The Use of Software Faults in Software Reliability Assessment and Software Mutation Testing

Dissertation

Presented in Partial Fulfillment of the Requirements for the Degree Doctor of Philosophy in the Graduate School of The Ohio State University

By
Xiang Li, M.S.
Graduate Program: Nuclear Engineering

The Ohio State University
2015

Dissertation Committee:
Carol Smidts, Advisor
Tunc Aldemir
Lei Cao
Abstract

A software fault is a structural imperfection in a software system that may lead to the system eventually failing [1]. Software faults have been heavily studied before, especially in the software reliability [2, 3] and software testing communities [4, 5]. In this research, we focus on two usages of software faults to improve the software quality: (1) Using software fault information at various stages of software development to assess software reliability, and (2) Seeding software faults into the original source code to drive software test case development. More specifically:

For (1), we have developed the Extended Finite State Machine-based Reliability Prediction System (EFSM-based RePS) method to assess software reliability [6, 7]. This method utilizes the uncovered faults from the N-1”th version of the software to predict the software reliability of the N”th version. The software documentation at different development life cycles, which includes Software Requirement Specifications (SRS), Software Design Documents (SDD) and the source code, is collected. All this information is analyzed and used to construct a hierarchical model of the software. The Operating Profile (OP) of the software is then used to assess software reliability. A tool called the Automated Reliability Prediction System (ARPS) is also developed which implements the EFSM-based RePS methodology. An experiment was conducted to evaluate the tool’s usability where human subjects were recruited, trained and tested.

For (2), we have investigated and improved the software mutation testing technique [34-37], which is a fault-based automatic software testing technique. In mutation testing, the tester first defines a set of rules to systematically seed faults into the source code. Each seeded fault results in a new version of the software, which is called a mutant. Test suites are then developed to
distinguish the mutants from the original program (i.e., to kill the mutants). Since these test suites can find the seeded faults, they are also helpful to find the indigenous ones. In this research we introduce heuristics to strongly kill mutants. To verify these heuristics, they are applied to sample programs. It is found that these heuristics indeed help propagate data state difference further than weak mutation, which will eventually lead to test cases that can strongly kill mutants.
Acknowledgements

I am grateful for the assistance of so many people in producing this work.

First of all, I would like to thank my advisor Professor Smidts. Without her immense help in guiding my research, this thesis would have been impossible. As an advisor, she assisted me in every aspect from research brainstorming to writing this thesis.

Second, I would like to thank my lab mate Chetan Mutha, for we work together on the Automated Reliability Prediction System (ARPS) project. His contribution to the project is very important and valuable.

Third, I would like to thank our three Java programmers: Jenna McAuley, Bill Dazey and Kevin Smearsoll. We worked together during my PhD study on the ARPS project. Without their hardworking we would have never been able to complete the ARPS tool.
Vita

July 2007............................................................... B.S. Engineering Physics, Tsinghua University

July 2009............................................................... M.S. Engineering Physics, Tsinghua University

September 2009 to present............................... Graduate Research Associate, Nuclear Engineering Program, The Ohio State University

Publications:


5. J. Gupta, X. Li, et al. Application of Hazard and Operability Study (HAZOP) to Safety Related Software(s). 29th ISSC, Las Vegas, NV, August 8–12, 2011. (Best paper award at ISSC 2011)


Field of Study

Major Field: Nuclear Engineering
Table of Contents

Abstract .................................................................................................................................................. ii
Acknowledgements ................................................................................................................................. iv
Vita .......................................................................................................................................................... v
List of Tables ......................................................................................................................................... xi
List of Figures ......................................................................................................................................... xiii
Acronyms and symbols ............................................................................................................................ xviii
Chapter 1: Introduction ............................................................................................................................. 1
Chapter 2: The Extended Finite State Machine Based Reliability Prediction System (EFSM-based RePS) ......................................................................................................................................................... 4
  2.1 Introduction to the EFSM-based RePS Methodology ................................................................. 4
    2.1.1 General Concepts of Software Reliability and the Reliability Prediction System .......... 4
    2.1.2 How Do Software Failures Occur? The Execution, Infection and Propagation Constraints ................................................................................................................................................. 5
    2.1.3 The EFSM-based RePS Methodology ............................................................................... 9
  2.2 Related Work ................................................................................................................................. 11
  2.3 Original High Level Extended Finite State Machine Model Construction ........................... 13
    2.3.1 Understanding the SRS .................................................................................................... 13
    2.3.2 Extracting Information from the SRS .................................................................................. 17
    2.3.3 Constructing the Original HLEFSM ................................................................................... 17
  2.4 SRS Defect Definitions and Templates ......................................................................................... 27
    2.4.1 Common SRS Defects Identified in this Study ................................................................. 27
    2.4.2 How a Mapped Defect is Represented in the Modified HLEFSM ..................................... 30
    2.4.3 Application of the Defects Templates to the SCS Case Study ......................................... 41
  2.5 Execution Analysis ......................................................................................................................... 47
2.5.1 What is an Operational Profile ................................................................. 47
2.5.2 An Example Illustrating How Execution Analysis is Conducted ................ 48
2.5.3 Methodology for Execution Analysis ......................................................... 50

2.6 Infection Analysis .......................................................................................... 63
2.6.1 Infection Analysis for “Missing Instance of Function” ............................... 63
2.6.2 Infection Analysis for “Extra Instance of Function” and “Incorrect/Ambiguous Function Call” ................................................................. 64
2.6.3 Infection Analysis for “Incorrect/Ambiguous Predicate” ............................ 64
2.6.4 Infection Analysis for “Missing Predicate” ................................................. 68
2.6.5 Infection Analysis for “Extra Predicate” ................................................... 69
2.6.6 Infection Analysis for the SCS Case Study ............................................... 70

2.7 Propagation Analysis ...................................................................................... 70

2.8 Modeling Defects at the SDD Level .............................................................. 73
2.8.1 Introduction to the SDD Level Analysis .................................................... 73
2.8.2 Construction of the Original Low Level Extended Finite State Machine Model ........... 74
2.8.3 Mapping the SDD Level Defects and Constructing the Low Level Extended Finite State Machine Model ................................................................. 74
2.8.4 Execution, Infection and Propagation Analysis at the SDD Level ............... 75
2.8.5 SRS Level Infection Analysis for the openValve3 Function at the SDD Level ........ 75
2.8.6 Software Reliability Assessment for the SCS Case Study ......................... 78

2.9 Software Reliability Assessment with the ARPS Tool .................................. 78
2.9.1 Introduction ............................................................................................... 78
2.9.2 Modules of the ARPS Tool .................................................................... 79
2.9.3 Constructing the Original HLEFSM ....................................................... 80
2.9.4 Collecting SRS level Defects ................................................................ 92
2.9.5 How to Map Defects related to the SRS Logic ........................................ 97
2.9.6 LLEFSM Construction and Defect Modeling ......................................... 103

2.10 Experimental Validation for the ARPS Tool .............................................. 106
2.10.1 Introduction ............................................................................................ 106
2.10.2 Research Subject Identification ............................................................. 107
2.10.3 Research Design ................................................................................... 107
2.10.4 Data Analysis ................................................................. 111
2.10.5 Threat to Validity .......................................................... 125
2.10.6 Summary of the Experimental Results ............................. 126

Chapter 3: Heuristics to Improve the Software Strong Mutation Score ................................................. 128
3.1 Introduction to the Software Mutation Testing Techniques .......................................................... 128
  3.1.1 What is Mutation Testing? .............................................. 128
  3.1.2 Mutant Operators .......................................................... 129
  3.1.3 Requirements for Killing Mutants .................................... 132
  3.1.4 Equivalent Mutants ...................................................... 134
  3.1.5 Mutation Score ............................................................ 135
  3.1.6 Current State of Mutation Testing ................................... 135
  3.1.7 Why Strong Mutation instead of Weak Mutation .......... 138
3.2 Problem Analysis and Heuristics for Strongly Killing Mutants ...................................................... 139
  3.2.1 Assumptions about the Programs under Study ................. 139
  3.2.2 Modified Control Flow Graph (CFG) for Program Logic Representation ............................. 140
  3.2.3 Strongly Killing Mutants: Essentially a Branch-Selection Problem .......................................... 144
  3.2.4 Definition of Data State ............................................... 150
  3.2.5 Heuristics to Strongly Kill Mutants ................................ 152
3.3 Verification of the Heuristics ................................................. 153
  3.3.1 The Sample Program for Verification of the Heuristics ....... 153
  3.3.2 Verification Results for the Sample Program ...................... 155
3.4 An Algorithm to Implement the Heuristics ........................................ 160
  3.4.1 An Algorithm to Determine the Data State ......................... 160
  3.4.2 An Algorithm to Implement the Heuristics ....................... 175
  3.4.3 Application of the Algorithm to a Sample Program .......... 180
3.5 Heuristics for Propagating Data State Difference Though Loops .................................................. 183
3.6 Current Status of the Implementation of the Automated Software Mutation Testing System (ASMTS) Tool ................................................................. 188

Chapter 4: Conclusions and Future Research .............................................................................. 189
References ................................................................................................................................. 191
Appendix A: The SCS Case Study for the ARPS Tool ............................................. 195
A.1. Software Requirement Specifications for a Simple Control System .................... 195
  A.1.1. Introduction to the Simple Control System (SCS) .................................... 195
  A.1.2. Specific Functional Requirements ................................................................ 195
A.2. Software Design Documentation for the SRS function openValve3 ..................... 197
  A.2.1. Introduction to the openValve3 function .................................................. 197
  A.2.2. Specific Design Description ....................................................................... 198
A.3. SRS Level Function Cards for the SCS Case Study ........................................... 199
A.4. SDD Level Function Cards for the SCS Case Study ........................................... 202
A.5. Original Version of the High Level Extended Finite State Machine (HLEFSM) Model for the SCS Case Study ................................................................. 204
A.6. Original Version of the SDD Level Extended Finite State Machine Model for openValve3 ............................................................................................................. 205
A.7. Defect Report for SCS’s SRS ............................................................................. 206
A.8. Defect Report for openValve3’s SDD ............................................................... 207
A.9. Modified Function Cards for the SCS Case Study ............................................. 208
A.10. Modified Function Cards at the SDD level for openValve3 ................................. 208
A.11. Modified HLEFSM Model for the SCS Case Study ....................................... 209
A.12. Modified SDD level EFSM Model for openValve3 ........................................... 210
A.13. Operational Profile (OP) of SCS ................................................................... 211
A.14. Operational Profile (OP) for openValve3 at the SDD level ................................. 211
A.15. Infection and Propagation Data for the PIE Analysis for the SRS ..................... 212
A.16. Infection and Propagation Data for the PIE Analysis for the SDD ..................... 214
A.17. Calculation of SCS’s Software Reliability ..................................................... 215
Appendix B: Defect Templates for SRS/SDD Level Defect Mapping ......................... 217
B.1. Defect Templates for Category 1 Defects: Defects for the Definition of Level 1 Functions .......................................................................................................... 217
  B.1.1. Missing (Definition of) Function ................................................................. 217
  B.1.2. Extra (Definition of) Function .................................................................... 218
  B.1.3. Incorrect/Ambiguous Function Name ....................................................... 218
  B.1.4. Function with Incorrect Logic .................................................................... 219
B.1.5. Function with Incorrect Functionality ................................................................. 219
B.2. The Defect Templates for Category 2 Defects: Defects Related to Inputs .................. 220
  B.2.1. Missing Input ........................................................................................................ 220
  B.2.2. Extra Input .......................................................................................................... 220
  B.2.3. Incorrect/Ambiguous Input Name ........................................................................ 221
  B.2.4. Input with Incorrect Type .................................................................................... 221
  B.2.5. Input with Incorrect Range ................................................................................ 222
B.3. Defect Templates for Category 3 Defects: Defects Related to Outputs .................... 223
B.4. Defect Templates for Category 4 Defects: Defects Related to Internal Variables ........ 223
B.5. Defect Templates for Category 5 Defects: Defects for the Logic of the Level 0 Function .................................................................................................................. 224
  B.5.1. Missing Instance of Function ................................................................................ 224
  B.5.2. Extra Instance of Function ................................................................................... 224
  B.5.3. Incorrect/Ambiguous Function Call ..................................................................... 225
  B.5.4. Missing Predicate ............................................................................................... 225
  B.5.5. Extra Predicate ................................................................................................... 226
  B.5.6. Incorrect/Ambiguous Predicate ......................................................................... 226
Appendix C: Description of the Functions Used in the Algorithms .................................. 228
  C.1 Functions Used in the Algorithm to Construct the Dependency Graph .................... 228
  C.2 Functions used in the algorithm to determine the data state .................................... 230
# List of Tables

Table 1 The Execution, Infection and Propagation constraints ........................................ 8  
Table 2 SRS/SDD Defects that have been identified in this study ...................................... 28  
Table 3 SCS’s Defect Report................................................................................................. 41  
Table 4 Sample OP of an ATM in a bank........................................................................... 47  
Table 5 The OP information for the example in Figure 48.................................................... 50  
Table 6 The Operational Profile of the SCS case study.......................................................... 61  
Table 7 Typical examples for different types of variables...................................................... 65  
Table 8 Summary of the conditions of the example ............................................................... 67  
Table 9 Summary Execution, Infection and Propagation data for openValve3....................... 77  
Table 10 Summary of all the Execution, Infection and Propagation data for the SCS case study 78  
Table 11 OP-related information for Temperature ................................................................ 95  
Table 12 Grading policy used in the experiment .................................................................. 108  
Table 13 The design of the experiment.................................................................................. 111  
Table 14 Summary of the statistics for $EI$ (a score expressed as the percentage of a task correctly completed by a subject) ................................................................................. 115  
Table 15 Summary test statistics for $NE$............................................................................ 119  
Table 16 $T$ statistics summary............................................................................................ 121  
Table 17 Missing statistics summary................................................................................... 122  
Table 18 Incorrect statistics summary.................................................................................. 123  
Table 19 Summary of all the results...................................................................................... 127  
Table 20 List of mutant operators......................................................................................... 130  
Table 21 List of mutants for the sample program in Figure 129............................................ 145  
Table 22 List of mutants for Case 3..................................................................................... 149  
Table 23 Branch performance information for the sample program.................................... 156  
Table 24 Path performance information for mutant #1......................................................... 156  
Table 25 Summary of the verification results for mutant #1................................................ 157  
Table 26 Summary of the verification results for mutant #2................................................ 158  
Table 27 Summary of the verification results for mutant #3................................................ 159  
Table 28 Statements where a certain variable should be killed from the data state vector..... 173  
Table 29 Data state calculation for the sample program...................................................... 174
Table 30 Branch performance information for the sample program in Figure 135 .................. 181
Table 31 Path definition for sample program ............................................................................ 181
Table 32 Static data state of the statements in the sample program ...................................... 185
Table 33 Descriptions for SCS’s SRS defects ........................................................................... 207
Table 34 Descriptions for openValve3’s SDD defects ................................................................. 207
Table 35 SCS’s operational Profile ......................................................................................... 211
Table 36 Operational Profile for openValve3 .......................................................................... 211
Table 37 Infection and Propagation data for the SCS’s PIE analysis ...................................... 212
Table 38 Infection and Propagation data required for openValve3’s PIE analysis ................. 214
List of Figures

Figure 1 RePS constitution ................................................................. 5
Figure 2 A sample program for explaining the Execution, Infection and Propagation constraints . 6
Figure 3 The control flow of the sample program ........................................ 6
Figure 4 Overall layout of the EFSM-based RePS method ................................. 6
Figure 5 A typical SRS outline based on the IEEE 830-1998 standard .................. 10
Figure 6 A typical outline of section 3 of the SRS ........................................... 15
Figure 7 The hierarchical structure of the SRS in this study .................................. 16
Figure 8 Template for the SRS level 0 function card ...................................... 18
Figure 9 The function card of the SRS level 0 function of the Simple Control System case study .............................................................................. 19
Figure 10 The function card of the level 1 function initializeTheSystem .................. 20
Figure 11 The SRS level 0 function represented using the EFSM graphical language .......................................................... 20
Figure 12 EFSM graphical representation for a typical state transition ...................... 21
Figure 13 State transition caused by openValve ................................................. 22
Figure 14 EFSM representation for “If Temperature is greater than 50 °C, then open valve #1.” 22
Figure 15 EFSM representation for “If Temperature is greater than 50 °C, then open valve #1; otherwise open valve #2” ....................................................... 23
Figure 16 EFSM representation for the if-elseif example ..................................... 24
Figure 17 A complex HLEFSM consisting of multiple state transitions and nested predicates .... 24
Figure 18 The state transition corresponding to the first line of the logic for the SCS case study 25
Figure 19 The predicate corresponding to the second line of the logic for the SCS case study .... 26
Figure 20 The predicate corresponding to the third line of the logic for the SCS case study .... 26
Figure 21 The state transition corresponding to the last line of the logic of the SCS case study .. 27
Figure 22 The original HLEFSM for the SCS case study ..................................... 27
Figure 23 A typical HLEFSM consisting of three consecutive state transitions .. 30
Figure 24 A general case for defect mapping ................................................... 31
Figure 25 Defect template for Function with Incorrect Logic ................................. 32
Figure 26 The original HLEFSM of the example ............................................. 33
Figure 27 The modified HLEFSM of the example ............................................ 33
Figure 28 The original HLEFSM where function F is called three times .................. 34
Figure 29 The modified HLEFSM when the erroneous function F is called three times .... 34
Acronyms and symbols

! (…) Negation
A ARPS tool analysis
ARPS Automated Reliability Prediction System
AT Test with the ARPS tool
ATM Automated Telling Machine
cp1 calcPos1
cp2 calcPos2
cp3 calcPos3
E Execution
EFSM Extended Finite State Machine
EI Error Index
Entity_m mth Entity
EP Extra Predicate
F False
f(…) A function
F_C Correct Function
FIC Function with Incorrect Logic
F_next Next Function
F_O Original Function
F_O_EI Original Function with Extra Input(s)
F_O_IAIN Original Function with Input(s) with Incorrect/Ambiguous Name
F_O_IAIT Original Function with Input(s) with Incorrect/Ambiguous Type
F_O_MI Original Function with Missing Input(s)
F_O_MV Original Function with Missing Variable(s)
FP Flat Parts
FP Complement of Flat Parts
F_prev Previous Function
H_0 Null Hypothesis
H_A Alternative Hypothesis
HLEFSM High Level Extended Finite State Machine
I Infection
IAP Incorrect/Ambiguous Predicate
<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LLEFSM</td>
<td>Low Level Extended Finite State Machine</td>
</tr>
<tr>
<td>M</td>
<td>Manual analysis in chapter 2. Mutant in chapter 3</td>
</tr>
<tr>
<td>md</td>
<td>Move Down</td>
</tr>
<tr>
<td>MF</td>
<td>Missing Instance of Function</td>
</tr>
<tr>
<td>MI</td>
<td>Missing Input</td>
</tr>
<tr>
<td>MP</td>
<td>Missing Predicate</td>
</tr>
<tr>
<td>MT</td>
<td>Manual Test</td>
</tr>
<tr>
<td>mu</td>
<td>Move Up</td>
</tr>
<tr>
<td>N/A</td>
<td>Not Available</td>
</tr>
<tr>
<td>NE</td>
<td>Total Number of Errors</td>
</tr>
<tr>
<td>Nodeᵢ</td>
<td>ith Node</td>
</tr>
<tr>
<td>O</td>
<td>Original</td>
</tr>
<tr>
<td>op</td>
<td>obtainPosition</td>
</tr>
<tr>
<td>OP</td>
<td>Operational Profile</td>
</tr>
<tr>
<td>P</td>
<td>Propagation in chapter 2. Original Program in chapter 3</td>
</tr>
<tr>
<td>p</td>
<td>p-value</td>
</tr>
<tr>
<td>Pathᵢ</td>
<td>&lt; Start, Entity₁, Entity₂, ... Entityₘ, Nodeᵢ &gt;</td>
</tr>
<tr>
<td>Pₖ</td>
<td>Correct Predicate</td>
</tr>
<tr>
<td>Pₖ̅</td>
<td>Complement of Pₖ</td>
</tr>
<tr>
<td>PIE</td>
<td>Propagation, Infection, Execution</td>
</tr>
<tr>
<td>Po</td>
<td>Original Predicate</td>
</tr>
<tr>
<td>Po̅</td>
<td>Complement of Po</td>
</tr>
<tr>
<td>Pₒw</td>
<td>Incorrect Portion of the Original Predicate Pₒ</td>
</tr>
<tr>
<td>Prob()</td>
<td>Probability</td>
</tr>
<tr>
<td>Ran()</td>
<td>Range</td>
</tr>
<tr>
<td>Re</td>
<td>Reliability</td>
</tr>
<tr>
<td>RePS</td>
<td>Reliability Prediction System</td>
</tr>
<tr>
<td>SCS</td>
<td>Simple Control System</td>
</tr>
<tr>
<td>SDD</td>
<td>Software Design Documents</td>
</tr>
<tr>
<td>SRS</td>
<td>Software Requirement Specification</td>
</tr>
<tr>
<td>T</td>
<td>True</td>
</tr>
<tr>
<td>Ti</td>
<td>Test i, i=1,4</td>
</tr>
<tr>
<td>Xₘₐₓ</td>
<td>Upper bounds</td>
</tr>
<tr>
<td>Xₘᵢₙ</td>
<td>Lower bounds</td>
</tr>
<tr>
<td>Ω</td>
<td>Universal set</td>
</tr>
<tr>
<td>Ø</td>
<td>Empty set</td>
</tr>
</tbody>
</table>
Chapter 1: Introduction

A software fault is a structural imperfection in a software system that may lead to the system eventually failing [1]. Software faults have been widely studied in academia, especially in the software reliability [2, 3] and software testing communities [4, 5]. In this research, we make use of software faults in two different ways to improve the software quality: (1) to utilize the software fault information to predicate software reliability, and (2) to utilize seeded faults to automatically derive test suites, i.e., software mutation testing.

Software reliability is defined as the probability that the software-based digital system will successfully perform its intended safety function (for all conditions under which it is expected to respond) upon demand with no unintended functions that might affect system safety [2, 8, 9]. It can be assessed based on the software failure probability which is closely related to software faults [2, 3]. In this research, we developed the Extended Finite State Machine-based Reliability Prediction System (EFSM-based RePS) method to assess software reliability [6, 7]. One of the most important aspects of this method is to model the relationship between software faults and software failure probability, for which we adopted the PIE theory [10] and extended it to different stages of the software development life cycle.

The EFSM-based RePS methodology utilizes the faults uncovered from the N-1’th version of the software to predict the software reliability of the N’th version. First the documentation of the software at different stages of the software development life cycle, which includes Software Requirement Specifications (SRS), Software Design Documents (SDD) and the source code, is collected. This information is then analyzed and used to construct a hierarchical model of the
software. The model is based on Extended Finite State Machines [11]. The level of abstraction in these three types of documents is that SRS level is higher than the SDD level which is in turn higher than the code level. Thus the SRS level model is at the highest level of abstraction, the SDD level model is at a medium level of abstraction, and the code level model is at the lowest level of abstraction. The software logic represented in the SRS level can be further refined at the SDD level and the code level to assist the modeling process. Once the software model is built, the uncovered faults are mapped to the model based on predefined defect templates. Then the Operating Profile (OP) of the software is used to assess the software reliability.

The EFSM-based RePS methodology is implemented into the Automated Reliability Prediction System (ARPS) tool. The tool can assist the analyst in conducting the software reliability analysis as well as reduce the modeling effort significantly. Human subjects were recruited, trained and their performance tested to evaluate the usability of the tool and it was found that the tool helped the subjects avoid errors and also reduce the error criticality.

Software mutation testing [34-37] is an automatic software testing technique based on seeded software faults. In mutation testing, the tester first defines a set of rules (i.e., mutant operators) to systematically seed faults into the source code. Each fault results in a new version of the software which is called a mutant. Since the mutants contain seeded faults, it is expected that their behavior differs from the original program when executing with certain test cases. Therefore, test cases should be developed to distinguish the mutants from the original program (i.e., to kill the mutants). Since these test cases can find the seeded faults, they are also helpful to find the indigenous ones.

Mutation testing can be divided into two categories: weak mutation and strong mutation. Weak mutation only requires that the test cases distinguish the mutants from the original program right after the mutated line, while strong mutation further requires that the difference can be preserved
and propagated to the exit of the program. Test cases for strong mutation are superior to those for weak mutation in terms of finding indigenous faults. However, previous research shows that strong mutation is a very difficult topic. In this research we introduce heuristics to improve the test cases such that more of them can strongly kill mutants. These heuristics are verified through application to sample programs. It is found that the heuristics help propagate the data state difference further than weak mutation, which will eventually lead to test cases that can strongly kill mutants.
Chapter 2: The Extended Finite State Machine Based Reliability Prediction System (EFSM-based RePS)

Disclaimer: Most of this chapter is the verbatim repetition of two documents: 1) a research report prepared by the author to the project sponsor, US Nuclear Regulatory Commission and 2) a journal paper titled “An Automated Software Reliability Prediction System for Safety Critical Software”.

2.1 Introduction to the EFSM-based RePS Methodology

2.1.1 General Concepts of Software Reliability and the Reliability Prediction System

Software reliability is one of the most important characteristics of software quality. It is concerned with the degree to which the software meets the requirements of the customer. In this study, software reliability is defined as the probability that the software-based digital system will successfully perform its intended safety function (for all conditions under which it is expected to respond) upon demand with no unintended functions that might affect system safety [1, 2, 9, 12].

In previous research, we introduced the Reliability Prediction System (RePS), which pairs subsets of software measures with associated models to predict software reliability [9, 12], as shown in Figure 1. From this research it was found that use of well defined measures and corresponding models leads to appropriate reliability prediction. However, there still remains uncertainty in the quality of the prediction which stems from possible variations in RePS model construction. The Extended Finite State Machine (EFSM) model is one of the possible RePS models selected for
fault propagation analysis. However, as discussed above, the construction of an EFSM is time consuming and error-prone. The ARPS project is directed towards:

1. Improving the speed at which an EFSM can be constructed and the repeatability of the construction process.
2. Reducing the uncertainty associated to the top measures. Two approaches are conducive to increasing speed and repeatability. One is training and the other one is automation. The ARPS project focuses on the increased use of automation in the modeling process.

![Figure 1 RePS constitution](image)

2.1.2 How Do Software Failures Occur? The Execution, Infection and Propagation Constraints

Suppose defects (or faults) exist in a software application. Triggering the software defects may result in software failure. In this research we adopt the concepts of Execution, Infection and Propagation originally discussed in [10] to model the process by which the existence of software faults will result in software failures. A software defect will be triggered when the appropriate external input conditions are met, i.e., the inputs should direct the execution towards the location
of the defect. In this study, this constraint on the input is called Execution (E). For the following programs (see Figure 2, the original defective version is on the left-hand side of the figure and the correct version is on the right-hand side of the figure), Execution requires that the defective line #5 be executed. Figure 3 displays the control flow of the program. It can be seen that the constraint for executing line #5 is b > 0.0 being true.

Figure 2 A sample program for explaining the Execution, Infection and Propagation constraints

```c
//Original code, which contains an error
1: double fun (double x) {
2:   double a=5.0, c, output=0.0;
3:   double b = x + 0.25;
4:     if (b > 0.0) {
5:       c = a * b;  // Error!
6:     }
7:     if (c > 6.0) {
8:       output = 3.0 * c;
9:     }
10:   return output;
11: }
```

```c
//Correct code, sometimes we do not have it
1: double fun (double x) {
2:   double a=5.0, c, output=0.0;
3:   double b = x + 0.25;
4:     if (b > 0.0) {
5:       c = a + b;  // Right!
6:     }
7:     if (c > 6.0) {
8:       output = 3.0 * c;
9:     }
10:   return output;
11: }
```

Figure 3 The control flow of the sample program
However, reaching the defective line does not guarantee that an “observable” effect will be produced. For instance, in the above code in Figure 2, if input \( x \) is equal to 1.0, then \( a*b \) is equal to \( a+b \) which is equal to 6.25. Therefore, there exists another constraint on the input. This constraint, if satisfied by the input, causes an “observable” difference between the original and the correct program state in existence after execution of the defective line. In this study, this constraint on the input is called Infection (I).

There exists a third constraint that allows a triggered software defect and its infected program state to finally propagate to the output of the software. Consider the same program in Figure 2. Let the input variable \( x \) equal to 0.0. It can be verified that variable \( c \) of the original (erroneous) version of the program is equal to 0.0, while \( c \) is equal to 5.25 in the correct version. The input can reach the defective line of code and cause a visible difference between the original version and the correct version of the program state succeeding execution of the defective line. Constraints E and I are satisfied. However, since 0.0 and 5.25 are less than 6.0, the “if” statement of line #7 is evaluated to false, and hence the variable \( output \) remains equal to 0.0. In other words, no software failures are externally observable. However, if the input \( x \) is chosen to be 0.85, variable \( c \) evaluates to 5.5 in the original version and 6.1 in the correct version. The variable \( output \) evaluates to 0.0 in the original version and 18.3 in the correct version. A software failure occurs in this case. In this study we term this constraint Propagation (P).

Table 1 lists the three constraints needing to be satisfied for depict how a software defect to cause a failure.
Table 1 The Execution, Infection and Propagation constraints

<table>
<thead>
<tr>
<th>Constraint</th>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution</td>
<td>E</td>
<td>The input needs to reach the defective line of code.</td>
</tr>
<tr>
<td>Infection</td>
<td>I</td>
<td>The state of the software after execution of the defective line should differ from the state</td>
</tr>
<tr>
<td></td>
<td></td>
<td>if the line were correct.</td>
</tr>
<tr>
<td>Propagation</td>
<td>P</td>
<td>The difference in state needs to propagate to the exit of the program.</td>
</tr>
</tbody>
</table>

These three constraints must all be satisfied by a given input for a software failure to occur. This means that the software can tolerate certain defects. When modeling software reliability, these three constraints are associated with a probability value. Determining these probabilities is called the Execution, Infection and Propagation analysis, or PIE analysis for short [6, 10]. Software reliability is calculated using the following equations.

\[
Prob(failure) = \sum_{i=1}^{N} E_i \times I_i \times P_i \\
Re = 1 - Prob(failure)
\]

where \(N\) is the total number of defects and \(E_i, I_i\) and \(P_i\) are the Execution, Infection and Propagation probabilities for the \(i_{th}\) defect.

The main assumptions necessary for the above equations to hold are that the total number of defects in the software should be small and the defects distributed sparsely [6, 10, 12]. These assumptions are reasonable for safety critical software which has undergone thorough software
testing, software verification and validation processes. Later sections will discuss equations 1 and 2 in more details.

2.1.3 The EFSM-based RePS Methodology

The aim of the EFSM-based RePS methodology is to utilize the software development documentation and corresponding defect reports of the N-1 version of the software to predict the software reliability of version N. The software development documents involved in this methodology are (1) the software requirement specifications (SRS), (2) the software design documents (SDD) and (3) the source code. The SRS contains information at the highest level of abstraction. The SDD contains more detailed information than the SRS and hence is at one level of abstraction lower than the SRS. The source code is a further refinement of the SDD information and is at the lowest level of abstraction. The EFSM-based RePS methodology models the software at these three different levels of abstraction and hence is a hierarchical methodology.

The EFSM-based RePS methodology always starts from the SRS level. The SRS is used to construct a system representation of the software logic. In this study the Extended Finite State Machine (EFSM) [6, 7, 11] is adopted for this purpose, and the resulting system representation is called the original High Level EFSM (HLEFSM). The SRS used to generate the original HLEFSM contains defects, and hence the original HLEFSM contains the same defects. Next the SRS level defects are mapped to the original HLEFSM to generate the modified HLEFSM, where each defect and is corrected equivalent is explicitly mapped out using defect templates. The modified HLEFSM contains both the original model and the corrected model. Finally the Execution, Infection and Propagation analysis is conducted based on the modified HLEFSM and the Operational Profile (OP) of the software, and the results are inserted into equation (1) and (2) to assess software reliability. In certain instances the SRS level information is not detailed enough to model the defects. In this situation SDD level information is utilized to further describe these
defects. However, the general philosophy of the EFSM-based RePS methodology is that the analysis should be kept at the highest level of abstraction. This is because the higher the level of the abstraction, the lesser modeling and computation are required.

Figure 4 illustrates the overall layout of the EFSM-based RePS method. The upper branch corresponds to the SRS level analysis and the lower branch corresponds to the SDD level analysis. For the SRS level analysis, “Load SRS Info”, “Generate Original HLEFSM” are discussed in 2.3. “Mapping Defects to HLEFSM” and “Generate Modified HLEFSM” are discussed in 2.4. The SRS portion of “Infection Analysis” and “Reliability Assessment” are discussed in 2.5 and 2.1, respectively. For the SDD level analysis, “Load SDD Info” and “Generate Original LLEFSM” are discussed in 2.8.2. “Mapping Defects to LLEFSM” and “Generate Modified LLEFSM” are discussed in 2.8.3. The SDD portion of “Infection Analysis” is discussed in 2.8.4. The ARPS tool, which implements the entire EFSM-based RePS method is introduced in 2.9.
2.2 Related Work

The EFSM-based RePS methodology and the ARPS tool are designed to model software logic and assess software reliability. In this section various software reliability modeling techniques will be reviewed to provide a general background. References 3, 13 and 14 provide a summary of software reliability models. More specifically, reference 13 classifies all models into three categories based on (1) which phases of the software development life-cycle are investigated to collect necessary information, (2) the specific information utilized for the modeling process, and (3) whether the specific software structure is required for modeling or not. Commonly used modeling techniques for each category are reviewed as follows.

Software reliability prediction models can be classified as early-prediction and late-prediction models. For early-prediction models, reference 15 introduces a research study where software architectural models were investigated to develop a software component reliability prediction framework. The researchers tried to resolve the issue of uncertainties introduced with software components being developed, which is a major problem for the reliability prediction methods on the architectural level. For later-prediction models, the most important category is the software reliability growth models (SRGM) [16-18]. These models are considered as later-prediction because they are applied during the software testing phase, where software faults are located and removed dynamically. Most SRGMs assume that the probability of introducing new software faults during testing/debugging is smaller than that of fault removal. Therefore, as the testing progresses, software reliability improves. In this research, the EFSM-based RePS methodology cannot really be categorized as an early or late prediction method since SRS, SDD and code information can be used. If only SRS or SDD information is used by the EFSM-based RePS methodology to predict software reliability, it is considered as early-prediction; if the source code and the testing information are involved as well, it should be considered a late-prediction method.
Reference 19 describes a software development phase-based model for predicting software reliability. In this research the software reliability is evaluated using the fault statistics obtained during the review of SRS and SDD as well as information obtained during the coding phase.

The software reliability prediction models can also be classified as fault-based models, failure-based models (e.g., the SRGMs) and development information-based models. Reference 20 describes a typical fault-based model, where the capture-recapture theory originating from biostatistics was adopted for software engineering. In this research, \( N_s \) faults were seeded by an independent researcher. These faults were designed such that they were representative of real software faults. The modified software was then reviewed or tested by other researchers, who identified some of the seeded and real faults. One can estimate the number of real faults remaining in the software from the number of seeded and real faults that are uncovered. Our EFSM-based RePS methodology differs significantly from the existing ones because the software fault information at the SRS, SDD and code level is used. In addition, our methodology only uses the uncovered, real faults but not any seeded ones.

One of the most typical development information-based models is the Bayesian Belief Networks (BBN) [21, 22]. Graphical network representations for the probability relationships among uncertain events are widely used in the BBNs. Directed acyclic graphs in which random variables are represented as nodes and conditional dependencies are represented as edges are the most dominant choices. In reference 21 a random variable that links “number of latent faults” and “operational usage” is used to represent software reliability, which can be assessed based on the relationship and the associated probability values.

Many prediction models are only based on the number of software faults or failures. These models can be classified as black box models. Some of the prediction models need the structural information of the software, and thus are classified as architectural models. In reference 22, the
Researchers built the SRS level functional architecture and used it as the system representation. They evaluated the software failure probability based on the failure probability of each element. In this thesis, the architectural information of the software was also needed to generate the EFSM system representation. Furthermore, the local effect of each defect was illustrated by mapping the defect information onto the EFSM model. Therefore, the EFSM-based RePS methodology can be considered as falling into the category of architectural models for software reliability modeling.

