2016-01-01

Box-Fusion: A Way to Enhance the Pairwise Testing Approach

Omar Ochoa
University of Texas at El Paso, omar.ochoa@gmail.com

BOX-FUSION: A WAY TO ENHANCE THE PAIRWISE TESTING APPROACH

OMAR OCHOA

Doctoral Program in Computer Science

APPROVED:

Ann Q. Gates, Chair, Ph.D.

Salamah Salamah, Ph.D.

Vladik Kreinovich, Ph.D.

Aaron Velasco, Ph.D.

Charles Ambler, Ph.D.
Dean of the Graduate School

Dedication

To the most important thing in my life,
Yanet

BOX-FUSION: AN APPROACH TO ENHANCE PAIRWISE TESTING

by

OMAR OCHOA, M.S.

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at El Paso

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

Department of Computer Science

THE UNIVERSITY OF TEXAS AT EL PASO

August 2016

Acknowledgements

I would like to thank my committee Dr. Ann Q. Gates, Dr. Salamah Salamah, Dr. Vladik Kreinovich, and Dr. Aaron Velasco for the support that they have shown me throughout the years. Thank you Dr. Aaron Velasco for being always available and ready to help with any issues that I had; you are a great mentor and friend! Thank you Dr. Vladik Kreinovich for being always available, always helpful, and always a joy to talk to. The impression you left on me when I took my first Computer Science class with you has taken me down this path. Thank you Dr. Salamah Salamah, for everything that you have done for me. Most importantly, thank you Dr. Ann Q. Gates for being a role model, mentor, teacher, guide and so many other things to me. Thank you for all that you have taught me!

I would also like to thank the Computer Science students, faculty and staff for the great work they do. The UTEP Computer Science Department is an excellent place, and I will miss being around it. Finally, I would like to thank my wife, Yanet. Without her none of this would have happened. Thank you and I love you.

This work was partially funded by the National Science Foundation through grants HRD-1242122 and CNS-1042341.

Abstract

Critical software systems that have failed due to the software errors are well documented. As our dependency on computer-based systems increases and such systems become more complex, software verification becomes even more important. Enhancing and improving the verification and defect correction techniques used in software engineering for the development of software systems is of utmost importance to keep pace with our increasing reliance on software.

Pairwise testing has emerged as an effective technique for software system-level testing that have large combinations of inputs, although a drawback is the lack of support for defect location. This research aims to increase the efficacy of the pairwise testing technique through a new approach called Box-Fusion that combines the capabilities of structural source code analysis with the ease of pairwise test input generation. The Box-Fusion approach generated annotated control-flow graphs that can support the identification of untested code, location of faults, and selection of test cases for regression testing. To evaluate the Box-Fusion approach, a case study was conducted using two software implementations: the Simple LTL Generator that builds Linear Temporal Logic (LTL) formulae with atomic propositions and the Prospec Algorithm that can generate LTL formulae from more than 31,000 possible input combinations. The case study evaluates the following four propositions: the Box-Fusion approach can determine the areas of code that require additional verification; the code defects identified by the pairwise test are located in the execution paths identified by the approach; analyzing the annotated control flow graphs can narrow the location of the defect; and the regression tests can be optimized through the Box-Fusion approach. The case-study results demonstrate the value added by the Box-Fusion approach. The approach can be expanded to other black-box techniques and can be adapted to support educational models on software verification.

Table of Contents

Acknowledgements v

Abstract vi

Table of Contents vii

List of Tables x

List of Figures xii

Chapter 1: Introduction 1

1.1 Research Problem 1

1.2 Research Goal 3

1.3 Organization of Dissertation 4

Chapter 2: Background 5

2.1 Software Testing Life-Cycle 5

2.1.1 Unit Testing 6

2.1.2 Integration Testing 6

2.1.3 System Testing 8

2.1.4 Performance Testing 8

2.1.5 Acceptance Testing 9

2.1.6 Installation Testing 9

2.1.7 Regression Testing 10

2.2 Software Testing Approaches 10

2.2.1 White-box Testing 10

2.2.2 Black-box Testing 14

2.2.3 Pairwise Testing 16

2.3 Other Common Verification Techniques 18

2.3.1 Formal Verification 18

2.3.2 Theorem Proving 18

2.3.3 Runtime Verification 19

2.3.4 Model Checking 21

Chapter 3: The Box-Fusion Approach 23

3.1 Overview 23

3.2 Definitions and Description of Basic Concepts 26
3.3 Processes 1-2: Box-Fusion Control Flow Graph Generation 29
3.4 Processes 3: Instrumentation....34
3.5 Process 4: Populating the BF-CFG 37
3.6 BF-CFG Analysis....41
3.6.1 Identification of Verification Gaps 41
3.6.2 Detecting Potential Defect Location 42
3.6.3 Test Case Selection for Regression Testing....44
Chapter 4 Case Study....46
4.1 Background 46
4.1.1 Linear Temporal Logic Background....46
4.1.2 LTL Formulas 47
4.1.3 SPS Algorithm 48
4.1.4 Prospec Algorithm 49
4.2 Research Questions and Propositions 50
4.3 Units of Analysis....51
4.4 Logic Linking....56
4.5 Criteria for Interpreting Findings....57
4.6 The Prospec Algorithm Implementations 58
4.6.1 Simple LTL Generator Implementation....59
4.6.2 Prospec Algorithm Implementation 59
Chapter 5: Results and Observations 61
5.1 Proposition 1 Analysis: The Box-Fusion Approach Determines Gaps in Testing 61
5.1.1 Measuring SUT Coverage....61
5.1.2 SimpleLTL Generator Coverage Analysis....63
5.1.3 Prospec Algorithm Coverage Analysis 64
5.1.4 Verification Gap Analysis....65
5.1.5 Gap Analysis of the Implementations....66
5.1.6 Summary of Analysis for Proposition 1....67
5.2 Proposition 2 Analysis: The Box-Fusion Approach Identifies Defect Location 67
5.2.1 SimpleLTL Generator Analysis 67
5.2.2 Prospec LTL Implementation Analysis 73

5.2.3 Summary of Analysis for Proposition 2 75
5.3 Proposition 3: Analysis of the Box-Fusion Approach To Narrow Defect Location 76
5.3.2 Summary of Analysis for Proposition 3 79
5.4 Proposition 4: The Box-Fusion Approach Can Optimize Regression Testing 80
5.4.1 Regression Testing 80
5.4.2 Summary of Analysis for Proposition 4 82
Chapter 6: Related Work 84
6.1. White Box Pairwise Test Case Generation 84
6.2. Coverage Tools 85
6.2.1. JCov a Java Code Coverage Tool 86
6.2.2. JaCoCo a byte-code analyzer tool for test coverage 86
6.2.3. JCover 87
6.2.4. Cobertura 87
6.2.5. Comparison with Box-Fusion approach 87
6.3. Regression testing 87
6.3.1. Test Suite Reduction 87
6.3.2. Dynamic Slicing Based Approach 88
6.3.3. Graph-Walk Approach 89
6.3.4. Test Suite Minimization 90
Chapter 7: Conclusions 91
7. 1 Summary of Work 91
7. 2 Intellectual Merit and Broader Impacts 94
7. 3 Future Work 95
References 96
Appendix A 101
Appendix B 109
Appendix C 114
Appendix D 116
Vita 119

List of Tables

Table 2. 1 Pairs for Three Variables with Two Input Values Each. 17

Table 2. 2 Tests for Three Variables with Two Input Values Each. 17

Table 3. 1 Definitions of Terms and Concepts. 26

Table 3. 2 CFG2BF-Mapping of Properties to Inspection Tasks. 33

Table 3. 3 Instrumentor Mapping of Properties to Inspection Tasks. 36

Table 3. 4 PopulateCFG Mapping of Properties to Inspection Tasks. 40

Table 4. 1 Composite Propositions. 49

Table 4. 2 Breakdown of Property Types. 50

Table 4. 3 Artifacts Used to Collect Data. 55

Table 4. 4 Data Collected for Each Proposition 56

Table 4. 5 Simple LTL System Size Metrics 59

Table 4. 6 Implemented System Size Metrics 60

Table 5. 1 CoverageMiner Mapping of Properties to Inspection Tasks. 63

Table 5. 2 Branch Coverage for the Simple LTL Generator. 63

Table 5. 3 Branch Coverage for the Prospec Algorithm Implementation 64

Table 5. 4 Branch Coverage for Different Pairwise Test Suites 64

Table 5. 5 VerificationGapMiner Mapping of Properties to Inspection Tasks. 66

Table 5. 6 *ExecutionPathFinder* Mapping of Properties to Inspection Tasks. 69

Table 5. 7 SimpleLTL Generator Defects Seeded. 70

Table 5. 8 Data Collected While Correcting Seeded Simple LTL Generator. 72

Table 5. 9 Participant 1 Data Collected While Correcting the Prospect Algorithm Implementation. 73

Table 5. 10 Participant 2 Data Collected While Correcting the Prospect Algorithm Implementation. 74

Table 5. 11 CFGIntersectionMethodFinder Mapping of Properties to Inspection Tasks. 77

Table 5. 12 Calculating the Intersecting Method Numbers for the Defect Group. 78

Table 5. 13 RegressionFinder Mapping of Properties to Inspection Tasks. 81

Table 5. 14 Results of Regression Steps on SUT. 82

Table A. 1 Formal Verification Tasks List for the Instrumentor implementation. 101

Table A. 2 Formal Verification Tasks List for the InstrumentingCode implementation. 102

Table A. 3 Formal Verification Tasks List for the CGF2BF implementation. 103

Table A. 4 Formal Verification Tasks List for the CoverageMiner implementation. 105

Table A. 5 Formal Verification Tasks List for the VerificationGapMiner implementation. 105

Table A. 6 Formal Verification Tasks List for the ExecutionPathFinder implementation. 106

Table A. 7 Formal Verification Tasks List for the CFGIntersectionFinder implementation. 107

Table A. 8 Formal Verification Tasks List for the RegressionFinder implementation. 107

Table B. 1 Tests Generated, Including IDs and CPs for the Prospec Algorithm. 109

Table B. 2 Tests Generated, Including IDs and CPs for Simple LTL Generator. 112

Table C. 1 Simple LTL Generator Coverage. 114

Table C. 2 Prospec Algorithm Coverage. 115

List of Figures

Figure 2. 1 Testing Life-Cycle. 5

Figure 2. 2 Sample Calling Unit Hierarchy. 7

Figure 2. 3 Top-Down Approach Using Stubs. 7

Figure 2. 4 Bottom-up Approach Using Drivers. 8

Figure 2. 5 Sample Code and CFG. 11

Figure 2. 6 Branch Coverage CFG. 12

Figure 2. 7 Path Coverage CFG. 13

Figure 2. 8 Boundary-Value Analysis Example. 15

Figure 2. 9 Equivalence Class Example. 15

Figure 2. 10 Percent of Defects Associated with Multiple Number of Inputs. 16

Figure 2. 11 High-Level View of a Dynamic Monitor [19]. 20

Figure 2. 12 Model Checking Process [14]. 21

Figure 3. 1 DFD Model for the Box-Fusion Approach. 25

Figure 3. 2 CFG Generated by the CFGF Tool. 28

Figure 3. 3 CFGF XML for a CFG. 28

Figure 3. 4 Example of a Populated Edge Node. 30

Figure 3. 5 Example of a Populated Origin Node. 30

Figure 3. 6 Side by Side CFG and BF-CFG XML Files. 31

Figure 3. 7 Instrumentation Algorithm. 34

Figure 3. 8 Sample Instrumented Code. 35

Figure 3. 9 Corrected Instrumentation Code 36

Figure 3. 10 Statechart Snippet of the PopulateCFG Algorithm that Tracks Traversed Edges 38

Figure 3. 11 Example BF-CFG with Marked/Unmarked Edges. 41

Figure 3. 12 Example Code and CFG for Letter Grade Calculation. 42

Figure 3. 13 Test ID Marked on a BF-CFG for Letter Grade Calculation. 43

Figure 3. 14 Example CFG with Modified Node. 44

Figure 4. 1 Pairwiser Parameter Inputs 52

Figure 4. 3 Constraints Used in Pairwiser. 53

Figure 4. 2 Sample of the Generated Test Cases. 53

Figure 4. 4 Atomic Prospec Pairwise Tests. 54

Figure 5. 1 Sample Results File. 62

Figure 5. 2 Small Output of the VerificationGapMiner 65

Figure 5. 3 Unreachable Code Example at Line 406. 66

Figure 5. 4 Example of Output of the ExecutionPathFinder Implementation. 68

Figure 5. 5 Seeded SimpleLTL Output. 71

Figure 5. 6 Defect Before and Correction at line 30. 75

Figure 5. 7 Sample Output of the CFGIntersectionMethodFinder. 78

Figure 6. 1 Performance Gain with WBPairwise. 85

Chapter 1: Introduction

As our dependency on computer-based critical systems continues to increase, software systems will continue to grow in complexity and size; thus, it is more vital that efforts in software development focus on decreasing the number of defects within developed software systems. It has been reported, however, that a significant number of software systems do not meet users' requirements and are of poor quality [1]. In addition, a study from the National Institute of Standards and Technology revealed that software defects are so prevalent and detrimental that they cost the U.S. economy an estimated $59.5 billion annually, or about 0.6 percent of the gross domestic product [2]. Many other examples have been documented of cost increases caused by software errors [3]. Besides the monetary loss, failures in software can also be deadly [4]. Enhancing and improving the verification and defect correction techniques used in software engineering for the development of software systems is of utmost importance to keep pace with our increasing reliance on software.

1.1 Research Problem

One of the key aspects in software development is the assurance that the software behaves correctly, i.e., as specified by requirements. Multiple techniques exist for achieving this, e.g., theorem proving, runtime verification, model checking, and software testing. Software testing is by far the one most commonly applied approach. A test case describes a specific set of input values and the expected output. Fundamentally, testing follows the general pattern of:

1) Create a test case by:
    a. defining a specific input,
    b. providing instructions on how to interact with the system specific to this test case, and
    c. defining expected behavior, i.e., specific output.
2) Execute the software system using the input of the test case.
3) Compare expected behavior with actual behavior.

If the expected behavior does not match the actual behavior, then a defect is deemed to exist within the system. When the defect is corrected, regression testing is applied to verify that the defect was removed and no additional defects were added.

The primary challenge in the application of software testing arises from the difficulty of defining the input to a test case and the impracticality of exhaustively testing all possible inputs. A test must be efficient in such a way that it either exposes a defect or gives some level of confidence on the absence of defects. The problem in determining efficient inputs for test cases has been tackled by different approaches, ranging from treating the system as a black-box, which focuses on selecting input for test cases from the specified requirements and input ranges, to white-box approaches, which requires analysis of the structure of the source code in order to select a set of input for test cases that provides adequate coverage of the code under test. Each of these approaches has advantages and disadvantages. In black-box testing approaches, it is easier to determine inputs for a test case, but a failed test case will not always provide direction on the potential location of the defect in the system. On the other hand, a white-box test case can be harder to define, but it narrows down the execution of the test to a small set of execution paths where the defect might exist. An execution path is the path that an instance of an execution travels through the system from start to end. Software developers typically select a combination of approaches.

Software systems that have a large combination of possible input values are typically too big to test all combinations of inputs. For such software systems, pairwise testing [5], a black-box technique based on combinatorial testing, is useful because it focuses on testing all pairs of test case inputs instead of all combinations of inputs. The generation of test cases under pairwise testing is mechanical in nature, only ensuring that each pair of input variables is accounted for. This leads to a practical way of generating test case input data for testing a wide range of systems with large combination of inputs.

Once a defect has been located and corrected, testing again is needed to determine if 1) the defect was successfully corrected and 2) no new defects were introduced by the code change. This

can be particularly challenging with a black-box approach as there is no way to determine which inputs were affected by the code change. Contrasting this, a white-box approach can assist developers in determining which code was not modified and does not require retesting, reducing the time and effort required to apply regression testing. An approach that can combine the capabilities of both black-box and white-box testing approaches can lead to an increase in the efficacy of testing, defect location and correction, and consequently, improving the quality of developed software systems.

1.2 Research Goal

Pairwise testing has emerged as an effective technique for testing programs with a large combination of inputs. However, due to pairwise testing being a black-box approach, the tester cannot always easily discern the location of faulty code or untested code, limiting the capability of pairwise testing [48]. Other approaches have attempted to enhance pairwise testing [41], or provide coverage measurements [44, 46, 47], or facilitate the identification of regression testing [51, 53, 55]. In comparison, this approach focuses on the pairwise testing technique. The **goal** of the research is to increase the efficacy of the pairwise testing technique by defining an approach that combines the capabilities of structural source code analysis with the ease of pairwise test input generation. To achieve this goal, the research posits that the inclusion of control-flow analysis can guide the identification of untested code, fault location, and selection of test cases for regression testing.

The research questions (RQ) are as follows:

RQ 1. How can pairwise testing be augmented to direct the developer to potential areas of faulty code, untested code, and related test failures?

RQ 2. How can pairwise test cases be identified to achieve the appropriate coverage required by regression testing?

The research presented in this dissertation presents the Box-Fusion approach, an enhancement to the pairwise testing technique by combining it with the capabilities of structural

source code analysis. The **expected outcomes** of the research is an enhanced pairwise testing approach that has the following features:

- Facilitates the identification of gaps in verification.
- Provides guidance on the location of defects in source code.
- Assists in the selection of test cases that can be used for regression testing.

1.3 Organization of Dissertation

The organization of this dissertation is as follows. Chapter 2 provides the background on the various strategies and approaches in software verification. Chapter 3 describes the Box-Fusion approach. Chapter 4 describes the case study used in the analysis of the Box-Fusion approach, with the necessary background on the algorithm chosen for the study. Chapter 5 discusses the observations and results of the case study conducted in this dissertation. Chapter 6 describes related work on existing approaches that are related to the Box-Fusion approach. Chapter 7 presents the summary of the work and the future direction of this research.

Chapter 2: Background

This chapter describes techniques and strategies that comprise software verification. This chapter is divided into four major sections. The first section introduces software testing, which is the most common software verification technique, and the software testing life-cycle. The second section expands on the different testing techniques used in software testing, categorized by black-box and white box testing. The third section presents two other very common verification techniques, walkthroughs and inspections. The last section describes other more formal software verification techniques.

2.1 Software Testing Life-Cycle

A key aspect in the software development process is the assurance that the software behaves correctly, i.e., as specified by requirements. There are multiple techniques that are used as part of software quality assurance. Different testing approaches are used in each of the software development life-cycle phases. The following sections detail the different life-cycle phases associated with software testing and highlight the differences between each of the techniques used in each of the phases. Fig. 2.1 below shows the different phases of testing that will be described in the following subsections.

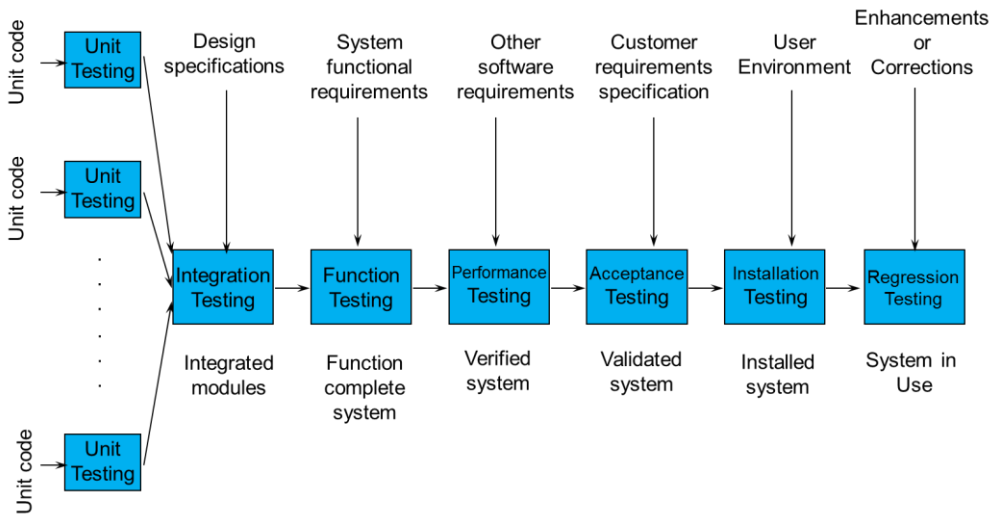


Figure 2. 1 Testing Life-Cycle.

2.1.1 Unit Testing

The purpose of unit testing [6, 7] is to test individual software units (components) independently of each other. A unit is typically defined as a class or a function. This type of testing aims at ensuring the correctness of each independent unit required for the correctness of the entire system. Unit testing allows for the early discovery of defects present in individual units before the integration of these units into the complete system. The small size of units makes it easier to detect defects compared to detecting defects in the whole system.

Another benefit of unit testing is that often developers who are responsible for creating the code are the ones that execute the unit testing, which can facilitate the defect correction process since these developers are the most familiar with the code. There are different approaches that can be applied to unit testing, Section 2.2 will expand on these.

2.1.2 Integration Testing

As each unit is successfully tested, it is then necessary to integrate the units into the larger subsystems that make up the whole system [8], refer to Fig. 2.1. This integration depends on the design specifications and design decisions that determined how units are coupled and how they interface in their communication. The process of assuring that each component has the correct interface implementation is called integration testing. Test cases are developed to test that all interface communications between components is correct. There are multiple approaches to integration testing including big-bang integration, top-down integration, and bottom-up integration.

In big-bang integration, all units are integrated at once and then tested for correctness of communication. This approach is only applicable for small systems as defective interfaces can be easily located. In top-down and bottom-up approaches, the system units are related by the "uses" relation; unit u_1 "uses" unit u_2 if u_1 makes a call to the elements of u_2. The units can be thought of as a hierarchy where the calling unit (u_1) is at a higher level than the called unit (u_2) as shown in Fig 2.2 below.

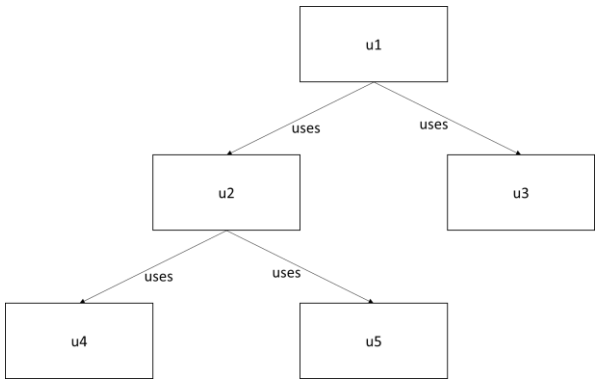


Figure 2. 2 Sample Calling Unit Hierarchy.

In the top-down approach, units are integrated from the highest level to the lowest; units at the highest level (level 1) are integrated with those units used by the units in level 1. To support the top-down approach, stubs are used to provide the data necessary to the higher level components as shown in Fig. 2.3. Stubs temporarily replace the functionality of those lower level units that are yet to be implemented or integrated.

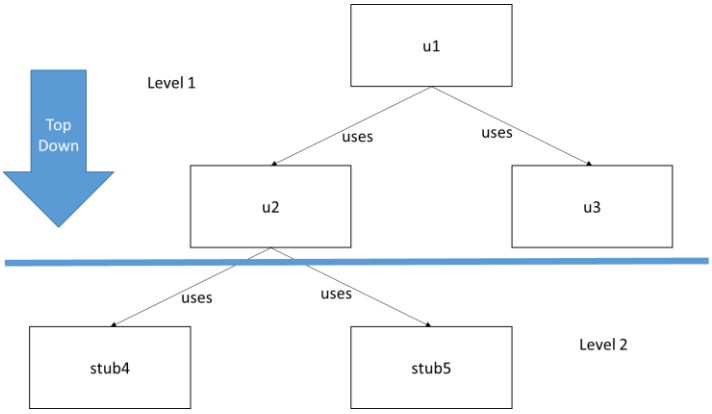


Figure 2. 3 Top-Down Approach Using Stubs.

In the bottom-up approach, lowest-level components are integrated and tested, then joined with higher-level components and tested. Drivers are used to pass communication between low level components, as shown Fig. 2.4 below.

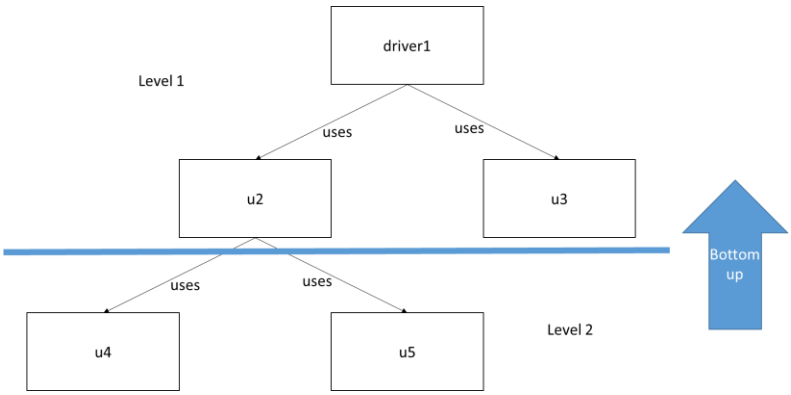


Figure 2. 4 Bottom-up Approach Using Drivers.

It is important to note that top-down approach allows for testing major design decisions which are, typically, represented in the higher level components. On the other hand, the bottom-up approach allows for more through and early testing of the concrete functionality and algorithms which are typically present in the lower level components

2.1.3 System Testing

Integration testing will only test the integration of components, any further assessment of the functional correctness is done at subsequent phases. Once each subsystem has been fully integrated and tested, as shown in Fig 2.1, the system as a whole is tested. This process is called system testing. The purpose of system testing is to verify that all the functionality been correctly implemented. Test cases are generated from the requirements specifications, and they are black-box type of tests. These tests often are created and executed by an independent team. System tests are aimed at verifying that the system correctly implements the high-level requirement specifications, which can include testing functionality that was not explicitly specified. Ultimately, the goal of this testing phase is to ensure that the system contains the correct functionality requested by the customer.

2.1.4 Performance Testing

Performance testing is the process of testing a system to evaluate how a system performs in terms of qualitative attributes such as:

- Reliability
- Scalability
- Performance
- Availability
- Interoperability
- Security

Test cases are generated that aim to measure if these quality attributes meet the non-functional requirement specifications as requested by the stakeholders. As shown on Fig. 2.1, in conjunction with system testing, the output of this phase is a system that has been verified and validated, meeting the requirement specifications.

