps throughout system specification, detailed design, implementation and final documentation.

2.3.2.8.3 DBTF Applications

The DBTF method and the OO1 Tool Suite have been used in the United States by many large companies, including McDonnell Douglas, Martin Marietta, IBM, Los Alamos National Lab., Citibank, etc., in their projects of software
system development (Hami92b). The results show that by using this method and
tool, the software development productivity gain is quite significant, ranging from
increases by factors of 40 to 100. After the evaluation by the National Test Bed
(Dod93) in June 1993 of a set of leading I-CASE tools, the DBTF method and the
OO1\textsuperscript{tm} System have received outstanding marks as the first in the overall
completeness of solution in the final report of the National Test Bed. Since then,
the HTI-DBTF base of customers greatly expanded nationwide.

2.4 SELECTION OF A FORMAL METHOD FOR USE IN THIS STUDY

As is mentioned in Chapter 1, the motivations of the study reported here are
based upon two concerns of developing software for safety-critical system
applications: (1) to ensure the correct and complete functional design and (2) to
have error-free specification. With those formal techniques and their industrial
application experience, the advancement of software engineering makes it possible
for us to pursue our goal by using these techniques. However, since there exists
such a large variance of characteristics in those formal techniques, that it is
necessary to use them in meeting the characters and requirements of our project.
Thus, we need a set of criteria to use in making this selection.

2.4.1 Criteria of the Selection

Although many formal methods and approaches have been developed,
industrial acceptance is still very slow. One reason is that there is a big gap of
understanding concerning formal methods between their developers (most of them
are from academia) and users (industrial practitioners). As one review describes
matters (Webe93), the former feels that the latter must be utterly stupid since they
cannot comprehend their elegant formalisms; the latter finds the former
exceedingly arrogant and their formalisms baffling, and thus unusable. This is an unfortunate situation. What is even worse - there is no one true formal method that will be all things to all projects. More points of propositions for industrial strength formal methods could be found from that review.

It must be understood that industrial software is the software created and maintained under adverse conditions such as that subjected to ever-changing specifications, unreasonable constraints, short deadlines, and understaffed projects. There are generally three clusters of industrial application cases (Crai93). They are (1) Regulatory clusters where an agency of some government requires a certification for the product and/or its development process; (2) Exploratory clusters where an organization is exploring the efficacy of formal methods on a specific product or in its development process; (3) Commercial clusters where a firm is producing a product in which high quality requirements or productivity, or reliability are of concern. An investigation of actual industrial usage (Crai93) finds that there is a clear need for improved integration of formal methods techniques with other software engineering practices. These practices can sometimes be used like quality assurance techniques (like hazard analysis, safety analysis, testing) or some existing design methods (structured design method). There is a need in industrial practice to have cost-effective tools to accompany the formal method, also the formal specification notation must be user-friendly and easy to use and to communicate.

Through communications with industrial practitioners (Abbc94a) and based upon an understanding of the industrial working situation, the criteria shown in Table 2.1 are used for the selection of a formal method:
Table 2.1 Criteria for Acceptance of a Formal Method For Use in Industrial Software Development

| 1. There must be an easy path for users to follow in learning and using method, with limited resources required to be spent on training in the techniques; |
| 2. At some level, the method must be usable by people unfamiliar with data process or mathematics, i.e., it must be user-friendly; |
| 3. The method must be capable of being automated, i.e., by using CASE tools on some platforms, and it must be viewed by all as more productive way to produce specification; |
| 4. The method must primarily assure the completeness of the requirement's collection and description process; |
| 5. Specifications developed by the method must be able to be audited and traceable by person without significant training unique to the method; |
| 6. The method must have flexibility to permit adaptation to changes of the requirements and adequately to support development of partial (in-progress) and complete specifications; |
| 7. The method must improve conceptual clarity, and |
| 8. Specifications developed by the method must serve as a common communications medium among designers, implementors, documenters, maintainers and managers. |

2.4.2 Result of the Selection

Emerging the best, according to above listed criteria, among the identified formal method techniques, the DBTF method is selected for use in this study. A method comparison matrix is presented in Table 2.2. The table shows the results of the selection of DBTF, which is observed to meet all required features of the criteria for the selection.

2.5 MAJOR FEATURES OF THE DBTF METHOD AND THE OO1 TOOL

Some selected features of the OO1 tool have been discussed in the previous section concerning the DBTF method. In this section, the major features of the DBTF method and the OO1 Tool (Hami91) are presented systematically, presenting a complete picture of the OO1 Tool Suite with detailed information concerning each component of the tool suite.
2.5.1 The OO1 Tool Suite

The OO1 Tool Suite is an integrated system and software development environment. An automation of the Development Before The Fact approach, it is used to define and generate itself. Its conceptual structure is shown in Figure 2.4.

2.5.2 Modeling Environment

The OO1 modeling environment makes the assumption that reliable systems are defined in terms of reliable systems. Only reliable systems are used as specification building blocks and only reliable systems are used as mechanisms to integrate these systems. The building blocks, as shown in Fig. 2.5, include types of objects; functions whose inputs and outputs are members of those types; control structure that relate a parent type to its child types and a function to its child functions; relations that link types to types and constraints (usually hidden in the window) that define the external boundaries within which a system may reside.
Table 2.2  Formal Methods Comparison Matrix

<table>
<thead>
<tr>
<th>Features</th>
<th>DBTF</th>
<th>Gypsy</th>
<th>HDM</th>
<th>HOS (USE.IT)</th>
<th>IBM Cleanroom</th>
<th>SCR</th>
<th>VDM</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>Meeting Requirements? (f)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Requirements? (numbers)</td>
<td>(OO1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mathematically Based</td>
<td>1, 7, 8</td>
<td>1, 7</td>
<td>none</td>
<td>1, 7, 8</td>
<td>none</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>Model-Based Specification</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>Specification Self-Checking</td>
<td>3, 4</td>
<td>none</td>
<td>none</td>
<td>3</td>
<td>none</td>
<td>none</td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td>Validation/Prototyping</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>none</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>User-Friendly (e) in Application</td>
<td>2</td>
<td>none</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>none</td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td>Specification (e) Auditable/Traceable</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>none</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>Specification (e) Easy to Modify</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>Specification Easy to Maintain</td>
<td>6, 8</td>
<td>none</td>
<td>none</td>
<td>6, 8</td>
<td>none</td>
<td>none</td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td>Automatic (e) Code Generation</td>
<td>3</td>
<td>none</td>
<td>none</td>
<td>3</td>
<td>none</td>
<td>none</td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td>Supporting (e) Tool Availability</td>
<td>2, 3</td>
<td>none</td>
<td>2, 3</td>
<td>none</td>
<td>none</td>
<td>none</td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td>Development (*) Productivity</td>
<td>highest</td>
<td>lower</td>
<td>lower</td>
<td>higher</td>
<td>higher</td>
<td>lower</td>
<td>lower</td>
<td>lower</td>
</tr>
<tr>
<td>Improvement (*) On Software Reliability</td>
<td>highest</td>
<td>average</td>
<td>average</td>
<td>higher</td>
<td>average</td>
<td>average</td>
<td>average</td>
<td>average</td>
</tr>
</tbody>
</table>

Legend:  
(e) Engineering Feature  
(f) Formality Feature  
(*) Subjective Assessment

none: The feature does not exist for the method

** : the number (N) in the cells shows the satisfaction of attributes N of Table 2.1.
Aspects of Using the OO1 Tool Suite

System Engineering:
Define FMaps and TMaps
Analyze
Simulate real-time behavior and performance (Executor)

Software Development:
Define FMaps and TMaps
Analyze
Generate complete and production ready code
Execute on target machine

Design Change and Maintenance:
Revise FMaps and TMaps
Repeat system engineering
or software development process

Management:
Organize projects into working libraries
Manage and trace requirements
Generate product and process metrics
Generate specification, design and test documentation

Four Major Elements of the OO1 Tool Suite

(re)Define
Analyze
Execute
Generate

- Reusable Objects
- Manage/trace requirements and matrices with RT(x)

Figure 2.4 The OO1 Tool Suite Structure

Functional Map (FMap)

(Constraints:)

Control Structure  Function  Members of Data Types

Data Type  Relations

Type Map (TMap)

(Constraints:)

Figure 2.5 Reliable Building Blocks

Each model is defined in terms of functional hierarchies (FMaps) and type hierarchies (TMaps). All model viewpoints can be obtained from FMaps and TMaps including data flow, control flow, state transitions, data structure and dynamics. A higher level function is defined on a FMap in terms of its relationships to lower level functions. A higher level data type is defined on a
TMap in terms of its relationships to lower level data types. For each model, FMaps are inherently integrated with TMaps. Each function on a FMap has one or more objects as its input and one or more objects as its output. FMaps are used to define, integrate, and control the transformations of objects from one state to another state. Each object resides in an object hierarchy (OMap) and is a member of a data type as obtained from the TMap. The bottom nodes on a FMap contain primitive operations on types defined in the TMap. The bottom nodes on a TMap contain primitive data types. When a system has all of its object values used for a particular performance pass (i.e., it is executing) it exists as an execution hierarchy (EMap). With these hierarchies, all system viewpoints are integrated.

2.5.3 The OO1 Primitive Control Structures

The OO1 FMaps and TMaps are ultimately defined in terms of three primitive control structures. Structure decomposition overlays the abstract concept of control onto a network of primitive nodes where each node has input and output interconnections to other nodes. A formal set of rules is associated with each primitive structure. The three primitive control structures are: the Join (J), for defining dependent relationships, the Include (I) for defining independent relationships and the Or (O) for defining decision-making relationships. Their definitions and rules of using them are shown in Figure 2.6. With the use of these primitive structures, interface errors are "removed" before the fact by preventing them from occurring in the first place.
2.5.4 Reusability with the OO1-Defined Structures

Any system could be defined completely in terms of primitive structures, but there is often a desire to use less primitive structures to accelerate the process of defining and understanding a system. The degree to which a system has symmetry or asymmetry determines the number of reusable patterns that can be derived in terms of structures. Non-primitive structures can be defined in terms of the primitive structures or in terms of other non-primitive structures. Figure 2.7a, which is a small FMap, shows the definition of the non-primitive control structure Coinclude in terms of three J and one I primitive structures. Within the Coinclude pattern, A and B are the only bottom level functions that perform actions. This concept was created for defining reusable patterns. Included with each structure definition is the definition of the syntax for its use (see Figure 2.7b). In both Figure 2.7a and 2.7b, shadowed functions are to be re-defined by the user when he uses the Conclude structure, as shown in Figure 2.7c. This kind of use provides a "hidden repeat" of the entire system as defined, but explicitly shows only the elements that are subject to change (i.e., functions Decision, Clone2, Select_input_for_left, Select_input_for_right, Ida and Idb are all hidden in the simplified syntax structure shown in Figure 2.7b, with only functions A and B appear in syntax structure; In Figure 2.7c, the use of this Co-Include control structure results in actual function TaskA replacing A, and actual function TaskB replacing B). Each defined structure has rules associated with it for its use just as with primitive control structures. Rules for the non-primitives are derived from the rules of the primitives. The major point here is that, with the use of mechanisms such as defined structures, a system is defined inherently from the very beginning to maximize the potential for its own reuse. For some detailed explanations of the defined structures and examples of using them, please refer to Ref. (Hami93).
'JOIN' Control Structure

\[
\begin{align*}
&c, d = \text{Parent}(a, b) \land; \\
&c, d = \text{Left}(x_1, x_2) \land x_1, x_2 = \text{Right}(a, b);
\end{align*}
\]

Rules Governing Join (J):

1. Inputs to 'Parent', which are a and b, are identical to inputs of the right offspring 'Right', which are also a and b (including order).
2. Outputs from 'Parent', which are c and d, are identical to outputs of the left offspring 'Left', which are also c and d (including order).
3. Inputs to the left offspring 'Left', which are x1 and x2, are identical to outputs of the right offspring 'Right', which are also x1 and x2 (including order).

'INCLUDE' Control Structure

\[
\begin{align*}
O_1, O_2, O_3 &= \text{Parent}(I_1, I_2, I_3, I_4) \land; \\
O_1 &= \text{Left}(I_1, I_2, I_3); \\
O_2, O_3 &= \text{Right}(I_4);
\end{align*}
\]

Rules Governing Include (I):

1. A parent sends all its inputs to its children.
2. Order of Inputs and Outputs is Maintained.
3. Children send all their outputs to their parent.
4. Children do not share inputs or outputs.
5. Left Child 'Left' receives the first batch of 'Parent' inputs, which are I1, I2, I3.
6. Right Child 'Right' receives the rest of 'Parent' inputs, which is I4.
7. Left Child 'Left' sends its outputs to 'Parent' as the latter's first batch of outputs, which is O1.
8. Right Child 'Right' sends its outputs to 'Parent' as the rest of the latter's outputs, which are O2 and O3.

'OR' Control Structure

\[
\begin{align*}
O_1, O_2 &= \text{Parent}(I_1, I_2) \lor \text{Partition Function}(I_1, I_2); \\
O_1, O_2 &= \text{Left}(I_1, I_2); \\
O_1, O_2 &= \text{Right}(I_1, I_2);
\end{align*}
\]

Rules Governing Or (O):

1. Inputs of both offspring, 'Right', and 'Left', are identical to inputs of 'Parent', which are I1 and I2 (including order).
2. Outputs of both offspring, 'Right' and 'Left' are identical to outputs of 'Parent', which are O1 and O2 (including order).
3. Inputs of 'Partition Function' are identical to inputs of 'Parent', I1 and I2.

General Rule: Arrows show the directions of data flows.

Figure 2.6 The Three OO1 Primitive Control Structures
2.5.5 Definition of Objects with Parameterized Data Types

Reusability can also be employed within a TMap model by using parameterized data types. A parameterized data type is a defined structure that provides the mechanism to define a TMap without its particular relations being explicitly defined. Each parameterized type assumes its own set of possible relations for its parent and child types. Each parameterized data type has a set of primitive operations associated with it for its use. Abstract data types which decompose with the same parameterized data type in a TMap inherit the same primitive operations.

Figure 2.8 shows several examples of uses of parameterized types. They are explained as follows. First, the TupleOf data type is defined as a collection of a fixed number of possible different types of objects. For example, the data type 'RobotA' has a TupleOf data type structure and always has 6 attributes which include "MfgObject," "PickUpTime," "PutDownTime," "TurnTime," "Rotation," which are all of the natural number data type, and "Ports" which itself is a TupleOf data type.

Second, the OneOf data type is defined as a classification of objects of different data types from which only one data type is selected. For example, the data type 'APortIds' has a OneOf data type structure and can only take value of "Stock" , or "ConveyorA" , or "ConveyorB" , or "Grinder" , or "Parts" each time.

Third, the OsetOf data type is a collection of a number of variables of the same type of objects (in linear order). For example, the data type 'IOPorts' has a OsetOf data type structure, and includes an array of elements of the same data type -- 'IOPort' which itself is OneOf data type structure. Note that 'AtPort' is just another OsetOf data type with the same structure of 'IOPorts' already defined.
2.5.6 Constraint Specification and Analysis

Constraints can be defined for both FMaps and TMaps. The Meta-language properties of OO1 can be used to define global and local constraints. The validity
of constraints and their interactions with other constraints can be analyzed by either static or dynamic means. Figure 2.4 indicates that constraints are specified on both FMap and TMap. However, they are hidden under the Maps, and can only be handled (added or deleted) by opening separate Map editing windows. Generally, constraints are specified to impose some conditions on the performance of the system. For example, for a function, if we want to limit its running time to 5 second, then this limit could be considered as a constraint. The influence of this constraint is that it will affect all other functions which use this time-constrained function.

2.6 SUMMARY

In this Chapter, three major tasks are performed. First, the conventional software development process known as the 'waterfall' model is compared with an advanced formal development process model. For the latter, the places in the process where advanced formal techniques could be applied are identified. Second, an extensive literature study has been performed and is summarized concerning various established formal methods and approaches for software development. This literature survey plays an important role in this research. In this project, one part of the task is to select a most appropriate formal technique available in the software engineering field. Once selected, the technique is to be used in a real case of industrial program development for a nuclear power plant safety system. A set of criteria reflecting the concerns incorporated from system program designers, implementors and engineering system management groups are identified. These criteria are used to guide the selection activity. Finally, a formal method, Development Before The Fact (DBTF) and its supporting tool suite, the OO1 system, is selected for the use in this project. For the DBTF method, the
 characteristics of the formal specification language, design procedure, and tool suite are discussed in detail. Its major application features are also summarized. This prepared the stage for the next step in this research. That is to apply this method to address our initial concerns: namely, how to ensure the correctness and completeness of software functional design and how to obtain an error-free software specification from which code could be produced. With building systematic approach of using the selected method and tool, we also want to perform case studies to investigate to what extent we can achieve these goals via structuring the system design process.
CHAPTER 3

The Integrated Formal Approach

3.1 INTRODUCTION

In Chapter 2, a high quality currently available formal method and its supporting CASE tool are selected for use in the work reported here, based upon both the information gathered in an extensive literature survey and the set of selection criteria. As indicated in the Chapter 1, for this research, the purpose is to use the best features of this technique and its tool in order to explore the extent of possible achievement in producing zero-defect software for safety-critical applications.

This Chapter presents the design of an Integrated Formal Approach that systematically uses the features of the selected formal method and CASE tool in the safety-critical software development process. Before a detailed description of this proposed approach is given, a section of this chapter is devoted to the discussion of various kinds of defects that might occur at different stages of the software development process. This discussion helps one to understand better the elements of the proposed Approach, i.e., their purposes and possible achievements in reducing the numbers of software defects. The simple case of the Reactor Protection System design is used as a vehicle for the formulation of the Integrated Formal Approach, especially the first three major elements of the Approach: the Information Transforming Rules, the Specification Development Procedure and the Specification Prototyping Method. Finally, the major advanced features and fundamental contributions of the Approach are summarized.
3.2 SYSTEMATIC WAYS FOR THE REDUCTION OF DEFECTS

Terms associated with defects and distributions of defects in software development processes are discussed in this section. Various types of errors and defects are examined, and corresponding formal and informal techniques used to reduce those defects are also discussed in detail.

Some literature articles use the term "bug" to refer to software problems and mistakes that programmers encounter. Others (Schu90, Rook90, Musa89b) argued that the term "bug" is not appropriate since it implies that the bug is an independent life form that somehow crawled into your computer all by itself, and it is not your fault. Some more refined definitions have been advocated by those studies. These refined definitions are adopted in this research and are thus listed as follows:

**Error:**

An *error* is a conceptual, syntactic, or clerical discrepancy that results in one or more *faults* in the software. Errors could appear in the original software functional design and specifications. An error may result in one or more faults in the program code.

**Fault:**

A *fault* is a specific manifestation of an error, a discrepancy in the software code that can impair the program's ability to function as intended. It may appear in both the specification and the program code. Experience shows that sometimes faults might be identified by observation; however, their existence is usually not obvious to the users.
**Defect:**

A *defect* is another term used to refer either to an error or a fault in the program or specifications. Generally, it represents a discrepancy between the code and documentation that compromises testing, or produces adverse effects in code installation, modification, maintenance, or testing.

**Failure:**

A software *failure* occurs when a fault in the computer program is evoked by some valid input data, resulting in the computer program not correctly computing the required function exactly. For a safety-critical system, this failure may lead to a failure of a safety function that might introduce a disastrous consequence.

In short, the relationships between the above defined terms can be described in one simple sentence: *errors* create *faults* that cause *failures* upon using some valid input data. Here a simple example explains this relationship (Musa89). Suppose that a word-processing program does not produce italic characters when commanded, then that event is a failure. The underlying fault might be an instruction that passes control to a routine that generates standard characters, rather than to one for italics, when the italicizing command is invoked.

Based on the understanding of connotation and extension of each of the defined terms, their relationships could also be represented by a diagram as shown in Fig. 3.1. In this figure, dark horizontal lines show the casual relationships; While the fainter vertical lines represent the place in the development process where each item appears.
In this study, we prefer to use term 'defect' to describe any error or fault encountered in the development process.

3.2.1 Classification and Distribution of Defects

From many studies (Beiz90, Rook90, Schu90, Musa89, Ghez88, Mart85, Hami79), defects are observed to have following typical types:

A. Defects in Requirements and Specifications:
Defects may exist in the specifications of the requirements. These defects may include incomplete and incorrect designs of functions, and ambiguous definitions of functions and terms. Also, insufficient precision of calculations might be specified; and hardware descriptions might be incorrect. Sometimes when a correct design has been changed, the corresponding specifications might not be correctly updated. Defects in specifications of requirements are
a major source of expensive defects. The range of its portion in the total
distribution of defects is from a few percentages to more than 50%,
depending on the applications and environment(Schu90). This kind of defect
is usually found when a code has been developed and users have started
running the program, thus they are expensive to correct.

B. Defects in control logic, decision logic and sequencing
This group of defects is sometimes called structural defects. In the
conventional way of programming, these defects include execution paths left
out, unreachable code with incorrect labels, improper nesting loops, incorrect
loop termination criteria, missing process steps, misunderstanding of logic
operators and case statements, improper negation of Boolean expressions,
improper simplification and combination of cases, and overlap of exclusive
cases.

C. Processing defects
These defects include arithmetic defects, algebraic and mathematical
function evaluation defects, algorithm selection defects, and general
processing defects. Many of them are related to the incorrect data
conversions from one representation to another.

D. Data flow and interface defects
These defects include accepting an initial value without a validation check on
the data; forgetting to initialize working place; data type mismatches between
inputs and outputs of functions, and incorrect values for loop control
parameters.
E. Coding defects

Coding errors (defects) of all kinds can create any of the other kinds of defects. Syntax errors are generally not important if the source language translator has adequate syntax checking. A good translator will also catch: undeclared data, undeclared routines, dangling code, and many initialization problems. Given good source-syntac checking, the most common pure coding errors are typographical, followed by errors caused by not understanding the operation of an instruction or statement. Some are purely incorrect documentation.

One study on basic cause categories of defects in software programs (Schu90) provides statistical data on the distributions of above discussed defects. The result of this study is shown in Fig. 3.2.

![Figure 3.2 Distribution of Defect Classes](image)

Figure 3.2 Distribution of Defect Classes
Generally, all defects could be divided into two groups: one group is called syntax defects and another is called semantic defects. Syntax refers to how something is being said, or sentence constructions and the grammatical arrangement of program code. Only type E defects, coding defects, identified above belong to this group. Semantic refers to what is being said, or the meaning of the language. It is further divided into internal semantics and external semantics. Internal Semantics relates to whether what is being said obeys the rules established in the basic axioms. External Semantics relates to whether the system is solving the right problem. According to this study, all data defects in the type D are internal semantic defects; a part of logic defects of type B and a part of specifications and requirements defects of type A are also internal semantic defects. External semantic defects include all processing defects of type C, a part of Requirement and Specification defects of type A and a part of logic defects of type B.

Here is the right place to point out that in this study, some other kinds of defects are not to be treated. The first kind is related to the defects of total input information of the system program. Since this kind of defects are more related to the engineering system design, the engineering system designers are more responsible for them. The second kind is related to the compilers for producing object code for real time running of the program. The third kind of defects is related to the supporting CASE tool itself.