Many research projects also developed tools to assist the prediction and assessment for software reliability. For example, tools implementing different SRGMs are discussed in reference 23-25. Reference 26 introduces an architecture-based software reliability modeling tool and explains how it could be used to support teaching. Reference 27 describes a tool implementing a continuous-time BBN based framework. However, our ARPS tool is very different from these existing ones because the reliability assessment method implemented in ARPS is based on the PIE theory [6, 7, 10], which is among the commonly used software reliability modeling techniques. In addition, ARPS represents the software logic using a hierarchical EFSM model. These two characteristics distinguish ARPS from existing software reliability tools.

### 2.3 Original High Level Extended Finite State Machine Model Construction

#### 2.3.1 Understanding the SRS

As mentioned in the previous section, the EFSM-based RePS methodology starts with constructing the original High Level Extended Finite State Machine (HLEFSM) model of the software. There are three steps in the HLEFSM construction:

1. Understanding the Software Requirement Specifications (SRS).
2. Extracting information about the functions and logic of the SRS.
3. Constructing the High-Level Extended Finite State Machine (HLEFSM) model.
A software requirements specification (SRS) is a complete description of the behavior of a software system to be developed. From the SRS the user extracts the high level definition of each function and the logic of the system works, both of which are used to construct the HLEFSM model. A typical SRS outline based on [28] is provided in Figure 5.

<table>
<thead>
<tr>
<th>Table of Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Introduction</td>
</tr>
<tr>
<td>1.1 Purpose</td>
</tr>
<tr>
<td>1.2 Scope</td>
</tr>
<tr>
<td>1.3 Definitions, acronyms, and abbreviations</td>
</tr>
<tr>
<td>1.4 References</td>
</tr>
<tr>
<td>1.5 Overview</td>
</tr>
<tr>
<td>2. Overall description</td>
</tr>
<tr>
<td>2.1 Product perspective</td>
</tr>
<tr>
<td>2.2 Product functions</td>
</tr>
<tr>
<td>2.3 User characteristics</td>
</tr>
<tr>
<td>2.4 Constraints</td>
</tr>
<tr>
<td>2.5 Assumptions and dependencies</td>
</tr>
<tr>
<td>3. Specific requirements</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

**Figure 5** A typical SRS outline based on the IEEE 830-1998 standard

In this study many of the entries are ignored or simplified. Section 1 and 3 will be the focus of the analysis. The sections are briefly described below:

- **Section 1:** The introduction provides a general description and the logic of the software system to be built.
- **Section 3:** Specific requirements provide information on function at the SRS level.

**Figure 6** displays a possible template for section 3 organized by mode.
In this study, a snippet of functional requirement extracted from section 3 will contain the following information:

1. Function name.

2. Input variables: an input variable is a tuple with four elements:
   a. Input name: the name of the input
   b. Type: can be integer, decimal, string or Boolean
   c. Cross application boundary: Boolean variable indicating if an input is directly provided by the user, or any other external entity.
   d. Range: The range of this input

3. Output variables: an output variable is a tuple defined in a similar way as an input variable.
4. Variables: A variable is a tuple defined similarly to input and output variables. They are similar to the variables of programming languages. Some are global to the entire program, and some are local. In this study only variables important to the logic of the SRS are considered.

5. Function logic: This information describes what the function does and how it achieves its goals. However, if the function does not call other functions, the logic does not need to be described in the HLEFSM. This is because the HLEFSM model treats the functions as black boxes and tries to hide its information. This helps save resources during Execution, Infection and Propagation analysis.

The HLEFSM model is hierarchical. The highest level (i.e., level 0) is always the SRS itself, and is modeled as a function. The SRS function has its inputs, outputs, variables and logic. It calls other functions defined in the SRS document, and these functions are at level 1 (i.e., they are SRS level 1 functions). Figure 7 displays the hierarchical structure of the SRS.

![Diagram of hierarchical structure]

**Figure 7** The hierarchical structure of the SRS in this study
Level 1 SRS functions may call functions at level 2, and so forth. However, in this study, these level 2 functions are not modeled. When the SRS level 0 and level 1 information is not sufficiently detailed to model a defect, the defect is decomposed to the SDD level.

### 2.3.2 Extracting Information from the SRS

As discussed earlier in 2.3.1, two sections of the SRS are used in the ARPS methodology:

- **Introduction.** This section provides a general description and the logic of the software system (i.e., the SRS function).
- **Specific Functional Requirements.** The SRS level 1 functions are listed in this section. Their inputs, outputs and functionality are provided as well.

Extracting information from the SRS is reviewing the corresponding sections and recording all the important information. The user should start from the “Introduction” section and extract information related to inputs, outputs, internal variables and logic of the SRS level 0 function. Then he/she should turn to the “Specific Functional Requirements” section to extract information related to the SRS level 1 functions. All information relevant to a given function is documented in a corresponding function card, whose format will be introduced in 2.3.3.

### 2.3.3 Constructing the Original HLEFSM

There are three steps to constructing the original HLEFSM:

1. Develop all function cards based on the template.
2. Describe the program logic using the EFSM graphical language and draw the original HLEFSM model.
3. Iterate as necessary.
If the ARPS methodology is applied manually, all function cards and the original HLEFSM are hand drawn (or typed) into word documents. If the ARPS tool is used, all information is recorded into the tool and the HLEFSM is automatically generated using MATLAB [29].

2.3.3.1 Constructing the Function Cards

Figure 8 provides the template of the SRS level 0 function card. It can be seen that the function name, its level, all the inputs, outputs and internal variables (for level 0 only) are listed. This information can be found in the “Introduction” section of the SRS document. The logic information of the level 0 function is also provided in the “Introduction” section. It will be represented by the original HLEFSM which will be constructed after all function cards are developed.

![Function Card Template](image)

**Figure 8** Template for the SRS level 0 function card

The template for level 1 functions is similar to Figure 8. The function name, inputs and outputs of the level 1 functions are given in the “Specific Functional Requirement” section of the SRS document. Note that their internal variables and corresponding logic are not modeled at the SRS level. This information is only needed if a certain defect cannot be modeled at the SRS level but can be modeled at the SDD level.
The function card of a typical SRS function is shown in Figure 9 (adopted from the Simple Control System (SCS) case study in Appendix A). The naming of functions and variables is not strict as long as they are unique. The logic of the SRS is represented by the HLEFSM and is completed after other Level 1 function cards are finalized.

The Simple Control System (SCS) takes the pressure and temperature signal from sensors as inputs and controls four different valves: Valve #1 to Valve #4. The temperature ranges from 0 ºC to 100 ºC; the pressure ranges from $5 \times 10^5$ Pa to $5 \times 10^6$ Pa. There is a Boolean variable “Status” in the system which is used to control Valve #4. The system does not return any numerical output. The pressure and temperature are both decimal type.

### Function Name and level: SRS, level 0

#### Input(s):

1. **Input 1:**
   - Name: Pressure
   - Type: decimal
   - Application Boundary: Yes
   - Range: $5 \times 10^5$ Pa to $5 \times 10^6$ Pa

2. **Input 2:**
   - Name: Temperature
   - Type: decimal
   - Application Boundary: Yes
   - Range: 0 ºC to 100 ºC

#### Output(s): N/A

#### Variable(s):

1. **Variable 1:**
   - Name: Status
   - Type: Boolean
   - Application Boundary: No
   - Range: N/A

**Figure 9** The function card of the SRS level 0 function of the Simple Control System case study

The function card of a typical level 1 function is as follows (Figure 10 from the SCS case study). It can be seen that the format of the level 1 function card is the same as the level 0 function card except the fact that no information exists on internal variables.
2.3.3.2 Constructing the Original HLEFSM using the EFSM Graphical Language

There are three basic elements in the original HLEFSM:

1. The SRS level 0 function,
2. The state transitions and
3. The predicates.

The EFSM graphical language for the SRS level 0 function is a block that contains all level 1 state transitions and predicates. Figure 11 provides the SRS level 0 function without any level 1 functions or predicates. There is an arrow on either side of the block used to indicate the direction of the data flow. The function name and the level are indicated in the top left corner of the block.

Figure 10 The function card of the level 1 function initializeTheSystem

Figure 11 The SRS level 0 function represented using the EFSM graphical language

Function No.1: initialize the system.
2.1.1 Inputs:
   • Temperature. This is a double precision numerical value. It comes from the pressure sensor and hence crosses the application boundary. The range is 0 ºC to 100 ºC.
   • Pressure. This is a double precision numerical value. It comes from the pressure sensor and hence crosses the application boundary. The range is 5*10^5 Pa to 5*10^6 Pa.
2.1.2 Outputs: This function does not return a value but it affects the variable “Status”.
2.1.3 Functionality: initialize the system.

Function Name and level: initializeTheSystem, level 1
Input(s):
1. Input 1:
   Name: Temperature
   Type: decimal
   Application Boundary: Yes
   Range: 0 ºC to 100 ºC
2. input 2
   Name: Pressure
   Type: decimal
   Application Boundary: Yes
   Range: 5*10^5 Pa to 5*10^6 Pa
Output(s): This function does not return a value but it affects the variable “Status”.
The graphical representation of a state transition consists of three elements.

1. The state in which the function has not been initiated
2. The state in which the function is completed
3. The function that causes the state transit.

The template for the state transition is displayed in. This figure displays three state transitions. For the displayed state transition in the middle of the figure, the two states are displayed rounded rectangles and connected by an arrow. The name of the function that causes the state transition is indicated above the arrow. The state transition is bounded by a larger rectangle.

![EFSM graphical representation for a typical state transition](image)

**Figure 12** EFSM graphical representation for a typical state transition

**Figure 13** provides an example of how to apply the template. Assume a valve is being opened by a function called “openValve”. The state “function incomplete” is “valve is not opened”; the state “function complete” is “valve is opened”; the function name is “openValve”. Applying this information to the template in **Figure 12** the state transition depicted in **Figure 13** can be obtained.
Predicates in the EFSM graphical language are similar to the conditions of a programming language. If the predicate is evaluated as true, then certain transitions take place; if the predicate is false, then other transitions take place. More specifically, the following typical descriptions may be found in the SRS:

“If Temperature is greater than 50 °C, then open valve #1.”

Or:

“If Temperature is greater than 50 °C, then open valve #1; otherwise open valve #2”

The logic of “If Temperature is greater than 50 °C, then open valve #1.” is represented in Figure 14. The condition is indicated in the diamond symbol. Our convention is that the upper branch is the true branch and the lower branch is the false branch. The openValve1 function is represented as before. The lower branch of the predicate contains no state transition.
The logic “If Temperature is greater than 50 ºC, then open valve #1; otherwise open valve #2” is depicted in Figure 15:

![Diagram of EFSM](image)

**Figure 15** EFSM representation for “If Temperature is greater than 50 ºC, then open valve #1; otherwise open valve #2”

The graphical language also supports the “if-elseif-elseif…-else” structure. Assume the following sentence is presented in the SRS:

“If Condition #1 is true, then do function1; Else if Condition #2 is true, then do function2. Otherwise do function3”.

The corresponding HLEFSM is shown in Figure 16. This if-elseif-else structure is equivalent to the if-elseif-else structure found in procedural languages.
As is experienced in other programming languages, predicates and state transitions can combine and result in complex HLEFSM. Figure 17 displays a complex HLEFSM with multiple state transitions and nested predicates.
2.3.3.3 Constructing the Original HLEFSM using the SRS of the SCS Case Study

In this section the SCS case study will be used to demonstrate how the original HLEFSM is constructed. Let us assume that the function cards are already available. The logic of the system is as follows:

1. The system is initialized.
2. If the temperature is greater than or equal to 15 °C, Valve #1 shall open. Register R1 is set to 1.
3. If the pressure is greater than or equal to $10^6$ Pa, Valve #3 shall open. Register R3 is set to 1.
4. Valve #4 shall open last. Register R4 is set to 1.

The first line of the logic is “The system is initialized”. This means that the first part of the HLEFSM is a state transition. The transition function is “initializeTheSystem”. By applying the template introduced earlier, the following state transition can be obtained:

![Figure 18](image)

**Figure 18** The state transition corresponding to the first line of the logic for the SCS case study

The second line of the logic is “If the temperature is greater than or equal to 15 °C, Valve #1 shall open. Register R1 is set to 1”. Because of the existence of the word “if”, it is known that a predicate exists. If the predicate is true, the function “openValve1” is called. Otherwise nothing
happens. By applying the template introduced earlier, the predicate shown in Figure 19 is obtained.

![Figure 19](image)

**Figure 19** The predicate corresponding to the second line of the logic for the SCS case study

The third line of the logic is “If the pressure is greater than or equal to $10^6$ Pa, Valve #3 shall open. Register R3 is set to 1”. Similarly **Figure 20** is obtained:

![Figure 20](image)

**Figure 20** The predicate corresponding to the third line of the logic for the SCS case study

The last line of the logic is “Valve #4 shall open last. Register R4 is set to 1”. This indicates that the last part of the HLEFSM is another state transition as shown in **Figure 21**.

![Figure 21](image)
Figure 21 The state transition corresponding to the last line of the logic of the SCS case study

The last step is linking and inserting the four sub-models in the SRS function, where Figure 22 is obtained. This is the original version of the HLEFSM model for the SCS system. It can be seen that a function called “openValve2” should exist that is never used. This is a defect in the original SRS, and it will be mapped based on defect templates. This mapping process will be introduced in the next section.

Figure 22 The original HLEFSM for the SCS case study

2.4 SRS Defect Definitions and Templates

2.4.1 Common SRS Defects Identified in this Study

The EFSM-based RePS methodology requires defect reports of the previous version of the program to evaluate software reliability. SRS, SDD and code have their own defect reports, and
in this section only the SRS defect report is discussed. SRS defect reports are typically prepared by the software inspection team. They may consist of tables which list detailed information on each defect. For the SRS level analysis, all defects identified are mapped to the original HLEFSM based on defect templates. This results in a modified HLEFSM in which the user can clearly identify the local effect of each defect.

SRS defects can be classified in a taxonomy of 26 defects. Table 2 provides identified defects. Note that the same taxonomy is also applied to the SDD level analysis.

<table>
<thead>
<tr>
<th>Defect category</th>
<th>Defect name</th>
</tr>
</thead>
</table>
| Category 1: Defects for the definition of level 1 functions | 1. Missing (definition of) function: The entire definition of a function is missing from the SRS/SDD.  
2. Extra (definition of) function: The entire function definition is extraneous.  
3. Incorrect/ambiguous function name: The name of the function is incorrect/ambiguous.  
4. Function with incorrect logic: The functionality is valid but the logic is erroneous.  
5. Function with Incorrect functionality: The functionality is not valid. |
| Category 2: Defects related to inputs | 1. Missing input: The definition of an input is missing from the SRS/SDD.  
2. Extra input: The definition of an input is extraneous.  
3. Incorrect/ambiguous input name: The name of the input is incorrect or ambiguous when it is defined.  
4. Input with incorrect type: The type of the input is erroneously defined.  
5. Input with incorrect range: The range of the input is erroneously defined. |
| Category 3: Defects related to outputs | 1. Missing output: The definition of an output is missing from the SRS/SDD.  
2. Extra output: The definition of an output is extraneous.  
3. Incorrect/ambiguous output name: The name of the output is incorrect or ambiguous when it is defined.  
4. Output with incorrect type: The type of the output is erroneously defined.  
5. Output with incorrect range: The range of the output is erroneously defined. |
| Category 4: Defects related to internal variables | 1. Missing variable: The definition of a variable is missing from the SRS/SDD.  
2. Extra variable: The definition of a variable is extraneous.  
3. Incorrect/ambiguous variable name: The name of the variable is incorrect or ambiguous when it is defined.  
4. Variable with incorrect type: The type of the variable is erroneously defined.  
5. Variable with incorrect range: The range of the variable is erroneously defined. |

Continue...
In Table 2, Category 1 defects relate to the definition of the SRS level 1 functions. When this category of defects takes place, all instances of the involved function are affected. Category 2 defects are related to the inputs. These inputs can either be the inputs of the level 0 function itself or of the level 1 functions. These defects affect all instances of the defective input. Category 3 and 4 are similar to Category 2 since inputs, outputs and internal variables are essentially variables. Category 5 defects are about the logic of the level 0 function and thus have nothing to do with the level 1 functions.

The defect template for each type of defect has been developed. The main advantage of using templates is to reduce the likelihood of making mistakes. In this section of the report only the defect templates for the following four defects are discussed since these are the ones involved in the experimental validation in section 2.10.

1. Function with Incorrect Logic
2. Missing Instance of Function
3. Missing Predicate
4. Incorrect/Ambiguous Predicate

The template for the other defects can be found in Appendix B.
As discussed earlier, the HLEFSM already constructed is the “original HLEFSM”. Since it is based on the original SRS document, all defects are hidden within. The HLEFSM to be constructed in the following section is the “modified HLEFSM”, with all defects mapped out.

2.4.2 How a Mapped Defect is Represented in the Modified HLEFSM

2.4.2.1 The General Case of Mapping Defects

In the original HLEFSM, state transitions are linked to each other. Let us assume the SRS contains F1, F2 and F3. They execute consecutively and can be considered as three “nodes” in the logic flow. The “Function Incomplete” and “Function Complete” labels are ignored due to space limitations.

![Figure 23](image)

**Figure 23** A typical HLEFSM consisting of three consecutive state transitions

Let us also assume that the state transition F2 is erroneous. Then the middle portion of **Figure 23** will be replaced by a block where the defect is explicitly mapped (**Figure 24**). However, the number of nodes remains identical. The defect templates provide a base “Signature” for the middle block.
2.4.2.2 Defect Template for Function with Incorrect Logic

Next defect definitions and templates will be introduced one by one. The first defect is “Function with Incorrect Logic”. The defect characterizes cases where functionality of a level 1 function is valid but its logic is erroneous. Given the SRS format, such defects will appear in “section 2, Specific Functional Requirements”. As discussed before, the logic of a level 1 function is never explicitly represented at the SRS level. The point of introducing this defect is to indicate that the logic of a certain level 1 function, although not represented anywhere, is erroneous. Because there typically isn’t enough information to evaluate its effect on the entire SRS logic, this type of defect should be analyzed by decomposition to the SDD level, which will be discussed in 2.8.

A typical example for this type of defect is when: the description of the original version of a level 1 function, “openValve1”, in the SRS document is “Valve #1 is open 100%”. However, in the defect report, it is said that the correct description should be “Valve #1 is open 75%”. Because this function is level 1, its inner logic is not represented by the HLEFSM. The defect report indicates that the description in the original SRS document is incorrect. Hence a defect exists in the logic of this level 1 function, which must be flagged.

Sometimes the description of the functionality of the level 1 function is correct but its implementation is erroneous. Although the implementation of the level 1 functions will not be
investigated, this type of error does exist and needs to be handled. For example, the original SRS document may say that “Valve #1 is open 75%”, which is correct; however, in the defect report it is said “The description of the functionality is correct but the implementation is erroneous”. In this case the function openValve1 should also be flagged as erroneous. In either case, the defect template shown in Figure 25 is used to map the defect.

It can be seen that the middle portion of the HLEFSM is subdivided into two branches. The upper branch (the original) is incorrect and the lower branch (the modified) is correct. The subscript “O” of “F₀” stands for “original function” and “C” of “Fₖ” stands for “correct function”. The two parallel branches constitute the “Node with defect mapped” (Figure 24) which is bounded by dotted lines.

Let us assume that the following three functions correspond to the actions in the logic:

1. initializeTheSystem (correct)
2. openValve1 (Function with incorrect logic)
3. openValve2 (correct).

The logic is (1) First initialize the system, (2) then open valve #1, (3) after that open valve #2.

The original HLEFSM is as follows:

![Figure 26 The original HLEFSM of the example](image)

Applying the defect template we obtain the modified EFSM of **Figure 27**:

![Figure 27 The modified HLEFSM of the example](image)

It should be noted that because the defect is concerned with the definition of a function, it affects all instances of that function. Therefore, when this type of defect is mapped to the EFSM, the template should be applied to all instances of the function. For example, if a function “F” with incorrect logic is called three times in the HLEFSM shown in **Figure 28**, all three instances must be modified as shown in **Figure 29**. Note that the nodes with defects mapped are bounded by dotted lines.
Figure 28 The original HLEFSM where function F is called three times

Figure 29 The modified HLEFSM when the erroneous function F is called three times

2.4.2.3 Defect Template for Missing Instance of Function

As discussed earlier, “Missing Instance of Function”, “Missing Predicate” and “Incorrect/Ambiguous Predicate” are concerned with the logic of the SRS level 0 function. In other words, they exist in the SRS logic but are not present in the logic of the level 1 functions.

A Missing Instance of a Function is such that the level 1 function is defined correctly, but a call to that function has been omitted from the logic of the SRS level 0 function. For example, the SRS
logic in the original SRS document is “…Open Valve #1. Then open Valve #3…”. However, the correct description provided in the defect report is “…Open Valve #1. Open Valve #2. Then open Valve #3…”. Obviously the original SRS only calls openValve1 and openValve3 consecutively, while the correct logic in the defect report requires calling openValve1, openValve2 and openValve3. The call to openValve2 was omitted from the original SRS document.

The Location for this type of defect is in the description of the SRS logic. Based on the SRS format adopted for this research, the defect should appear in “section 1, Introduction”. Figure 30 below shows how the defect is represented.

![Defect template for Missing Instance of Function](image)

**Figure 30** Defect template for Missing Instance of Function

The middle portion of the HLEFSM is once again divided into two branches. The upper branch is the original, from which the function call was omitted; the lower branch is the correct version, where the missing function call F is mapped. The template can be applied to an example. Consider three functions corresponding to the logic:

1. openValve1 (correct)
2. openValve2 (correct, but an instance is missing from the logic)
3. openValve3 (correct).
The original logic is: (1) Open valve #1, (2) then open valve #3. The corresponding original HLEFSM is as follows:

![Figure 31 The original HLEFSM for an example of Missing Instance of a Function](image)

The correct logic is: (1) Open valve #1, (2) open valve #2, (3) and then open valve #3. Thus the modified HLEFSM is as follows:

![Figure 32 The modified HLEFSM for an example of Missing Instance of a Function](image)

### 2.4.2.4 Defect Template for Missing Predicate

The definition of this defect is that a predicate of the level 0 function is missing. A predicate, as discussed before, is similar to a condition in a programming language. This defect corresponds to the situation where a set of statements (e.g., function calls or other decisions) should execute if a certain predicate “P” is true. However, the determination of the value of the predicate “P” is missing. This causes the statements to always execute, which is erroneous.
For example, the logic of the original version of the SRS is “open valve #1, open valve #2. Then open valve #3”. However, the defect report indicates that the correct logic should be “Open valve #1. If temperature is greater than 10 °C, open valve #2. Then open valve #3”. Following the original (incorrect) logic the valve #2 is always open, while it should depend on the temperature. This causes the state transition to occur under the wrong set of conditions. This defect is located in the section of the SRS devoted to the description of the SRS logic. Based on the SRS document structure, it appears in “section 1, Introduction”. The template for this defect is as follows:

![Figure 33 Defect template for Missing Predicate](image)

In **Figure 32** there are again two branches in the middle portion of the HLEFSM. The function F always executes in the upper branch (the original). However, in the lower branch (the correct) F will execute only if the missing predicate P is true. Similarly to the previous two defects, the logic flow traverses both branches in parallel. However, for the lower branch, the logic flow has to pass the predicate P. In 2.6 it will be seen that the operational profile (OP) needs to be mapped to the lower branch to carry out the Infection analysis of this defect.

An example is provided below. Let us assume the following logic:
1. openValve1
2. openValve2
3. openValve3

The logic of the original version of the SRS is (1) Open valve #1 (2) open valve #2 (3) then open valve #3. The original HLEFSM is as follows:

![Original HLEFSM](image)

**Figure 34** The original HLEFSM of the example for Missing Predicate

The defect report indicates that the correct logic should be (1) Open valve #1 (2) if temperature is greater than 10 °C, open valve #2, (3) then open valve #3. The corresponding modified HLEFSM is shown in **Figure 35**.

![Modified HLEFSM](image)

**Figure 35** The modified HLEFSM for an example of Missing Predicate
2.4.2.5 Defect Template for Incorrect/Ambiguous Predicate

For this defect the logical expression of a predicate is incorrect or ambiguous. Let us assume that a set of statements should execute if a predicate P is true. However, the predicate is erroneously defined. This causes the set of statements to execute under the wrong conditions. An example is as follows. The logic of the original SRS is “Open valve #1. If temperature is greater than 15 °C, open valve #2. Then open valve #3”. However, the correct logic as provided in the defect report is “Open valve #1. If temperature is greater than 25 °C, open valve #2. Then open valve #3”. Obviously the valve #2 sometimes is opened incorrectly as per the original SRS. The template for this defect is provided below:

![Diagram of Defect Template](image)

**Figure 36** Defect template for an Incorrect/Ambiguous Predicate

It can be seen that there are again two parallel branches in this case. The original predicate $P_O$ is in the upper branch (bounded by red lines) and the correct predicate $P_C$ is in the lower branch (bounded by green lines). The following example illustrates how the template should be applied. The logic of this example is provided below:
1. openValve1
2. openValve2
3. openValve3

The original logic is as follows: (1) Open valve #1, (2) If temperature is greater than 15 °C, open valve #2, (3) then open valve #3. The original HLEFSM is as follows.

![Figure 37](image1.png)

**Figure 37** The original HLEFSM of the example of Incorrect/Ambiguous Predicate

The correct logic defined in the defect report is: (1) Open valve #1, (2) If temperature is greater than 25 °C, open valve #2, (3) then open valve #3”. The modified EFSM is provided in **Figure 38**.

![Figure 38](image2.png)

**Figure 38** The modified HLEFSM of the example of Incorrect/Ambiguous Predicate
2.4.3 Application of the Defects Templates to the SCS Case Study

Let us assume that the user has already constructed the original HLEFSM. The next task is to read the defect report thoroughly and apply the templates previously introduced. Mapping the defects to the original HLEFSM is applying the defect templates to the original HLEFSM. The specific defects used in this section are provided in Table 3.

### Table 3 SCS’s Defect Report

<table>
<thead>
<tr>
<th>Defect No.</th>
<th>Location</th>
<th>Defect Type</th>
<th>Original Description</th>
<th>Correct Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Section 1 2)</td>
<td>Incorrect/Ambiguous Predicate</td>
<td>“If the temperature is greater than or equal to 15 °C, Valve #1 shall open.”</td>
<td>If the temperature is greater than or equal to 45 °C, Valve #1 shall open.</td>
</tr>
<tr>
<td>2</td>
<td>Section 1</td>
<td>Missing Instance of Function</td>
<td>N/A</td>
<td>Valve #2 shall open after Section 1, 2).</td>
</tr>
<tr>
<td>3</td>
<td>Section 2.4.3</td>
<td>Function with Incorrect Logic</td>
<td>“Functionality: open valve #3 to 100%.”</td>
<td>Valve #3 shall be opened based on the temperature.</td>
</tr>
<tr>
<td>4</td>
<td>Section 1 4)</td>
<td>Missing Predicate</td>
<td>“Valve #4 shall open at last.”</td>
<td>If the Boolean variable Status is true, open Valve #4.</td>
</tr>
</tbody>
</table>

### 2.4.3.1 Mapping the Defect “Incorrect/Ambiguous Predicate”

Defect #1 is of type “Incorrect/Ambiguous Predicate”. Based on the defect report shown in Table 3, the defective area is circled in the original HLEFSM (Figure 39).
Figure 39 The original HLEFSM of the SCS case study with the location of Defect #1 pointed out.

Applying the defect template in Figure 36 the local modified HLEFSM is obtained. Note that T is short for Temperature.

Figure 40. The local modified HLEFSM corresponding to Defect #1

2.4.3.2 Mapping the Defect “Missing Instance of Function”

Defect #2 in Table 3 belongs to this category. Because the function call is missing from the original HLEFSM, there is no “Original Description” in the defect report. The defective area is circled in the original HLEFSM in Figure 41.
Figure 41 The original HLEFSM for the SCS case study with the pinpointed location of Defect #2

Applying the defect template for “Missing Instance of Function” to the circled area one obtains the local modified HLEFSM in Figure 42. Note that “T” is short for Temperature and “P” is short for Pressure.

Figure 42 The local modified HLEFSM corresponding to Defect #2

2.4.3.3 Mapping the Defect “Function with Incorrect Logic”

Defect #3 is “Function with Incorrect Logic”. The original description of the logic of the function is erroneous. When manually applying the ARPS methodology, the function card should be
modified to indicate the defect, as shown in Figure 43. Then the defect template should be applied to all instances of the defective function.

![Figure 43](image)

**Figure 43** Modification of the function card of a function affected by the defect “Function with Incorrect Logic”

**Figure 44** provides the original HLEFSM for the SCS case study with the location of Defect #3 circled out.

![Figure 44](image)

**Figure 44** The original HLEFSM for the SCS case study with the pinpointed location of Defect #3

Applying the defect template we obtain the local modified HLEFSM, as provided in **Figure 45**.

44
2.4.3.4 Mapping the Defect “Missing Predicate”

The last defect is a “Missing Predicate”. The defective area is circled in **Figure 46**.

Applying the defect template, one obtains the following local modified HLEFSM.
2.4.3.5 Construct the Entire Modified HLEFSM

The entire modified HLEFSM is constructed by linking the correct portions of the original HLEFSM with the locally modified HLEFSM, as shown in Figure 48. The data flow is from left to right (i.e., the case of Figure 48) and the convention followed is for the upper branch to depict the defective node, (i.e., the original (incorrect) version of the software) and the lower branch to depict the modified (correct) version of the software.

Figure 47 The local modified HLEFSM corresponding to Defect #4
2.5 Execution Analysis

2.5.1 What is an Operational Profile

As discussed in the previous sections, the Operational Profile (OP) [2] is used for Execution and Infection analysis. The OP describes the set of probabilities that defines the behavior of the system thereby reflecting the way a system behaves in the real world. For example, the OP of an ATM in a bank may be as follows.

Table 4 Sample OP of an ATM in a bank

<table>
<thead>
<tr>
<th>No.</th>
<th>Description of the Event</th>
<th>Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Check account balance</td>
<td>15 %</td>
</tr>
<tr>
<td>2</td>
<td>Withdraw 20 $</td>
<td>8 %</td>
</tr>
</tbody>
</table>
Table 4 Sample OP of an ATM in a bank (continue)

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>Withdraw 40 $</td>
<td>10 %</td>
</tr>
<tr>
<td>4</td>
<td>Withdraw 60 $</td>
<td>7 %</td>
</tr>
<tr>
<td>5</td>
<td>Withdraw 80 $</td>
<td>11 %</td>
</tr>
<tr>
<td>6</td>
<td>Withdraw 100 $</td>
<td>15 %</td>
</tr>
<tr>
<td>7</td>
<td>Deposit money</td>
<td>34 %</td>
</tr>
</tbody>
</table>

It can be seen that the probabilities in Table 4 sum up to 100%. This indicates that all possible behaviors of the ATM system are described by such profile.

2.5.2 An Example Illustrating How Execution Analysis is Conducted

The OP is used during Execution Analysis when predicates are involved. As introduced earlier, Execution describes the probability that an input reaches a defect. If there are no predicates in the SRS logic, the probability of reaching a statement (function calls, assignments or if statements, etc.) is always 100%. However, if predicates exist, the probability of reaching a particular statement might be less than 100%. First consider the following HLEFSM in Figure 49:

![Figure 49](image_url)

**Figure 49** A typical HLEFSM used for Execution analysis (T stands for “True” and F stands for “False”)
For this HLEFSM one can derived the following equations:

1. \( P(fun_1) = \text{Prob}(P_1==\text{true}) \)
2. \( P(fun_2) = \text{Prob}(P_1==\text{false}) \)
3. \( P(fun_3) = \text{Prob}(P_1==\text{true} \text{ AND } P_2==\text{true}) \)
4. \( P(fun_4) = \text{Prob}(P_1==\text{true} \text{ AND } P_2==\text{false}) \)

In the above equations, \( P(fun_i) \) is the probability that a function “fun\(_i\)” is reached. For \( P(fun_1) \) this probability is equal to the probability that \( P_1 \) equals to true. Here \( P_1 \) is a predicate, and \( P_1==\text{true} \) or \( P_1==\text{false} \) can be considered as events. If occurrence data is made available, \( P(fun_1) \) can be calculated easily. The OP (see for example Table 4) provides such data.

Let us assume for now that all variables in the predicates are statistically independent. This assumption is also valid for all predicates involved in the SCS case study and the various case studies and exams used in the experimental validation discussed in 2.10. Note that this assumption is introduced for convenience only. It will be seen that even if the variables are not independent, a symbolic solver such as MATLAB [11] or Mathematica [12] can be used to calculate the interval for which the predicates hold. For the example in Figure 49, if the OP data for the two predicates is given in Table 5, the truth values of the predicates evaluated as:

1. \( P(fun_1) = \text{Prob}(P_1==\text{true}) = 37\% \)
2. \( P(fun_2) = \text{Prob}(P_1==\text{false}) = 1-37\% = 63\% \)

We can also calculate that

1. \( P(fun_3) = \text{Prob}(P_1==\text{true} \text{ AND } P_2==\text{true}) = \text{Prob}(P_1==\text{true})*\text{Prob}(P_2==\text{true}) = 37\%*26\% = 9.6\% \)
2. \( P(fun_4) = \text{Prob}(P_1==\text{true} \text{ AND } P_2==\text{false}) = \text{Prob}(P_1==\text{true})*\text{Prob}(P_2==\text{false}) = 37\%*(1-26\%) = 27.4\% \)
### Table 5 The OP information for the example in Figure 48

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Prob(Pi == true)</th>
</tr>
</thead>
<tbody>
<tr>
<td>P₁</td>
<td>37%</td>
</tr>
<tr>
<td>P₂</td>
<td>26%</td>
</tr>
</tbody>
</table>

### 2.5.3 Methodology for Execution Analysis

#### 2.5.3.1 Introduction to the Methodology

When evaluate the probability of a software failure, Execution analysis is only conducted on the modified HLEFSM. This is because only the Execution probability of nodes with mapped defects (or defective nodes) contributes to software failure probability. Correct functions and predicates do not contribute to software failure. The methodology follows the principles described in the above example, but is generalized to consider more complex HLEFSM. The generalized methodology is as follows:

1. Identify all the paths from the start point to the $i_{th}$ defective node ($Node_i$). More specifically, the $j_{th}$ path to $Node_i$ can be expressed as:

   $$Path^i_j = \langle Start, Entity_1, Entity_2, ..., Entity_m, Node_i \rangle$$

   Where “Start” denotes the start point; $Entity_m$ denotes the $m_{th}$ entity on the $Path^i_j$. An entity can be either a function call (i.e., a state transition), a predicate or a defective node. $Node_i$ denotes the $i_{th}$ defective node, which is the one of interest.

2. Identify all the predicates on a path to reach $Node_i$ and simplify the path by removing all other nodes. The $j_{th}$ simplified path has the following form:

   $$Path^i_j = \langle Start, P_1 == true/false, P_2 == true/false, ..., P_k = true/false, Node_i \rangle$$
Where $P_k$ denotes the $k$th predicates on this path. Other nodes were removed because they will not affect the Execution probability of Node $i$.

3. Calculate the Execution probability of Node $i$ using the following formula.

$$E(\text{Node}_i) = \sum_{j=1}^{n} \text{Prob}(\text{Path}_j)$$

Where $n$ is the number of paths from the “start” point to Node $i$. The probability of $\text{Path}_j$ depends on the probability data of each predicate on the path. The simplified path from step 2 is as follows:

$\text{Path}_j = \langle \text{Start}, P_1 = \text{true/false}, P_2 = \text{true/false}, ... P_k = \text{true/false}, \text{Node}_i \rangle$

The probability of $\text{Path}_j$ can be calculated as follows:

$$\text{Prob}(\text{Path}_j) = \text{Prob}(P_1 = \text{true/false} \text{ AND } P_2 = \text{true/false AND } ... P_k =$$

$$= \text{true/false})$$

$$= \text{Prob}(P_1 = \text{true/false}) \times \text{Prob}(P_2 = \text{true/false}) \times \text{Prob}(P_k = \text{true/false})$$

In the above formula, all the data of $\text{Prob}(P=\text{true/false})$ is extracted from the OP.

The various concepts discussed above are summarized below:

1. In a modified HLEFSM there may exist multiple defective nodes, and the $i$th defective node $\text{Node}_i$ is of interest.

2. There may exist multiple paths starting from the start point “Start” to $\text{Node}_i$. The total number of such paths is $n$.

3. Our analysis first focuses on the $j$th path ($\text{Path}_j$) and then generalizes to all the paths.

4. For the $j$th path, there may exist up to $m$ entities. Each entity can be a function call, predicate or defective node.
5. After simplification the $j_{th}$ path may consist of up to $k$ predicates whose probabilities can be obtained or derived from the OP.