2.1.5 Acceptance Testing

Acceptance testing is the testing process used to determine if the customer accepts the implementation of the system. In this phase, as shown in Fig. 2.1, the customer is responsible for validating the functionality of the system, i.e., that the system's functionality delivered is the functionality that the customer requested. The environment that the tests are performed must be very close to the actual environment that the customer or end-users will use. The outcome of this phase is the acceptance of the system by the customer signifying that the system meets their full expectations.

2.1.6 Installation Testing

In this phase, the system is tested to ensure that the system can and will be successfully installed on the customer's specified environment, referred to as the *production* environment. Some tasks during installation testing include ensuring that the pre-requisite software and hardware resources are available, that the system is correctly deployed and communicating with other systems, that the system is behaving as expected in the new environment, and that there is minimal impact to existing systems. The result of this process is a system that is now ready to be used.

2.1.7 Regression Testing

The more a software system is in use, the higher the likelihood of new modifications or enhancements being requested. Additionally as systems get used, previously undetected defects will be discovered and must be corrected. It is typically the case that in the process of maintaining a software system, new defects are introduced into the code by the modifications or corrections. To protect against this, regression testing is used to ensure that defects have not been introduced with changes to the system. Both tests related to the change and tests unrelated to the change, such as previously passed tests, are executed on the system in order to check that unwanted changes were not introduced. One of the challenging aspects of regression testing is selecting what to test because retesting the entire system can be inefficient, and testing only the changes might not detect the full impact of the change on the system.

2.2 SOFTWARE TESTING APPROACHES

The challenges in the application of software testing arises from both the difficulty of defining the input to a test case and the difficulty of generating the expected output. The brute force approach of testing all possible inputs is rarely possible in practice. The generation of test cases must be efficient such that it either exposes a defect or gives some level of confidence regarding the absence of defects. As described earlier, the problem of defining such test cases has been tackled by treating the system as a black-box (black-box or functional testing) to analyzing and building tests based on the structure of the software (white-box or structural testing). In the following sections, techniques associated with each approach are described in more detail.

2.2.1 White-box Testing

A white-box testing approach requires the analysis of the internal structure of a component in order to select sets of input that provide adequate coverage of the source code. Determining test input data in white-box testing can be difficult because it requires construction of control flow graphs (CFG) from the source code to determine what input must be used to meet the desired coverage criteria.

A CFG G is defined as a finite set N of nodes and a finite set E of edges. An $edge_{(i, j)}$ in E connects two nodes n_i and n_j in G. $G = (N, E)$ denotes a flow graph G with nodes given by N and *edges* by E. Each node of the graph represent a basic block of code, i.e., code that does not have any transfer of control. Edges are used to indicate the flow of control between blocks. An $edge_{(i, j)}$ connecting basic blocks n_i and n_j implies that control can go from block n_i to block n_j. Fig. 2.5 is an example code and its corresponding CFG.

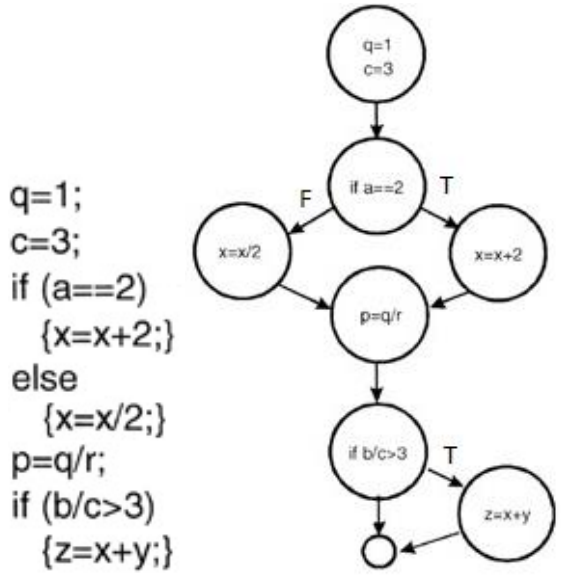


Figure 2. 5 Sample Code and CFG.

The coverage criteria describes how to use the CFG to guide the selection of test cases. The weakest criterion is called statement coverage. This criterion states that to achieve 100% coverage, each node in the CFG must be executed at least once by a test case [6]. One of the weakness of this approach is shown by a simple if-statements, such as:

```
if (condition==true)
    doSomething();
```

Statement coverage will not test the *false* evaluation of the if-statement because in order to execute the node containing the doSomething() function, the if-statement has to evaluate to *true*.

Branch coverage criterion is one that requires that each edge of every control structure be traversed at least once by a test. Test cases are generated by ensuring that every edge in the CFG has been touched at least once. Using the example in Fig. 2.6, two test cases are needed achieve 100% branch coverage. Test case 1, with input of a=2 and b=30, will touch all the edges marked by the green arrows. Test case 2, with input a=4 and b=3, will touch all the edges marked by the red arrows. Branch coverage is a stronger criterion for testing than statement coverage as it covers both true and false results of conditions (i.e., a set of test cases that ensure branch coverage is guaranteed to ensure statement coverage).

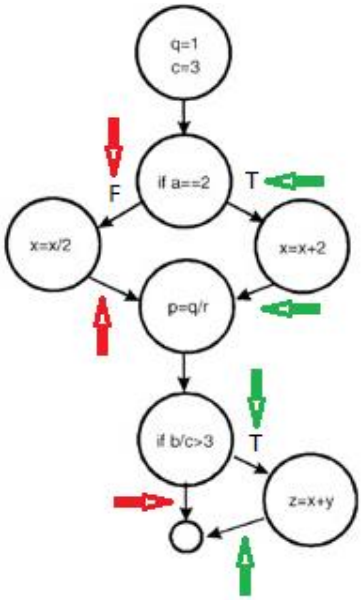


Figure 2. 6 Branch Coverage CFG.

The strongest coverage criterion is that of path coverage, which requires that every possible path is executed at least once. This coverage criterion is deemed the most expensive type of coverage as the number of tests required grows exponentially with the number of condition statements in the source code. Fig. 2.7 describes the four paths necessary to test to achieve 100% path coverage. Test case 1, with input of a=2 and b=30, will follow the path marked with the blue line. Test case 2, with input a=4 and b=3, will follow the path marked with the red line. Test case

3, with input of a=2 and b=3, will follow the path marked with the green line. Test case 4, with input a=4 and b=30, will follow the path marked with the yellow line.

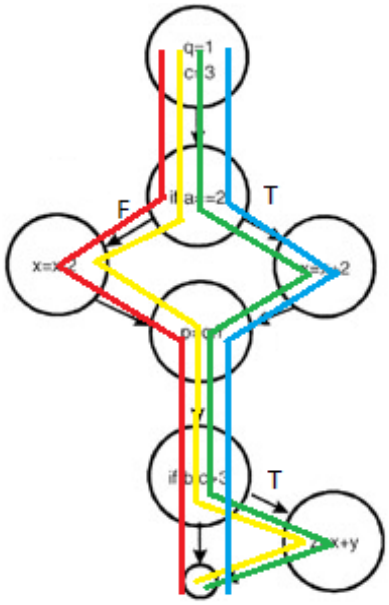


Figure 2. 7 Path Coverage CFG.

Another coverage criteria is def-use coverage. Def-use focuses on defining test cases that ensure that paths through the definition and use of variables in a program. The def-use technique requires that testing cover every path from every variable definition to every use of that definition. As is described by the definition, def-use allows testing to focus on a particular critical variable of interest. For example, if at some point in the code, the assignment statement "x = 1/y" exists, it becomes imperative that the value of *y* never reaches zero. As such, def-use can be used over variable *y* to ensure that for any definition of *y* to any use of *y*, the variable will never reach zero.

Other white-box coverage criteria include condition coverage, all-uses, all predicate uses/some computational uses, and all computational uses/some predicate uses [9]. One of the main advantages of a white-box approach is that since test cases are associated with the sections of the CFG followed, a test case that fails will be associated with the part of the CFG used to

generate that test case, and thus potentially framing the location of where the defect in the code exists.

2.2.2 Black-box Testing

The black-box testing approach primarily focuses on selecting input for test cases from the specified requirements. An advantage of the this testing approach is the fact that test cases can be generated as soon as the component specifications have been established without the need for a complete implementation. Another advantage of using this approach is that it is typically easier to determine inputs for a test case by treating the component as a black test without the need to examine the source code. A disadvantage is that a failed test case (one where the actual behavior does not match the expected one) will not identify the potential location of the defect in the system.

The strategies associated with the black-box testing approach are equivalence class testing and boundary-value analysis. Equivalence class testing is a strategy in which the input domain is partitioned into a finite number of valid and invalid classes. Input values within each equivalence class are expected to behave the same, and as such, it is sufficient to run a single test from each equivalence class. By executing one test from each equivalence class, the amount of testing necessary to cover all types of inputs is reduced. An example of such testing can be seen in Fig 2.8 below. In this example, the specification states that "Children under 2 ride the bus for free. Young people (those between 2 and 14 year old) pay $10, Adults (those between 15 and 64) pay $15, and Senior Citizen pay $5." Four equivalence classes are defined, with each equivalence class having a price value associated. This reduces the total testing to four tests.

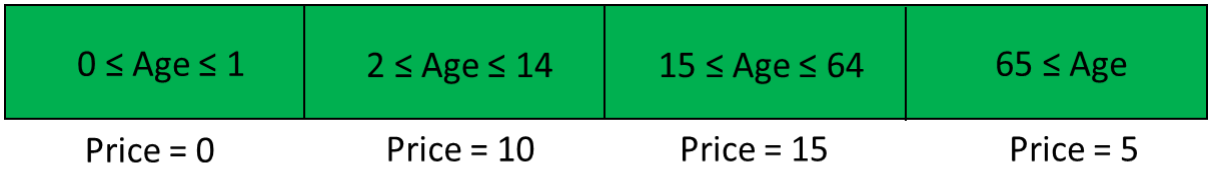


Figure 2. 9 Equivalence Class Example.

Another strategy is boundary-value analysis, which selects test inputs from the range of input values around the boundary. The values for a set of test cases are selected by identifying a value from above, one from below, and one at the boundary of the range. Software defects are commonly encountered at these boundaries. An example of such testing can be seen in the Fig. 2.9 below, where a total of nine tests are used to test each boundary.

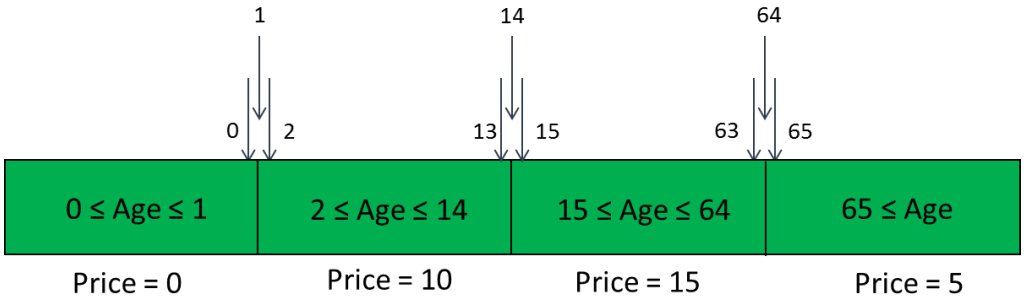


Figure 2. 8 Boundary-Value Analysis Example.

Equivalence classes can be used to define the boundaries used in boundary-value analysis testing. Combining both techniques can reduce the number the testing while retaining a high probability of discovering defects.

2.2.3 Pairwise Testing

Software systems of any significance are typically too large to be tested for all possible input combinations. For these software systems, pairwise testing [5] is useful because it focuses on testing all pairs of test case inputs instead of all combinations of inputs. This is a powerful approach because most defects are associated with one input or two input parameters interacting with each other. Fig 2.10 below shows that a significant number of defects are associated with one or two parameters [10]. Because pairwise testing is a black-box testing technique, the selection of test case inputs is mechanical in nature; the only requirement is that each combination of values of each pair of input variables account for at least one test case.

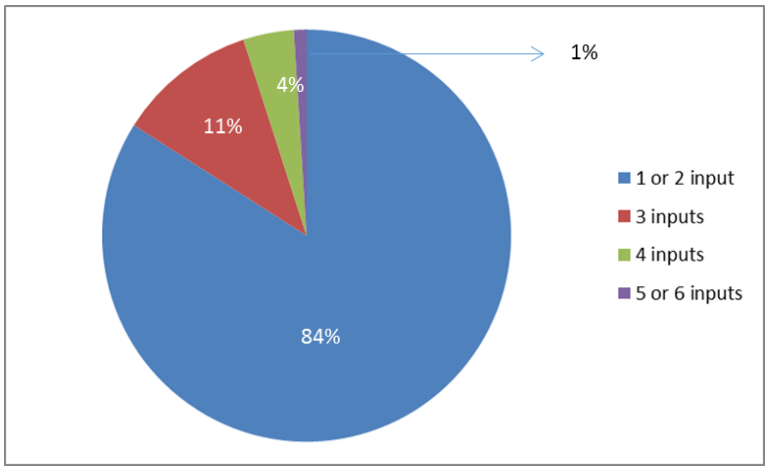


Figure 2. 10 Percent of Defects Associated with Multiple Number of Inputs.

For example, consider a hypothetical system with three variables a, b, and c, in which each variable can assume the values of 1 or 2. In order to achieve exhaustive testing of such a system a set of 23 test cases (eight test cases) needs to be defined, one for each possible combination of a, b, and c, as shown in Table 2.1 below. However, using pairwise testing, a set of test cases that ensures that each of all possible values for each pair is represented in the final set of tests. Using the previous example, this can be accomplished using a set of four tests instead of eight as shown in Table 2.2.

Table 2. 2 Tests for Three Variables with Two Input Values Each.

a	*b*	*c*
1	1	1
1	1	2
1	2	1
1	2	2
2	1	1
2	1	2
2	2	1
2	2	2

In Table 2.2, it can be observed that each possible combination of values for the variable *a* and *b* is produced in at least one test. For example, by only looking at the values in column *a* and *b*, all four possible combinations of values are present. This is also the case for *a* and *c*, as well as *b* and *c*, thus each pair is accounted for in at least one test.

Table 2. 1 Pairs for Three Variables with Two Input Values Each.

a	*b*	*c*
1	1	1
1	2	2
2	1	2
2	2	1

2.3 Other Common Verification Techniques

Besides testing, other common verification techniques include the use of review, specifically two types of reviews used for verification are inspections and walkthroughs. Reviews of work aim at examining the artifacts created, looking for flaws or improvements that can be addressed, and usually done by peers and experts. An inspection is a formal review that follows a well-defined set of steps, in which each participant is assigned a specific role in the review [11]. A walkthrough is an informal type of review, i.e. it doesn't follow a pre-define process and participants don't have specific roles [12]. Both of these techniques have been shown to be effective at detecting defects in systems [13].

2.3.1 Formal Verification

While testing remains the most commonly used verification technique, there exists other techniques including formal approaches to software verification. Ideally, formal methods provide a greater assurance over testing. The use of formal methods assist developers in managing the complexity of the system and in developing more reliable systems. Formal methods depends on formal specifications which are mathematically-based and, therefore, unambiguous. Formal methods can also be used to derive properties and detect inconsistencies in the specifications through the use of formal verification techniques and tools [14]. There are multiple types of formal verification techniques. This section introduces three of the most common types of verification techniques that use formal specifications; theorem proving, runtime verification and model checking. The following subsections provide a description of each of these techniques.

2.3.2 Theorem Proving

Theorem proving is a formal verification technique in which both the system behavior and the desired properties are expressed as formulas in a mathematical logic that define a set of axioms and a set of inference rules [15, 16]. The process of theorem proving consists of finding a proof that the desired property can be derived from axioms representing the system behavior. Although part of the process may be automated, the user must provide a sketch of a proof, which requires a

great degree of mathematical sophistication on the part of the user. As the size of the system increases, the complexity of the proof and the user interaction with the theorem prover increases. This requires that users have strong mathematical background to exploit theorem provers successfully. Still, there are several industrial projects where theorem provers are being applied. As described in Clark's survey [17], the Verity verification tool was used at IBM for process design; the ACL2 theorem prover was used for specifications and verification of a DSP microprocessor; and the PVS theorem prover was used for specifications and verification of the AAMP5 microprocessor. Another well-known theorem prover is the Stanford Temporal Prover (STeP) [18]. STeP allows for the formal verification of concurrent and reactive systems against specifications represented as temporal logic formula. STeP combines the deductive approach of theorem provers with the idea of model checking, visual representation of formulas, automatic generation of invariants, simplification, and verification rules. STeP automates most of the verification processes, provides users with visual verification diagrams, and provides support for hierarchical construction of proofs.

2.3.3 Runtime Verification

Runtime verification [19, 20, 21] is another technique that provides assurance that the current computation preserves the formally specified properties. The use of this approach neither proves the correctness of a system nor asserts a particular property for all possible states of a system. Runtime verification continuously checks whether the specified system property has been violated during a particular computation. Some disadvantages of runtime verification are the impacts on the target system, e.g., the ordering and timing of events, and on the overall effect on performance or computation time. Runtime verification is not adequate in verifying certain properties such as liveness properties [14]. Runtime verification, however, adds a layer of assurance that the system is behaving with respect to the software property specification.

The major parts of runtime-verification are: the specification of the desired system properties, the monitor and the event handler. Fig. 2.13 shows the interaction between these three

elements [19]. System properties, described as formal specifications, capture properties of the program behavior and are distinguished from the actual program. The specification language used by the monitor is often different than the language used to implement the monitored system. The

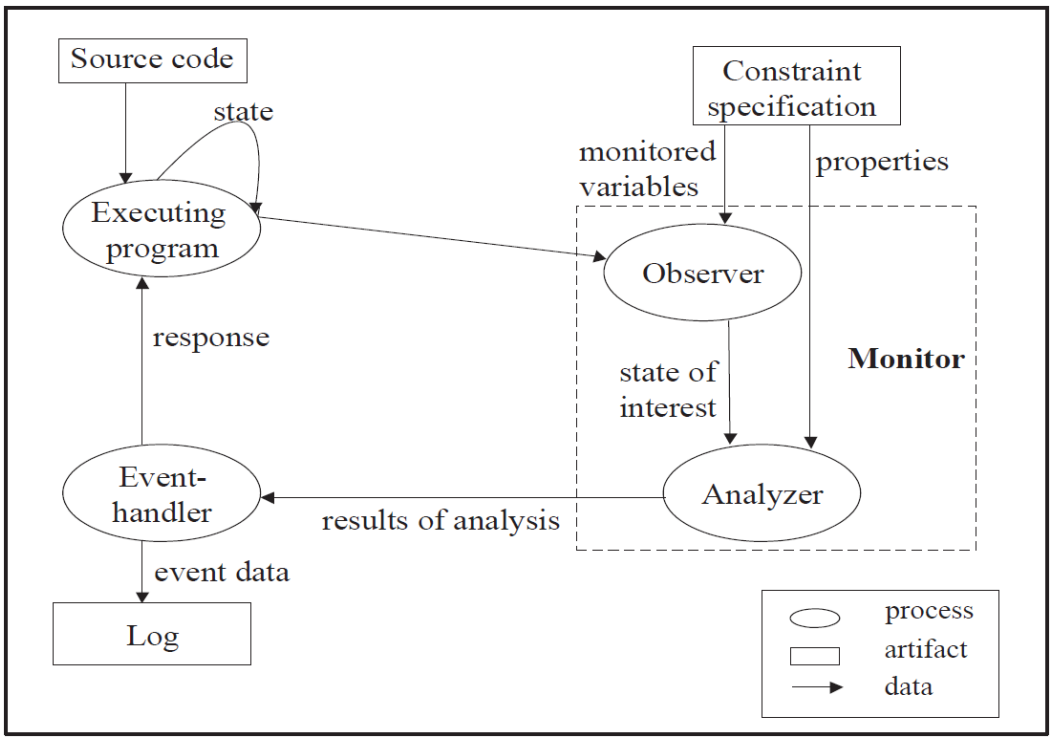


Figure 2. 11 High-Level View of a Dynamic Monitor [19].

specification languages of the runtime monitor vary based on the type of software properties that can be specified and the support for assertion at different levels within the program, e.g., module, event, and statement.

The second element, the monitor, has two components: the observer and the analyzer. The monitor is responsible for checking the system's current state of computation and evaluating whether the specified properties are preserved in this current state. The event-handler captures and communicates results to the system and/or user, and reacts to results.

2.3.4 Model Checking

Model checking [22, 23, 24] is a formal technique for verifying finite-state concurrent systems. The approach of model checking is to check the consistency of the system with the system specifications in every possible execution of the system. Unlike theorem proving, model checking is completely automated.

The process of model checking consists of three tasks: modeling, specification, and verification. The modeling phase consists of converting the design into a formalism that is accepted by the model checker. In some cases, modeling is simply compiling the source code representing the design. In most cases, however, the limits of time and memory mean that additional abstraction is required to come up with a model that ignores irrelevant details.

As part of model checking a system, it is necessary to specify the system properties to be checked. Properties are commonly expressed in some logical notation. Usually, temporal logic is used in the case of hardware and software systems, as this type of logic allows for reasoning about time, which becomes important in the case of reactive systems. In model checking, specifications are used to verify that the system satisfies some behavior. It is impossible to decide whether the given specification covers all the properties that the system should satisfy [25].

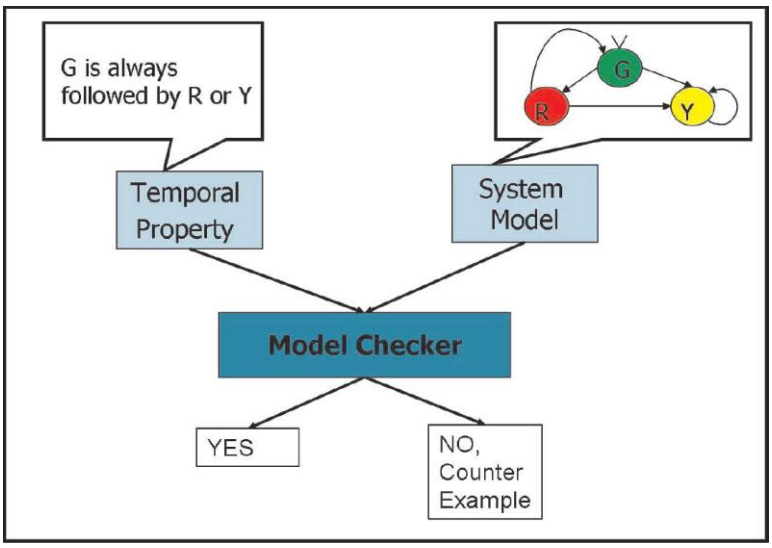


Figure 2. 12 Model Checking Process [14].

Once the system model and properties are specified, the model checker verifies the consistency of the model and specification. The model checker relies on building a finite model of the system and then an algorithm traverses the system model to verify whether the desired property holds in very execution of the model [26]. If there is an inconsistency between model and the property being verified, a counter example is provided to assist in identifying the source of the error. Fig. 2.12 shows the process of model checking.

The main problems in model checking are the difficulties in the formalization of system properties, the creation of an abstract model of the system, and the explosion in the state space that must be explored. System models may be represented by a special type of finite state-transition graphs called Kripke structure [25]. By using special data structures, e.g., binary decision diagrams that represent Boolean functions as a rooted, directed acyclic graph, it is possible to verify properties of complex systems.

Model checking has been widely used to verify sequential circuit designs and communication protocols at companies such as AT&T and Intel. The two model checkers most commonly used are the SPIN model checker [23, 27] and the Symbolic Model Verifier (SMV) [27]. SMV is used more in model checking of hardware systems, where SPIN is considered a model checker for software systems.

The SPIN model checker [23] was written in 1989 at Bell Labs. SPIN makes use of the high-level language PROMELA (PROcess MEta LAnguage) to specify system behavior. SPIN checks the logical consistency of a specification and report deadlocks, incompleteness and race conditions. Correctness requirements can be expressed as Linear Temporal Logic (LTL) formulas, as process invariants, or indirectly, as Büchi automata.

SMV was developed at Carnegie Mellon University and makes use of symbolic rather than explicit state representations [27]. In SMV, the user models the system as a set of states and guarded transitions, and specifies the desired properties in Computational Tree Logic (CTL). SMV makes use of the Ordered Binary Decision Diagram (OBDD) and handles non-deterministic behavior [25].

Chapter 3: The Box-Fusion Approach

This dissertation presents the Box-Fusion approach that enhances the pairwise testing technique. This chapter organizes the sections by the features of the approach as first described in Chapter 1. The first section provides an overview of the processes involved in the use the Box-Fusion approach. The second section establishes definitions and descriptions of basic concepts used through this dissertation. Section three expands on the process of generating the Box-Fusion CFGs (BF-CFG). Section four describes the instrumentation process while section five discusses the BF-CFG population process. The final section discusses the analysis of the BF-CFGs.

3.1 Overview

As described in Chapter 2, pairwise testing is a powerful technique to generate a set of test cases that can be used to test systems with a large number of input combinations. To support code analysis, such as the type provided by white-box testing, the approach uses control-flow graphs (CFGs) to instrument the system. The Box-Fusion approach includes the following processes:

Process 1. Generate Control Flow Graphs (CFGs) for the System Under Test (SUT).

Process 2. Extend the CFG to capture dynamic information about the path of test cases, referred to as a Box-Fusion CFG (BF-CFG).

Process 3. Instrument the SUT based on the BF-CFG for each method to mark each visited edge.

Process 4. Generate a set of pairwise test cases for the SUT and execute the pairwise tests over the instrumented SUT to populate the BF-CFG.

Process 5. Use the BF-CFG to support analysis that can result in the following: determination of the parts of the SUT associated with failed test cases, branch coverage, and regression test identification, i.e., test cases that should be rerun after a modification to the code.

In order to measure the amount of coverage of SUT with respect to a particular set of pairwise tests, a decision needed to be made regarding the coverage criteria to use. Drawing from the Software Considerations in Airborne Systems and Equipment Certification, DO-178C, a standard for developing commercial avionics software systems, defines three levels for structural coverage:

- Level C software must demonstrate statement coverage.
- Level B software must demonstrate statement and branch coverage.
- Level A software must demonstrate modified condition/decision coverage.

The decision to use branch coverage as the coverage measurement in the Box-Fusion approach is inspired by these standards because 1) branch coverage will suffice for Level B and C software and 2) the modification required to cover Level A software only requires modification in the generation on the CFGs to account for the extra nodes and edges for multiple conditions.

Fig. 3.1 depicts a Data Flow Diagram (DFD) for the process described above; the boxes depict sources/consumers of data, the lines show the data flowing from one source to a destination, the ellipses represents a process of data, and the parallel lines signify storage of data.