3.2.2 The Way to Reduce the Frequency of Defects

Modern software engineering practice sees ever more validation and verification activities upon the specifications and programs. However, most of
these activities are not based upon formal mathematical rules and are carried out manually most of the time. The IBM's Cleanroom approach that includes manual verification for decomposition of operations to ensure the correctness of functional designs is just one of the examples.

It can be said that the DBTF formal specification technique and OO1 Tool Suite, which supports rapid system prototyping, automatic verification of program development, and automatic code generation, are really large advances toward the capability for zero-defect software production. The basic idea behind the DBTF method is that verification ought to take place at every stage in a design process, and it allows for rigorous verification at each step of the design process. The DBTF technique and OO1 tool together provide four stages of verification in order to eliminate internal syntax defects, internal semantic defects, and external semantic defects. The following discussion describes how these four stages work.

At the first stage of the DBTF technique, the OO1 Analyzer checks the format of data type specifications in the TMap and the format of functions and statements in the FMaps. This stage catches internal syntax defects from both Maps. The OO1 Analyzer works like a compiler or an interpreter in the conventional programming environment.

At the second stage of the DBTF technique, in the FMaps, the OO1 Analyzer checks the input and output relationships between parent functional statements and their dependent offspring functional statements. This ensures that the control maps can be decomposed into primitive operations and primitive control structures that obey the rules of consistency. Those rules are established in the basic axioms defined in the OO1 system. Using the TMap, the Analyzer checks to ensure that abstract data types are correctly structured and also obey the rules of consistency. Work at this stage catches some internal semantic defects that might
be all sorts of logical defects, data processing defects and original design requirement defects.

At the third stage of the DBTF technique, in the FMaps, the *Analyzer* is used to check the leaf node functional statements and to find interface defects and recursion defects in them. Those interface defects may be incorrect linkages to other operations or defined structures. The main causes for these interface defects are the inconsistent uses of data types and references to nodes not yet defined. Also, based upon the result of analysis of the TMap at the first stage of work, all primitive operations upon leaf nodes are checked for their validation and consistency with the data types defined in the TMap. This stage of work catches another part of the internal semantic defects that include all data related defects, some of the data processing defects and some the design requirement defects.

Once the third-stage defects are eliminated, all specified modules and module-integrated programs can be linked and run by the OO1 *Executor*. The *Executor* allows developers to prototype both modules and programs in order to observe the characteristics from the original design of their behavior and to check upon whether they provide the expected results. Using this fourth level of verification, developers can find any hidden defects in the external semantics. However, as indicated in references (Hami94, Mart85a), this fourth level of verification is not always guaranteed to capture all external semantic defects. This is because these external semantic defects involve the conceptual problems or problems of changing requirements that could have not been caught by mathematical verification (The previously stated one kind of not-treated defects, the defects in the information of total inputs of system program, belongs to that part of external semantics). A strategy of test case selection for complete functional prototyping of the specifications should be required in order to improve
the effectiveness of this fourth-level of verification in treating the external semantic defects. This matter is discussed in Section 3.4.3.

3.3 A SIMPLE SYSTEM DESIGN DOCUMENTATION

In this Section, a simple system design is presented, which illustrates the original documentation of the requirements for designing a microprocessor-based prototype safety protection system for a research nuclear reactor. Here the system is simply called as Reactor Protection System (RPS). This software design has been used as the example case for study in prior research (Bloo86, Froo88) for discussing the application of VDM formal method.

Most of time, the original requirements of a system design are documented using plain text plus some complementary forms of information summaries, like Data Flow Diagrams (DFD), Logical Flow Diagrams (LFD), etc., or sometimes simply in pseudo-program structured text. No matter how they are structured, the plain text in natural language is always ambiguous. A DFD is merely a better structured form of documentation than text for presenting detailed information of data inputs, processes and data storage of the target system. An LFD could provide a better profile of the entire system design logic than could text. None has built-in mathematical formalities to ensure unambiguity and consistency.

The following is the original documentation of the RPS design requirements in plain text and a DFD as shown in Fig. 3.3.

- Functionally, the program is quite simple. It takes a number of pairs of input signals (up to 40), with each pair representing the trip condition of a particular plant parameter (TRIP or OK) and corresponding veto condition (VETO or OK) of the parameter.
• It provides a guard line output signal. If a non-vetoed trip is present in the input signals, the signal latches and the guardline trips. It remains tripped until it is reset.

• A same number (equal to that of inputs) of output signals, going to indicator lights is also provided, corresponding to the trip inputs, each of which latches "ON" if the equivalent input is in the trip state, whether vetoed or not, and is "OFF" otherwise. The outputs are initialized with all indicators being in the "OFF" state.

Various forms of data flow diagrams have been discussed in the literature (Mart85, Rand90). The data flow diagram in Fig. 3.3 is read in the following way. It represents operation being carried out on data in a particular state. In this operation, the data state is represented by data stores that are on the right-hand side of the diagram (as in shown Fig. 3.3) and represented graphically by two horizontal parallel lines joined by one vertical line. Inputs to the system are represented by external entities, shown in circles on the left-hand side of Fig. 3.3. Processes are represented by the squares in the middle of the diagram. The movement of data around the system is shown by arrows called, data flows. A data flow from a process to a data store is interpreted as showing a change of state of the data store.
3.4 THE PROPOSED INTEGRATED FORMAL APPROACH

This section presents the Integrated Formal Approach (IFA) designed from this study. The case of RPS design is used to explain all of the design details of this approach and to show how it can be applied in a practical case. The approach is called an Integrated approach because it covers all program designing activities from engineering requirement analyses to quantitative assessment of the quality of the program specifications. This approach consists of four elements

1. A set of Rule for transforming information from original engineering design documents into the basic elements of a formal specification language;

2. A Procedure for developing complete and unambiguous specifications by employing formal specification techniques to the transformed information (Applying this procedure will also help to identify any internal syntactical and semantically defects in original design);
(3) A Strategy for rapid prototyping in which complete testing cases are selected from the specification to use in testing the specifications and in detecting any external semantic defects in the original design; and

(4) Automatic program generation from the CASE tool.

The Figure 3.4 describes the interfaces among these IFA elements. Each IFA element is described in detail in the following sections.

![Figure 3.4 Schematic Diagram of The Integrated Formal Approach](image)

In applying this approach, it should be noted that the first three IFA elements have strong interactive relationships. It is expected that development of specifications using the Procedures will identify some design defects in the original documents. This information needs to be passed to the program designer immediately in order to permit a check of its validity. The rapid prototyping activity in the third IFA element might identify still more defects in specifications that, although they would have passed the CASE tool's analyses, can deliver unexpected results of system behavior. This last information is also given to the designer to find out whether identified defects are due to original design or due to the specifications itself. When the behavior of the specifications is satisfied after
several cycles of interactions among first three blocks of activities, the target
source code is then automatically generated by the CASE tool.

It must be stated here that the devised IFA offers the promises of
eliminating Logic Defects, Processing Defects, Coding Defects, Data-Interface
Defects and Requirement and Specification Defects. It can not affect defects
arising from Input, Compilers and the CASE tool itself.

The major expected benefit of using the IFA in restructuring design process
are that automation potentially will permit elimination of defects otherwise
introduced after specification development, and the discipline and feedback
provided by use of a formal method's CASE tool during specification development
can potentially eliminate defects arising there.

3.4.1 Rules for Transforming Information

This section explains, by means of RPS example, how to use the Rules
developed in this study to translate text (structured, unstructured, or pseudo code
structured) and data flow diagrams into the basic elements of the DBTF formal
specification. The DBTF's OO1 structure is described in Figure 2.3.

The basic elements of a specification include are as follows:
(1) primitive data types and abstract data types needed to construct the unique
TMap. This TMap must specify the object types of unique states of the
system, and the object types of the inputs and outputs of the program;
(2) names (or labels) and data types of those variables used by each process in
the original design;
(3) names (labels) of those processes performing data manipulation -- these
names (labels) will be used as operation names in the specification;
(4) design logic of the entire program and each process in it.
In a text document, a process is usually defined in a sentence using a verb as the identifier. Design logic could be identified by looking for a logical phrase such as "IF .... THEN .... ELSE .... ". State variables, input variables and output variables are given as nouns in the sentence. In data flow diagrams, the transformation activity becomes simpler. This is because the names (labels) of the system state, the input and output variables, and the processes are mostly presented in the diagram. However, the variable types and the design logic of the system still need to be determined from the accompanying text documents. For example, in a DFD, the labels of data stores are usually the components of the system state; the labels of the data flow lines are the input and output variables; and the labels of processes could be used for the names of operations.

We designed the following set of rules that should be followed in order to transform the information in the original documents into the basic elements of the OO1 formal specifications:

- **Rule 1:** Obtain the names for *operation, internal functions* of the program, names for *input* and *output variables* and names for *data storage* from the original design, proceeding in the following way:

  (A) **If** the original document is in text form, then the uses of all verbs should be studied to see whether they are independent actions which serve as internal functions; also, uses of all nouns have to be studied carefully to see whether they belong to *input/output variables*, or *data storage*;

  (B) **If** the original document is in a DFD and text form, then from the DFD, identify labels of the processes, the external entities, the data flows and the data stores. Those labels are used for the three classes of names mentioned above.
Rule 2: Identify the data type of each variable and of each data store from the detailed descriptions of the text documents; The four major primitive data types are the following: Natural number, Rational number, Boolean value, String. Other complex data types are structured data types made from the primitive data types.

Rule 3: Obtain the design logic of the entire program and its internal functions from the original document. This is done in the following way:

(A) If the entire original document is in text form, then to look for such logical phrases as "if...", "whether...", and "if ... then ... else ..." in the sentences that contain verbs used for names of internal functions or of the program itself;

(B) If a part of the original document is in the pseudo program structured text form, then look for such logical structure as "if ... then ... else ... " structures after each verb;

(C) If a part of the original document is in the form of a logic flow diagram, then all decision nodes are considered to contain the design logic.

Rule 4: Eliminate any ambiguities in the definitions of operations, and internal functions, for all variables, and data types as given in text documents; and resolve any inconsistencies in using the same names for different functions and operations in the original document.

By following these rules, the information obtained from RPS documents are listed as follows.
The results from Rule 1 are as follows:

Program Operation: Update Data Store: *Update*;

Input: *Preen*; Output: *Newstate*;

Internal Functions: 1) Update Indicators: *Update_Indicators*;

Input: *OldIndicators*; Outputs: *NewIndicators*;

2) Update Guardline: *Update_GL*;

Input: *State1, Localtrip*; Output: *State2*;

Data Store: *Guardline; Indicators*.

The results from Rule 2 are as follows:

Data Types: *Guardline* = \{ Trip, Ok\};

*Indicators* = Set of Indication;

*Indication* = \{On, Off\};

*Signals* = Set of Signal;

*Signal* = \{TripCondition, VetoCondition\};

*TripCondition* = \{ Trip, Ok\};

*VetoCondition* = \{ Veto, Ok\};

*System State* = \{ Guardline, Indicators, Signals \}.

The results from Rule 3 are as follows:

Operation:

*Update* = *Update_Indicators* + *Update_GL*;

(the order of the two internal functions is not clear)
Internal Function:

\[ Update\_Indicators = \text{Update all indications in the indicator set; } \]
\[ \text{Indication}_i = \text{On, if } \text{Signal}_i, \text{TripCondition} = \text{Trip} \]
\[ \text{Else, } = \text{OFF} \]

\[ Update\_GL = \text{Check all input signals in the signal set} \]
\[ \text{Guardline} = \text{Trip, if for one or more signal:} \]
\[ \text{TripCondition} = \text{Trip and VetoCondition} = \text{Ok} \]
\[ \text{Else} = \text{OK} \]

The results from Rule 4 are as follows:

The action "latch" is ambiguous. It is taken to be equivalent to setting the guardline to the state "Trip".

### 3.4.2 Procedure for Constructing Specifications

In this Section, the IFA Procedure for constructing specifications is presented. Initially, the overview of the entire Procedure is given. Then detailed explanations and rationales for each step are provided, along with the results of application of each step of the Procedure in treating the RPS example case. This Procedure consists of following seven steps:

**Step I.** Construct the TMap of the system based upon the basic elements gathered in applying the Transforming Rule 2. This TMap should include all data types of system state and all of the identified variables; Then, compare the TMap with the design documents in order to validate it. Finally, analyze the TMap using the OO1 Analyzer in order
to test its internal consistency, and to install the resulting consistent TMap into the example case's OO1 environment.

**Step II.** If necessary, develop an FMap of an operation that specifies the set of valid program states. This map serves as the "data type invariance" structure of the system. These valid states are derived combinatorially from all elements of the system state data types. The size of the set might be less than the size of all possible combinations. Then, analyze the this FMap for consistency and install it into the example case's OO1 environment.

**Step III.** Use an independent FMap to construct the specification of the entire system program using the information from Transforming Rules 1 and 3. This information includes the labels of internal functions, inputs and outputs of the operations and all internal functions, and the design logic of the operation and all of the internal functions. The "data type invariance" of the FMap developed in Step II is also used here as one of the internal functions.

**Step IV.** Specify the individual internal function modules using separate FMaps, based upon the information gathered from Transforming Rules 1 and 3.

**Step V.** Perform a series of checks on the individual modules, then analyze the individual modules using the OO1 Analyzer to eliminate internal syntax and semantic defects, and to install all modules into the example case's OO1 environment.

The following is the series of needed checks:
Check #1  For each decision node having a partition function, identify all Boolean variables, and find the corresponding abstract functions or primitive functional statements at the leaf nodes that assign values to these Boolean variables;  

Check #2  If any leaf node function (from the Check #1 operation) assigns a Boolean value to its output variable by processing an input variable of "OneOf" type, then one needs to make sure that there is the same number of child-functions that correspond to the different actions of each "OneOf" type input variable. These child-functions should follow the partition function node that uses the Boolean output of this leaf node.  

Check #3  For sibling child-functions decomposed by the "O" or the "CO" control structure, it is necessary to check whether the same input variables have been processed in an equivalent number of actions, but logically in an opposite manner; if not, then one needs to add any more actions;  

Check #4  For sibling child-functions decomposed by the "O" or the "CO" control structure, one needs to check whether the same outputs have been obtained through an equivalent number of actions but logically in an opposite manner; and to add any more actions if necessary;  

Check #5  Continue to check other partition function-controlled decomposition of abstract functions using the same ordered checking, using Checks #1, #2, #3, and #4, until the end of the module is reached; and then make all of the necessary modifications;
Step VI. Apply the same series of Checks to the specifications of the operation of the entire program developed in Step III, then analyze the specification using the OO1 Analyzer to eliminate any internal defects, and install it into the example case's OO1 environment.

Step VII. Link all of the installed modules together into an executable specification of the program that is ready for the subsequent rapid prototyping activity.

3.4.2.1 The Reason for Step I and the RPS TMap

The advantage of structuring the program's TMap at this early stage of specification is that it not only elucidates all of the types of data objects to be used in the software, but it also explicitly states the entire system data structure. This structure can then be discussed, refined, agreed upon and completed in a process potentially involving all groups of the system designing team. This system data structure uses the object-oriented characteristics of the DBTF methodology, which states the inheritances of the data objects. This TMap is unique in its mathematical expression for the system, and thus eliminates the ambiguities and inconsistencies of the system data structure that may appear in the original English documentation. It also aids the collection of all data elements for use in the structured operation. With the RPS example, the developed TMap is shown in Figure 3.5.
3.4.2.2 The Reason for Step II and the RPS Data Type Invariance

At Step II of the Procedure, it is necessary to ensure that the system is always in a valid state both before and after an operation. Doing this is equivalent to specifying the properties of the system that are to hold invariantly both before and after the operation. This is the concept of Data Type Invariance (Jone80). If the system data type invariance does not hold, then the specification of the operation is proved to be wrong. This provides one way to check the correctness of the specification of the operation. The input to the specification of the data type invariance is the object defined in (instantiated from) the TMap of the system. The operational outputs define the complete set of valid states of the system.

The analysis for the RPS program shows that there is no requirement for data type invariance in this program. Any state of system resulting from any of the input data states is valid.

3.4.2.3 The Reasons for Steps III and IV

The purpose of Step III is to specify the entire program's design logic. If the program is very large, then a better means of specification is modulization, where all internal functions are identified as individual operational modules to be
specified in other separate FMaps. The FMap of the operation of this program only defines the data and logic interfaces among modules.

In Step IV, all of the modules are specified in detail in their respective separate FMaps. In this way, the processes of specifying each internal function and of performing independent validation and verification of functions and operation become much easier. This also aids the task of testing the specifications immediately after the development effort. The OO1 environment provides a Road Map (RMap) to show the interactions between operation and modules of the internal functions.

3.4.2.4 The Reasons for Steps V and VI, and the RPS Results

Although in Steps III and IV, the OO1 Analyzer is constantly used to eliminate internal syntax and internal semantic defects, it is still possible that some external semantic defects might exist in the specification. They are beyond the control of the Analyzer. This means that the program might not do what we expect of it. Due to the characteristics of the DBTF specification techniques, the critical places for checking these external defects are at the abstract function decomposition nodes controlled by partitioning functions. The reason for this is that these nodes give the logic of the specifications. The correctness of the specification logic is an important part of the functional correctness of the program.

At the stage of developing specifications, there is an advantage to using the consistent and installed TMap to ensure the completeness of the specification logic at those nodes. That is the reason to have this series of fine checks.

The Checks #1 and #2 together will treat the 'Case Statement' and 'Multivalued Logic' situations. Usually in handwritten programs, a function might be missing due to missing logic, or incomplete treatment for default logic in
multivalued logic statements, or the confusing structure of nested logic statements, e.g., statements such as "if --- then --- else if --- then --- else if ---" that are controlled by some arbitrarily selected variables that provide insufficient controls upon logical completeness. The OO1 specification techniques provide that means of control.

The following simple example illustrates what the checks #1 and #2 could do. Let a variable CarColor have the following data type structure as specified in one segment of the overall OO1 TMap and shown in Figure 3.5a.

![Diagram of CarColor data type structure]

Also, a Boolean variable chk1 is assigned a value by the following OO1 primitive functional statement

\[ \text{chk1}=\text{is:Red:Color(CarColor)}. \]

The value of chk1 will be true if the variable CarColor is holding the value "Red", and false otherwise. Now, in the following partition function node which is a piece of the OO1 FMap specification as shown in Figure 3.5b, the variable chk1 is used to decompose the abstract function "Do_Something_To_Car" into child functions, "Treat_on_NotRed" and "Treat_on_Red" (here the names of the abstract function and the child functions are created as symbols to represent some different actions to be done in these functions. They are just symbols, nothing else).
The important things in this piece of OO1 FMap specification is the overall control structure controlled by the partition function "CO"Copy:Boolean(chk1). According to the OO1 control structure rules, the left child function, "Treat_On_Red", corresponds to the 'chk1 = true' condition, and the right child function, "Treat_On_NotRed" corresponds to the 'chk1 = false' condition. Is this the logically complete decomposition? Asking this question at any partition function node is always legitimate. The Checks #1 and #2 should identify any defect at this point.

In the TMap, CarColor could have any one of three possible values, i.e., Black, Red, and White. Thus, the complete specification of the abstract function "Do_Something_To_Car" should have three child functions, namely, "Treat_On_Black", "Treat_On_Red", and "Treat_On_White", as shown in Figure 3.5c.
This will make a complete logical and functional specification. After Checks #1 and #2 are performed, three sub-functions "Treat_On_White", "Treat_On_Black", and "Treat_On_Red" are specified. Checks #3 and #4 are then applied to these sub-functions to ensure mutually exclusivity.

Fig. 3.6 shows the specification of the RPS program's operation, 'Update', from developmental Step III and the Checks of Step VI. In this figure, leaf node functions with "-op-" marks are the modules specified in other FMaps. Figures 3.7 to 3.10 show the specifications of the modules from developmental Step IV and the Checks of Step V. These FMaps are all constructed in the OO1 environment. The circled numbers identifying the nodes are put on there for later use in analyzing the specifications in order to generate test cases for them.

![Diagram](image_url)

Figure 3.6 The Reactor Protection System Program FMap

3.4.2.5 Experience From Applying the Seven-Step Procedure

In developing each FMap from Step III to Step VI, the OO1 Analyzer is constantly used to detect any internal syntax and semantic defects in the specifications. Typically these defects are (a) typographical errors in input/output variables of each abstract function; (b) redundant uses of the same name for different variables in different functions; (c) redundant use of the same name for different functions; (d) failure to use an input variable in the decomposition of the abstract functions; (e) use of variables in a fashion not consistent with the control structure; (f) typographical errors creating undefined primitive functions in the
specifications; (g) different ordering of the variables in the same named functions which appear at different places (this is especially important with the "-R-" type of recursive functions); (h) unspecified or inconsistent user-defined structures and operations used as modules in other FMaps. Once these defects are identified by the OO1 Analyzer, they can be eliminated immediately. The RPS example shows that the development Procedure is more effective in eliminating those defects than conventional programming practice, where such defects must be corrected at a later development stage of using the compiler.