2.5.3.2 Identify All the Paths from the Start Point to the Defective Node

In this section the methodology discussed in 2.5.3.1 is applied to six examples to demonstrate how Execution analysis should be conducted. Step 1, which is identifying all paths from starting point to the $i_{th}$ defective node of a modified HLEFSM, which can be elaborated as follows:

1. Start from Node, and reverse the logic flow from Node, back to the starting point “Start”.
2. Record all the entities (i.e., function calls, predicates and defective nodes) on the path.
3. Then rewrite each path starting from the “Start” to Node.

Examples 1 to 6 are six typical cases found during Execution analysis. When the ARPS methodology is applied manually, the user needs to start from the defective node and trace back to the entry node of the HLEFSM. For large HLEFSM this process is tedious and error-prone. However, when the ARPS tool is used, all calculation will be done automatically.

Example 1 in Figure 50 describes a typical situation encountered during an Execution Analysis.

![Figure 50 Example 1](image)

In this example the defective node is Node. Obviously there is only one path between the starting point and Node, which is:

$$Path_{1} = < Start, fun_{1}, fun_{2}, Node_{1} >$$
It should be noted that Node\(_1\) is a defective node. The defect is of one of the types defined in Table 2. Therefore, the detailed morphology of this node follows the defect templates introduced in section 2.4.1, i.e. two parallel branches (the original branch and the correct branch).

Example 2 is shown in Figure 51. Predicate \(P_1\) divides in the logic flow. There is only one path to Node\(_1\), which is:

\[ Path_{1}^1 = \langle \text{Start, fun}_1, P_1 == \text{true}, \text{Node}_1 \rangle \]

Predicate \(P_1\) splits the logic flow, and the path requires that \(P_1==\text{true}\). This will reduce the Execution probability below 100%.

Example 3 is provided in Figure 52. There is still only one path to Node\(_1\), which is:

\[ Path_{1}^1 = \langle \text{Start, fun}_1, P_1 == \text{true}, \text{Node}_1 \rangle \]
Example 4 is given in Figure 53. Two paths to Node\(_1\) exist case which are caused by the predicate \(P_1\).

\[
Path_1^1 = \langle \text{Start, } \text{fun}_1, P_1 = true, \text{fun}_2, \text{Node}_1 \rangle
\]

\[
Path_2^1 = \langle \text{Start, } \text{fun}_1, P_1 = true, \text{fun}_3, \text{Node}_1 \rangle
\]

Figure 53 Example 4

Example 5 is provided in Figure 54. Three paths exist to Node\(_1\).

\[
Path_1^2 = \langle \text{Start, } P_1 = true, \text{fun}_1, \text{Node}_1 \rangle
\]

\[
Path_2^2 = \langle \text{Start, } P_1 = false, \text{fun}_2, P_2 = true, \text{fun}_3, \text{Node}_1 \rangle
\]

\[
Path_3^2 = \langle \text{Start, } P_1 = false, \text{fun}_2, P_2 = false, \text{fun}_4, \text{Node}_1 \rangle
\]
Example 6 is given in Figure 55. The logic flow for this example differs somewhat from previous examples because two branches enter Node$_1$. However, this does not cause any fundamental differences. Only two paths to Node$_1$ exist.

\[ Path_1^1 = \langle \text{Start, } P_1 = \text{true, } P_2 = \text{false, Node}_1 \rangle \]

\[ Path_2^1 = \langle \text{Start, } P_1 = \text{false, } P_3 = \text{true, Node}_1 \rangle \]
2.5.3.3 Identify All the Predicates within a Path and Simplify the Path

The second step is to identify all the predicates within a path and simplify the path expression. This step is fairly trivial once all the entities of a path have been identified. One needs only keep all the predicates, remove all the function calls and remove all the defective nodes. In the following paragraphs simplified paths for Example 1 to 6 are provided.

Example 1:

$$Path_1^1 = \langle \text{Start, fun}_1, \text{fun}_2, \text{Node}_1 \rangle$$

$$\rightarrow Path_1^1 = \langle \text{Start, Node}_1 \rangle$$

Example 2:

$$Path_1^1 = \langle \text{Start, fun}_1, P_1 == \text{true}, \text{Node}_1 \rangle$$

$$\rightarrow Path_1^1 = \langle \text{Start, P}_1 == \text{true}, \text{Node}_1 \rangle$$

Example 3:

$$Path_1^1 = \langle \text{Start, fun}_1, P_1 == \text{true}, P_2 == \text{false}, \text{Node}_1 \rangle$$

$$\rightarrow Path_1^1 = \langle \text{Start, P}_1 == \text{true}, P_2 == \text{false}, \text{Node}_1 \rangle$$

Example 4:

$$Path_1^1 = \langle \text{Start, fun}_1, P_1 == \text{true}, \text{fun}_2, \text{Node}_1 \rangle$$

$$\rightarrow Path_1^1 = \langle \text{Start, Node}_1 \rangle$$

$$Path_2^1 = \langle \text{Start, fun}_1, P_1 == \text{false}, \text{fun}_3, \text{Node}_1 \rangle$$

$$\rightarrow Path_2^1 = \langle \text{Start, Node}_1 \rangle$$

Example 5:

$$Path_1^1 = \langle \text{Start, P}_1 == \text{true}, \text{fun}_1, \text{Node}_1 \rangle$$

$$\rightarrow Path_1^1 = \langle \text{Start, P}_1 == \text{true}, \text{Node}_1 \rangle$$

$$Path_2^1 = \langle \text{Start, P}_1 == \text{false}, \text{fun}_2, P_2 == \text{true}, \text{fun}_3, \text{Node}_1 \rangle$$
Example 6:

2.5.3.4 Calculate the Execution Probability of a Defective Node

The third step is calculating the Execution probability using the formula introduced earlier. Probabilities are obtained directly from the OP table.

\[ E(\text{Node}_i) = \sum_{j=1}^{n} \text{Prob}(\text{Path}_j) \]  

Where for Path\(_i\) the Probability is:

\[ \text{Prob}(\text{Path}_i) = \text{Prob}(P_1 \text{ true/false AND } P_2 \text{ true/false AND } \ldots \text{ P}_k \text{ true/false}) \]

\[ = \text{Prob}(P_1 \text{ true/false}) \times \text{Prob}(P_2 \text{ true/false}) \times \ldots \times \text{Prob}(P_k \text{ true/false}) \]

For Example 1 in Figure 50:

Path\(_1\) = < Start, Node\(_1\) >

This example is very simple. There is only one path, and hence, variable \( n \) in equation (3) is equal to 1. Since there is no predicate on the path, the logic flow does not split, and thus the Execution probability is simply 100%. i.e. \( E(\text{Node}_1) = \text{Prob}(\text{Path}_1) = 100\% \)

For Example 2 in Figure 51:

Path\(_1\) = < Start, P\(_1\) == true, Node\(_1\) >
Again there is only one path. The probability of \((P_1 = \text{true})\) is given in the HLEFSM (Figure 55).

Then the Execution probability is:

\[ E(\text{Node}_1) = \text{Prob}(\text{Path}_1^1) = \text{Prob}(P_1 = \text{true}) = 32\% \]

For Example 3 in Figure 52:

\[ \text{Path}_1^1 = \langle \text{Start}, P_1 = \text{true}, P_2 = \text{false}, \text{Node}_1 \rangle \]

\[ E(\text{Node}_1) = \text{Prob}(\text{Path}_1^1) = \text{Prob}(P_1 = \text{true} \text{ AND } P_2 = \text{false}) = \text{Prob}(P_1 = \text{true}) \times \text{Prob}(P_2 = \text{false}) = 17\% \times 65\% = 11.1\%. \]

For Example 4 in Figure 53 there are two paths:

\[ \text{Path}_1^1 = \langle \text{Start}, P_1 = \text{true}, \text{Node}_1 \rangle \]

\[ \text{Path}_2^1 = \langle \text{Start}, P_1 = \text{false}, \text{Node}_1 \rangle \]

And hence:

\[ E(\text{Node}_1) = \text{Prob}(\text{Path}_1^1) + \text{Prob}(\text{Path}_2^1) = \text{Prob}(P_1 = \text{true}) + \text{Prob}(P_1 = \text{false}) \]

\[ = 21\% + 79\% = 100\%. \]

It can be seen that although a predicate splits the logic flow, the Execution probability of a defective node logically after this predicate is not necessarily less than 100\%. This is because the logic flow may merge back as can be seen from this example.

For Example 5 (see Figure 54) there are three paths:

\[ \text{Path}_1^1 = \langle \text{Start}, P_1 = \text{true}, \text{Node}_1 \rangle \]

\[ \text{Path}_2^1 = \langle \text{Start}, P_1 = \text{false}, P_2 = \text{true}, \text{Node}_1 \rangle \]

\[ \text{Path}_3^1 = \langle \text{Start}, P_1 = \text{false}, P_2 = \text{false}, \text{Node}_1 \rangle \]

Applying equation (3) the Execution is obtained as:

\[ E(\text{Node}_1) = \text{Prob}(\text{Path}_1^1) + \text{Prob}(\text{Path}_2^1) + \text{Prob}(\text{Path}_3^1) \]
Again, as long as the logic flow merges back into a single path, the Execution probability may still be equal to 100%.

For Example 6 in Figure 55:

Path\textsubscript{1} = \langle Start, P_{1} == \text{true}, P_{2} == \text{false}, Node_{1} >

Path\textsubscript{2} = \langle Start, P_{1} == \text{false}, P_{2} == \text{true}, Node_{1} >

And the Execution is:

\[ E(Node_{1}) = \text{Prob}(Path_{1}) + \text{Prob}(Path_{2}) \]

\[ = \text{Prob}(P_{1} == \text{true} \text{ AND } P_{2} == \text{false}) + \text{Prob}(P_{1} == \text{false} \text{ AND } P_{3} == \text{true}) \]

\[ = \text{Prob}(P_{1} == \text{true}) \times \text{Prob}(P_{2} == \text{false}) + \text{Prob}(P_{1} == \text{false}) \times \text{Prob}(P_{3} == \text{true}) \]

\[ = 36\% \times 71\% + 64\% \times 83\% = 78.7\% \]

### 2.5.3.5 The Simple Control System (SCS) Case Study

In this sub-section the methodology will be applied to the Simple Control System (SCS) case study. The modified HLEFSM is provided in Figure 48. There are four defects mapped in the modified HLEFSM. The modified HLEFSM can be simplified by replacing each mapped defect with a defective node, as shown below.
The modified HLEFSM with all mapped defects replaced by defective nodes.

In the above simplified HLEFSM in Figure 56 the four nodes correspond to the four defects. “IAP” stands for “Incorrect/Ambiguous Predicate”; “MF” for “Missing Instance of Function”; “FIC” for “Function with Incorrect Logic”; “MP” for “Missing Predicate”. Next the Execution analysis methodology is applied. First all the paths for each node are identified and the results are provided below:

1. Node 1: only one path.
   \[ Path_1^1 = < Start, initializeTheSystem, Node_1 > \]

2. Node 2: only one path.
   \[ Path_2^1 = < Start, initializeTheSystem, Node_1, Node_2 > \]

3. Node 3: only one path.
   \[ Path_3^1 = < Start, initializeTheSystem, Node_1, Node_2, P_2 = true, Node_3 > \]

   \[ Path_4^1 = < Start, initializeTheSystem, Node_1, Node_2, P_2 = true, Node_3, Node_4 > \]
   \[ Path_4^2 = < Start, initializeTheSystem, Node_1, Node_2, P_2 = false, Node_4 > \]

The second step is to simplify all the paths:

1. Node 1: only one path.
\[ Path_1^{1} = \langle \text{Start}, \text{Node}_1 \rangle \]

2. Node\(_2\): only one path.
\[ Path_1^{2} = \langle \text{Start}, \text{Node}_2 \rangle \]

3. Node\(_3\): only one path.
\[ Path_1^{3} = \langle \text{Start}, P_2 == \text{true}, \text{Node}_3 \rangle \]

4. Node\(_4\): two paths.
\[ Path_1^{4} = \langle \text{Start}, P_2 == \text{true}, \text{Node}_3 \rangle \]
\[ Path_2^{4} = \langle \text{Start}, P_2 == \text{false}, \text{Node}_4 \rangle \]

Then equation (3) can be applied using the OP data displayed in Table 6.

### Table 6 The Operational Profile of the SCS case study

<table>
<thead>
<tr>
<th>No.</th>
<th>Description of the Event</th>
<th>Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Low temperature. i.e., 0 °C ≤ Temperature &lt; 15 °C</td>
<td>15%</td>
</tr>
<tr>
<td>2</td>
<td>Medium temperature. i.e., 15 °C ≤ Temperature ≤ 45 °C</td>
<td>30%</td>
</tr>
<tr>
<td>3</td>
<td>High temperature. i.e., 45 °C ≤ Temperature ≤ 100 °C</td>
<td>55%</td>
</tr>
<tr>
<td>4</td>
<td>Other temperature. i.e., Temperature &lt; 0 °C or Temperature &gt; 100 °C</td>
<td>0%</td>
</tr>
<tr>
<td>5</td>
<td>Low pressure. i.e., 5*10^5 Pa ≤ Pressure &lt; 10^6 Pa</td>
<td>27%</td>
</tr>
<tr>
<td>6</td>
<td>High pressure. i.e., 10^6 Pa ≤ Pressure ≤ 5*10^6 Pa</td>
<td>73%</td>
</tr>
<tr>
<td>7</td>
<td>Other pressure. i.e., Pressure &lt; 5<em>10^5 Pa or Pressure &gt; 5</em>10^6 Pa</td>
<td>0%</td>
</tr>
<tr>
<td>8</td>
<td>The Boolean variable \text{“Status”} being true</td>
<td>34%</td>
</tr>
<tr>
<td>9</td>
<td>The Boolean variable \text{“Status”} being false</td>
<td>66%</td>
</tr>
</tbody>
</table>

The Execution probability for each node is calculated as follows. First recall the equations:

\[
\text{Prob}(Path_1^i) = \text{Prob}(P_1 == \text{true/false}) \times \text{Prob}(P_2 == \text{true/false}) \times \ldots \times \text{Prob}(P_k == \text{true/false})
\]

Incorporating the OP information, one obtains:
1. Node₁: Because there is no predicate on the path, we simply have:

\[ E(\text{Node}_1) = \text{Prob}(\text{Path}_1) = 100\% \]

2. Node₂: Again there is no predicate, thus we have:

\[ E(\text{Node}_2) = \text{Prob}(\text{Path}_2) = 100\% \]

3. Node₃: There is a predicate on the path, thus we have:

\[ E(\text{Node}_3) = \text{Prob}(\text{Path}_3) = \text{Prob}(P_2 = \text{true}) = \text{Prob}((\text{Pressure} > 10^6) = \text{true}) = 73\% \]

4. Node₄: There are two paths for Node₄,

\[ E(\text{Node}_4) = \text{Prob}(\text{Path}_4) + \text{Prob}(\text{Path}_5) = \text{Prob}(P_2 = \text{true}) + \text{Prob}(P_2 = \text{false}) = \text{Prob}((\text{Pressure} > 10^6) = \text{true}) + \text{Prob}((\text{Pressure} > 10^6) = \text{false}) = 73\% + 27\% = 100\% \]

2.5.3.6 Execution Analysis with Loops

There may exist loops in the HLEFSM. For instance, the back edge after “Stmt” and returning to “P₃” in Figure 57 defines a loop. In our methodology the predicates in loops are treated exactly the same as the predicates of the “if-else” structures. i.e., if the statement is within a loop body like “Stmt”, the path should conjunct with the loop predicate. More specifically, the simplified path for “Stmt” is as follows:

Path = <Start, P₁=true, P₂=false, P₃=true>

If the corresponding OP information is available, one can conduct the Execution analysis.
2.6 Infection Analysis

Infection will be said to have occurred when the state of a program which exists right after execution of a defect differs from what it would have been, had the program not been defective. Infection analysis for each defect is specific to the defect itself, which is one of those defined in Table 2. In this section the Infection analysis of Category 5 defects are introduced because it can be directly conducted on the SRS level. “Missing Instance of Function”, “Incorrect/Ambiguous Predicate” and “Missing Predicate” are used in the SCS case study and examples are provided correspondingly. The Infection analysis for other defects including “Function with Incorrect Logic” will be discussed in 2.8.

2.6.1 Infection Analysis for “Missing Instance of Function”

Functions at the SRS level have significant functionalities. Thus it is reasonable (although conservative) to assume that when a function call is missing, the data state that immediately follows the defect will be erroneous. Applying this assumption to the Infection analysis for “Missing Instance of Function” one can obtain an Infection probability which is 100%.

![Figure 57 HLEFSM with a loop](image_url)
2.6.2 Infection Analysis for “Extra Instance of Function” and “Incorrect/Ambiguous Function Call”

Similarly to a “Missing Instance of Function”, an extra function call is assumed to lead to an Infection probability of 100%. For an Incorrect/Ambiguous Function Call, two cases should be considered. In the case of an Incorrect Function Call, the name of the function called is entirely different from the correct one. In the case of an Ambiguous Function Call the name of the function called is similar to the correct one. In the first case, the functions should be quite different and therefore the data state is likely to be significantly different. The Infection probability is conservatively assumed to be 100%. In the second case the Infection probability depends on the number of functions whose name is similar to the correct one. However, because it is impossible to predict how many such functions exist, a conservative assumption is that the Infection probability is also equal to 100%.

2.6.3 Infection Analysis for “Incorrect/Ambiguous Predicate”

From the previous sections it is known that if a predicate is incorrect or ambiguous, the probability of executing the statements (i.e., nested if structures or just function calls) within the if structure is likely incorrect. Here it is assumed that as long as incorrect statements are executed, local results are definitely incorrect, which is consistent with the assumption made for the Infection analysis of “Missing Instance of Function”. Based on this assumption, the Infection analysis for “Incorrect/Ambiguous Predicate” consists in:

1. Finding the incorrect portion of the original predicate
2. Calculating the probability

Based on our defect template for “Incorrect/Ambiguous Predicate” in Figure 36, the problem to be solved is finding the incorrect portion $P_{ow}$ of the original (incorrect) predicate $P_O$. First some useful notations will be defined and then the results are directly given without proof.
First let us assume that all predicates only use one variable (again this restriction is only for convenience of discussion at this point. The theory below can be extended to multi-variable cases), which is $X$. $\Omega$ is the entire range of variable $X$, i.e.,:

$$X \in \Omega, \text{where } \Omega = [X_{\text{min}}, X_{\text{max}}]$$

where $X_{\text{min}}$ and $X_{\text{max}}$ are the upper and lower bounds of a continuous interval. The brackets “[” and “]” indicate that the points at the boundaries are included in the interval. If they were excluded, the brackets would be replaced by parentheses “(” and “)”. $X_{\text{min}}$ and $X_{\text{max}}$ can take values $-\infty$ and $+\infty$, respectively.

As discussed earlier, some variables range information is explicitly described by the SRS document. For those variables whose range is not explicitly given, it is assumed that:

1. If the type of the variable is Integer or Decimal, $X_{\text{min}}$ is $-\infty$ and $X_{\text{max}}$ is $+\infty$.
2. If the type of the variable is Boolean, the range of $X$ is a set with two elements, which are:
   $$X \in \Omega = \{\text{true, false}\}$$
   This means $X$ is either true or false.
3. If the type of the variable is String, all string literals are in the range.

Some typical examples are listed below.

<table>
<thead>
<tr>
<th>#</th>
<th>Variable</th>
<th>Type</th>
<th>Range ($\Omega$)</th>
<th>$X_{\text{min}}$</th>
<th>$X_{\text{max}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$v_1$</td>
<td>Integer</td>
<td>$5 &lt; v_1 \leq 15$</td>
<td>5, excluded</td>
<td>15, included</td>
</tr>
<tr>
<td>2</td>
<td>$v_2$</td>
<td>Decimal</td>
<td>$v_2 &gt; 0.1$</td>
<td>0.1, excluded</td>
<td>$+\infty$, excluded</td>
</tr>
<tr>
<td>3</td>
<td>$v_3$</td>
<td>Decimal</td>
<td>$v_3 &lt; 100.5$</td>
<td>$-\infty$, excluded</td>
<td>100.5, excluded</td>
</tr>
</tbody>
</table>
Table 7 Typical examples for different types of variables (continue)

<table>
<thead>
<tr>
<th></th>
<th>v4</th>
<th>Integer</th>
<th>v4 can be any valid integer</th>
<th>-∞, excluded</th>
<th>+∞, excluded</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>v4</td>
<td>Integer</td>
<td>v4 can be any valid integer</td>
<td>-∞, excluded</td>
<td>+∞, excluded</td>
</tr>
<tr>
<td>5</td>
<td>v5</td>
<td>Boolean</td>
<td>v5 ∈ {true, false}</td>
<td>N/A</td>
<td></td>
</tr>
</tbody>
</table>

In Table 7, examples #2 and #3 are straight forward. The variables in example #1 and #4 are integers, which means that the range is discrete. The variable in example #5 is a Boolean, which means it is either true or false, and the concept of minimum and maximum value does not exist.

First let us discuss the case where the variable X is defined in a continuous interval. The original predicate can be written as:

\[ X \in P_\text{O} = [X_1, X_2] \]

Where \( X_1 \) is the lower boundary of the original predicate and \( X_2 \) is the upper boundary. \( X_1 \) and \( X_2 \) can be \(-\infty\) and \(+\infty\) for integers and decimals, respectively. The brackets can be parentheses as well.

The correct predicate given in the defect report can be written as:

\[ X \in P_\text{C} = [X_3, X_4] \]

Where \( X_3 \) is the lower boundary of the correct predicate and \( X_4 \) is the upper boundary. \( X_3 \) and \( X_4 \) can be \(-\infty\) and \(+\infty\) for integers and decimals, respectively. The brackets can be parentheses as well.

The formula used to identify the wrong portion \( P_{\text{OW}} \) of the original predicate \( P_\text{O} \) is as follows:

\[
P_{\text{OW}} = (P_\text{C} \cap (\Omega \setminus P_\text{O})) \cup (P_\text{O} \cap (\Omega \setminus P_\text{C}))
\]

Next a simple example is discussed to demonstrate why this formula is valid. Suppose there is a variable Temperature which is defined on \([0, 100]\). The original predicate is “If Temperature is greater than or equal to 15, then openValve2”; the correct predicate is “If Temperature is greater than or equal to 45, then openValve2”. Rewriting these conditions using the notations just defined, Table 8 is obtained.
Table 8 Summary of the conditions of the example

<table>
<thead>
<tr>
<th>Name</th>
<th>Symbol</th>
<th>Interval</th>
<th>Lower Boundary</th>
<th>Upper Boundary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Range</td>
<td>$\Omega$</td>
<td>$[0, 100]$</td>
<td>$X_{min} = 0$, included</td>
<td>$X_{max} = 100$, included</td>
</tr>
<tr>
<td>$P_O$</td>
<td>$P_O$</td>
<td>$[15, 100]$</td>
<td>$X_1 = 15$, included</td>
<td>$X_2 = 100$, included</td>
</tr>
<tr>
<td>$P_C$</td>
<td>$P_C$</td>
<td>$[45, 100]$</td>
<td>$X_3 = 45$, included</td>
<td>$X_4 = 100$, included</td>
</tr>
</tbody>
</table>

Applying the formula the following result is obtained:

$$P_{OW} = ([45, 100] \cap ([0, 100] \setminus [15, 100])) \cup ([15, 100] \cap ([0, 100] \setminus [45, 100]))$$

$$= ([45, 100] \cap [0, 15]) \cup ([15, 100] \cap [0, 45])$$

$$= \emptyset \cup [15, 45) = [15, 45)$$

Let us assume the OP is available (Table 6), the probability of $P_{OW}$ is calculated as:

$$\text{Prob}(P_{OW} = \text{true}) = \text{Prob}(\text{Temperature} \in [15, 45)) = 30\%$$

In the above, the original predicate states “If Temperature is greater than or equal to 15, then openValve2”. This means that nothing should happen as long as Temperature is $[0, 15)$; when Temperature belongs to $[15, 100]$, valve #2 should open. However, the correct predicate states “If Temperature is greater than or equal to 45, then openValve2”. This means that nothing will happen as long as Temperature is within $[0, 45)$; Valve #2 should open if Temperature belongs to $[45, 100]$. Comparing the two predicates, it can be seen that:

1. When Temperature is in $[0, 15)$, both predicates implicitly state that “nothing should happen”. The original predicate is correct.
2. When Temperature is in [15, 45), the original predicate requires to “open valve #2” while the correct predicate implicitly states that “nothing happens”. The original predicate is incorrect.

3. When Temperature is in [45, 100], both predicates require to “open valve #2”. The original predicate is correct.

4. Therefore, the interval where the original predicate is incorrect is [15, 45), which is identical to the $P_{ow}$ calculated by using equation (4).

Equation (4) is also applicable to cases where $X$ is a discrete variable, i.e., $X$ is an integer or a Boolean variable. When $X$ is a Boolean variable, $X_{\text{min}}$ and $X_{\text{max}}$ do not exist. The universal set $\Omega$ is \{true, false\}.

The more general case is that the range and the predicates are composed of multiple intervals. In this case the formula for $P_{ow}$ is still valid. The only difference is that the range and the predicates may have the form of:

$$\Omega = [X_{\text{min},1}, X_{\text{max},1}] \cup [X_{\text{min},2}, X_{\text{max},2}] \cup \ldots \cup [X_{\text{min},n}, X_{\text{max},n}]$$

Which does not result in any fundamental differences. In the case of discrete variables the expression of $\Omega$ would use discrete sets instead of intervals.

2.6.4 Infection Analysis for “Missing Predicate”

For “Missing predicate” there is no $P_{o}$ to evaluate before the actions take place. However, another way to solve this problem is to assume the existence of a virtual predicate $P_{o}$ of variable $X$ before the function call ($X$ is the variable appearing in the correct predicate $P_{c}$). In addition, the probability that ($P_{o}$=true) is equal to 100%.
Therefore, the interval for which $P_o$ is true is equal to all possible values of $X$. In other words, the interval for which $P_o$ is true is equal to the range of $X$, which is $\Omega$. Then the formula for $P_{ow}$ can be applied by substituting $\Omega$ for $P_o$:

$$P_{ow} = (P_c \cap (\Omega \setminus P_o)) \cup (P_o \cap (\Omega \setminus P_c))$$

$$= (P_c \cap (\Omega \setminus \Omega)) \cup (\Omega \cap (\Omega \setminus P_c))$$

$$= \emptyset \cup (\Omega \cap (\Omega \setminus P_c))$$

$$= \Omega \cap (\Omega \setminus P_c)$$

$$= (\Omega \setminus P_c)$$

$$= \overline{P_c} \quad (5)$$

Where $\overline{P_c}$ in equation (5) is the complement of $P_c$. Thus the defect “Missing predicate” can be treated as a special case of “Incorrect/Ambiguous predicate”.

### 2.6.5 Infection Analysis for “Extra Predicate”

The technique presented in Figure 58 can be applied for an “Extra Predicate”, but assume the existence of a virtual predicate is added onto the modified (correct) branch. To obtain $P_{ow}$, the $P_c$ is substituted with $\Omega$, which leads to the following derivation:

$$P_{ow} = (P_c \cap (\Omega \setminus P_o)) \cup (P_o \cap (\Omega \setminus P_c))$$

$$= (P_c \cap (\Omega \setminus P_o)) \cup (\Omega \cap (\Omega \setminus P_c))$$
\[ = (\Omega \cap (\Omega \setminus P_O)) \cup (\Omega \cap \emptyset) \]
\[ = (\Omega \cap (\Omega \setminus P_O)) \cup \emptyset \]
\[ = \Omega \cap (\Omega \setminus P_O) \]
\[ = (\Omega \setminus P_O) \]
\[ = \overline{P_O} \quad (6) \]

Where \(\overline{P_O}\) in equation (6) is the complement of \(P_O\).

### 2.6.6 Infection Analysis for the SCS Case Study

In this section Infection analysis is applied to the SCS case study. The OP related data and the defect report can be found in Table 6 and Table 3, respectively. Defect #1 and #4 are the two defects for which the \(P_{OW}\) formulae applies. Defect #1 was already discussed when introducing equation (4), and \(I_1 = 30\%\). Defect #4 will be discussed next. The variable of the predicate is “Status”. Thus the following equations are obtained:

\[ P_C = \{\text{true}\} \]
\[ \Omega = \{\text{true, false}\} \]

Therefore, applying equation (5) for “Missing predicate”:

\[ P_{OW} = \Omega \setminus P_C = \{\text{true, false}\} \setminus \{\text{true}\} = \{\text{false}\} \]

Thus the Infection probability based on the OP data is:

\[ I_4 = \text{Prob}(P_{ow}==\text{true}) = \text{Prob}(\text{Status==false}) = 66\% \]

The Infection probability of Defect #2 is 100% as mentioned in 2.6.1. Defect #3 will be discussed in 2.8.

### 2.7 Propagation Analysis

As discussed earlier in this report, ARPS’s Propagation analysis feature is currently incomplete. The propagation probabilities used in ARPS have to be provided by the user. However,
Propagation analysis is one of the important features of a PIE analysis and should be carried out automatically. In this section, we provide a preliminary mathematical foundation for the extension of ARPS to Propagation.

A defect in a system causes a disturbance that propagates to an output and causes a system failure. If the disturbance does not propagate the system is considered to remain functional and reliable. Fault propagation analysis is an important step in determining system reliability, and defining fault tolerance strategies. It is particularly challenging for software systems. Typically, in the early software design phases the propagation probability for a fault is assumed to be one. However, the assumption that faults will always propagate highly underestimates reliability and valuable resources may be wasted on fixing faults that may never propagate. It is thus essential to understand the fault propagation mechanism, and the conditions under which the disturbance caused by a fault reaches the output. An analytical function-based approach that considers a special property of the function that will operate on the disturbance created by a fault is hence proposed.

The special property of the function is called “flat parts”. A flat part is a continuous range of the function over which the input range of the function, \( f(input) \), satisfies the equation, \( f(x_{correct})=f(x_{faulty}) \). The equation implies, when a faulty input is passed through a function \( f() \), the function produces an output which is the same as the output produced by the correct input. When different functions interact with each other, which is often the case in any software, flat parts will either be preserved, killed or generated. In this research flat part generation and preservation rules are defined. Furthermore, the use of flat parts in determining the fault propagation probability is defined. Interval-arithmetic based rules to determine the flat parts are proposed as well. Flat parts are a property of a function; when multiple functions containing flat parts interact with each other these flat parts undergo a transformation. During this transformation, the original flat parts may
be killed, preserved or new flat parts may be generated. The functional interactions in this research are limited to basic arithmetic operators and advanced arithmetic operators. Four unique rules are defined for the basic arithmetic operations: addition, subtraction, multiplication, and division, and five for advanced operators: integration differentiation and composition.

A sample rule for the composition operator and the corresponding equation are as follows:

**Rule:** If $f_i$ contains a flat part (FP), then the FP is **preserved** by subsequent composition operations given $f_i$ is not an integration operation.

Since the entire FP is preserved, a new FP can be **generated** only from the non FP portion ($FP$) of the function $f_i$. For composition, $g_n(x) = f_n \circ \ldots \circ f_i \circ f_1(x); \ \forall i = 1..n$. $FP$ can be calculated as follows:

$$FP \text{ of } g_n(x) \text{ over } x = Ran \ f_1(x) \cap [Ran \ in \ x \ of \ f_2 \circ f_1(x)] \cap \ldots \cap [Ran \ in \ x \ of \ f_n \circ \ldots \circ f_1(x)]$$

It should be noted that not all functions in a software application will be affected by a defect. The software control flow highly dictates the function interaction. Further, not all faults propagate from one function to another, especially when a faulty output from one function is not used by the subsequent function in the software control flow. In proposed fault propagation analysis only the functions that will be affected by a fault must be determined. Such a determination is made based on a state-machine that determines the state of inputs and outputs to a function. Each state machine diagram includes five states namely defined, used, faulty defined, faulty used, and killed and transitions between these states. The state-machine is used in a control-flow algorithm that traces the function flow in forward and backward directions.

In conclusion, we propose to integrate the flat part rules into the control-flow algorithm, which can determine the faulty variables and the sequencing of functions during execution. This flat part based control-flow algorithm will output an analytical expression for fault propagation analysis.
Flat part based fault propagation analysis can give immediate and reasonably accurate results early on. The proposed fault propagation analysis approach can be applied to component-based systems as well by considering a component being composed of several functions interacting with each other. Further, different fault types have different fault propagation characteristics, which can be quantified using the flat part.

As a part of future research, a fault propagation analysis tool may be built. The tool may implement the flat-part based control-flow algorithm and obtain an analytical expression for fault propagation analysis. In addition, research to extend flat part calculation for multiple variables is required. Present research is a starting point for future multi-variable theories of fault propagation using interval arithmetic. The tool development can be an evolving project which matures as proposed fault propagation analysis approach matures.

2.8 Modeling Defects at the SDD Level

2.8.1 Introduction to the SDD Level Analysis

The underlying philosophy of our methodology is that analysis should be kept at the SRS level (the highest abstraction level) because the SRS level contains the minimal amount of information about the software system. Conducting analysis at the highest abstraction level and hiding unnecessary information will significantly reduce the needed modeling and computational resources. However, in some instances the information at the SRS level is not detailed enough to conduct the Infection analysis. This is usually the case for level 1 SRS functions. In this situation the SDD (Software Design Documentation) level information is needed to further elaborate the description of defective nodes.

The methodology for the SDD level analysis is similar to that of the SRS level. For each defective level 1 SRS function, the corresponding original SDD is used to construct the original SDD level
EFSM (i.e., the Low Level EFSM or LLEFSM). The SDD level defect report is then reviewed and the defects are mapped to the original LLEFSM. The resulting LLEFSM is the modified LLEFSM for that level 1 function. Next Execution, Infection and Propagation analysis are conducted. However, the result is not the failure probability of the entire software but is the failure probability of that specific level 1 function. In other words, the result is the Infection probability of that level 1 function at the SRS level.

2.8.2 Construction of the Original Low Level Extended Finite State Machine Model

The construction of the original LLEFSM is very similar to that of the original HLEFSM. The N-1’th version of the original SDD is reviewed by the user who will use the EFSM graphical language to build the original LLEFSM. Since the original SDD is for the defective level 1 SRS function, there also exist uncovered SDD level defects. The same defect definitions and templates provided in 2.4.1 and Appendix B are adapted for SDD level defects. However, when the LLEFSM is constructed, the SRS level 1 function of interest becomes the SDD level 0 function. This function calls other SDD level 1 functions whose specific logic is not explicitly displayed in the LLEFSM.

2.8.3 Mapping the SDD Level Defects and Constructing the Low Level Extended Finite State Machine Model

As mentioned in the previous sub-section, the defect definitions and templates of the SRS level defects are adopted in the SDD level. Therefore, mapping the SDD level defects follows the same procedure as mapping defects the SRS level. The resulting modified LLEFSM contains all the SDD level defects mapped and will be used for the Execution, Infection and Propagation analysis at the SDD level.
2.8.4 Execution, Infection and Propagation Analysis at the SDD Level

The Execution, Infection and Propagation analysis at the SDD level is similar to that of the SRS level. The only difference is in the Infection analysis for level 1 SDD functions when there is insufficient information at the SDD level. In this case the analysis will have to be conducted at a lower level of abstraction than the SDD level, which is the code level in our methodology. However, code level analysis features have not yet been implemented in ARPS and hence this cannot currently be done. In section 2.9 extensions to code level analysis are discussed. At this point all case studies involved SDD level analysis use fictitious, Infection probabilities for defects such as “Function with Incorrect Logic” and directly given by the Execution, Infection and Propagation data tables without any calculation.

2.8.5 SRS Level Infection Analysis for the openValve3 Function at the SDD Level

In this section SRS level Infection analysis for the openValve3 function of the SCS case study is conducted using SDD level information. This function is a SRS level 1 function affected by a defect “Function with Incorrect Logic”. Based on the original SDD and the defect report provided in A.2 and A.8, the modified LLEFSM in Figure 59 is constructed. In this figure “op” is short for “obtainPosition”; “P” is short for “Position”; “cp1”, “cp2” and “cp3” are short for “calcPos1”, “calcPos2” and “calcPos3”, respectively; “mu” is short for “moveUp” and “md” is short for “moveDown”. The function cards are not repeated in this sub-section for simplicity.
Four defects are represented in Figure 59. These four defects in the categories of “Missing Instance of Function”, “Function with Incorrect Logic”, “Missing Predicate” and “Incorrect/Ambiguous Predicate”, when reading the figure from left to right respectively. Because no predicate precedes Defect #1, #3 and #4, their SDD level Execution Probability is equal to 100%. Defect #2 is preceded by a predicate “if (Position==1)” located before the defective function “calcPos2”, therefore the Execution probability is equal to Prob(Position==1 is false). Based on the OP information presented in A.14, this probability is equal to 100%-15%=85%.