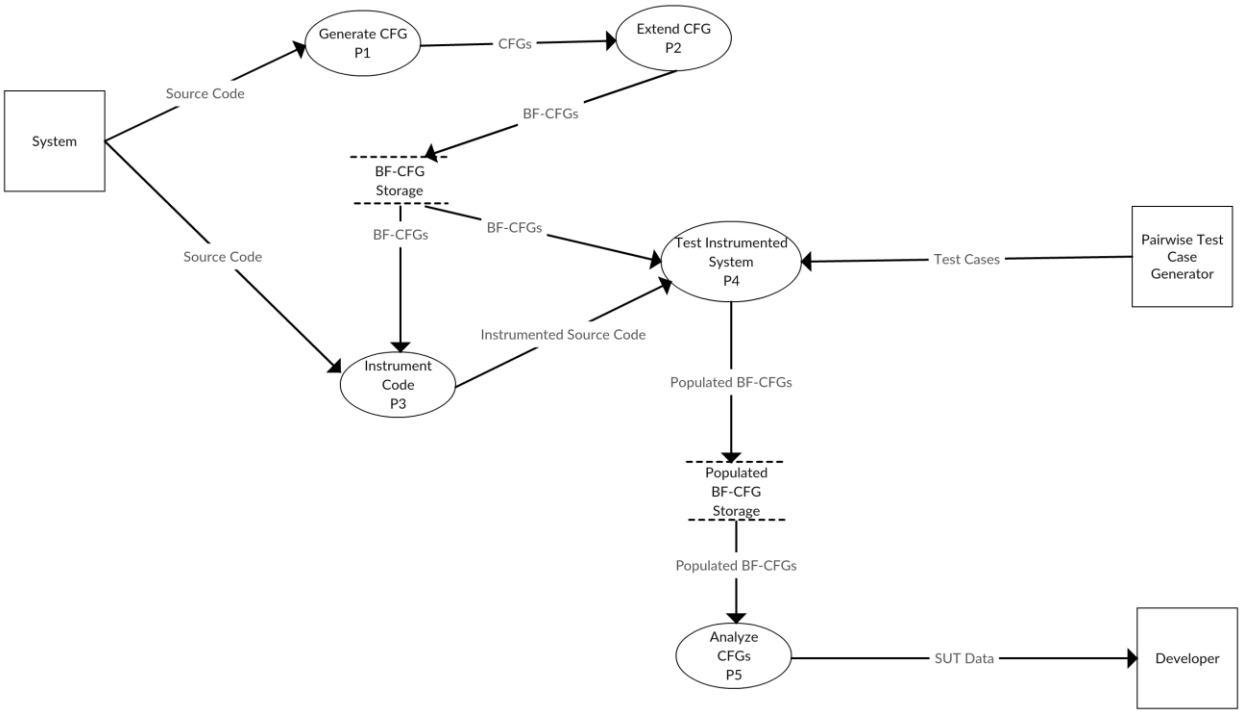


Figure 3. 1 DFD Model for the Box-Fusion Approach.

The DFD describes the movement and transformation of data for the Box-Fusion approach. Processes inside each ellipse represent the process that is transforming the input data into outgoing output data. For example in P2, the process receives as input a CFG coming from P1 and transforms it by creating a BF-CFG, which is outputted into a BF-CFG storage repository. The process depicted in the figure can be automated to take the SUT source code and the pairwise test cases as input and generate the BF-CFG FG. The BF-CFG contains the data of the edges and nodes that tests have been marked as touched stored as XML files, which can be analyzed to understand the behavior of the pairwise tests and the system.

3.2 DEFINITIONS AND DESCRIPTION OF BASIC CONCEPTS

Table 3. 1 Definitions of Terms and Concepts.

Basic block	A series of sequential statements in which the last statement is one that transfers control to another basic block or an EXIT statement; a basic block *B* may be represented by a node ID.
Box-Fusion CFG (BF-CFG)	A CFG that integrates test execution path data into the CFG.
Control-Flow graph (CFG)	Graph comprised of a set of basic blocks and set of directed edges, where a directed edge $B_i \rightarrow B_j$ exists if control flows from basic block B_i to basic block B_j.
CFG location	The pathname that points to the location where the CFG is stored.
Edge	Represents transfer of control from one node to another and is defined as a triple (source node, target node, Boolean).
Edge list	A list containing the edges from a CFG.
Exit node	A node with a special value for the line number of "EXIT." This node is the last node in a CFG, only one Exit node exists in a CFG.
Global instruction counter (Global Counter)	A counter to keep track of the execution path sequence.
Hash map	A data structure that holds all the mark statements to be inserted in the source code.
Line number	A unique number associated with a line of code; line numbers are assigned sequentially to all lines of a source code.
Node	Depending on context, represents a) a line of code, or b) a basic block; defined as a pair consisting of a unique identifier and a line number.
Node identifier (Node ID)	A unique number within a method's CFG assigned to a node.
Mark statement	An instruction statement used to populate a BF-CFG with parameters that include CFG location, source node, and target node.
Marked edge	An edge that includes a collection of Test IDs.
Method	Composed of one or more basic blocks.
Origin tag	A place holder in a BF-CFG that stores the information related to what method called the method associated with the BF-CFG.
Source node	Represents the node at which a transfer of control initiates and is represented by the node identifier, i.e., a positive natural number.
Start node	A node with a special value for the line number of "START." This node is the first node in a CFG, only one Start node exists in a CFG.
Target node	Represents the destination of the transfer of control and is represented by the node identifier, i.e., a positive natural number.
Test identifier (Test ID)	A unique identifier associated with a pairwise test case.
Transfer of control statements	Selection statements, iteration statements, and method calls.

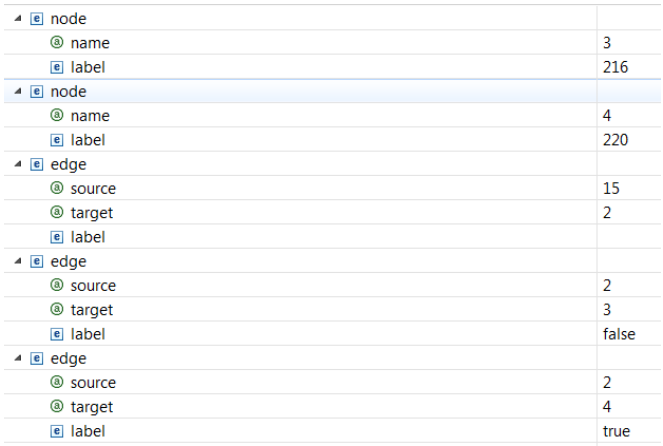
Table 3.1 provides the definitions of terms used in this dissertation within a context of a control-flow graph.

To generate CFGs, an Eclipse plugin, named Control Flow Graph Factory (CFGF) tool [37], was selected. This plugin provides the capability to generate CFGs from source code and bytecode and can export them in multiple formats such as GraphML, GraphXML, and DOT. In the context of this work, the CFG's generated by CFGF tool will be referred to as CFG. The description of the CFGF tool follows:

- The CFGF tool receives a source code file as input.
- The CFGF tool assigns a line number of every line of the source code file.
    - For each class in the source code file, the CFGF tool generates a CFG for each method in that class and represents it in xml format.
- The name of the xml file is as follows: <ClassName>.<MethodSignature>.src.graph.xml, where the names in the angle brackets reflect the name of the class and the signature of the method being represented, respectively.
- To store the xml files, the CFGF tool creates a directory structure based on standard Java package conventions [38].
- The CFGF tool does not recognize a call to another method as a transfer of control statement.
- A node in the CFGF tool represents a statement in a method.
- The CFGF tool assigns a START and EXIT node to signify the entrance and exit points of the CFG.

As shown in Fig. 3.2, the CFGF tool generates a CFG that consists of a header followed by a list of nodes and then a list of edges. In the notation used below, the words and symbols in italics represent constants and the double angle brackets enclose the type of data. Each node in the list is represented as follows in xml:

<*node name*= "<<Node ID>>>" <*label*> <<Line number>> </*label*> </*node*>



node	
name	3
label	216
node	
name	4
label	220
edge	
source	15
target	2
label	
edge	
source	2
target	3
label	false
edge	
source	2
target	4
label	true

Figure 3. 2 CFG Generated by the CFGF Tool.

Each edge in the CFG is represented as follows:

<*edge source*= "<<Source node>>" *target*= "<<Target node>>" >

<*label*> <<Value>> </*label*> </edge>, where *Value* is of type Boolean or blank.

Figure 3.3 illustrates the xml representation for a CFG generated by the CFGF tool.

```
<node name="2">
 <label>214</label>
</node>
<node name="3">
 <label>216</label>
</node>
<node name="4">
 <label>220</label>
</node>
<edge source="15" target="2">
  <label></label>
</edge>
<edge source="2" target="3">
  <label>false</label>
</edge>
<edge source="2" target="4">
  <label>true</label>
</edge>
```

Figure 3. 3 CFGF XML for a CFG.

3.3 PROCESSES 1-2: BOX-FUSION CONTROL FLOW GRAPH GENERATION

A translation algorithm, *CFG2BF* [62], was developed to create BF-CFGs from CFGs. A BF-CFG will store test execution paths that include method calls. As described in the previous section, the CFGF tool generates a CFG for every method within a class, but it does not provide a way to associate method calls. For example, if a method in class A called a method in class B, this transition of control is not captured by the CFG generated by the CFGF tool.

A BF-CFG consists of a list of nodes, a list of edges, and an origin tag. Each of these tags are stored under a parent tag, named <CFG>, and given in xml as follows:

<CFG><Node/><Edge/><Origin/></CFG>

Note that the xml tags like *<Node/>* represent empty tags that are properly enclosed, e.g., *<Node/>* is equivalent to *<Node></Node>*. The representation of the list of nodes given in xml is as follows:

<Node><Name><<Node ID>> *</Name> <Line>* <<Line number>> *</Line></Node>*.

Note that the keyword *label* is replaced the keyword *Line.* The list of edges is represented as follows:

<Edge><Source><<Source node>> *</Source><Target>*<<Target node>>*</Target>*

followed by either:

<Boolean><<Value>> *</Boolean><ID/></Edge>*, if there is a Boolean value for this edge, or

<Boolean/><ID/></Edge>, if there is not.

Note that the *<ID/>* placeholder will be populated during test execution and will consist of 0 or more triple as follows: *(<Val>,* <<Test ID>>@<<Global Counter>>, *</Val>)* as shown in Fig. 3.4. The Global Counter is used to keep track of the execution path sequence.

```
<Edge>
  <Source>15</Source>
  <Target>2</Target>
  <Boolean/>
    <ID>
      <Val>1@359</Val>
      <Val>132@1356</Val>
    </ID>
</Edge>
```

Figure 3. 4 Example of a Populated Edge Node.

The origin tag is given as follows: <*Origin*/>, where the <*Origin*/> placeholder will be populated during test execution and will consist of 0 or more pairs as follows:

(<Val>, <<Class Name>>. <<Method Name>>:<<Test ID>>@<<Global Counter>>, *</Val>)*

where "Class Name" denotes the actual name of the class that called this method and "Method Name" denotes the actual method associated with the call. Fig. 3.5 shows an example origin tag populated with test data.

```
<Origin>
  <Val>Template.getCompositeProposition:1@358</Val>
</Origin>
```

Figure 3. 5 Example of a Populated Origin Node.

Figure 3. 6 Side by Side CFG and BF-CFG XML Files.

The *CFG2BF* algorithm creates a BF-CFG xml file by reading the xml information of a CFG and storing it using the previously mentioned format. Fig. 3.6 shows a snippet of a CFG xml with the corresponding BF-CFG xml.

To verify the correctness of CFG2BF, the author and a team of two students did the following: conducted an inspection of the CFG and BF-CFG xml files generated by the algorithm, and they verified the overall correctness of the CFG by manually inspecting each CFG and comparing it with the corresponding source code. The inspection of the CFGs revealed the problem with the conditional values and the lack of connections between the methods. The edges of the CFG were labeled incorrectly for denoting the condition associated with transfer of control. The correction required a simple flip of the condition, which was included in the *CFG2BF* algorithm. The connections between the methods was addressed with the *Origin* tag. To show that the *CFG2BF* algorithm is correctly implemented, a formal inspection was conducted over the code. Inspections tasks were defined to verify that the code correctly created a BF-CFG from the CFG's generated by the CFGF tool. The following properties about the algorithm were the basis for the creation of the inspection tasks.

- Property 1: The *CFG2BF* algorithm creates a BF-CFG with a parent node with the tag <*CFG*> containing three children nodes with the tags <*Node*>, <*Edge*> and <*Origin*>.
- Property 2: The *CFG2BF* algorithm creates a BF-CFG with nodes for the <*Node*> tag that contains data for <*Name*> and <*Line*> that match the <*node*> tag data of <*name*> and <*label*> from CFG.
- Property 3: The *CFG2BF* algorithm creates a BF-CFG with nodes for the <*Edge*> tag that contains data for <*Source*> and <*Target*> that match the <*edge*> tag data of <*source*> and <*target*> from CFG.
- Property 4: The *CFG2BF* algorithm creates a BF-CFG with two additional nodes for the <*Edge*> tag, <*Boolean*> and <*ID*>. The value for the <*Boolean*> will be the inverse of the value from the <*label*> under the matching <*edge*> from the CFG.
- Property 5: The *CFG2BF* algorithm shortens the names of the BF-CFGs compared to the original CFGs by removing the package prefixes for the datatypes used in parameters and return types.
- Property 6: The *CFG2BF* algorithm creates an archive directory and stores the original CFGs in a directory ending with "-Archive", the BF-CFGs are stored in a directory ending with "–XML."

The formal inspection was done by a faculty member in the Computer Science Department at UTEP. The full list and description of the inspection tasks is listed in Appendix A. The results are provided in Table 3.2, which shows that all the inspection tasks successfully passed. The first column in the table lists the ID of the inspection task, the second column is the Property that it maps to, the third column lists if the task passed/failed the inspection, and the last column describes any comments that the inspector had; in this case, the comments reflect the lines in the code that were critical to show that the property was met, or if it failed, it will list the defect identified.

Table 3. 2 CFG2BF-Mapping of Properties to Inspection Tasks.

Inspection Task ID	Property	Pass/Fail	Comments
1	6	Pass	38
2	6	Pass	44-47
3	2	Pass	74-76
4	1	Pass	78
5	2	Pass	87-88
6	2	Pass	90-92
7	2	Pass	95-99
8	3	Pass	108
9	1	Pass	117-118
10	3	Pass	120-123
11	3	Pass	125-128
12	4	Pass	130-133, 134-137
13	4	Pass	141-142
14	1	Pass	154-155
15	5	Pass	160-190
16	1	Pass	197

3.4 PROCESSES 3: INSTRUMENTATION

As reflected by Process 3 in Fig. 3.1, the *Instrumentation* algorithm (provided in Fig. 3.7) reads a BF-CFG and for every *Edge* triple, it inserts a *mark statement* (as defined below) into the

```
1 Get all BF-CFGs from TeamXMLFolder.
2 Get all Files that are Java source files from TeamFolder.
3 For each BF-CFG in all CFGs:
4       Build an EdgeList by extracting all edges
5       For each JavaFile in all Files:
6               If JavaFile matches CFG
7                       Store name of CFG
8                       For each edge in EdgeList
9                               Store in a HashMap a mark statement containing CFG name,
10                              fromLine, toLine. Hashed by toLine
11                      For each codeLine in JavaFile
12                              Check if codeLine exists in the HashMap
13                              If codeLine exists
14                                      If toLine points to a START
15                                              Insert mark statement to the right of the code.
16                                      Else
17                                              Insert mark statement to left of the code.
18                              Check if name of CFG matches codeLine
19                                      If CFG matches codeline
20                                              Insert saveOrigin statement to the right of the code.
21                              Copy codeLine into NewJavaFile
22              Replace JavaFile with NewJavaFile
```

Figure 3. 7 Instrumentation Algorithm.

original source code at the line number given by *Target* node of the triple.

Furthermore, to capture that transfer of control has switched to another BF-CFG, the algorithm will insert the following Java code, *InstrumentingCode.saveOrigin()* into the beginning every method that has a corresponding BF-CFG. An example code that has been instrumented using the Instrument algorithm is shown in Fig. 3.8.

```
package prospec.generator; import instrumentor.InstrumentingCode;

public class SimpleLTL_Generator extends Generator{

    private String generateUniversalityAfterLuntilR(Property P) {   InstrumentingCode.saveOrigin();InstrumentingCode.mark("C:\\Users\\ochoao\\Desktop\\Diss\\FinalWorkspace\\TeamDriv
InstrumentingCode.mark("C:\\Users\\ochoao\\Desktop\\Diss\\FinalWorkspace\\TeamDriver\\SimpleLTLGenerator-XML\\prospec\\generator\\SimpleLTL_Generator\\","generateUniversalityAfterLu
    }

    private String generateUniversalityAfterL(Property P) {InstrumentingCode.saveOrigin(); InstrumentingCode.mark("C:\\Users\\ochoao\\Desktop\\Diss\\FinalWorkspace\\TeamDriver\\Simp
InstrumentingCode.mark("C:\\Users\\ochoao\\Desktop\\Diss\\FinalWorkspace\\TeamDriver\\SimpleLTLGenerator-XML\\prospec\\generator\\SimpleLTL_Generator\\","generateUniversalityAfterL(
    }
```

Figure 3. 8 Sample Instrumented Code.

To show that the *Instrument* algorithm is correctly implemented, a formal inspection was conducted over the code. Inspection tasks were defined previous to the inspection with the aim to verify that the code correctly instrumented the source code. The following properties were the basis for the creation of the inspection tasks.

- Property 1: The *Instrument* algorithm inserts a saveOrigin instrumentation statement as the first statement in the method matching the BF-CFG being instrumented.
- Property 2: The *Instrument* algorithm inserts the START instrumentation statement at the right side of the line that the edge is pointing from.
- Property 3: The *Instrument* algorithm inserts EXIT instrumentation statements at the right side of the line provided by the toLine value in the edge.
- Property 4: The *Instrument* algorithm inserts every edge that exists in a BF-CFG (minus START and EXIT) into the corresponding Java source file, at the beginning of the line that the edge is pointing to.
- Property 5: The *Instrument* algorithm instruments every Java source file that has a corresponding BF-CFG method.

The formal inspection was done by a faculty member in the Computer Science Department at UTEP. The full list and description of the inspection tasks is listed in Appendix A. The results of the inspection is documented in Table 3.3, following the same convention as the previous formal inspection.

Table 3. 3 Instrumentor Mapping of Properties to Inspection Tasks.

Inspection Task ID	Property	Pass/Fail	Comments
1	5	Pass	383, 384
2	1-4	Pass	430 origin, 456 start, 473 exit, 351 edge
3	2	Pass	315
4	2	Pass	567-570
5	3	Pass	303
6	3	Fail	Defect 1
7	4	Pass	291
8	4	Fail	Defect 2
9	1	Pass	448
10	1	Pass	405
11	5	Pass	454 origin, 363 edges, 471 start, 487 exit

Defect 1

Issue Description: The issue was caused by the order of instrumentation, the exit mark statements were being placed before the normal edge mark statements.

```
for (String file : allFiles)
{
    if (file.contains(javaName))
    {
        File codeFile = instrumentor.getFile(instrumentor.getDirectory(file), instrumentor.getFileName(file));

        instrumentor.createExitInstrumentation(edgeList, codeFile, cfgFile);
        instrumentor.createInstrumentation(edgeList, codeFile, cfgFile);
        instrumentor.createOriginInstrumentation(codeFile, cfgFile);
        instrumentor.createStartingInstrumentation(edgeList, codeFile, cfgFile);
    }
}
```

Figure 3. 9 Corrected Instrumentation Code.

Resolution: The code was corrected by changing the order of the instrumentations to instrument the exit edges first (placing them at the leftmost side), then the normal edges (placing them to the left of the exit mark statement). This corrected the issue, the new code follows below and is shown in Fig. 3.9.

Defect 2

Issue Description: This defect was related to *Defect 1*. The mark statement was being placed to the left side of the line, but not at the leftmost side. The mark exit statement was being placed before the mark statement, so the intent of this inspection task was deemed to be incorrect.

Resolution: The code fix that was done under *Defect 1* corrected this issue.
A review was done after the correction was applied by the author, and no further issues were found.

3.5 PROCESS 4: POPULATING THE BF-CFG

As shown by Process 4, the *PopulateCFG* algorithm is initiated during test execution to insert values into the <*ID*/> and <*Origin*/> placeholders. Recall that the BF-CFG stores marked edges by inserting the values <<Test ID>>@<<Global Counter>> into the <*Edge*> node. To populate the <*Origin*> node, the algorithm inserts the values <<Class Name>>.<<Method Name>>:<<Test ID>>@<<Global Counter>>, where the names associated with *Class Name* and *Method Name* are for the caller method, and the *Test ID* into the callee's BF-CFG. Note that the insertions are triggered by the instrumented source code.

The class *InstrumentingCode* contains two methods to support the marking of edges and capturing the origin data. The *mark* method takes as input the name of the current CFG, the method that is being instrumented, and the lines corresponding to the nodes to be marked by this edge. The *saveOrigin* method takes no inputs as it calculates who the caller method and class is by looking at the runtime environment [39]. To support the creation of execution paths, the order of how the edges were marked must be recorded to ensure this the algorithm keeps a *globalCounter* variable which gets stored along the test ID when the data is saved to an edge. Every time an edge or origin is marked, the global counter is incremented by one. In addition, because different edges might join at one point in the code, disambiguation is needed to know the actual edge that was traversed. To solve this problem a variable called *previousLine* is introduced. This variable holds the previous line that was visited, making it possible to compare, if a line does not match what the *mark* statement is providing, it means: 1) it is a new method and the line is a "START" line, or 2) this point in the code is a join of multiple edges, and the correct edge is the identified using the

previousLine and the *toLine* data. Because transferring control to a different method switches the context of what the previous line was, and the previous line will be needed when returning to control after the method is done, a stack is required to save the *previousLine* value for future use. The *previousLine* value is pushed into the stack when the *saveOrigin* method is invoked (this means a method was called) and the *previousLine* is popped when a *mark* statement is encountered. To document and facilitate the understanding of this part of the algorithm, a Statechart [40] is provided in Fig. 3.10.

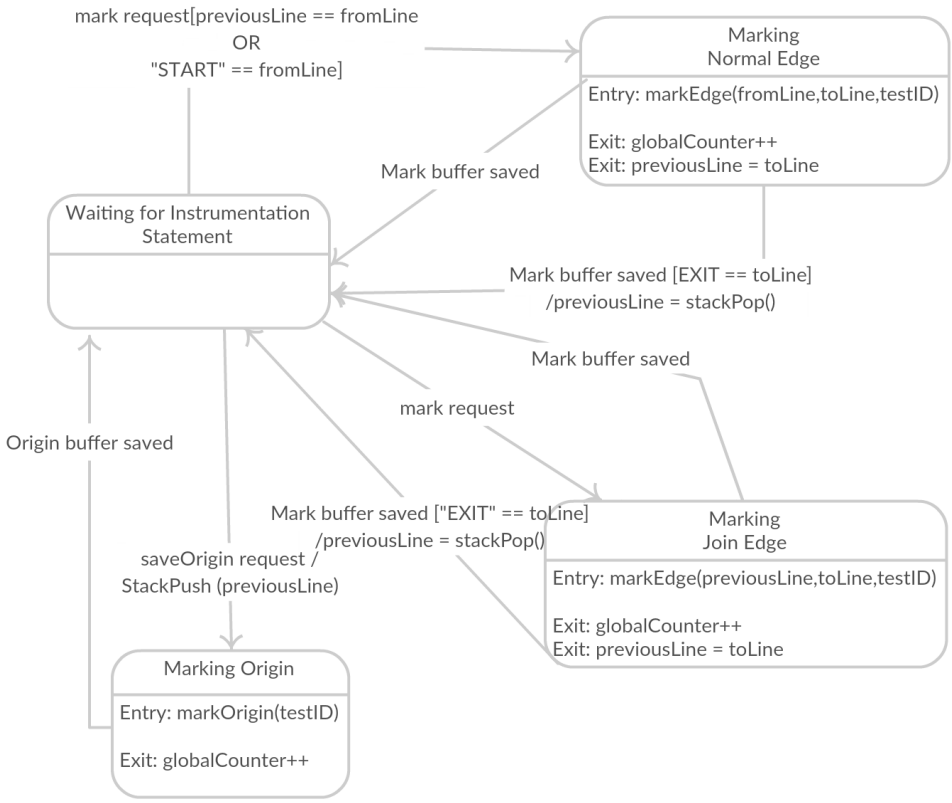


Figure 3. 10 Statechart Snippet of the PopulateCFG Algorithm that Tracks Traversed Edges.

The rectangles with circular corners represent states of the system, while the text associated with arrows represents the event or trigger that causes a transition to another state. The text inside the square brackets ([*condition*]) denote the condition that must be satisfied for the transition to occur. The text following the forward slash (/) represents an action that takes place during the

transition. The "Entry" label denotes what action takes place upon entering a state and the "Exit" label what action takes place upon leaving the state.

Because the *PopulateCFG* algorithm is I/O intensive and in order to mitigate the performance slowness caused by opening and populating many BF-CFG files, the *PopulateCFG* algorithm uses buffers to store data until the end of a test run. At the end of a test run, the *saveFiles()* method is invoked, which will extract all the common data for a BF-CFG from the buffers and populate that BF-CFG with only one file open and close.

To verify that the *PopulateCFG* algorithm was correctly implemented, a formal inspection process was conducted over the code as was done for the *Instrument* algorithm. The following properties about the algorithm were the basis for the creation of the inspection tasks.

- Property 1: Every time that an instrumentation is triggered, the *globalCounter* is incremented; everytime a *mark* statement is triggered the *previousLine* is updated with the *toLine* value.
- Property 2: Whenever there is a transfer of control, i.e., the *saveOrigin* method was called, the *previousLine* is pushed into the stack and the origin data is stored.
- Property 3: When the *PopulateCFG* encounters an "EXIT" edge, the stack will be popped and the value stored in the *previousLine*.
- Property 4: When the *PopulateCFG* encounters a "START" edge or the *fromLine* matches the *previousLine*, the instrumentation mark data will be stored.
- Property 5: When the *PopulateCFG* encounters an edge for which the *fromLine* does not match the *previousLine*, the instrumentation mark data will be stored, replacing the *fromLine* with the *previousLine*.
- Property 6: The *saveOrigin* method stores the origin data in the origin buffer.
- Property 7: The *saveToFiles* populates all the BF-CFGs in the correct format and resets the state of the *PopulateCFG*.