3.1.2.6 Data Summary of the Developed RPS Specification

The entire specifications of the RPS are presented in Figures 3.5 to 3.10. At this point, a set of quantitative data can be summarized from the specifications. These include (1) the total number of each primitive data type, and the total numbers of each abstract data type; (2) for the program's operation, the total number of modules, the total number of primitive statements, and the total number of partition functions; (3) within each module, the total numbers of sub-modules, the total number of primitive statements, and the total numbers of partition functions. Table 3.1 and 3.2 summarize these data. These data reveal the complexity of the specifications, and are used later in deriving complete test cases for the specifications.
Figure 3.7 FMap Specification of the Module "Update_Indi"
Figure 3.8 FMap of the Defined Structure "SetIndiUpdate"

Figure 3.9 FMap Specification of Module "CheckGLOK"
State 1 = Update_GL(PreState, NewSignals) CJ*2

Guard = KF: Boolean(PreState);

GL_Trip = Update_1(Guard, NewSignals) CJ*2

Set 1 = Locate: Sgl_Inputs("1", NewSignals);
Set 2 = AtNull: Sgl_Inputs(Set 1);

GL_Trip = DoAllTillFinish(Set 1, Guard, Set 1) CO: Or: Boolean(Guard, Set 1);

GL_Trip = Do Next Element(Set 1) CJ*8, J;

E = MoveTo: Sgl_Inputs(Set 1);
Tc 1 = MoveTo: TripCond: Signal(E);
Chktrip = Is: TRIP: TripCond(Tc 1);
Vc 1 = MoveTo: VetoCond: Signal(E);
ChkVeto = Is: OK: VetoCond(Vc 1);
Localtrip = And: Boolean(Chktrip, Chkveto);
Set 2 = MoveTo: Sgl_Inputs: Signal(E);
Set n = Next: Sgl_Inputs("R", Set 2);
Set 1 = AtNull: Sgl_Inputs(Set n);

GL_Trip = DoAllTillFinish(Set n, Localtrip, Set 1) CJ*8, J;

GL_Trip = Clone 1: Any(Guard);

State 1 = Update_2(GL_Trip, PreState) CO: Copy: Boolean(GL_Trip);

State 1 = Clone 1: Any(PreState);

State 1 = Do_Switch(PreState) CJ*3

GL 1, State 2 = Get: Guardline: Sgl_State(PreState);
Bo 1 = D: Guardline(GL 1);
GL 2 = K: TRIP: Guardline(Bo 1);

State 1 = Put: Guardline: Sgl_State(GL 2, State 2);

Figure 3.10 FMap Specification of Module "Update_GL"
Table 3.1. The RPS Specification Data Summary - I: TMap

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Total Number</th>
<th>Names of Defined Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>TupleOf</td>
<td>2</td>
<td>Sgl_State, Signal</td>
</tr>
<tr>
<td>OSetOf</td>
<td>2</td>
<td>Indicators, Sgl İnput:s</td>
</tr>
<tr>
<td>OneOf</td>
<td>4</td>
<td>Indication, Guardline, TripCond, VetoCond.</td>
</tr>
<tr>
<td>Value</td>
<td>5</td>
<td>ON', OFF', OK', TRIP', VETO'.</td>
</tr>
</tbody>
</table>

Table 3.2. The RPS Specification Data Summary - II: FMaps

<table>
<thead>
<tr>
<th>Modules</th>
<th>Name</th>
<th>Parent</th>
<th>Sub-Modules</th>
<th>No. of Primitive Functional Statements</th>
<th>No. of Partition Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operation</td>
<td>Update( )</td>
<td></td>
<td>#2. Update Indi #3. CheckGLOK #5. Update_GL</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Modules</td>
<td>Update_Indi</td>
<td>Update( )</td>
<td>Defined Structure: SetIndiUpdate( )</td>
<td>22</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>CheckGLOK</td>
<td>Update( )</td>
<td></td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Update_GL</td>
<td>Update( )</td>
<td></td>
<td>18</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>SetIndiUpdate</td>
<td>Update_Indi</td>
<td></td>
<td>11</td>
<td>1</td>
</tr>
</tbody>
</table>

3.4.3 Strategy for Testing Specifications

In Section 3.2.2, in the discussion of reducing the frequency of defects using the DBTF technique and its OO1 tool, the four-stage verification technique is discussed. As identified, 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? Taking the specifications as the object, this action will test all of the functions in it to ensure that their designs are complete and consistent with what is expected. The OO1
system provides the testing facility to help perform this action. With the testing
gility provided, the specifications are executable at an earlier stage of design and
enable one to check the behavior of the designed system. Any deviation of the
behavior from what is expected will be noted and analyzed further. The identified
defects can then be corrected in the specifications. The modified specifications go
through the first three stages of verification again to ensure their correctness and
consistency.

In the proposed Integrated Formal Approach of this study, the Strategy for
testing specifications addresses the following concerns:

- What are the types of functions that need to be tested in the entire
  specifications, and how are they identified? Should "black-box" testing
  techniques (function oriented) be used alone or in combination with "white-
  box" testing techniques (structure oriented)?

- In each module (unit), how are the testing cases selected in order to cover all of
  the identified functions completely?

- How is the integrated testing to be conducted for the entire specification of the
  program? Should it be a top-down integration approach, or a bottom-up
  integration approach?

In this section, a literature survey is performed in order to understand the
functional testing techniques necessary for testing specification. Based upon both
this literature survey and our understanding of the characteristics of the DBTF
formal specification treatment, a test case selection method is devised. This
method is presented in Sections 3.4.3.2 and 3.4.3.3. The method employs the
multiple condition coverage criterion and newly defined equivalence classes of
paths coverage criterion to achieve complete functional testing of the specification. Five types of DBTF formal language functional statements are identified as potentially having defects. These five types of functional statements include (a) arithmetic operations; (b) data assignments; (c) order of child-functions using the OO1 "O" and "CO" control structures; (d) uses of Boolean variables for partition functions; and (e) partition function variable assignments. The effectiveness of the devised test cases selection method is also studied, in Section 3.4.3.3.5.2 by deliberately introducing a single defect for each type of statements (b), (c), (d) and (e). The results of the testing shows that these introduced defects are quickly identified and removed.

3.4.3.1 Literature Survey on Functional Testing Techniques

Since the 1970s, much research has been conducted concerning functional testing for software. Most of the functional testing methods rely on effective and easy identification of the functions specified. Some are based variously upon narrative documentation, informally developed specifications using a structured method, and formal specifications such as VDM. Others are based upon the actual program code and upon the analyses of the structure of the program. So far, the number of effective tools and techniques for functional testing is small. Below, several representative research results are briefly summarized to illustrate some fundamental and stimulating works in this area.

Traditional design-based functional testing techniques have involved examining a (usually narrative) specification and carrying out a number of "good testing practices" such as those described by Myers (Myer79). For example, equivalence partitioning involves dividing the input domain into a number of equivalence classes such that it can be assumed that a representative value from a class is equivalent to a test of any other value. However, in order to identify the
equivalence classes from such a narrative specification may be very arbitrary and difficult.

Bauer (Baue79) has developed a test case generation technique that uses a regular grammar derived from the functional specification. The formal specification takes the form of a finite-state automaton (which tends to favor on-line environments) from which both input and output sequences of grammars are generated.

A more formal approach is Howden's design-based functional testing (Howd80). Design functions are those required for implementing a set of requirements. They may be composed of further design functions (i.e., a program is not seen as a monolithic entity, but rather as a hierarchical synthesis of smaller design functions). The functions are chosen according to the criterion that they be independently testable. Data are then chosen for use in testing each of these functions and combinations of functions.

Partition analysis, or revealing sub-domains, (Rich81) is an example of a "hybrid" method, since it involves a combination of both structural testing (usually at the code level) and functional testing. The input domain of the program is partitioned into structural equivalence classes of data, each of which should be treated in the same way for structural testing of the program, according to the program specification. The input domain is then again divided into functional equivalence classes which are treated in the same way by the actual program for program's functional testing. The intersection of these two domains provides a number of sub-domains from which test data may be selected.

Budd (Budd85) has applied the technique of program mutation testing to the area of specifications. The specification is written in a predicate calculus using a notation chosen to show more clearly the input-output relationships. Changes are
then introduced into the specification and the results of these changes are then examined.

Some researchers have recognized the difficulty of developing functional tests from narrative specifications and have turned to more formal presentations. Bouge (Boug86) generates test sets from algebraic data type specifications and uses an extension of the PROLOG programming environment to aid testing of the functions in the specifications. Scullard (Scul88) attempted to develop a procedure for extracting test cases from the VDM formal specifications of an operation. In doing this, the VDM specification is stated in the first order predicate calculus. This procedure involves analyzing the signatures of functions down to basic functional types and taking into account the effects of any function pre-condition or data type invariants. From the analyses of the domain of the reduced signature, obtained by taking each of the basic types and extracting a set of typical values as test data, all of the different ways are identified in which the post-condition may be satisfied.

Dick's research (Dick93) studies an automatic technique for partition analysis in state-based specifications, especially using VDM. There, test domains for individual operations are calculated by the reduction of the mathematical description to a Disjunctive Normal Form. Following this, a partition analysis of the system state can be performed which permits the construction of a Finite State Automaton from the specification. This in turn can be used to sequence the required tests in a valid and sensible way.

Roper (Rope93) in his research, presents a set of techniques that concentrate on the analysis and design stage to generate test data. Specifically, the test data are drawn from the informally developed specifications in the forms of a logical data model, the dataflow model and the entity-event model. Data flow diagrams are analyzed in a way that generates tests to exercise the passage of each
set of input data through the system. This information is augmented with details from the process (function) specifications to ensure that the specifications themselves are tested down to the branch coverage level. For an entity-event model of the system, the entity life histories are examined and tests are generated to evaluate the full range of possible states experienced by the entities within the system. The result is a common pool of test data requirements and expected results that may be used in tests at the corresponding development stage.

In all of these above cited research results, the testing techniques are mostly referred as "black-box" testing techniques since the tests are performed on specifications, although a few of them looked into part of the internal structure of the specification. However, other researchers, as presented below, approach functional testing focusing upon the developed program, not the specifications.

Howden (Howd81) developed a method for functionally testing programs which aims to make up for the deficiencies in specifications and relies far less on the tester's intuition than do other methods. In this method, a criterion for completely testing elementary program functions is set up. Five types of program statement functions are identified (generally applicable for any program) and reliable complete function test cases are derived from these identified statement functions.

A program path consists of the sequence of program operations used in taking an input data set and operating upon it until a valid output data set is formulated. Test path selection based upon a program is a well-studied field and a number of criteria may be used in attempting to devise a strategy that will provide a level of coverage of the set of program paths. The simplest technique of choosing paths is to inspect the graph of program paths manually as described by Myers (Myer79) and Adrion (Adri82). The fundamental work on the completeness of the coverage, achieved based upon program graph is given by Prather (Prat83). There,
he indicates that generally there are five types of coverage: (1) statement coverage: which executes all statements in the graph; (2) node coverage: which encounters all decision node entry points in the graph; (3) branch coverage: which encounters all exit branches of each decision node in the graph; (4) multiple condition coverage: which achieves all possible combinations of condition outcomes at each decision node of the graph; and (5) path coverage: which traverses all paths in the graph. These five types are related in their degree of coverage as shown in Figure 3.11, with weaker criteria being at the bottom and the stronger criteria at the top of the figure:

![Figure 3.11 Relationship Among Five Coverage Criteria](image)

Later, Howden (Howd86) examined the functional approach to program testing and analysis by studying the elementary functional forms or complex structures in the program. Three kinds of forms discussed there are algebraic, conditional, and iterative; while structural synthesis of functions includes function sequence structure and state transition structure. This approach simply avoids the difficulty of dealing with paths, and is entirely focused upon the analysis of forms and structures as indicated above.

3.4.3.2 Selecting Program Functions for Testing

The OO1 Tool Suite-supported formal specifications have their own unique character that is different from those of other forms of specifications. From the
subsequent example of the RPS specifications, we see that specifications not only have various functional statements at the leaf nodes, but also have clearly structured forms for connecting those functional statements. So the system specification cannot be said completely "black".

Based upon this characteristic, in this study, a unique strategy for systematically conducting complete functional testing on the specification is formulated. This strategy uses both the function-analysis technique from the "black-box" testing and structural-coverage technique from the "white-box" testing. The principles of this strategy are presented first. The ways of using the two important components of this strategy, Identification of functional statements, and Derivation of test cases for complete testing coverage of all functional statements, are presented later in this and the next Section respectively.

In all of the studies cited above, only that of Howden (Howd86) discusses both identifying functional statements and considering functional forms and structures synthesis in connecting those identified function statements to derive a test data set. His subject, however, is a program. In other discussions concerning identifying functional statements and generating test data from specifications, like that of Parrington (Parr89), no systematic ways of generating test data from the identified function statements are provided. Rather, only arbitrary combinations of the tests are performed according to tester's whim and compulsiveness. In the specification, the important information of the structural connections of those functional statements is either not explored, or not used. However, in our work, with the unique character of OO1 based specifications being identified, a strategy for using both function statements and the OO1 control structures is formulated in order to conduct an efficient and complete functional test of the specifications.

Then, how are the functional statements to be used for testing in the OO1 specification identified? From the works of Howden (Howd81) and Parrington
the following five types of functional statements are important for the testing of specifications:

- **Data Access**: concerned with the function that reaches a piece of data;
- **Data Storage**: concerned with the function that stores a piece of data;
- **Arithmetic Expression**: concerned with the function which involves arithmetic operations (addition, subtraction, multiplication, exponentiation, etc.)
- **Arithmetic Relation**: concerned with the function which takes the form of \( X \mathrel{r} Y \)
  where "r" is one of the relational operators, < (less than), > (greater than), = (equal to), <= (less than or equal to), >= (greater than or equal to), <> (not equal to), and X and Y are constants, variables or the results of an arithmetic expression;
- **Boolean Expression**: concerned with the function which combines Boolean operands with the Boolean operators (AND, OR, NOT) to yield a Boolean result.

In any OO1 specification FMap, like the RPS specification shown in Figures 3.6 to 3.10, we can also identify the above five types of functional statements and select them for testing. Their formats and the way to identify them are discussed below.

In the OO1 system environment, for each primitive data type and user-defined abstract data type (in the form of "OneOf", "OSetOf", "TupleOf" ) used in the TMap specification, there is a set of primitive operation formats which must be used. For the RPS specifications shown in Figures 3.6 to 3.10, the five types of functional statements are identified and summarized for the RPS specification as follows:
Data Access functional statements and Data Storage functional statements
(Due to the OO1 system's unique style of specification, when a data accessing function is used, a new output variable is created in order to store the data accessed, thus any data access function is also a data storage function.) These functions are:

All primitive functions in the format:
"Output = Moveto:ChildNode:Parent(Input)";

All primitive functions in the format:
"Output = Clone1:Any(Input)";

All primitive functions in the format:
"Output = Identify1:Any(Input1, Input2)";

All primitive functions in the format:
"Output = Put:Child:Parent(Childnode, Input)";

All primitive functions in the format:
"Output = Locate:SETNode("1",SetName)";

All primitive functions in the format:
"Output=Next:SETNode("R",SetName)";

All primitive functions in the format:
"Output=AtNull:SETNode(SetName)"

and also the functions as follows:
Size1=Size:Indicators(Ind1);
Size2=Size:Sgl_Inputs(NewInputs);
Disp1=Display:Str("", Message1);
Cutoff,IndNew=Get:Indicators(Ind001);
Create=K:ON:Indication(kill);
Guard=KF:Boolean(PreState);
GL2=K:TRIP:Guardline(Bo1).

- **Arithmetic Expression functional statements:**
  No arithmetic functional statements exist in the RPS specification.

- **Arithmetic Relation functional statements:**
  Arithmetic Relation functional statements include:
  
  - Chksize1=GE: Nat(Size1, Size2);
  - Chksize2=LE: Nat(Size1, Size2);
  - Chk1=Is: OFF: Indication(E1);
  - Chk2=Is: OK: TripCond(InputTC);
  - Chk1=Is: OK: Guardline(OldGL);
  - Chktrip=Is: TRIP: TripCond(Tc1);
  - ChkVeto=Is: OK: VetoCond(Vc1).

- **Boolean Expression functional statements:**
  Boolean Expression functional statements include:
  
  - Partition function: CO: Copy: Boolean(ChkGL);
  - Partition function: CO: And: Boolean(Chksize1, Chksize2);
  - Partition function: CO: Copy: Boolean(Chk1);
  - Partition function: CO: Copy: Boolean(Chk2);
  - Partition function: CO: Copy: Boolean(Chkend);
  - Partition function: CO: Or: Boolean(Guard, Setck);
  - Partition function: CO: Copy: Boolean(GLtrip);
  - Localtrip=And: Boolean(Chktrip, Chkveto).
In specifying the RPS requirements, it is noted that after the Procedure of seven development steps (presented in Section 3.4.2) is applied to the development of the system specifications, the specifications might still have external semantic defects hiding in some functional statements in the leaf nodes. These defects will not be found in analyses by the OO1 Analyzer, because the internal syntax and semantics of the erroneous functional statements will agree with those of the OO1 AXES language. If this is true, then the system's behavior might deviate from designer's intention and remain undetected. The causes of these defects might be design errors in the original documents, or incorrect interpretations of the original documental information, or merely sloppiness by the specifier in editing the specifications into the OO1 Maps.

In this study, based upon the entire RPS specification, these defects are identified to include the following forms and involve the following functional statements:

(a) *Incorrect arithmetic operations*: For example, such defects can include cases where instead of addition, multiplication may be used; the wrong number may be added to, or subtracted from, or multiplied with, or divided by another variable.

For the RPS example, no functional statements are identified in this group.

(b) *Incorrect data assignments*: For example, such defects can include cases of wrong data access, wrong data value assignment to a variable.

Functional statements of the RPS potentially having these kinds of defects are:

Create=K:ON:Indication(kill);
Guard=KF:Boolean(PreState);
GL2=K:TRIP:Guardline(Bo1).

(c) *Wrong order of child-functions using the "O" and "CO" control Structures*;
With these defects, the left and right child functions are switched in position, thus, leading to a completely different specification logic.

Functional statements in the RPS potentially having these kinds of defects are:
Partition function: CO:Copy:Boolean(ChkGL);
Partition function: CO:And:Boolean(Chksize1, Chksize2);
Partition function: CO:Copy:Boolean(Chk1);
Partition function: CO:Copy:Boolean(Chk2);
Partition function: CO:Copy:Boolean(Chkend);
Partition function: CO:Or:Boolean(Guard, Setck);
Partition function: CO:Copy:Boolean(GLtrip).

(d) *Wrong Boolean variables used for partition function(s)*; With such defect, an incorrect Boolean variable is assigned as a functional argument.

Functional statements in the RPS potentially having this kind of defect are:
Partition function: CO:Copy:Boolean(ChkGL);
Partition function: CO:And:Boolean(Chksize1, Chksize2);
Partition function: CO:Copy:Boolean(Chk1);
Partition function: CO:Copy:Boolean(Chk2);
Partition function: CO:Copy:Boolean(Chkend);
Partition function: CO:Or:Boolean(Guard, Setck);
Partition function: CO:Copy:Boolean(GLtrip);
(e) Partition function input variables (Boolean type) having values assigned incorrectly; With such defect, an incorrect value is assigned to a correct Boolean variable.

Functional statements in the RPS potentially having this kind of defect are:

`Chksize1=GE: Nat(Size1, Size2);`
`Chksize2=LE: Nat(Size1, Size2);`
`Chk1=Is: OFF: Indication(E1);`
`Chk2=Is: OK: TripCond(InputTC);`
`Chk1=Is: OK: Guardline(OldGL);`
`Chktrip=Is: TRIP: TripCond(Tc1);`
`ChkVeto=Is: OK: VetoCond(Vc1);`
`Localtrip=And: Boolean(Chktrip, ChkVeto).`

All other types of functional statements are not the subjects for functional testing. This is because the rigid format requirements of the strong type-matching and semantic checking of the OO1 Analyzer prevents any defect occurring in those functional statements to escape detection.

3.4.3.3 How to Select Testing Cases for A Module

For each of the identified functional statements, we can derive a set of testing cases in order to examine its effect upon the behavior of the system. Here we start with each RPS module that includes the identified functional statements to be tested.
3.4.3.3.1 Identifying the Basic Testing Cases

There are three modules in specification of the "Update" operation, as shown in Figure 3.6. They are Module #2: Update_Indi, Module #3: CheckGLOK and Module #5: UpdateGL.

[A] Module #2: Update_Indi

Chksize1=GE:Nat(Size1, Size2); --- statement #24

The test cases for this statement are:

Size1>Size2, and Size1=Size2 for Chksize1=True;
Size1<Size2 for Chksize1=False.

Chksize2=LE:Nat(Size1, Size2); --- statement #25

The test cases for this statement are:

Size1<Size2 and Size1=Size2 for Chksize2=True;
Size1>Size2 for Chksize2=False.

Partition function: CO:And:Boolean(Chksize1, Chksize2); --- statement #25'

The test cases for this statement are:

Chksize1=True, Chksize2=True;
Chksize1=Fault, Chksize2=True;
Chksize1=True, Chksize2=Fault;
Chksize1=Fault, Chksize2=Fault.

Partition function: CO:Copy:Boolean(Chkend); --- statement #284'

The test cases for this statement are:

Chkend=True;
Chkend=Fault.

Chk1=Is:OFF:Indication(E1);  --- statement #2880
The test cases for this statement are:
   Indication=OFF;
   Indication=ON.

Chk2=Is:OK:TripCond(InputTC);  --- statement #2882
The test cases for this statement are:
   InputTC=OK;
   InputTC/TRIP.

Partition function: CO:Copy:Boolean(Chk1);  --- statement #2882'
The test cases for this statement are:
   Chk1=True;
   Chk1=False.

Partition function: CO:Copy:Boolean(Chk2);  --- statement #2883'
The test cases for this statement are:
   Chk2=True;
   Chk2=False.

Create=K:ON:Indication(kill);  --- statement #2888
The test cases for this statement are:
   K:ON:Indication(kill);
   K:OFF:Indication(kill).
[B] Module #3: CheckGLOK

Chk1=Is:OK:Guardline(OldGL); --- statement #32

The test cases for this statement are:

OldGL=OK;
OldGL=TRIP.

[C] Module #5: Update_GL

Guard=KF:Boolean(PreState); --- statement #50

The test cases for this statement are:

KF:Boolean(PreState);
KT:Boolean(PreState).

Partition function: CO:Or:Boolean(Guard, Setck); --- statement #53'

The test cases for this statement are:

Guard=True, Setck=True;
Guard=False, Setck=True;
Guard=True, Setck=False;
Guard=False, Setck=False.

Chktrip=Is:TRIP:TripCond(Tc1); --- statement #57

The test cases for this statement are:

Tc1=OK;
Tc1=TRIP.

ChkVeto=Is:OK:VetoCond(Vc1); --- statement #59
The test cases for this statement are:

\[ \text{Vc1}=\text{OK}; \]
\[ \text{Vc1}=\text{VETO}. \]

\[ \text{Localtrip}=\text{And:Boolean}(\text{Chktrip}, \text{Chkveto}); \quad \text{--- statement #5a} \]

The test cases for this statement are:

\[ \text{Chktrip}=\text{true, Chkveto}=\text{True}; \]
\[ \text{Chktrip}=\text{False, Chkveto}=\text{True}; \]
\[ \text{Chktrip}=\text{True, Chkveto}=\text{False}; \]
\[ \text{Chktrip}=\text{False, Chkveto}=\text{False}. \]

Partition function: \[ \text{CO:Copy:Boolean}(\text{GLtrip}); \quad \text{--- statement #5e'} \]

The test cases for this statement are:

\[ \text{GLtrip}=\text{True}; \]
\[ \text{GLtrip}=\text{False}. \]

\[ \text{GL2}=\text{K:TRIP:Guardline(Bo1); \quad \text{--- statement #5j}} \]

The test cases for this statement are:

\[ \text{K:TRIP:Guardline(Bo1)}; \]
\[ \text{K:OK:Guardline(Bo1)}. \]

So, for all functional statements of the five types potentially having defect, test cases for each individual statements are obtained. The total number of individual test is 40. However, these test cases do not tell anything about the specification. They have to be combined together to cover the complete functional
testing of the specification. The ways of combining them is related to the
discussion of selection of criterion for complete testing below.

3.4.3.3.2 Selecting A Criterion for Complete Testing

Simply listing all of the test cases obtained from all of functional statements
is not enough for testing completely. We also need to know the connections
between those functional statements, so that, in the testing campaign, they can be
combined systematically. That is the reason that we need to investigate the
structure of the OOI specifications.