The SDD level Infection analysis for Defect #1 is the same as the one performed at the SRS level. Since the entire function call of “obtainPosition” is missing, the Infection probability is 100%. The Infection analysis for Defect #3 and #4 also makes use of the SDD level OP information. Defect #3 is a SDD level “Missing Predicate” defect. Based on the methodology introduced in 2.6.3, the Infection probability is equal to Prob(Position==3 is false) which is equal to 100%-

Figure 59 The modified LLEFSM of the openValve3 function for the SCS case study
49% = 51%. Defect #4 is a SDD level “Incorrect/Ambiguous Predicate”. Based on 2.6.2, the Infection probability is equal to:

\[ P_{OW} = (P_c \cap (O \setminus P_o)) \cup (P_o \cap (O \setminus P_c)) \]

\[ I_4 = \text{Prob}(P_{OW}) = \text{Prob}([(1,2) \cap [2,3]) \cup ([1] \cap [3])) = \text{Prob}([2]) = 36\% \]

The Infection analysis for Defect #2 requires more detailed information for the SDD level 1 function “calcPos2”. However, since the code level analysis features have not been implemented in the ARPS tool, the Infection probability for this defect is selected arbitrarily and directly given in Table 38. Similarly all SDD level Propagation probabilities are arbitrary and directly provided in Table 38.

**Table 9** summarizes the SDD level Execution, Infection and Propagation data for openValve3. The failure probability of openValve3 can be calculated as:

\[ \text{Prob}(\text{failure at SDD level}) = \sum_{j=1}^{m} E_j \times I_j \times P_j \]

\[ = 100\% \times 100\% \times 5\% + 85\% \times 12\% \times 13\% + 100\% \times 51\% \times 7\% + 100\% \times 36\% \times 15\% \]

\[ = 15.3\% \]

**Table 9** Summary Execution, Infection and Propagation data for openValve3

<table>
<thead>
<tr>
<th>Defect No.</th>
<th>Defect Type</th>
<th>Execution</th>
<th>Infection</th>
<th>Propagation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Missing Instance of Function</td>
<td>100%</td>
<td>100%</td>
<td>5%</td>
</tr>
<tr>
<td>2</td>
<td>Function With Incorrect Logic</td>
<td>85%</td>
<td>12%</td>
<td>13%</td>
</tr>
<tr>
<td>3</td>
<td>Missing Predicate</td>
<td>100%</td>
<td>51%</td>
<td>7%</td>
</tr>
<tr>
<td>4</td>
<td>Incorrect/Ambiguous Predicate</td>
<td>100%</td>
<td>36%</td>
<td>15%</td>
</tr>
</tbody>
</table>
2.8.6 Software Reliability Assessment for the SCS Case Study

All defects data is listed in Table 10. Apply the equation (1) of software reliability calculation to the SCS case study and the software reliability of SCS can be evaluated. Again all Propagation probabilities are directly provided. The software reliability is calculated as follows:

\[ Prob(\text{failure at SRS level}) \]

\[
= 100\% \times 30\% \times 7\% + 100\% \times 100\% \times 5\% + 73\% \times 15.3\% \times 10\% + 100\% \times 66\% \times 3\%
\]

\[ = 10.2\% \]

\[ Re = 1 - Prob(\text{failure at SRS level}) = 1 - 10.2\% = 89.8\% \]

<table>
<thead>
<tr>
<th>Defect No.</th>
<th>Defect Type</th>
<th>Execution</th>
<th>Infection</th>
<th>Propagation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Incorrect/Ambiguous Predicate</td>
<td>100%</td>
<td>30%</td>
<td>7%</td>
</tr>
<tr>
<td>2</td>
<td>Missing Instance of Function</td>
<td>100%</td>
<td>100%</td>
<td>5%</td>
</tr>
<tr>
<td>3</td>
<td>Function With Incorrect Logic</td>
<td>73%</td>
<td>15.3%</td>
<td>10%</td>
</tr>
<tr>
<td>4</td>
<td>Missing Predicate</td>
<td>100%</td>
<td>66%</td>
<td>3%</td>
</tr>
</tbody>
</table>

2.9 Software Reliability Assessment with the ARPS Tool

2.9.1 Introduction

In the previous sections the EFSM-based RePS methodology was introduced, it includes HLEFSM model construction, mapping defects, mapping the operational profile (OP) and evaluating software reliability. However, till this point all analysis performs conducted manually.
In this section the procedure by which the ARPS tool is used to perform the software reliability calculation will be discussed. The principles underlying the use of the ARPS tool are as follows:

1. The SRS, which is the highest level description for the software project, is loaded into the tool and displayed to the user. The user should already be familiar with the SRS before using the tool to evaluate software reliability.

2. The user enters information related to each function and constructs the High-Level (i.e., SRS level) EFSM model of the software.

3. The user enters defect reports for the previous version of the project. The tool will generate a modified HLEFSM with all defects marked out explicitly.

4. The user enters information related to the operational profile (OP) into the tool. The OP describes the probabilities that define the behavior of the system, thereby, reflecting the way a system behaves. The tool will analyze Execution (E) based on the OP.

5. Based on the type of defects the tool will perform the Infection (I) and Propagation (P) analysis. The resulting E, I and P values will be used to calculate software reliability.

6. If there exists defects that cannot be modeled at the SRS level, the user can analyze them at the SDD level. The tool allows the loading of the SDD, constructing LLEFSM and conducting PIE analysis at the SDD level. The resulting failure probability is used as the value for Infection (I) of the defect at the SRS level.

It should be noted that the defect report and operational profile data are from a previous version of the software project. There may still exist other defects that were not identified. Therefore, the software reliability calculated by ARPS is essentially the lower limit value of the current version.

2.9.2 Modules of the ARPS Tool

Based on the principles of the EFSM-based RePS methodology, there are five modules in the ARPS tool:
1. Module 1: This module loads the original version of the SRS and allows the user to construct the original HLEFSM model.

2. Module 2: This module allows the user to enter the identified software defects and map them onto the HLEFSM. This resulting modified HLEFSM has all the defects marked out.

3. Module 3: A module that allows the user to enter the operational profile of the software project. The ARPS tool uses this information to conduct Execution and Infection analysis.

4. Module 4: This module calculates software reliability using the Execution, Infection and Propagation data provided by the user.

5. Module 5: This module allows the user to load the SDD and the corresponding defect reports, to construct the LLFESM and conduct SDD level PIE analysis.

These five modules are implemented into four modes in the ARPS tool, the “Corrected SRS Data Collection” mode, the “Defected SRS Data Collection” mode, the “Corrected SDD Data Collection” mode and the “Defected SDD Data Collection” mode.

2.9.3 Constructing the Original HLEFSM

First, the procedure to construct the original HLEFSM using the ARPS tool is introduced in this section. This is completed under the “SRS Information Collection” mode. There are two steps:

1. Enter the information related to each function into the ARPS tool

2. Enter the logic of the SRS function into the tool and automatically generate the HLEFSM using MATLAB

2.9.3.1 Enter Function-Related Information to the ARPS Tool

As mentioned before, information describing functions is provided in the SRS document, and hence the SRS document should be loaded into the tool. **Figure 60** displays the user interface of the ARPS tool, which the user can use to either create a new project or load an existing project.
To create a new project the .doc file of a SRS document (e.g., the SCS case study) should be loaded. The loaded SRS is displayed in the left side of the interface, as Figure 61 illustrates.
Figure 61 Load the SRS into the ARPS tool

Then, click the “Add Function” button in the lower middle portion of the window and enter the Component Name “SRS” (Figure 62). This name serves as the name of the SRS level 0 function.

Figure 62 Create the SRS level 0 function "SCS"
The SRS level 1 functions are added by clicking the “Add Function” button again. The added functions will be shown in a tree menu (Figure 63).

![Image](image.png)

**Figure 63** Add the SRS level 1 functions

**2.9.3.2 Enter the Input, Output and Variable Information**

Once a function is added, the right side of the interface changes. Information related to Inputs, Outputs, Variables and Logic (just for SRS level 0 function) can be entered through the corresponding tabs. For example, to add the input “Temperature” of the SCS, enter the name in the blank and click the “+” button, then the input “Temperature” will appear in the list in Figure 64. Click on “Type”, “Range” and so forth to enter other relevant information.
The “Range” column is important. It is where the user enters the range within which a variable is allowed to vary. Sometimes range information is not directly specified in the SRS, and the tool will assume one based on variable type.

1. Integer type: between the min and max value of a 32-bit integer.
2. Decimal type: between the min and max value of a 32-bit double.
3. Boolean type: \{true, false\}
4. String type: N/A

In this study the range of Integer and Decimal variable is of most interest. For the Integer type, the range can be provided using the “\{\}” notation and the “\[\]” notation. For example:

1. An integer variable that ranges from 1 to 4 can be represented as \{1, 2, 3, 4\} or \[1, 4\]
2. An integer variable that is equal to 5 is represented as \{5\}

The range of a decimal is represented as follows:

1. An interval with two boundaries is used. If the boundary values are included, bracket “[” and “]” should be used; if the boundary values are excluded, parentheses “(” and “)”
should be used. A fictitious variable “infinity” is also introduced to handle seemingly infinite valves. For example, if a decimal variable “Y” ranges from 5.1 to 9.5 with both bounds included, its range is written as [5.1, 9.5]; an decimal variable “Z” greater than 10 is represented as (10, infinity).

The tool also supports union operation for the range. For example, the range of a decimal variable “Z” can be: “[3.1, 4.2) or [5, 6]” and the range of an integer variable “K” can be: “{1, 2, 3} or {5}”.

For Boolean variables {true} and {false} are used. The braces are required. Figure 65 shows the range of the two SCS inputs.

![Figure 65 Range of the two SCS inputs](image)

Note that variables have unique names. All references to the same variable share the same data. If one instance is modified, then all references will be affected.
2.9.3.3 Construct the Logic of the HLEFSM Using the EFSM Language

When manually constructing the HLEFSM, the graphical representation is drawn by the analyst based on the logic of the SRS. However, when constructing the HLEFSM using the ARPS tool, the SRS logic should be described in a formal manner so that the tool input can be processed. Then the tool will be able to automatically construct the HLEFSM by calling MATLAB. For that, the SRS logic is specified using the “EFSM language”. The EFSM language shares some features with the popular procedural languages such as C. However, its keywords are designed to be more readable. The language is case free, although some of the words are capitalized to be consistent with the manual analysis. Currently the language has features such as function call, if-then-else structure, loop and so on. Each EFSM language grammar construct is equivalent to an EFSM graphical construct, which are shown below. The corresponding commands in C are provided as well.

Figure 66 The EFSM language command for a function call, the corresponding function call in C and the HLEFSM

Figure 66 specified a function call using the EFSM language a function call using the EFSM language. It can be seen that the inputs for the function call are not displayed in the EFSM language. This is because in ARPS the program will not be executed. Only the logic of the
program is of interest. The graphical representation is essentially the same as discussed in earlier sections of this document, but it will be generated using MATLAB. In addition, the flow direction is top-down rather than from left to right.

A simple if statement without the “else” clause is shown in Figure 67. The keywords and symbols are highlighted in blue.

![Diagram of a simple if statement without an else clause]

Figure 67 The EFSM language command for a “if” statement without the "else" clause

An “if” statement with an “else” clause is shown in Figure 68.
The “while” loop is represented in Figure 69.

Figure 68 The EFSM language command for an “if” statement with an "else" clause

The language also supports comments. However, each comment must occupy a different line. This is stricter than other programming languages. There are two types of comments:
1. Comments that start with “//”. See for example Figure 70.

```
... 
fun1  
// This is a comment 
fun2  
...
```

**Figure 70** Comments that start with "//"

2. Comments bounded by “/*….*/”. This command is used for comments that stretch over multiple lines. See for example Figure 71.

```
... 
fun1  
/* This 
is 
a comment */ 
fun2  
...
```

**Figure 71** Comments bounded by "/*....*/"

2.9.3.4 The SCS Case Study

If there are multiple successive state transitions (i.e. functions), the corresponding EFSM language commands are listed in sequence. For example, recall the original SCS logic:

1. The system is initialized.

2. If the temperature is greater than or equal to 15 °C, Valve #1 shall open. Register R1 is set to 1.
3. If the pressure is greater than or equal to $10^6$ Pa, Valve #3 shall open. Register R3 is set to 1.

4. Valve #4 shall open. Register R4 is set to 1.

The corresponding EFSM language commands are displayed in Figure 72.

```
initializeTheSystem
if (Temperature is_greater_than_or_equal_to 15) then {
  openValve1
}
if (Pressure is_greater_than_or_equal_to 1000000) then {
  openValve3
}
openValve4
```

**Figure 72** EFSM language commands for the original logic of SCS

The above commands should be entered in the logic tab of the SRS level 0 function. More specifically, click the button “Enter/Modify Logic” in Figure 73 and the window shown in Figure 74 will be displayed.
Function names, inputs, outputs and variables are provided to the right side of the interface. The key words can be either entered by the user or selected from the panel. After entering all EFSM commands, the “Submit Logic” button should be clicked. Then “Chart Function” to generate the MATLAB input file. The MATLAB representation is saved in a .m file which is located in the ARPS project folder by default. The name is “level 0 function name” + “_original.m”. If we use “SCS” as our level 0 function name, the file name is “SCS_original.m”. Figure 75 displays the resulting HLEFSM. Note that the logic flow is top-to-down instead from left-to-right.
2.9.4 Collecting SRS level Defects

2.9.4.1 Switching to the “Corrected SRS Data Collection Mode”

The ARPS tool also enables defect mapping and construction of a modified HLEFSM. To access this capability the analyst should switch to the “Corrected SRS Data Collection Mode” (Figure 76).
After switching to the defect collection mode, the interface changes as displayed in Figure 77.

The resulting interface is divided into two windows:

1. The left shows the original SRS document.
2. The right shows the inputs, outputs, etc. of a certain function. There are two new tabs called “Logic” and “Operational Profile” which will be discussed in detail.

2.9.4.2 How to Enter the Operational Profile

The “Operational Profile” tab will be used to enter OP-related information. As has been discussed earlier, the OP data is used to conduct the Execution and Infection analysis. When the ARPS tool is used to model the software, these analyses are conducted automatically. Figure 78 shows how the user interface changes if “Operational Profile” is selected.

![Figure 78 The "Operational Profile" tab](image)

It can be seen that there are three entries in the drop-down list for “SCS”. Each entry is a variable (i.e., one of the inputs, outputs and internal variables) of SCS. If the “Temperature” is selected, the interface provides information relevant to this variable. This information is only for review and cannot be modified. The window below the summary of the information is used to enter the operational profile data. The OP data table entries for “Temperature” extracted from the SCS documentation are described in Table 11.
Table 11 OP-related information for Temperature

<table>
<thead>
<tr>
<th>No.</th>
<th>Description of the Event</th>
<th>Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Low temperature, i.e., 0 °C ≤ Temperature &lt; 15 °C</td>
<td>15%</td>
</tr>
<tr>
<td>2</td>
<td>Medium temperature, i.e., 15 °C ≤ Temperature &lt; 45 °C</td>
<td>30%</td>
</tr>
<tr>
<td>3</td>
<td>High temperature, i.e., 45 °C ≤ Temperature ≤ 100 °C</td>
<td>55%</td>
</tr>
<tr>
<td>4</td>
<td>Other temperature, i.e., Temperature &lt; 0 °C or Temperature &gt; 100 °C</td>
<td>0%</td>
</tr>
</tbody>
</table>

Only the OP with nonzero probabilities needs to be entered. The syntax for entering an OP interval range and corresponding probability is given in Figure 79 and Figure 80.

Figure 79 The syntax of entering OP

For example:

Figure 80 Applying the OP syntax to Temperature

The keyword “Prob” is colored blue, and so are the parentheses. The “Interval” is similar to the variable range discussed earlier. The first portion of the OP for variable “Temperature” is for interval [0, 15), and the “Value” is equal to 15%. It can be either a percentage or a decimal between 0 and 1. The interpretation of “Prob([0, 15]) = 15%” is hence “The probability that the
Temperature is greater than or equal to 0 °C and less than 15 °C is equal to 15%”. If there are multiple OP intervals for one variable, each is entered separately one by one.

For Boolean variables such as “Status”, the syntax is identical. However, the “Interval” becomes a singleton set bounded by “{” and “}”.

\[
\text{Prob}\{\text{true}\} = 34\%
\]

**Figure 81** All OP-related information for Temperature has been entered

**Figure 82** Applying the OP syntax to “Status”
2.9.5 How to Map Defects related to the SRS Logic

As discussed in 2.4, all Category 5 defects in Table 2 can be analyzed at the SRS level. Based on the same reasoning these defects can be directly mapped at the SRS level using the EFSM language. The other defects have to be handled at the SDD level, which will be discussed in 2.9.6.

Mapping the Category 5 defects requires changing the EFSM commands of the original HLEFSM. If one clicks the “Logic” tab of “SRS”, the EFSM language command entered in the SRS collection mode is displayed. These commands can only be reviewed in this window. However, if one clicks on the “Enter/Modify Logic” button, the window in Figure 83 appears.

![Figure 83 SCS SRS level logic for modification](image)

The commands in the “Defect SRS” window can be edited, while commands in the “Original SRS” cannot. The syntax which enables mapping all the defects to the original EFSM logic is discussed in the following sections.
2.9.5.1 Syntax for Mapping “Missing Instance of Function”

The syntax for “Missing Instance of Function” is as displayed in Figure 84.

![Figure 84 Syntax for “Missing Instance of Function”](image)

It can be seen that the missing function “fun1” is surrounded by tags.

- The `<MF_Begin>` and `<MF_End>` tags indicate the type of defect.
- The `[Propagation = y%]` tag is used to enter Propagation probability provided in the defect report. By default this value is 100% for all defects. The Infection probability for a missing function is to be assumed 100% as discussed in 2.6.1.

For the SCS case study, Defect #2 falls into this type. The function “openValve2” is missing from the logic. Therefore, the logic should be modified as displayed in Figure 84.

```plaintext
initializeTheSystem
if ( Temperature is_greater_than_or_equal_to 15.0 ) then {
    openValve1
}
<MF_Begin> [Propagation = 5%]
openValve2
<MF_End>
if ( Pressure is_greater_than_or_equal_to 1000000.0 ) then {
    openValve3
}
openValve4
```

Figure 85 SCS logic with Defect #2 mapped
2.9.5.2 Syntax for Mapping “Extra Instance of Function”

The syntax for “Extra Instance of Function” is given in Figure 85.

Similarly, the propagation probability, is provided by the user, he/she substitute “y” with an actual value.

2.9.5.3 Syntax for Mapping “Incorrect/Ambiguous Function Call”

The syntax for an “Incorrect/Ambiguous Function Call” is given in Figure 86.

In this case the [Fc = fun] is used to provide the correct function call, where “fun” should be substituted with the correct function name. The Propagation probability still needs to be provided.

2.9.5.4 Syntax for Mapping “Missing Predicate”

The syntax for a “Missing Predicate” is given in Figure 88.
Again the <MP_Begin> and <MP_End> tags indicate that the defect is of type “Missing predicate”. The [Pc = correct_predicate] is used to enter the predicate that was missing. The EFSM language syntax of writing a predicate should be used.

The function “fun1” needs to be bounded by braces “{“ and “}”. This is because multiple functions may be nested in one if-then structure. If that is the case, then all function calls must be bounded by the braces. The [Propagation = y%] tag is still used to enter the Propagation probability.

In the SCS case study, Defect #4 falls into this type. Function “openValve4” should be opened only when the variable “Status” is evaluated as true. Therefore, the logic should be modified as displayed in Figure 89.

```
initializeTheSystem
if (Temperature is_greater_than_or_equal_to 15.0) then {
  openValve1
}
if (Pressure is_greater_than_or_equal_to 1000000.0) then {
  openValve3
}
<MP_Begin> [Pc = Status is_equal_to true] [Propagation = 3%] {
  openValve4
}<MP_End>
```

**Figure 89** The SCS logic with Defect #4 mapped
2.9.5.5 Syntax for Mapping an “Extra Predicate”

The syntax for an “Extra Predicate” is given in Figure 90.

![Figure 90 Syntax for "Extra Predicate"

In the above figure, the predicate “x is_greater_than 5” is the extra predicate. The entire “if” structure should be bounded by the tags and the Propagation probability should be provided.

2.9.5.6 Syntax for Mapping an “Incorrect/Ambiguous Predicate”

The syntax for an “Incorrect/Ambiguous Predicate” is given in Figure 91.

![Figure 91 Syntax for "Incorrect/Ambiguous Predicate"

Similar as before, the <IAP_Begin> and <IAP_End> tags indicate that the defect type is “Incorrect/Ambiguous Predicate”. The [Pc = correct_predicate] tag is used to provide the correct
predicate. The [Propagation = y%] tag is still used to provide the Propagation probability. It should be noted that the original predicate, which is incorrect, is not modified.

In the SCS case study, Defect #1 is of this type. The modified logic is given in Figure 92.

```plaintext
initializeTheSystem
  <IAP_Begin> [Pc = Temperature is_greater_than_or_equal_to 45.0] [Propagation = 7%] if ( Temperature is_greater_than_or_equal_to 15.0 ) then {
    openValve1
  }
  <IAP_End>
  if ( Pressure is_greater_than_or_equal_to 1000000.0 ) then {
    openValve3
  }
openValve4
```

**Figure 92** The SCS logic with the Defect #1 mapped

### 2.9.5.7 The EFSM Language Commands for Mapping Defects #1, #2 and #4 for the SCS Case Study

The EFSM language commands for mapping Defects #1, #2 and #4 discussed in the previous sections are combined and provided in Figure 93. Note that Defect #3, a “Function with Incorrect Logic”, is not mapped using the EFSM language at the SRS level. This is because this type of defect must be modeled at the SDD level, which will be discussed in 2.9.6.
2.9.6 LLEFSM Construction and Defect Modeling

The LLEFSM features of the ARPS tool are used when the defects cannot be precisely modeled at the SRS level. This is usually the case for Category 1 to 4 defects in Table 2. To model these defects, the ARPS mode of entry corresponding should be switched to “SDD Collection Mode”.

For Defect #3 in the SCS case study, because the logic is defective, the user needs to:

1. Enter all SDD level 1 functions-related information.
2. Check the “Logic Defected” button shown in Figure 94.
Once these two steps are completed, the “Go to SDD Level” button can be clicked (before it was grey). For other types of defects the process is similar. **Figure 95** displays the user interface after switching to the “SDD Collection Mode”.

**Figure 94** Switch to "SDD Collection Mode" through defect "Function with Incorrect Logic"
The SDD collection mode parallels the SRS collection mode and gathers the LLEFSM logic. Recall the analysis conducted in 2.8.5. The original LLEFSM logic is given in Figure 96.

```
IF ( Position IS_EQUAL_TO 1 ) THEN {
  calcPos1
} else {
  calcPos2
}
calcPos3
IF ( Position IS_EQUAL_TO 1 ) THEN {
  moveUp
} else {
  moveDown
}
```

**Figure 96 The logic of the original LLEFSM of SCS’s “openValve3”**

The OP information and the modified LLEFSM logic can be entered in similar ways as the SRS level, which are ignored here. Figure 97 directly provides the modified LLEFSM commands for
“openValve3”. After this click on “Calculate Infection Rate” the Infection probability of openValve3 is calculated, and then the user can switch back to the SRS level to finish up the reliability calculation. The result will be exactly the same as the manual analysis.

Figure 97 The logic of the modified LLEFSM of SCS’s “openValve3”

2.10 Experimental Validation for the ARPS Tool

2.10.1 Introduction

As discussed in the previous sections, one major goal of this study is to investigate whether the ARPS tool is really helpful for improving the performance of the analysts. This is because although the EFSM based RePS methodology can be manually applied, it is tedious and error prone. It is expected that the automation features introduced in the tool can help reduce the
number of the errors as well as lower the error criticality. To verify if this expectation is true, an experiment with human subjects was conducted, where the subjects were taught how to manually apply the methodology and how to use the tool. Usability measures were calculated based on the subjects’ performance. The study shows that the tool helped lower the number and criticality of the subjects’ mistakes.

2.10.2 Research Subject Identification

The expected user of the ARPS tool should possess the following background knowledge:

1. He/she should understand system modeling concepts
2. He/she should be able to conduct engineering level of mathematical and statistical analysis
3. He/she should be able to write computer programs using procedural programming languages

The above requirements indicate that the user should have an engineering degree. Junior and senior undergraduate students were recruited for the experiment because their expertise is close to the one of an expected user. Fifteen subjects were recruited from mechanical engineering, electrical engineering, computer science and engineering and integrated system engineering. They were equally divided into two groups based on their year and major.

2.10.3 Research Design

2.10.3.1 The Variables of the Experiment

1. The independent variable—the software reliability modeling approach. The approach is either manual or tool-based. In other words, experiment groups either manually apply the EFSM-based RePS methodology (will be referred as $M$) or use the ARPS tool (will be referred as $A$).
2. The controlled variable—the background knowledge and experience of the subjects and it is measured on an ordinal scale.
3. The dependent variable—the dependent variable is the usability measures (defined below) of
the software reliability modeling techniques.

2.10.3.2 The Usability Measures of the Experiment

Training sessions were designed to teach the students the manual and automated version of the method. Measures were collected during exams which followed these sessions. Two usability measures were considered in this experiment:

1. Error Index ($EI$): $EI$ is a score expressed as the percentage of a task correctly completed by a subject. A higher $EI$ indicates a higher performance. The specific grading policy used in this study is provided in Table 12. The grading policy is based on types of mistakes a subject may make.

<table>
<thead>
<tr>
<th>No</th>
<th>Error Type</th>
<th>Points obtained</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Missing</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>Partially incorrect</td>
<td>0.5</td>
</tr>
<tr>
<td>3</td>
<td>Correct</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>Extra</td>
<td>-1</td>
</tr>
</tbody>
</table>

There are three steps in each exam problem, for both manual and tool analysis: (1) reviewing the SRS and building the original HLEFSM; (2) reviewing the SRS level defect report and mapping the defects onto the original HLEFSM; (3) conducting the Execution, Infection and Propagation analysis to assess software reliability. Although the specific errors committed by the subjects vary from one step to another, they can all be categorized as missing, partially incorrect or extra. Thus the grading policy in Table 12 is designed to be applicable to for all three steps.

2. Total Number of Errors ($NE$): The total number of errors committed by a subject. This
measure counts the total number of mistakes made by a subject without considering the error type. It is an objective complement to EI because both the EI grading policy and the researcher’s judgment with respect to the type to be assigned to a certain error are subjective.

3. **Time (T):** $T$ is the total amount of time to complete a test problem, in number of minutes. Each subject completed one small problem manually, one small problem using the tool, one larger problem manually and one larger problem using the tool. Therefore, which method is more time consuming can be investigated.

4. **Difference between number of “missing” errors for the manual analysis and number of “missing” errors for the tool analysis (Missing).** This measure only considers the number of “missing” errors committed by a subject. For a certain subject, this measure is defined as the number of “missing” errors committed during manual analysis minus the number of “missing” errors committed during tool analysis.

5. **Difference between number of “partially incorrect” errors for the manual analysis and number of “partially incorrect” errors for the tool analysis (Incorrect).** This measure only considers the number of “partially incorrect” errors committed by a subject. For a certain subject, this measure is defined as the number of “partially incorrect” errors committed during manual analysis minus the number of “partially incorrect” errors committed during tool analysis.

6. **Ease of Learning (Ease):** A subjective measure evaluating the ease of learning. Four levels of ease are considered: 4-very easy, 3-easy, 2-moderately difficult, 1-very difficult. The subjects were asked to select one of the four options.

7. **Satisfaction (Sat):** A subjective measure evaluating the degree of subject satisfaction. Four levels of satisfaction are considered: 4-very satisfied, 3-satisfied, 2-moderately dissatisfied, 1-very dissatisfied. The subjects were asked to select one of the four options.
The above two measures are introduced to answer the question in this experiment: “Determine whether the ARPS tool support makes the original HLEFSM construction, the defect mapping and the reliability assessment more usable than it would otherwise be (without tool support) ”.

The corresponding general hypothesis is that there is no difference between the usability measures of the manually analysis (\(M\)) and the ARPS tool analysis (\(A\)). The general alternative hypothesis (\(H_A\)) is that the difference in the usability measures of \(M\) and \(A\) is significant. These two hypotheses are tailored for the two different measures. Based on whether or not the data follows a normal distribution, different approaches to hypothesis testing are used.

### 2.10.3.3 Design of the Experiment

The design of the experiment is provided in **Table 13**. The initial training session was exactly the same for each group. The training session lasted 3.5 days, and there were two classes in each morning and afternoon session. Each class lasted 80 minutes with a 15 minutes break. The same classroom was used for the entire training session. The training was conducted in a manner identical to typical class lectures, where the materials were displayed on the blackboard or using slides. The training covered an introduction to the RePS methodology and the ARPS tool as well as a description of the specific manual and tool analysis techniques. The subjects were encouraged yet not required to review the materials after the classes.

During the training session, the lectures emphasized four types of defects “Missing Instance of Function”, “Function with Incorrect Logic”, “Missing Predicate” and “Incorrect/Ambiguous Predicate”. These four defects were selected because they were representative of the most important and challenging defect templates and versions of the PIE analysis techniques. Note that Propagation analysis was not mentioned and all Propagation probabilities were directly provided in the defect reports.
The schedules of the two groups were different during the two testing sessions. In testing session #1 each subject was asked to solve two small scale problems, one manually and the other using the ARPS tool. Group1 subjects first solved problem Test #1 manually (hence called MT1 in Table 13) and then solved problem Test #2 using the tool (hence called AT2 in Table 13). The order of the tests was reversed for Group#2. In testing session #2, two larger scale problems were solved by each subject. The scale factor between small and large applications is 5, and quantified based on the total number of state transitions and predicates in each application. There was no time limit set for the two testing sessions.

**Table 13** The design of the experiment

<table>
<thead>
<tr>
<th></th>
<th>Test. Session #1</th>
<th>Test. Session #2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Group1</td>
<td>Training</td>
<td>MT1</td>
</tr>
<tr>
<td>Group2</td>
<td>Training</td>
<td>AT1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MT3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 2.10.4 Data Analysis

#### 2.10.4.1 Error Index (EI)

First a normality test [30] was conducted on the data, and the results are shown in Figure 98 to Figure 101. It can be seen that none of the raw data follows a normal distribution. Thus non-parametric hypothesis test techniques such as Wilcoxon signed test and Sign test were considered [31, 32]. However, a Wilcoxon signed test requires the distribution to be symmetric, which was not the case in this study. Therefore, a Sign test was used to analyze the available data.
Figure 98 Normality test for $EI$ (Manual, T1 and T2)

Figure 99 Normality test for $EI$ (Tool, T1 and T2)
Figure 100 Normality test for EI (Manual, T3 and T4)

Figure 101 Normality test for EI (Tool, T3 and T4)

Figure 102 and Figure 103 illustrate the EI results with respect to the subjects. Note that a higher EI indicates a better performance. For the small scale problems, the manual EI value crosses the tool EI value. From this, one infers that tool results are not necessarily better than manual. However, for the larger scale problems the results are different. In this case the tool EI value is systematically higher than the manual EI value. Table 14 lists the corresponding statistics. The
statistical power is calculated based on [33]. For the small scale problems, the p-value is fairly large (0.180), which indicates that the null hypothesis cannot be rejected at the significance level of $\alpha = 0.05$. In other words, there is no obvious difference between manual and tool results. The statistical power is also low (0.195). However, for the larger scale problems, the p-value is small (1E-4) which means that the null hypothesis can be strongly rejected. The statistical power is also high (0.970). The 98.7% confidence interval (1.0, 28.9) is above 0, from which we also infer that subjects’ tool performance is superior to manual performance.

**Figure 102** EI for small scale problems (T1 & T2)
*Figure 103* $EI$ for larger scale problems (T3 & T4)

**Table 14** Summary of the statistics for $EI$ (a score expressed as the percentage of a task correctly completed by a subject)

<table>
<thead>
<tr>
<th></th>
<th>$EI_{tool} - EI_{manual}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tests</td>
<td>Test 1&amp;2</td>
</tr>
<tr>
<td>No. data points</td>
<td>15 (14 nonzero)</td>
</tr>
<tr>
<td>Mean</td>
<td>1.4</td>
</tr>
<tr>
<td>Standard Deviation</td>
<td>15.8</td>
</tr>
<tr>
<td>p-value (two-tailed)</td>
<td>0.180</td>
</tr>
<tr>
<td>Statistical Power ($\alpha \leq 0.05$)</td>
<td>0.195</td>
</tr>
<tr>
<td>Confidence Interval (CI)</td>
<td>98.7% CI is (-20.3, 13.4)</td>
</tr>
</tbody>
</table>

### 2.10.4.2 Total Number of Errors (NE)

The normality test was first conducted on the raw NE data (see in *Figure 104* to *Figure 107*). The plots indicate that the data does not follow a normal distribution and the Sign test is used for data analysis. *Figure 108* and *Figure 109* provide the plots for the small scale and the larger scale
problems, respectively. For the small scale problems, the manual and the tool NE values still cross each other, from which it can be inferred that there is probably no obvious difference between the results. However, for larger scale problems, the tool NE values are systematically lower than the manual NE, which means that the subjects made fewer mistakes when they used the tool to solve the problems. A summary of the test statistics is shown in Table 15. It can be seen that for the small scale problems, the p-value is large (0.180) and the power is small (0.195). However, for the larger scale problems the p-value is small (0.002) and the power is much larger (0.915). Thus for the small scale problems the null hypothesis cannot be rejected, which says that there is no obvious difference between manual and tool analysis performance. However, for the larger scale problems the null hypothesis can be safely rejected. Based on the confidence interval it is deduced that the subjects made fewer mistakes when using the tool than when performing the analysis manually.

Figure 104 Normality test for \( NE \) (Manual, T1 & T2)
Figure 105 Normality test for \( NE \) (Tool, T1 & T2)

\[
y = 0.8124x + 2E-15 \\
R^2 = 0.7236
\]

Figure 106 Normality test for \( NE \) (Manual, T3 & T4)

\[
y = 0.8539x + 8E-16 \\
R^2 = 0.7994
\]
Figure 107 Normality test for $NE$ (Tool, T3 & T4)

Figure 108 $NE$ for small scale problems (T1 & T2)
Figure 109 *NE* for larger scale problems (T3 & T4)

**Table 15** Summary test statistics for *NE*

<table>
<thead>
<tr>
<th>Tests</th>
<th><em>NE</em>&lt;sub&gt;Manual&lt;/sub&gt; – <em>NE</em>&lt;sub&gt;Tool&lt;/sub&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>No. data points</td>
<td>15 (14 nonzero) – 15 (14 nonzero)</td>
</tr>
<tr>
<td>Mean</td>
<td>1.4 – 22.1</td>
</tr>
<tr>
<td>Standard Deviation</td>
<td>4.3 – 22.2</td>
</tr>
<tr>
<td>p-value (two-tailed)</td>
<td>0.180 – 0.002</td>
</tr>
<tr>
<td>Statistical Power (α ≤ 0.05)</td>
<td>0.195 – 0.915</td>
</tr>
<tr>
<td>Confidence Interval (CI)</td>
<td>98.7% CI is (−4, 6) – 98.7% CI is (1, 46)</td>
</tr>
</tbody>
</table>

2.10.4.3 Time (*T*)

The measure *Time* does not follow a normal distribution either. Hence the Sign test is used. (In fact none of the measures follow a normal distribution and hence the Sign test is used in all cases). **Figure 110** and **Figure 111** provide the plots for *T* for the small scale problems and larger scale problems, respectively. For the small scale problems the manual analysis and the tool analysis prove to be equally time consuming. However, for the larger scale problems the tool analysis saves time for 12 out of 15 subjects. **Table 16** lists the specific statistics: for the small scale
problems the p-value is large (0.791) and the statistical power is small; however, for the larger scale problems the p-value is small (0.035), meaning that the difference between the manual $T$ value and the tool $T$ value is significant. The statistical power (0.648) is also much larger than that observed for small scale problems.