This formal inspection was also done by the same faculty member in the Computer Science Department at UTEP. The full list and description of the inspection tasks are provided in Appendix A. The results of the inspection is documented in Table 3.4, following the same convention as previously described.

Table 3. 4 PopulateCFG Mapping of Properties to Inspection Tasks.

Inspection Task ID	Property	Pass/Fail	Comments
12	3,4	Fail	Defect 1
13	1	Pass	234
14	5	Pass	231
15	4	Pass	231
16	3	Pass	236, 237
17	1	Pass	240
18	6	Pass	245
19	1	Pass	247
20	2	Pass	316
21	7	Pass	178 store, 181 remove, 172 to stop
22	7	Pass	198 store, 202 remove, 189 to stop
23	7	Pass	168 mark, 185 origin, 218 counter
24	7	Pass	118
25	7	Pass	126-128
26	7	Pass	137
27	7	Pass	298
28	7	Pass	198

Defect 1

Issue Description: Defect 1 was identified because the Inspection Task ID 1 asks "Does the *mark* method store the mark statement data (i.e., *fileLocation, fileName, fromLine, toLine, globalCounter*) into the mark buffer?" The task failed because the *mark* method uses the *previousLocation* if the *fromLine* does not match it.

Resolution: The issue lies in the wording of the Inspection Task. The *PopulateCFG* implementation is correct.

3.6 BF-CFG ANALYSIS

This section describes the types of analysis that are afforded with a BF-CFG. These include: identification of verification gaps, coverage analysis, fault guidance, and regression test selection.

3.6.1 Identification of Verification Gaps

A coverage measurement percentage is based on the number of marked edges relative to the total number of edges. Knowing the percent of coverage that the test cases have achieved in the SUT gives a general measurement of how much of the system has been tested, but it can also add the feature of facilitating the identification of what parts of the system were not tested, i.e. verification gaps. This can be achieved by analyzing the BF-CFGs to determine the missing coverage in testing, i.e., identifying the edges that have not been marked at least once. Those paths that were not touched, i.e., the edges unmarked by the testing, represent the parts of the SUT that have a gap in verification.

Fig 3.11 shows an example preliminary BF-CFG with unmarked edges (in red) between nodes N4, N6 and N6, N7. For this example BF-CFG, in order to achieve full branch coverage, the edges connecting nodes N4 to N6, and N6 to N7 are identified as requiring additional

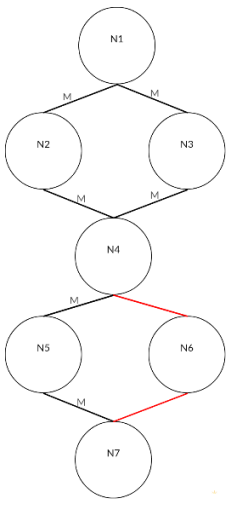


Figure 3. 11 Example BF-CFG with Marked/Unmarked Edges.

verification. Once the edges are identified, additional verification techniques can be applied, in particular inspection or walkthrough since identifying test cases may be difficult because of the analysis needed to determine how to reach that part of the SUT. While a white-box testing approach could be used instead of the Box-Fusion approach, the white-box approach may be infeasible in cases where there are a large number of input combinations.

3.6.2 Detecting Potential Defect Location

An additional feature of the Box-Fusion approach is the ability to identify a relationship between test cases and the covered code. For example, Fig. 3.12 shows a small program (containing a defect) to determine what letter maps to a numeric average. The nodes are labeled with the corresponding lines of code.

```
10          if(average > 90.0f)
11              finalGrade = "A";
12          else if (average >= 80.0f)
13              finalGrade = "B";
14          else if (average >= 70.0f)
15              finalGrade = "C";
16          else if (average >= 60.0f)
17              finalGrade = "D";
18          else finalGrade = "F";
19          System.out.println("The final grade is " + finalGrade);
```

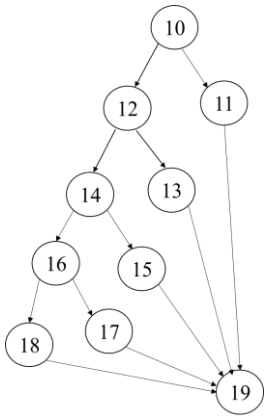


Figure 3. 12 Example Code and CFG for Letter Grade Calculation.

Consider the following simple tests:

- *test ID*=1 {average=90, expected finalGrade = A}
- *test ID*=2 {average=80, expected finalGrade = B}
- *test ID*=3 {average=70, expected finalGrade = C}
- *test ID*=4 {average=60, expected finalGrade = D}
- *test ID*=5 {average=50, expected finalGrade = F}

For a BF-CFG, each test case is uniquely associated with a corresponding set of marked edges identified by a unique number, i.e., the *test ID*, as shown in Fig 3.13. The numbers by the edges represent the *test ID* of the test that marked that edge. Inspecting the code shows that *test IDs* 2, 3, 4, and 5 passed while *test ID* 1 fails (calculates grade of *F*). In the example above, *test ID* 1 is associated with nodes 19, 13, 12, and 10. These nodes are the potential candidates for the location of the defect (the defect is located in line 10, where the operator should be >=). The test cases that failed can be traced back to the edges that were marked by the executed tests.

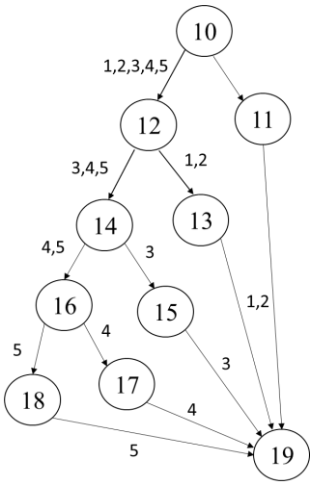


Figure 3. 13 Test ID Marked on a BF-CFG for Letter Grade Calculation.

Under a white-box testing approach, the goal is to generate the least amount of test cases while providing the coverage with respect to the criterion. When a test fails, every node in that path must be debugged to detect where the defect is located. However, because the Box-Fusion

approach uses pairwise testing, the number of tests will be relative to the number of pairs of input values, which provides an opportunity to draw conclusions about the shared parts of the system between tests. That is, if a majority of tests pass through a common place in the system, and that majority of tests pass, then it is probably the case that there are no defects within that shared place in the system.

3.6.3 Test Case Selection for Regression Testing

A consequence of the Box-Fusion approach is that each test in the set of pairwise test cases will result in an execution path within the BF-CFG. If a change is introduced in the source code, the corresponding nodes in the CFG can be used to determine which tests execute over a particular section of the code. This can be useful because the *test IDs* connected to the modified nodes represent the tests, and only those tests, that must be executed to fulfill optimal regression testing.

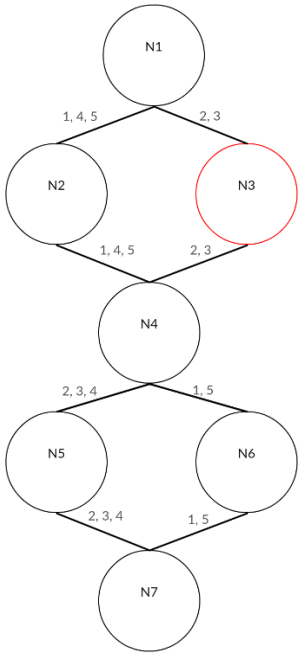


Figure 3. 14 Example CFG with Modified Node.

For example, as shown in Fig 3.14, if a change is introduced into node N3, the regression testing can be simply determined by looking at the associated tests that pass through node N3. For this

example, *test IDs* 2 and 3 would need to be re-executed. In addition, changes also need to be tracked on those *test IDs* that have called this BF-CFG.

Chapter 4 Case Study

This chapter presents a case study that was conducted to analyze the Box-Fusion approach and evaluate tools that can support the analysis. Because of the emphasis on pairwise testing, it was critical to select an algorithm that includes a large set of input combinations. The benefit of conducting a case study is that a deeper understanding of the capability of the subject of the investigation, that is the Box-Fusion approach, can be gained. The case study is driven by research questions and provides a systematic approach to gather data.

This chapter is divided as follows: the first section provides a background of the selected algorithm for the case study: the Prospec algorithm, which generates formal specifications in LTL from classifications of patterns, scope, and composite propositions; the second section discusses the research questions and propositions; the third section discusses the unit of analysis; the fourth section describes the logic that links the propositions to the data; and the fifth section describes the criteria for interpreting the findings; and the last section describes the algorithm implementations.

4.1 Background

The Prospec algorithm was selected for the case study due to the complexity stemming from its numerous requirements (specified using first order logic). The choice of such system was intentional for the following two reasons: 1) the algorithm is complex and challenging to verify because of the domain knowledge required to generate formal specifications in LTL, making it a non-trivial real-world example for evaluating the Box-Fusion approach; and 2) the implemented system can generate a large number of formulas from a large range of inputs, which lends itself to the application of the pairwise testing technique.

4.1.1 Linear Temporal Logic Background

The use of formal verification techniques, such as the ones mentioned in Chapter 2, has shown promise in discovering subtleties that can be missed by non-formal techniques such as testing. Most verification techniques use formal languages to specify system properties. Linear Temporal Logic (LTL) is a prominent formal specification language that is highly expressive and

is widely used in formal verification tools such as the model checkers SPIN [26], NUSMV [22] and the Java Path-Finder [28], and runtime verification of Java programs [29].

4.1.2 LTL Formulas

Formulas in LTL are constructed from elementary propositions and the usual Boolean operators for *not*, *and*, *or*, *imply* ($\neg$, $\wedge$, $\vee$, $\rightarrow$, respectively). In addition, LTL allows for the use of temporal operators *next* (X), *eventually* ($\Diamond$), *always* ($\Box$), *until* (U), *weak until* (W), and *release* (R). In this work the operators W and R are not used. A temporal formula is constructed inductively from a set of propositions P by applying Boolean connectives $\neg$ and $\vee$ and temporal operators next (X) and until (U) as follows:

- A proposition is a temporal formula.
- If p and q are temporal formulas then so are the following:
  - $\neg p$
  - $p \vee q$
  - $p \wedge q$

These formulas assume discrete time, i.e., state s may be denoted as *0*, or *1*, or *2*, or … *n*. The meaning of the temporal operators is straightforward [cite]:

- The formula Xp holds at state s if p holds at the next state $s + 1$.
- The formula $\Diamond p$ holds at state s if p is *true* at some state $s' \geq s$.
- The formula $\Box p$ holds at state s if p is *true* at all states $s' \geq s$.
- The formula pUq holds at state s, if there is a state $s' \geq s$ at which q is *true* and, if s' is such a state, then p is *true* at all states s_i for which $s \leq s_i < s'$.

Since LTL formulas assume a discrete time, then an LTL formula is satisfied in a specific state if the behavior specified by the formula is specified in every possible future path in the Kripke structure. A more detailed description can be found in [30].

4.1.3 SPS Algorithm

To assist users in the generation of formal specifications, the Specification Pattern System (SPS) was developed [31, 32]. The work defined a set of patterns to represent the most commonly used software properties and to guide the practitioner through generating formal specifications of these properties. Patterns capture the expertise of developers by describing behaviors of recurrent properties. SPS also defined a set of scopes of system execution where the pattern of interest must hold. Each pattern and scope combination can be mapped to specifications in LTL and Computational Tree Logic (CTL). The main patterns defined by SPS are: *Universality*, *Absence*, *Existence*, *Precedence*, and *Response*. Their descriptions can be viewed in the SPS website [32]. SPS defines, five types of scopes: *Global*, *Before R*, *After L*, *Between L* and *R* and *After L until R*. SPS is presented as a website [32] with links to descriptions of the patterns. The website provides a mapping of each pattern and scope combination into different formal specification languages.

Composite propositions (CP) [33] expand the expressiveness of patterns and scopes to include the specification of sequential and concurrent behaviors. In practical applications, there is often a need to describe properties where one or more of the pattern or scope parameters are made of multiple (i.e., composite) propositions. To describe such properties, SPS was extended by introducing a classification for defining sequential and concurrent behavior to describe pattern and scope parameters. Specifically, the work [33] defined the following CP classes: $AtLeastOne_C$, $AtLeastOne_E$, $Parallel_C$, $Parallel_E$, $Consecutive_C$, $Consecutive_E$, $Eventually_C$, and $Eventually_E$. The subscripts $_C$ and $_E$ describe whether the propositions within a CP class are asserted as conditions or events, respectively. A proposition defined as a condition holds in one or more consecutive states. A proposition defined as event means that there is an instant at which the proposition changes value in two consecutive states.

4.1.4 Prospec Algorithm

Previous work [34] defined a set of templates that can be used to ease the generation of LTL specifications for all pattern, scope, and CP combinations. This work resulted in an algorithm, referred to in this work as the Prospec algorithm, which generates the LTL formulas

Table 4. 1 Composite Propositions.

#	Composite Proposition
1	$AtLeastOne_C$
2	$AtLeastOne_E$
3	$Parallel_C$
4	$Parallel_C$
5	$Consecutive_C$
6	$Consecutive_E$
7	$Eventually_C$
8	$Eventually_E$

from the set of templates. The Prospec algorithm defined nine templates to generate formulas within the *Global* scope, fourteen for formulas within the *Before R* scope, and another five templates to generate the formulas within the *Between L and R* and the *After L until R* scopes. The templates used by the algorithm generate one property by selecting one out of five possible pattern types, one out of five possible scopes, and each pattern and scope can be described by zero to two CPs, where each CP can be one out of eight possible types, as shown in Table 4.1. Thus, more than 31,000 properties can be generated from these combinations (refer to Table 4.2). Formal proofs and software inspections were used to validate the correctness of the LTL templates.

Table 4. 2 Breakdown of Property Types.

Pattern/ Scope	*Global*	*Before R*	*After L*	*Between L and R*	*After L Until R*	**Total**
Absence P	8	64	64	512	512	1160
Existence P	8	64	64	512	512	1160
UniversalityP	8	64	64	512	512	1160
Q Precedence P	64	512	512	4096	4096	9280
Q Strict Precedence P	64	512	512	4096	4096	9280
Q Response P	64	512	512	4096	4096	9280
Total	216	1728	1728	13824	13824	31320

4.2 RESEARCH QUESTIONS AND PROPOSITIONS

The research questions that are driving the research are as follows:

RQ 1. How can pairwise testing be augmented to direct the developer to potential areas of faulty code, untested code, and related test failures?

RQ 2. How can pairwise test cases be identified to achieve the appropriate coverage required by regression testing?

To address these questions, the research developed a Box-Fusion approach that enhances pairwise testing with a BF-CFG that drives the instrumentation of a system-under-test and captures the execution path of the pairwise tests. Driven by RQ1, the case study will examine the capabilities of the approach in finding faults, areas of the code that have not been tested, and determine the techniques that are needed to assist in the analysis of the of test failures. Determining whether the Box-Fusion approach addresses RQ2 is driven by analyzing the impact of change to the code.

A proposition in a case study is a claim about the research questions that provides a way to direct the evidence needed to support the claim. The propositions are as follows:

P1. The Box-Fusion approach can determine the areas of the code that require additional verification.

P2. The code defects, which are identified through one or more pairwise tests, are located in the execution path(s) identified by the Box-Fusion approach.

P3. Analyzing the BF-CFG using a suite of pairwise tests can narrow the potential sections of code where a defect is located.

P4. Regression tests can be optimized by matching the modified nodes of a BF-CFG to the tests that pass through the node.

Note that P3 is an exploratory effort that will include identification of common paths, success in classifying failed tests, and reduction in the execution path, i.e., the length of the execution path.

4.3 UNITS OF ANALYSIS

Units of analysis define what the "case" is in the case study [35]. In the context of this study, there are two main systems that are studied. The first system is an implementation of a subset of the Prospec algorithm that only uses atomic propositions, i.e., those that do not consider composite propositions. The second system is an implementation of the Prospec algorithm, which includes composite propositions. The Box-Fusion approach will be used on each of the implementations, which will yield a BF-CFG and instrumented source code for of the algorithms.

The task of generating the pairwise test cases to verify the possible properties generated by the Prospec algorithm implementations required the analysis of the input to the algorithm because of difference in parameters. The Prospec algorithm takes as input parameters in terms of patterns, scopes as shown in Table 4.2. The values represent the number of possible composite combinations, for example for *Global* column with *Absence* row, represents that the *Global* scope does not require any parameters, whereas the *Absence* patterns requires one parameter, *P*. Recall that parameter *P* can be any of the 8 possible combinations of composite propositions as shown in Table 4.1. Something more complicated, such as the pattern *Precedence P, Q* with the scope *Between L and R* requires 4 different composite propositions (8^4) for a total of 4096 possible combinations.

Instead of testing all 31,320 possible combinations, a pairwise approach will reduce the number of combinations by testing only pairs of parameters. To facilitate the generation of this reduction, the online tool Pairwiser [36] was used. Pairwiser has a web-based interface that can be used to enter parameter definitions, generate tests and export test inputs as excel files.

The first required step is to enter the testing parameters. This was done by entering the six parameter names: *Pattern*, *Scope*, *P*, *Q*, *L*, and *R*. Then for each of these parameters, the possible values were entered, as shown in Fig. 4.1 below.

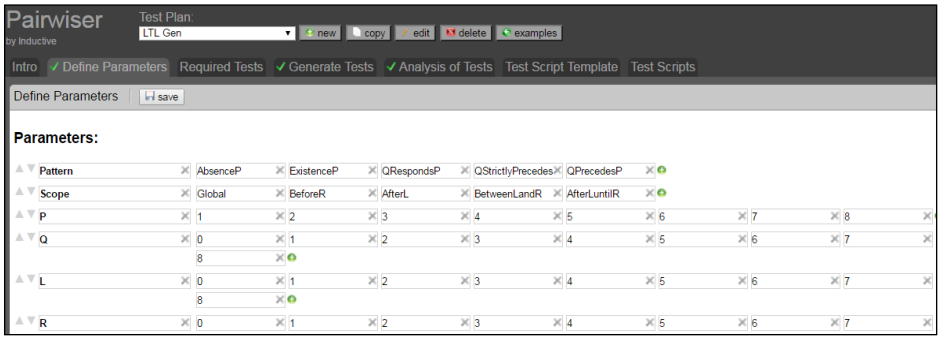


Figure 4. 1 Pairwiser Parameter Inputs

Since not all patterns and scopes require *P*, *Q*, *L* or *R*, a value of 0 was used to constrain the generation of combinations to only generate valid combinations. The value 0 represents an empty parameter; for example for the *Absence* pattern, when *Q* is set to the value of 0, it signifies that Q is not a required parameter. The following were the constraints added and Fig. 4.2 shows the actual constraints entered into the Pairwiser tool:

- The patterns *Absence* and *Existence* were constrained to always have a value of 0 for *Q*.
- The patterns *Responds*, *StrictlyPrecedes*, and *Precedes* were constrained to not allow a value of 0 for *Q*.
- The scope *Global* was constrained to always have a value of 0 for both *L* and *R*.
- The scope *After L* was constrained to always have a value of 0 for *R* and to not allow a value of 0 for *L*.

- The scope *Before R* was constrained to always have a value of 0 for *L* and to not allow a value of 0 for *R*.

122	QRespondsP	Global	1	6	0	0
123	QStrictlyPrecedesP	AfterLuntilR	5	3	5	1
124	AbsenceP	Global	2	0	0	0
125	QRespondsP	Global	5	2	0	0
126	QRespondsP	AfterL	4	8	6	0
127	QRespondsP	AfterL	2	4	8	0
128	ExistenceP	BetweenLandR	5	0	1	5
129	QRespondsP	Global	6	5	0	0
130	QStrictlyPrecedesP	BeforeR	5	4	0	1
131	QStrictlyPrecedesP	AfterLuntilR	4	7	5	3
132	QStrictlyPrecedesP	AfterLuntilR	3	2	4	7

Figure 4. 3 Sample of the Generated Test Cases.

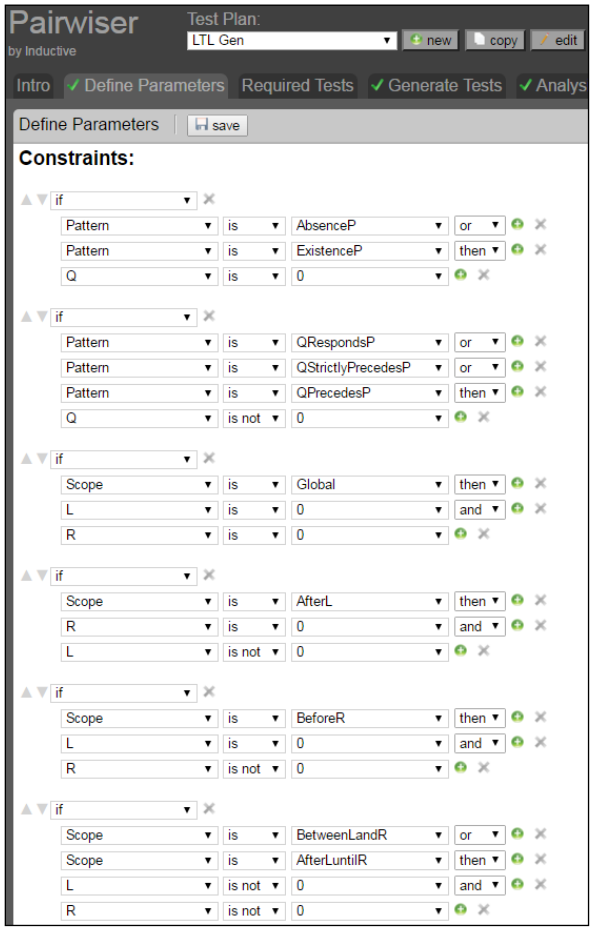


Figure 4. 2 Constraints Used in Pairwiser.

- The scopes *BetweenLandR* and *AfterLuntilR* were constrained to not allow a value of 0 for both *L* and *R*.

Requesting the generation of the tests resulted in 132 tests cases that were exported into an *excel* file. Note that column 1 in Fig. 4.3 presents the Test ID. The last four columns provide the CP combinations. A sample of the lower portion of the tests is shown in Fig. 4.3 below. The full set of tests appear on Appendix A.

The Pairwise *excel* file containing all of the 132 generated test cases was verified to confirm that the set of test cases is complete, i.e., by ensuring that every combination of pairs of Pattern, Scope, and CP Type was represented at least once. For example, Pattern *Absence of P* was checked to be paired with every type of Scope (5 possible pairs). Another example was to check that every CP Type was tested in every combination of Pattern and Scope, i.e., it could include P, Q, L and R, with the exception of *Global*. Additionally, each pair of CP combination was checked in at least one test. No missing test cases were found. Similarly, for the first Prospec implementation

	Pattern	Scope	P	Q	L	R
1	UniversalityP	AfterLuntilR	1	0	1	1
2	QStrictlyPrecedesP	Global	1	1	0	0
3	QPrecedesP	BeforeR	1	1	0	1
4	QPrecedesP	Global	1	1	0	0
5	ExistenceP	BetweenLandR	1	0	1	1
6	UniversalityP	BeforeR	1	0	0	1
7	QRespondsP	AfterL	1	1	1	0
8	QRespondsP	BeforeR	1	1	0	1
9	ExistenceP	BeforeR	1	0	0	1
10	QStrictlyPrecedesP	BeforeR	1	1	0	1
11	QPrecedesP	AfterLuntilR	1	1	1	1
12	UniversalityP	Global	1	0	0	0
13	AbsenceP	AfterL	1	0	1	0
14	QPrecedesP	AfterL	1	1	1	1
15	ExistenceP	Global	1	0	0	0
16	QRespondsP	BetweenLandR	1	1	1	1
17	AbsenceP	AfterLuntilR	1	0	1	1
18	ExistenceP	AfterLuntilR	1	0	1	1
19	AbsenceP	Global	1	0	0	0
20	QStrictlyPrecedesP	BetweenLandR	1	1	1	1
21	QRespondsP	AfterLuntilR	1	1	1	1
22	UniversalityP	BetweenLandR	1	0	1	1
23	QPrecedesP	BetweenLandR	1	1	1	1
24	QStrictlyPrecedesP	AfterL	1	1	1	1
25	QStrictlyPrecedesP	AfterLuntilR	1	1	1	1
26	UniversalityP	AfterL	1	0	1	0
27	QRespondsP	Global	1	1	0	0
28	AbsenceP	BeforeR	1	0	1	1
29	ExistenceP	AfterL	1	0	1	1
30	AbsenceP	BetweenLandR	1	0	1	1

Figure 4. 4 Atomic Prospec Pairwise Tests.

that only uses atomic propositions, the pairwise test generation resulted in a total of 30 tests. The results are shown in Fig. 4.4.

The results of the pairwise testing are stored in GitHub [62]. For each unit of analysis, a set of data was collected as shown below in Table 4.3.

Table 4. 3 Artifacts Used to Collect Data.

Artifact	Data Collected
BF-CFG	Execution path Coverage achieved Uncovered edges found
Implemented System	Defect location Change location Version of the code
Pairwise Test Suite	Passed/failed tests Tests identified for regression testing
Participant	Time spent finding defect

From the BF-CFG, the execution path is identified for each test case, which is used to calculate a coverage percentage for each implementation. From the BF-CFG, those edges that were not covered are marked as uncovered edges and related parts of the system identified. For each system, the location of found defects and changes in code were documented. When modifications to the system is done, a new version was saved and tested. The tests that pass and fail were identified from the pairwise test suite. Tests that were identified for regression testing were recorded. The participants of the study recorded the time spent finding defects following the Box-Fusion approach.

4.4 LOGIC LINKING

Logic linking connects the data to propositions [35]. The data that the study collected from the unit of analysis identified in the previous section, was analyzed for each proposition as Table 4.4 shows. Note that P3 is an exploratory effort that will include identification of common paths, success in classifying failed tests, and reduction in the execution path, i.e., the length of the execution path.

Table 4. 4 Data Collected for Each Proposition

Proposition	Data Collected
P1	Coverage achieved Edges not traversed
P2	Execution path Defect location Time spent Passed/failed tests Version of the code
P3	Execution paths Passed/failed tests
P4	Version of the code Change location Tests identified to execute

The propositions were evaluated by using the data gathered from the analysis of the study as follows:

- For Proposition 1: The BF-CFG is analyzed to identify edges and nodes that have been touched by any test from the pairwise test suite. This will be used to calculate the percentage of branch coverage for each software system. To detect the parts of the system that are not covered by the testing, the edges that have no Test ID assigned are identified. The resulting edges and nodes that are identified as not covered are then mapped to the lines of code in the SUT. These nodes, edges, and lines of code are collected along with the test suite.