As shown in Figure 3.11, Prather(Prat83) pointed out that there are
generally five path selection criteria for program structure analyses. For the OOI
method, specifications are developed as graphs subject to strict mathematical rules.
We recognize in this study that, with OOI FMaps, "node" coverage is equivalent
to "branch coverage" coverage because "node" and "branch" refer to the same
partition function. Thus, only four criteria are applicable in the OOI
specifications. These four path selection criteria are:

1. Statement coverage: executing all of the statements (blocks) in the graph.

2. Branch coverage: encountering all of the exit branches of each decision
   node in the graph.

3. Multiple condition coverage: achieving all of the possible combinations of
   condition outcomes at each decision node of the graph.

4. Path coverage: traversing all of the paths in the graph.

Their relationships in terms of strength of coverage remains the same as
shown in Figure 3.11. As an example illustrating the different requirements of
these various criteria, consider the portion of the RPS specification shown in
Figure 3.12.
The specification of Figure 3.12 is presented in a more visually "structural" diagram in Figure 3.13.

In order to achieve complete statement coverage, two tests:

A=2, B=0, X=1 for statements 1, 2, 2', 4, 5, 6, 6', 7;

A=3, B=1, X=2 for statements 1, 2, 2', 3, 5, 6, 6', 7

are sufficient. Notice that statements 1, 2, 2', 5, 6, and 6' are tested twice.

For complete branch coverage of this example, however, at least two input data tests are required, for example,

A=3, B=0, X=3 for the left branch of 2' and the right branch of 6'.

This also applies to statements 1, 2, 2', 4, 5, 6, 6', 3

A=2, B=1, X=2 for the right branch of 2' and the left branch of 6'.

This also applies to statements 1, 2, 2', 3, 5, 6, 6', 7.
According to the complete multiple condition coverage criterion, there are \((2^2+2^2)=8\) possible outcomes which can be achieved by different combinations for the two simple input conditions at each decision node. These may be satisfied, for example, by the selection of four separate tests, as with the conditions

- \(A=3, B=0, X=3\) for \(1=\text{True}, 2=\text{True}, 5=\text{False}, 6=\text{False}\)
- \(A=3, B=1, X=2\) for \(1=\text{True}, 2=\text{False}, 5=\text{False}, 6=\text{True}\)
- \(A=1, B=0, X=2\) for \(1=\text{False}, 2=\text{True}, 5=\text{True}, 6=\text{True}\)
- \(A=1, B=1, X=1\) for \(1=\text{False}, 2=\text{False}, 5=\text{True}, 6=\text{False}\).

We can see that all eight combinations of the multiple conditions are covered, since for the first Boolean expression in node 2', we have the combinations \{\((\text{True, True}), (\text{True, False}), (\text{False, True}), (\text{False, False})\)\}, and for the second Boolean
expression in node 6', we have the combinations \{(False, False), (False, True), 
(True, True), (True, False)\}.

For the complete path coverage, we also have four test cases, e.g.,

A=2, B=0, X=4 for path 1, 2, 2', 4, 5, 6, 6', 7;
A=3, B=0, X=3 for path 1, 2, 2', 4, 5, 6, 6', 3;
A=2, B=1, X=2 for path 1, 2, 2', 3, 5, 6, 6', 7
A=3, B=1, X=1 for path 1, 2, 2', 3, 5, 6, 6', 3.

It is observed from these examples that statement coverage by itself
provides a rather weak strategy for testing, representing necessary but by no means
sufficient criteria for complete structural test. The branch coverage criterion,
however, implies the complete coverage of all statements. The stronger criteria of
the multiple condition coverage and the path coverage together imply complete
program branch coverage. Finally, from the union of the test cases selected above,
seven tests from multiple condition coverage and path coverage are enough to
completely cover the test of the functions of the specification. So in theory, the
union of sets of test cases from multiple condition coverage and path coverage
implies the complete test case for the specification. Here for the example discussed
above, we have the following seven test cases:

1. A=3, B=0, X=3;
2. A=3, B=1, X=2;
3. A=1, B=0, X=2;
4. A=1, B=1, X=1;
5. A=2, B=0, X=4;
6. A=2, B=1, X=2;
7. A=3, B=1, X=1.

3.4.3.3 Defining the Equivalence Classes of Paths

In fact, complete path testing is really difficult only when a loop exists in the graphic structure. Sometime testing of all paths that could be identified from the loop-involved structure is practically impossible, simply because the path number is too big. So in practice, a path testing criterion is usually relaxed to the extent that only equivalence classes of paths are represented for testing. Prather (Prat83) proposes a definition of equivalence classes as "two paths are considered 'equivalent' if they differ only in their number of loop traversals". For the OO1 RPS specifications of this study, we find a different way to define the equivalence classes. The following discussion shows how to identify the equivalence classes of paths in the OO1 specification for testing.

Identifying equivalence classes of paths involves two steps. These steps are best introduced by an example. Take module #2 shown in Figure 3.6, Update_Indi, as the example. First, a structural graph could be built using its specifications as shown in Figures 3.7 and 3.8. Second, from this structural graph, path equivalence classes are identified using the new definition produced from this study.

Obtaining the structural graph is equivalent to transforming the specification shown in Figures 3.7 and 3.8 into a statement expansion tree as shown in Fig. 3.14 below.
Figure 3.14. Expansion Tree of Specifications of Module "Update_Indi" as Shown in Figures 3.7 and 3.8

By taking out nodes on the left-hand sides of all equal symbols (except for the nodes of Recursive Functions), replacing them with the structures on the right-hand sides, and connecting recursive nodes, we have the detailed specification structure that includes all functional statements of "Update_Indi" module shown in Figure 3.15.
Fig. 3.15 is further simplified by collapsing consecutive statements that are represented by nodes into one large node. The condition for this collapsing is that there be no decision node among those consecutive nodes. In this way, we obtain Figure 3.16 showing the simplified structural graph of the specification, Update_Indi.
In Figure 3.16, there are four decision nodes (A, D, F, H) which lead to alternative branches, one branch-joining node (K), and one loop starting with the node D and ending with the node K.

As far as we have determined in this study, in all OO1 specifications the loop structure is only used to traverse an ordered set. Generally there are two types of loops: "While" loops and "For" loops.

From Figure 3.16, we can identify the following two paths: \{A-C-D-F-G-K-D-E\}, and \{A-C-D-F-G-K-D-F-G-K-D-E\}. According to the classic path definition (Beiz90), they are considered to constitute different paths. However, their only difference is the number of loop traversals. The first path has a single loop-traversal on the decision-ordered branch \{D-F-G-K\}, while the second has
two traversals on the same decision-ordered branch, i.e., \{D-F-G-K-D-F-G-K\}. In this study, these two paths are considered to be in the same equivalence class. This means that the functional tests of these two paths are not different.

Another case of an equivalence class of paths can be seen in the Figure 3.16. Consider the following three paths:


These three paths all traverse two decision-ordered branches \{D-F-G-K\} and \{D-F-H-I-K\}. The only difference between the first path and the second path is the order of the traversal of the two decision-ordered branches. Between the first and the third path, the difference is the number of loops through the combination of the same two decision-ordered branches. These three paths are considered to be in the same equivalence class.

Thus, according to the above analyses, the following definition is produced from this study for the new concept of equivalence classes of paths:

**Definition:** In paths involving loop traversals, two paths are considered to be in the same equivalence class if they either:

1) Traverse the same decision-ordered branch but only with a different number of traversals; or

2) Traverse any combinations of the same set of decision-ordered branches, but with a different number of traversals in those combinations.

By following this definition, the nine representative equivalence classes of paths for the module "Update_Indi" are identified as follows:
(1) A-B
(2) A-C-D-E
(3) A-C-D-F-G-K-D-E
(4) A-C-D-F-H-I-K-D-E
(5) A-C-D-F-H-J-K-D-E

3.4.3.3.4 Deriving A Complete Set of Test Cases

In Section 3.4.3.3.2, we concluded that criteria of multiple condition coverage and path coverage could be used in combination to derive the set of test cases to be used for achieving complete path test coverage. Using the union of the test cases derived from each criterion is the correct way to obtain the desired test set. The complete set of test cases for each of the three modules of the RPS specifications are derived as follows.

[A] Module #2: Update_Indi

For the "Update_Indi" module, the nine equivalence classes of path have already been derived. In Section 3.4.3.3.1, the functional statements to be tested have also been identified. Here, the multiple conditions stated in those functional statements are combined with the equivalence paths in order to derive the complete testing cases for this module. From Figure 3.16, we can identify the paths to be covered:
Path (1): A-B Path A-B includes the following sequence of functional statements: 21, 22, 23, 24, 25, 25', 261, 262, 263, 264. Its corresponding multiple condition is the "False" value from the Boolean expression in statement #25'. Thus, it includes the following three conditions:

i. \( \text{Size1}<\text{Size2} \) from statement #24, \( \text{Size1}\leq\text{Size2} \) from statement #25; for satisfying both conditions, one testing case is required: \( \text{Size1}<\text{Size2} \);

ii. \( \text{Size1}\geq\text{Size2} \) from statement #24, \( \text{Size1}>\text{Size2} \) from statement #25; for satisfying both conditions, one testing case is required: \( \text{Size1}>\text{Size2} \);

iii. \( \text{Size1}<\text{Size2} \) from statement #24, \( \text{Size1}>\text{Size2} \) from statement #25; for satisfying both conditions, no testing case exists, as these two conditions are mutually incompatible.

So finally for path #1, we have the following two cases for test:

Test #1: \( \text{Size1}>\text{Size2} \); for it, we may have, arbitrarily, \( \text{Size1}=2, \text{Size2}=1 \);
Test #2: \( \text{Size1}<\text{Size2} \); for it, we may have, arbitrarily, \( \text{Size1}=1, \text{Size2}=2 \).

Path (2): A-C-D-E Path A-C-D-E includes the following sequence of functional statements: 21, 22, 23, 24, 25, 25', 281, 282, 283, 284, 284', 28d, 29. Its first multiple condition at statement 25' requires the "True" value, which has only one condition:

i. \( \text{Size1}\geq\text{Size2} \) from statement #24, \( \text{Size1}\leq\text{Size2} \) from statement #25; for satisfying both conditions, one testing case is required: \( \text{Size1}=\text{Size2} \).

The second multiple condition of this path at statement 284' requires the "True" value which means that the value of both indicators and signals are zero. i.e., \( \text{Size1}=\text{Size2}=0 \). So finally for path #2, we have one test case:

\( \text{Size1}=\text{Size2}=0 \).
Path (3): A-C-D-F-G-K-D-E  Path A-C-D-F-G-K-D-E includes the following sequence of functional statements: 21, 22, 23, 24, 25, 25', 281, 282, 283, 284, 284', 286, 287, 2880, 2881, 2882, 2882', 2883, 289, 28a, 28b, 28c, 283, 284, 284', 28d, 29. Its first multiple condition at statement 25' requires the "True" value, which has only one condition: Size1=Size2.

The second multiple condition of this path at statement 284' is executed twice: it takes the value "False" the first time that statement 284' is executed, then the value "True" at the second time that 284' is executed. This means that the value of the indicator set is equal to 1, i.e., Size1=1.

The third multiple condition in this path at statement 2882' requires the "False" value, which, from the arithmetic relation in statement 2880, means E1.Indication=ON.

There is no requirement on how signal conditions are to be defined. Thus we can set the new input signal conditions in one of the following four ways:

(TripCond=OK, VetoCond=OK)
(TripCond=TRIP, VetoCond=OK)
(TripCond=OK, VetoCond=VETO)
(TripCond=TRIP, VetoCond=VETO).

Finally, we obtain one test of the following input data setting:

Size1=Size2=1;
PreState.Indicators={ ON }; PreState.Guardline= either TRIP or OK
NewInputs= { (Any one of the four ways shown above) }.

For the multiple conditions at statement 2882', the "True" value is now required. From the arithmetic relation in statement 2880, this requires that E1.Indication=OFF.

For the multiple conditions at statement 2883', the "True" value is required. From the arithmetic relation in statement 2882, this requires that E2.TripCond=OK. There is no requirement on the value of E2.VetoCond, so it could take values of either OK or VETO.

Finally, we obtain a single test case having the following input data setting:

\[
\begin{align*}
\text{Size1}=\text{Size2}=1; \\
\text{PreState.Indicators} &= \{ \text{OFF} \}; \\
\text{PreState.Guardline} &= \text{either TRIP or OK} \\
\text{NewInputs} &= \{ \text{OK, either VETO or OK} \}.
\end{align*}
\]


For the multiple conditions at statement 2882', the "True" value is now required. From the arithmetic relation of statement 2880, this means that E1.Indication=OFF.
For the multiple condition at statement 2883', the "False" value is required. From the arithmetic relation in statement 2882, this means that E2.TripCond=TRIP.

There is no requirement concerning the value of E2.VetoCond, so it could take either the OK or VETO setting.

Finally, we obtain a single test case having the following input data setting:

\[ \text{Size1}=\text{Size2}=1; \]
\[ \text{PreState.Indicators}=\{ \text{OFF} \}; \text{PreState.Guardline}=\text{either TRIP or OK} \]
\[ \text{NewInputs}= \{ (\text{TRIP, either VETO or OK}) \}. \]


The second multiple condition at statement 284' is executed three times: The value "False" is required the first and second time that 284' is executed, and the value "True" in the third execution. This means that size of the indicator set is 2, i.e., Size1=2.

The third multiple condition at statement 2882' is executed twice: the value "False" is taken at the first, and value "True" at the second time of execution. This means that the first Indication=ON, and the second Indication=OFF. It also means that the first signal type can be any one of the four types given in the Path #3 analyses.
The fourth multiple condition at statement 2883' requires the "True" value. Since statement 2883' appears after the second 2882' execution, it means that second signal should have the condition TripCond=OK. It is noticed that the second signal could take either the value of VetoCond=Veto, or VetoCond=OK.

Finally, we obtain a single test case having the following input data setting:

\[ \text{Size1}=\text{Size2}=2 \]

\[ \text{PreState.Indicators} = \{ \text{ON}, \text{OFF} \}; \text{PreState.Guardline} = \text{either TRIP or OK} \]

\[ \text{NewInputs} = \{ \text{(any one of four types)}, \text{(OK, either VETO or OK)} \} \].

Path (7): A-C-D-F-G-K-D-F-H-J-K-D-E  Path A-C-D-F-G-K-D-F-H-J-K-D-E includes the following sequence of functional statements: 21, 22, 23, 24, 25, 25', 281, 282, 283, 284, 284', 286, 287, 2880, 2881, 2882, 2882', 2883, 289, 28a, 28b, 28c, 283, 284, 284', 286, 287, 2880, 2881, 2882, 2882', 2883', 2885, 2886, 2887, 2888, 2889, 288a, 289, 28a, 28b, 28c, 283, 284, 284', 28d, 29. Following the same arguments as in the Path #6 for those four multiple conditions, with noticing the difference that after the statement 2883' comes the statement 2885, not 288b as in Path #6. Finally we obtain a single test case having following input data setting:

\[ \text{Size1}=\text{Size2}=2 \]

\[ \text{PreState.Indicators} = \{ \text{ON}, \text{OFF} \}; \text{PreState.Guardline} = \text{either TRIP or OK} \]

\[ \text{NewInputs} = \{ \text{(any one of four types)}, \{\text{TRIP, either VETO or OK}\} \}. \]

Following the same method of analyses, we obtain a single test case having the following input data setting:

\[ Size1 = Size2 = 2 \]

\[ \text{PreState.Indicators} = \{ \text{OFF, OFF} \}; \ \text{PreState.Guardline} = \text{either TRIP or OK} \]

\[ \text{NewInputs} = \{ \text{(OK, either VETO or OK)}, \ (\text{TRIP, either VETO or OK}) \}. \]


21, 22, 23, 24, 25, 25', 281, 282, 283, 284, 284', 286, 287, 2880, 2881, 2882, 2882', 2883, 289, 28a, 28b, 28c, 283, 284, 284', 286, 287, 2880, 2881, 2882, 2882', 2883', 288b, 289, 28a, 28b, 28c, 283, 284, 284', 286, 287, 2880, 2881, 2882, 2882', 2883', 2885, 2886, 2887, 2888, 2889, 288a, 289, 28a, 28b, 28c, 283, 284, 284', 28d, 29.

The first multiple condition at statement 25' requires the value "True", which has only one condition: Size1=Size2.

The second multiple condition at statement 284' is executed four times: the value "False" is required the first, second and third times of execution, and the value "True" in the fourth execution. This means that size of indicator set is equal to 3, i.e., Size1=3.

The third multiple condition at statement 2882' is executed three times: the value "False" is required the first, and the value "True" at the second and third times of execution. This means that the first Indication=ON, second Indication=OFF, and third indication=OFF. It also means that the first signal type can be any one of the four types given in the Path #3 analyses.
The fourth multiple condition at statement 2883' is executed twice: the value "True" is required in the first, and the value "False" in the second time of execution. Since statement 2883' appears after the second and third execution of statement 2882', it means that second signal should have the condition TripCond=OK, while third signal should have the condition TripCond=TRIP. It is noticed that the second and the third signals could take either the condition VetoCond=Veto, or VetoCond=OK. Finally, we obtain a single test case having following setting:

\[ \text{Size1=Size2=3} \]

\[ \text{PreState.Indicators = \{ ON, OFF, OFF \}}; \]

\[ \text{PreState.Guardline = either TRIP or OK} \]

\[ \text{NewInputs = \{ (Any one of four types discussed above),} \]

\[ \text{(OK, either VETO or OK),} \]

\[ \text{(TRIP, either VETO or OK).} \]

The Table 3.3 summarizes the results of the test using the ten testing cases identified above. The testing results are compared with the expected results, and comments are provided.

[B] Module #3: CheckGLOK

Analysis of the module #3, operation "CheckGLOK", follows the same treatment as with module #2. However, since the module #3 is very simple in its structure, its two test cases could thus be simply observed and selected as:

Test Case #1: \[ \text{State1.Guardline=OK, State1.Indication = \{ N \} N=0,1,...} \]

Test Case #2: \[ \text{State1.Guardline=TRIP, State1.Indication = \{ N \} N=0,1,...} \]
Table 3.3  Summary of Selected Set of Test Cases and Testing Results for Module #2 "Update_Indi"

<table>
<thead>
<tr>
<th>Test Case &amp; ID.</th>
<th>Test Result</th>
<th>Expected Result</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test 1:</td>
<td>Warning Message! No Update Action!</td>
<td>Warning Message! No Update Action!</td>
<td>Correct Result for Size1 &gt; Size2</td>
</tr>
<tr>
<td>Indicators = [ON,OFF ]; Guardline=OK; Signals = ([OK, OK]).</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Test 2:</td>
<td>Warning Message! No Update Action!</td>
<td>Warning Message! No Update Action!</td>
<td>Correct Result for Size1&lt; Size2</td>
</tr>
<tr>
<td>Indicators = [ON]; Guardline=OK; Signals = ([OK,OK],[OK,OK]).</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Test 3:</td>
<td>System Interrupt stops the running of program with rejection status of the 001 system.</td>
<td>No Message At all!</td>
<td>Later examination showed that the specification does not provide statements for Size1=Size2=0 operation.</td>
</tr>
<tr>
<td>Indicators = {} ; Guardline=OK; Signals = { }.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Test 4:</td>
<td>Indicators = [ON]; Guardline=TRIP; Signals = ([TRIP, OK]).</td>
<td>Indicators = [ON]; Guardline=TRIP; Signals = ([TRIP, OK]).</td>
<td>This is one test case from path #3. Correct result from test.</td>
</tr>
<tr>
<td>Indicators = [ON ]; Guardline=TRIP; Signals = ([TRIP, OK]).</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Test 5:</td>
<td>Indicators = [OFF ]; Guardline=TRIP; Signals = ([OK, VETO]).</td>
<td>Indicators = [OFF ]; Guardline=TRIP; Signals = ([OK, VETO]).</td>
<td>This is one test case from path #4. Correct result from test.</td>
</tr>
<tr>
<td>Indicators = [OFF ]; Guardline=TRIP; Signals = ([OK, VETO]).</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Test 6:</td>
<td>Indicators = [ON ]; Guardline=Ok; Signals = ([TRIP, VETO ]).</td>
<td>Indicators = [ON ]; Guardline=Ok; Signals = ([TRIP, VETO ]).</td>
<td>This is one test case from path #5. Correct result from test. Indicators change value.</td>
</tr>
<tr>
<td>Indicators = [OFF ]; Guardline=OK; Signals = ([TRIP, VETO ]).</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Test 7:</td>
<td>Indicators = [ON, OFF]; Guardline=TRIP; Signals=([OK,OK],[OK,OK])</td>
<td>Indicators = [ON, OFF]; Guardline=TRIP; Signals=([TRIP,OK],[OK,OK]).</td>
<td>This is one test case from path #6. Correct result from test. No Changes!</td>
</tr>
<tr>
<td>Indicators = [ON, OFF]; Guardline=TRIP; Signals=([OK,OK],[OK,OK])</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Test 8:</td>
<td>Indicators = [ON, ON ]; Guardline=TRIP; Signals = ([OK, VETO), (TRIP, OK)].</td>
<td>Indicators = [ON, ON ]; Guardline=TRIP; Signals = ([OK, VETO), (TRIP, OK)].</td>
<td>This is one test case from path #7. Correct result from test. Indicators change value.</td>
</tr>
<tr>
<td>Indicators = [ON, OFF ]; Guardline=TRIP; Signals = { (OK, VETO), (TRIP, OK) }.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Test 9:</td>
<td>Indicators = [OFF, ON ]; Guardline=TRIP; Signals = ([OK,OK],[TRIP, VETO]).</td>
<td>Indicators = [OFF, ON ]; Guardline=TRIP; Signals = ([OK,OK],[TRIP, VETO]).</td>
<td>This is one test case from path #8. Correct result from test. Indicators change value.</td>
</tr>
<tr>
<td>Indicators = [OFF, OFF ]; Guardline=TRIP; Signals = { (OK,OK), (TRIP, VETO) }.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Test 10:</td>
<td>Indicators = [ON, OFF, OFF ]; Guardline=OK; Signals = { (TRIP, VETO), (OK, VETO), (TRIP, OK) }.</td>
<td>Indicators = [ON, OFF, ON ]; Guardline=OK; Signals = { (TRIP, VETO), (OK, VETO), (TRIP, OK) }.</td>
<td>This is one test case from path #9. Correct result from test. Indicators change value.</td>
</tr>
<tr>
<td>Indicators = [ON, OFF, OFF ]; Guardline=OK; Signals = { (TRIP, VETO), (OK, VETO), (TRIP, OK) }.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Module #5: Update_GL

The operation "Update_GL" of module #5 is analyzed in the same way as with module #2. First, the expansion tree is constructed from the module's specification (shown in Figure 3.10), and is shown in Figure 3.17.

![Expansion Tree of Operation "Update_GL"
As Shown in Figure 3.10](image)

The simplified structure of Figure 3.17 is shown in Figure 3.18.