**Figure 110** $T$ for the small scale problems

**Figure 111** $T$ for the larger scale problems
2.10.4.4 Number of “Missing” Errors (Missing) and “Partially Incorrect” Errors (Incorrect)

The tool is effect on the number of “missing” errors and “partially incorrect” errors is also investigated. Note that the number of “extra” errors is not considered because there are too few to support any statistically meaningful comparisons.

Figure 112 displays the plot for (the number of errors of type) Missing, where the difference between the manual Missing and the tool Missing is directly shown. When the test problems are small, the tool does not really help reduce the number of “missing” errors, since the plot oscillates around the horizontal axis. However, when the problems become larger, the manual Missing is much larger than the tool Missing, implying that the tool significantly reduces the number of “missing” errors. Table 17 provides the detailed statistical results: For the small scale problems the p-value is very large (0.774) meaning that the null hypothesis cannot be rejected, while for the larger scale problems the p-value is very small (0.006). Both confidence intervals support the statistical inference, too.
Table 17 Missing statistics summary

<table>
<thead>
<tr>
<th>Tests</th>
<th>Missing (Manual - Tool)</th>
</tr>
</thead>
<tbody>
<tr>
<td>No. data points</td>
<td>Test 1&amp;2</td>
</tr>
<tr>
<td></td>
<td>Test 3&amp;4</td>
</tr>
<tr>
<td>Mean</td>
<td>-0.5</td>
</tr>
<tr>
<td></td>
<td>18.4</td>
</tr>
<tr>
<td>Standard Deviation</td>
<td>4.1</td>
</tr>
<tr>
<td></td>
<td>20.9</td>
</tr>
<tr>
<td>p-value (two-tailed)</td>
<td>1.000</td>
</tr>
<tr>
<td></td>
<td>0.006</td>
</tr>
<tr>
<td>Statistical Power</td>
<td>(α ≤ 0.05)</td>
</tr>
<tr>
<td></td>
<td>0.039</td>
</tr>
<tr>
<td></td>
<td>0.920</td>
</tr>
<tr>
<td>Confidence Interval (CI)</td>
<td>96.1% CI is (-6, 3)</td>
</tr>
<tr>
<td></td>
<td>96.1% CI is (3, 38)</td>
</tr>
</tbody>
</table>

Figure 113 provides the plot for (the number of errors of type) Incorrect, where the difference between the manual Incorrect and the tool Incorrect is displayed. In this case both plots are mostly above the horizontal axis, which means that the tool is useful for mitigating “partially incorrect” errors, regardless of the problem scale. Table 18 shows that both p-values are small. However, it should be noted that for both cases the statistical power is medium.
### Table 18 Incorrect statistics summary

<table>
<thead>
<tr>
<th>Tests</th>
<th>Incorrect</th>
<th>Incorrect (Manual - Tool)</th>
</tr>
</thead>
<tbody>
<tr>
<td>No. data points</td>
<td>15 (14 nonzero)</td>
<td>15 (14 nonzero)</td>
</tr>
<tr>
<td>Mean</td>
<td>1.7</td>
<td>2.8</td>
</tr>
<tr>
<td>Standard Deviation</td>
<td>2.0</td>
<td>3.9</td>
</tr>
<tr>
<td>p-value (two-tailed)</td>
<td>0.013</td>
<td>0.057</td>
</tr>
<tr>
<td>Statistical Power $\alpha \leq 0.05$</td>
<td>0.676</td>
<td>0.4</td>
</tr>
<tr>
<td>Confidence Interval (CI)</td>
<td>94.3% CI is (1, 3)</td>
<td>94.3% CI is (1, 5)</td>
</tr>
</tbody>
</table>

#### 2.10.4.5 Ease of Learning (Ease)

**Figure 114** displays the plot for *Ease*, which is designed to investigate the subjects’ perception of the ease of use of the two methodologies. For all subjects the tool *Ease* is higher than or equal to the manual *Ease*. However, in only 4 instances is the difference different from zero and hence the statistics are not adequate: The two-tailed p-value is 0.125, which is not small enough to make any meaningful inference. Neither confidence interval nor statistical power can be calculated because of the small number of non-zero data points.
2.10.4.6 Satisfaction (Sat)

Figure 115 displays the plot for Sat, which is designed to investigate the subjects’ perceived level of satisfaction with the two methodologies. For 14 out of 15 subjects, the level of satisfaction with the tool is equal to or higher than the level of satisfaction with the manual analysis. However, because the number of non-zeros is small (6), the statistics are inadequate: the two-tailed p-value is as high as 0.219 and the statistical power is still fairly low (0.339).
2.10.5 Threat to Validity

2.10.5.1 Internal Validity

1. Selection bias

As mentioned earlier, our research subjects are junior and senior engineering undergraduate students. However, the potential difference in knowledge between these two groups should not affect the results. Indeed the EFSM-based RePS methodology and the ARPS tool are new to both groups of subjects. In addition, the materials taught are not based on pre-existing knowledge that is selectively accessible to one group but not to the other. A difference in programming knowledge may exist between the mechanical engineering subjects and the electrical engineering and computer science subjects. However, the electrical engineering and computer science subjects were randomly assigned in equal proportion to the two groups. This will mitigate the difference in programming knowledge.

2. Rivalry

The two groups of subjects were trained using the same lecture materials, followed the same classes and were not informed a priori of the fact that they would be divided into two groups for testing. Hence no rivalry between the two groups could have existed during the training process. The two groups were separated into two different classrooms during the testing sessions but were told that they would be solving identical problems. They were never aware of the fact that the techniques were administered in a different order. Thus no desire to out-perform the other group should have existed during the testing sessions either.

2.10.5.2 External Validity

The expected user of the ARPS tool should possess an engineering degree, while our research subjects were junior and senior undergraduate students. However, it is believed that generalizing
our research findings from these subjects to the expected users is valid because they should share similar background and learning potential. The subjects are likely to be less knowledgeable than the expected users in their particular area of expertise, but only average level of knowledge in programming, mathematics and modeling are required. Since these characteristics are shared by the subjects and the expected user, a generalization of our research findings should be valid.

2.10.6 Summary of the Experimental Results

Table 19 summarizes all results. As already discussed, the usability measures for the ARPS tool are not superior to those for the manual analysis when applied to the small scale problems (Test 1&2): for \( EI, T, NE \) and \( Missing \), the two-tailed \( p \)-values are large and the statistical powers are small. However, for the larger scale problems (Test 3&4) these four usability measures indicate that the tool not only helps mitigate various types of errors but also saves time for the analyst. All four \( p \)-values are small and the corresponding statistical powers are fairly large. The measure \( Incorrect \) is slightly different from the others: for both small scale and larger scale problems the \( p \)-values are small and the statistical powers are medium. While there is evidence that the tool helps avoid “Partially incorrect” errors, the statistics indicate that further evidence is needed.

Observation of subjects’ performance shows that the automation features of the ARPS tool help avoid errors. When the subjects applied the defect templates to map the defects manually, they made mistakes in drawing the modified branches. This did not occur during the tool analysis since the tool automatically generated the modified branches. Mistakes in the modified HLEFSM also propagated to the reliability assessment, which further deteriorated the manual results. Additional mistakes were made during the manual Execution, Infection and Propagation analysis, which can be effectively avoided by using the tool as well. Therefore, automation features are the most valuable contribution of our tool and hence more such features should be introduced in the future.
Table 19 Summary of all the results

<table>
<thead>
<tr>
<th>Measure</th>
<th>Small Scale</th>
<th>Larger Scale</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Two-tailed p-value</td>
<td>Power</td>
</tr>
<tr>
<td>EI\textsuperscript{Tool-Manual}</td>
<td>0.180</td>
<td>0.195</td>
</tr>
<tr>
<td>T\textsuperscript{Manual-Tool}</td>
<td>0.791</td>
<td>0.028</td>
</tr>
<tr>
<td>NE\textsuperscript{Manual-Tool}</td>
<td>0.180</td>
<td>0.195</td>
</tr>
<tr>
<td>Missing\textsuperscript{Manual-Tool}</td>
<td>1.000</td>
<td>0.039</td>
</tr>
<tr>
<td>Incorrect\textsuperscript{Manual-Tool}</td>
<td>0.013</td>
<td>0.676</td>
</tr>
<tr>
<td>Ease</td>
<td>Based on observation, tool is equally or easier to learn than manual analysis. P-value is 0.125. Power cannot be calculated</td>
<td></td>
</tr>
<tr>
<td>Sat</td>
<td>Based on observation, tool is equally or more satisfying than manual analysis. P-value is 0.219. Power is 0.339</td>
<td></td>
</tr>
</tbody>
</table>

127
Chapter 3: Heuristics to Improve the Software Strong Mutation Score

In Chapter 2 how software faults can be used to assess software reliability is explained. In this chapter our research advances in software strong mutation are discussed, where software faults are automatically seeded into the source code to create the so called “software mutants”. These mutants are then used to support automated test case generation, which is used to identify the real faults.

In this chapter the mutation testing techniques are first reviewed. After that heuristics for propagating the data state difference are introduced. These heuristics are verified using several sample C programs. The result shows that the proposed heuristics can help propagate the data state difference as expected, and ultimately can generate strong mutation test cases.

3.1 Introduction to the Software Mutation Testing Techniques

3.1.1 What is Mutation Testing?

Software mutation testing is a testing technique first developed by researchers from Yale University and the Georgia Institute of Technology [34-37]. It is based on intentionally seeding small faults into the source code of a program. The fault seeding is a systematic process and each seed produces a very small change in the program, usually within one line of source code. Other portions of the source code remain unchanged. In this thesis, the original version of the source code will be referred as P (for “program”); the modified source code with small faults are called mutants, and will be referred as M (for “mutants”).

128
The mutation testing technique is essentially a test design criterion and used to evaluate the quality of test cases. Test cases must be able to distinguish mutants from the original program even though the mutant faults are typically small. The more mutants distinguished by a test case, the better the test case.

Figure 116 displays an example of an original program and one possible mutant. The mutant is the same as the original program except for the mutated line, where the arithmetic operator * is replaced by +. Similarly, if the operator * is replaced by other operators such as -, /, or ^, three more mutants are generated. The other lines of code can be mutated as well. Therefore, the total number of mutants is large even for small programs.

<table>
<thead>
<tr>
<th>P</th>
<th>M</th>
</tr>
</thead>
</table>
```
int Function1 (int in1, int in2) {
    int a, b, c, d, output;
    a = in1 + 3;
    b = in2 * 5;
    c = a ^ b;
    if (c >= 10) {
        d = 100;
    } else {
        d = b * c;
    }
    output = a + b + d;
    return output;
}
```
```
int Function1 (int in1, int in2) {
    int a, b, c, d, output;
    a = in1 + 3;
    b = in2 + 5;  // Mutated
    c = a ^ b;
    if (c >= 10) {
        d = 100;
    } else {
        d = b * c;
    }
    output = a + b + d;
    return output;
}
```

Figure 116 A sample program and one possible mutant

3.1.2 Mutant Operators

A mutant operator defines how to systematically modify the original source code. A list of mutant operators for FORTRAN 77 is provided in Table 20 [38]. Although these mutant operators are defined for FORTRAN, most are generic and can be applied to other programming languages such as C, C++, Java, etc.
### Table 20 List of mutant operators

<table>
<thead>
<tr>
<th>Operand Replacement Operators</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>AAR</td>
<td>Array reference for array reference replacement</td>
</tr>
<tr>
<td>ABS</td>
<td>Absolute value insertion</td>
</tr>
<tr>
<td>ACR</td>
<td>Array reference for constant replacement</td>
</tr>
<tr>
<td>AOR</td>
<td>Arithmetic operator replacement</td>
</tr>
<tr>
<td>ASR</td>
<td>Array reference for scalar variable replacement</td>
</tr>
<tr>
<td>CAR</td>
<td>Constant for array reference replacement</td>
</tr>
<tr>
<td>CNR</td>
<td>Comparable array name replacement</td>
</tr>
<tr>
<td>CRP</td>
<td>Constant replacement</td>
</tr>
<tr>
<td>CSR</td>
<td>Constant for scalar variable replacement</td>
</tr>
<tr>
<td>DER</td>
<td>DO statement end replacement</td>
</tr>
<tr>
<td>DSA</td>
<td>DATA statement alterations</td>
</tr>
<tr>
<td>GLR</td>
<td>GOTO label replacement</td>
</tr>
<tr>
<td>LCR</td>
<td>Logical connector replacement</td>
</tr>
<tr>
<td>ROR</td>
<td>Relational operator replacement</td>
</tr>
<tr>
<td>RSR</td>
<td>RETURN statement replacement</td>
</tr>
<tr>
<td>SAN</td>
<td>Statement analysis (replacement by TRAP)</td>
</tr>
<tr>
<td>SAR</td>
<td>Scalar variable for array reference replacement</td>
</tr>
<tr>
<td>SCR</td>
<td>Scalar for constant replacement</td>
</tr>
<tr>
<td>SDL</td>
<td>Statement deletion</td>
</tr>
<tr>
<td>SRC</td>
<td>Source constant replacement</td>
</tr>
<tr>
<td>SVR</td>
<td>Scalar variable replacement</td>
</tr>
<tr>
<td>UOI</td>
<td>Unary operator insertion</td>
</tr>
</tbody>
</table>

Because this research focuses on scientific software, we restricted ourselves a few selected mutant operators from Table 20. Previous studies [39, 40] have shown that five mutant operators are particularly adept at killing mutants. These five mutant operators are described below:

- **AOR** – arithmetic operator replacement

  Each instance of one of the arithmetic operators \{\(+, -, *, /, ^\)\} is replaced with all other possible arithmetic operators.

  Example: \(X = A + B \rightarrow X = A * B\)
• ROR – relational operator replacement

   Each instance of one of the relational operators \{\leq, <, \geq, >, ==, !=\} is replaced with all other possible relational operators.

   Example: \(X \leq Y \rightarrow X > Y\)

• LCR – logical connector replacement

   Each instance of one of the logical connectors \{\&\&, ||\} is replaced with the other possible logical connectors.

   Example: \(A \&\& B \rightarrow A || B\)

• UOI – unary operator insertion

   Each instance of a variable used within a relational expression, on the right side of an assignment, or as an array index is preceded with the unary operator; to avoid duplication of mutants, this is only done if the outcome is not equivalent to an AOR mutant.

   Example: \(X = Y \rightarrow X = -Y\)

• ABS – absolute value insertion

   Each instance of a variable used within a relational expression, on the right side of an assignment, or as an array index is enclosed in an absolute value function call; to avoid equivalent mutants, this is only done if the variable is not already within an absolute value function call and the variable can be a negative value.

   Example: \(X = Y \rightarrow X = \text{ABS}(Y)\)
3.1.3 Requirements for Killing Mutants

To kill a mutant (i.e., distinguish a mutant from the original program), the data state (DS) of the original program and the mutant must be different after the mutated line when executing the same test case. The data state at a certain point of execution is a vector containing the set of variables and predicates that may directly or indirectly affect the final output (a more comprehensive definition of data state can be found in section 3.2.4). To kill a mutant, the test case must satisfy the following three requirements [41-44].

- **Reachability:** If a test case cannot reach the modified statement, it will never detect the difference between P and M. See Figure 117 for example.

- **Necessity:** A test case must create some difference in data state (DSD) between the original program and the mutant immediately after the execution of the mutated statement. There are two types of data state difference. (1) The value of a variable becomes different—this occurs when the modification is in statements which have side effects (Figure 118). These statements are usually “expression” statements that contain assignments. (2) The control flow of the mutant traverses a different branch after the modified statement—this occurs when the modification is on “predicates” (Figure 119).

![Figure 117: The reachability requirement](image-url)
Sufficiency: The data state difference caused by the mutated statement must be able to propagate to the end of the program (Figure 120).

Figure 118 The first type of data state difference

Figure 119 The second type of data state difference

Figure 120 The sufficiency requirement
It is well-known that sufficiency is intractable [42-44]. For this reason, the concept of weakly and strongly killing mutants was introduced [45-50]. In terms of reachability, necessity, and sufficiency, weakly killing mutants and strongly killing mutants can be defined as follows.

1. A mutant is **weakly killed** by a test case if reachability and necessity are satisfied. (Weak Mutation)

2. A mutant is **strongly killed** by a test case if reachability, necessity, and sufficiency are all satisfied. (Strong Mutation)

In this research, we generate test cases superior to weak mutation in the sense that not only the reachability and necessity are satisfied, but the data state difference will be propagated to a location in the code which is as close to the exit of the program as possible. Ultimately the test case will be able to strongly kill mutants.

**3.1.4 Equivalent Mutants**

Occasionally there exists no test case that can cause a data state difference between the original program and a mutant. These mutants are semantically equivalent to the original code and are called **equivalent mutants** (Figure 121). It is also possible that a test case can only cause a local data state difference that can never propagate to the exit of the program. In this research, we call this type of mutants “**can be weakly killed but not strongly killed.**”
3.1.5 Mutation Score

As mentioned in Section 3.1.1 and 3.1.2, the more mutants that can be killed by a test case, the better that test case will be. A measure called mutation score (MS) has been defined to quantify the adequacy of a test case. The formula is,

\[
MS = \frac{\text{no. of mutants killed}}{\text{total no. of mutants} - \text{no. of equivalent mutants}}
\]

The mutation score ranges between 0 and 1.

As an example, assume 100 mutants are generated from a program. They are tested by a test suite. Two mutants are identified as equivalent mutants, three mutants can be weakly killed but not strongly killed, and the other 95 can be strongly killed. The mutation scores of the test suite for weakly kill (WK) and strongly kill (SK) are as follows:

\[
MS(\text{WK}) = \frac{\text{no. of weakly killed mutants}}{\text{total no. of mutants} - \text{no. of equivalent mutants}} = \frac{100 - 2}{100 - 2} = 1
\]

\[
MS(\text{SK}) = \frac{\text{no. of strongly killed mutants}}{\text{total no. of mutants} - \text{no. of equivalent mutants}} = \frac{100 - 2 - 3}{100 - 2} = 0.969
\]

3.1.6 Current State of Mutation Testing

A series of research studies had been conducted to automatically generate test cases able to weakly kill mutants [44, 51-53]. The main idea is to convert the problem of weakly killing
mutants (i.e., satisfying both reachability and necessity) to the problem of enhancing the code coverage of an automatically generated metaprogram (i.e., only satisfy modified reachability requirements, see below in this section). The metaprogram includes both the original program and the constraints corresponding to reachability and necessity. A symbolic execution engine [44, 53] is used to solve the constraints and generate test cases. The problem can then be considered as a type of constraint satisfaction problem. This category of problems has been heavily studied in the literature [54-61].

For example, consider the short sample program in Figure 122 [54].

```
1. int foo (int a) {
2.    if (a > 10) {
3.        return a - b;
4.    } else {
5.        return a;
6.    }
7. }
```

Figure 122 A sample program

If we apply the mutant operator ROR to line 2 of the sample program, we will obtain three mutants: if (a >= 10), if (a == 10), and if (a != 10). (Actually, there should be two more (< and <=) but [44] does not mention them, and we follow their example without any changes.) Similarly, by applying AOR to line 3, another three mutants are obtained. In [44, 51–53], all three mutants are generated; test cases are designed; all the mutants are compiled; and the test cases are run. However, the idea of a metaprogram simplifies this process. Figure 123 provides a metaprogram of the sample program. The metaprogram is the original program plus the necessity constraints. The reachability constraints are not explicitly provided because all the constraints have been inserted such that the reachability constraints are automatically satisfied.
Because the metaprogram already includes the reachability and necessity constraints for the mutants, the goal of designing test cases is transformed into traversing as many branches in the metaprogram as possible, or in other words, to enhancing the code coverage. Symbolic execution [44, 53, 62–68] and concolic execution [69-72] are two commonly used methods that can achieve high code coverage. These two methods run test cases and simultaneously collect information about which branches have been traversed and what the corresponding constraints are. Then they use a constraint solver [73, 74] to solve the new constraints and as such generate test cases that traverse the branches that were not traversed in previous runs. In this way the code coverage achieved can be very high. In addition, because there is only one metaprogram, the compilation time is shorter than that for compiling all mutants.

![Figure 123](image)

Figure 123 The metaprogram of the sample program in Figure 122

If the program is for scientific computation and its size is not too large (i.e., not larger than 1000 lines of code), the above methodology solves the problem of weakly killing mutants. Part of this methodology [44] is adopted in our work to handle the sufficiency requirement. Besides the above methodology, another interesting idea to enhance MS(WK) is using statistical methods
such as clustering analysis to group mutants or test cases. This has been performed by [74–81] to improve the efficiency of mutation testing.

### 3.1.7 Why Strong Mutation instead of Weak Mutation

In this work, we aim to generate test cases for strong mutation because these test cases are more powerful than those created through weak mutation (which may or may not satisfy strong mutation criterion by coincidence). This point can be argued as follows.

Weak mutation (WK) requires that test cases satisfy “Reachability $\land$ Necessity”, while strong mutation (SK) requires “Reachability $\land$ Necessity $\land$ Sufficiency”. Thus the set of all strong mutation test cases is a subset of weak mutation test cases. The different portion, which is usually fairly large for medium to large program units, can be called as “weakly killed but not strongly killed” (WKNSK) test cases. When we conduct weak mutation testing, many test cases will fall into the WKNSK category. These test cases can temporarily distinguish a certain mutant M from the original program P, but not in the final output. However, in software testing the ultimate goal is to distinguish a wrong version from the correct version in the final output. This is because there can be many different ways to implement the same functionality. As long as all these different ways always provide the same final output, they are all considered as correct. Whether their intermediate states are the same is not important at all. The WKNSK test cases, however, cannot distinguish a mutant in the final output, which means that they cannot distinguish an obviously incorrect version of the program, i.e. the mutant M, from the original program P. By just running WKNSK test cases one cannot tell that the mutant M is incorrect, which is an apparently flawed conclusion. (Of course many WK test cases can also strongly kill the mutants by coincidence, which makes WK a valid testing technique). Therefore, in this research we emphasize the Sufficiency constraint to help our test cases strongly kill more mutants, which improves the quality of the test cases.
There are cases where strong mutation is impossible. In such cases, our methodology propagates the DSD further than weak mutation, which we believe is an improvement to the test cases that only weakly kill the mutant. To do so, we investigate a heuristic-based technique that uses constraint solving to identify test cases that will strongly kill mutants.

### 3.2 Problem Analysis and Heuristics for Strongly Killing Mutants

In this section we will first introduce the basic concepts of data state absorption (section 3.2.1 to 3.2.3) and then go through some sample programs (section 3.2.3). After that we will discuss the heuristics defined for the “if” statements (section 3.2.4 and 3.2.5). The heuristics for loops are discussed in section 3.5. All sample programs are written in C language [87] but it will be found that our methodologies can be easily applied to programs written in other languages.

#### 3.2.1 Assumptions about the Programs under Study

We make the following assumptions regarding the programs we will analyze.

1. The programs are for scientific computation: the inputs and outputs are numbers and the functionality is limited to the mathematical processing of data.
2. The programs follow structured programming paradigms and there are no jumps caused by GOTO statements that can lead to “spaghetti code”.

If these assumptions hold, the analysis for the sufficiency requirement will be more straightforward. However, these two assumptions will result in minimal limitations to the analysis. Assumption 1 just follows the nature of scientific software. Assumption 2 is a general recommendation for all software and is followed by many of the emerging software applications.
3.2.2 Modified Control Flow Graph (CFG) for Program Logic Representation

A modified control flow graph (CFG) is developed to represent the program logic. The modified CFG is different from the typical CFG described in reference 82 to 84. This is because the typical CFG is generated after the original program is transformed to three address code, while the modified CFG developed in this research is generated directly from the original program. Due to time limit only a subset of the C specifications [87] are supported, which are listed below:

- The “declaration” statements
- The “expression” statements
- The “block” statements
- The “if” statements
- The “while” statements
- The “do-while” statements
- The “for” statements
- The “return” statements

Each “declaration”, “expression” and “return” statement is a separate line of code and is represented by a node in the modified CFG (Figure 124). The arrows indicate the direction of the flow. The # sign should be replaced by the serial number of a CFG node. “Block” statement is just a sequence of any statements that are bounded by “{}”.
The “if”, “while”, “do-while” and “for” statements usually occupy multiple lines. The corresponding CFG representations are provided in Figure 125, Figure 126, Figure 127 and Figure 128, respectively. For “if” statements the predicates are represented by a separate CFG node “P” because of their importance in the data state difference propagation analysis. The letter “P” in the node should be replaced by the node’s serial number. The “true statement” of the “if” statement is represented as a rounded rectangle. Its inner logic is recursively represented by other CFG nodes. The “false statement” of the “if” statement, if it exists, is represented as a rounded rectangle, too. Its inner logic is also recursively represented by other CFG nodes. A CFG node “E” that represents the end of the “if” statement is introduced to explicitly display the merging point of the flow. The letter “E” in the node should be replaced by the node’s serial number. The flow of the entire statement is indicated by the arrows.
The “while” statement also consists of a “P” node for the predicate, a body statement that should be recursively represented by other CFG nodes, and an “E” node for the end of the statement (Figure 126).

Figure 126 Modified CFG for the "while" statement
The “do-while” statement consists of the “P” node and the body statement (Figure 127) as well. The “P” node works as the end node of the statement. The body statement is recursively represented by other CFG nodes.

**Figure 127** Modified CFG for the "do-while" statement

The “for” statement is displayed in Figure 128. The “initialization”, “test” and “increment” are all expressions, and the “test” expression is the predicate. An end node “E” is also introduced for convenience.

**Figure 128** Modified CFG for the "for" statement
3.2.3 Strongly Killing Mutants: Essentially a Branch-Selection Problem

The following three cases are used to show why strongly killing mutants is essentially a branch-selection problem.

Case 1: There exist programs whose mutants can never be strongly killed

Figure 129 displays a program whose mutants can never be strongly killed. An Entry node and an Exit node are added to the modified CFG for convenience. Node #1 corresponds to line #2 and Node #2 corresponds to line #3. Node #2 is red because it absorbs the data state difference. The data state difference is defined as the difference between the data state of the original program and of the mutant. The data state at a certain line of code is the vector of all variables that directly or indirectly affect the final output of the program. The specific definition of data state is provided in section 3.2.4. As discussed in Section 3.1.2, we only have five selected mutant operators in this study and only three are applicable (AOR, UOI, and ABS) to the code in Figure 129. The other two (ROR and LCR) are not applicable because there are no relational or logical operators in this program.

```c
1   int func (int in1, int in2) {
2       int a = in1 + in2;
3       return 1;
4   }
```

Figure 129 A program whose mutants can never be strongly killed

Only line #2 of the sample program can be mutated and there are only eight mutants for the three mutant operators (listed in Table 21).
### Table 21 List of mutants for the sample program in Figure 129

<table>
<thead>
<tr>
<th>Mutant Number</th>
<th>Mutated Line of Code</th>
<th>Mutant Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>int a = in1 − in2;</td>
<td>AOR</td>
</tr>
<tr>
<td>2</td>
<td>int a = in1 * in2;</td>
<td>AOR</td>
</tr>
<tr>
<td>3</td>
<td>int a = in1 / in2;</td>
<td>AOR</td>
</tr>
<tr>
<td>4</td>
<td>int a = in1 ^ in2;</td>
<td>AOR</td>
</tr>
<tr>
<td>5</td>
<td>int a = −in1 + in2;</td>
<td>UOI</td>
</tr>
<tr>
<td>6</td>
<td>int a = abs(in1) + in2;</td>
<td>ABS</td>
</tr>
<tr>
<td>7</td>
<td>int a = in1 + abs(in2);</td>
<td>ABS</td>
</tr>
<tr>
<td>8</td>
<td>int a = abs(in1 + in2);</td>
<td>ABS</td>
</tr>
</tbody>
</table>

All mutants can be weakly killed by the test case (-5, -2), but none can be strongly killed. This is because there is only one path in the CFG and the data state difference is always absorbed at line #3.

**Case 2**: A program consists of two paths with different propagation characteristics

**Figure 130** displays a sample program whose CFG has two paths. Node #1 (statement #1) and #2 (statement #2) represent the two variable declarations in line #2 and #3. Node #3 represents the predicate of the “if” statement (i.e., statement #3) at line #4. Node #4 (line #5) and #5 (line #7) compose the two branches of the “if”. Node #4 is green because it does not absorb the data state difference, while Node #5 is red because it does absorb the data state difference. Node #6 is the end node of the “if” and does not correspond to any statement. Node #7 (line #9) is for the “return” statement (statement #4).

Consider line #2 and #3. Because there are no relational or logical operators, only AOR, UOI, and ABS are applicable. Similar to Case 1, all mutants of line #2 and #3 can be weakly killed. However, choosing the correct CFG branch is necessary to strongly kill the same mutants.
Because line #7 always returns 1 regardless of the input, if a test case traverses the 1-2-3-5-6 branch for both the original and the mutant programs, the mutant cannot be strongly killed even if it can be weakly killed. In other words, all inputs map to the same output and the data state difference is absorbed. However, if the test case traverses the 1-2-3-4-6 branch for both the original and the mutant programs, strongly killing the mutant is possible. This is because the output range is not a single number but very “large”. In other words, different inputs map to different outputs and the data state difference can be maintained and propagated to the end.

Also possible is that a test case causes the original program to traverse one branch and the mutant program to traverse another. For example, it is possible that the original program traverses the 1-2-3-4-6 branch while the mutant traverses the 1-2-3-5-6 branch. If this occurs, it is possible to strongly kill the mutant because different branches manipulate data in very different ways after the mutated line, thereby maintaining the data state difference.
If the program contains more nodes after Node #6, the data state difference will be propagated. If the test case can always select branches that do not absorb the data state difference, the data state difference will be maintained and propagated to the end of the program, enabling strong mutation.

**Figure 131** illustrates different situations of how the domain of a function maps to its range (or how the local inputs of a statement of a function map to the local output of the same statement).

**Figure 131** Different situations of how the function domain maps to its range

**Figure 131** (a) corresponds to Case 1 and the 1-2-3-5-6 branch of Case 2. Because the entire domain maps to a constant, it is impossible to strongly kill the mutants if both the original and the mutants traverse this branch.

**Figure 131** (b) corresponds to the 1-2-3-4-6 branch of Case 2. If a test case has weakly killed some mutants and both the original and the mutants traverse this branch, the mutants can be strongly killed because the domain of a strictly monotonic function maps to its range one-to-one.

**Figure 131** (c) corresponds to the entire function of Case 2. The function is a non-strictly monotonic function. From the above discussion, it is known that if the test case can weakly kill the mutants and execute branch 1-2-3-4-6, it is easy to strongly kill the mutants. If the test case traverses the 1-2-3-5-6 branch, it is impossible to strongly kill the mutants even if they weakly kill the mutants.
Figure 131 (d) depicts a general situation. The three curves $\overline{AB}$, $\overline{BC}$, and $\overline{DE}$ are all strictly monotonic functions. Therefore, test cases that can weakly kill the mutants will be able to strongly kill the same mutants if they traverse the paths corresponding to these three curves. If the test cases traverse the path corresponding to $\overline{CD}$, they will not be able to do so. Similar analysis can also be found in [85], where the mechanisms of software input failure propagation are studied.

In Case 1 and 2, the mutated lines are “expression” statements with side effects. The value of certain variable(s) is modified. However, these modifications do not affect the flow of the CFG. For this situation we conclude as follows:

- There exist mutants that can be weakly killed but never strongly killed. These mutants are not semantically equivalent to the original code. However, they can never be strongly killed because the program (or branches within the program) can tolerate errors. In other words, inputs from the domain map to the same output for both the original program and the mutants. The data state difference is absorbed.

- For some branches within a program, different inputs map to different outputs (or the local outputs of a certain statement of the CFG). If a test case can weakly kill some mutants, the data state difference will be maintained and propagated to the next statement of the CFG as long as both the original and the mutants traverse that branch.

- Many programs contain multiple branches after the mutated line; some branches absorb the data state difference while others maintain the difference. Test cases should always avoid the branches that absorb the data state difference and instead traverse the branches that maintain the data state difference.

Case 3: A sample program where the mutated line is a predicate
Consider the code from Case 2 but with line #4 mutated. In this case, ROR, UOI, and ABS are applicable; the mutants are listed in **Table 22**.

```c
int func (int in1, int in2) {
    int a = in1 + in2;
    int b = in1 * in2;
    if (a > b) { // This line is mutated
        output = a * b;
    } else {
        output = 1;
    }
    return output;
}
```

**Figure 132** Case 3 sample program and CFG

**Table 22** List of mutants for Case 3

<table>
<thead>
<tr>
<th>Mutant Number</th>
<th>Mutated Line of Code</th>
<th>Mutant Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>if (a &gt;= b) {</td>
<td>ROR</td>
</tr>
<tr>
<td>2</td>
<td>if (a &lt; b) {</td>
<td>ROR</td>
</tr>
<tr>
<td>3</td>
<td>if (a &lt;= b) {</td>
<td>ROR</td>
</tr>
<tr>
<td>4</td>
<td>if (a == b) {</td>
<td>ROR</td>
</tr>
<tr>
<td>5</td>
<td>if (a != b) {</td>
<td>ROR</td>
</tr>
<tr>
<td>6</td>
<td>if (a &gt; - b) {</td>
<td>UOI</td>
</tr>
<tr>
<td>7</td>
<td>if (- a &gt; b) {</td>
<td>UOI</td>
</tr>
<tr>
<td>8</td>
<td>if (a &gt; abs(b)) {</td>
<td>ABS</td>
</tr>
<tr>
<td>9</td>
<td>if (abs(a) &gt; b) {</td>
<td>ABS</td>
</tr>
</tbody>
</table>
To strongly kill the mutants, the test case must cause the original program and the mutants to traverse different branches. This is because mutants can never be strongly killed without being weakly killed first. Since the mutated line is a predicate, “weakly kill” means that the control flow of the mutant should traverse a branch different from the original program after the mutated line. Otherwise it is impossible to distinguish between the predicate expression of the mutant and the original. Similar to the conclusions of Case 1 and 2, the test case should avoid branches that can absorb the data state difference and choose the ones that can maintain it.

3.2.4 Definition of Data State

The data state is a vector of variables and predicates which will directly or indirectly affect the output of a program. It is one of the most important concepts in this research. More specifically, there are four different cases where a variable should be considered as part of the data state:

- If a variable is returned as the output it is considered as directly affecting the output of the program. The variable “output” at line #9 in Figure 132 falls into this category.

- A variable is not a return value and is only visible within the program scope, but is used to calculate the return value directly. It is considered as indirectly affecting the output of the program. The variables “a” and “b” at line #5 in Figure 132 fall into this category.

- A variable is indirectly used to calculate the output. It is also considered as indirectly affecting the output and hence should be included into the data state. The variables “in1” and “in2” at line #2 and line #3 in Figure 132 fall into this category.

- Predicates are never directly used to calculate the output. However, they determine which branches are selected or how many times a loop is executed, which may directly or indirectly relate to the calculation of the output. These predicates are considered as indirectly affecting the output and hence should be included into the data state as well. The predicate “a > b” at
line #4 in Figure 132 falls into this category.

The data state is an attribute of a statement or a predicate. It can be mathematically represented as follows:

\[
\{\text{Variable}_1, \text{Variable}_2, \ldots \text{Variable}_m, \text{Predicate}_1, \text{Predicate}_2, \ldots \text{Predicate}_n, \text{Ret}_1, \\
\text{Ret}_2, \ldots \text{Ret}_k\}
\]

(1)

Where the subscript indicates the serial number of the variables, predicates and returns. Section 3.4.1 provides an algorithm to determine the data state.

The data state difference (DSD) between the original program and the mutant program is the difference between the data state of these two programs when they are running with the same test case. The comparison is made after the same statement is executed. Otherwise the DSs are not comparable. For instance, for the sample program in Figure 132, DS comparison can be conducted after both the original and the mutant execute statement #1 or statement #2. Of course there will not be any DSD since the comparison is before the mutated line (i.e., line #4).