- For Proposition 2, two approaches are used: 1) Implementation 1 (Simple LTL Generator), which has been verified through pairwise testing, will be seeded with defects to ensure that the execution paths of failed tests are returned by the Box-Fusion approach, and 2) Implementation 2 (Prospec Algorithm), which has a known defect for the *Response-AfterLuntilR* formulas, will be used to check whether the BF-CFG assists in locating the defect(s). The participant will strictly follow the identified execution path to determine the location of the defect within the system. The time required for two participants to locate the defect will be recorded for future work to assist in evaluating the approach's efficiency in discovering the location of a defect.
- For Proposition 3, the study will analyze the execution paths for the failed tests from the pairwise testing suite. The goal is to compare variations in the algorithms that retrieve intersections of nodes within a BF-CFG to determine the best approach for reducing the number of BF-CFG nodes where defects may be located.
- For Proposition 4, lines of code will be modified resulting in a new version, denoted as Version 2. The BF-CFG nodes, which represent methods in which the modified lines occur, will be examined to determine what *test IDs from Test Suite A* (a set of test cases generated by the Pairwiser tool) have touched those nodes. This set of tests, which we call Test Suite B, is a subset of Test Suite A, and is the set of test to be executed during regression testing. The data to be collected are the results of running 1) Test Suite A on the original code, 2) Test Suite B on Version 2 of the code, and 3) Test Suite A \ Test Suite B on the both the original code and Version 2.

4.5 CRITERIA FOR INTERPRETING FINDINGS

The following was the criteria used to interpret the findings from the gathered data:

1. Proposition 1: The correctness of the algorithms that instrument and create the BF-CGFs, was accomplished using formal inspections as described in the previous chapter. In addition, the correctness of the algorithms to compute the coverage data and verification

gaps, were also accomplished using formal inspections. The data collected from the algorithms were analyzed to provide support for Proposition 1.

2. Proposition 2: The seeded defects in the SimpleLTL Generator, are identified through one or more pairwise tests and were checked to determine if they were in the execution path(s) identified by the Box-Fusion approach. For the Prospec Algorithm implementation the criterion is whether the participants can locate a defect by examining the execution path provided by the Box-Fusion approach.
3. Proposition 3: The criteria for interpreting the results was based on the effectiveness of the Box-Fusion algorithms in narrowing the potential sections of code where a defect is located. The results of the algorithms were compared.
4. Proposition 4: To evaluate the results of Test Suite A \ Test Suite B (as defined earlier) on the original version of the code and on the modified version of the code, will be compared with Version 2 to ensure that the results were identical.

4.6 THE PROSPEC ALGORITHM IMPLEMENTATIONS

As mentioned before, two different implementations of the Prospec algorithm were used by this case study. The first system constructed only implements a subset of the Prospec algorithm that only takes as input propositions of the type atomic. The second system constructed implements the whole algorithm, including taking input as composite propositions. The following sections describe the implementation details.

4.6.1 Simple LTL Generator Implementation

The Simple LTL Generator implementation was completed in 2010 by the author. The resulting implementation size is described by Table 4.5 below:

Table 4. 5 Simple LTL System Size Metrics

Number of Classes	Number of Methods	Total Lines of Code
28	113	1146

The benefit of using this system in the case study is that the pairwise testing for this system is the exhaustive combination of all possible inputs. This allows the analysis the propositions of the case study for the Box-Fusion approach in a small controlled system.

4.6.2 Prospec Algorithm Implementation

The construction of the Prospec algorithm was conducted by teams of students in the 2015 Software Construction class at the University of Texas at El Paso. The class consisted of 30 (12 undergrads, 18 grads) students and was divided into five six-person teams of both novice and experienced developers. Every team contained at least one student who had been exposed to the algorithm in previous classes. The students were given an article describing the algorithm [34], and they spent six weeks to develop a working implementation in Java. The decision of the programming language stems from the language being a pre-requisite to the course, thus ensuring that all the students had the technical knowledge to implement the system.

Each team delivered a system implementing the Prospec algorithm. As shown in Table 4.6 below teams had a certain level of code structure variance as seen by the different numbers of classes, methods and total lines of code. To facilitate instrumentation and CFG generation, two constraints were placed on the implementation, the else-if statement was not allowed to be used and basic blocks of one line had to be enclosed in scoping brackets.

Table 4. 6 Implemented System Size Metrics

Team #	Number of Classes	Number of Methods	Total Lines of Code
Team 1	53	135	3468
Team 2	72	109	3274
Team 3	84	237	4024
Team 4	9	47	1499
Team 5	36	122	1834

Each of the implementations was reviewed and evaluated to select the system for this case study. The criteria for selection was 1) completeness of the implementation, 2) design of the system, 3) adherence to coding standards and good programming practices. The system that best met this criteria was Team 2, which is the system that is used in this case study.

Chapter 5: Results and Observations

This chapter presents the results and observations of a systematic effort in analyzing the Box-Fusion approach. The analysis presented in this chapter is guided by the propositions established for the case study in Chapter 4. As such, the chapter is organized by each proposition. Section 1 concentrates on Proposition 1, the identification of areas that require additional verification in a system. Section 2 focuses on discussing the analysis of Proposition 2, which aims to show that defects are located within the execution path identified by the Box-Fusion approach. Section 3 explores Proposition 3, discussing how the Box-Fusion approach can be used to narrow down the potential location of defects. Section 4 describes the analysis on Proposition 4, i.e., how the Box-Fusion approach can be used to optimize regression testing. Throughout this chapter two implementations are discussed, the Simple LTL Generator, which generates LTL formulas using only atomic propositions and the Prospec Algorithm which generates formulas using composite propositions.

5.1 Proposition 1 Analysis: The Box-Fusion Approach Determines Gaps in Testing

This section discusses the results of measuring branch coverage criteria to the pairwise testing suite on the SUTs. The intent of Proposition 1 is to establish that the Box-Fusion approach can be used to determine what parts of the SUT have been tested (i.e., coverage measurement) and what parts have not been tested (i.e., gap identification) by a specific pairwise testing test suite.

5.1.1 Measuring SUT Coverage

As mentioned in Chapter 3, one of the capabilities that the Box-Fusion approach can provide is the measurement of coverage. The *CoverageMiner* tool was developed to gather the measurement data for each implementation. The tool accesses each BF-CFG, parses each edge and determines the percentage of edges that were marked over edges that were not marked. For example if a CFG has 10 edges, and only 7 have been marked, the BF-CFG is deemed to have 70% coverage. The tool then calculates the total number of edges marked over the total number

of edges for all the BF-CFGs in an implementation. The tool stores the results for each CFG and all the combined CFGs file within the location of the BF-CFGs. An example snippet of the output file is shown in Fig 5.1.

In order to trust the results generated by the *CoverageMiner* tool, a formal inspection was

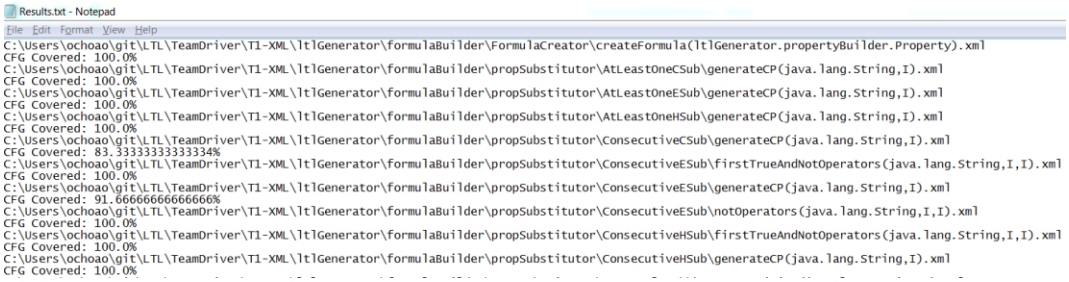


Figure 5. 1 Sample Results File.

conducted over the code. Inspections tasks were defined prior to the inspection, aiming to verify that the code correctly identified all edges, identified visited edges, and correctly calculated the percentage. The following properties about the tool were the basis for the creation of the inspection tasks.

- Property 1: The *CoverageMiner* tool inspects every BF-CFG for a given implementation.
- Property 2: The *CoverageMiner* tool measures the branch coverage for a given system by dividing all the tested edges over the total number of edges.
- Property 3: The *CoverageMiner* tool stores the results in a file.

The formal inspection was done by a faculty member in the Computer Science Department at UTEP. The full list and description of the inspection tasks is listed in Appendix A. The results of the inspection is documented in Table 5.1, the first column represents the task ID that maps Table N with the corresponding checklist table in Appendix A, the second column maps the Property to the inspection task, the third column documents if the inspection task passed or failed and the last column documents the lines that were found to satisfy the inspection task. The results of the inspection found no defects.

Table 5. 1 CoverageMiner Mapping of Properties to Inspection Tasks.

Inspection Task ID	Property	Pass/Fail	Comments
1	1	Pass	30-34
2	2	Pass	57
3	2	Pass	73
4	2	Pass	57, 66
5	2	Pass	73-75
6	2	Pass	57, 66 , 73-75, 80-82
7	3	Pass	90-93

5.1.2 SimpleLTL Generator Coverage Analysis

Applying the tool to the SimpleLTL Generator measured the coverage as shown in Table 5.2. The table shows the system measured, test suite used, number of edges touched by a test, total edges counted and the resulting branch coverage achieved for every BF-CFG generated and instrumented into the system. The results are particularly surprising because the pairwise test suite

Table 5. 2 Branch Coverage for the Simple LTL Generator.

Simple LTL Generator	Test Suite	Edges touched	Total edges	Branch Coverage
Total Covered	30 Tests	307	598	51.33 %

was also the exhaustive set of all possible combinations, not including erroneous combinations. How can an exhaustive test suite only cover 51% of the SUT?

Inspecting the code yields the reason of why this percentage is so low. The original code relied on a Model subsystem that includes classes for composite propositions, which are not actually used in the algorithm. Because the system was not pruned of those classes, BF-CFGs were generated for them and became part of the total edges counted, thus lowering the branch coverage percentage. In addition, the exhaustive test suite does not account for invalid

combinations; thus there was code that was related to error handling that did not include invalid combinations by the pairwise test case suite. This can confirmed by the classes (except Proposition.java) under the package "prospec.model.proposition". The full measurements broken down by classes and methods are given in Appendix C.

5.1.3 Prospec Algorithm Coverage Analysis

The second implementation, the Prospec Algorithm was measured using the pairwise test suite described in Chapter 4. The results of the coverage measurement can be seen in Table 5.3.

Table 5. 3 Branch Coverage for the Prospec Algorithm Implementation

Prospec Algorithm Implementation	Test Suite	Edges touched	Total edges	Branch Coverage
Total Covered	132 Tests	1410	1981	71.14%

The results for this coverage are aligned with the expectation that pairwise testing will not provide complete coverage, and as such over 70% of coverage is a solid result for only 132 tests out of more than 31,000 possible combinations. For further study, the Pairwiser online tool will generate different set of tests depending on the order of the parameters. An addition two sets of tests were generated, which a different combination and number of tests for the pairs. The coverage was measured for these test suites and it was found that they were consistent with each other. Table 5.4 shows the coverage results for other pairwise test suites.

Table 5. 4 Branch Coverage for Different Pairwise Test Suites

Prospec Algorithm Implementation	Test Suite	Edges touched	Total edges	Branch Coverage
First Test Suite	132 Tests	1410	1981	71.14%
Variation 1 Test Suite	136 Tests	1430	1989	71.90%
Variation 2 Test Suite	137 Tests	1427	1989	71.75%

5.1.4 Verification Gap Analysis

Knowing the branch coverage percentage can be useful in gauging the amount that the system is actually tested by a pairwise test suite. However, and more importantly, the parts of the system that were not touched need to be identified. First, just as it was observed on the SimpleLTL Generator, some pieces of the system might be unreachable through testing. Secondly, identifying the gaps in the testing coverage can be a guiding factor in deciding how to complement pair-wise testing with other verification techniques.

To identify these untested parts of the code, the tool VerificationGapMiner was developed. The purpose of the tool is -to traverse through the system's BF-CFGs, checking which parts of the system have not been tested at all, i.e., the untouched edges. Once these edges are identified, the tool can gather what parts of the code correspond to the untested edges and map them to actual lines in the system. A sample result is shown for the SimpleLTL Generator in Fig. 5.2.

To show that the VerificationGapMiner tool correctly identifies these untested edges, a

```
\prospec\model\scope\BetweenLandR\setR(prospec.model.proposition.Proposition).xml GAP @ 29 -> 30
\prospec\model\scope\BetweenLandR\setR(prospec.model.proposition.Proposition).xml GAP @ 30 -> 31
\prospec\model\scope\BetweenLandR\setR(prospec.model.proposition.Proposition).xml GAP @ 31 -> EXIT
```

Figure 5. 2 Small Output of the VerificationGapMiner

formal inspection was performed over the implementation. The following are the properties established to act as the basis of the verification tasks.

- Property 1: The *VerificationGapMiner* tool inspects every BF-CFG for a given implementation.
- Property 2: The *VerificationGapMiner* tool detects the correct corresponding lines of code for untested edges.
- Property 3: The *VerificationGapMiner* tool stores the results in a file.

The formal inspection was done again by a faculty member in the Computer Science Department at UTEP. The full list and description of the inspection tasks is listed in Appendix A. The results of the inspection, which found no defects, is documented in Table 5.5.

Table 5. 5 VerificationGapMiner Mapping of Properties to Inspection Tasks.

Inspection Task ID	Property	Pass/Fail	Comments
1	1	Pass	31-35
2	2	Pass	52
3	2	Pass	53
4	2	Pass	73
5	2	Pass	74
6	2	Pass	65-66
7	2	Pass	76
8	3	Pass	31-35

5.1.5 Gap Analysis of the Implementations

Putting the SimpleLTL Generator implementation through the *VerificationGapMiner* tool yielded the lines of code and possible execution paths of the untested system. The full results are documented on Appendix X. Analysis of the results confirm that the majority of the code is unreachable code that remained from the model subsystem being part of another system. Additionally, other parts of the code are not touched because they are only used for error handling. An example is the code shown in Fig. 5.3. Line 406 from the SimpleLTL is not reached because the code used to display an error. This path is not easily reachable by testing the system as a whole.

```
400                 {
401                     if (P.getScope() instanceof AfterL)
402                     {
403                         return generatePrecedenceAfterL(P);
404                     }
405                     else
406                     {
407                         return "Error, invalid pattern or scope combination defined, did you extend scope and patterns?";
408                     }
409                 }
```

Figure 5. 3 Unreachable Code Example at Line 406.

A more appropriate technique would be to do a walkthrough or a formal inspection on this section of code. This would ensure that the code is verified at this point, without having to take apart the system to reach this line.

5.1.6 Summary of Analysis for Proposition 1

Two observations can be made after analyzing the data collected while measuring the branch coverage and identifying the verification gaps.

Result 1: The Box-Fusion approach can identify the gaps in the pairwise testing. This is complementary to gathering the branch coverage data, which the Box-Fusion approach also supports.

Observation 1: Using the tools to determine gaps in the coverage can also serve as a resource that can be used to support other verification techniques, such as walkthroughs and inspections. In addition, the result from the gap analysis can lead to a broader approach than using unit testing because the pairwise tests support system testing by its nature. What this means is that if the tester wishes to target a specific set of lines, coming up with the test data to reach that path might be difficult. Instead, one of the capabilities of the Box-Fusion approach when using gap verification identification is the benefit of being able to systematically verify an integrated system by following the execution paths to inspect or walk through the code.

5.2 Proposition 2 Analysis: The Box-Fusion Approach Identifies Defect Location

This section details the analysis performed for the case study for Proposition 2. The goal of this proposition is to show that the Box-Fusion approach can establish the location of a defect within an execution path generated from the BF-CFGs. Two target systems are used, the SimpleLTL Generator and the full Prospec implementation.

5.2.1 SimpleLTL Generator Analysis

As previously described in Chapter 3, the SimpleLTL Generator is an implementation that generates LTL formulas using only atomic propositions. To be able to identify the potential location of the defects, the execution path had to be identified from the BF-CFG. An execution

path will show the sequence and location of the SUT as the testing data progressed through the system. To expose this data, a tool called *ExecutionPathFinder* was developed. This tool receives as input a Test ID and a location of BF-CFGs, and extracts the execution path for that Test ID from the BF-CFGs. A snippet output is shown in Fig. 5.4, and a full example can be seen in Appendix D. The Step number indicates the order of the execution, the second column describes the BF-CFG and edge touched, denoted by "Line: N1 -> Line: N2", which signifies that the edge connecting Line 6 to Line 7 was traversed. A line with Class.Method -> BF-CFG indicates transfer of control from one BF-CFG to another, e.g. Step 21 and Step 24 on Fig 5.4.

```
Step 9 Absence(prospec.model.proposition.Proposition).xml Line:START -> Line:6
Step 10 Absence(prospec.model.proposition.Proposition).xml Line:6 -> Line:7
Step 11 Absence(prospec.model.proposition.Proposition).xml Line:7 -> Line:8
Step 12 Absence(prospec.model.proposition.Proposition).xml Line:8 -> Line:9
Step 13 Absence(prospec.model.proposition.Proposition).xml Line:9 -> Line:EXIT
Step 14 Property(prospec.model.scope.Scope,Lprospec.model.pattern.Pattern).xml Line:START -> Line:22
Step 16 Property(prospec.model.scope.Scope,Lprospec.model.pattern.Pattern).xml Line:22 -> Line:23
Step 17 Property(prospec.model.scope.Scope,Lprospec.model.pattern.Pattern).xml Line:23 -> Line:24
Step 18 Property(prospec.model.scope.Scope,Lprospec.model.pattern.Pattern).xml Line:24 -> Line:25
Step 19 Property(prospec.model.scope.Scope,Lprospec.model.pattern.Pattern).xml Line:25 -> Line:26
Step 20 Property(prospec.model.scope.Scope,Lprospec.model.pattern.Pattern).xml Line:26 -> Line:EXIT
Step 21 DriverLTL.main->\prospec\generator\SimpleLTL_Generator\getStringRepresentation(prospec.model.property.Property).xml
Step 22 getStringRepresentation(prospec.model.property.Property).xml Line:START -> Line:168
Step 23 getStringRepresentation(prospec.model.property.Property).xml Line:168 -> Line:169
Step 24 SimpleLTL_Generator.getStringRepresentation->\prospec\model\property\Property\getPattern().xml
Step 25 getPattern().xml Line:START -> Line:60
Step 26 getPattern().xml Line:60 -> Line:61
Step 27 getPattern().xml Line:61 -> Line:EXIT
```

Figure 5. 4 Example of Output of the ExecutionPathFinder Implementation.

To demonstrate that the *ExecutionPathFinder* tool was correctly implemented and will display the execution path for a testID. The following are the properties established to act as the basis of the verification tasks.

- Property 1: The *ExecutionPathFinder* tool inspects every BF-CFG for a given implementation.
- Property 2: The *ExecutionPathFinder* tool detects the corresponding lines of code for a given testID edges.
- Property 3: The *ExecutionPathFinder* displays the Execution Path in incremental order.

The formal inspection was executed by a faculty member of the Computer Science Department at UTEP. The full list and description of the inspection tasks is listed in Appendix A. The results of the inspection is documented in Table 5.6. The inspection found one defect.

Table 5. 6 ExecutionPathFinder Mapping of Properties to Inspection Tasks.

Inspection Task ID	Property	Pass/Fail	Comments
1	1	Pass	46-47
2	2	Pass	59-61
3	2	Pass	112
4	2	Pass	113
5	2	Pass	Val: 164 ID 159
6	2	Pass	171
7	3	Pass	173-175
8	2	Pass	117
9	2	Pass	124-136
10	3	Fail	Defect 1
11	3	Pass	71-74

Defect 1

Issue Description: This defect was deemed as fail because the output of a method call step is in the wrong format according to Task 10.

Resolution: The documentation was corrected.

Now that the Box-Fusion algorithm has been verified, the next task is to show that the Box-Fusion approach supports the ability to identify defects as stated in Proposition 2. Towards this purpose, the system was seeded with defects by a person with no knowledge of the code and the defects were unbeknown to the author. The system was turned over to the author with the seeded defects as shown below in Table 5.7.

The Box-Fusion approach was applied to the SUT, BF-CFGs were created and instrumentation was applied to the SimpleLTL implementation. The Pairwiser tool generated 30 test cases, which is the exhaustive combination of inputs for this program. For this section, Test IDs 1..N match the 30 test cases defined in the test suite. In order to facilitate the testing, a test

driver was developed to initialize, input and execute all 30 formulas to the SUT, outputting the results to the console and storing them in a file.

The results of the testing showed that Test IDs 3, 4, 5, 10, 14, 16, 22, 23, 24 and 28 failed.

Table 5. 7 SimpleLTL Generator Defects Seeded.

No.	Line (SimpleLTL_Generator.java)	Defect Seeded	Comments
1	19	Changed "AND ¬" to "OR"	
2	39	Changed ") U (" to ")(" and "AND ¬<> " to "AND <>"	
3	59	Took out of one of the closing parenthesis	
4	69	Changed "¬<>" to "¬[]"	
5	99	Changed "¬((¬" to "¬(¬("	
6	104	Changed "AND" to "OR"	
7	164	Changed "→<>" to "AND <>"	
8	181	Changed "Response" to "Universality"	

The output of the seeded program is shown in Fig. 5.5. The Test ID is associated with the output LTL.

The code was versioned as SimpleLTL Seeded V1.0, to establish a baseline for the seeds introduced.

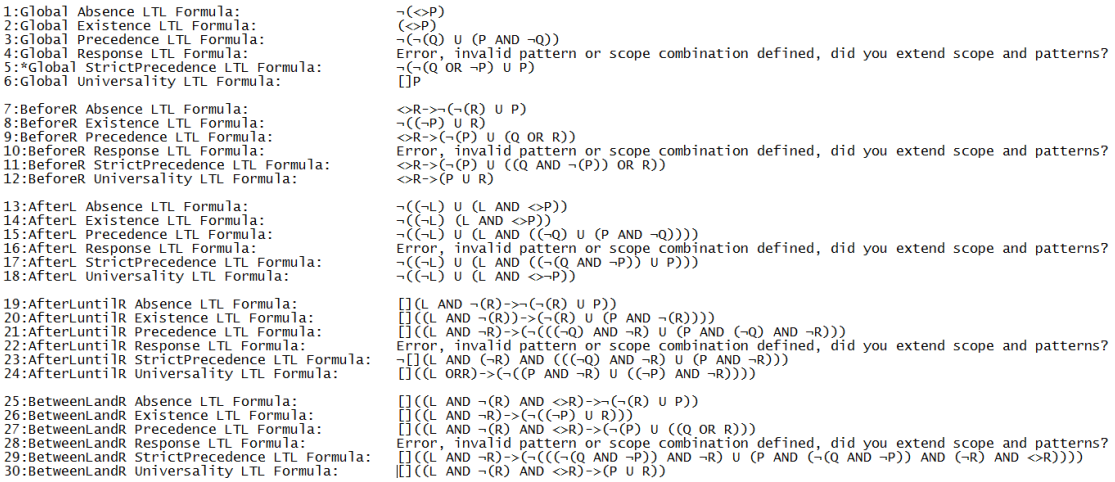


Figure 5. 5 Seeded SimpleLTL Output.

The process to detect and correct the seeded defects while conducting this case study is defined as follows:

1. Record time of start.
2. Select the Test ID at the front of the list and record the failed Test ID.
3. Extract the execution path for this Test ID using the *ExecutionPathFinder* tool.
4. Using the execution path, follow the code and examine every line it touches to identify the source of the defect.
5. Once the defect is located, correct the defect and record the line number.
6. Determine the impact of changing the line(s) of code by using the *RegressionFinder* tool (see Section 5.4 for details on this tool). Record Test IDs identified by tool.
7. Re-test selected failed Test ID. Record if test passes/fails.
8. Reorganize the list of failed Test IDs placing Test IDs identified by the regression tool towards the end of the list.
9. Record ending time.
10. Create a new version of the SUT.

Table 5. 8 Data Collected While Correcting Seeded Simple LTL Generator.

Test ID	Defect Line (SimpleLTL_Generator.java)	Regression Results	Version of the Code	Observation	Time Spent (mins)
3	99	3	SimpleLTL Seeded V1.1	Test passed after correction.	11
4	181	10, 14, 16, 22, 28	SimpleLTL Seeded V1.2	Test failed after first correction.	9
4	164	4	SimpleLTL Seeded V1.3	Test passed after second correction.	4
5	104	5	SimpleLTL Seeded V1.4	Test passed after correction.	7
23	69	23	SimpleLTL Seeded V1.5	Test passed after correction.	5
24	19	24	SimpleLTL Seeded V1.6	Test passed after correction.	3
10	N/A	N/A	N/A	Test 10 passed by the correction applied to Test 4.	1
14	39	14	SimpleLTL Seeded V1.7	Test passed after correction.	4
16	N/A	N/A	N/A	Test 16 passed by the correction applied for Test 4.	1
22	59	22	SimpleLTL Seeded V1.8	Test passed after correction.	3
28	N/A	N/A	N/A	Test 28 passed by the correction applied for Test 4.	1

The results of the process applied to correcting the seeded defects can be seen in Table 5.8. The first column lists the ordering of correction of seeded defects, per Test ID. The second column contains the line number where the defect was found; all of the seeded defects belonged to the same class. The third column displays the Test IDs that were identified by the regression tool as impacted by the change in the line of code. The fourth and fifth column contain the version number for each correction applied and any observations made, respectively. The final column displays the time spent on following the process above and correcting each defect.

5.2.2 Prospec LTL Implementation Analysis

The other part of the study for Proposition 2 is the application of the Box-Fusion approach to the Prospec Algorithm implementation. This system implements composite propositions which explode the input combinations to over 31,000. This part of the study will focus on detecting the location of a defect that affects properties of the type *Response-AfterLuntilR*. The defect that all these formulas exhibit is the error of having a closing parenthesis followed immediately by an open parenthesis. The process will be similar to one applied in detecting seeded defects in the Simple LTL Generator. The pairwise Test IDs that were identified to exhibit the parenthesis defect are: 2, 5, 17, 20, 23, 28, 30, 54, 58, 59, and 60.

Table 5. 9 Participant 1 Data Collected While Correcting the Prospect Algorithm Implementation.