![Simplified Structure of Operation "Update_GL"
As Derived From Figure 3.17](image)

Legend

<table>
<thead>
<tr>
<th>Statement</th>
<th>Detailed Statements</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>50 – 53</td>
</tr>
<tr>
<td>B</td>
<td>53'</td>
</tr>
<tr>
<td>C</td>
<td>55 – 53'</td>
</tr>
<tr>
<td>D</td>
<td>5e – 5e'</td>
</tr>
<tr>
<td>E</td>
<td>5h – 5k</td>
</tr>
<tr>
<td>F</td>
<td>5f</td>
</tr>
</tbody>
</table>
The representatives of all equivalence classes of path in Fig. 3.18 are obtained as follows:

(1) A-B-D-F
(2) A-B-C-B-D-F
(3) A-B-C-B-D-E.

Following the same methods of path analyses as with the module "Update_Indi", it is seen that the Path A-B-D-E can not occur. This is because, with zero traversal of the only loop (B-C-B) in the structure, the multiple condition of the Boolean expression at statement 53' results in statement 5e' always having the "False" value, which takes one to F, never to E. So this path is not included in the analysis.

For the multiple conditions at statement 5a, there exist the following four conditions:

(i) Chktrip=True, Chkveto=True; i.e., TripCond=TRIP, VetoCond=OK;
(ii) Chktrip=True, Chkveto=False; i.e., TripCond=TRIP, VetoCond=VETO;
(iii) Chktrip=False, Chkveto=True; i.e., TripCond=OK, VetoCond=OK;
(iv) Chktrip=False, Chkveto=False; i.e., TripCond=OK, VetoCond=VETO.

For the multiple conditions at the partition function statement 53', there are the following four conditions:

(1) Guard=True, Setck=True
(2) Guard=True, Setck=False
(3) Guard=False, Setck=True
(4) Guard=Flase, Setck=False

Using the multiple condition coverage and equivalence classes of path coverage criteria, as are illustrated in the example of module "Update_Indi", we obtain the following selected testing cases:

**Path (1): A-B-D-F:** the derived test case is:

\[ PreState.Guardline=OK; PreState.Indicators = \{ \}; \]
\[ NewSignals = \{ \}. \]

**Path (3): A-B-C-B-D-F:** the derived test case is:

\[ PreState.Guardline=OK, PreState.Indicators = \{ N \} N=0, 1, 2... \]
\[ NewSignals = \{ (any one of: <ii>, <iii>, <iv>) \}. \]

To ensure the coverage of the multiple conditions of statement 5a, each of the three choices in the \textit{NewSignals} has to be tested, not just any one of them. So the test cases from this path are obtained with following settings:

Test Case #1: \[ PreState.Guardline=OK, PreState.Indicators = \{ N \} N=0, 1, 2... \]
\[ NewSignals = \{ (TRIP, OK) \}; \]

Test Case #2: \[ PreState.Guardline=OK, PreState.Indicators = \{ N \} N=0, 1, 2... \]
\[ NewSignals = \{ (TRIP, VETO) \}; \]
Test Case #3: \texttt{PreState.Guardline=OK, PreState.Indicators = \{ N \} \ N=0, 1, 2...}

\texttt{NewSignals = \{ (OK, OK) \} .}

These three test cases also cover two conditions of multiple condition 53'.

The conditions are:

condition (3) Guard=False, Setck=True, and
condition (4) Guard=Flase, Setck=False.

Path (4): A-B-C-B-D-E: the derived test cases are:

Test Case #1: \texttt{PreState.Guardline=OK, PreState.Indicators = \{ N \} \ N=0, 1, 2...}

\texttt{NewSignals = \{ (TRIP, OK) \} .}

Test Case #2: \texttt{PreState.Guardline=OK, PreState.Indicators = \{ N \} \ N=0, 1, 2...}

\texttt{NewSignals = \{ (TRIP, OK), (any one of: \langle i \rangle, \langle ii \rangle, \langle iii \rangle, \langle iv \rangle ) \} .}

Test case #1 in this path complements the multiple condition coverage of statement 5a by testing the condition (TRIP, OK). This test case also tests the condition (Guard=True, Setck=True).

Test case #2 in this path complements the multiple condition coverage of statement 53' by testing the condition (Guard=True, Setck=False).

Table 3.4 summarizes the results of the test for module #5 using these six test cases. The testing results are compared with expected results, and comments are provided.
Table 3.4 Summary of Selected Set of Test Cases and Testing Results for Module #5 "Update_GL"

<table>
<thead>
<tr>
<th>Test Case &amp; ID</th>
<th>Test Result</th>
<th>Expected Result</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test 1:</td>
<td>System interrupt stops the running of program with rejection status of the OO1 system.</td>
<td>No Message At all!</td>
<td>Later examination shows that the specification does not provide statements for Size1=Size2=0 operation.</td>
</tr>
<tr>
<td>Test 2:</td>
<td>Guardline=OK; Signals = {(TRIP, VETO)}; Indicators = {OFF}.</td>
<td>Guardline=OK; Signals = {(TRIP, VETO)}; Indicators = {OFF}.</td>
<td>This is one test case from path #3. Correct result from test.</td>
</tr>
<tr>
<td>Test 3:</td>
<td>Guardline=OK; Signals = {(OK, OK)}; Indicators = {ON, OFF};</td>
<td>Guardline=OK; Signals = {(OK, OK)}; Indicators = {ON, OFF};</td>
<td>This is one test case from path #3. Correct result from test.</td>
</tr>
<tr>
<td>Test 4:</td>
<td>Guardline=OK; Signals = {(OK, VETO)}; Indicators = {ON, OFF};</td>
<td>Guardline=OK; Signals = {(OK, VETO)}; Indicators = {ON, OFF};</td>
<td>This is one test case from path #3. Correct result from test.</td>
</tr>
<tr>
<td>Test 5:</td>
<td>Guardline=TRIP; Signals = {(TRIP, OK)}; Indicators = {OFF}.</td>
<td>Guardline=TRIP; Signals = {(TRIP, OK)}; Indicators = {OFF}.</td>
<td>This is one test case from path #4, Test case #1 set. Correct result from test. Guardline Changes</td>
</tr>
<tr>
<td>Test 6:</td>
<td>Guardline=TRIP; Signals = {(TRIP, OK), (OK, VETO)}; Indicators = {ON, OFF}.</td>
<td>Guardline=TRIP; Signals = {(TRIP, OK), (OK, VETO)}; Indicators = {ON, OFF}.</td>
<td>This is one test case from path #4, Test case #2 set. Correct result from test. Guardline Changes</td>
</tr>
</tbody>
</table>

3.4.3.3.5 Effectiveness of the Test Case Selection Method

We wish to examine the effectiveness of our test case selection method for treating two potential problems: (1) the effectiveness for enforcing a correct construction of the simplified structural graphs obtained from the OO1 specifications. The graphs are important since they are used for identifying equivalence classes of paths in the specifications; (2) the effectiveness for identifying external semantic defects of the five types of statements using the
selected test cases. Those five types of defects, labeled as (a), (b), (c), (d), (e), are defined and discussed in Section 3.4.3.2;

3.4.3.3.5.1 Effectiveness in Building Structural Graphs from Specifications

The build-up of structural graphs from specifications as a way to assist in the identification of equivalence classes of paths requires skill. No doubt, the process will suffer from the human commission and omission errors. Could any of these errors be prevented in the process? Again, let us take Figure 3.16 as an example. One question which might be raised is: could the loop connection from node "K" be made to node "C" rather than node "D"?

By following the requirements of constructing a simplified structural graph, it is recognized that connection from node "K" to node "C" will violate two basic requirements: (1) such a connection is not consistent with that of a simplified structure because node "C" and node "D" could then be collapsed into a single larger node, node "C'"; (2) the connection of the first node "C" and the last node "K" in a loop requires that the last functional statement number in the node "K" be the same as that of the first functional statement in node "C". The second requirement will not be met if such a connection is made. Thus, in conclusion, using this checking Procedure in meeting the requirements of the simplified structural graph is very efficient in identifying and removing any human errors introduced while building-up the graphs.

3.4.3.3.5.2 Effectiveness of Detecting Defects of External Semantics

In the OO1 Tool Suite, the path tracing facility is very useful for the activity of testing specifications. This facility, at the end of the test, provides information concerning those primitive functions that the OO1 Executor has just traversed and executed in the test. If the test result is not the one expected, then
with the knowledge of which equivalence class of path this test is in, and with the
information from the OO1 path tracing facility, one can analyze what is wrong,
and what causes the unexpected test result. This would lead to the discovery of
possible external semantic defects in the specifications.

In Section 3.4.3.2, for the RPS example, possible defects in four of five
types have been identified. Let us examine how effective the test case selection
method is in finding following defects in these four types.

**Incorrect assignments** [Defect Type b]

Take one functional statement, Create=K:ON:Indication(kill) listed under
defect type (b), for example. It is the statement #2888 in the module #2. Suppose,
due to some reason, that this statement is edited as
Create=K:OFF:Indication(kill). It will pass the OO1 Analyzer since its syntax and
internal semantics are consistent with those of the OO1 AXES language. However,
its meaning will have deviated from the intended one. The test cases are still
selected using the joint criteria of multiple condition and path equivalence class
coverage. In testing those cases, it was found that Tests 6, 8, 9, and 10 provide
unexpected results: The indication(s) which has the value "OFF" has not changed
to "ON" as expected. Using the OO1 path tracing facility, it was immediately
found that there was an editing error in this functional statement.

Due to the tight match of data types in the OO1 specification language, the
chance of making such an editing error is very small. From the TMap specification
in Figure 3.5, the indication can only take the value of either "ON" or "OFF". If
the editing error results in an indication taking a different value, e.g., "UP", this
error would be caught by the OO1 Analyzer earlier in developing the specification.

This conclusion also applies to two other functional statements listed with
this type of defects.
Wrong order of child functions of "O" and "CO" control structures [Defect Type c]

Consider one partition function, \( CO:\text{And}(:\text{Boolean}(Chksize1, Chksize2)) \) as listed for this type of defect. This is statement \#25' in module \#2. Suppose, due to the sloppiness of the specifier, the child functions, 'No_Update_On_Indicators' and 'EqualSizeUpdate' change their left-right order. Actually in testing of the specification using these test cases, it is found that all Size1=Size2 test cases have the warning message and do not update the indicators as the situation requires. The opposite is true for the test cases of Size1>Size2 and Size1<Size2 which produce no warning message and output whatever when they are entered the module.

Wrong Boolean variables are used for partition functions [Defect Type d]

From understanding of the characteristics of the OO1 specification we recognize that this type of defect rarely occurs in the specifications. The OO1 specification language has very strict rules which control data passing between parent and child functions, and among child functions. For any variable used in a partition function, it must be either passed down from the parent function or produced by its remaining sibling functions. In a partition function, any variable that comes from a far away place in the same FMap or from nowhere, and that goes nowhere after the partition function, will be identified automatically by the OO1 Analyzer. Thus, this kind of defect has a very small chance to avoid automatic verification by the Analyzer. Avoiding such verification is not the object of the testing of the specifications.

Input variables of Partition function are assigned wrong value [Defect Type e]

Experience shows that type (e) defects are very important and have a large chance of occurring in the OO1 specification context. For Boolean variables of
partition functions, their values are usually determined by arithmetic relations or Boolean expressions. The list of functional statements of Section 3.4.3.2 for type (e) defects clearly illustrates this point. Here, from module #2, we can consider the following two functional statements:

\[ Chk1 = Is:OFF:Indication(E1); \quad Chk2 = Is:OK:TripCond(InputTC); \]

The first example is statement #2880, the second is statement #2882, both within that module.

Now let us consider statement #2880 alone initially. Suppose that it were edited as \( Chk1 = Is:ON:Indication(E1) \). The Value 'ON' is the only other value that an indication could take. Here, the specifier committed a commission error of logic negation. However, he thinks that he is checking upon whether 'Indication' has the value "OFF". The meaning of this specification has thus deviated from the original one. The test cases are still derived using the same criteria and still include the same sized set of 10 tests as given before. Tests 1 to 5 show the expected results (this is just coincidental!). However, the result of test 6 is unexpected. What we expected in test 6 is the result:

\[
\text{Indicators} = \{ \text{ON} \}; \\
\text{Guardline} = \text{OK} \\
\text{Signals} = \{ \text{(TRIP, VETO)} \};
\]

but we obtained the result:

\[
\text{Indicators} = \{ \text{OFF} \}; \\
\text{Guardline} = \text{OK} \\
\text{Signals} = \{ \text{(TRIP, VETO)} \}.
\]
The Indicators have, in fact, not changed when the value 'OFF' should have changed to "ON"! Further examination through a tracing of the functional statements reveals an editing error committed in this functional statement.

Consider the second statement, $Chk2=Is:OK:TripCond(InputTC)$. Since the only other value that TripCond could take is TRIP, this statement might be edited as $Chk2=Is:TRIP:TripCond(InputTC)$ and it will pass the analysis of the Analyzer. Otherwise, the test cases are derived using the same criteria as in the previous example, and they have the same set size of 10. Tests 1 to test 4 give the expected results (This is merely coincidental!). At test 5, the result is unexpected. The expected result is:

\[
\begin{align*}
\text{Indicators} &= \{ \text{OFF} \}; \\
\text{Guardline} &= \text{TRIP} \\
\text{Signals} &= \{ (\text{OK},\text{VETO}) \};
\end{align*}
\]

but the result obtained is:

\[
\begin{align*}
\text{Indicators} &= \{ \text{ON} \}; \\
\text{Guardline} &= \text{TRIP} \\
\text{Signals} &= \{ (\text{OK},\text{VETO}) \}.
\end{align*}
\]

The Indicators have changed to the value 'ON', when it should not change. Further examination, through tracing the functional statements, reveals an editing error in this functional statement.

As another example of this type of defect, consider the functional statement $Localtrip=\text{And}:\text{Boolean}(Chktrip, Chkveto)$ occurring in statement #5a in module #5. A commission error by the specifier could cause an editing defect that changes the Boolean expression to: $Localtrip=\text{Or}:\text{Boolean}(Chktrip, Chkveto)$. When only this change is made to the specification of module #5, and the test cases are kept
unchanged (should we assume that the specifier still thinks he still is dealing with the statement \texttt{Localtrip=And:Boolean(Chktrip, Chkveto)}). When the six test cases are exercised again, tests 1, 4, 5, and 6 pass the testing and generate the expected results, while tests 2 and 3 generated unexpected results. The results for tests 2 and 3 are shown in Table 3.5:

Table 3.5 Results of the Two Tests on the New Specification

<table>
<thead>
<tr>
<th>Test Case ID &amp; Inputs</th>
<th>Test Result</th>
<th>Expected Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test 2&lt;br&gt;Guardline=OK; Signals=$((TRIP, VETO))&lt;br&gt;Indicators={ ON, OFF}</td>
<td>Guardline=TRIP; Signals=$((TRIP, VETO))&lt;br&gt;Indicators={ ON, OFF}</td>
<td>Guardline=OK; Signals=$((TRIP, VETO))&lt;br&gt;Indicators={ ON, OFF}</td>
</tr>
<tr>
<td>Test 3&lt;br&gt;Guardline=OK; Signals={ (OK, OK) }&lt;br&gt;Indicators={ ON, OFF}</td>
<td>Guardline=TRIP; Signals={ (OK, OK)&lt;br&gt;Indicators={ ON, OFF}</td>
<td>Guardline=OK; Signals={ (OK, OK)&lt;br&gt;Indicators={ ON, OFF}</td>
</tr>
</tbody>
</table>

Further analysis, using the functional statement tracing technique, shows that statement 5a has been incorrectly specified.

In a summary, this section examines the effectiveness of the test case selection method in detecting all five types of external semantic defects of the OO1 specifications. Five defects are embedded in test examples and an attempt to detect them is made using the selected test cases. The conclusion from the study is that the selected test cases are very efficient in detecting these defects.

3.4.3.4 Integrated Testing on The Entire Specification

From the characteristics of the OO1 FMap, the specifications are developed on a top-down basis. An operation could have many modules at sub-levels and
these modules could be specified in their respective separate FMaps. As is discussed in the previous Sections, the OO1 system provides an environment for testing those modules independently before they are integrated into the higher levels of the specifications. With the RPS example, the relationships between its main operational structure and its constituent modules are clearly shown in the Road Map developed by the OO1 system. This road map is shown in Figure 3.19.

![Road Map of the Reactor Protection System Program](image)

In this RoadMap, only a single level of module integration exists. Since 'SetIndiUpdate' is a user-defined structure in module 'Update_Indi', both belong to a larger module. Here, three modules, 'Update_Indi', 'CheckGLOK', and 'Update_GL' are considered to be in the same level. They are immediately integrated into the main operation 'Update_State'. Thus, there only exists one stage of integration for conducting the test of the entire specification. There are other cases where the RoadMap indicates more than one level of module integration, and there more steps are required to conduct integration tests. In that situation, the entire specification would be subjected to testing, starting from the very bottom level individual modules (as was done in the previous Section). Through a bottom-
up integration procedure, the specification of the main operation can be tested. The
procedure works according to the following steps:

Step I. Build a structural graph of the main operation
Step II. Identify all of the multiple conditions
Step III. Identify all of the equivalence classes of paths
Step IV. For each path, identify the modules used within it and identify a union
set of test cases from these modules
Step V. Check that all of the path test cases also cover the multiple conditions
completely.

Applying this procedure to the RPS example, we obtained following results:

[At Step I]:

A structural graph is obtained from the specification of Fig. 3.6 and is
shown in Fig. 3.20.

```
A
  ^
  \ 
   B
  / \ 
 C   D
    / 
   /   
  /     
A = Modules 2, 3
B = Partition function 3'
C = Module 4
D = Module 5
```

Figure 3.20 Structural Graph of the RPS Main Operation
[At Step II]:

Multiple conditions are identified for the variable "ChkGL" at the partition function of statement #3'. Two conditions exist:

ChkGL=True, and ChkGL=False.

[At Step III]:

Two equivalence classes of path are identified. They are:

A-B-C
A-B-D.

[At Step IV]:

For Path A-B-C, test cases are derived from the union of modules #2 and #3. Their testing results for the main program operation are listed in Table 3.6.

For Path A-B-D, test cases are derived from the union of modules #2, #3, and #5. Their testing results for the main program operation are listed in Table 3.7.

[At Step V]:

It is confirmed that all multiple conditions are tested in the testing cases of the two paths.

From using the integrated testing procedure, the RPS specifications are efficiently tested. It must be recognized that the modular integration of the RPS example only represents one of many types of module integration. In path A-B-D, the two modules are integrated. The variables changed in module #2 (the system state Indicators) are not modified by the operation of module #5. Since module #5
only changes the values of other parts of system state (the system state Guardline), the resulting integration is made much easier by means of union of test cases from each module. In this instance, the variables of one module are not affected by those of the others. However, if in a path, one module's output is reprocessed by the next module, then we have a difficult type of integration which has been recognized in some researches (Hale81, Parr89). That is the Data Passing Chain (DP Chain) type of integration. This type is not further discussed in this study.

Table 3.6  Summary of the Set of Test Cases and Testing Results for the Path A-B-C of the Main Program Operation

<table>
<thead>
<tr>
<th>Test Case &amp; ID</th>
<th>Test Result</th>
<th>Expected Result</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test 1:</td>
<td>Indicators = {ON}; Guardline=TRIP; Signals = {(TRIP, OK)}.</td>
<td>Indicators = {ON}; Guardline=TRIP; Signals = {(TRIP, OK)}.</td>
<td>Correct result from test.</td>
</tr>
<tr>
<td>Test 2:</td>
<td>Indicators = {OFF}; Guardline=TRIP; Signals = {(OK, VETO)}.</td>
<td>Indicators = {OFF}; Guardline=TRIP; Signals = {(OK, VETO)}.</td>
<td>Correct result from test.</td>
</tr>
<tr>
<td>Test 3:</td>
<td>Indicators = {ON, OFF}; Guardline=TRIP; Signals= {(OK,OK), (OK,OK)}.</td>
<td>Indicators = {ON, OFF}; Guardline=TRIP; Signals= {(TRIP,OK), (OK,OK)}.</td>
<td>Correct result from test.</td>
</tr>
<tr>
<td>Test 4:</td>
<td>Indicators = {ON, OFF}; Guardline=TRIP; Signals = {(OK, VETO), (TRIP, OK)}.</td>
<td>Indicators = {ON, ON}; Guardline=TRIP; Signals = {(OK, VETO), (TRIP, OK)}.</td>
<td>Correct result from test.</td>
</tr>
<tr>
<td>Test 5:</td>
<td>Indicators = {OFF, OFF}; Guardline=TRIP; Signals = {(OK,OK), (TRIP, VETO)}.</td>
<td>Indicators = {OFF, ON}; Guardline=TRIP; Signals = {(OK,OK), (TRIP, VETO)}.</td>
<td>Correct result from test.</td>
</tr>
</tbody>
</table>
Table 3.7  Summary of the Set of Test Cases and Testing Results for Path A-B-D of the RPS Main Program Operation

<table>
<thead>
<tr>
<th>Test Case &amp; ID.</th>
<th>Test Result</th>
<th>Expected Result</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test 1:</td>
<td>System Interrupt stops the running of program with rejection status of the OO1 system.</td>
<td>No Message At all!</td>
<td>The same with Test1 of module #5 in Table 3.4</td>
</tr>
<tr>
<td>Guardline=OK;</td>
<td>Guardline=OK;</td>
<td>Guardline=OK;</td>
<td>Correct result from test.</td>
</tr>
<tr>
<td>Signals = {} (TRIP, VETO );</td>
<td>Signals = {} (TRIP, VETO );</td>
<td>Signals = {} (TRIP, VETO );</td>
<td></td>
</tr>
<tr>
<td>Indicators = {} (OFF).</td>
<td>Indicators = {} (OFF).</td>
<td>Indicators = {} (OFF).</td>
<td></td>
</tr>
<tr>
<td>Test 2:</td>
<td>Guardline=OK;</td>
<td>Guardline=OK;</td>
<td>Correct result from test.</td>
</tr>
<tr>
<td>Guardline=OK;</td>
<td>Signals = {} (OK, OK);</td>
<td>Signals = {} (OK, OK);</td>
<td></td>
</tr>
<tr>
<td>Signals = {} (OK, OK);</td>
<td>Indicators = {} (ON, OFF);</td>
<td>Indicators = {} (ON, OFF);</td>
<td></td>
</tr>
<tr>
<td>Test 3:</td>
<td>Guardline=OK;</td>
<td>Guardline=OK;</td>
<td>Correct result from test.</td>
</tr>
<tr>
<td>Guardline=OK;</td>
<td>Signals = {} (OK, VETO);</td>
<td>Signals = {} (OK, VETO);</td>
<td></td>
</tr>
<tr>
<td>Signals = {} (OK, VETO);</td>
<td>Indicators = {} (ON, OFF);</td>
<td>Indicators = {} (ON, OFF);</td>
<td></td>
</tr>
<tr>
<td>Test 4:</td>
<td>Guardline=OK;</td>
<td>Guardline=OK;</td>
<td>Correct result from test.</td>
</tr>
<tr>
<td>Guardline=OK;</td>
<td>Signals = {} (OK, VETO);</td>
<td>Signals = {} (OK, VETO);</td>
<td></td>
</tr>
<tr>
<td>Signals = {} (OK, VETO);</td>
<td>Indicators = {} (ON, OFF);</td>
<td>Indicators = {} (ON, OFF);</td>
<td></td>
</tr>
<tr>
<td>Test 5:</td>
<td>Guardline=TRIP;</td>
<td>Guardline=TRIP;</td>
<td>Correct result from test.</td>
</tr>
<tr>
<td>Guardline=TRIP;</td>
<td>Signals = {} (TRIP, OK);</td>
<td>Signals = {} (TRIP, OK);</td>
<td></td>
</tr>
<tr>
<td>Signals = {} (TRIP, OK);</td>
<td>Indicators = {} (OFF).</td>
<td>Indicators = {} (OFF).</td>
<td></td>
</tr>
<tr>
<td>Test 6:</td>
<td>Guardline=OK;</td>
<td>Guardline=OK;</td>
<td>Correct result from test.</td>
</tr>
<tr>
<td>Guardline=OK;</td>
<td>Signals = {} (TRIP, OK);</td>
<td>Signals = {} (TRIP, OK);</td>
<td></td>
</tr>
<tr>
<td>Signals = {} (TRIP, OK);</td>
<td>Indicators = {} (ON, OFF);</td>
<td>Indicators = {} (ON, OFF);</td>
<td></td>
</tr>
<tr>
<td>Test 7:</td>
<td>Warning Message! No Update Action!</td>
<td>Warning Message! No Update Action!</td>
<td>Correct result for: Size1&lt; Size2</td>
</tr>
<tr>
<td>Indicators = {} (ON);</td>
<td>Guardline=OK;</td>
<td>Guardline=OK;</td>
<td></td>
</tr>
<tr>
<td>Signals = {} (OK, OK), (OK, OK).</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Test 8:</td>
<td>Indicators = {} (ON, OFF, OFF);</td>
<td>Indicators = {} (ON, OFF, ON);</td>
<td>Correct result from test.</td>
</tr>
<tr>
<td>Indicators = {} (ON, OFF, OFF);</td>
<td>Guardline=TRIP;</td>
<td>Guardline=TRIP;</td>
<td></td>
</tr>
<tr>
<td>Signals = {} (TRIP, VETO);</td>
<td>Signals = {} (TRIP, VETO);</td>
<td>Signals = {} (TRIP, VETO);</td>
<td></td>
</tr>
<tr>
<td>(OK, VETO), (TRIP, OK).</td>
<td>(OK, VETO), (TRIP, OK).</td>
<td>(OK, VETO), (TRIP, OK).</td>
<td></td>
</tr>
</tbody>
</table>

3.4.3.5 Quality of the Tested Specification: What do We Know?

In this section, a conceptual process of updating one's knowledge of the quality of the tested specification is discussed. The purpose of the discussion is to show some possible ways to address the following concern: after applying the four
stages of the verification process, including testing of the specification using the selected T tests, what is the probability that one would detect a previously undetected defect in the next \((T+1)\text{th}\) test of the specification? Here, the state of the knowledge of the quality of the tested specification can be obtained by using Bayes' theorem, based upon our knowledge of the specification prior to the testing and the information acquired in testing of the specification during the T tests.