Sometimes the original program and the mutant program take totally different branches. In this case the comparison can be conducted only when (1) the same statement is reached later by both programs, or (2) both programs return an output and exit. For the sample program in Figure 132, assume the predicate at line #4 is mutated from “(a > b)” to “(a < b)”. A test case \{in1==2, in2==3\} is used to execute both the original program and the mutant. For the original program, “(a > b)” is evaluated as false since \{a==5, b==6\} at line #4. Therefore, line #7 of the original program will be executed. However, for the mutant, “(a < b)” is evaluated as true at line #4. Thus line #5 of the mutant will be executed. Because the two programs take different branches, the comparison cannot be conducted at this point. Once line #7 of the original program and line #5 of the mutant program are both executed, the comparison can be made between the “return” statements of both programs, since the two programs once again execute identical statements.
3.2.5 Heuristics to Strongly Kill Mutants

Based on the previous analysis, we developed the following heuristics to strongly kill mutants. These heuristics work with “if” statements. The heuristics for loop statements will be discussed in section 3.5:

1. If the mutated line is located at a statement containing side effects (e.g., an assignment) and the test case can weakly kill the mutant, then there are two methods to maintain and propagate the data state difference to the next statement of the CFG:

   A. Either the original program or the mutant (or both) traverse a branch (in the statement immediately following the one where the mutated line is located) which corresponds to a strictly monotonic function. The branches traversed by the two programs can be the same. (H1A)

   B. The original program and the mutant traverse different branches in the statement immediately following the one where the mutated line is located, no matter if the branches are similar to monotonic functions or not. (H1B)

2. If the mutated line is a predicate, the only method to weakly kill the mutant is to make the original program and the mutant traverse different branches after the mutated line.

   To maintain the propagation of the data state difference through the next statement in the CFG, it is required that:

   A. Either the original program or the mutant (or both) traverse a branch (in the statement immediately following the one where the mutated line is located) which is a strictly monotonic function. The branches traversed by the two programs can be the same. (H2A)
B. The original program and the mutant traverse different branches in the statement immediately following the one where the mutated line is located, no matter if the branches contain monotonic functions or not. (H2B)

3.3 Verification of the Heuristics

3.3.1 The Sample Program for Verification of the Heuristics

We have written several sample programs to verify our heuristics. Due to space limitations we only discuss one in this thesis. The sample program’s CFG is composed of six statements. Two statements contain more than one branch. The sample program is completely procedural and strictly follows the two assumptions described in section 3.2.1. Figure 133 displays the sample program and its CFG.

Statement #1(Nodes #1) and statement #2 (Node #2) are the two declarations in line #2 and #3. Statement #3(Nodes #3 to #13) is the “if” statement from line #4 to line #16. Within this “if” statement, Nodes #4 and #5 correspond to the two “expression” statements in line #5 and #6. Nodes #6 to #9 are for the nested “if” statement from line #7 to line #11. Node #10 is for the predicate of the else-if portion of the outer “if” statement on line #12. Node #11 and #12 are for the “expression” statements on line #13 and line #15, respectively. Node #13 just represents the end of the outer “if” statement. Statement #4 (Node #14) is the “expression” statement on line #17. Statement #5 (Nodes #15 to #20) is the “if” statement from line #18 to #24. Statement #6 (Node #21) is the “return” statement at the end.

18 nodes of the CFG are colored green and 3 are colored red. Green nodes correspond to the case where the DSD is not absorbed; red nodes correspond to the case where the DSD can be absorbed. Based on our heuristics, test cases able to traverse “good” branches (or avoid “bad” branches) would exhibit good performance in terms of data state difference propagation.
1 double f(double x, double y) {
2     double a, b, c, d, output;
3     a = x * x;
4     if (a + x < 1000) {
5         b = 3 * x + y - 2;
6         c = a * b / 5;
7         if (c - b > 0) {
8             d = 10 * x + b;
9         } else {
10             d = 5;
11         }
12     } else if (1000 <= a + x && a + x <= 3000) {
13         d = 20 * x - y;
14     } else {
15         d = 20 * x + y;
16     }
17     d = d * 5;
18     if (d + a < 1000) {
19         output = 500;
20     } else if (1000 <= d + a && d + a <= 5000) {
21         output = d + a - 400;
22     } else {
23         output = 600;
24     }
25     return output;
26 }

**Figure 133** Sample Program for Heuristics Verification
3.3.2 Verification Results for the Sample Program

All five mutant operators are applicable to the sample program. We verified the heuristics for all of them but only provide a portion of the verification results due to space limitations.

1. Mutant #1: an AOR mutant for line #5 (“expression” statement)

   Original: b=3.0*x+y-2.0; Mutated: b=3.0*x*y-2.0.

As previously mentioned, a test case must satisfy the reachability, necessity, and sufficiency conditions/criteria to strongly kill this mutant. The reachability condition requires Node #1, #2, #3 and #4 to be traversed; the necessity condition requires the DS of the original and the mutant to be different immediately after line #5; and the sufficiency condition requires the DSD to propagate to the end. There are 12 paths in this sample program. Some of them have good propagation characteristics and others do not. The branches for each statement are listed in Table 23 and all paths are listed in Table 24. Our focus is on Node #3 to #13 (i.e., line #4 to line #16) and Node #15 to #20 (i.e., line #18 to line #24). Line #17 and line #25 do not absorb the DSD because they are monotonic (or non-constant) functions. Among the possible paths, path #2 should have the best performance because both branches 3-4-5-6-7-9-13 and 15-17-18-20 are similar to monotonic (or non-constant) functions which will map the inputs to the outputs one-to-one. The performance of branches 3-4-5-6-8-9-13, 15-16-20, and 15-17-19-20 is not good because the computation involves constant functions (Figure 131 a)) that absorb the DSD. Note that the branch information in Table 23 is independent of the mutant considered, while the path information in Table 24 is only valid for mutant #1. This is because the branch performance in Table 23 only assesses whether the DSD will propagate through a given statement of the CFG. In other words, it only evaluates the local sufficiency (i.e., local to the statement) requirement but does not assess reachability, necessity, or global sufficiency (i.e., the ability for the DSD to further propagate to the end of the program). The global sufficiency requirement may change.
from mutant to mutant. However, the path performance in Table 24 considers all three requirements and is specific to mutant #1. Therefore, the path performance must be re-analyzed for the other mutants.

**Table 23** Branch performance information for the sample program

<table>
<thead>
<tr>
<th>Statement</th>
<th>Branch</th>
<th>Branch Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>3-4-5-6-7-9-13</td>
<td>Good, the branch is monotonic</td>
</tr>
<tr>
<td>3</td>
<td>3-4-5-6-8-9-13</td>
<td>Bad, line #10 can absorb the DSD</td>
</tr>
<tr>
<td>3</td>
<td>3-10-11-13</td>
<td>Good, the branch is monotonic</td>
</tr>
<tr>
<td>3</td>
<td>3-10-12-13</td>
<td>Good, the branch is monotonic</td>
</tr>
<tr>
<td>5</td>
<td>15-16-20</td>
<td>Bad, line #19 can absorb the DSD</td>
</tr>
<tr>
<td>5</td>
<td>15-17-18-20</td>
<td>Good, the branch is monotonic</td>
</tr>
<tr>
<td>5</td>
<td>15-17-19-20</td>
<td>Bad, line #23 can absorb the DSD</td>
</tr>
</tbody>
</table>

**Table 24** Path performance information for mutant #1

<table>
<thead>
<tr>
<th>#</th>
<th>Path</th>
<th>Path Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1-2-3-4-5-6-7-9-13-14-15-16-20-21</td>
<td>Bad, line #19 absorbs the DSD</td>
</tr>
<tr>
<td>2</td>
<td>1-2-3-4-5-6-7-9-13-14-15-17-18-20-21</td>
<td>Good, the path is monotonic</td>
</tr>
<tr>
<td>3</td>
<td>1-2-3-4-5-6-7-9-13-14-15-17-19-20-21</td>
<td>Bad, line #23 absorbs the DSD</td>
</tr>
<tr>
<td>4</td>
<td>1-2-3-4-5-6-8-9-13-14-15-16-20-21</td>
<td>Bad, line #10 and #19 absorb the DSD</td>
</tr>
<tr>
<td>5</td>
<td>1-2-3-4-5-6-8-9-13-14-15-17-18-20-21</td>
<td>Bad, line #10 absorbs the DSD</td>
</tr>
<tr>
<td>6</td>
<td>1-2-3-4-5-6-7-9-13-14-15-17-19-20-21</td>
<td>Bad, line #10 and #23 absorb the DSD</td>
</tr>
<tr>
<td>7</td>
<td>1-2-3-10-11-13-14-15-16-20-21</td>
<td>Cannot SK, reachability not satisfied</td>
</tr>
<tr>
<td>8</td>
<td>1-2-3-10-11-13-14-15-17-18-20-21</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>1-2-3-10-11-13-14-15-17-19-20-21</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>1-2-3-10-12-13-14-15-16-20-21</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>1-2-3-10-12-13-14-15-17-18-20-21</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>1-2-3-10-12-13-14-15-17-19-20-21</td>
<td></td>
</tr>
</tbody>
</table>
A random test generator is used to generate test cases. Table 25 summarizes the results of eight test cases. Column “x” and “y” list the two inputs; column “Path (O)” and “Path (M)” list the paths traversed by the test cases in the original program and the mutant, respectively. For instance, the number “4” in column Path(O) for Test #1 means that the test case \{x = -13.33, y = -6.12\} traverses path #4 (1-2-3-4-5-6-8-9-13-14-15-16-20-21) of the original program defined in Table 24.

**Table 25** Summary of the verification results for mutant #1

<table>
<thead>
<tr>
<th>Test #</th>
<th>x</th>
<th>y</th>
<th>Path (O)</th>
<th>Path (M)</th>
<th>WK?</th>
<th>SK?</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-13.33</td>
<td>-6.12</td>
<td>4</td>
<td>1</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>2</td>
<td>6.60</td>
<td>-15.67</td>
<td>1</td>
<td>4</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>3</td>
<td>19.70</td>
<td>3.14</td>
<td>2</td>
<td>2</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>4</td>
<td>4.73</td>
<td>18.98</td>
<td>1</td>
<td>2</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>5</td>
<td>15.30</td>
<td>-17.56</td>
<td>2</td>
<td>4</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>6</td>
<td>-19.04</td>
<td>18.46</td>
<td>4</td>
<td>4</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>7</td>
<td>-0.098</td>
<td>2.12</td>
<td>1</td>
<td>1</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>8</td>
<td>-24.31</td>
<td>-33.44</td>
<td>4</td>
<td>3</td>
<td>Y</td>
<td>Y</td>
</tr>
</tbody>
</table>

Table 25 shows that even though all test cases can weakly kill the mutant, only some of them can kill it strongly. Tests #1 and #2 show that the DSD is absorbed by the bad branch 15-16-20 (both path #1 and path #4 share this branch in the “if” statement from line #18 to #24). Tests #3 to #5 show that as long as the test traverses a good branch (in this case, branch 15-17-18-20 of path #2) of either the original program or the mutant, the DSD can be propagated (H1). From Tests #6 and #7 we see that the mutant is not strongly killed if the tests traverse the same bad branch of the original and the mutant programs (again branch 15-16-20). However, Test #8 shows that it is possible to strongly kill the mutant when the test traverses different bad branches of the original and the mutant programs (H2).
2. Mutant #2: an AOR mutant of line #4 (predicate)

Original: if (a+x<1000.0) {; Mutated: if (a/x<1000.0) {.

Because the mutant is located at a predicate, weakly killing the mutant requires that the test case traverses different branches in the original program and the mutants. To propagate the DSD through the following statements, the test case should either traversing good branches (H1) of the original program or the mutant (or both), or it should avoid traversing the same bad branch (H2) for both programs. The random test generator is used again to generate five test cases (Table 26).

<table>
<thead>
<tr>
<th>Test #</th>
<th>x</th>
<th>y</th>
<th>Path (O)</th>
<th>Path (M)</th>
<th>WK?</th>
<th>SK?</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>17.62</td>
<td>-16.13</td>
<td>2</td>
<td>2</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>2</td>
<td>1.52</td>
<td>1.17</td>
<td>4</td>
<td>4</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>3</td>
<td>-10.52</td>
<td>38.89</td>
<td>1</td>
<td>1</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>4</td>
<td>-49.05</td>
<td>31.82</td>
<td>7</td>
<td>5</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>5</td>
<td>-51.25</td>
<td>-44.84</td>
<td>7</td>
<td>5</td>
<td>Y</td>
<td>Y</td>
</tr>
</tbody>
</table>

Tests #1 to #3 demonstrate that different branches after the mutated line are required to weakly kill the mutant, regardless of whether the branch can absorb the DSD. This is because the mutated line is a predicate. Tests #4 and #5 again show that different branches are necessary to weakly kill the mutant; in addition, to propagate the DSD, either H1 or H2 should be satisfied. In Table 23 it can be seen that path #5 (1-2-3-4-5-6-8-9-13-14-15-18-20-21) is good for this mutant because 15-17-18-20 is a good branch in the “if” statement from line #18 to #24. Again, note that the path performance in Table 24 may change from mutant to mutant, but the branch performance in Table 23 remains unchanged. Therefore, based on H1, Tests #4 and #5 are able to strongly kill the mutant.
3. Mutant #3: a ROR mutant of line #4 (predicate)

Original: if (a+x<1000.0) {; Mutated: if (a+x>1000.0) {}.

The random test generator is used to generate the five test cases provided in Table 27.

Table 27 Summary of the verification results for mutant #3

<table>
<thead>
<tr>
<th>Test #</th>
<th>x</th>
<th>y</th>
<th>Path (O)</th>
<th>Path (M)</th>
<th>WK?</th>
<th>SK?</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-43.04</td>
<td>58.61</td>
<td>7</td>
<td>5</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>2</td>
<td>15.21</td>
<td>-10.28</td>
<td>2</td>
<td>11</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>3</td>
<td>-19.24</td>
<td>-31.21</td>
<td>4</td>
<td>10</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>4</td>
<td>-58.63</td>
<td>-10.08</td>
<td>10</td>
<td>5</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>5</td>
<td>1.34</td>
<td>-13.91</td>
<td>1</td>
<td>10</td>
<td>Y</td>
<td>N</td>
</tr>
</tbody>
</table>

As in the previous case, the predicate is mutated. All test cases weakly killed the mutant because they take different branches after the mutated line in the original program and the mutant. However, only Tests #1, #2, and #4 strongly kill the mutant. The reason is that Test #1 takes different branches in the “if” statement from line #18 to #24 for the original program and the mutant (branch 15-16-20 for the original and branch 15-17-18-20 for the mutant, respectively). Based on H2, Test #1 is able to strongly kill the mutant. For the same reason, Test #4 also strongly kills the mutant. Test #2 traverses the same branch 15-17-18-20 in the “if” statement for both the original program and the mutant. According to Table 23, branch 15-17-18-20 is able to propagate the DSD and hence Test #2 strongly kills the mutant (H1). Tests #3 and #5 demonstrate that to propagate the DSD, the bad branches after the mutated line must be avoided (H2). More specifically, both Test #3 and Test #5 take branch 15-16-20 in the “if” statement for the original program and mutant (see Table 23). However, this branch absorbs the DSD.
We also verified the heuristics for the other three mutant operators LOR, UOI, and ABS, and the heuristics work well for them all. However, due to space limitations these results are not presented in this thesis.

3.4 An Algorithm to Implement the Heuristics

3.4.1 An Algorithm to Determine the Data State

As introduced in section 3.2.4, the data state is a vector of variables and predicates which will directly or indirectly affect the program output. Because the data state of the original program and the mutant are compared at the same statements or predicates, the data state is defined as an attribute of statements and predicates. There are two major steps in the algorithm to determine the data state, both of which can be conducted statically:

1. Construct a dependency graph which describes if a variable/predicate at a certain statement can eventually affect the program output.
2. Based on the dependency graph, traverse each statement of the parse tree recursively and determine the data state.

3.4.1.1 Construct the Dependency Graph

Figure 134 describes the algorithm to construct the dependency graph. It provides rules about how to handle different types of statements that are currently supported. The dependency graph built from the algorithm contains three types of nodes: variable nodes, predicate nodes and return nodes. A node is a pair of <variable name, statement number> (see Figure 136 for more information). New variable nodes are mostly introduced in the formal parameter list and “declaration” statements; Some “for” statements also declare new variables, and new variable nodes are introduced correspondingly. “Expression” statements may have side effects and new
variable nodes are introduced for those variables that are updated. New predicate nodes (PredNode#) are introduced in “if”, “while”, “do-while” and “for” statements, because these statements all use predicates to determine the direction of the control flow or the number of iterations. For each “return” statement, a new return node (RetNode#) is introduced correspondingly. All nodes in the dependency graph potentially compose the data state of a certain statement/predicate.

The links between the nodes are constructed based on the following rules:

1. Expressions and predicates may have side effects. I.e., some variables are updated when the expressions/predicates are executed. Links should be built from the variables whose value is read to the variables whose value is written.

2. Each predicate is assigned a new predicate node (i.e., PredNode# in Figure 134). Links should be built from the variables of a predicate to the PredNode#.

3. The predicate of the “if” statement decides whether the true body or the false body will be executed, thus it also affects the variables that are updated within the true body and the false body. More specifically, the following sub-rules for each type of statement should be recursively applied.
   a. If the body is a “declaration” statement or an “expression” statement, a link should be built from the PredNode# to the variables that are declared or updated.
   b. If the body is a “block” statement, the predicate affects each statement of the “block”. The specific way that the predicate affects the statements within in a “block” depends on the statement type.
   c. If the body is an “if” statement or a “while” statement, a link should be built from the PredNode# to the predicate of the nested “if” or “while”, but not to the variables in the body statements of the nested “if” or “while”. This is because the predicate of the outer
“if” cannot directly affect the variables within the body of the nested “if” or “while”.

d. If the body is a “do-while” statement, the predicate affects the body statement depending on its type. A link should also be built from the PredNode# to the predicate of the “do-while”.

e. If the body is a “for” statement, the predicate affects the “initialization”, “test” and “increment”. However, it does not affect the variables in the body of the “for”.

f. If the body is a “return” statement, a link should be built from the PredNode# to the RetNode# for the “return”.

4. The way that the predicate of the “while” affects the body statement is the same as that of “if”. Thus the rule of building links is the same as rule 3.

5. Although the predicate of the “do-while” statement is executed after the first execution of the body statement, it decides whether or not there will be further iterations. Thus it may still affect the body statement. Therefore, the same rule as “while” should be applied.

6. The predicate of the “for” statement (i.e., the “test” expression) affects the body statement as well as the “increment” expression (see Figure 128). Thus links should be built correspondingly.

7. Each “return” statement is assigned a return node “RetNode#” (Figure 134). Links should be built from the variables of a “return” statement to the RetNode#.

The algorithm calls many functions that are introduced in Appendix C.1.
Algorithm: Dependency graph construction

Input:
The parse tree of the source code
The dependency graph. Initially it is empty
The current statement/predicate

Output:
The dependency graph which describes if a variable/predicate at a certain statement can eventually affect the program output. If so, how this variable/predicate affects the program output.

1. DependencyGraphConstruction(Graph, CurrStmt)
2. assignStmtNumber(CurrStmt)
3. if (CurrStmt == FORMAL)
4. for each Formal in CurrStmt
5. FormalNode ← genNewNode(Formal)
6. addNewNode(Graph, FormalNode)
7. end for
8. else if (CurrStmt == DECLARATION)
9. AllDeclVar ← getAllDeclVar(CurrStmt)
10. for each DeclVar in AllDeclVar
11. DeclNode ← genNewNode(DeclVar)
12. addNewNode(Graph, DeclNode)
13. end for
14. DeclExpr ← getDeclExpr(CurrStmt)
15. if (DeclExpr != NULL)
16. for each DeclVar in AllDeclVar
17. AllReadVar ← getAllReadVar(DeclVar, DeclExpr)
18. for each ReadVar in AllReadVar
19. buildLink(getNode(Graph, ReadVar), getNode(Graph, DeclVar))
20. end for
21. end for
22. recursively call DependencyGraphConstruction(Graph, DeclExpr)
23. end if
24. else if (CurrStmt == EXPRESSION)
25. AllModVar ← getAllModVar(CurrStmt)
26. for each ModVar in AllModVar
27. ModNode ← genNewNode(ModVar)
28. addNewNode(Graph, ModNode)
29. AllReadVar ← getAllReadVar(ModVar, CurrStmt)
30. for each ReadVar in AllReadVar
31. buildLink(getNode(Graph, ReadVar), getNode(Graph, ModVar))
32. end for
33. end for
34. else if (CurrStmt == BLOCK)
35. AllStmt ← getAllStmt(CurrStmt)
36. for each Stmt in AllStmt
37. recursively call DependencyGraphConstruction(Graph, Stmt)
38. end for
39. else if (CurrStmt == IF)
40. Pred ← getPred(CurrStmt)

Figure 134 An algorithm to construct the dependency graph
Figure 134 An algorithm to construct the dependency graph (continue)

```plaintext
41  PredNode ← genNewNode(Pred)
42  addNewNode(Graph, PredNode)
43  AllReadVar ← getAllReadVar(Pred)
44  for each ReadVar in AllReadVar
45      buildLink(getNode(Graph, ReadVar), PredNode)
46  end for
47  recursively call DependencyGraphConstruction(Graph, Pred)
48  TrueStmt ← getTrueStmt(CurrStmt)
49  AllAffectVar1 ← getAllAffectVarFromStmt(TrueStmt)
50  for each AffectVar1 in AllAffectVar1
51      AffectNode1 ← genNewNode(AffectVar1)
52      addNewNode(Graph, AffectNode1)
53      buildLink(PredNode, AffectNode1)
54  end for
55  recursively call DependencyGraphConstruction(Graph, TrueStmt)
56  FalseStmt ← getFalseStmt(CurrStmt)
57  if (FalseStmt != NULL)
58      AllAffectVar2 ← getAllAffectVarFromStmt(FalseStmt)
59      for each AffectVar2 in AllAffectVar2
60          AffectNode2 ← genNewNode(AffectVar2)
61          addNewNode(Graph, AffectNode2)
62          buildLink(PredNode, AffectNode2)
63  end for
64  recursively call DependencyGraphConstruction(Graph, FalseStmt)
65  end if
66  else if (CurrStmt == WHILE)
67      Pred ← getPred(CurrStmt)
68      PredNode ← genNewNode(Pred)
69      AddNewNode(Graph, PredNode)
70      AllReadVar ← getAllReadVar(Pred)
71      for each ReadVar in AllReadVar
72          buildLink(getNode(Graph, ReadVar), PredNode)
73  end for
74  recursively call DependencyGraphConstruction(Graph, Pred)
75  WhileBodyStmt ← getWhileBodyStmt(CurrStmt)
76  AllAffectVar ← getAllAffectVarFromStmt(WhileBodyStmt)
77  for each AffectVar in AllAffectVar
78      AffectNode ← genNewNode(AffectVar)
79      addNewNode(Graph, AffectNode)
80      buildLink(PredNode, AffectNode)
81  end for
82  recursively call DependencyGraphConstruction(Graph, WhileBodyStmt)
83  else if (CurrStmt == DOWHILE)
84      DoWhileBodyStmt ← getDoWhileBodyStmt(CurrStmt)
85  recursively call DependencyGraphConstruction(Graph, DoWhileBodyStmt)
86      Pred ← getPred(CurrStmt)
87      PredNode ← genNewNode(Pred)
88      AddNewNode(Graph, PredNode)
89      AllReadVar ← getAllReadVar(Pred)
90  for each ReadVar in AllReadVar
91  end for
```

Continue...
Figure 134 An algorithm to construct the dependency graph (continue)

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>91</td>
<td><code>buildLink(getNode(Graph, ReadVar), PredNode)</code></td>
</tr>
<tr>
<td>92</td>
<td><code>end for</code></td>
</tr>
<tr>
<td>93</td>
<td><code>AllAffectVar ← getAllAffectVarFromStmt(DoWhileBodyStmt)</code></td>
</tr>
<tr>
<td>94</td>
<td><code>for each AffectVar in AllAffectVar</code></td>
</tr>
<tr>
<td>95</td>
<td><code>AffectNode ← genNewNode(AffectVar)</code></td>
</tr>
<tr>
<td>96</td>
<td><code>addNewNode(Graph, AffectNode)</code></td>
</tr>
<tr>
<td>97</td>
<td><code>buildLink(PredNode, AffectNode)</code></td>
</tr>
<tr>
<td>98</td>
<td><code>end for</code></td>
</tr>
<tr>
<td>99</td>
<td><code>recursively call</code> DependencyGraphConstruction(Graph, Pred)`</td>
</tr>
<tr>
<td>100</td>
<td><code>else if (CurrStmt == FOR)</code></td>
</tr>
<tr>
<td>101</td>
<td><code>Init ← getInit(CurrStmt)</code></td>
</tr>
<tr>
<td>102</td>
<td><code>recursively call</code> DependencyGraphConstruction(Graph, Init)`</td>
</tr>
<tr>
<td>103</td>
<td><code>Pred ← getTest(CurrStmt)</code></td>
</tr>
<tr>
<td>104</td>
<td><code>recursively call</code> DependencyGraphConstruction(Graph, Pred)`</td>
</tr>
<tr>
<td>105</td>
<td><code>Incr ← getIncr(CurrStmt)</code></td>
</tr>
<tr>
<td>106</td>
<td><code>recursively call</code> DependencyGraphConstruction(Graph, Incr)`</td>
</tr>
<tr>
<td>107</td>
<td><code>PredNode ← genNewNode(Pred)</code></td>
</tr>
<tr>
<td>108</td>
<td><code>addNewNode(Graph, PredNode)</code></td>
</tr>
<tr>
<td>109</td>
<td><code>AllReadVar ← getAllReadVar(Pred)</code></td>
</tr>
<tr>
<td>110</td>
<td><code>for each ReadVar in AllReadVar</code></td>
</tr>
<tr>
<td>111</td>
<td><code>buildLink(getNode(Graph, ReadVar), PredNode)</code></td>
</tr>
<tr>
<td>112</td>
<td><code>end for</code></td>
</tr>
<tr>
<td>113</td>
<td><code>ForBodyStmt ← getForBodyStmt(CurrStmt)</code></td>
</tr>
<tr>
<td>114</td>
<td><code>AllAffectVar ← getAllAffectVarFromStmt(ForBodyStmt)</code></td>
</tr>
<tr>
<td>115</td>
<td><code>for each AffectVar in AllAffectVar</code></td>
</tr>
<tr>
<td>116</td>
<td><code>AffectNode ← genNewNode(AffectVar)</code></td>
</tr>
<tr>
<td>117</td>
<td><code>addNewNode(Graph, AffectNode)</code></td>
</tr>
<tr>
<td>118</td>
<td><code>buildLink(PredNode, AffectNode)</code></td>
</tr>
<tr>
<td>119</td>
<td><code>end for</code></td>
</tr>
<tr>
<td>120</td>
<td><code>recursively call</code> DependencyGraphConstruction(Graph, ForBodyStmt)`</td>
</tr>
<tr>
<td>121</td>
<td><code>else if (CurrStmt == RETURN)</code></td>
</tr>
<tr>
<td>122</td>
<td><code>RetExpr ← getRet(CurrStmt)</code></td>
</tr>
<tr>
<td>123</td>
<td><code>RetNode ← genNewNode(RetExpr)</code></td>
</tr>
<tr>
<td>124</td>
<td><code>addNewNode(Graph, RetNode)</code></td>
</tr>
<tr>
<td>125</td>
<td><code>AllReadVar ← getAllReadVar(RetExpr)</code></td>
</tr>
<tr>
<td>126</td>
<td><code>for each ReadVar in AllReadVar</code></td>
</tr>
<tr>
<td>127</td>
<td><code>buildLink(getNode(Graph, ReadVar), RetNode)</code></td>
</tr>
<tr>
<td>128</td>
<td><code>end for</code></td>
</tr>
<tr>
<td>129</td>
<td><code>AllModVar ← getAllModVar(CurrStmt)</code></td>
</tr>
<tr>
<td>130</td>
<td><code>for each ModVar in AllModVar</code></td>
</tr>
<tr>
<td>131</td>
<td><code>ModNode ← genNewNode(ModVar)</code></td>
</tr>
<tr>
<td>132</td>
<td><code>addNewNode(Graph, ModNode)</code></td>
</tr>
<tr>
<td>133</td>
<td><code>AllReadVar ← getAllReadVar(ModVar, CurrStmt)</code></td>
</tr>
<tr>
<td>134</td>
<td><code>for each ReadVar in AllReadVar</code></td>
</tr>
<tr>
<td>135</td>
<td><code>buildLink(getNode(Graph, ReadVar), ModNode)</code></td>
</tr>
<tr>
<td>136</td>
<td><code>end for</code></td>
</tr>
<tr>
<td>137</td>
<td><code>end if</code></td>
</tr>
<tr>
<td>138</td>
<td><code>else</code></td>
</tr>
<tr>
<td>139</td>
<td><code>print(&quot;Unsupported statement!&quot;)</code></td>
</tr>
<tr>
<td>140</td>
<td><code>end if</code></td>
</tr>
<tr>
<td>141</td>
<td><code>return Graph</code></td>
</tr>
</tbody>
</table>
Figure 135 provides a sample program where each statement/predicate is already given a serial number. Each serial number begins with a letter indicating the category of the statement/predicate. The digits after the letter indicate the sequence and depth of a statement/predicate in the parse tree. For instance, for the first line of the program, “F” indicates that this line is a formal parameter list, and “0” means that this is the very first piece of code to be considered. At line #3 there are two serial numbers, “I2” and “P200”, where the former is for the entire “if” statement and the latter is for the predicate of the “if”. The numbering reflects the recursive nature of the parse tree, and the specific rules are not provided in this thesis due to space limitations.

The dependency graph for this sample program is displayed in Figure 136. It is constructed based on the algorithm in Figure 134. The variables that are from the formal parameter list or are declared are colored yellow, which means they are the starting points of the dependency graph. The “return” node is colored green, which means that it is the end point of the dependency graph. The arrows display the direction of the dependencies between the nodes.
double f(double x, double y) { // F0
    double a=0.0, b=0.0, output=0.0; // D1
    if (x+y < 10) // I2; P2-0-0
        a = 5*y; // E2-0-0
    else
        if(10 <= x+y && x+y <= 30) // I2-1-0; P2-1-0-0-0
            a = 10*y; // E2-1-0-0-0
        else
            a = 15*y; // E2-1-0-1-0
    b = a+10; // E3
    if (b+x <= 5) // I4; P4-0-0
        output = 3*b; // E4-0-0
    else
        if(5 < b+x && b+x <= 10) // I4-1-0; P4-1-0-0-0
            output = b*b; // E4-1-0-0-0
        else
            if(10 < b+x && b+x <= 20) // I4-1-0-1-0; P4-1-0-1-0-0-0
                output = 1; // E4-1-0-1-0-0-0
            else
                output = 0; // E4-1-0-1-0-1-0
    return output; // R5
}
3.4.1.2 Determine the Data State based on the Dependency Graph

The determination of the data state is closely related to the dependency graph. This is because the dependency graph not only provides information about which variables directly/indirectly affect the program output but also how this process takes place. Certain variables can only affect the program output through other variables, and thus they only exist in the data state temporarily.

Four attributes “IN”, “GEN”, “KILL” and “OUT” are introduced for each statement/predicate to represent the data state transformation among statements/predicates. The definition of these attributes is provided below:

1. “IN” is the data state coming into this statement/predicate.
2. “GEN” is the set of nodes that should be integrated into the data state. These nodes can be from new variables declared, variables that are updated, predicates or “return” statements.
3. “KILL” is the set of nodes that should be removed from the data state. For non-loop statements including “declaration”, “expression”, “block”, “if” and “return”, these nodes will not further affect the program output but their effects are entirely transferred to other nodes. For loop statements including “while”, “do-while” and “for”, this attribute is further divided into LocalKILL and GlobalKILL. LocalKILL is the set of variables that do not affect the program output from the next iteration of the loop on, while GlobalKILL represents the variables that do not affect the program output from the next statement after the loop on. LocalKILL relates to the LocalOUT of loop statements and GlobalKILL relates to the OUT of loop statements (see the next paragraph).

4. “OUT” represents the data state after this statement/predicate is executed. For non-loop statements and predicates, it is the actual data state of this statement. For loop statements, the data state can be further divided into local data state (represented by LocalOUT) and global data state (represented by OUT). The reason is that some of the variables only propagate the data state difference to the next iteration of the loop but not after the loop statement. These variables are included in LocalOUT but not OUT. The LocalOUT corresponds to the local data state (local DS) of the loop statements and OUT corresponds to the global data state (global DS, see below for more details).

The algorithm for data state calculation investigates the relationship among these four attributes for each type of statement/predicate. Figure 137 provides the specific algorithm. Note that the four attributes “IN”, “GEN”, “KILL” and “OUT” are renamed in the algorithm to precisely describe the specific meaning of the attribute in the context.
Algorithm: Data state calculation

Input:
- The parse tree of the source code
- The dependency graph of the program
- The data state of the statement/predicate preceding the current statement
- The current statement

Output:
- The data state vector of each statement/predicate.

1. DataStateCalculation(Graph, PrevDS, CurrStmt)
2. if (CurrStmt == FORMAL)
3.   IN ← PrevDS
4.   GEN ← getAllFormalNodes(Graph)
5.   KILL ← getAllKillNodes(Graph, CurrStmt)
6.   OUT ← (IN \ KILL) \ U \ GEN
7. else if (CurrStmt == DECLARATION)
8.   IN ← PrevDS
9.   GEN ← getAllDeclNodes(Graph, CurrStmt)
10. DeclExpr ← getDeclExpr(CurrStmt)
11. if (DeclExpr != NULL)
12.   GEN ← GEN \ U \ getAllModNodes(Graph, DeclExpr)
13. end if
14. KILL ← getAllKillNodes(Graph, CurrStmt)
15. OUT ← (IN \ KILL) \ U \ GEN
16. else if (CurrStmt == EXPRESSION)
17.   IN ← PrevDS
18.   GEN ← getAllModNodes(Graph, CurrStmt)
19. KILL ← getAllKillNodes(Graph, CurrStmt)
20. OUT ← (IN \ KILL) \ U \ GEN
21. else if (CurrStmt == BLOCK)
22.   AllStmt ← getAllStmt(CurrStmt)
23.   OUT ← recursively call DataStateCalculation(Graph, PrevDS, AllStmt[1])
24. for each Stmt except AllStmt[1] in AllStmt
25.   OUT ← recursively call DataStateCalculation(Graph, OUT, Stmt)
26. end for
27. else if (CurrStmt == IF)
28.   Pred ← getPred(CurrStmt)
29.   PredIN ← PrevDS
30.   PredGEN ← getPredNode(Graph, Pred)
31.   PredGEN ← PredGEN \ U \ getAllModNodes(Graph, Pred)
32.   PredKILL ← getAllKillNodes(Graph, Pred)
33.   PredOUT ← (PredIN \ PredKILL) \ U \ PredGEN
34.   TrueStmt ← getTrueStmt(CurrStmt)
35.   TrueOUT ← recursively call DataStateCalculation(Graph, PredOUT, TrueStmt)
36.   FalseStmt ← getFalseStmt(CurrStmt)
37.   if (FalseStmt != NULL)
38.     FalseOUT ← recursively call DataStateCalculation(Graph, PredOUT, FalseStmt)
39.   end if
40.   OUT ← TrueOUT \ U \ FalseOUT

Figure 137 An algorithm for data state calculation
An algorithm for data state calculation (continue)

```plaintext
else if (CurrStmt == WHILE)
    Pred ← getPred(CurrStmt)
    PredIN ← PrevDS
    PredGEN ← getPredNode(Graph, Pred)
    PredGEN ← PredGEN U getAllModNodes(Graph, Pred)
    PredKILL ← getAllKillNodes(Graph, Pred)
    PredOUT ← (PredIN \ PredKILL) U PredGEN
    BodyStmt ← getWhileBodyStmt(CurrStmt)
    BodyOUT ← recursively call DataStateCalculation(Graph, PredOUT, BodyStmt)
    LocalKILL ← getAllLocalKillNodes(Graph, BodyStmt)
    LocalOUT ← BodyOUT \ LocalKILL
    GlobalKILL ← getAllKillNodes(Graph, BodyStmt)
    OUT ← BodyOUT \ GlobalKILL
else if (CurrStmt == DOWHILE)
    BodyStmt ← getDoWhileBodyStmt(CurrStmt)
    BodyOUT ← recursively call DataStateCalculation(Graph, PrevDS, BodyStmt)
    Pred ← getPred(CurrStmt)
    PredIN ← BodyOUT
    PredGEN ← getPredNode(Graph, Pred)
    PredGEN ← PredGEN U getAllModNodes(Graph, Pred)
    PredLocalKILL ← getAllLocalKillNodes(Graph, Pred)
    LocalOUT ← (PredIN \ PredLocalKILL) U PredGEN
    PredGlobalKILL ← getAllKillNodes(Graph, Pred)
    OUT ← (PredIN \ PredGlobalKILL) U PredGEN
else if (CurrStmt == FOR)
    Init ← getInit(CurrStmt)
    InitIN ← PrevDS
    InitGEN ← getInitNode(Graph, Init)
    InitGEN ← InitGEN U getAllModNodes(Graph, Init)
    InitKILL ← getAllKillNodes(Graph, Init)
    InitOUT ← (InitIN \ InitKILL) U InitGEN
    Pred ← getTest(CurrStmt)
    PredIN ← InitOUT
    PredGEN ← getPredNode(Graph, Pred)
    PredGEN ← PredGEN U getAllModNodes(Graph, Pred)
    PredKILL ← getAllKillNodes(Graph, Pred)
    PredOUT ← (PredIN \ PredKILL) U PredGEN
    BodyStmt ← getForBodyStmt(CurrStmt)
    BodyOUT ← recursively call DataStateCalculation(Graph, PredOUT, BodyStmt)
    Incr ← getIncr(CurrStmt)
    IncrIN ← BodyOUT
    IncrGEN ← getIncrNode(Graph, Incr)
    IncrLocalKILL ← getAllLocalKillNodes(Graph, Incr)
    LocalOUT ← (IncrIN \ IncrLocalKILL) U IncrGEN
    IncrGlobalKILL ← getAllKillNodes(Graph, Incr)
    OUT ← (IncrIN \ IncrGlobalKILL) U IncrGEN
else if (CurrStmt == RETURN)
    IN ← PrevDS
    GEN ← getRetNode(Graph, CurrStmt)
    KILL ← getAllKillNodes(Graph, CurrStmt)
    OUT ← (IN \ KILL) U GEN
return OUT
```
The algorithm defines how the four attributes for each statement/predicate can be calculated.