Test ID	Defect Line	Regression Results	Version of the Code	Observation	Time Spent
2	AfterLUntilRc Line 30	2,5,17,20,23, 28,30, 54,58,59,60	T2 V1.1	Test passed after correction.	1 hour 45 mins
5	N/A	N/A	T2 V1.1	Test passed. Fixed by correction for Test 2.	3mins
17	N/A	N/A	T2 V1.1	Test passed. Fixed by correction for Test 2.	2mins
20	N/A	N/A	T2 V1.1	Test passed. Fixed by correction for Test 2.	1min
23	N/A	N/A	T2 V1.1	Test passed. Fixed by correction for Test 2.	1min
28	N/A	N/A	T2 V1.1	Test passed. Fixed by correction for Test 2.	1min
30	N/A	N/A	T2 V1.1	Test passed. Fixed by correction for Test 2.	1min
54	N/A	N/A	T2 V1.1	Test passed. Fixed by correction for Test 2.	30secs
58	N/A	N/A	T2 V1.1	Test passed. Fixed by correction for Test 2.	30secs
59	N/A	N/A	T2 V1.1	Test passed. Fixed by correction for Test 2.	30secs
60	N/A	N/A	T2 V1.1	Test passed. Fixed by correction for Test 2.	30secs

Table 5.9 and Table 5.10 display the results of the following the process to detect the defect, following the same descriptions as the results table described in Section 5.2.1.

Table 5. 10 Participant 2 Data Collected While Correcting the Prospect Algorithm Implementation.

Test ID	Defect Line	Regression Results	Version of the Code	Observation	Time Spent
2	AfterLUntilRc Line 30	2,5,17,20,23, 28,30, 54,58,59,60	T2 V2.1	Test passed after correction.	1 hour 59 minutes
5	N/A	N/A	T2 V2.1	Test passed. Fixed by correction for Test 2.	1 min
17	N/A	N/A	T2 V2.1	Test passed. Fixed by correction for Test 2.	1 min
20	N/A	N/A	T2 V2.1	Test passed. Fixed by correction for Test 2.	30 secs
23	N/A	N/A	T2 V2.1	Test passed. Fixed by correction for Test 2.	30 secs
28	N/A	N/A	T2 V2.1	Test passed. Fixed by correction for Test 2.	30 secs
30	N/A	N/A	T2 V2.1	Test passed. Fixed by correction for Test 2.	30 secs
54	N/A	N/A	T2 V2.1	Test passed. Fixed by correction for Test 2.	30 secs
58	N/A	N/A	T2 V2.1	Test passed. Fixed by correction for Test 2.	30 secs
59	N/A	N/A	T2 V2.1	Test passed. Fixed by correction for Test 2.	30 secs
60	N/A	N/A	T2 V2.1	Test passed. Fixed by correction for Test 2.	30 secs

After 1 hour and 45 minutes, Participant 1 located the error on line 30 as shown in Fig. 5.6. Regression testing was applied to the line, and the resulting test IDs were all marked as candidates for regression testing. A new version was documented and observations annotated. Subsequent testing of the rest of the formulas found that the error had been corrected for every one of them. The same process was repeated, and Participant 2 located the same defect in 1 hour and 59 minutes. The error is shown in Fig 5.6, showing the line before and after the correction was inserted.

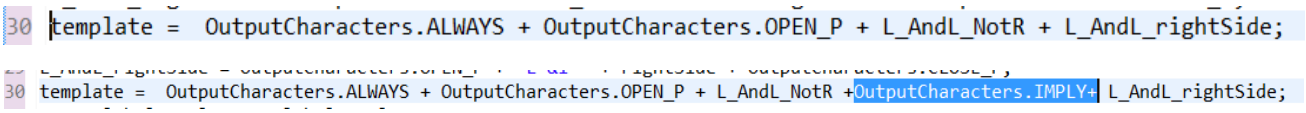


Figure 5. 6 Defect Before and Correction at line 30.

5.2.3 Summary of Analysis for Proposition 2

Three observations can be made after analyzing the data collected while correcting the seeded defects.

Result 1: The Box-Fusion approach does indeed capture the defect location within an execution path, directly supporting Proposition 2. The data in defect Table 5.6 in Section 5.2.1 matches exactly what was seeded, and following the execution path strictly led into the location of the defects. For the second implementation, following the path provided by the *ExecutionPathFinder* tool yielded the defect for all the tests in that set.

Observation 1: For the SimpleLTL Generator, an interesting feature that was gleamed from this analysis is that the execution path assisted in the correction of the second seeded defect found under Test ID 4 in an unexpected way. After it was determined that the test still failed, it was discovered that there was no need to start from the beginning of the execution path, but instead the developer could continue from where the defect was fixed. The execution path had already been inspected; therefore, there was no need to inspect that part again.

Observation 2: Another observation of interest was made when using the regression tool. Because the tool identifies the pairwise tests that touch each node, using the tool can give a measure of the impact that a modification might have in the system. For example, for Test ID 4 in the seeded SimpleLTL Generator, the regression tool identified that 20 other tests passed through this line of code. Meaning that a change in this part of the system could have affect other parts of the system. The impact of this change was seen because by correcting this defect, the defects associated with

test IDs 10, 26, and 28 were also corrected. This approach can provide a way to capture what tests could be impacted by a change in a part of the code.

5.3 Proposition 3: Analysis of the Box-Fusion Approach To Narrow Defect Location

This section discussed potential ways to use the Box-Fusion approach to narrow down the location of defects in a system. Because pairwise testing is a black-box approach, a failed test case will not say anything about the location of the defect. For this proposition, the study aimed to explore what potential techniques could be used in conjunction with the data of failed/passed tests, and knowledge of the execution path to guide the developer on the potential location of defects.

The first step is to select a good set of tests to use for this proposition. In order to have a much better understanding of what techniques might work, the selection of the test cases focused on a known defect. For this analysis, 10 test cases from the pairwise test suite that contained the *Response-AfterLUntilR* were selected, and the defect was the one found on section 5.2.2, the missing "->" in the formula.

Exploring the set of tests using the *ExecutionPathFinder* quickly shows that the execution paths are lengthy and hard to abstract. Attempting to find any similarities among the execution paths of failed test cases through examination was not feasible; tool support is required. The first tool to be developed to confirm Proposition 3 was a tool that would identify all the common nodes for failed test IDs. However, the resulting intersection was still too lengthy and ineffective at assisting in the reducing the number of nodes to be examined. The next approach was to abstract the data to facilitate the understanding; the premise for this approach is that defects are located within the BF-CFG (under the Origin tag), and the BF-CFG contains method calls. The next tool developed identified the intersection of method calls between test IDs, displays the size of the intersection, and the methods in the list of test IDs. This tool is called *CFGIntersectionMethodFinder*. This tool was formally inspected to show correctness in its implementation. The Properties for this tool are as follow:

- Property 1: The *CFGIntersectionMethodFinder* tool inspects every BF-CFG for a given set of test IDs.
- Property 2: The *CFGIntersectionMethodFinder* identifies all the methods that called this BF-CFG and matched the given testIDs and puts them into lists for each test ID.
- Property 3: The *CFGIntersectionMethodFinder* calculates the intersection of each test ID list and displays the size of the commonality and methods in the list.

The formal inspection was done by a faculty member of the Computer Science Department at UTEP. The full list and description of the inspection tasks is listed in Appendix A. The results of the inspection is documented in Table 5.11. The results of the inspection found one defect.

Table 5. 11 CFGIntersectionMethodFinder Mapping of Properties to Inspection Tasks.

Inspection Task ID	Property	Pass/Fail	Comments
1	1	Pass	46-47
2	1	Pass	55-59
3	2	Pass	132-133
4	2	Pass	143-151
5	2	Fail	Defect 1
6	3	Pass	61-76

Defect 1

- **Issue Description:** The output format does not match the expected format on the inspection item.
- **Resolution:** The defect has been documented for a future update. The defect does not impact the output. The difference is: …\calleeClass\callerMethod instead of <calleeClassName>.<callerMethodName>.

An example output of the *CFGIntersectionMethodFinder* is shown below in Fig 5.7 for input test IDs: 2, 5, and 17. The output shows how the number of intersection execution path methods shrink from 47 to 35.

```
Test ID: 2 5 17
47
47
35
OutputToFile.outputInitialMessage->\edu\tx\utep\ltlgenerator\inputoutput\OutputToFile\getDateTime().xml
OperatorGenerator.getAndedPropositions->\edu\tx\utep\ltlgenerator\factories\OperatorFactory\getOperator(
OutputToFile.outputLTLFormulaResultSet->\edu\tx\utep\ltlgenerator\inputoutput\OutputToFile\saveResultsTo
```

Figure 5. 7 Sample Output of the CFGIntersectionMethodFinder.

Running the tool on the tests that exhibit the same defect (i.e., test IDs 2, 5, 17, 20, 23…)

Table 5. 12 Calculating the Intersecting Method Numbers for the Defect Group.

Test Group ID	Test ID	Intersecting number of BF-CFGs
1	2	57
2	2,5	57
3	2, 5, 17	35
4	2,5,17,20	35
5	2,5,17,20, 23	35
6	2,5,17,20, 23, 28	35
7	2,5,17,20, 23, 28, 30	34
8	2,5,17,20, 23, 28, 30, 54	34
9	2,5,17,20, 23, 28, 30, 54, 58	34
10	2,5,17,20, 23, 28, 30, 54, 58, 59	34
11	2,5,17,20, 23, 28, 30, 54, 58, 59, 60	34

yields a reduction to 35 common methods (as can be seen in Table 5.12). If another test is added, a test that is not part of the defect group, such as test ID 4, the intersections drops to 23 common methods. Thus, if the test IDs represent the same class of defect, e.g., they impact the same categories of pattern and scope of the Prospec algorithm, then the intersection can reduce the execution path that needs to be examined. Reconsider the example of finding the defect for test ID 2. If it is known that the defects exposed by a set of test IDs have similar characteristics, then

by calculating the intersection, the number of BF-CFGs to examine may be lowered. To confirm this observation, taking the intersection of tests IDs 2, 5, and 17 (which are part of the same defect group) leads to 35 nodes in the BF-CFG, indicating that the defect is in the intersection since the intersection represents the code shared by the execution paths. Table 5.10 shows the results of calculating the *CFGIntersectionMethodFinder* for the defect group, adding one by one and calculating the number of methods in the intersection.

It is not always known if tests are exhibiting the same defect; however, tests that pass and share common code are likely to not contain defects in the common code. If this is true, the *CFGIntersectionMethodFinder* can be used to identify the common methods in the passing tests. Thus, when following an execution path, methods that belong to the intersection of passing tests can be lowered in priority as they are least likely to have a defect. The more tests that pass, the common code is more likely to not contain the defect.

The proposed steps to utilize the intersection from the same class of defects is:

1. For N passing tests, identify corresponding test IDs.
2. Calculate *CFGIntersectionMethodFinder* of the test IDs from Step 1.
3. For M failed tests that exhibit the same defect, identify their test IDs
4. Calculate *CFGIntersectionMethodFinder* of the test IDs from Step 3.
5. Execute the *ExecutionPathFinder* on a test from M.
6. On the execution path generated by Step 5, methods listed by Step 4 have a higher probability of containing the defect, while methods listed by Step 2 have a lower probability of containing the defect.

5.3.2 Summary of Analysis for Proposition 3

Two observations can be made after analyzing the data collected while measuring the branch coverage and identifying the verification gaps.

Observation 1: Failed tests coming from a pairwise test suite that exhibit the same defect can be used to narrow down the location of the common defect. This can be done by using the

CFGIntersectionMethodFinder tool to determine what methods are common between a set of tests. Those intersecting methods represent the commonality of the tests, and if the defect is the same, then the defect must be located in the common code.

Observation 2: The intersection of methods can also be used to identify common code between tests that have passed. This common code represents code that has a high probability of not being defective, because many tests that pass execute correctly over the code. This information can be used to lower those common methods in priority, leaving them as the least likely place to contain defective code. As more tests pass, the easier it can become to detect the defects in the code using this approach.

5.4 Proposition 4: The Box-Fusion Approach Can Optimize Regression Testing

This section discusses the analysis of using the Box-Fusion approach to determine regression tests. This is particularly important for black-box testing approaches, since selecting tests to run after a modification may not be as efficient, and may over test the system.

5.4.1 Regression Testing

To determine what tests passed through the nodes where a modification has been made, a tool called *RegressionFinder* was developed. The tool takes as input a BF-CFG and a line code that represents the line changed. The tool will find the edge corresponding to that line and extract all the test IDs that pass through that edge. This tool was also formally inspected to show correctness in its implementation. The properties for this tool are as follow:

- Property 1: The *RegressionFinder* searches the provided BF-CFG and identifies the edge associated with the line given.
- Property 2: The *RegressionFinder* returns all the test IDs that match the edge associated with the line given.

The formal inspection was executed by a faculty member of the Computer Science Department at UTEP. The full list and description of the inspection tasks is listed in Appendix A. The results of the inspection is documented in Table 5.13. The results of the inspection found no

Table 5. 13 RegressionFinder Mapping of Properties to Inspection Tasks.

Inspection Task ID	Property	Pass/Fail	Comments
1	1	Pass	41 and 78
2	1	Pass	63-81
3	1	Pass	106
4	2	Pass	107
5	2	Pass	127
6	2	Pass	134
7	3	Pass	133-136
8	2	Pass	43-43

defects.

To show that the regression approach does correctly identify all the tests impacted by a change, the following steps are performed on the Prospec Algorithm implementation:

1 Insert a modification on the SUT, create new Version: T2.REG1.

2 *RegressionFinder* that line number. Store this set of test IDs as RegressTestSet.

3 Remove RegressTestSet from the pairwise test suite.

4 Create a new pairwise test input containing the remaining tests, name it PairTestsMinus.

5 Run the testing with the PairTestsMinus

6 Search for the inserted modification.

7 If found, then document that *RegressionFinder* did not find all the tests.

8 If not found, then document that *RegressionFinder* found all the impacted tests.

9 Run testing with RegressTestSet.

10 Ensure every test yields a modified output.

The results of the tasks above are presented in Table 5.14.

Table 5. 14 Results of Regression Steps on SUT.

Step Number	Action	Test Set	Test IDs	Result
1	Modify Line 30 in QRespondsToPBeforeRe	N/A	N/A	New T2REG1 SUT
2	*RegressionFinder*	RegressTestSet	12,21,27,35,37,44,53, 55,61,64,71,84,120	New RegressTestSet
3	Remove RegressTestSet from pairwise set	N/A	N/A	
4	Create new PairTestsMinus	PairTestsMinus	All minus RegressTestSet	NewPairTestMinus
5	Execute Testing	PairTestsMinus	All minus RegressTestSet	Testing Complete
6	Search for modification	N/A	N/A	Modification not found
7	If found	N/A	N/A	N/A
8	If not found	N/A	N/A	*RegressionFinder found all the impacted tests*
9	Execute Testing	RegressTestSet	N/A	
10	Did every test result have the modification?			True

5.4.2 Summary of Analysis for Proposition 4

Two observations can be made after analyzing the data collected while measuring the branch coverage and identifying the verification gaps.

Results 1: The regression exercise correctly identified necessary regression tests and no extra testing, thus showing that the results are consistent with Proposition 4. As expected, the set of tests that did not contain the modification yielded the same results as the original test run.

Observation 1: An observation has been noted about the potential capability of the regression testing component of the Box-Fusion approach. When a defect correction is applied to the system, the regression testing done on that section of the SUT will identify the tests that are impacted, as briefly mentioned in Section 5.1. The tests that are identified at this point can be particularly useful in the following two ways: 1) If there is a large number of tests that have

failed in the pairwise test suite, then the impact of this is that tests that were identified by the regression testing algorithm should be delayed in priority since they might have been fixed by the changes. 2) If the majority of tests in the pairwise test are passing, then the impact of tests identified by regression testing warns of extra testing that must be done, in case the change caused the introduction of a new unintended defect.

Chapter 6: Related Work

This chapter describes techniques and strategies that are related to the enhancements provided by the Box-Fusion approach. The first section describes an approach to extend pairwise testing to generate white box test cases. The second section talks about existing tools for measuring coverage. The last section discusses existing work on selecting test cases for regression testing.

6.1. White Box Pairwise Test Case Generation

The White Box Pairwise test case generation [41] is a white-box extension to traditional black-box pairwise test case generation. This extension selects additional test cases for the system based on specifications for one or more internal sub-operations. The algorithm for generating test cases for the full system achieve pairwise coverage of the sub-operations. The authors based their *White Box Pairwise* (WBPairwise) algorithm on a case study for an elevator door control mechanism. The system has 14 parameters and served 3 floors, which yielded an exhaustive testing total of 2,359,296 test cases. They applied both the pairwise method and WBPairwise to 500 different input parameter sets with different orders. The results indicated that WBPairwise testing is both practical and effective. WBPairwise alone performed nearly as well as pairwise testing. The authors concluded that the number of test cases generated and the algorithm execution run times are reasonable. They also could show that White box test sets are effective at revealing faults, and when combined with black-box tests such as pairwise, they could improve the fault detection by nearly 4% [41].

The algorithm is recursive and traverses the system tree depth first. For each visited node N there are three phases:

1. Child processing. Two kinds of test sets are generated. First, a pairwise test set is generated for N's inputs, based solely on the domains of N's children. More precisely, test set B is generated by applying an algorithm such as IPO [63] to the Cartesian product of

N.c0,N.c1,…,N.cn−1, where n = N.c.len. Then, the sequence W of test sets is generated, by recursively calling WBPairwise once for each of N's children.

2. Horizontal expansion. Each test case b in B is expanded horizontally by replacing bi with an element of W*i*. Initially, b has one element for each child of N. At the end of this phase, b will have one element for each leaf in the subtree rooted at N.
3. Vertical expansion. The horizontal expansion phase inserts elements of W into elements of B. Because the elements of W provide pairwise coverage of N's children, it is essential that every element of W be selected for insertion at least once. If this is not the case then new test cases are added in this phase for the uncovered elements of W.

The authors evaluated the approach and the algorithm with a case study. The results indicated that WBPairwise testing is both practical and effective. The number of test cases generated and the algorithm execution run times are reasonable. By themselves, white-box test sets are effective at revealing faults. In combination with black box test sets, white box test sets offer improved fault detection as can be seen in Figure 6.1. In comparison with the Box-Fusion approach, the work in the white box pairwise testing focuses on one or more internal sub-operations, while the Box-Fusion approach focuses on the system as a whole.

	Number of test cases	Execution time (seconds)	Number of mutants killed	Percent of mutants killed
WBPairwsie	64.54	6.2	192.5	79.9
BBPairwise	128.00	2.0	215.0	89.2
Combined	192.42	-	224.5	92.9
All Combinations	2,359,296	160	241	100.0

Figure 6. 1 Performance Gain with WBPairwise [41].

6.2. COVERAGE TOOLS

This section discussed similar types of tool that aim at measuring the code coverage from testing. These tools are specific to the Java language, and focus on coverage level analysis.

6.2.1. JCov a Java Code Coverage Tool

The JCov open source project [42] is used to gather quality metrics associated with the production of test suites. JCov is being opened in order to facilitate the practice of verifying test execution of regression tests in OpenJDK development. The main motivation behind JCov is *transparency of test coverage metrics*. JCov is a java code coverage tool which provides a means to measure and analyze dynamic code coverage of Java programs. JCov provides functionality to collect method, linear block and branch coverage, as well as showing uncovered execution paths. It is also able to show a program's source code annotated with coverage information. JCov works by instrumenting Java bytecode using two different approaches: 1) static instrumentation which is done upfront, changing the tested code; and 2) dynamic instrumentation which is done on the fly by means of Java agent [42]. From a testing perspective, JCov is most useful to determine execution paths (in a Java application) that a test suite is (or is not) executing. JCov supports applications on JDK 1.0 and higher (including JDK 8), CDC/CLDC 1.0 and higher, and JavaCard 3.0 and higher.

6.2.2. JaCoCo a byte-code analyzer tool for test coverage

JaCoCo is an open source toolkit for measuring and reporting Java code coverage. JaCoCo is distributed under the terms of the Eclipse Public License. It was developed as a replacement for EMMA under the umbrella of the EclEmma eclipse project [43]. JaCoCo offers line and branch coverage. JavaCodeCoverage is a byte-code analyser tool for test coverage analysis for Java software which neither requires neither the language grammar nor the source code [44]. An important aspect of JavaCodeCoverage is that it stores the coverage information for individual test case thereby facilitating detailed coverage analysis. Another important aspect of JavaCodeCoverage is that it records all vital code-elements and test coverage information in open source database software MySQL [45].

6.2.3. JCover

JCover [46] is a code coverage analyzer for Java programs. It provides a mechanism to generate statistical information on the coverage of an application during a test run. It can be used to calculate the percentage of code that was executed, percentage not executed, what sources were not used in files and so on. JCover supports statement and branch coverage.

6.2.4. Cobertura

Cobertura [47], a free Java tool that calculates the percentage code accessed by tests, shows the McCabe cyclomatic code complexity of each class, and the average cyclomatic code complexity for each package and for the overall product. Cyclomatic complexity represents the number of paths through a particular section of the code, such as a method in an object-oriented language. It is helpful in pinpointing areas of code that may require additional attention during testing, maintenance or refactoring.

6.2.5. Comparison with Box-Fusion approach

The tools mentioned in this section focus on the coverage of testing, but not for any particular type of testing. The focus of the Box-Fusion approach is to capture pairwise testing data into Box-Fusion Control Flow Graphs and goes beyond just examining coverage. The approach allows the developers to analyze gaps and assist in fault detection and regression testing.

6.3. REGRESSION TESTING

Regression testing is a testing activity that is performed to provide confidence that changes do not harm the existing behavior of the software. When a modification occurs to the system, regression testing must be efficient and only execute the necessary tests. The following efforts describes the support that is provided to assist in regression testing.

6.3.1. Test Suite Reduction

Jeffrey and Gupta extended the HGS (Harrold-Gupta-Soffa) heuristic [50] so that certain test cases are selectively retained [51, 52]. This ‘selective redundancy’ is obtained by introducing a secondary set of testing requirements. When a test case is marked as redundant with respect to

the first set of testing requirements, Jeffrey and Gupta considered whether the test case is also redundant with respect to the second set of testing requirements. If it is not, the test case is still selected, resulting in a certain level of redundancy with respect to the first set of testing requirements. The empirical evaluation used branch coverage as the first set of testing requirements and all-uses coverage information obtained by data-flow analysis. The results were compared to two versions of the HGS heuristic based on branch coverage and def-use coverage. The results showed that, while their technique produced larger test suites, the fault detection capability was better preserved compared to single-criterion versions of the HGS heuristic. This approach focuses on reducing the redundancy of testing due to maintenance of a system. The Box-Fusion approach focuses on detecting testing that needs to be done to correctly regress test a modification.

Schroeder and Korel proposed an approach of test suite minimization for black-box software testing [53]. They noted that the traditional approach of testing black-box software with combinatorial test suites may result in redundancy, since certain inputs to the software may not affect the outcome of the output being tested. They first identified, for each output variable, the set of input variables that can affect the outcome. Then, for each output variable, an individual combinatorial test suite is generated with respect to only those input variables that may affect the outcome. The overall test suite is a union of all combinatorial test suites for individual output variables. This approach requires a mapping from the actual code change to the input variable, while the Box-Fusion approach uses the location of the modification to determine the affected tests.

6.3.2. Dynamic Slicing Based Approach

Yau and Kishmoto presented a test case selection technique based on symbolic execution of the SUT [54]. In symbolic execution of a program, the variables' values are treated as symbols, rather than concrete values [55]. Yau and Kishimoto's approach can be thought of as an application of symbolic execution and input partitioning to the test case selection problem. First,

the technique statically analyses the code and specifications to determine the input partitions. Next, it produces test cases so that each input partition can be executed at least once. Given information on where the code has been modified, the technique then identifies the edges in the control flow graph that lead to the modified code. While symbolically executing all test cases, the technique determines test cases that traverse edges that do not reach any modification. The technique then selects all test cases that reach new or modified code. For the symbolic test cases that reach modifications, the technique completes the execution; the real test cases that match these symbolic test cases should be retested.

While it is theoretically powerful, the most important drawback of the symbolic execution approach is the algorithmic complexity of the symbolic execution [56]. Yau and Kishmoto acknowledge that symbolic execution can be very expensive. Pointer arithmetic can also present challenging problems for symbolic execution based approaches. In comparison, the Box-Fusion approach is a simpler approach as compared to symbolic execution.

6.3.3. Graph-Walk Approach

Rothermel and Harrold proposed the graph walking approach based on CFGs [58]. The CFG-based technique essentially follows the approach introduced for the CDG-based technique, but on CFGs rather than on Control Dependence Graphs (CDGs). Since CFG is a much simpler representation of the structure of a program, the CFG-based technique may be more efficient. However, the CFG lacks data dependence information, so the CFG-based technique may select test cases that are not capable of producing different outputs from the original programs` as explained above. The technique has been evaluated against various combinations of subject programs and test suites [57]. Ball improved the precision of the graph walk approach with respect to branch coverage [59]. One strength of the graph walk approach is its generic applicability. For example, it has been successfully used in black-box testing of re-usable classes [60]. In comparison to this work, the Box-Fusion approach can also identify gaps in coverage and assist in the detection of faults for pairwise testing.

6.3.4. Test Suite Minimization

Test suite minimization techniques aim to identify redundant test cases and to remove them from the test suite in order to reduce the size of the test suite. Whereas other minimization approaches primarily considered code-level structural coverage, Marre and Bertolino formulated test suite minimization as a problem of finding a spanning set over a graph [49]. They represented the structure of the SUT using a decision-to-decision graph (ddgraph). A ddgraph is a more compact form of the normal CFG since it omits any node that has one entering edge and one exiting edge, making it an ideal representation of the SUT for branch coverage. They also mapped the result of data-flow analysis onto the ddgraph for testing requirements such as def-use coverage. Once testing requirements are mapped to entities in the ddgraph, the test suite minimization problem can be reduced to the problem of finding the minimal spanning set. This technique focuses on defining a minimal test suite using a structure similar to a CFG. The Box-Fusion approach focuses on enhancing pairwise testing with respect to regression testing. Once a pairwise test suite has been used to test the system, it can be reused for regression testing by selecting the tests that are impacted by the modifications in the code.

Chapter 7: Conclusions

This chapter concludes the work presented in this dissertation. The first section will summarize the work described in the previous sections. The second section will discuss the intellectual merit and broader impact of the work. The last section will discuss future work for the Box-Fusion Approach.