The information of the specification and its testing results are presented in Section 3.4.2 and of Sections 3.4.3.1 to 3.4.3.4 respectively.

First, after all of the system specifications pass the test of the \textit{Analyzer}, the program's main operation and all of its module operations at the sub- and sub-sub-levels are specified and installed into the OO1 system, ready for the testing by the \textit{Executor}. Here, according to the analyses of Section 3.4.3.2, five types of functional statements ( types (a) to (e) ) which potentially have external semantic defects can be identified.

From identification of equivalence classes of paths, as discussed in Section 3.4.3.3.4, we see that those five types of functional statements appear in each class.

\textbf{3.4.3.5.1 Before the Selected T Tests are Performed}

Before the selected T tests, we have our basic knowledge of the developed specification. In it, each type of the functional statements has its special formats of OO1 primitive functions. When we talk about the types of the functional statements, we talk about each statement's functional structural format. The argument of the functional statement does not contribute to the type definition. For example, in a type (b) statement \textit{(Incorrect Assignment)} the formats of the primitive functions include following ones:
Output=K:ChildNode:ParentNode(input), and
Output=KF:Boolean(Input); Output=KF:Boolean(Input); etc..

In statement type (e) (Partition function's input variables are assigned wrong values) the forms of the primitive functions include following ones:

Output=GE:Nat(Input1, Input2);
Output=IS:Value:ParentNode(Input); and
Output=And:Boolean(Input1, Input2); etc..

This study assumes that an adequately trained specifier will probably commit commission errors in specifying these primitive functions. In this study, it is also assumed that, theoretically, a statistical database of probabilities of those commission errors would be required for any further estimation. A field study performed using all of the OO1 projects could produce such a database. However, no such database is available right now for use in this study. We identified this as one of limitations to this work.

Based upon the hypothetical assumption of an available database, the distribution of a parameter, \( PFi \), can be obtained from the database. Here \( PFi \) is defined as the human error rate of committing commission errors for the functional statements of type (i), \( i \) is to range from (a) to (e). Usually, the probability distribution of this rate, \( PFi \), follows a log-normal distribution.

Based upon the derived equivalence classes of paths (with total number of \( k \) classes), we can also identify how many of the five types of functional statements exist in each class of path. Let us use \( SVj \) to denote the number of the five types of functional statements in class \( j \) (\( j = 1 \) to \( k \)).
Now, before the selected T test cases are performed, we are interested in the following question: what is the probability that one would detect one or more defects in any single one of the T tests? Let us denote this probability as \( P \). This probability can be expressed using the following function:

\[
P = f(PF_a, PF_b, PF_c, PF_d, PF_e, SV_1, ..., SV_k)
\] (3-1)

where \( k \leq T \);

\( PF_i \) = commission error rate in functional statement type \( i \); \( i = a, b, c, d, e \)

\( SV_j \) = vector of numbers of each of the five types of functional statements, respectively in class \( j \);
\( j = 1 \) to \( k \).

Since each \( PF_i \) has an assumed distribution, then for the value \( P \), we would obtain a form of distribution \( \pi(P) \). This distribution, \( \pi(P) \), is our basic knowledge of the quality of the developed specification before the selected T tests are performed, and it is going to be updated after the T tests are performed.

3.4.3.5.2 In the Process of Performing the Selected T Tests

Using the test case selection method discussed in Section 3.4.3.3, one can obtain a set of test cases for the specification. In running the tests, if a defect is found in a functional statement, the defect is then corrected. The functional statement is tested again to ensure that it is correct now. After each derived test case goes through such a testing cycle, one should have following statistical data from the tests, \( d_T \), the total number of defects detected and corrected, and its composition, \( d_{Ti} \), where \( d_T \) is the sum of \( d_{Ti} \), \( i \) is to range from (a) to (e).
3.4.3.5.3 After the T Tests

Here, we are still interested in the following question: what is the probability that one would detect one or more defects in the (T+1)th test? Let us denote this probability also as \( P_{(T+1)} \).

But now, the distribution of the new \( P_{(T+1)} \) should be updated by Bayes' theorem using its prior distribution of Equation (3-1) and a likelihood function of finding \( d \) defects, \( L(d_T|P_T) \). The question of how to formulate the generally applicable likelihood function, \( L(d_T|P_T) \), remains as another limitation identified in this work, since the sparse \( d \) data in this study do not provide any basis for this formulation. However, it is generally expected that the rate for finding more defects in the remaining tests would become steadily reduced, as the testing proceeds. One then can look into modeling the process of finding defects by using Non-homogeneous Poisson distribution. When the likelihood function is obtained, then our current knowledge of the quality of the specification can be represented by the following posterior distribution obtained by using Bayes' theorem:

\[
\pi'(P_{(T+1)}|d_T) = \kappa \ast \pi(P_T) \ast L(d_T|P_T) \quad (3-2)
\]

where \( \kappa \) = factorial value obtained in a integration process.

This posterior distribution, \( \pi'(P_{(T+1)}|d_T) \), represents our updated knowledge of the quality of the specification.

In the above discussion of the updating process, two very important assumptions are used. The first is that, in the prior distribution \( \pi(P) \), statistical data base should be available for obtaining commission error rates, PF\(_i\), values of five types of functional statements. The second assumption is that the likelihood function for obtaining \( d_T \) defects, \( L(d_T|P_T) \), is obtainable in an explicit
mathematical form. We suggested here that independent works be conducted to obtain these data for all the OO1 Tool Suite supported formal specification cases.

3.4.4 The Automatic Target Code Generation

The OO1 Tool Suite provides a capability for automatic target code generation from the developed specifications. In addition, the OO1 tool can generate English documents derived from the specifications with descriptions of the operations and all of the functions in the operation. These documents, called collected files, mainly describe the activities of all of the functions and the relationships among them. They also serve better to annotate the generated target code. For the RPS program, its resulting C language code and corresponding English documents are generated and listed in Appendix A.

The current version of the OO1 tool suite only provides a prefixed configuration of the target source code structure. However, different users of the OO1 system might prefer to have different configurations of the target source code structures. This problem has been identified within this study and is being remedied by the OO1 developer: Hamilton Technology, Inc. (HTI). The potential solution of this problem is for HTI to deliver User-oriented configurations of code generation mechanisms. This is understood to be a solvable problem using the additional technical capability of the RAT facility of the OO1 Tool Suite.

At this stage, there is a concern about the reliability of the generated source code with respect to the specifications. Without any statistical data being available from all OO1 supported projects, it can only be assumed that the probability that the code will be generated absolutely correctly with respect to the specifications is of equal to unity. Whether this is true remains unknown.
Another way of verifying that the generated code is correct is to compile the code on the target machine, and to test the compiled code using those specification-generated test cases. Of course, the formats of the data types of the test cases may have to be transformed into the formats of data types applicable in the target machine environment.

3.5 SUMMARY ON TREATING DEFECTS BY THE IFA

The Integrated Formal Approach has been devised to eliminate various types of defects from the specification of the system software design. As we discuss in Section 3.2.1, for program design and implementation, there are five major types of defects. They are (A) defects in requirements and specifications; (B) defects in control logic, decision logic and sequencing of actions; (C) data processing defects; (D) data flow and interface defects; and (E) coding defects. As the example of applying the IFA to the RPS case shows, the IFA could help efficiently to eliminate all defects from types (B), (C), (D) and (E). We also recognize that for some of the defects of type (A), like the missing data input requirements of the system, this IFA could not eliminate them completely. This is because the IFA relies heavily upon the current design information to build the firstly agreed TMap of the entire system data structure. However, that first TMap provides a common base for communications between programmers and designers. By following an appropriate procedure in discussing the completeness of the TMap design, some of such defects, like missing input data might be caught. However, that is out of the control of the IFA. Also, it is recognized that the IFA cannot affect defects arising the OO1 Tool Suite Generator, as well as from the compiler and system program of the machines.
3.6 SUMMARY OF THE CHAPTER

In this Chapter, an Integrated Formal Approach for improving software design and reliability is proposed. An example of a Reactor Protection System software design is used to explain the details of this approach.

The idea of developing this Approach comes from the understanding of the varieties of software defects that are usually introduced throughout the software development process. This study shows that the number of these defects could be systematically reduced with the use of a formal methods-based technique (in this instance, called Development Before The Fact), and its supporting CASE tool (in this instance, called the OO1 Tool Suite) in the software development process. The formal technique and its supporting CASE tool have already been selected and discussed in details in the proceeding Chapter.

The Integrated Formal Approach, which is shown in Figure 3.4, includes the following four elements: (1) Rules for transforming information from design documents; (2) Procedure of constructing specifications; (3) Strategy of selecting test cases for testing specifications; and (4) Automatic target code generation. Section 3.4 presents detailed discussions on the designs of these elements.

In setting up the Rules for transforming information from design documents into the basic elements of the OO1 specification, different kinds of design documents have been considered. These documents include plain English text, pseudo program structured text, data flow diagrams, logic flow diagrams, etc.. The rules are seen to be efficient in collecting the information for use in the specifications.

Section 3.4.2 and 3.4.3 discuss the designs of the two major elements of the Integrated Formal Approach: the development procedure and testing the strategy for the specification. The procedure and the strategy are employing the techniques
of four stages of verification for developing correct, reliable specifications. The RPS example is used to show that these two elements can efficiently correct the conventional types of defects from the specifications. For the OO1 specifications, all of the corrected defects are easily identified and classified. Finally, a conceptual process of updating one's knowledge of the quality of the tested specification is discussed, with two important assumptions to be justified in further study.
CHAPTER 4

A System Design Case Study Using The Integrated Formal Approach

4.1 INTRODUCTION

This Chapter presents the results of a case study in which the designed Integrated Formal Approach (IFA) of Chapter 3 is applied to the development of formal specification of a larger program used in a nuclear power plant's reactor coolant system. The program is the generic Signal Validation Algorithm (SVA) used for calculating the values of parameters of the reactor coolant system and assuring their validity. The program specification were provided for study by a commercial reactor manufacturer firm. The purpose of this case study is to assess the effectiveness of the IFA for improving the quality of the original design of the program, and to examine the efficiency of the approach in developing reliable safety-critical software in a CASE tool supported environment. The assessments are based upon the following categories of information from the application:

(1) Categories of possible improvements, and the places in the original documents where these improvements should be made;
(2) Statistical data of the program specifications as embedded in TMap and FMaps;
(3) Results from the tests of the developed formal specification;
(4) Comparison of the SVA results with data from ad-hoc approach.
All of the information are presented in detail throughout this Chapter. Assessment on the obtained data are provided. Conclusions are drawn from this case study.

4.2 SIGNAL VALIDATION ALGORITHM (SVA)

4.2.1 Signal Validation Features

In the nuclear power industry, many efforts have been directed toward improving plant performance and availability. A major factor that is important for the success of these efforts is improved access to reliable information for the entire plant, especially concerning safety-critical systems. Many nuclear utilities are installing or upgrading their plant computer systems to provide such a capability to handle plant data. It is recognized that corresponding software developments are critical in those safety-critical systems for providing operators and other personnel with data interpretations that are accurate, extensive, reliable and useful.

The main benefit of the application of these computer systems arises from provisions for ready access to a variety of sensor signals. This feature permits operators to use an increased number of redundant, independent measurements in the decision-making processes performed during normal and emergency plant operations. However, simply displaying those redundant measurements is not enough, and will adversely complicate the operator's mental processing tasks of data comparison and evaluation. One way for reducing the operators' work load and stress in comparing, analyzing, and evaluating sensor signals is to apply automatic signal validation algorithms rather than the human's mental processing.

The SVA of this case study is a generic algorithm that performs signal validation for the plant processes that contain multiple sensors measuring the same
or closely related process parameters. It is called the generic SVA because the principle of the algorithm is generally applicable to all kinds of parameters in the system. Only some slight changes are made for each particular parameter. This is because for different parameters, different number of groups of sensors are used to collect the measured data.

4.2.2 Original Design Documents of the SVA

There are two volumes of original SVA design documents upon which this case study is based. They are mutually complementary. The first volume is a pseudo-program structure of English text that provides definitions of terms, functions, description of all the data calculations, and logical connections between functions and calculations. The text occupies fourteen pages. The second volume provides descriptions of the algorithm in terms of object diagrams and data-flow diagrams. It also provides definitions of the signal system state in terms of sensor and process Tables.

4.2.3 SVA Formal Specifications Using the IFA

This Section provides a summary of the identified possible improvements in the original SVA design, and a summary of the SVA's formal specification in the OO1 language. First, a process of deriving at a set of valid system states from the SVA's TMap is presented. This set provides the basis for specifying the SVA's property of "data-type invariance" in its FMaps. The purpose of finding this property is discussed in Chapter 3, Section 3.4.2. After assuring that property, the statistical data of the SVA's TMap and FMaps are summarized. In developing these TMap and FMaps using the transforming Rules and constructing Procedure
of the IFA, several categories of possible design improvements were identified in the original design documents. The results of the Testing of the specifications are then presented and discussed. Finally, the quality of the formal specification is assessed.

4.2.3.1 Deriving the Set of Valid States From The TMap

The TMap of the SVA program is first constructed from the basic elements transformed from the original design documents. It was first agreed upon and finalized through several cycles of discussions between the specifier and system designers.

The way to specify a program's "data type invariance" is discussed in Step II of the Procedure of Constructing Specifications in Section 3.4.2. The SVA's TMap includes two elements: 'ProcessRecord' and 'SensorRecord'. These two elements are specified independently in separate data structures. Therefore, at first, a set of valid states is derived for each element independently. Later, the two sets of valid states from two elements are combined to obtain the complete set of the entire SVA's valid states.

For the element 'ProcessRecord', seven valid states are derived from the total 36 conceivable states that are the results of the simple combination of all values of all objects in the element 'ProcessRecord'. Twenty-nine states are eliminated because their physical meanings in terms of the plant's operation make them unacceptable. Table 4.1 provides the definitions of the OBJECT and the range of their VALUES. Table 4.2 presents the set of valid states of the element 'ProcessRecord' using the definitions of Table 4.1.

For example, in Table 4.1, The object #1 has the identifier AlarmMessage. It can take any one of the two values, Clear, ValidationFault (given in the
original design). For the first value Clear, we use symbol 1.A to represent it. The other objects and their values can be read in the same way.

For another example, in Table 4.2, the derived #2 valid state of the element 'ProcessRecord' is composed of values of 1.A, 2.C, 3.B and 4.A in symbols. Its physical meaning is that, in the #2 valid state of the element 'ProcessRecord', the object AlarmMessage takes the value Clear, the object PermissiveMessage takes the value PamiFailOpSelect, the object DisplayMessage takes the value PamiFault, and the object ProcessValueOut takes the value Averaged. Other valid states in Table 4.2 are read in the same way.

Table 4.1 Definitions of the Object Values of the Element 'ProcessRecord'

<table>
<thead>
<tr>
<th>Object Identifying Number in the Entire SVA</th>
<th>Object's Identifier</th>
<th>Object's Possible Values</th>
<th>Symbols of the Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>AlarmMessage</td>
<td>Clear</td>
<td>1.A</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ValidationFault</td>
<td>1.B</td>
</tr>
<tr>
<td>2</td>
<td>PermissiveMessage</td>
<td>Clear</td>
<td>2.A</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ValidFailOpSelect</td>
<td>2.B</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PamiFailOpSelect</td>
<td>2.C</td>
</tr>
<tr>
<td>3</td>
<td>DisplayMessage</td>
<td>Pami</td>
<td>3.A</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PamiFault</td>
<td>3.B</td>
</tr>
<tr>
<td>4</td>
<td>ProcessValueOut</td>
<td>Averaged</td>
<td>4.A</td>
</tr>
<tr>
<td></td>
<td></td>
<td>FaultCaled</td>
<td>4.B</td>
</tr>
<tr>
<td></td>
<td></td>
<td>OpSelected</td>
<td>4.C</td>
</tr>
</tbody>
</table>

The TMap's another element, 'SensorRecord', has two objects. One is the group of 'PamiRange' sensors, another is the group of 'NormalRange' sensors. It is required from the original design that, in all of the valid system states before and after the algorithm's scan of the sensors, no single sensor has the quality 'suspect'. This means that any sensor could have only either a 'good' or 'bad' quality before
and after the validation. However, during the validation, some sensors could be of 'suspect' quality. The definitions of the two objects and their values are presented in Table 4.3. Table 4.3 can be read in the same way as Table 4.1.

Table 4.2 The Set of Valid States of Element 'ProcessRecord'

<table>
<thead>
<tr>
<th>State Number</th>
<th>State Contents (In terms of object values of Table 4.1)</th>
</tr>
</thead>
</table>

Table 4.3 Definitions of Object Values in the Element 'SensorRecord'

<table>
<thead>
<tr>
<th>Object's Number in the Entire SVA</th>
<th>Object's Identifier</th>
<th>Object's Possible Values</th>
<th>Symbols of the Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>PamiRangeSensors</td>
<td>Bad</td>
<td>5.A</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Suspect</td>
<td>5.B</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Good</td>
<td>5.C</td>
</tr>
<tr>
<td>6</td>
<td>NormalRangeSensors</td>
<td>Bad</td>
<td>6.A</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Suspect</td>
<td>6.B</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Good</td>
<td>6.C</td>
</tr>
</tbody>
</table>

Therefore, the valid state of the element 'SensorRecord' is logically given as:

\[
\text{valid state} = \{(\text{not 5.B}) \text{ and (not 6.B)}\}. \tag{4.1}
\]

Finally, the complete set of valid states of the SVA is obtained by combining the valid states of the two independent elements as expressed in Table
4.2 and the logical expression of the Equation 4.1. This complete set is presented in Table 4.4. As we can see, there are a total of seven valid states for the entire SVA program. Based upon the definitions of these seven valid states, the FMap for specifying the "data-type invariance" property has been constructed. That FMap is named "Pre_Condition".

4.2.3.2 The SVA's TMap

The SVA TMap includes two major elements: "ProcessRecord" and "SensorRecord". The entire TMap specification takes two pages. In summary, the 'TupleOf' data type is used nine times, the 'OneOf' data type eight times, the 'OSetOf' data type once, the 'Value' data type seventeen times, the 'Rat' data type nine times, and the 'Str' data type three times.

<table>
<thead>
<tr>
<th>State Number</th>
<th>State Content (In Terms of Previously Defined Symbols)</th>
</tr>
</thead>
</table>

4.2.3.3 The SVA's FMaps

For the FMaps of the specifications, the following data are summarized:

- There are thirty-one independent FMaps used to specify the main operation of the algorithm and all supporting modules;
• There are twenty-six supporting modules and four independent structures. In those twenty-six modules, eighteen of them are independent. These independent modules are analyzed and the test cases are to be derived from them.

• From the RoadMap of the entire SVA specification, it is seen that there are three levels of integration of modules into the main operation.

• Finally, there are a total of 416 primitive statements and a total 49 partition functions in the entire SVA specification.

4.2.3.4 Results From Applying The Rules and The Procedure

In applying the Rules to transform information from the original design documents and the Procedure to construct the specifications, there are six categories of possible improvements identified from the original design. The definitions of those categories are given in Table 4.5.

Table 4.5. Improvements in the Original Design in Developing the SVA Specifications

<table>
<thead>
<tr>
<th>Categories of Improvements</th>
<th>Details of the Improvements</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Identified Ambiguities in using WORDs and TERMs</td>
</tr>
<tr>
<td>2</td>
<td>Identified Inconsistencies in defining and using the same WORDs and TERMs at different places in the same document</td>
</tr>
<tr>
<td>3</td>
<td>Identified Ambiguities in defining OPERATIONs and FUNCTIONs</td>
</tr>
<tr>
<td>4</td>
<td>Identified Inconsistencies in defining the same OPERATIONs and FUNCTIONs at different places in the same document</td>
</tr>
<tr>
<td>5</td>
<td>Identified Incompleteness of the LOGIC DESIGN of operations and functions, based upon the data type structures of the input variables of the operations and functions</td>
</tr>
<tr>
<td>6</td>
<td>Identified Incompleteness of OUTPUT STATE DEFINITION of the operations and functions, based upon the data type structures of the output variables of the operations and functions</td>
</tr>
</tbody>
</table>
4.2.3.4.1 Category 1: Ambiguities in Words and Terms

This category of improvements was identified by applying the transforming Rules to the original design documents. They let the specifier truly understand the meaning of the words and terms used. There are a total of five places in the original documents where improvements in using words and terms are suggested. A typical example is summarized as follows:

"On page 30 of the SVA design document, at step 8, on the second line, the word sensor is used ambiguously. Should we do a deviation check on the 'Good' PAMI sensors or on all sensors regardless of their quality?"