More specifically:

1. For all statements/predicates, attribute “IN” is equal to the “OUT” of the statement/predicate that is logically preceding the statement/predicate under consideration.

2. Attribute “GEN” for the formal list, the “declaration” statements and those “for” statements that declare new variables contain the nodes for the newly declared variables. “Expression” statements or the expressions of all declarations and predicates may have side effects, and the variables that are updated will be included into “GEN” as well.

3. Attribute “KILL” that corresponds to the data state of non-loop statements and the global data state of the loop statements is determined using the functions getAllKillNodes(Graph, Stmt). This function traverses the dependency graph “Graph” and determines the statements at which a certain variable’s effect on the program output is entirely transferred to other variables. This variable should then be killed at these statements. This is because this variable will not affect the program output in any ways after these statements are executed. For instance, the variable “a” in the sample program in Figure 135 should be killed at statement “E3”. This is because all the nodes of “a” (i.e., <a, E200>, <a, E21000>, <a, E21010> and <a, D1>) point to the node <b, E3> (Figure 136), which is the only node that leads the nodes of “a” to <Ret1, R5>.

4. Attribute “KILL” that corresponds to the local data state of the loop statements is determined using the functions getAllLocalKillNodes(Graph, Stmt). This function works in a very similar manner to getAllKillNodes(Graph, Stmt). However, it keeps the variables that only affect the program outputs in the following iterations of the loop but not into the next statement. Note that these variables are all removed in getAllKillNodes(Graph, Stmt).

5. Attribute “OUT” for the formal list, the “declaration” statements, the “expression” statements
and the predicates is calculated using the formula \( \text{OUT} = (\text{IN} \setminus \text{KILL}) \cup \text{GEN} \). For the other statements the algorithm should be recursively applied.

6. As already discussed in the previous paragraphs, the loop statements (i.e., “while”, “do-while” and “for” statements) have two data states: “local DS” and “global DS”. The “local DS” contains some of the variables which affect the predicate (and hence the number of iterations) but do not propagate the DSD to the succeeding statements. The “local DS” is equal to the “LocalOUT” in the algorithm in Figure 137. The “global DS” only contains variables that propagate the DSD to the following statements and is equal to the “OUT” in the algorithm.

Appendix C.2 provides the descriptions of the functions used in the algorithm.

For the sample program in Figure 135, the results from all calls to getAllKillNodes(Graph, Stmt) are provided in Table 28. Most variables will be killed multiple times because the control flow splits in the “if” statements. The total number of times a variable is killed is equal to the total number of entries in the corresponding row of the table.

Table 28 Statements where a certain variable should be killed from the data state vector

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Statements where the variable should be killed from the data state vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>P400, P41000, P410100</td>
</tr>
<tr>
<td>y</td>
<td>E200, E21000, E21010</td>
</tr>
<tr>
<td>a</td>
<td>E3</td>
</tr>
<tr>
<td>b</td>
<td>E400, E41000, P410100</td>
</tr>
<tr>
<td>output</td>
<td>R5</td>
</tr>
<tr>
<td>pred1</td>
<td>E200, P21000</td>
</tr>
<tr>
<td>pred2</td>
<td>E21000, E21010</td>
</tr>
<tr>
<td>pred3</td>
<td>E400, P41000</td>
</tr>
<tr>
<td>pred4</td>
<td>E41000, P410100</td>
</tr>
<tr>
<td>pred5</td>
<td>E4101000, E4101010</td>
</tr>
</tbody>
</table>

Table 29 summarizes the data state calculation for the sample program in Figure 135. Some nodes have a superscript value which indicates the total number of times this node has already been killed, e.g., the “\(<\text{pred1, P200}>^{(1)}\) in the “KILL” cell for statement “E200”. Also note that
the node “<x, F0>(1)” in the “OUT” cell for statement “P400” is in parenthesis. This means that “<x, F0>(1)” is not part of the data state of “P400” but is put aside for the calculation for the other predicates “P41000” and “P4101000”.

Table 29 Data state calculation for the sample program

<table>
<thead>
<tr>
<th>Statement</th>
<th>IN</th>
<th>GEN</th>
<th>KILL</th>
<th>OUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>F0</td>
<td>Ø</td>
<td>&lt;x, F0&gt;, &lt;y, F0&gt;</td>
<td>Ø</td>
<td>&lt;x, F0&gt;, &lt;y, F0&gt;</td>
</tr>
<tr>
<td>D1</td>
<td>&lt;x, F0&gt;, &lt;y, F0&gt;</td>
<td>&lt;a, D1&gt;, &lt;b, D1&gt;, &lt;output, D1&gt;</td>
<td>Ø</td>
<td>&lt;x, F0&gt;, &lt;y, F0&gt;, &lt;a, D1&gt;, &lt;b, D1&gt;, &lt;output, D1&gt;</td>
</tr>
<tr>
<td>P200</td>
<td>&lt;x, F0&gt;, &lt;y, F0&gt;, &lt;a, D1&gt;, &lt;b, D1&gt;, &lt;output, D1&gt;</td>
<td>&lt;pred1, P200&gt;</td>
<td>Ø</td>
<td>&lt;x, F0&gt;, &lt;y, F0&gt;, &lt;a, D1&gt;, &lt;b, D1&gt;, &lt;output, D1&gt;, &lt;pred1, P200&gt;</td>
</tr>
<tr>
<td>E200</td>
<td>&lt;x, F0&gt;, &lt;y, F0&gt;, &lt;a, D1&gt;, &lt;b, D1&gt;, &lt;output, D1&gt;, &lt;pred1, P200&gt;</td>
<td>&lt;a, E200&gt;</td>
<td>&lt;pred1, P200&gt;, &lt;y, F0&gt;</td>
<td>&lt;x, F0&gt;, &lt;a, D1&gt;, &lt;b, D1&gt;, &lt;output, D1&gt;, &lt;a, E200&gt;</td>
</tr>
<tr>
<td>P21000</td>
<td>&lt;x, F0&gt;, &lt;y, F0&gt;, &lt;a, D1&gt;, &lt;b, D1&gt;, &lt;output, D1&gt;, &lt;pred1, P200&gt;</td>
<td>&lt;pred2, P21000&gt;</td>
<td>&lt;pred1, P200&gt;, &lt;y, F0&gt;</td>
<td>&lt;x, F0&gt;, &lt;y, F0&gt;, &lt;a, D1&gt;, &lt;b, D1&gt;, &lt;output, D1&gt;, &lt;a, E200&gt;</td>
</tr>
<tr>
<td>E21000</td>
<td>&lt;x, F0&gt;, &lt;y, F0&gt;, &lt;a, D1&gt;, &lt;b, D1&gt;, &lt;output, D1&gt;, &lt;pred2, P21000&gt;</td>
<td>&lt;a, E21000&gt;</td>
<td>&lt;pred2, P21000&gt;, &lt;y, F0&gt;</td>
<td>&lt;x, F0&gt;, &lt;a, D1&gt;, &lt;b, D1&gt;, &lt;output, D1&gt;, &lt;a, E200&gt;</td>
</tr>
<tr>
<td>E21010</td>
<td>&lt;x, F0&gt;, &lt;y, F0&gt;, &lt;a, D1&gt;, &lt;b, D1&gt;, &lt;output, D1&gt;, &lt;pred2, P21000&gt;</td>
<td>&lt;a, E21010&gt;</td>
<td>&lt;pred2, P21000&gt;, &lt;y, F0&gt;</td>
<td>&lt;x, F0&gt;, &lt;a, D1&gt;, &lt;b, D1&gt;, &lt;output, D1&gt;, &lt;a, E21000&gt;</td>
</tr>
<tr>
<td>I210</td>
<td>&lt;x, F0&gt;, &lt;y, F0&gt;, &lt;a, D1&gt;, &lt;b, D1&gt;, &lt;output, D1&gt;, &lt;pred1, P200&gt;</td>
<td>Do not need to calculate</td>
<td>Do not need to calculate</td>
<td>&lt;x, F0&gt;, &lt;a, D1&gt;, &lt;b, D1&gt;, &lt;output, D1&gt;, &lt;a, E21000&gt;, &lt;a, E21010&gt;</td>
</tr>
<tr>
<td>I2</td>
<td>&lt;x, F0&gt;, &lt;y, F0&gt;, &lt;a, D1&gt;, &lt;b, D1&gt;, &lt;output, D1&gt;, &lt;pred1, P200&gt;</td>
<td>Do not need to calculate</td>
<td>Do not need to calculate</td>
<td>&lt;x, F0&gt;, &lt;a, D1&gt;, &lt;b, D1&gt;, &lt;output, D1&gt;, &lt;a, E200&gt;, &lt;a, E21000&gt;, &lt;a, E21010&gt;</td>
</tr>
<tr>
<td>E3</td>
<td>&lt;x, F0&gt;, &lt;a, D1&gt;, &lt;b, D1&gt;, &lt;output, D1&gt;, &lt;a, E200&gt;, &lt;a, E21000&gt;, &lt;a, E21010&gt;</td>
<td>&lt;b, E3&gt;</td>
<td>&lt;a, D1&gt;(1), &lt;a, E200&gt;(1), &lt;a, E21000&gt;(1), &lt;a, E21010&gt;(1)</td>
<td>&lt;x, F0&gt;, &lt;b, D1&gt;, &lt;output, D1&gt;, &lt;b, E3&gt;</td>
</tr>
<tr>
<td>P400</td>
<td>&lt;x, F0&gt;, &lt;b, D1&gt;, &lt;output, D1&gt;, &lt;b, E3&gt;</td>
<td>&lt;pred3, P400&gt;</td>
<td>&lt;x, F0&gt;(1)</td>
<td>&lt;b, D1&gt;, &lt;output, D1&gt;, &lt;b, E3&gt;, &lt;pred3, P400&gt;</td>
</tr>
<tr>
<td>E400</td>
<td>&lt;b, D1&gt;, &lt;output, D1&gt;, &lt;b, E3&gt;, &lt;pred3, P400&gt;</td>
<td>&lt;output, E400&gt;</td>
<td>&lt;pred3, P400&gt;, &lt;b, D1&gt;(1), &lt;b, E3&gt;(1)</td>
<td>&lt;output, D1&gt;, &lt;output, E400&gt;</td>
</tr>
<tr>
<td>P41000</td>
<td>&lt;b, D1&gt;, &lt;output, D1&gt;, &lt;b, E3&gt;, &lt;pred3, P400&gt;, &lt;x, F0&gt;(1)</td>
<td>&lt;pred4, P41000&gt;</td>
<td>&lt;x, F0&gt;(2), &lt;pred3, P400&gt;(2)</td>
<td>&lt;b, D1&gt;, &lt;output, D1&gt;, &lt;b, E3&gt;, &lt;pred4, P41000&gt;</td>
</tr>
</tbody>
</table>

Continue...
Table 29 Data state calculation for the sample program (continue)

<table>
<thead>
<tr>
<th>E41000</th>
<th>&lt;b, D1&gt;, &lt;output, D1&gt;, &lt;b, E3&gt;, &lt;pred4, P41000&gt;</th>
<th>&lt;output, E41000&gt;</th>
<th>&lt;pred4, P41000&gt;(1), &lt;b, D1&gt;(2), &lt;b, E3&gt;(3)</th>
<th>&lt;output, D1&gt;, &lt;output, E41000&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>P4101000</td>
<td>&lt;b, D1&gt;, &lt;output, D1&gt;, &lt;b, E3&gt;, &lt;pred4, P41000&gt;, (&lt;x, F0&gt;(2))</td>
<td>&lt;pred5, P4101000&gt;</td>
<td>&lt;x, F0&gt;(1), &lt;pred4, P41000&gt;(2), &lt;b, D1&gt;(3), &lt;b, E3&gt;(3)</td>
<td>&lt;output, D1&gt;, &lt;pred5, P4101000&gt;</td>
</tr>
<tr>
<td>E4101000</td>
<td>&lt;output, D1&gt;, &lt;pred5, P4101000&gt;</td>
<td>&lt;output, E4101000&gt;</td>
<td>&lt;pred5, P4101000&gt;(1)</td>
<td>&lt;output, D1&gt;, &lt;output, E4101000&gt;</td>
</tr>
<tr>
<td>E4101010</td>
<td>&lt;output, D1&gt;, &lt;pred5, P4101000&gt;</td>
<td>&lt;output, E4101010&gt;</td>
<td>&lt;pred5, P4101000&gt;(2)</td>
<td>&lt;output, D1&gt;, &lt;output, E4101010&gt;</td>
</tr>
<tr>
<td>I41010</td>
<td>&lt;x, F0&gt;, &lt;b, D1&gt;, &lt;output, D1&gt;, &lt;b, E3&gt;, &lt;pred4, P41000&gt;</td>
<td>Do not need to calculate</td>
<td>Do not need to calculate</td>
<td>&lt;output, D1&gt;, &lt;output, E4101000&gt;, &lt;output, E4101010&gt;</td>
</tr>
<tr>
<td>I410</td>
<td>&lt;x, F0&gt;, &lt;b, D1&gt;, &lt;output, D1&gt;, &lt;b, E3&gt;, &lt;pred4, P41000&gt;</td>
<td>Do not need to calculate</td>
<td>Do not need to calculate</td>
<td>&lt;output, D1&gt;, &lt;output, E4101000&gt;, &lt;output, E4101010&gt;</td>
</tr>
<tr>
<td>I4</td>
<td>&lt;x, F0&gt;, &lt;b, D1&gt;, &lt;output, D1&gt;, &lt;b, E3&gt;, &lt;pred4, P41000&gt;</td>
<td>Do not need to calculate</td>
<td>Do not need to calculate</td>
<td>&lt;output, D1&gt;, &lt;output, E4101000&gt;, &lt;output, E4101010&gt;</td>
</tr>
<tr>
<td>R5</td>
<td>&lt;output, D1&gt;, &lt;output, E41000&gt;, &lt;output, E4101000&gt;, &lt;output, E4101010&gt;</td>
<td>&lt;Ret1, R5&gt;</td>
<td>&lt;output, D1&gt;(1), &lt;output, E41000&gt;(1), &lt;output, E4101000&gt;(1), &lt;output, E4101010&gt;(1)</td>
<td>&lt;Ret1, R5&gt;</td>
</tr>
</tbody>
</table>

The data state calculated above is a vector of variables that can be statically determined, and hence the data state should be called static data state (SDD). When the program is actually running with inputs, the variables of the data state will be associated with numerical values. The data state is then called dynamic data state (DDD). For instance, assume an input is \{x=5.0, y=3.0\}, the dynamic data state at E200 is \{x=5.0, y=3.0, a=15.0, output=0.0\}.

### 3.4.2 An Algorithm to Implement the Heuristics

In section 3.3, we discussed the heuristics and applied them to a sample program. The calculation of the data state was explained in section 3.4.1. In this section, we will introduce an algorithm to
implement the heuristics and apply it to a sample program. Before that, we will further discuss the branch selection problem.

Assume a statement “Stmt” of a program has several branches. Among these branches there are $n$ good branches (i.e., branches that do not absorb the DSD or are similar to a non-constant function, as discussed in section 3.3.2) and $m$ bad branches. The set of all good branches is $BG$, and the set of all bad branches is $BB$. The set of all branches of “Stmt” is $B$. We then have $BG = \{bg_1, bg_2, \ldots, bg_n\}$, $BB = \{bb_1, bb_2, \ldots, bb_m\}$, and $B = BG \cup BB$.

Assume the execution of a test case up to the entry of the “Stmt” has led to a DSD between the original program P and a certain mutant M. The DSD may or may not be able to propagate through “Stmt”, depending on what branch of P and M the test case will execute. The total number of combinations of possible branches executed in P and M is $(n+m)^2$. This number can be decomposed into the following categories.

1. The first category is based on the first heuristic, H1, i.e., this category considers pairs of branches (one from P, one from M) where at least one branch of P or M is considered a good branch. In other words, when a test case in P takes one of the $n$ $BG$’s, the test case in M can take any branch ($n+m$ possibilities), and vice versa. The total number of combinations is $n^2+2nm$.

2. The second category is based on the second heuristic, H2, i.e., the test case takes different bad branches in P and M. The total number of combinations is simply a permutation of bad branches, i.e., $P_m^2 = m(m-1)$.

3. In other cases where the DSD cannot be propagated, the test case takes the same bad branch in P and M. The total number of combinations is just equal to $m$.

It can easily be verified that $n^2 + 2nm + m(m-1) + m = (n+m)^2$. From the above analysis, there are many combinations that allow a DSD to propagate.
During the execution of a program, which branch is taken is represented by constraints. For example, if we want a test case to take a good branch in P such as 15-17-18-20 defined in Table 23, the constraint is “(1000.0 <= d + a && d + a <= 5000.0) == true”. This constraint is one possible propagation constraint of the “if” statement from line #18 to #24 in Figure 133. It can be used in conjunction with the reachability, necessity, and other intermediate constraints (mainly the assignments between the mutated line and statement #4) to form a complete constraint for propagating the DSD through the “if” statement.

Although there are \( n^2 + 2nm \) combinations that might satisfy heuristic H1, it is unnecessary to attempt them all. This is because if the test case takes a good branch in either P or M, the branch taken in the other program does not matter. Therefore, in the implementation of H1, only up to \( 2n \) different constraints must be investigated.

Another question is in what order should we search for a test case that satisfies the constraints? To evaluate if a test case satisfies the constraints of H2, we must consider the conjunction of constraints from both P and M. In result, evaluating whether a test case satisfies the constraints from H1 is quicker than doing the same for H2. For this reason, our chosen strategy is to first check if we can find a test case that satisfies the constraints of H1 and then check if we can find a test case that satisfies the constraints of H2.

Based on these considerations, an algorithm to strongly kill a given mutant is provided in Figure 138.
In the above algorithm, the variables “ConRe” and “ConNe” refer to the “reachability constraint” and “necessity constraint”, respectively. The functions genConRe(M) and genConNe(M) take the mutant as input and generate reachability and necessity constraints. For a given mutant, the reachability constraint is the conjunction of all the predicates and “expression” statements within the same path as the mutated line but logically preceding it. Although the “expression” statements
are not logical expressions, they can easily be converted and joined with the predicates. The
necessity constraints can be automatically generated based on the template discussed in [44].

The function solve(contraint) takes a constraint and attempts to identify values of variables to
satisfy the constraint. If the constraint is solvable within a certain time limit, the function will
return a set of values; otherwise it returns an empty set. Currently, this function calls the Z3
theorem prover [73] developed by Microsoft Research, but any similar theorem provers can be
used. Because test1 is the result of solve(ConReNe), it can at least weakly kill the mutant.

When the mutated line is a predicate, the “ConNe” needs to be updated after test1 is obtained.
This is because the necessity template for mutants of predicates only requires the test case to take
different branches in the original and the mutant programs, but does not specifically indicate
which branches should be taken. This can only be known after “test1” is obtained. The function
updateConNe(test1,P) takes “test1” and “P” as inputs and returns the specific constraints of the
branch taken by the test case in the original and the mutant programs.

The function execute(P,M,test1,ConInter) performs several tasks. It simultaneously executes both
“P” and “M” with “test1”. As “P” and “M” are executing, this function calculates the DSD
between them at the end of each statement of the control flow and checks if the DSD is equal to
zero (note that DSD is usually a vector as mentioned in section 3.2.4 and therefore “zero” also
means a zero vector). In the meanwhile, this function collects the predicates and “expression”
statements and generates constraints, and then associates them with the variable “ConInter”. The
value of “ConInter” is not used by this function, but it indicates which constraint variable should
be updated. The return value of this function is the statement serial number (variable
“statementNumber”) where the DSD disappears. If this number is less than or equal to the total
number of statements (variable “totalNumber”) in the CFG, it means that the DSD is absorbed
before the exit of the program, and the heuristics must be applied to improve the test case such that the DSD can propagate through the statement with serial number “statementNumber”.

The task of applying the heuristics is performed by genConPr(M, statementNumber). This function takes the mutant and the statement serial number where the DSD disappears as inputs and applies the heuristics in the sequence discussed in the paragraph starting with “Another question is in what order..” of this section. Because only 2n possible constraints of H1 must be considered and the number of constraints of H2 is m(m-1), the maximum number of constraints that would be generated by genConPr() is 2n+m(m-1), where n is the total number of good branches and m is the total number of bad branches of the statement with serial number “statementNumber” (see paragraph starting with “The first category is based on the first heuristic” of this section for the definition of n and m). The variable “test2” is used to save the result from solve(Con). This test case is better than “test1” because it not only weakly kills the mutant but also propagates the DSD further than “test1”.

Note that the algorithm does not always return a strong mutation test case. If the constraints are infeasible or too complex to solve, the returned test case may only weakly kill the mutant and propagate the DSD through several additional statements. However, this test case is still more powerful than the one that is only able to weakly kill the mutant. This is because it satisfies constraints that are stricter than just weakly killing the mutant.

3.4.3 Application of the Algorithm to a Sample Program

In this section, the algorithm in section 3.4.2 is applied to the sample program in Figure 135. There are five statements in the body of the function and two lines in statement #4 can absorb DSD (i.e., line #18 and #20). For statement #4, the total number of good branches is $n = 2$, and
the total number of bad branches is \( m = 2 \). Similar to the analysis for the sample program in Figure 133, Table 30 and Table 31 provides the definition of the branches and paths.

Table 30 Branch performance information for the sample program in Figure 135

<table>
<thead>
<tr>
<th>Statement #</th>
<th>Branch</th>
<th>Branch Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2-3-7</td>
<td>Good, the branch is monotonic</td>
</tr>
<tr>
<td>2</td>
<td>2-4-5-7</td>
<td>Good, the branch is monotonic</td>
</tr>
<tr>
<td>2</td>
<td>2-4-6-7</td>
<td>Good, the branch is monotonic</td>
</tr>
<tr>
<td>4</td>
<td>9-10-16</td>
<td>Good, the branch is monotonic</td>
</tr>
<tr>
<td>4</td>
<td>9-11-12-16</td>
<td>Good, the branch is monotonic</td>
</tr>
<tr>
<td>4</td>
<td>9-11-13-14</td>
<td>Bad, line #18 can absorb DSD</td>
</tr>
<tr>
<td>4</td>
<td>9-11-13-15</td>
<td>Bad, line #20 can absorb DSD</td>
</tr>
</tbody>
</table>

Table 31 Path definition for sample program

<table>
<thead>
<tr>
<th>Path #</th>
<th>Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1-2-3-7-8-9-10-16-17</td>
</tr>
<tr>
<td>2</td>
<td>1-2-3-7-8-9-11-12-16-17</td>
</tr>
<tr>
<td>3</td>
<td>1-2-3-7-8-9-11-13-14-16-17</td>
</tr>
<tr>
<td>4</td>
<td>1-2-3-7-8-9-11-13-15-16-17</td>
</tr>
<tr>
<td>5</td>
<td>1-2-4-5-7-8-9-10-16-17</td>
</tr>
<tr>
<td>6</td>
<td>1-2-4-5-7-8-9-11-12-16-17</td>
</tr>
<tr>
<td>7</td>
<td>1-2-4-5-7-8-9-11-13-14-16-17</td>
</tr>
<tr>
<td>8</td>
<td>1-2-4-5-7-8-9-11-13-15-16-17</td>
</tr>
<tr>
<td>9</td>
<td>1-2-4-6-7-8-9-10-16-17</td>
</tr>
<tr>
<td>10</td>
<td>1-2-4-6-7-8-9-11-12-16-17</td>
</tr>
<tr>
<td>11</td>
<td>1-2-4-6-7-8-9-11-13-14-16-17</td>
</tr>
<tr>
<td>12</td>
<td>1-2-4-6-7-8-9-11-13-15-16-17</td>
</tr>
</tbody>
</table>

1. Mutant #1: an AOR mutant of line #4 ("expression" statement)

Original: \( a = 5.0 \times y \); Mutated: \( a = 5.0 + y \)
First consider a mutant at line #4 of the program in **Figure 135**. Based on the algorithm, “ConRe” is \((x+y<10.0)\) and “ConNe” is \((a^P==5.0*y) \land (a^M==5.0+y) \land (a^P!=a^M)\), where “\(a^P\)” is the variable “\(a\)” in the original program and “\(a^M\)” is the variable “\(a\)” in the mutant program. Thus “ConReNe” is \((x+y<10.0) \land (a^P==5.0*y) \land (a^M==5.0+y) \land (a^P!=a^M)\). The fourth line of the algorithm in **Figure 138** solves the constraint and returns “test1” equal to \(\{x=1.0, y=0.1\}\). Because the mutated line is not a predicate, lines #5 to #8 of the algorithm in **Figure 138** are not executed. Line #14 of the algorithm executes both P and M with the same test \(\{x=1.0, y=0.1\}\), calculates the DSD at the exit of each block, and returns the block number where the DSD disappears. It can be verified that the variable “\(b^P\)” becomes equal to 10.5 and “\(b^M\)” becomes equal to 15.1, thus the DSD propagates through statement #2. In statement #4, \(b^P+x\) and \(b^M+x\) is equal to 11.5 and 16.1, respectively. Because both values are between 10.0 and 20.0, test 1 traverses the same bad branch 9-11-13-14-16 in both P and M, and output\(^P\) = output\(^M\) = 1.0, which means that the DSD disappears at statement #4. The variable “ConInter” is updated to \((b^P==a^P+10) \land (b^M==a^M+10)\) during the execution. The return value of line #14 of the algorithm in **Figure 138** is “4”, and lines #15 to #29 of the algorithm execute to generate “ConPr” and solve the constraint “Con”. Based on the propagation constraint generation sequence recommended in the paragraph starting with “Another question is in what order…” in section 3.4.2, the first “ConPr” generated is \((b^P+x<=5.0)\), which will lead the newly generated test case to take the first good branch 9-10-16 of statement #4 in P (see **Figure 135**). The resulting complete constraint “Con” is ConReNe \(\land\) ConInter \(\land\) ConPr, which is \((x+y<10.0) \land (a^P==5.0*y) \land (a^M==5.0+y) \land (a^P!=a^M) \land (b^P==a^P+10) \land (b^M==a^M+10) \land (b^P+x<=5.0)\). Solving for this constraint, we obtain “test2” \(\{x=-12.5, y=1.5\}\), which can propagate the DSD through statement #4 and strongly kill the mutant since \(\{\text{output}^P=52.5, \text{output}^M=49.5\}\). Therefore, the final test case returned by the algorithm is \(\{x=-12.5, y=1.5\}\).

2. Mutant #2: an ROR mutant of line #3 (predicate)
The mutants of predicates can also be resolved by the algorithm. The “ConRe” is trivial. The “ConNe” is \((x+y<10.0) \land \neg(x+y>10.0)) \lor (\neg(x+y<10.0) \land (x+y>10.0))\). Line #4 of the algorithm in Figure 138 returns test1 as \(\{x=6.0, y=1.0\}\). Because the mutated line is a predicate, lines #5 to #8 of the algorithm in Figure 138 update the “ConNe” to \((x+y<10)\). In this case, line #14 of the algorithm in Figure 138 again returns statementNumber = 4, which means that the DSD disappears in statement #4 (The reader can simply verify that \(\{x=6.0, y=1.0\}\) causes both P and M to take the same bad branch 9-11-13-15-16 in statement #4, and output\(^P\)=output\(^M\)=0.0). The “ConInter” here is \((a\(^P\)==5.0*y) \land (a\(^M\)==15.0*y) \land (b\(^P\)==a\(^P\)+10) \land (b\(^M\)==a\(^M\)+10)\). Line #17 of the algorithm applies the heuristics and returns ConPr = \((b\(^P\)+x<=5.0)\), and the complete constraint “Con” is equal to \((x+y<10.0) \land (a\(^P\)==5.0*y) \land (a\(^M\)==15.0*y) \land (b\(^P\)==a\(^P\)+10) \land (b\(^M\)==a\(^M\)+10) \land (b\(^P\)+x<=5.0)\). Solving this constraint, we obtain “test2” as \(\{x=0.0, y=-1.0\}\), which can strongly kill the mutant \(\{output\(^P\)=15.0, output\(^M\)==-15.0\}\).

It should be noted that, in both cases, the first possible “ConPr” (which generates test cases traversing the first good branch 9-10-16 in P in Figure 135) is able to propagate the DSD through statement #4 and strongly kill the mutants. This is just a coincidence. If this “ConPr” cannot find appropriate test cases, the function genConPr(M,blockNumber) will continue finding new “ConPr” until it exhausting all possibilities.

3.5 Heuristics for Propagating Data State Difference Though Loops

In previous sections the heuristics about how to propagate DSD through “if” statements are discussed. Besides “if” statements, loops are sources of DSD absorption as well since there may be multiple branches in the loops and some of them may absorb DSD. In this research we propose a heuristic for loops which has been preliminarily verified.
First let us consider an example in Figure 139. Assume line #4 is mutated to “y = a – x;”. The branch of 4-5-6-8-9 is good because variable “b” is updated by a monotonic function. However, line #9 in the loop body makes “b” always equal to a constant 5. Based on the previous discussion the branch of 4-5-7-8-9 is a bad one. When this statement is executed, the DSD will be partially absorbed.

```
1 double fun(double x) { // F0
2   double a=0.0,b=0.0,y=0.0,output=0.0;
3       //D1
4   a = x * x;    // E2
5   y = a + x;   // E3, Mutated line
6   while (y < 115) {
7       if (a > y) {
8         b = 5 * y + 3; // E4
9       } else { // B4
10         b = 5;  // E4
11       }
12     y = y + 30;  // E4-0-2
13   }
14   output = a + b; // E5
15   return output;  // R6
16 }
```

**Figure 139** Sample program to illustrate how the DSDs are absorbed in a "while" loop.
Table 32 Static data state of the statements in the sample program

<table>
<thead>
<tr>
<th>Statement</th>
<th>IN</th>
<th>GEN</th>
<th>KILL</th>
<th>OUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>F0</td>
<td>Ø</td>
<td>&lt;x, F0&gt;</td>
<td>Ø</td>
<td>&lt;x, F0&gt;</td>
</tr>
<tr>
<td>D1</td>
<td>&lt;x, F0&gt;</td>
<td>&lt;a, D1&gt;, &lt;b, D1&gt;, &lt;y, D1&gt;, &lt;output, D1&gt;</td>
<td>Ø</td>
<td>&lt;x, F0&gt;, &lt;a, D1&gt;, &lt;b, D1&gt;, &lt;y, D1&gt;, &lt;output, D1&gt;</td>
</tr>
<tr>
<td>E2</td>
<td>&lt;x, F0&gt;, &lt;a, D1&gt;, &lt;b, D1&gt;, &lt;y, D1&gt;, &lt;output, D1&gt;</td>
<td>&lt;a, E2&gt;</td>
<td>Ø</td>
<td>&lt;x, F0&gt;, &lt;a, D1&gt;, &lt;b, D1&gt;, &lt;y, D1&gt;, &lt;output, D1&gt;, &lt;a, E2&gt;</td>
</tr>
<tr>
<td>E3</td>
<td>&lt;x, F0&gt;, &lt;a, D1&gt;, &lt;b, D1&gt;, &lt;y, D1&gt;, &lt;output, D1&gt;, &lt;a, E2&gt;</td>
<td>&lt;y, E3&gt;</td>
<td>&lt;x, F0&gt;</td>
<td>Ø</td>
</tr>
<tr>
<td>P400</td>
<td>&lt;a, D1&gt;, &lt;b, D1&gt;, &lt;y, D1&gt;, &lt;output, D1&gt;, &lt;a, E2&gt;, &lt;y, E3&gt;, &lt;b, 40101&gt;, &lt;b, 40111&gt;, &lt;y, E402&gt;, &lt;pred1, P400&gt;</td>
<td>&lt;pred1, P400&gt;</td>
<td>Ø</td>
<td>&lt;a, D1&gt;, &lt;b, D1&gt;, &lt;y, D1&gt;, &lt;output, D1&gt;, &lt;a, E2&gt;, &lt;y, E3&gt;, &lt;b, 40101&gt;, &lt;b, 40111&gt;, &lt;y, E402&gt;, &lt;pred1, P400&gt;</td>
</tr>
<tr>
<td>P40100</td>
<td>&lt;a, D1&gt;, &lt;b, D1&gt;, &lt;y, D1&gt;, &lt;output, D1&gt;, &lt;a, E2&gt;, &lt;y, E3&gt;, &lt;b, 40101&gt;, &lt;b, 40111&gt;, &lt;y, E402&gt;, &lt;pred2, P40100&gt;</td>
<td>&lt;pred2, P40100&gt;</td>
<td>&lt;pred1, P400&gt;</td>
<td>Ø</td>
</tr>
<tr>
<td>E40101</td>
<td>&lt;a, D1&gt;, &lt;b, D1&gt;, &lt;y, D1&gt;, &lt;output, D1&gt;, &lt;a, E2&gt;, &lt;y, E3&gt;, &lt;b, 40101&gt;, &lt;b, 40111&gt;, &lt;y, E402&gt;, &lt;pred1, P40100&gt;</td>
<td>&lt;b, 40101&gt;</td>
<td>&lt;pred2, P40100&gt;</td>
<td>Ø</td>
</tr>
<tr>
<td>B40100</td>
<td>&lt;a, D1&gt;, &lt;b, D1&gt;, &lt;y, D1&gt;, &lt;output, D1&gt;, &lt;a, E2&gt;, &lt;y, E3&gt;, &lt;b, 40101&gt;, &lt;b, 40111&gt;, &lt;y, E402&gt;, &lt;pred2, P40100&gt;</td>
<td>Do not need to calculate</td>
<td>Do not need to calculate</td>
<td>Ø</td>
</tr>
<tr>
<td>E40111</td>
<td>&lt;a, D1&gt;, &lt;b, D1&gt;, &lt;y, D1&gt;, &lt;output, D1&gt;, &lt;a, E2&gt;, &lt;y, E3&gt;, &lt;b, 40101&gt;, &lt;b, 40111&gt;, &lt;y, E402&gt;, &lt;pred2, P40100&gt;</td>
<td>&lt;b, 40111&gt;,</td>
<td>&lt;pred2, P40100&gt;</td>
<td>Ø</td>
</tr>
<tr>
<td>B40110</td>
<td>&lt;a, D1&gt;, &lt;b, D1&gt;, &lt;y, D1&gt;, &lt;output, D1&gt;, &lt;a, E2&gt;, &lt;y, E3&gt;, &lt;b, 40101&gt;, &lt;b, 40111&gt;, &lt;y, E402&gt;, &lt;pred2, P40100&gt;</td>
<td>Do not need to calculate</td>
<td>Do not need to calculate</td>
<td>Ø</td>
</tr>
<tr>
<td>I401</td>
<td>&lt;a, D1&gt;, &lt;b, D1&gt;, &lt;y, D1&gt;, &lt;output, D1&gt;, &lt;a, E2&gt;, &lt;y, E3&gt;, &lt;b, 40101&gt;, &lt;b, 40111&gt;, &lt;y, E402&gt;</td>
<td>Do not need to calculate</td>
<td>Do not need to calculate</td>
<td>Ø</td>
</tr>
</tbody>
</table>

Continue...
The static data state of each statement is provided in Table 32. Assume we have an input of \( \{x = -5\} \). The dynamic data state after the mutated line (i.e., E3) is \( \{a=25, b=0, y=20, output=0\} \) for the original program and \( \{a=25, b=0, y=30, output=0\} \) for the mutant, respectively. The mutant is weakly killed since the data states are different. When the loop is executed for the first time, the original program takes the good branch 4-5-6-8-9, while the mutant takes the bad branch 4-5-7-8-9. The global data state of the original program is \( \{a=25, b=103, output=0\} \) and the global data state of the mutant is \( \{a=25, b=5, output=0\} \). The DSD is preserved. However, after the second iteration, the global data state of both programs is \( \{a=25, b=5, output=0\} \), which means that the global DSD is already absorbed. After that the “while” loop iterates 2 more times for the original program and 1 more time for the mutant, with no global DSD.
To propagate the DSD through the “while” loop, a constraint regarding the number of iterations of the “while” can be added. More specifically, we can require that the “while” loop in the original program is terminated after the first iteration, where the DSD still exists. The constraint is added to the original program because it takes the good branch in the first iteration. The entire constraint system is as follows:

\[(a=x^2) \land (y^P=a^P+x) \land (y^M=a^M-x) \land (y^P \neq y^M) \land (y^P<115) \land (a^P>y^P) \land (b^P=5*3^P+3) \land (y^P=y^P+30) \land (y^P\geq 115)\]

If we solve it, we obtain \(\{x^{new}=-10\}\). Applying it to both programs and we see that the “while” loop of the original program only executes once as expected, and the global data state is \(\{a=100, b=453, output=0\}\). The “while” loop of the mutant also only executes once, and the global data state is \(\{a=100, b=5, output=0\}\). Therefore, the DSD propagates through the “while” loop successfully. Based on the analysis, we propose the following heuristic:

If the DSD is absorbed within a loop, the specific iteration during which the absorption occurs should be recorded. If the original program takes a good branch in the previous iteration, the loop of the original program should be terminated right before the iteration during which the absorption occurs; otherwise if the mutant takes a good branch in the previous iteration, the loop of the mutant should be terminated right before the iteration during which the absorption occurs. (H3)

Note that this heuristic is only at its initial stages of verification and not been integrated into the system. Future research will need a further investigation of this topic.
3.6 Current Status of the Implementation of the Automated Software Mutation Testing System (ASMTS) Tool

The ASMTS is implemented using the ROSE compiler infrastructure [86], which supports C, C++, Java, FORTRAN, etc. It reads the source files, parses them and generates the abstract syntax tree (AST) [84]. It stores all the information about the program and provides convenient access through a set of C++ application programming interfaces (API). At this point the ASMTS is able to read C source code, automatically generate mutants for the five mutant operators, and generate test cases to weakly kill the mutants. The features for strong mutation are not yet completed due to time limitations and can be continued in the future research.
Chapter 4: Conclusions and Future Research

In this research we investigate how to utilize the software fault information to predict software reliability, and how to utilize seeded software faults to derive test suites. For software reliability prediction we developed the EFSM-based RePS methodology. This methodology utilizes the software development artifacts (including SRS, SDD and source code) to construct a hierarchical model for the software. The SRS level model is at the highest level of abstraction; the SDD level model is at the medium level of abstraction; the code level model is at the lowest level of abstraction. The model is based on EFSM, which is a widely used graphical representation. Software faults uncovered in the N-1’th version of the software are mapped onto the EFSM model, and the Execution, Infection and Propagation analysis are conducted to assess the software reliability. We also developed the ARPS tool which implements the EFSM-based RePS methodology. It can automatically construct the EFSM model based on the user inputs, and also conduct the reliability assessment. An experiment was designed to evaluate the usability of the ARPS tool, where human subjects were recruited, trained and their performance examined. It was found that the ARPS tool not only assisted the subjects in achieving a better performance but also was preferred to the manual analysis.