7. 1 Summary of Work

Pairwise testing has emerged as an effective technique for testing programs with a large combination of inputs. However, due to pairwise testing being a black-box approach, the tester cannot always easily discern the location of faulty code or untested code, limiting the capability of pairwise testing. The dissertation presented the Box-Fusion approach, an enhancement of the pairwise testing technique by combining the capabilities of structural source code analysis with the ease of pairwise test input generation. The research questions that drove the research are:

RQ 1. How can pairwise testing be augmented to direct the developer to potential areas of faulty code, untested code, and related test failures?

RQ 2. How can pairwise test cases be identified to achieve the appropriate coverage required by regression testing?

By answering these questions, the outcome was the Box-Fusion approach, which has the following capabilities:

- Facilitates the identification of gaps in verification.
- Provides guidance on the location of defects in source code.
- Assists in the selection of test cases that can be used for regression testing.

To analyze the Box-Fusion approach and evaluate tools that can support the analysis, the dissertation used a case study of an algorithm that includes a large set of input combinations. The benefit of conducting a case study is that a deeper understanding of the capability of the subject of the investigation, that is the Box-Fusion approach, can be gained. The case study uses propositions

as a claim about the research questions, which provides a way to direct the evidence needed to support research claim. The propositions are as follows:

P1. The Box-Fusion approach can determine the areas of the code that require additional verification.

P2. The code defects, which are identified through one or more pairwise tests, are located in the execution path(s) identified by the Box-Fusion approach.

P3. Analyzing the Box-Fusion Control Flow Graphs (BF-CGF) using a suite of pairwise tests can narrow the potential sections of code where a defect is located.

P4. Regression tests can be optimized by matching the modified nodes of a BF-CFG to the tests that pass through the node.

To analyze these propositions, two different implementations of the Prospec algorithm are used by this case study. The Prospec algorithm takes as input parameters in terms of patterns, scopes and composite propositions, and can generate well over 31,000 possible combinations of input. The selection of the algorithm aligns well with the applicability of a pairwise testing approach. The first system, SimpleLTL Generator, implements a subset of the Prospec algorithm, that is it only takes as atomic propositions as input. The second system implements the complete Prospec algorithm, which takes composite propositions as input.

To analyze the approach, each of the propositions was applied to the implementations, data was gathered, and results and observations were made as summarized below:

Proposition 1

Result 1: The Box-Fusion approach can identify the gaps in the pairwise testing. This is complementary to gathering the branch coverage data, which the Box-Fusion approach also supports.

Observation 1: Using the tools to determine gaps in the coverage can also serve as a resource that can be used to support other verification techniques, such as walkthroughs and inspections. In addition, the result from the gap analysis can lead to a broader approach than using unit testing because the pairwise tests support system testing by its nature.

Proposition 2:

Result 1: The Box-Fusion approach does indeed capture the defect location within an execution path, directly supporting Proposition 2. The detected defects matched exactly what was seeded, and following the execution path strictly led into the location of the defects. For the second implementation, following the path provided by the *ExecutionPathFinder* tool yielded the defect for all the tests in that set.

Observations: For the SimpleLTL Generator, an interesting feature that was gleamed from this analysis is that the execution path assisted in the correction of the second seeded defect in an unexpected way. After it was determined that the test still failed, it was discovered that there was no need to start from the beginning of the execution path, but instead the developer could continue from where the defect was fixed. Another observation came from using the regression tool. Because the tool identifies the pairwise tests that touch each node, using the tool can give a measure of the impact that a modification might have in the system. Box-Fusion can provide a way to capture what tests could be impacted by a change in the code.

Proposition 3:

Observations: Failed tests coming from a pairwise test suite that exhibit the same defect can be used to narrow down the location of the common defect. This can be done by using the *CFGIntersectionMethodFinder* tool to determine what methods are common between a set of tests. Those intersecting methods represent the commonality of the tests, and if the defect is the same, then the defect must be located in the common code. The intersection of methods can also be used to identify common code among tests that have passed. This common code represents code that has a high probability of not being defective because many tests that have executed correctly over the code. This information can be used to lower those common methods in priority, leaving them as the least likely place to contain defective code. As more tests pass, the easier it can become to detect the defects in the code using this approach.

Proposition 4

Results: The regression exercise correctly identified necessary regression tests and no extra testing was needed, thus showing consistency with Proposition 4. As expected, the set of tests that did not contain the modification yielded the same results as the original test run.

Observations: When a defect correction is applied to the system, the regression testing done on that section of the SUT will identify the tests that are impacted. The tests that are identified at this point can be particularly useful in the following two ways: 1) If there is a large number of tests that have failed in the pairwise test suite, then the impact of this is that tests that were identified by the regression testing algorithm should be given a lower priority since they may have been fixed by the changes; and 2) If the majority of tests in the pairwise test are passing, then the impact of tests identified by regression testing warns of extra testing that must be done.

7. 2 Intellectual Merit and Broader Impacts

The **intellectual merit** of the research is that it will enhance the way that software developers verify software systems with large number of inputs. The Box-Fusion approach minimizes the discrepancy between the uses of different testing approaches, and ultimately allows developers to locate and correct defects with more ease. The work is particularly significant because it defines an approach that enhances pairwise testing by aiding in the identification of untested, faulty code. In addition, the work facilitates the use of pairwise testing as a regression testing technique, by identifying test cases that need to be rerun for regression based on the impact of code changes, which assists in assuring developers that they are retesting only what is needed that are needed and nothing more.

The **broader impacts** are that the results of this work can be expanded to other black-box techniques, and it can be adapted to provide an educational component on software verification. Students can be taught different techniques and asked to reflect on their experiences while applying each technique in a significantly complex software system. Students learn by analyzing and

gaining an appreciation for the subtleties in the differences between different verification approaches, the advantages and disadvantages of each approach, and the difficulty in detecting defects in source code. The Box-Fusion approach defined by this dissertation will facilitate student learning.

7. 3 Future Work

Future work includes further validation using controlled experimentation for the approach. Experiments that can evaluate the use of the approach among different levels of experts can be beneficial to further assess the approach. Another future work direction is the visualization of the execution path data that it gathers, providing mechanisms that abstract information to the developer in a way that can assist in the detection of defects or verification gaps.

In addition, as systems continue to grow and expand into more interconnected systems, such as cyber-physical systems, the need arises for software verification approaches that are more scalable and flexible. The future work of the Box-Fusion approach is in this area. Because the basis for its analysis is the BF-CGF, it can analyze multiple interconnected systems in the same way it would analyze stand-alone systems. To do this, further scaling and robustness needs to be incorporated into the instrumentation tool, expanding the range of programs that it can successfully instrument.

References

[1] Salamah, S., Ochoa, O., & Jacquez, Y. (2015, January). Using Pairwise Testing to Verify Automatically-Generated Formal Specifications. In *2015 IEEE 16th International Symposium on High Assurance Systems Engineering* (pp. 279-280). IEEE.

[2] National Institute of Standards and Technology (NIST), June 2002, http://www.nist.gov/director/planning/upload/report02-3.pdf accessed April 2015.

[3] Tassey, G. (2002). The economic impacts of inadequate infrastructure for software testing. National Institute of Standards and Technology, RTI Project, 7007(011).

[4] Zhivich, M., & Cunningham, R. K. (2009). The real cost of software errors.

[5] Bach, J., & Schroeder, P. J. (2004, October). Pairwise testing: A best practice that isn't. In Proceedings of 22nd Pacific Northwest Software Quality Conference (pp. 180-196).

[6] Zhu, H., Hall, P. A., & May, J. H. (1997). Software unit test coverage and adequacy. *Acm computing surveys (csur)*, *29*(4), 366-427.

[7] Radatz, J., Geraci, A., & Katki, F. (1990). IEEE standard glossary of software engineering terminology. *IEEE Std*, *610121990*(121990), 3.

[8] Leung, H. K., & White, L. (1990, November). A study of integration testing and software regression at the integration level. In *Software Maintenance, 1990, Proceedings., Conference on* (pp. 290-301). IEEE.

[9] Pfleeger, S. L. (2001). Les Hatton, and Charles C. Howell, Solid Software.

[10] Black, R. (2007). Pragmatic software testing: Becoming an effective and efficient test professional. John Wiley & Sons.

[11] Brown, N. (1999). High-Leverage Best Practices---What Hot Companies are Doing to Stay Ahead and How DoD Programs Can Benefit. Crosstalk, October.

[12] Melo, W., Shull, F., & Travassos, G. H. (2001). Software Review Guidelines. COPPE/UFRJ Systems Engineering and Computer Science Program Technical Report ES-556/01.

[13] Humphrey, W. S. (1989). Managing the software process. Addison-Wesley Longman Publishing Co., Inc.

[14] Salamah, S. I. (2007). Generating linear temporal logic formulas for complex pattern-based specifications. The University of Texas at El Paso.

[15] Kaufmann, M., & Moore, J. S. (1996, June). ACL2: An industrial strength version of Nqthm. In Computer Assurance, 1996. COMPASS'96, Systems Integrity. Software Safety. Process Security. Proceedings of the Eleventh Annual Conference on (pp. 23-34). IEEE.

[16] Rushby, J. (2001). Theorem proving for verification. In Modeling and verification of parallel processes (pp. 39-57). Springer Berlin Heidelberg.

[17] Clarke, E. M., & Wing, J. M. (1996). Formal methods: State of the art and future directions. ACM Computing Surveys (CSUR), 28(4), 626-643.

[18] Bjørner, N., Browne, A., Chang, E., Colón, M., Kapur, A., Manna, Z., & Uribe, T. E. (1996, July). STeP: Deductive-algorithmic verification of reactive and real-time systems. In International Conference on Computer Aided Verification (pp. 415-418). Springer Berlin Heidelberg.

[19] Delgado, N., Gates, A. Q., & Roach, S. (2004). A taxonomy and catalog of runtime software-fault monitoring tools. IEEE Transactions on software Engineering, 30(12), 859-872.

[20] Luckham, D., Sankar, S., & Takahashi, S. (1991). Two-dimensional pinpointing: Debugging with formal specifications. IEEE Software, 8(1), 74-84.

[21] Leucker, M., & Schallhart, C. (2009). A brief account of runtime verification. The Journal of Logic and Algebraic Programming, 78(5), 293-303.

[22] Cimatti, A., Clarke, E., Giunchiglia, F., & Roveri, M. (1999, July). NuSMV: A new symbolic model verifier. In International conference on computer aided verification (pp. 495-499). Springer Berlin Heidelberg.

[23] Holzmann, G. J. (1997). The model checker SPIN. IEEE Transactions on software engineering, 23(5), 279.

[24] Srinivasan, G. R., & Gluch, D. P. (1998). A Study of Practice Issues in Model-Based Verification Using the Symbolic Model Verifier (SMV) (No. CMU/SEI-98-TR-013). CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST.

[25] Clarke, E., Grumberg, O., and D. Peled. Model Checking. MIT Publishers, 1999

[26] Holzmann, G. J., \The SPIN Model Checker: Primer and Reference Manual." Addison-Wesley, 2003

[27] Model Checking at Carnegie Mellon, http://www-2.cs.cmu.edu/ model-check/index.html, May, 2007.

[28] Havelund, K., & Pressburger, T. (2000). Model checking java programs using java pathfinder. *International* Journal *on Software Tools for Technology Transfer*, *2*(4), 366-381.

[29] Stolz, V., & Bodden, E. (2006). Temporal assertions using AspectJ. *Electronic Notes in Theoretical Computer Science*, *144*(4), 109-124.

[30] Manna, Z., & Pnueli, A. (1991). Completing the temporal picture. *Theoretical Computer Science*, *83*(1), 97-130.

[31] Dwyer, M. B., Avrunin, G. S., & Corbett, J. C. (1999, May). Patterns in property specifications for finite-state verification. In *Software Engineering, 1999. Proceedings of the 1999 International Conference on* (pp. 411-420). IEEE.

[32] Specification Patterns System, http://patterns.projects.cis.ksu.edu accessed July 2016

[33] Mondragon, O., Gates, A. Q., & Roach, S. (2003). Prospec: Support for elicitation and formal specification of software properties. *Electronic Notes in Theoretical Computer Science*, *89*(2), 67-88.

[34] Salamah, S., Gates, A., & Kreinovich, V. (2012). Validated templates for specification of complex LTL formulas. *Journal of Systems and Software*, *85*(8), 1915-1929.

[35] Easterbrook, S., & Aranda, J. (2006). Case studies for software engineers. *ICSE'06*.

[36] Inductive. Pairwiser is Applicable to your system, https://inductive.no/pairwiser/overview/ accessed July 2016.

[37] Eclipse Marketplace. Control Flow Graph Factory, http://www.drgarbage.com/howto/cfgf-tutorial/ accessed July 2016.

[38] Dr. Garbage. Control Flow Graph Factory Tutorial, http://www.drgarbage.com/howto/cfgf-tutorial/ accessed July 2016

[39] Class Thread, https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html accessed July 2016

[40] OMG Unified Modeling Language TM (OMG UML), http://www.omg.org/spec/UML/2.5 accessed July 2016

[41] Kim, J., Choi, K., Hoffman, D. M., & Jung, G. (2007, October). White box pairwise test case generation. In *Seventh International Conference on Quality Software (QSIC 2007)* (pp. 286-291). IEEE.

[42] Open JDKWiki, https://wiki.openjdk.java.net/display/CodeTools/jcov accessed July 2016.

[43] Comparison of code coverage tools, https://confluence.atlassian.com/display/CLOVER/Comparison+of+code+coverage+tools accessed July 2016.

[44] Lingampally, R., Gupta, A., & Jalote, P. (2007, January). A multipurpose code coverage tool for java. *In System Sciences, 2007. HICSS 2007*. 40th Annual Hawaii International Conference on (pp. 261b-261b). IEEE.

[45] Shelke, S., & Nagpure, S. The Study of Various Code Coverage Tools. *International Journal of Computer Trends and Technology (IJCTT), 13*.

[46] JCover, http://www.mmsindia.com/JCover.html accessed July 2016.

[47] Yang, Q., Li, J. J., & Weiss, D. M. (2009). A survey of coverage-based testing tools. *The Computer Journal*, *52*(5), 589-597.

[48] Smith, B.D., Feather M.S., Muscettola N. (2000) Challenges and Methods in Testing the Remote Agent Planner. *Artificial Intelligence Planning Systems*.

[49] IEEE Standard Glossary of Software Engineering Terminology. IEEE Press, 10 Dec 1990

[50] Chen TY, Lau MF. Dividing strategies for the optimization of a test suite. Information Processing Letters 1996; 60(3):135–141.

[51] Jeffrey, D., & Gupta, N. (2005, September). Test suite reduction with selective redundancy. In *21st IEEE International Conference on Software Maintenance (ICSM'05)* (pp. 549-558). IEEE.

[52] Jeffrey, D., & Gupta, N. (2007). Improving fault detection capability by selectively retaining test cases during test suite reduction. *IEEE Transactions on software Engineering*, *33*(2), 108-123.

[53] Schroeder, P. J., & Korel, B. (2000). *Black-box test reduction using input-output analysis* (Vol. 25, No. 5, pp. 173-177). ACM.

[54] Yau, S. S., & Kishimoto, Z. (1987). METHOD FOR REVALIDATING MODIFIED PROGRAMS IN THE MAINTENANCE PHASE. In *IEEE*.

[55] IEEE Standard Glossary of Software Engineering Terminology. IEEE Press, 10 Dec 1990.

[56] Yoo, S., & Harman, M. (2012). Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification and Reliability*, *22*(2), 67-120.

[57] Rothermel, G., & Harrold, M. J. (1998). Empirical studies of a safe regression test selection technique. *IEEE Transactions on Software Engineering*, *24*(6), 401-419.

[58] Rothermel, G., & Harrold, M. J. (1997). A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, *6*(2), 173-210.

[59] Ball, T. (1998). On the limit of control flow analysis for regression test selection. *ACM SIGSOFT Software Engineering Notes*, *23*(2), 134-142.

[60] Martins, E., & Vieira, V. G. (2005, April). Regression test selection for testable classes. In *European Dependable Computing Conference* (pp. 453-470). Springer Berlin Heidelberg.

[61] Wong, W. E., Horgan, J. R., London, S., & Mathur, A. P. (1995, April). Effect of test set minimization on fault detection effectiveness. In *Proceedings of the 17th international conference on Software engineering* (pp. 41-50). ACM.

[62] Ochoa, O. (2016) Box-Fusion, GitHub Repository, http://github.com/omarochoa811/Box-Fusion

[63] Yilmaz, C., Cohen, M. B., & Porter, A. A. (2006). Covering arrays for efficient fault characterization in complex configuration spaces. *IEEE Transactions on Software Engineering*, *32*(1), 20-34.

Appendix A

This appendix provides the Formal Verification Tasks Lists for the eight implementations that were used when formally inspecting the tools implemented. Each entry includes the name of the tool, description of the algorithm and the assumptions that this program makes to operate correctly.

Instrumentor

Algorithm implementation that parses BF-CFGs and inserts instrumentation support code into a matching Java source file for Start edges, Exit edges, Edges (regular transitions, i.e. those that don't include Start or Exit) and Origin calls (instrumentation code to capture which method transferred control to a new method).

Assumptions: BF-CFG exist in the given directory. Source code exists in the given directory. Java source code adheres to coding standards (no else-if statements, scoping brackets for single statements, no multiple lines in one statement)

Table A. 1 Formal Verification Tasks List for the Instrumentor Implementation.

ID	Inspection Task
1	Does the **Instrumentor** program instrument all Java source files that exist in the given source file folder?
2	Does the **Instrumentor** program identify the four types of instrumentation mark statements?
3	Does the **Instrumentor** program create a *Start* instrumentation statement in the following format? *InstrumentingCode.mark(<DirectoryName>,<FileName>,<fromLine>,<toLine>);*
4	Does the **Instrumentor** program place the *Start* instrumentation statement in the toLine from the BF-CFG at the right side of the source code line?
5	Does the **Instrumentor** program create an *Exit* instrumentation statement in the following format? *InstrumentingCode.mark(<DirectoryName>,<FileName>,<fromLine>,<toLine>);*
6	Does the **Instrumentor** program place the *Exit* instrumentation statement in the toLine from the BF-CFG at the left side of the source code line but on the right side of any other instrumentation marks?

7	Does the **Instrumentor** program create an *Edge* instrumentation statement in the following format? *InstrumentingCode.mark(<DirectoryName>,<FileName>,<fromLine>,<toLine>);*
8	Does the **Instrumentor** program place the *Edge* instrumentation statement in the toLine from the BF-CFG at the left side of the source code line?
9	Does the **Instrumentor** program create an *Origin* statement in the following format? *InstrumentingCode.saveOrigin();*
10	Does the **Instrumentor** program place the *Origin* instrumentation statement in the beginning of the method before any other instrumentation marks?
11	Does the **Instrumentor** program save all changes to the Java source files?

InstrumentingCode

Algorithm used to populate the BF-CFGs with execution data as the system is pairwise tested. Each test will be associated with a unique ID, and that test ID will be inserted into the BF-CFG's edges that it visits. This InstrumetingCode is called when triggered by the instrumented system. Along with the test ID, the algorithm maintains and inserts a global counter to keep track of the ordering of transitions. In addition, the algorithm keeps track of the last visited line in order to correctly mark the edge transition that matches the previous node. A stack is used to maintain previous nodes across transfer of control between method calls.

Assumptions: BF-CFG exist in the given directory. Source code has been instrumented. System has been initialized with Test ID and BF-CFG locations.

Table A. 2 Formal Verification Tasks List for the InstrumentingCode Implementation.

ID	Inspection Task
1	Does the **mark** method store the mark statement data (*i.e., fileLocation, fileName, fromLine, toLine, globalCounter*) into the mark buffer?
2	Does the **mark** method store the toLine into the previousLine?
3	Does the **mark** method replace the fromLine with the previousLine when the fromLine does not match the previousLine?
4	Does the **mark** method replace the fromLine with the previousLine when the fromLine does not match the START?
5	Does the **mark** method pop the previousLine from the stack when the Exit mark is encountered?
6	Does the **mark** method increment the globalCounter?

7	Does the **saveOrigin** method store the origin mark statement data (i.e., *teamFolder, calleeClass, calleeMethod, callerClass, callerMethod, testID, globalCounter*) into the origin buffer?
8	Does the **markOrigin** method increment the globalCounter?
9	Does the **saveOrigin** method push the previous line into a stack?
10	Does the **saveToFiles** method store every mark data from the mark buffer into the BF-CFG? (*i.e., inspect iteration to check that every mark's data is passed to* ***markEdge***).
11	Does the **saveToFiles** method store every origin mark data from the origin buffer into the BF-CFG? (*i.e., inspect iteration to check that every origin's data is passed to* ***markOrigin****)*
12	Does the **saveToFiles** method reset origin and mark buffers, previousLine and globalCounter when done saving data?
13	Does the **markEdge** method read all the nodes that match the "edge" tag into an edgeList for the given BF-CFG?
14	Does the **markEdge** method search for the node that matches the fromLine and toLine with a node from edgeList?
15	Does the **markEdge** method store the testID and global counter in the matching node by inserting a "Val" node with a sub node named "ID" and values in the following format? *<TestID>.<globalCounter>*
16	Does the **markOrigin** method store the testID and global counter in the origin tag matching the given BF-CFG?
17	Does the **markOrigin** method store the caller class, caller method, testID and global counter in the matching node by inserting a "Val" node within the sub node named "Origin" and values in the following format? *<callerClassName>.<callerMethodName>:<TestID>@<globalCounter>*

CGF2BF

This algorithm reads the data from CFG generated by the CFGF tool, creates a new BF-CFG and inserts the data needed to capture instrumentation information. In addition it changes the names of nodes to more appropriately reflect their use, and corrects an issue found with flipped Boolean expressions. It also shortens up the names of the CFGs.

Assumptions: CFG exist in the given directory. Given directory ends in "-XML".

Table A. 3 Formal Verification Tasks List for the CGF2BF Implementation.

ID	Inspection Task
1	Does the **CGF2BF** program copy the given CFG directory to a new directory named the same but with the extension "-Archived" instead of "-XML"?

2	Does the **CGF2BF** program modify every CFGs that exist in the given CFG directory? (*i.e., inspect iteration to check that every CFG is modified)*
3	Does the **CGF2BF** program create a new BF-CFG XML document with a parent "CFG"?
4	Does the **CGF2BF** program read all the nodes that match the "node" tag into a nodeList?
5	Does the **CGF2BF** program create a new tag named "Node" under "CFG"?
6	Does the **CGF2BF** program read every node in nodeList and saves the values for "name" as "Name" under the "Node" tag in the new BF-CFG XML?
7	Does the **CFG2BF** program read every node in nodeList and saves the values for "label" as "Line" under the "Node" tag in the new BF-CFG XML?
8	Does the **CGF2BF** program read all the nodes that match the "edge" tag into an edgeList?
9	Does the **CGF2BF** program create a new tag named "Edge" under "CFG"?
10	Does the **CGF2BF** program read every node in edgeList and saves the values for "source" as "Source" under the "Edge" tag in the new BF-CFG XML?
11	Does the **CGF2BF** program read every node in edgeList and saves the values for "target" as "Target" under the "Edge" tag in the new BF-CFG XML?
12	Does the **CGF2BF** program read every node in edgeList, flips (*i.e., true->false, false->true*) and saves the values for "label" as "Boolean" under the "EDGE" tag in the new BF-CFG XML?
13	Does the **CGF2BF** program read every node in edgeList and insert a new tag of "ID" under the "Edge" tag in the new BF-CFG XML?
14	Does the **CGF2BF** program create a new tag named "Origin" under "CFG"?
15	Does the **CGF2BF** program simplify (*i.e., remove return type, simplify parameter types*) the BF-CFG file name?
16	Does the **CGF2BF** program save the created BF-CFG?

CoverageMiner

This tool is used to measure the branch edge coverage of a system after being pairwise tested. The tool traverses through all the BF-CFGs in a directory, reads them and counts the number of tested edges, and total edges. It calculates a percent for each BF-CFG visited, and for the total BF-CFGs read.

Assumption: BF-CFG exist in the given directory.

Table A. 4 Formal Verification Tasks List for the CoverageMiner Implementation.

ID	Inspection Task
1	Does the **CoverageMiner** program measure every BF-CFG for the given directory?
2	Does the **CoverageMiner** program read all the nodes that match the "Edge" tag into an edgeList?
3	Does the **CoverageMiner** program read all the nodes that match the "Val" tag under the "ID" tag into a nodeTestIDList?
4	Does the **CoverageMiner** identify edges in a BF-CFG and count them?
5	Does the **CoverageMiner** identify edges that contain a <TestID> and count them?
6	Does the **CoverageMiner** correctly calculate the percent of edges tested?
7	Does the **CoverageMiner** save the results?

VerificationGapMiner

This tool is used to detect the areas of the system that were not tested by the pairwise testing. The tool traverses through all the BF-CFGs in a directory, reads them and detects the edge transitions that do not have a test ID associated.

Assumption: BF-CFG exist in the given directory.

Table A. 5 Formal Verification Tasks List for the VerificationGapMiner Implementation.

ID	Inspection Task
1	Does the **VerificationGapMiner** program measure every BF-CFG for the given directory?
2	Does the **VerificationGapMiner** program read all the nodes that match the "Edge" tag into an edgeList?
3	Does the **VerificationGapMiner** program read all the nodes that match the "Node" tag into a nodeList?
4	Does the **VerificationGapMiner** program read all the nodes that match the "Val" tag into a nodeTestIDList?
5	Does the **VerificationGapMiner** identify nodes in nodeTestIDList that do not have testID data?
6	Does the **VerificationGapMiner** correctly map nodes to lineNumbers for nodes with no test data?
7	Does the **VerificationGapMiner** correctly display the untested lines in the following format? *<BF-CFG> GAP @ <lineFrom> -> <lineTo>*

ExecutionPathFinder

This tool is used to identify the path a Test ID marked over the BF-CFGs. The output is sorted according the globalCounter to represent the sequence that the system followed through.

Assumptions: BF-CFG exist in the given directory and are populated with Test IDs. A Test ID greater than 0 is given as input.

Table A. 6 Formal Verification Tasks List for the ExecutionPathFinder Implementation.