4.2.3.4.2 Category 2: Inconsistencies in Words and Terms

This category of improvements was identified by applying the transforming rules to the original design documents. The difference between this category and category 1 is that certain terms and words as used are understood clearly. However, in different places in the same document, they have different means. There are a total of six places in the original documents where improvements in using words and terms consistently are suggested. A typical example of this category is summarized as follows:

"On page 19 of the SVA specification, in the first sentence of the last paragraph, the word all is used inconsistently in two phrases: '...averages all sensors...', and '...and deviation checks all sensors...'. The use in the first place is inconsistent with its use in the average function specified on page 25, where the phrase all good is used instead of just the single word all. The use of all in the second instance is inconsistent with the definition of the deviation check function specified on page 25, where the phrase all good is used."
4.2.3.4.3 Category 3: Ambiguities in Operations and Functions

Some possible improvements in this category are found by transforming information from the original documents, and some of them are found in constructing specification by using the Procedure. There are a total of three places in the original documents where improvements in defining unambiguous operation and functions are suggested. A typical example of a possible improvement identified by the transforming process is summarized as follows:

"On page 20 of the SVA specification, at line 11, the phrase '...deviation check is satisfactory...' is ambiguously written. What does satisfactory really mean?"

A possible improvement identified by the constructing process is summarized as follows:

"On page 27 of the SVA specification, at step 4 of the specification of the operation Pami_Check, there is an ambiguity in specifying the deviation check criteria on the YES condition: Are all PAMI sensors, or only the PAMI sensors having good quality, to be used?"

4.2.3.4.4 Category 4: Inconsistencies in Operations and Functions

This category of improvements was only identified in constructing the specifications. There are two places in the original documents where such possible improvements are identified. An example of these possible improvements is summarized as follows:

"On page 21 of the SVA specification, in paragraph 5 describing the term Fault_Select, the definition of how Fault Select is included in the output as part of the 'process presentation' is inconsistent with the definition at Step 6 on page 29, where the output condition includes
another check upon the 'Operator-Select-Permissive' and upon observing the operator's selection action."

4.2.3.4.5 Category 5: Incomplete Logic Design of Functions and Operations

This category of improvements was only identified by applying the Procedure to construct the formal specifications of the program. There are a total of two places in the original documents where such possible improvements are identified. One of them is summarized as follows:

"On page 29 of the SVA specification, at step 5, in the logic of ' the previous scan was fault select ', one input possibility is not checked: the pre-scan fault selected sensor now has a value that is different from the value that it had in the last scan. Should the current value be the output to the calculated signal or should the value from the last scan be the output to the calculated signal?"

4.2.3.4.6 Category 6: Incomplete Output State Definitions

This category of improvements was only identified by applying the Procedure to construct the formal specifications of the program. There are a total of three places in the original documents where such possible improvements are identified. One such improvement suggestion is summarized as follows:

"On page 28 of the SVA specification, the design of the operation Failed Validation at step 5 has produced an unspecified state of the output of the entire algorithm --- the value of the display message, either PAMI or Pami_Fault is not specified for this scan. One possible consequence is that no operator selection is committed at step 7, then the process presentation will finally have the Fault_Select sensor value as output. However, at the
end of the program, the display message will never be updated. It will still contain the old value from the last scan."

4.2.3.5 Results From Testing the Specifications

In the entire specifications, there are 18 independent modules subjected to independent Testing. There are extra 9 modules subjected to integration testing, including the testing of the main operation. For the testing of the entire specification, three levels of integration of these 27 modules are required. The Testing method has been designed and explained in Section 3.4.2 using the RPS example. The functional Testing of the SVA case is conducted by using the same method in the following two steps:

1. First, the five types of primitive functional statements are identified from the 18 independent modules and 9 integration modules. The numbers of these statements found in this example are summarized in Table 4.6

2. Second, using the criteria of *multiple condition coverage* and *equivalence classes of paths coverage*, testing cases are selected from the 18 independent modules and the 9 integration modules, and then are applied. There are a total of 134 testing cases. This number is larger than the size of the set of testing
cases used by the SVA designers in their program testing. The size of the set of the test cases from the designers is 16.

In conducting the test of the specification using the derived test cases, the following results are observed:

(1) Total of 6 defects were discovered among the five different types of the primitive functional statements; 1 in type (a), 3 in type (b), 1 in type (c) and 1 in type (e).

(2) It was found in integration testing that no case of the data-passing chain occurs in integrating process.

(3) One module used same method of calculations among different groups of sensors. Doubts were raised about its general applicability in the different groups. Testing results of the specification helped to eliminate these doubts and confirmed the generic character. The specification itself does not answer this question.

(4) The total number of test cases is larger than that employed conventionally, but includes the set of testing cases suggested by the designer of the SVA program.

4.3 SUMMARY AND CONCLUSIONS

The major results obtained from applying the Integrated Formal Approach to the development of the SVA specification is summarized here.

Six categories of possible design improvements are identified during the development of specifications. They provide very useful information to the program designers for purpose of checking the correctness of the original design of the program. This actually is the first line of defense in improving design quality. This first defense represents the first feedback from Procedure element to the
design documents as shown in Figure 3.4. After we identified these possible design improvements, they were presented to the system designers and were extensively discussed. It was found that some of the suggested improvements were also identified by the dedicated reviewers of the original design through manual validation and verification (V&V) process.

However, several additional complicated cases relating to the correctness of the software functional designs were found only by using the constructing Procedure of the Integrated Formal Approach. This is because of the Procedure of building FMaps and TMap enforced by the strict the DBTF mathematical formality (structuring the data processing so that all of the data processing logic has been checked to ensure the correct data flows and complete functional specifications). This later point was recognized, by all the groups involved in the discussions of the example software's possible improvements, to be the greatest advantage of the DBTF method and the Integrated Formal Approach.

Not only has the quality of the specifications resulting from this approach been greatly improved, but maintaining the entire program efficiently is also made easier. So also are subsequent modifications of the design via the specifications which may be deemed necessary from point of the view of the system designers. This ease is due to the focus of the Integrated Formal Approach upon the development of the correct and complete specification of the program, not upon the program code itself. Once this specification is developed (and verified by use of the Integrated Formal Approach), the program will automatically be generated by the OO1 Tool Suite. Thus, the tedious work related to propagating any changes of the program to the code level is removed permanently from the program development effort. In comparison, the total effort spent in the development of the entire correct and tested specification is about one and half man-month (when the specifier is very familiar with using the DBTF and OO1 techniques). By contract,
in the general conventional method of program code development, the effort spent to reach the same results is larger (with the program code finally developed manually). The magnitude of the manual development effort of the SVA was 5 man-months (Nova95). This improvement in quality-cost ratios is another recognized advantage of the Integrated Formal Approach. Table 4.7 summarizes the data of program development via conventional method and via specification development in using the Integrated Formal Approach.

<table>
<thead>
<tr>
<th>Original Program Design</th>
<th>Specification Developed in This Study</th>
</tr>
</thead>
<tbody>
<tr>
<td>14 pages of structured text;</td>
<td>2 pages of TMap specification with two elements:</td>
</tr>
<tr>
<td>8 pages of Data Flow Diagrams</td>
<td>--- ProcessRecord</td>
</tr>
<tr>
<td>5 pages of Sensor Table and Process Table</td>
<td>--- SensorRecord</td>
</tr>
<tr>
<td>55 pages of Program Code Listing with:</td>
<td>31 pages of FMaps of the main operation, with:</td>
</tr>
<tr>
<td>--38 sub functions</td>
<td>--- 18 independent modules</td>
</tr>
<tr>
<td>--1925 Lines of C code</td>
<td>--- 9 integration modules</td>
</tr>
<tr>
<td>4 person-weeks for developing specification of the SVA</td>
<td>--- 3 levels of integration of all modules</td>
</tr>
<tr>
<td>4 person-months for producing program of the SVA</td>
<td>--- 416 primitive functional statements</td>
</tr>
<tr>
<td></td>
<td>--- 49 partition functions</td>
</tr>
<tr>
<td></td>
<td>--- 7 valid system states</td>
</tr>
<tr>
<td></td>
<td>6 categories of design improvements;</td>
</tr>
<tr>
<td></td>
<td>Total number of improvements = 21</td>
</tr>
<tr>
<td></td>
<td>134 testing cases derived from the specification</td>
</tr>
<tr>
<td></td>
<td>6 defects detected in the testing process</td>
</tr>
<tr>
<td></td>
<td>Program development effort = 1.5 person-months</td>
</tr>
</tbody>
</table>
CHAPTER 5

Reliability of the Generated Code

At this point, one could consider how the final program code is obtained. One should note that, in this study, the final program code is always generated automatically, based upon the finalized formal specification, by using the OO1 tool’s RAT component. We have not studied quality of the generated code. We assume that this code is generated perfectly by the OO1 tool suite. This may be untrue.

However, based upon part of our literature study on the program reliability assessment, we observe that all of the information of specification development, like defects already found and testing results of the specification, could be used in building a program code reliability model.

This Chapter discusses both the possibilities and limitations of basing initial reliability assessment of the generated code upon the development process.

As is indicated in Chapter 1, it is noted in this study that, so far, the best developed methods for assessing software reliability are based upon the observation of failures of products, augmented by appropriate statistical analysis. This later assessment of reliability usually ignores the abundant information available regarding the development process. Several projects (Pulm93, Ghez88, Hump87, Ehrl87, Litt87, Goel85, Rama82) have studied and summarized a number of analytical models developed during past 20 years. Those models utilize a variety of modeling approaches. These approaches are based mainly on the failure histories of software, and can be classified according to the nature of the failure process as listed below (Goel85):
- **Times Between Failures Models**; the best available models in this class are those of Jelinski and Moranda (JM) model, Schick and Wolverton (SW) model, Goel and Okumoto Imperfect Debugging model, and Littlewood-Verrall Bayesian model.

- **Failure Count Models**; the best available models in this class are the Goel-Okumoto Nonhomogeneous Poisson Process model, the Goel Generalized Nonhomogeneous Poisson Process model, the Musa Execution Time model, Shooman Exponential model, the Generalized Poisson model, the IBM Binomial and Poisson models, and the Musa-Okumoto Logarithmic Poisson Execution Time model.

- **Fault Seeding Models**; the best available model in this class is the Mills Seeding model.

- **Input Domain Based Models**; the best available models in this class are the Nelson model, and the Ramamoorthy and Bastani model.

Again, these best-developed software reliability assessment models do not use information about the software development process, and are formulated such that can be only applied at a later stage of the development cycle.

We conclude from this study that assessment of the development process should be a necessary complement to any such quantitative assessment of the product, carried out subsequently. For example, any confidence placed in the results of a reliability prediction produced during random testing of the program code must depend upon the confidence in the specification against which the software is tested. Furthermore, particularly for high-integrity systems, there simply may not be enough failure data to permit high confidence to be placed upon the software reliability estimate, based upon the available data alone. The assessor and licensor of such systems clearly need to have confidence that the development
of the software is following the proper professional standards, in addition to any failure data-based prediction.

The current state of art in program reliability assessment, based upon observations of the development process, is very limited. In the work reported here, it is found that two difficulties prevent us from doing such an assessment. The first is that a high quality approach was not available for quickly gathering the necessary information from past development processes. The second difficulty is that, even if some information from an ad hoc development process is gathered, there is no sophisticated model yet developed to use it for purposes of program reliability assessment.

In the work reported here, an Integrated Formal Approach is developed. It can help in attacking the first difficulty. This approach uses the best available formal technique and its supporting tool, systematically to develop software from its original system design documents. A large amount of information is produced in the two major elements of this approach. The information produced in the element, Procedure of Constructing Specifications, shows the complexity of the program in terms of the number of data, number of operation and functions, number of levels of data interfaces, and possible missing functions and logic in the original design, etc.. The information produced in another element, Strategy for Testing Specifications, reveals the quality of the specifiers and provides a quality assessment of the developed specifications. However, after all of this effort, one question remains: how to use the above information in the program reliability assessment? This Chapter examines the possibilities and limitations of answering such a question.

A program, as well as its specification, produced from the Integrated Formal Approach of this study usually will have very high quality. However, for its use in safety-critical situations, users still may have concerns about its expected
reliability, i.e., the probability that a program will work properly for the required period of system operation time. When the program is subjected to an extensive random testing regime before release to users, the most important matter of concern is the testing time required to cause the first failure. This is because this failure datum gives the tester a basis for estimating the initial failure rate of the program, and it lets him decide when to end the overall testing regime. Let us define the time to the first program failure as $t_1$.

From our literature survey, we identify the Littlewood-Verrall Bayesian model (Goel85, Litt73) concerning the Times Between Failures Models as the one most suitable for the situation described above. However, we have made some changes in it.

Specifically, in this identified model, the times between failures are assumed to follow an exponential distribution but the parameter of this distribution is treated as being a random variable obeying a gamma distribution. That is, the probability distribution function (pdf) of the time between failures is given by:

$$\text{pdf}(t_i|\lambda_i) = \lambda_i e^{-\lambda_i t_i} \quad (5-1)$$

here $\lambda_i$ is the failure rate of the $i$th failure, while $t_i$ is the time when the $i$th failure occurs. However, the pdf of the parameter $\lambda_i$ is estimated by using the gamma distribution as:

$$\text{pdf}(\lambda_i|\alpha, \psi(i)) = \frac{[\psi(i)]^{\alpha \lambda_i} e^{-\psi(i) \lambda_i}}{(\Gamma \alpha)} \quad (5-2)$$

Corresponding to the conventional forms of parameters of the gamma distribution, we obtain the results:
\[ \alpha = \alpha; \ \beta = 1/\psi(i); \]

As defined by Littlewood and Verrall, the parameter \( \lambda_i \) is a random variable with a distribution given by Equation (5-2). The parameter, \( \lambda_i \), represents a failure rate measure (by this, we mean that a program with a small value of \( \lambda \) is more reliable than a program with a large value of \( \lambda \); and usually \( \lambda_i \) is small than \( \lambda_{i-1} \)). The function \( \psi(i) \) describes the quality of the programmer and the difficulty of the programming task. The function \( \psi(i) \) could be a monotonically increasing function of \( i \). For example, it could be either linearly increasing, or exponentially increasing. It is this function, \( \psi(i) \), that provides a means for incorporating the information of the Integrated Formal Approach into the estimation of the pdf of Equation (5-2). This is explained below.

Our greatest concern is about the value of \( t_1 \), the time of the first failure. The value of \( t_1 \) is determined by the pdf:

\[
\text{pdf}(t_1|\lambda_1) = \lambda_1 e^{-\lambda_1 t_1} \tag{5-3}
\]

and

\[
\frac{[\psi(1)]^\alpha \lambda_1^{(\alpha-1)} e^{-\psi(1) \lambda_1}}{\Gamma(\alpha)}
\]

\[
\text{pdf}(\lambda_1|\alpha, \psi(1)) = \frac{[\psi(1)]^\alpha \lambda_1^{(\alpha-1)} e^{-\psi(1) \lambda_1}}{\Gamma(\alpha)}. \tag{5-4}
\]

In Littlewood and Verrall's theory (Litt73), the parameter \( \alpha \) of the gamma distribution of Equation (5-4) is estimated by Bayesian updating using data of previous failures, as

\[
p_1(\alpha) = p_1(\alpha| \text{failure data}) \propto p(\text{data|}\alpha) * p_0(\alpha) \tag{5-5}
\]
where \( p_0(\alpha) \) is the prior distribution for \( \alpha \) (which can be assumed by the user). However, for the case of the first failure, as we are discussing, there would be no failure data available to permit the estimation of \( p_1(\alpha) \). In that instance, a uniform distribution would be required.

We argue here that the information concerning the number of defects detected in testing the specification could be used to estimate the value of the parameter \( \alpha \) of Equation (5-5). How to conduct this estimation is recommended as a research topic of the future work.

Also, for modeling the increasing function \( \psi(I) \) used in Equation (5-4), previously obtained information concerning the complexity of specifications and information from test cases generated could be used. This modeling, somehow, involves a little bit of subjective treatment concerning the available information. The general trends for \( \psi(I) \) are that it is directly proportional to the number of test cases, inversely proportional to the number of specification defects, and directly proportional to the small probability of defects remaining in specification. However \( \psi(I) \) is inversely proportional to the complexity of the specifications. The speed of increase of \( \psi(I) \) depends upon how the available information is used to estimate the parameters of this function. For example, in Littlewood and Verrall's research (Litt73), an exponential form of \( \psi(i) \) is proposed for fitting a set of data, in the form

\[
\psi(i) = \exp(\beta_0 + \beta_1 i) \tag{5-6}
\]

where the calculation of a set of goodness-of-fit statistics is performed using this relationship and values of different pairs of \( <\beta_0, \beta_1> \). The readers who are interested in modeling \( \psi(I) \) should read papers by Ramamoorthy (Rama82) and Littlewood (Litt73).
CHAPTER 6

Conclusions

6.1 RESULTS OF THIS STUDY

This study is motivated by two major concerns from the industrial experience of developing software for safety-critical systems: 1) we want to find ways to ensure the correct functional design of the programs and to specify that correct design unambiguously; 2) we then want to find ways to produce error-free specifications of software.

Based upon the extensive literature survey of the current available formal techniques and their supporting tools, the Development Before The Fact (DBTF) formal method and its supporting CASE tool: the OO1 Tool Suites are selected to address these concerns. The selection is based upon the set of criteria fixed before the project started.

The selected DBTF techniques and its OO1 Tool Suite were then systematically used in the Integrated Formal Approach designed in this study. This Approach is designed to handle the four stages of verification of development of the specification in order to eliminate all of the classes of specification and coding defects, it leads to a major restructuring of the system design process. The first two elements of the Approach, the transforming Rules and the constructing Procedure treat the first three stages of verification in order to eliminate internal syntax and semantics defects; While the third element of the Approach, the Strategy for Testing Specification, treats the external semantic defects of the specification.

The designed Approach was applied in two case studies, one case is that of the Reactor Protection System, and another case is that of the Signal Validation
Algorithm. The first case is concerned with developing the specification of a functionally simple and small-sized hypothetical program of a safety-critical system. The second case is concerned with developing the specification of a real industrial application program in a safety-critical system, i.e., the Reactor Coolant System of a nuclear power plant.

The results of these applications show several things. 1. This Approach can quickly identify any ambiguities and inconsistencies in the used words and terms, in the original written specification, and incompleteness in defining functions and operations in the original written specification. Some of these defects like incompleteness of the defining functions, can only be identified by this approach. 2. This Approach helps in correcting those identified defects quickly with much less work than in a conventional ad-hoc process. 3. This approach also eases the quick, systematic and complete selection of the required set of test cases for testing its functional correctness. This is accomplished by the designed element of the Integrated Formal Approach: Strategy of test case selection. In this Strategy, a new definition of Equivalence Class of Paths is given based upon the characteristics of the formal specification in DBTF formal language. 4. This Approach supplies some information of specification and its testing. The information is used in a discussion of a conceptual process of updating one's knowledge of the quality of the developed formal specification. However, in this process, two important unjustified assumptions are used which should be further studied in an independent work.

The most valuable result from this research is that the selected DBTF formal method, its supporting tool -- The OO1 Tool Suite, and the proposed Integrated Formal Approach prove to be useful in a practical industrial project right now for improving the current development of safety-critical system software.
6.2 SUGGESTED FUTURE WORK

First, a future effort that could be an extension from this study is to justify the two important assumptions made in the conceptual process of updating one's knowledge of the quality of the tested specification. From our understanding from this study, the commission error rates for all five types of functional statements may be statistically obtained from all DBTF and OO1 Tool related projects. For the assumption on the likelihood function, one might be interested in exploring the Non-Homogeneous Poisson Distribution by examining the trend of the defects detected.

Another future effort which could be extension from this study is the study of the ways of constructing the increasing function, $\psi(i)$ of Equation (5-6) discussed in Chapter 5, by using the information produced in the specification development process.
REFERENCES


Hami92b  Overview: OO1 Super CASE Tool Suite, Hamilton Technology Inc. Document, 1992


Litt87  B. Littlewood, "How Good are Software Reliability Predictions?" Software Reliability: Achievement and Assessment (Edited by B. Littlewood) Blackwell Scientific Publications, 1987, Chapter 11


Parr89 N. Parrington, M. Roper, Understanding Software Testing, Ellis Horwood Limited, 1989


APPENDIX A. THE RPS PROGRAM COLLECT FILE AND C CODE

HAMILTON TECHNOLOGIES INC.
Collector Version: V3.2.3.3-B; Copyright 1991-94.

Documentation for system: update_state
Directory path: /fs/wc/ool/mtl/showcase/protectionnew2/

----------------------------------------------
CONTENTS
----------------------------------------------

FUNCTIONS:
  1 update_state
    1.1 Update_Indi
    1.2 CheckGLOK
    1.3 Update_GL

----------------------------------------------
*** OVERVIEW ***
----------------------------------------------

OVERVIEW of FUNCTIONS. The following section gives an overview of the functionality of the system. The functions in the overview are listed in the order in which they first appear.

1 update_state.

1.1 Update_Indi.

1.2 CheckGLOK.

1.3 Update_GL.

OVERVIEW of STRUCTURES: (TBD).
FUNCTIONS

1 update_state.

UPDATE_STATE uses PRESTATE (which is of type SGL_STATE) and NEWSIGNALS (which is of type SGL_INPUTS) to produce NEWSTATE (which is of type SGL_STATE). Following are the types of objects being used: SGL_STATE, BOOLEAN and SGL_INPUTS.

The following alternatives are performed based on the result of COPY_BOOLEAN:

   Alternative #0. CLONE1_ANY,
   Alternative #1. UPDATE_GL.

Alternative CLONE1_ANY requires the following activity:
NEWSTATE is assigned to be the same object as STATE1.

Alternative UPDATE_GL requires the following activity:

1.1 Update_Indi.

UPDATE_INDI uses PRESTATE (which is of type SGL_STATE) and NEWINPUTS (which is of type SGL_INPUTS) to produce NEWSTATE (which is of type SGL_STATE). Following are the types of objects being used: INDICATION, INDICATORS, BOOLEAN, TRIPCOND, SIGNAL, STR, NAT, SGL_STATE and SGL_INPUTS.

Move to a INDICATORS object, IND1, of a SGL_STATE, named PRESTATE. If the NATural number SIZE1 is greater than or equal to the NATural number SIZE2 then CHKSIZE1 is 'TRUE', otherwise, it is 'FALSE'. If the NATural number SIZE1 is less than or equal to the NATural number SIZE2 then CHKSIZE2 is 'TRUE', otherwise, it is 'FALSE'. The following alternatives are performed based on the result of AND_BOOLEAN:

   Alternative #0. NO_UPDATE_ON_INDICATORS,
   Alternative #1. EQUALSIZEUPDATE.