Software mutation testing is a very powerful test case development technique. It systematically seeds faults into the source code and creates mutated programs (i.e., mutants) that are almost identical to the original program. Test cases are then designed to distinguish the mutants and the original program. Weak mutation just requires that the mutants can be distinguished right after the mutated line of code, while strong mutation further requires that the data state difference
propagate to the end of the program. In this research strong mutation is investigated, where several heuristics are introduced to help the data state propagate. Sample programs are used to validate the heuristics, and the results are promising. The ASMTS tool implements a subset of the methodology and can be completed in the future.

Future research includes the following aspects: for the EFSM-based RePS methodology, the SDD level method should be finalized and the code level method should be developed. The Propagation analysis is one of the most difficult topics in this methodology, and it should be completed in the future. For the ASMTS tool, the heuristics regarding how to propagate the data state through loops should be further verified and improved. More features of the C language should be included, and the implementation and testing of the entire system should be conducted.
References

192


Appendix A: The SCS Case Study for the ARPS Tool

A.1. Software Requirement Specifications for a Simple Control System

A.1.1. Introduction to the Simple Control System (SCS)

The Simple Control System (SCS) takes the pressure and temperature signals from sensors as inputs and controls four different valves: Valve #1 to Valve #4. Each valve has its own register. If the valve is fully open, the register is set to 1; otherwise it remains 0. The temperature is a decimal value. It ranges from 0 ºC to 100 ºC; the pressure is also a decimal and ranges from 5*10^5 Pa to 5*10^6 Pa. There is a Boolean variable “Status” in the system which is used to control Valve #4. The system does not return any numerical output. The way it affects the external world is controlling the valves.

The logic of the system is as follows:

1. The system is initialized.
2. If the temperature is greater than or equal to 15 ºC, Valve #1 shall open. Register R1 is set to 1.
3. If the pressure is greater than or equal to 10^6 Pa, Valve #3 shall open. Register R3 is set to 1.
4. Valve #4 shall open at last. Register R4 is set to 1.

A.1.2. Specific Functional Requirements

A.1.2.1 Function No.1: initialize the system.
A.1.2.1.1 Inputs:

A. Temperature. This is a decimal value. It comes from the pressure sensor and hence crosses the application boundary. The range is 0 ºC to 100 ºC.

B. Pressure. This is a decimal value. It comes from the pressure sensor and hence crosses the application boundary. The range is $5 \times 10^5$ Pa to $5 \times 10^6$ Pa.

A.1.2.1.2 Outputs: This function does not return a value, but it affects the variable “Status”.

A.1.2.1.3 Functionality: initialize the system.

A.1.2.2 Function No.2: open Valve #1.

A.1.2.2.1 Inputs:

A. R1. This is a register allocated for Valve #1. It is an integer value that crosses the application boundary.

A.1.2.2.2 Outputs:

A. R1. The same as above.

A.1.2.2.3 Functionality: open Valve #1 to 100%.

A.1.2.3 Function No. 3: open Valve #2.

A.1.2.3.1 Inputs:

A. R2. This is a register allocated for Valve #2. It is an integer value that crosses the application boundary.

A.1.2.3.2 Outputs:

A. R2. The same as above.

A.1.2.3.3 Functionality: open Valve #2 to 100%.

A.1.2.4 Function No.4: open Valve #3.

A.1.2.4.1 Inputs:

A. Temperature. This is the same as 2.1.1.A
B. Pressure. This is the same as 2.1.1.B

C. R3. This is a register allocated for Valve #3. It is an integer value that crosses the application boundary.

A.1.2.4.2 Outputs:

A. R3. The same as above.

A.1.2.4.3 Functionality: open Valve #3 to 100%.

A.1.2.5 Function No.5: open Valve #4.

A.1.2.5.1 Inputs:

A. R4. This is a register allocated for Valve #4. It is an integer value that crosses the application boundary.

A.1.2.5.2 Outputs:

A. R4. The same as above.

A.1.2.5.3 Functionality: open Valve #4 to 100%.

A.2. Software Design Documentation for the SRS function openValve3

A.2.1. Introduction to the openValve3 function

This function is used to control valve #3. Its inputs are temperature, pressure and Register R3. Their definition can be found in the SRS. Its output is R3. This function has an inner variable. Its name is position. It is an integer and ranges from 1 to 3. The logic of openValve3 is provided as follows:

1. If position is equal to 1, position is updated by the function calcPos1. Otherwise position is updated by the function calcPos2.
2. After that, position is updated by the function calcPos3.
3. If position is equal to 1, the valve is adjusted by the function moveUp. Otherwise it is adjusted by the function moveDown.
A.2.2. Specific Design Description

A.2.2.1 Function No.1: obtain the current position

A.2.2.1.1 Inputs:

A.2.2.1.2 Outputs:

A. Position. This is an integer value. It is an inner variable and hence does not cross the application boundary. The range is 1 to 3.

A.2.2.1.3 Functionality: obtain the current position of the valve.

A.2.2.2 Function No.2: calculate position #1

A.2.2.2.1 Inputs:

A. Temperature. This is the same as the SRS

A.2.2.2.2 Outputs: This function does not return an output but it affects Position

A.2.2.2.3 Functionality: calculate the position of the valve using method #1.

A.2.2.3 Function No.3: calculate position #2

A.2.2.3.1 Inputs:

A. Pressure. This is the same as the SRS

A.2.2.3.2 Outputs: This function does not return an output but it affects Position

A.2.2.3.3 Functionality: calculate the position of the valve using method #2.

A.2.2.4 Function No.4: calculate position #3

A.2.2.4.1 Inputs:

A. Temperature. This is the same as the SRS

A.2.2.4.2 Outputs: This function does not return an output but it affects Position

A.2.2.4.3 Functionality: calculate the position of the valve using method #3.

A.2.2.5 Function No.5: move up

A.2.2.5.1 Inputs:
A. Position. This is the same as above

A.2.2.5.2 Outputs: This function does not return an output but it affects Register R3

A.2.2.5.3 Functionality: the valve is moved up based on Position

A.2.2.6 Function No.6: move down

A.2.2.6.1 Inputs:

A. Position. This is the same as above

A.2.2.6.2 Outputs: This function does not return an output but it affects Position

A.2.2.6.3 Functionality: the valve is moved down based on Position

A.3. SRS Level Function Cards for the SCS Case Study

The SRS level function cards for the original version of the SCS are provided in Figure 140 to Figure 145.

---

**Function Name and Level:** SRS, level 0  
**Inputs:**

1. Input 1:  
   - Name: Temperature  
   - Type: Decimal  
   - Application Boundary: Yes  
   - Range: 0 °C to 100 °C  
2. Input 2:  
   - Name: Pressure  
   - Type: Decimal  
   - Application Boundary: Yes  
   - Range: 5*10^5 Pa to 5*10^6 Pa

**Outputs:** N/A

**Variables:**

1. Variable 1:  
   - Name: Status  
   - Type: Boolean  
   - Application Boundary: No  
   - Range: N/A

---

**Figure 140** Function card for SRS
Function Name and Level: `initializeTheSystem`, level 1
Inputs:
1. Input 1:
   Name: Temperature
   Type: Decimal
   Application Boundary: Yes
   Range: 0 ºC to 100 ºC
2. Input 2:
   Name: Pressure
   Type: Decimal
   Application Boundary: Yes
   Range: 5*10^5 Pa to 5*10^6 Pa
Outputs: This function does not return a value but it affects the variable “State”.

Figure 141 Function card for `initializeTheSystem`

Function Name and Level: `openValve1`, level 1
Inputs:
1. Input 1:
   Name: R1
   Type: Integer
   Application Boundary: Yes
   Range: N/A
Outputs:
1. Output 1:
   Name: R1
   Type: Integer
   Application Boundary: Yes
   Range: N/A

Figure 142 Function card for `openValve1`
Function Name and Level: openValve2, level 1
Inputs:
1. Input 1:
   Name: R2
   Type: Integer
   Application Boundary: Yes
   Range: N/A

Outputs:
1. Output 1:
   Name: R2
   Type: Integer
   Application Boundary: Yes
   Range: N/A

**Figure 143** Function card for openValve2

Function Name and Level: openValve3, level 1
Inputs:
1. Input 1:
   Name: Temperature
   Type: Decimal
   Application Boundary: Yes
   Range: 0 ºC to 100 ºC
2. Input 2:
   Name: Pressure
   Type: Decimal
   Application Boundary: Yes
   Range: 5*10^5 Pa to 5*10^6 Pa
3. Input 3:
   Name: R3
   Type: Integer
   Application Boundary: Yes
   Range: N/A

Outputs:
1. Output 1:
   Name: R3
   Type: Integer
   Application Boundary: Yes
   Value: N/A
   Range: N/A

**Figure 144** Function card for openValve3
Function Name and Level: openValve4, level 1
Inputs:
1. Input 1:
   Name: R4
   Type: Integer
   Application Boundary: Yes
   Range: N/A

Outputs:
1. Output 1:
   Name: R4
   Type: Integer
   Application Boundary: Yes
   Range: N/A

**Figure 145** Function card for openValve4

**A.4. SDD Level Function Cards for the SCS Case Study**

SDD level function cards for the original version of the openValve3 are provided in **Figure 146** to **Figure 151**.

Function Name and Level: obtainPosition, level 2
Inputs:

Outputs:
1. Output 1:
   Name: Position
   Type: Integer
   Application Boundary: No
   Range: 1 to 3

**Figure 146** Function card for obtainPosition
Function Name and Level: calcPos1, level 2
Inputs:
1. Input 1:
   Name: Temperature
   Type: Decimal
   Application Boundary: Yes
   Range: 0 ºC to 100 ºC
Outputs: This function does not return an output but it affects Position

Figure 147 Function card for calcPos1

Function Name and Level: calcPos2, level 2
Inputs:
1. Input 1:
   Name: Pressure
   Type: Decimal
   Application Boundary: Yes
   Range: 5*10^5 Pa to 5*10^6 Pa
Outputs: This function does not return an output but it affects Position

Figure 148 Function card for calcPos2

Function Name and Level: calcPos3, level 2
1. Input 1:
   Name: Temperature
   Type: Decimal
   Application Boundary: Yes
   Range: 0 ºC to 100 ºC
Outputs: This function does not return an output but it affects Position

Figure 149 Function card for calcPos3
A.5. Original Version of the High Level Extended Finite State Machine (HLEFSM) Model for the SCS Case Study

Figure 152 displays the original version of SCS’s HLEFSM model. Since it is the original version, no defects have been mapped.
A.6. Original Version of the SDD Level Extended Finite State Machine Model for openValve3

Figure 153 displays the original version for openValve3’s SDD level EFSM model. Since it is the original version, no defects have been mapped.
Figure 153 openValve3’s original SDD level EFSM model

A.7. Defect Report for SCS’s SRS

Table 33 provides the defects identified in the original SRS document.
Table 33 Descriptions for SCS’s SRS defects

<table>
<thead>
<tr>
<th>Defect #</th>
<th>Location</th>
<th>Defect Type</th>
<th>Original Description</th>
<th>Correct Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Section 1 2)</td>
<td>Incorrect/Ambiguous Predicate</td>
<td>“If the temperature is greater than or equal to 15 °C, Valve #1 shall open.”</td>
<td>If the temperature is greater than or equal to 45 °C, Valve #1 shall open.</td>
</tr>
<tr>
<td>2</td>
<td>Section 1</td>
<td>Missing Instance of Function</td>
<td>N/A</td>
<td>Valve #2 shall open after 2) of Section 1.</td>
</tr>
<tr>
<td>3</td>
<td>Section 2.4.3</td>
<td>Function with Incorrect Logic</td>
<td>“Functionality: open valve #3 to 100%.”</td>
<td>Valve #3 shall be opened based on the temperature.</td>
</tr>
<tr>
<td>4</td>
<td>Section 1 4)</td>
<td>Missing Predicate</td>
<td>“Valve #4 shall open at last.”</td>
<td>If the Boolean variable Status is true, open Valve #4.</td>
</tr>
</tbody>
</table>

A.8. Defect Report for openValve3’s SDD

Table 34 provides the defects identified in the original SDD document.

Table 34 Descriptions for openValve3’s SDD defects

<table>
<thead>
<tr>
<th>Defect #</th>
<th>Location</th>
<th>Defect Type</th>
<th>Original Description</th>
<th>Correct Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Section 1 1)</td>
<td>Missing Instance of Function</td>
<td>“If position is equal to 1, position is updated by the function calcPos1.”</td>
<td>First obtain the current position. If position is equal to 1, position is updated by the function calcPos1.</td>
</tr>
<tr>
<td>2</td>
<td>Section 2.3.3</td>
<td>Function with Incorrect Logic</td>
<td>“Calculate the position of the valve using method #2”</td>
<td>The description is correct but the implementation is erroneous</td>
</tr>
<tr>
<td>3</td>
<td>Section 1 2)</td>
<td>Missing Predicate</td>
<td>“After that, position is updated by the function calcPos3.”</td>
<td>After that, if position is equal to 3, position is updated by the function calcPos3.</td>
</tr>
<tr>
<td>4</td>
<td>Section 1 3)</td>
<td>Incorrect/Ambiguous Predicate</td>
<td>“If position is equal to 1, the valve is adjusted by the function moveUp. Otherwise it is adjusted by the function moveDown.”</td>
<td>If position is equal to 1 or 2, the valve is adjusted by the function moveUp. Otherwise it is adjusted by the function moveDown.</td>
</tr>
</tbody>
</table>

207
A.9. Modified Function Cards for the SCS Case Study

Figure 154 shows the modified function card for openValve3.

<table>
<thead>
<tr>
<th>Function Name and Level: openValve3, level 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inputs:</td>
</tr>
<tr>
<td>1. Input 1:</td>
</tr>
<tr>
<td>Name: Temperature</td>
</tr>
<tr>
<td>Type: Decimal</td>
</tr>
<tr>
<td>Application Boundary: Yes</td>
</tr>
<tr>
<td>Range: 0 °C to 100 °C</td>
</tr>
<tr>
<td>Outputs: N/A</td>
</tr>
<tr>
<td>Defect: Incorrect Logic. Will be modified using SDD level analysis</td>
</tr>
</tbody>
</table>

Figure 154 Modified Function Card for openValve3

A.10. Modified Function Cards at the SDD level for openValve3

Figure 155 provides the modified function card for function calcPos2 in openValve3.

<table>
<thead>
<tr>
<th>Function Name and Level: calcPos2, level 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inputs:</td>
</tr>
<tr>
<td>1. Input 1:</td>
</tr>
<tr>
<td>Name: Pressure</td>
</tr>
<tr>
<td>Type: Decimal</td>
</tr>
<tr>
<td>Application Boundary: Yes</td>
</tr>
<tr>
<td>Range: 5<em>10^5 Pa to 5</em>10^6 Pa</td>
</tr>
<tr>
<td>Outputs: This function does not return an output but it affects Position</td>
</tr>
<tr>
<td>Defect: Incorrect Logic.</td>
</tr>
</tbody>
</table>

Figure 155 Modified Function Card for calcPos2 in openValve3
A.11. Modified HLEFSM Model for the SCS Case Study

Figure 156 Modified HLEFSM with all Defects Mapped
A.12. Modified SDD level EFSM Model for openValve3

Figure 157 shows the modified SDD level EFSM model for openValve3.

Figure 157 The modified SDD level EFSM model for openValve3
A.13. Operational Profile (OP) of SCS

Table 35 displays SCS’s operational profile.

<table>
<thead>
<tr>
<th>No.</th>
<th>Description of the Event</th>
<th>Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Low temperature. I.e., 0 °C ≤ Temperature &lt; 15 °C</td>
<td>15%</td>
</tr>
<tr>
<td>2</td>
<td>Medium temperature. I.e., 15 °C ≤ Temperature &lt; 45 °C</td>
<td>30%</td>
</tr>
<tr>
<td>3</td>
<td>High temperature. I.e., 45 °C ≤ Temperature ≤ 100 °C</td>
<td>55%</td>
</tr>
<tr>
<td>4</td>
<td>Other temperature. I.e., Temperature &lt; 0 °C or Temperature &gt; 100 °C</td>
<td>0%</td>
</tr>
<tr>
<td>5</td>
<td>Low pressure. I.e., 5*10^5 Pa ≤ Pressure &lt; 10^6 Pa</td>
<td>27%</td>
</tr>
<tr>
<td>6</td>
<td>High pressure. I.e., 10^6 Pa ≤ Pressure ≤ 5*10^6 Pa</td>
<td>73%</td>
</tr>
<tr>
<td>7</td>
<td>Other pressure. I.e., Pressure &lt; 5<em>10^5 Pa or Pressure &gt; 5</em>10^6 Pa</td>
<td>0%</td>
</tr>
<tr>
<td>8</td>
<td>The Boolean variable “Status” being true</td>
<td>34%</td>
</tr>
<tr>
<td>9</td>
<td>The Boolean variable “Status” being false</td>
<td>66%</td>
</tr>
</tbody>
</table>

A.14. Operational Profile (OP) for openValve3 at the SDD level

Table 36 describes openValve3’s operational profile at the SDD level.

<table>
<thead>
<tr>
<th>No.</th>
<th>Description of the Event</th>
<th>Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Low position. I.e., Position == 1</td>
<td>15%</td>
</tr>
<tr>
<td>2</td>
<td>Medium position. I.e., Position == 2</td>
<td>36%</td>
</tr>
<tr>
<td>3</td>
<td>High position. I.e., Position == 3</td>
<td>49%</td>
</tr>
</tbody>
</table>
SDD’s OP is used in an identical manner. Based on Figure 157, the Execution probability for SDD level Defect #1, #3 and #4 is 100%. The Execution probability for SDD Defect #2 is calculated as follows:

\[
E(\text{Node}_3) = \text{Prob}(\text{Path}_i^3) = \text{Prob}(!(\text{Position} == 1)) = 1 - 15\% = 85\%
\]

**A.15. Infection and Propagation Data for the PIE Analysis for the SRS**

The following formulas are used for the PIE analysis:

\[
\text{Prob(failure)} = \sum_{i=1}^{n} E_i \times I_i \times P_i
\]

\[
\text{Re} = 1 - \text{Prob(failure)}
\]

Where the subscript \( i \) indicates the \( i \text{th} \) defect. **Table 37** provides the Infection and Propagation data used for SCS’s PIE analysis. Infection probabilities for Defect #1 and #4 are calculated using the OP data and all other data is provided directly.

**Table 37** Infection and Propagation data for the SCS’s PIE analysis

<table>
<thead>
<tr>
<th>Defect No.</th>
<th>Defect Type</th>
<th>Infection %</th>
<th>Propagation %</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Incorrect/Ambiguous Predicate</td>
<td>(Should be calculated by the user)</td>
<td>7%</td>
</tr>
<tr>
<td>2</td>
<td>Missing Instance of Function</td>
<td>100%</td>
<td>5%</td>
</tr>
<tr>
<td>3</td>
<td>Function with Incorrect Logic</td>
<td>(Should be calculated using the SDD level information)</td>
<td>10%</td>
</tr>
<tr>
<td>4</td>
<td>Missing Predicate</td>
<td>(Should be calculated by the user)</td>
<td>3%</td>
</tr>
</tbody>
</table>

Based on V.3, we have:

\[
P_{DW} = (P_c \cap (\Omega \setminus P_o)) \cup (P_o \cap (\Omega \setminus P_c))
\]
where $P_{ow}$ is the incorrect portion of the original predicate $P_o$. $P_c$ is the correct predicate given in the defect report. $\Omega$ is the entire range of the predicate variable. For $I_1$ this variable is Temperature, and for $I_4$ this variable is Status. Using data provided in Table 6, we can calculate the probability of $P_{ow}$ being true, which is the Infection probability for defects “Incorrect/Ambiguous Predicate” and “Missing Predicate”.

The Infection probability for defect #1 is calculated as follows.

$$I_1 = \text{Prob}(P_{ow})$$

First let us calculate $P_{ow}$:

$$P_{ow} = (P_c \cap (\Omega \setminus P_o)) \cup (P_o \cap (\Omega \setminus P_c))$$

$$= ([45, 100] \cap ([0, 100]\setminus[15, 100])) \cup ([15, 100] \cap ([0, 100]\setminus[45, 100]))$$

$$= (\emptyset) \cup ([15, 45)) = [15,45)$$

Therefore, we have:

$$I_1 = \text{Prob}(P_{ow}) = \text{Prob}(\text{Temperature} \in [15,45))$$

$$= 30\%$$

For defect #4 we have:

$$P_{ow} = (P_c \cap (\Omega \setminus P_o)) \cup (P_o \cap (\Omega \setminus P_c))$$

Where

$$P_o = \{\text{Status == true}, \text{Status == false}\}$$

$$P_c = \{\text{Status == true}\}$$

$$\Omega = \{\text{Status == true}, \text{Status == false}\}$$

Thus we have
\[ P_C \cap (\Omega \setminus P_O) = \emptyset \]
\[ P_O \cap (\Omega \setminus P_C) = \{\text{Status} == \text{false}\} \]

Therefore,

\[ P_{ow} = \{\text{Status} == \text{false}\} \]

Hence the Infection probability is:

\[ I_4 = \text{Prob}(P_{ow}) = \text{Prob}(	ext{Status} == \text{false}) \]

\[ = 66\% \]

The resulting \( I_1 \) and \( I_4 \) should be added to Table 37.

A.16. Infection and Propagation Data for the PIE Analysis for the SDD

Infection probability for Defect #3 and #4 calculated using OP-related data. All the other necessary data is fictitious and provided in Table 38.

Table 38 Infection and Propagation data required for openValve3’s PIE analysis

<table>
<thead>
<tr>
<th>Defect No.</th>
<th>Defect Type</th>
<th>Infection %</th>
<th>Propagation %</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Missing Instance of Function</td>
<td>100%</td>
<td>5%</td>
</tr>
<tr>
<td>2</td>
<td>Function with Incorrect Logic</td>
<td>12%</td>
<td>13%</td>
</tr>
<tr>
<td>3</td>
<td>Missing Predicate</td>
<td>(Should be calculated by the user)</td>
<td>7%</td>
</tr>
<tr>
<td>4</td>
<td>Incorrect/Ambiguous Predicate</td>
<td>(Should be calculated by the user)</td>
<td>15%</td>
</tr>
</tbody>
</table>

The Infection probability for defect #3 is calculated as follows.

\[ P_{ow} = \Omega \setminus P_C \]
For defect #4 we have:

\[ I_3 = \text{Prob}(P_{OW}) = 1 - \text{Prob}(<\text{Position}==3>) = 1 - 49\% = 51\% \]

For defect #4 we have:

\[ P_{OW} = (P_c \cap (\Omega \setminus P_o)) \cup (P_o \cap (\Omega \setminus P_c)) \]

\[ I_4 = \text{Prob}(P_{OW}) = \text{Prob}(((1,2) \cap \{2,3\}) \cup (\{1\} \cap \{3\})) \]

\[ = \text{Prob}(\{2\}) = 36\% \]

The probability values come from Table 35 and Table 36, i.e., the operational profile (OP-related) data. Once evaluated the resulting values for \( I_1 \) and \( I_4 \) should be added to Table 38.

**A.17. Calculation of SCS’s Software Reliability**

SCS’s software reliability can be evaluated using the previous formulas and data. E, I and P should be introduced into the reliability calculation formula. However, the Infection probability for defect #3 needs to be calculated using SDD-related information. More specifically, we have:

\[ \text{Prob}(\text{failure at SRS level}) = \sum_{i=1}^{n} E_i \times I_i \times P_i \]  \hspace{1cm} (A.17.1)

\[ = 100\% \times 30\% \times 7\% + 100\% \times 100\% \times 5\% + 73\% \times I_3 \times 10\% + 100\% \times 66\% \]

\[ \times 3\% \]

Where \( I_3 \) is evaluated using the same equation and the data is obtained from SDD analysis.

\[ I_3 = \text{Prob}(\text{failure at SDD level}) = \sum_{j=1}^{m} E_j \times I_j \times P_j \]  \hspace{1cm} (A.17.2)

\[ = 100\% \times 100\% \times 5\% + 85\% \times 12\% \times 13\% + 100\% \times 51\% \times 7\% + 100\% \times 36\% \]

\[ \times 15\% \]

\[ = 15.3\% \]

Replacing the above value in equation (A.17.1), we obtain:
\[ Prob(failure \ at \ SRS \ level) \quad (A.17.3) \]

\[
= 100\% \times 30\% \times 7\% + 100\% \times 100\% \times 5\% + 73\% \times 15.3\% \times 10\% + 100\% \\
\times 66\% \times 3\%
\]

\[ = 10.2\% \]

\[ Re = 1 - Prob(failure \ at \ SRS \ level) = 1 - 10.2\% = 89.8\% \quad (A.17.4) \]
Appendix B: Defect Templates for SRS/SDD Level Defect Mapping

In Appendix B defect templates for all defects identified in this study (summarized in Table 2) are provided. For each template, the upper branch is the original branch (defective/erroneous) and the lower branch is the modified branch (corrected). Note some templates are almost identical and therefore omitted.

B.1. Defect Templates for Category 1 Defects: Defects for the Definition of Level 1 Functions

B.1.1. Missing (Definition of) Function.

If the function whose definition is missing is never instantiated, no defect is introduced. Otherwise the defect template in Figure 158 should be used. This template is identical to that of “Missing Instance of Function”.

![Figure 158 Defect Template for Missing (Definition of) Function](image)

Figure 158 Defect Template for Missing (Definition of) Function
B.1.2. Extra (Definition of) Function

As for Missing (Definition of) Function, if the function whose definition is extraneous is never instantiated, no defect is introduced. Otherwise the template in Figure 159 should be used. It can be seen that it is exactly identical to “Extra Instance of Function”.

![Defect Template for Extra (Definition of) Function](image)

**Figure 159** Defect Template for Extra (Definition of) Function

B.1.3. Incorrect/Ambiguous Function Name

![Defect Template for Missing (Definition of) Function](image)

**Figure 160** Defect Template for Missing (Definition of) Function
In Figure 160, $F_O$ is the original function call with an incorrect/ambiguous name. The subscript “O” indicates that it is the original.

**B.1.4. Function with Incorrect Logic**

The defect template for a Function with Incorrect Logic is found in Figure 161.

![Figure 161 Defect Template for a Function with Incorrect Logic](image)

**B.1.5. Function with Incorrect Functionality**

The defect template for a Function with Incorrect Functionality is found in Figure 162.

![Figure 162 Defect Template for Function with an Incorrect Functionality](image)
B.2. The Defect Templates for Category 2 Defects: Defects Related to Inputs

B.2.1. Missing Input

The defect template for a function with Missing Input(s) is found in Figure 163.

In this case $F_{O,MI}$ is the function whose one or multiple inputs are missing. The subscript “MI” is an abbreviation for “Missing Input”.

B.2.2. Extra Input

The defect template for a function with Extra Input is displayed in Figure 164.
B.2.3. Incorrect/Ambiguous Input Name

The defect template for a function with Incorrect/Ambiguous Input Name is given in Figure 165.

B.2.4. Input with Incorrect Type

The defect template for a function with an Input with Incorrect Type is given in Figure 166.
B.2.5. Input with Incorrect Range

The defect template for a function with an Input with an Incorrect Range is given in Figure 167.

Figure 166 Defect Template for a function with Input with Incorrect Type

Figure 167 Defect Template for a function with an Input with an Incorrect Range
B.3. Defect Templates for Category 3 Defects: Defects Related to Outputs

Defect templates for Category 3 defects are similar to those of Category 2 defects. Therefore, only the template for Missing Output is provided (see Figure 168). The others can be derived easily from Figure 164 to Figure 167.

![Diagram of Defect Template for a function with a Missing Output]

**Figure 168** Defect Template for a function with a Missing Output

B.4. Defect Templates for Category 4 Defects: Defects Related to Internal Variables

Defect templates for Category 4 defects are also similar to those of Category 2 and 3 defects. Therefore, only the template for Missing Variable is provided (see Figure 169). The others can be derived easily from Figure 164 to Figure 167.
B.5. Defect Templates for Category 5 Defects: Defects for the Logic of the Level 0 Function

B.5.1. Missing Instance of Function

The defect template for a Missing Instance of Function is given in Figure 170.

B.5.2. Extra Instance of Function

The defect template for an Extra Instance of Function is given in Figure 171.
Figure 171 Defect Template for an Extra Instance of a Function

B.5.3. Incorrect/Ambiguous Function Call

The defect template for an Incorrect/Ambiguous Function is given in Figure 172.

Figure 172 Defect Template for an Incorrect/Ambiguous Function Call

B.5.4. Missing Predicate

The defect template for Missing Predicate is given in Figure 173.
B.5.5. Extra Predicate

The defect template for an Extra Predicate is given in Figure 174.

B.5.6. Incorrect/Ambiguous Predicate

The defect template for an Incorrect/Ambiguous Predicate is given in Figure 175.
Figure 175 Defect Template for an Incorrect/Ambiguous Predicate
Appendix C: Description of the Functions Used in the Algorithms

C.1 Functions Used in the Algorithm to Construct the Dependency Graph

This section introduces all the functions in the dependency graph construction algorithm. The sequence is the same as where they are called in the algorithm in Figure 134.

**assignStmtNumber(CurrStmt)**: This function recursively assigns a serial number to the statements/predicates “CurrStmt”.

**genNewNode(Var)**: This function generates a new node for “Var”.

**addNewNode(Graph, Node)**: This function adds a node “Node” into the graph “Graph”.

**getAllDeclVar(DeclStmt)**: This function obtains the variables declared in the “declaration” statement “DeclStmt”.

**getDeclExpr(DeclStmt)**: This function obtains the expression used to initialize the variable declared in the “declaration” statement “DeclStmt”.

**getAllReadVar(Var, Expr)**: This function obtains all variables from the expression “Expr” that are used to update the variable “Var”.

**buildLink(Node1, Node2)**: This function builds an edge from “Node1” to “Node2”.

**getNode(Graph, ReadVar)**: This function obtains the node for “ReadVar” from the graph “Graph”.

228
getAllModVar(ExprStmt): This function obtains all variables that are updated in the “expression” statement “ExprStmt”.

ggetAllStmt(BlockStmt): This function obtains all statements of the “block” statement “BlockStmt”.

gPred(Stmt): This function obtains the predicate of the “if”, “while” or “do-while” statement “Stmt”.

gTrueStmt(IfStmt): This function obtains the true body statement of the “if” statement “IfStmt”.

gAllAffectVarFromStmt(Stmt): This function obtains all the variables of “Stmt” that can be affected by an outer predicate.

gFalseStmt(IfStmt): This function obtains the false body statement of the “if” statement “IfStmt”.

gWhileBodyStmt(WhileStmt): This function obtains the body statement of the “while” statement “WhileStmt”.

gDoWhileBodyStmt(DoWhileStmt): This function obtains the body statement of the “do-while” statement “DoWhileStmt”.

gInit(ForStmt): This function obtains the “initialization” expression of the “for” statement “ForStmt”.

gTest(ForStmt): This function obtains the “test” expression of the “for” statement “ForStmt”.

gIncr(ForStmt): This function obtains the “increment” expression of the “for” statement “ForStmt”.

229
**getForBodyStmt(ForStmt):** This function obtains the body statement of the “for” statement “ForStmt”.

**getRet(RetStmt):** This function obtains the expression of the “return” statement “RetStmt”.

### C.2 Functions used in the algorithm to determine the data state

This section introduces all the functions in the algorithm to determine the data state. The sequence is the same as they are called in the algorithm in Figure 137.

**getAllFormalNodes(Graph):** This function obtains all the formal variable nodes from the dependency graph “Graph”.

**getAllKillNodes(Graph, CurrStmt):** This function obtains all the nodes that should be killed for the current statement “CurrStmt”.

**getAllLocalKillNodes(Graph, CurrStmt):** This function only works for loop statements. It obtains all the nodes that should be killed since they do not propagate data state different to the next iteration of the loop.

**getAllDeclNodes(Graph, CurrStmt):** This function obtains all the nodes for the variables that are declared in “CurrStmt”.

**getAllModNodes(Graph, CurrStmt):** This function obtains all the nodes for the variables that are modified in “CurrStmt”.

**getPredNode(Graph, Pred):** This function obtains the node for the predicate “Pred”.

**getInitNode(Graph, Init):** This function obtains the node for the “initialization” expression “Init” of a “for” statement.

**getIncrNode(Graph, Incr):** This function obtains the node for the “increment” expression “Incr” of a “for” statement.
getRetNode(Graph, CurrStmt): This function obtains the “return” node of the “return” statement “CurrStmt”.