ID	Inspection Task
1	Does the **ExecutionPathFinder** program provide a way to enter a Test ID?
2	Does the **ExecutionPathFinder** program traverse every BF-CFG for the given directory?
3	Does the **ExecutionPathFinder** program read all the nodes that match the "Edge" tag into an edgeList?
4	Does the **ExecutionPathFinder** program read all the nodes that match the "Node" tag into a nodeList?
5	Does the **ExecutionPathFinder** program read all the nodes that match the "Val" tag under the "ID" tag into a nodeTestIDList?
6	Does the **ExecutionPathFinder** identify nodes in nodeTestIDList that match the given testID data?
7	Does the **ExecutionPathFinder** insert strings for the nodes found in order of globalCounter in the following format? *<fileName>" "Line:" <lineSource>" -> Line:" <lineTarget>*
8	Does the **ExecutionPathFinder** program read all the nodes that match the "Val" tag under the "Origin" tag into an originNodeList?
9	Does the **ExecutionPathFinder** identify nodes in the originNodeList that match the given testID data?
10	Does the **ExecutionPathFinder** insert strings for the nodes found in order of globalCounter strings in the following format? Class.Method -> BF-CFG *<callerClassName>.<callerMethodName>-><callerClassName>\<callerMethodName>*
11	Does the **ExecutionPathFinder** display the execution path following the globalCounter order?

CFGIntersectionFinder

This tool calculates the intersection of BF-CFGs between a set of test IDs.

Assumptions: BF-CFG exist in the given directory and are populated with Test IDs. A set of Test IDs greater than 0 is given as input.

Table A. 7 Formal Verification Tasks List for the CFGIntersectionFinder Implementation.

ID	Inspection Task
1	Does the **CFGIntersectionFinder** program provide a way to enter multiple test Test IDs?
2	Does the **CFGIntersectionFinder** program traverse every BF-CFG for the given directory?
3	Does the **CFGIntersectionFinder** program read all the nodes that match the "Val" tag under the "Origin" tag into an originNodeList?
4	Does the **CFGIntersectionFinder** identify nodes in the originNodeList that match the given testID data?
5	Does the **CFGIntersectionFinder** collect a list for each testID of the found nodes in the following format? *<callerClassName>.<callerMethodName>-><callerClassName>.<callerMethodName>*
6	Does the **CFGIntersectionFinder** identify the intersection of each testID list?

RegressionFinder

This tool is used to identify tests to be rerun for regression testing.

Assumptions: BF-CFG exist in the given directory and are populated with Test IDs. A line number greater than 0 is given as input. A method name is also given as input. The line number must belong to the method given.

Table A. 8 Formal Verification Tasks List for the RegressionFinder Implementation.

ID	Inspection Task
1	Does the **RegressionFinder** program provide a way to provide a method name and line number for the given directory?
2	Does the **RegressionFinder** program provide a way to disambiguate similar named method names for the given directory?
3	Does the **RegressionFinder** program read all the nodes that match the "Edge" tag into an edgeList?
4	Does the **RegressionFinder** program read all the nodes that match the "Node" tag into a nodeList?
5	Does the **RegressionFinder** program read all the nodes that match the "Val" tag under the "ID" tag into a nodeTestIDList?
6	Does the **RegressionFinder** identify every node in nodeTestIDList that match the given lineNumber?

7	Does the **RegressionFinder** correctly store testID that matches the lineNumber?
8	Does the **RegressionFinder** display the testIDs that are required for regression testing?

Appendix B

This appendix contains the pairwise tests generated by the Pairwiser online tool. Table B.1 shows the tests suite for the Prospec Algorithm implementation. Table B.2 shows the test suite for the SimpleLTL Generator. The integers under each P, Q, L and R map to one of the 8 CP types as follows: AtLeastOneC=1, AtLeastOneE=2, ParallelC=3, ParallelE=4, ConsecutiveC=6, ConsecutiveE=6, EventualC=7 and EventualE=8.

Table B. 1 Tests Generated, Including IDs and CPs for the Prospec Algorithm.

Tests \ Parameters	Pattern	Scope	P	Q	L	R
1	AbsenceP	AfterL	5	0	7	0
2	QPrecedesP	AfterLuntilR	6	8	5	5
3	QRespondsP	BeforeR	2	5	0	7
4	QPrecedesP	Global	3	3	0	0
5	QRespondsP	AfterLuntilR	2	6	7	5
6	ExistenceP	AfterLuntilR	2	0	7	6
7	ExistenceP	BetweenLandR	4	0	5	7
8	QStrictlyPrecedesP	Global	1	1	0	0
9	QRespondsP	BetweenLandR	2	7	8	3
10	QStrictlyPrecedesP	AfterLuntilR	3	4	4	6
11	QStrictlyPrecedesP	AfterL	7	3	6	0
12	QRespondsP	AfterLuntilR	7	5	6	8
13	ExistenceP	BetweenLandR	1	0	1	2
14	QPrecedesP	BeforeR	6	6	0	6
15	QStrictlyPrecedesP	BetweenLandR	4	7	3	6
16	QPrecedesP	BetweenLandR	1	1	5	6
17	QRespondsP	AfterLuntilR	5	3	2	7
18	ExistenceP	BeforeR	7	0	0	3
19	QStrictlyPrecedesP	BetweenLandR	7	2	5	4
20	QRespondsP	AfterLuntilR	1	7	4	5
21	QRespondsP	AfterLuntilR	2	8	6	6
22	QRespondsP	AfterL	7	1	8	0
23	QRespondsP	AfterLuntilR	5	1	7	5
24	QPrecedesP	AfterLuntilR	5	4	1	4
25	QStrictlyPrecedesP	AfterLuntilR	7	8	7	4
26	QStrictlyPrecedesP	BeforeR	8	6	0	3

27	QRespondsP	BeforeR	2	4	0	2
28	QPrecedesP	AfterLuntilR	1	3	8	3
29	AbsenceP	BeforeR	6	0	0	5
30	QPrecedesP	AfterLuntilR	3	7	4	1
31	AbsenceP	BetweenLandR	8	0	4	8
32	QRespondsP	BetweenLandR	3	8	3	5
33	ExistenceP	BetweenLandR	3	0	4	4
34	QPrecedesP	AfterL	8	2	5	0
35	QRespondsP	AfterLuntilR	6	2	3	2
36	QPrecedesP	BetweenLandR	1	7	7	8
37	QRespondsP	BetweenLandR	1	2	1	6
38	QStrictlyPrecedesP	AfterL	2	3	1	0
39	QPrecedesP	BetweenLandR	3	6	1	8
40	QRespondsP	BetweenLandR	4	4	6	3
41	QStrictlyPrecedesP	Global	6	7	0	0
42	QStrictlyPrecedesP	AfterLuntilR	5	4	8	6
43	QStrictlyPrecedesP	AfterLuntilR	2	1	2	2
44	QRespondsP	BetweenLandR	4	7	8	4
45	QStrictlyPrecedesP	BetweenLandR	4	4	7	5
46	ExistenceP	AfterL	3	0	6	0
47	QPrecedesP	BetweenLandR	8	6	2	2
48	QRespondsP	AfterL	5	6	2	0
49	QRespondsP	BetweenLandR	8	7	1	7
50	QPrecedesP	BetweenLandR	2	5	3	6
51	QPrecedesP	BetweenLandR	4	2	6	1
52	AbsenceP	AfterLuntilR	8	0	2	4
53	QRespondsP	BetweenLandR	5	8	6	2
54	QRespondsP	AfterLuntilR	5	7	2	1
55	QRespondsP	BetweenLandR	5	3	6	8
56	AbsenceP	BetweenLandR	5	0	4	3
57	QStrictlyPrecedesP	BetweenLandR	6	1	4	7
58	QPrecedesP	AfterLuntilR	6	6	6	7
59	QStrictlyPrecedesP	AfterLuntilR	2	5	5	1
60	AbsenceP	AfterLuntilR	1	0	3	7
61	QRespondsP	AfterLuntilR	1	5	3	4
62	AbsenceP	AfterLuntilR	3	0	7	1
63	QRespondsP	AfterLuntilR	3	5	2	5
64	QRespondsP	AfterLuntilR	4	3	3	2
65	QRespondsP	Global	7	6	0	0
66	QStrictlyPrecedesP	AfterL	6	5	4	0
67	ExistenceP	BeforeR	3	0	0	1

68	ExistenceP	Global	4	0	0	0
69	QPrecedesP	BetweenLandR	7	8	4	7
70	QStrictlyPrecedesP	AfterL	1	5	1	0
71	QRespondsP	AfterLuntilR	2	3	4	4
72	QRespondsP	BetweenLandR	4	1	5	3
73	QStrictlyPrecedesP	AfterLuntilR	7	7	3	2
74	QStrictlyPrecedesP	AfterLuntilR	6	4	2	8
75	QStrictlyPrecedesP	AfterLuntilR	1	8	2	8
76	QPrecedesP	AfterLuntilR	8	6	8	5
77	QStrictlyPrecedesP	BetweenLandR	2	2	7	3
78	ExistenceP	BetweenLandR	8	0	3	1
79	QStrictlyPrecedesP	BetweenLandR	4	6	4	1
80	QPrecedesP	AfterLuntilR	5	2	3	3
81	QStrictlyPrecedesP	Global	8	8	0	0
82	AbsenceP	AfterLuntilR	2	0	5	6
83	QPrecedesP	AfterLuntilR	4	2	2	8
84	QRespondsP	BeforeR	8	3	0	6
85	QPrecedesP	BetweenLandR	3	2	5	8
86	QStrictlyPrecedesP	BeforeR	4	7	0	4
87	QPrecedesP	BeforeR	5	1	0	4
88	ExistenceP	AfterLuntilR	5	0	2	7
89	QStrictlyPrecedesP	AfterL	4	6	3	0
90	QStrictlyPrecedesP	BetweenLandR	1	4	4	2
91	QStrictlyPrecedesP	AfterLuntilR	6	3	1	5
92	AbsenceP	BeforeR	4	0	0	8
93	QRespondsP	AfterL	8	7	6	0
94	QStrictlyPrecedesP	BetweenLandR	8	1	7	7
95	QStrictlyPrecedesP	BeforeR	7	2	0	5
96	QRespondsP	AfterLuntilR	6	4	5	3
97	QRespondsP	BetweenLandR	4	8	8	7
98	AbsenceP	AfterLuntilR	6	0	6	4
99	ExistenceP	BetweenLandR	1	0	6	8
100	QPrecedesP	AfterLuntilR	6	1	3	1
101	ExistenceP	BetweenLandR	6	0	8	8
102	QPrecedesP	BeforeR	1	2	0	7
103	QStrictlyPrecedesP	BetweenLandR	7	5	7	6
104	QStrictlyPrecedesP	BetweenLandR	4	8	1	3
105	QStrictlyPrecedesP	AfterLuntilR	5	6	5	4
106	QStrictlyPrecedesP	AfterLuntilR	3	3	7	3
107	QPrecedesP	BetweenLandR	3	2	8	2
108	QRespondsP	AfterLuntilR	5	5	8	3

109	QRespondsP	AfterLuntilR	8	5	1	1
110	QRespondsP	Global	7	4	0	0
111	AbsenceP	AfterLuntilR	6	0	7	2
112	QStrictlyPrecedesP	AfterLuntilR	2	1	1	8
113	AbsenceP	BetweenLandR	7	0	1	1
114	QPrecedesP	BetweenLandR	8	4	3	8
115	AbsenceP	AfterLuntilR	1	0	8	1
116	QPrecedesP	BeforeR	3	4	0	7
117	AbsenceP	AfterLuntilR	7	0	2	6
118	QRespondsP	BetweenLandR	3	1	2	3
119	QPrecedesP	AfterLuntilR	4	1	6	5
120	QRespondsP	AfterLuntilR	4	5	5	2
121	QRespondsP	BeforeR	8	8	0	1
122	QRespondsP	Global	1	6	0	0
123	QStrictlyPrecedesP	AfterLuntilR	5	3	5	1
124	AbsenceP	Global	2	0	0	0
125	QRespondsP	Global	5	2	0	0
126	QRespondsP	AfterL	4	8	6	0
127	QRespondsP	AfterL	2	4	8	0
128	ExistenceP	BetweenLandR	5	0	1	5
129	QRespondsP	Global	6	5	0	0
130	QStrictlyPrecedesP	BeforeR	5	4	0	1
131	QStrictlyPrecedesP	AfterLuntilR	4	7	5	3
132	QStrictlyPrecedesP	AfterLuntilR	3	2	4	7

Table B. 2 Tests Generated, Including IDs and CPs for Simple LTL Generator.

Tests \ Parameters	Pattern	Scope	P	Q	L	R
1	UniversalityP	AfterLuntilR	1	0	1	1
2	QStrictlyPrecedesP	Global	1	1	0	0
3	QPrecedesP	BeforeR	1	1	0	1
4	QPrecedesP	Global	1	1	0	0
5	ExistenceP	BetweenLandR	1	0	1	1
6	UniversalityP	BeforeR	1	0	0	1
7	QRespondsP	AfterL	1	1	1	0
8	QRespondsP	BeforeR	1	1	0	1
9	ExistenceP	BeforeR	1	0	0	1
10	QStrictlyPrecedesP	BeforeR	1	1	0	1

11	QPrecedesP	AfterLuntilR	1	1	1	1
12	UniversalityP	Global	1	0	0	0
13	AbsenceP	AfterL	1	0	1	0
14	QPrecedesP	AfterL	1	1	1	1
15	ExistenceP	Global	1	0	0	0
16	QRespondsP	BetweenLandR	1	1	1	1
17	AbsenceP	AfterLuntilR	1	0	1	1
18	ExistenceP	AfterLuntilR	1	0	1	1
19	AbsenceP	Global	1	0	0	0
20	QStrictlyPrecedesP	BetweenLandR	1	1	1	1
21	QRespondsP	AfterLuntilR	1	1	1	1
22	UniversalityP	BetweenLandR	1	0	1	1
23	QPrecedesP	BetweenLandR	1	1	1	1
24	QStrictlyPrecedesP	AfterL	1	1	1	1
25	QStrictlyPrecedesP	AfterLuntilR	1	1	1	1
26	UniversalityP	AfterL	1	0	1	0
27	QRespondsP	Global	1	1	0	0
28	AbsenceP	BeforeR	1	0	1	1
29	ExistenceP	AfterL	1	0	1	1
30	AbsenceP	BetweenLandR	1	0	1	1

Appendix C

This appendix shows the system measured, test suite used, number of edges touched by a test, total edges counted and the resulting branch coverage achieved for every BF-CFG generated and instrumented into the system. The full measurements are broken down by classes. Table C.1 refers to the coverage of Simple LTL Generator and Table C.2 refers to the coverage of Prospec Algorithm.

Table C. 1 Simple LTL Generator Coverage.

Class	Edges touched	Total edges	% of edges touched
Generator	0	7	0 %
SimpleLTL_Generator	205	221	92.76 %
Pattern - Absence	5	5	100 %
Pattern - Existence	5	5	100 %
Pattern - Pattern	3	30	10.00 %
Pattern - Precedence	9	13	69.23 %
Pattern - Response	9	13	69.23 %
Pattern - StrictPrecedence	9	13	69.23 %
Pattern - Universality	5	5	100 %
Property	12	63	19.04 %
Proposition - AtLeastOneC	0	7	0 %
Proposition - AtLeastOneE	0	7	0 %
Proposition - Atomic	0	11	0 %
Proposition - CompositePropositions	0	10	0 %
Proposition - ConsecutiveC	0	7	0 %
Proposition - ConsecutiveE	0	7	0 %
Proposition - EventualC	0	7	0 %
Proposition - EventualE	0	7	0 %
Proposition - Incomplete	0	4	0 %
Proposition - ParallelC	0	7	0 %
Proposition - ParallelE	0	7	0 %
Proposition - Proposition	9	60	15 %
Scope - AfterL	7	11	63.63 %
Scope - AfterLuntilR	11	19	57.89 %
Scope - BeforeR	7	11	63.63 %
Scope - BetweenLandR	11	19	57.89 %
Scope - Global	0	2	0 %
Scope - Scope	0	20	0 %
Total Covered	307	598	51.33 %

Table C. 2 Prospec Algorithm Coverage.

Class	Edges touched	Total edges	% of edges touched
CPGenerator	10	17	58.82 %
cps - AtLeastOneC	5	8	62.5 0 %
cps - AtLeastOneE	12	15	80.00 %
cps - AtLeastOneH	5	8	62.50 %
cps - CompositeProposition	24	35	68.57 %
cps - ConsecutiveC	14	19	73.68 %
cps - ConsecutiveE	23	27	85.18 %
cps - ConsecutiveH	13	21	61.90 %
cps - EventualC	16	24	66.66 %
cps - EventualE	21	28	75.00 %
cps - EventualH	18	24	75.00 %
cps - ParallelC	5	8	62.50 %
cps - ParallelE	12	15	80.00 %
cps - ParallelH	5	8	62.50 %
cps - ParallelInverse	5	8	62.50 %
Exceptions	0	21	0 %
Factories - AfterL	18	30	60.00 %
Factories - AfterLUntilR	48	72	66.66 %
Factories - BeforeR	40	72	55.55%
Factories - BetweenLAndR	46	72	63.88 %
Factories - CP	40	44	90.90 %
Factories - Global	20	30	66.66 %
Factories - LTLGenerator	27	37	72.97 %
Factories - Operator	8	12	66.66 %
Factories - Template	205	241	85.06 %
inputoutput	201	251	80.07 %
LTLCharacters	0	2	0 %
LTLGenerator	4	9	44.44 %
OperatorGenerator	5	10	50.00 %
Operators - AndL	23	29	79.31 %
Operators - AndLForBeforeR	27	35	77.14 %
Operators - AndMinusL	16	21	76.19 %
Operators - AndR	16	21	76.19 %
OutputCharacters	0	2	0 %
RunInputOutput	4	10	40.00 %
Templates - Afterl	62	67	92.53 %
Templates - beforeR	266	408	65.19%
Between	33	43	76.74 %
Templates - global	89	148	60.13 %
Templates - template	24	30	80.00 %
Total Covered	1410	1982	71.14%

Appendix D

This appendix shows one example of the output of the ExecutionPathFinder tool.

Test ID: 4
Step 1 Proposition(java.lang.String,Ljava.lang.String).xml Line:START -> Line:8
Step 3 Proposition(java.lang.String,Ljava.lang.String).xml Line:8 -> Line:9
Step 4 Proposition(java.lang.String,Ljava.lang.String).xml Line:9 -> Line:10
Step 5 Proposition(java.lang.String,Ljava.lang.String).xml Line:10 -> Line:11
Step 6 Proposition(java.lang.String,Ljava.lang.String).xml Line:11 -> Line:12
Step 7 Proposition(java.lang.String,Ljava.lang.String).xml Line:12 -> Line:EXIT
Step 9 Proposition(java.lang.String,Ljava.lang.String).xml Line:START -> Line:8
Step 11 Proposition(java.lang.String,Ljava.lang.String).xml Line:8 -> Line:9
Step 12 Proposition(java.lang.String,Ljava.lang.String).xml Line:9 -> Line:10
Step 13 Proposition(java.lang.String,Ljava.lang.String).xml Line:10 -> Line:11
Step 14 Proposition(java.lang.String,Ljava.lang.String).xml Line:11 -> Line:12
Step 15 Proposition(java.lang.String,Ljava.lang.String).xml Line:12 -> Line:EXIT
Step 17
Response(prospec.model.proposition.Proposition,Lprospec.model.proposition.Proposition).xml
Line:START -> Line:9
Step 18
Response(prospec.model.proposition.Proposition,Lprospec.model.proposition.Proposition).xml
Line:9 -> Line:10
Step 19
Response(prospec.model.proposition.Proposition,Lprospec.model.proposition.Proposition).xml
Line:10 -> Line:11
Step 20
Response(prospec.model.proposition.Proposition,Lprospec.model.proposition.Proposition).xml
Line:11 -> Line:12
Step 21
Response(prospec.model.proposition.Proposition,Lprospec.model.proposition.Proposition).xml
Line:12 -> Line:13
Step 22
Response(prospec.model.proposition.Proposition,Lprospec.model.proposition.Proposition).xml
Line:13 -> Line:EXIT
Step 23 Property(prospec.model.scope.Scope,Lprospec.model.pattern.Pattern).xml Line:START
-> Line:22
Step 25 Property(prospec.model.scope.Scope,Lprospec.model.pattern.Pattern).xml Line:22 ->
Line:23
Step 26 Property(prospec.model.scope.Scope,Lprospec.model.pattern.Pattern).xml Line:23 ->
Line:24
Step 27 Property(prospec.model.scope.Scope,Lprospec.model.pattern.Pattern).xml Line:24 ->
Line:25
Step 28 Property(prospec.model.scope.Scope,Lprospec.model.pattern.Pattern).xml Line:25 ->
Line:26

Step 29 Property(prospec.model.scope.Scope,Lprospec.model.pattern.Pattern).xml Line:26 -> Line:EXIT
Step 30 DriverLTL.main->\prospec\generator\SimpleLTL_Generator\getStringRepresentation(prospec.model.property.Property).xml
Step 31 getStringRepresentation(prospec.model.property.Property).xml Line:START -> Line:168
Step 32 getStringRepresentation(prospec.model.property.Property).xml Line:168 -> Line:169
Step 33 SimpleLTL_Generator.getStringRepresentation->\prospec\model\property\Property\getPattern().xml
Step 34 getPattern().xml Line:START -> Line:60
Step 35 getPattern().xml Line:60 -> Line:61
Step 36 getPattern().xml Line:61 -> Line:EXIT
Step 37 getStringRepresentation(prospec.model.property.Property).xml Line:169 -> Line:175
Step 38 SimpleLTL_Generator.getStringRepresentation->\prospec\model\property\Property\getPattern().xml
Step 39 getPattern().xml Line:START -> Line:60
Step 40 getPattern().xml Line:60 -> Line:61
Step 41 getPattern().xml Line:61 -> Line:EXIT
Step 42 getStringRepresentation(prospec.model.property.Property).xml Line:175 -> Line:181
Step 43 SimpleLTL_Generator.getStringRepresentation->\prospec\model\property\Property\getPattern().xml
Step 44 getPattern().xml Line:START -> Line:60
Step 45 getPattern().xml Line:60 -> Line:61
Step 46 getPattern().xml Line:61 -> Line:EXIT
Step 47 getStringRepresentation(prospec.model.property.Property).xml Line:181 -> Line:183
Step 48 getStringRepresentation(prospec.model.property.Property).xml Line:183 -> Line:EXIT
Step 49 SimpleLTL_Generator.getStringRepresentation->\prospec\generator\SimpleLTL_Generator\generateResponseScope(prospec.model.property.Property).xml
Step 50 generateResponseScope(prospec.model.property.Property).xml Line:START -> Line:292
Step 51 generateResponseScope(prospec.model.property.Property).xml Line:292 -> Line:293
Step 52 SimpleLTL_Generator.generateResponseScope->\prospec\model\property\Property\getScope().xml
Step 53 getScope().xml Line:START -> Line:52
Step 54 getScope().xml Line:52 -> Line:53
Step 55 getScope().xml Line:53 -> Line:EXIT
Step 56 generateResponseScope(prospec.model.property.Property).xml Line:293 -> Line:295
Step 57 generateResponseScope(prospec.model.property.Property).xml Line:295 -> Line:EXIT
Step 58 SimpleLTL_Generator.generateResponseScope->\prospec\generator\SimpleLTL_Generator\generateResponseGlobal(prospec.model.property.Property).xml
Step 59 generateResponseGlobal(prospec.model.property.Property).xml Line:START -> Line:162
Step 60 generateResponseGlobal(prospec.model.property.Property).xml Line:162 -> Line:164

Step 61 generateResponseGlobal(prospec.model.property.Property).xml Line:164 -> Line:EXIT
Step 62 SimpleLTL_Generator.generateResponseGlobal-
>\prospec\model\property\Property\getPattern().xml
Step 63 getPattern().xml Line:START -> Line:60
Step 64 getPattern().xml Line:60 -> Line:61
Step 65 getPattern().xml Line:61 -> Line:EXIT
Step 66 SimpleLTL_Generator.generateResponseGlobal-
>\prospec\model\pattern\Pattern\getP().xml
Step 67 getP().xml Line:START -> Line:57
Step 68 getP().xml Line:57 -> Line:58
Step 69 getP().xml Line:58 -> Line:EXIT
Step 70 SimpleLTL_Generator.generateResponseGlobal-
>\prospec\model\proposition\Proposition\getName().xml
Step 71 getName().xml Line:START -> Line:13
Step 72 getName().xml Line:13 -> Line:14
Step 73 getName().xml Line:14 -> Line:EXIT
Step 74 SimpleLTL_Generator.generateResponseGlobal-
>\prospec\model\property\Property\getPattern().xml
Step 75 getPattern().xml Line:START -> Line:60
Step 76 getPattern().xml Line:60 -> Line:61
Step 77 getPattern().xml Line:61 -> Line:EXIT
Step 78 SimpleLTL_Generator.generateResponseGlobal-
>\prospec\model\pattern\Response\getQ().xml
Step 79 getQ().xml Line:START -> Line:16
Step 80 getQ().xml Line:16 -> Line:17
Step 81 getQ().xml Line:17 -> Line:EXIT
Step 82 SimpleLTL_Generator.generateResponseGlobal-
>\prospec\model\proposition\Proposition\getName().xml
Step 83 getName().xml Line:START -> Line:13
Step 84 getName().xml Line:13 -> Line:14
Step 85 getName().xml Line:14 -> Line:EXIT

Vita

Omar Ochoa earned his Bachelors of Science in Computer Science from The University of Texas at El Paso in the fall of 2002 and joined the Ph.D. program in 2005 under the guidance of Dr. Ann Gates. Omar has over 8 years of experience working in industry with companies such as the Army Research Lab, IBM, HP, and currently with HPE. Within HPE, Omar acts a technical leader which includes responsibilities such as mentoring, leadership and client-facing operations. Since 2010, Omar has been a Certified Software Development Professional.

At The University of Texas at El Paso, Omar was a teaching assistant in the Computer Science Department, for the Software Engineering capstone course. Later, Omar became lecturer and taught the undergraduate course in Advanced Object-Oriented Programming and the graduate course in Software Construction.

As a researcher, Omar participated in interdisciplinary efforts with Geology at the Cyber-ShARE Center. The work focused in finding ways to fuse models of the earth coming from different sources. This work resulted in a Master's of Science Degree in Computer Science with Dr. Vladik Kreinovich as his advisor.

Omar has been the recipient of numerous honors and awards such as the Murchison Graduate Scholarship, Alliance for Graduate Education and Professoriate Fellowship, IBM Fellowship, and the Most Outstanding Ph.D. Student in Computer Science in 2016.

Contact Information: omar@miners.utep.edu

This thesis/dissertation was typed by Omar Ochoa.