Alternative NO_UPDATE_ON_INDICATORS requires the following activities:
Create the STRING object MESSAGE1 with the value "Warning!! Number of Signals is not equal to the number of indicators:\n No indicators' update!". Display the STRING " " followed by the Natural number object MESSAGE1. The output object DISPL is the same as the input MESSAGE1. DISPL is explicitly dropped from the current data-flow, and IND1 is identified as the same object as IND1. Move from a INDICATORS object named IND11 to the SGL_STATE object NEWSTATE that contains it.

Alternative EQUALSIZEUPDATE requires the following activities:
CHK1 is 'TRUE' if E1 (a INDICATION) has a child sub-class of "OFF". Move to a TRIPCOND object, INPUTTC, of a SIGNAL, named E2. CHK2 is 'TRUE' if INPUTTC (a TRIPCOND) has a child sub-class of "OK". The following alternatives are performed based on the result of COPY_BOOLEAN:
Alternative #0. CLONE1_ANY,
Alternative #1. DECISION_2.

Alternative CLONE1_ANY requires the following activity:
THISINDSTATE is assigned to be the same object as El.

Alternative DECISION_2 requires the following activities:
The following alternatives are performed based on the result of COPY_BOOLEAN:

Alternative #0. PUTON,
Alternative #1. CLONE1_ANY.

Alternative PUTON requires the following activities:
Move to the INDICATORS set named IND001 from the current INDICATION element named El. The current element of IND001 is now El. Get CUTOFF (a INDICATION; from the set IND001 (a INDICATORS). IND001 now becomes INDNEW. Create CREATE (a INDICATION) with a child sub-class "ON" from its TMap type node KILL. Put CREATE (a INDICATION) into INDNEW (a INDICATORS) to produce IND02. Move to the current INDICATION element named THISINDSTATE of the INDICATORS set named IND02.

Alternative CLONE1_ANY requires the following activity:
THISINDSTATE is assigned to be the same object as El. Move from a INDICATORS object named IND2 to the SGL_STATE object NEWSTATE that contains it.

1.2 CheckGLOK.

CHECKGLOK uses PRESTATE (which is of type SGL_STATE) to produce CHK1 (which is of type BOOLEAN). Following are the types of objects being used: GUARDLINE, BOOLEAN and SGL_STATE.

Move to a GUARDLINE object, OLDGL, of a SGL_STATE, named PRESTATE. CHK1 is "TRUE" if OLDGL (a GUARDLINE) has a child sub-class of "OK".

1.3 UPDATE_GL.

UPDATE_GL uses PRESTATE (which is of type SGL_STATE) and NEWSIGNALS (which is of type SGL_INPUTS) to produce STATE1 (which is of type SGL_STATE). Following are the types of objects being used: SGL_STATE, BOOLEAN, SGL_INPUTS, GUARDLINE, CHAR, VETOCOND, TRIPCOND, SIGNAL and NAT.

Assign the value of GUARD to be 'FALSE'. SET1 is the same object as NEWSIGNALS with its current element located at "1". If the current element of SET1 is the NULL set element, then SETCK is "TRUE"; otherwise it is "FALSE".

The function DOALLUNTILLFINISH, now repeated several times, is comprised of the following activities. The following alternatives are performed based on the result of CR_BOOLEAN:
Alternative #0. DO_NEXT_ELEMENT,
Alternative #1. CLONE1_ANY.

Alternative DO_NEXT_ELEMENT requires the following activities:
Move to the current SIGNAL element named E of the SGL_INPUTS set named SET1. Move to a TRIPCOND object, TC1, of a SIGNAL, named E. CHKTRIP is "TRUE" if TC1 (a TRIPCOND) has a child sub-class of "TRIP". Move to a VETOCOND object, VC1, of a SIGNAL, named E. CHKVETO is "TRUE" if VC1 (a VETOCOND) has a child sub-class of "OK". If the BOOLEANs CHKTRIP and CHKVETO are both 'TRUE', then LOCALTRIP is 'TRUE'; otherwise, it is 'FALSE'. Move to the SGL_INPUTS set named SET2 from the current SIGNAL element named E. The current element of SET2 is now E. Set SET2, a SGL_INPUTS set, to the next element in the "R" direction. SET2 now becomes SETN. If the current element of SETN is the NULL set element, then SETCK1 is "TRUE"; otherwise it is "FALSE". Repeat the function DOALLUNTILLFINISH with the following input changes: replace SET1 with SETN, GUARD with LOCALTRIP and SETCK with SETCK1.

Alternative CLONE1_ANY requires the following activity:
GLTRIP is assigned to be the same object as GUARD. The following alternatives are performed based on the result of COPY_BOOLEAN:

Alternative #0. CLONE1_ANY,
Alternative #1. DO_SWITCH.

Alternative CLONE1_ANY requires the following activity:
STATE1 is assigned to be the same object as PRESTATE.

Alternative DO_SWITCH requires the following activities:
Get GL1 (a GUARDLINE) from PRESTATE (a SGL_STATE). PRESTATE now becomes STATE2. Create GL2 (a GUARDLINE) with a child sub-class "TRIP" from its TMap type node B01. Put GL2 (a GUARDLINE) into STATE2 (a SGL_STATE) to produce STATE1.
*/

VERSION: V3.2.3.7
: 1.7.0.0
MAIN MODULE for: UPDATE_STATE
000000

SCCSID: @(#) %M% %I% of %G%.

Mon Aug 7 11:57:14 1995

*/

#include <stdio.h>
#include <errno.h>
#include <math.h>
#include "UNIXDEFS.h"
#include "OS.h"
#include "SGL_STATE.h"
#include "BOOLEAN.h"
#include "SGL_INPUTS.h"
int do_IDSC=1; int HTIENVOS=0;

DECLARE_OS(globalOS)

main(argc,argv,envp)
int argc;
char **argv,**envp;
{
DECLARE_SGL_STATE(V0PRESTATE)
DECLARE_SGL_INPUTS(V0NEWSIGNALS)
DECLARE_SGL_STATE(V0NEWSTATE)

IMPORT_CMDLINE

INIT_IDSC();
NEWSTACK_IDSC();
XIN_SGL_STATE("PreState",V0PRESTATE)
XIN_SGL_INPUTS("NewSignals",V0NEWSIGNALS);
fUPDATE_STATE(V0PRESTATE,V0NEWSIGNALS,&V0NEWSTATE);
XOUT_SGL_STATE("NewState",V0NEWSTATE)

EXITMAP_OS(0)
ACCESS_DEBUG_OMAP
}
/*

001-Generated C code for functional specification 'UPDATE_STATE'

VERSION: 3.2.3.9 C-RAT
: 1.6.0.0
OPERATION: UPDATE_STATE
GENERATED: Mon Aug 7 11:55:03 1995

SCCSID: @(#) %M% %I% of %G%.
*/

#include "SGL_STATE.h"
#include "BOOLEAN.h"
#include "SGL_INPUTS.h"
#include <stdio.h>
#include <math.h>
#include <errno.h>
#include "BOOLEAN.h"
#include "NAT.h"

fUPDATE_STATE(V0PRESTATE, V0NEWSIGNALS, V0NEWSTATE)
IDECLAIM SGL_STATE (V0PRESTATE)
IDECLAIM SGL_INPUTS (V0NEWSIGNALS)
ODECLAIM SGL_STATE (V0NEWSTATE)
{
    /*__LOCAL_VARIABLE_DECLARATIONS__ */

DECLARE BOOLEAN (V0CHKGL)
DECLARE SGL_STATE (V0STATE1)
DECLARE BOOLEAN (V0PF4)

    /*__ITERATION_VARIABLE_DECLARATIONS__ */

    /*__CONSTANT_DECLARATIONS_AND_ASSIGNMENTS__ */

    /*__FUNCTION_SOURCE_CODE_BEGINNING__ */

fUPDATE_INDI (V0PRESTATE, V0NEWSIGNALS, &V0STATE1);
fCHECKGLOK (V0STATE1, &V0CHKGL);
COPY BOOLEAN (V0CHKGL, V0PF4)
if (V0PF4<1)
    {if (V0PF4 == REJECT_BOOLEAN) (REJECT_TEST_BOOLEAN()); /* 0GUARDLINE_UPDATE */
    CLONE SGL_STATE (V0STATE1, *V0NEWSTATE)
    }else{
    /*UPDATE_GL*/
    fUPDATE_GL (V0STATE1, V0NEWSIGNALS, V0NEWSTATE);
    }

return;
}

/*----------------- end of source -----------------*/
/*
001-Generated C code for functional specification 'UPDATE_INDI'

VERSION: 3.2.3.9 C-RAT
: 1.6.0.0
OPERATION: UPDATE_INDI

SCCSID: @(#) %M% %I% of %G%
*/
#include "INDICATORS.h"
#include "INDICATION.h"
#include "BOOLEAN.h"
#include "TRIPCOND.h"
#include "SIGNAL.h"
#include "STR.h"
#include "NAT.h"
#include "SGL_STATE.h"
#include "SGL_INPUTS.h"
#include "CHAR.h"

#include <stdio.h>
#include <math.h>
#include <errno.h>
#include "BOOLEAN.h"
#include "NAT.h"

fUPDATE_INDI(V0PRESTATE,V0NEWINPUTS,
 V0NEWSTATE)
IDeclare SGL_STATE(V0PRESTATE)
IDeclare SGL_INPUTS(V0NEW INPUTS)
ODeclare SGL_STATE(V0NEWSTATE)
{
   /* ___ LOCAL_VARIABLE_DECLARATIONS ___ */
DECLARE_INDICATORS(V0IND02)
DECLARE_INDICATION(V0CREATE)
DECLARE_BOOLEAN(V0KILL)
DECLARE_INDICATORS(V0INDNEW)
DECLARE_INDICATION(V0CUTOFF)
DECLARE_INDICATORS(V0IND001)
DECLARE_BOOLEAN(V0CHK2)
DECLARE_TRIPCOND(V0INPUTTC)
DECLARE_BOOLEAN(V0CHK1)
DECLARE_INDICATION(V0THISINDSTATE)
DECLARE_SIGNAL(V0E2)
DECLARE_INDICATION(V0E1)
DECLARE_INDICATORS(V0IND2)
DECLARE_INDICATORS(V0IND11)
DECLARE_STR(V0DISP1)
DECLARE_STR(C16)
DECLARE_STR(V0MESSAGE1)
DECLARE_STR(C18)
DECLARE_BOOLEAN(V0CHKSIZE2)
DECLARE_BOOLEAN(V0CHKSIZE1)
DECLARE_NAT(V0SIZE2)
DECLARE_NAT(V0SIZE1)
DECLARE_INDICATORS(V0IND1)
DECLARE BOOLEAN (V0PF7)
DECLARE BOOLEAN (V0PF19)
DECLARE BOOLEAN (V0PF21)
DECLARE SGL_INPUTS (V1_0NEWSIGNALS03)
DECLARE CHAR (C31)
DECLARE SGL_INPUTS (V1_0NEWSIGNALS02)
DECLARE INDICATORS (V1_0IND03)
DECLARE CHAR (C34)
DECLARE INDICATORS (V1_0IND02)
DECLARE BOOLEAN (V1_0CHK2)
DECLARE BOOLEAN (V1_0CHK1)
DECLARE SGL_INPUTS (V1_0NEWSIGNALS01)
DECLARE NAT (C39)
DECLARE INDICATORS (V1_0IND01)
DECLARE NAT (C41)
DECLARE BOOLEAN (V1_PF7)

/* ITERATION_VARIABLE_DECLARATIONS__ */

int recl_0DOALLTILLFINISH;
DECLARE INDICATORS (R171_0IND01)
DECLARE SGL_INPUTS (R171_0NEWSIGNALS01)

/* ___CONSTANT_DECLARATIONS_AND_ASSIGNMENTS__ */

DOT_K_STR ("", C16)
DOT_K_STR ("Warning!! Number of Signals is not equal to the number of indicators!\n"
DOT_K_CHAR (’R’, C31)
DOT_K_CHAR (’R’, C34)
DOT_K_NAT (1, C39)
DOT_K_NAT (1, C41)

/* ___FUNCTION_SOURCE_CODE_BEGINNING__ */

MOVETO_INDICATORS_SGL_STATE (V0PRESTATE, V0IND1)
SIZE_INDICATORS (V0IND1, V0SIZE1)
SIZE_SGL_INPUTS (V0NEWINPUTS, V0SIZE2)
GE_NAT (V0SIZE1, V0SIZE2, V0CHKSIZE1)
LE_NAT (V0SIZE1, V0SIZE2, V0CHKSIZE2)
AND_BOOLEAN (V0CHKSIZE1, V0CHKSIZE2, V0PF7)
if (V0PF7 < 1)
  {if (V0PF7 == REJECT_BOOLEAN) {REJECT_TEST_BOOLEAN()} /* 0SIZECHECK */
    K_STR (C18, V0IND1, V0MESSAGE1)
    DISPLAY_STR (C16, V0MESSAGE1, V0DISP1)
    CLONE_INDICATORS (V0IND1, V0IND11)
    MOVETO_SGL_STATE_INDICATORS (V0IND11, *V0NEWWSTATE)
  }
else{
  /* 0EQUALSIZEUPDATE*/
  LOCATE_INDICATORS (C41, V0IND1, V1_0IND01)
  LOCATE_SGL_INPUTS (C39, V0NEWINPUTS, V1_0NEWSIGNALS01)
  R171_0IND01=V1_0IND01;
  R171_0NEWSIGNALS01=V1_0NEWSIGNALS01;
  recl_0DOALLTILLFINISH=1;
  while (recl_0DOALLTILLFINISH--){
    ATNULL_INDICATORS (V1_0IND01, V1_0CHK1)
    ATNULL_SGL_INPUTS (V1_0NEWSIGNALS01, V1_0CHK2)
    OR_BOOLEAN (V1_0CHK1, V1_0CHK2, V1_PF7)
    if (V1_PF7 < 1)
      {if (V1_PF7 == REJECT_BOOLEAN) {REJECT_TEST_BOOLEAN()} /* 1_0FOWARD_OR_STOP */
        MOVETO_INDICATORS (V1_0IND01, V0E1)
        MOVETO_SGL_INPUTS (V1_0NEWSIGNALS01, V0E2)
        IS_OFF_INDICATION (V0E1, V0CHK1)
        MOVETO_TRIPCND_SIGNAL (V0E2, V0INPUTTC)
IS_OK_TRIPCOND (V0INPUTC, V0CHK2)
COPY_BOOLEAN (V0CHK1, V0PF19)
    if (V0PF19 < 1)
        {if (V0PF19 == REJECT_BOOLEAN) {REJECT_TEST_BOOLEAN() } /* ODECISION_1 */
         CLONE_INDICATION (V0E1, V0THISINDSTATE)
        }
else{ /* ODECISION_2 */
    COPY_BOOLEAN (V0CHK2, V0PF21)
        if (V0PF21 < 1)
            {if (V0PF21 == REJECT_BOOLEAN) {REJECT_TEST_BOOLEAN() } /* ODECISION_2 */
             MOVETO_INDICATORS_INDICATION (V0E1, V0IND01)
             GET_INDICATIONS (V0IND01, V0CUTOFF, V0INDNEW)
             D_INDICATION (V0CUTOFF, V0KILL)
             K_ON_INDICATION (V0KILL, V0CREATE)
             PUT_INDICATORS (V0CREATE, V0INDNEW, V0IND02)
             MOVETO_INDICATORS (V0IND02, V0THISINDSTATE)
            }
else{ /* CLONE1_ANY */
             CLONE_INDICATION (V0E1, V0THISINDSTATE)
            }
    }
    MOVETO_INDICATORS_INDICATION (V0THISINDSTATE, V1_0IND02)
    NEXT_INDICATORS (C34, V1_0IND02, V1_0IND03)
    MOVETO_SGL_INPUTS_SIGNAL (V0E2, V1.ONEWSIGNALS02)
    NEXT_SGL_INPUTS (C31, V1.ONEWSIGNALS02, V1.ONEWSIGNALS03)
        rec1_0DOALLTILLFINISH=1;
        V1_0IND01=V1_0IND03;
        V1.ONEWSIGNALS01=V1.ONEWSIGNALS03;
else{ /* CLONE1_ANY */
             CLONE_INDICATORS (V1_0IND01, V0IND2)
            }
}
V1_0IND01=R171_0IND01;
V1.ONEWSIGNALS01=R171.ONEWSIGNALS01;
    MOVETO_SGL_STATE_INDICATORS (V0IND2, *V0NEWSTATE)
}
return;
}
/*
001-Generated C code for functional specification 'CHECKGLOK'

VERSION: 3.2.3.9 C-RAT
: 1.6.0.0
OPERATION: CHECKGLOK

SCCSID: @(#) %M% %I% of %G%.
*/
#include "GUARDLINE.h"
#include "BOOLEAN.h"
#include "SGL_STATE.h"
#include <stdio.h>
#include <math.h>
#include <errno.h>
#include "BOOLEAN.h"
#include "NAT.h"

fCHECKGLOK(V0PRESTATE,
V0CHK1)
IDECLARE_SGL_STATE(V0PRESTATE)
ODECLARE_BOOLEAN(V0CHK1)
{
   /* * LOCAL_VARIABLE_DECLARATIONS__ */
DECLARE_GUARDLINE(V0OLDGL)
   /* * ITERATION_VARIABLE_DECLARATIONS__ */
   /* * __CONSTANT_DECLARATIONS_AND_ASSIGNMENTS__ */
   /* * FUNCTION_SOURCE_CODE_BEGINNING__ */
    MOVETO_GUARDLINE_SGL_STATE(V0PRESTATE,V0OLDGL)
    IS_OK_GUARDLINE(V0OLDGL,*V0CHK1)

return;
}

/* --------------- end of source ---------------*/
/*

001-Generated C code for functional specification 'UPDATE_GL'

VERSION: 3.2.3.9 C-RAT
         : 1.6.0.0
OPERATION: UPDATE_GL

SCCSID: @(#) %M% %I% of %G%.

*/
#include "SGL_STATE.h"
#include "BOOLEAN.h"
#include "SGL_INPUTS.h"
#include "GUARDLINE.h"
#include "CHAR.h"
#include "VETOCOND.h"
#include "TRIPCOND.h"
#include "SIGNAL.h"
#include "NAT.h"

#include <stdio.h>
#include <math.h>
#include <errno.h>
#include "BOOLEAN.h"
#include "NAT.h"

fUPDATE_GL(V0PRESTATE,V0NEWINPUTS, V0STATE1)
IDeclare_SGL_STATE(V0PRESTATE)
IDeclare_SGL_INPUTS(V0NEWINPUTS)
ODeclare_SGL_STATE(V0STATE1)
{
   /* __LOCAL_VARIABLE_DECLARATIONS__ */
DECLARE_GUARDLINE(V0GL2)
DECLARE_BOOLEAN(V0BO1)
DECLARE_SGL_STATE(V0STATE2)
DECLARE_GUARDLINE(V0GL1)
DECLARE_BOOLEAN(V0SETCK1)
DECLARE_SGL_INPUTS(V0SETN)
DECLARE_CHAR(C7)
DECLARE_SGL_INPUTS(V0SET2)
DECLARE_BOOLEAN(V0LOCALTRIP)
DECLARE_BOOLEAN(V0CHKVETO)
DECLARE_VETOCOND(V0VC1)
DECLARE_BOOLEAN(V0CHKTRIP)
DECLARE_TRIPCOND(V0TC1)
DECLARE_SIGNAL(V0E)
DECLARE_BOOLEAN(V0SETCK)
DECLARE_SGL_INPUTS(V0SET1)
DECLARE_NAT(C17)
DECLARE_BOOLEAN(V0GLTRIP)
DECLARE_BOOLEAN(V0GUARD)
DECLARE_BOOLEAN(V0PF6)
DECLARE_BOOLEAN(V0PF19)
/* __ITERATION_VARIABLE_DECLARATIONS__ */

int rec0DOALLTILLFINISH;
DECLARE_SGL_INPUTS(R60SET1)
DECLARE_BOOLEAN(R60GUARD)
DFCLARE_BOOLEAN(R60SETCK)

/* __CONSTANT_DECLARATIONS_AND_ASSIGNMENTS__ */

DOT_K_CHAR('R', C7)
DOT_K_NAT(1, C17)

/* __FUNCTION_SOURCE_CODE_BEGINNING__ */

KF_BOOLEAN(V0PRESTATE, V0GUARD)
LOCATE_SGL_INPUTS(C17, V0NEWSIGNALS, V0SET1)
ATNULL_SGL_INPUTS(V0SET1, V0SETCK)
R60SET1 = V0SET1;
R60GUARD = V0GUARD;
R60SETCK = V0SETCK;
rec0DOALLTILLFINISH = 1;
while (rec0DOALLTILLFINISH == 1) {
    OR_BOOLEAN(V0GUARD, V0SETCK, V0PF6)
    if (V0PF6 == 1)
        {if (V0PF6 == REJECT_BOOLEAN) {REJECT_TEST_BOOLEAN()}} /* 0DOALLTILLFINISH */
        MOVETO_SGL_INPUTS(V0SET1, V0E)
        MOVETO_TRIPCOND_SIGNAL(V0E, V0TC1)
        IS_TRIP_TRIPCOND(V0TC1, V0CHKTRIP)
        MOVETO_VETOCOND_SIGNAL(V0E, V0VC1)
        IS_OK_VETOCOND(V0VC1, V0CHKVETO)
        AND_BOOLEAN(V0CHKTRIP, V0CHKVETO, V0LOCALTRIP)
        MOVETO_SGL_INPUTS_SIGNAL(V0E, V0SET2)
        NEXT_SGL_INPUTS(C7, V0SET2, V0SETN)
        ATNULL_SGL_INPUTS(V0SETN, V0SETCK1)
        rec0DOALLTILLFINISH = 1;
        V0SET1 = V0SETN;
        V0GUARD = V0LOCALTRIP;
        V0SETCK = V0SETCK1;
    }
else

        /* CLONE1_ANY*/
        CLONE_BOOLEAN(V0GUARD, V0GLTRIP)

    }

V0SET1 = R60SET1;
V0GUARD = R60GUARD;
V0SETCK = R60SETCK;
COPY_BOOLEAN(V0GLTRIP, V0PF19)
    if (V0PF19 == 1)
        {if (V0PF19 == REJECT_BOOLEAN) {REJECT_TEST_BOOLEAN()}} /* 0UPDATE_2 */
        CLONE_SGL_STATE(V0PRESTATE, *V0STATE1)
    }
else

        /* 0DO_SWITCH*/
        GET_GUARDLINE_SGL_STATE(V0PRESTATE, V0GL1, V0STATE2)
        D_GUARDLINE(V0GL1, V0BO1)
        K_TRIP_GUARDLINE(V0BO1, V0GL2)
        PUT_GUARDLINE_SGL_STATE(V0GL2, V0STATE2, *V0STATE1)

}

return;

/* ---------------- end of source -------------------*/