GENERATING COST-EFFECTIVE CRITERIA-BASED TESTS
FROM BEHAVIORAL MODELS

by

Nan Li
A Dissertation
Submitted to the
Graduate Faculty
of
George Mason University
In Partial Fulfillment of
The Requirements for the Degree
of
Doctor of Philosophy
Information Technology

Committee:

_____ Dr. Jeff Offutt, Dissertation Director

_____ Dr. Paul Ammann, Committee Member

_____ Dr. Sam Malek, Committee Member

_____ Dr. Stephen Nash, Committee Member

_____ Dr. Fei Li, Committee Member

_____ Dr. Stephen Nash, Senior Associate Dean

_____ Dr. Kenneth S. Ball, Dean, Volgenau School of Engineering

Date: _____ Spring Semester 2014
George Mason University
Fairfax, VA

Generating Cost-effective Criteria-based Tests from Behavioral Models

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy at George Mason University

By

Nan Li
Master of Science
Fairleigh Dickinson University, 2008
Bachelor of Engineering
Beihang University, 2006

Director: Dr. Jeff Offutt, Professor
Department of Computer Science

Spring Semester 2014
George Mason University
Fairfax, VA

# Dedication

I dedicate this dissertation to my wife Yun, my parents Yukui Li and Dehua Lv, my sister Xianzhong Li, my parents in law Jin Guo and Xiaoxia Ma, my advisor Dr. Jeff Offutt and my dear friends.

# Acknowledgments

I would like to thank Dr. Jeff Offutt for advising me and giving me great help for conducting research. I want to thank Dr. Paul Ammann to help me set up the experiments and the experiment participants including Lin Deng, Upsorn Praphamontripong, and Mariman Mirzaei. I thank Dr. Fei Li for his help in proving NP-complete problems and reviewing algorithms. I also thank my other committee members Dr. Sam Malek and Dr. Stephen Nash for giving me suggestions for my proposal and dissertation. I thank Dr. Daniel Menascé, and Dr. Bo Zhang for giving me additional suggestions for writing dissertation.

# Table of Contents


Page

List of Tables . . . viii
List of Figures . . . ix
Abstract . . . xi
1 Introduction . . . 1
1.1 Model-based Testing . . . 1
1.2 The Problem . . . 3
1.2.1 Problem Description . . . 3
1.2.2 An Example . . . 7
1.2.3 The Minimum Cost Test Paths Problem (MCTP) . . . 9
1.2.4 The Mapping Problem . . . 11
1.2.5 The Test Oracle Problem . . . 13
1.3 Hypotheses and Approaches . . . 15
1.3.1 Approaches for Solving the Minimum Cost Test Path Problem . . . 15
1.3.2 Hypotheses and Approaches for Solving the Mapping Problem . . . 18
1.3.3 Hypotheses and Approaches for Solving the Test Oracle Problem . . . 19
2 Background and Related Work . . . 21
2.1 Model Selection . . . 22
2.2 Test Graph Coverage Criteria . . . 22
2.3 Mutation Analysis . . . 24
2.4 The Minimum Cost Test Paths Problem . . . 26
2.4.1 The Test Suite Minimization Problem . . . 26
2.4.2 The Shortest Superstring Problem . . . 27
2.4.3 The Greedy Set-Covering Algorithm . . . 28
2.4.4 The Matching-based Prefix Graph Algorithm . . . 29
2.5 The Mapping Problem . . . 30
2.6 The Test Oracle Problem . . . 33
3 Proof of the Hardness of MCTP . . . 38
4 Using Behavioral Models to Generate Tests . . . 48

4.1 Solutions to the Minimum Cost Test Paths Problem . . . . . 48
4.1.1 The Previous Breadth-first Search Solution . . . . . 49
4.1.2 A Set-covering Based Solution . . . . . 51
4.1.3 A Matching-based Prefix Graph Solution . . . . . 52
4.1.4 Splitting the Super-test Requirement into Test Paths . . . . . 53
4.2 The Structured Test Automation Language (STAL) . . . . . 54
4.2.1 Creating Mappings and Generating Test Values . . . . . 56
4.2.2 Graph Transformation and Test Path Generation . . . . . 61
4.2.3 Solving Constraints and Concrete Test Generation . . . . . 63
4.3 Test Oracle Strategies . . . . . 64
4.4 The Structured Test Automation Language Framework (STALE) . . . . . 67
5 Experiments . . . . . 74
5.1 The Experiments for the Minimum Cost Test Paths Problem . . . . . 74
5.1.1 Experimental Environment and Process . . . . . 75
5.1.2 Experimental Results . . . . . 76
5.1.3 Experimental Analysis . . . . . 82
5.1.4 Threats to Validity . . . . . 83
5.1.5 Recommendations . . . . . 83
5.2 The Experiments for the Mapping Problem . . . . . 84
5.2.1 Experimental Design . . . . . 85
5.2.2 Experimental Subjects . . . . . 87
5.2.3 Experimental Procedure . . . . . 88
5.2.4 Experimental Results . . . . . 89
5.2.5 Experimental Analysis . . . . . 92
5.2.6 Threats to Validity . . . . . 92
5.2.7 Discussion . . . . . 94
5.3 The Experiments for the Test Oracle Problem . . . . . 94
5.3.1 Experimental Design . . . . . 95
5.3.2 Experimental Subjects . . . . . 100
5.3.3 Experimental Procedure . . . . . 101
5.3.4 Experimental Results . . . . . 103
5.3.5 Discussion and Recommendations . . . . . 133
5.3.6 Threats to Validity . . . . . 135
6 Conclusions and Thoughts . . . . . 136
6.1 Conclusions . . . . . 136

6.2 Contributions and Recommendations . . . 138
6.3 Future Work . . . 141
A A Manual for the Structured Test Automation Language . . . 143
B A Complete Example using the Vending Machine System . . . 150
B.1 The Implementation of Vending Machine System . . . 150
B.2 Abstract Tests . . . 150
B.3 Mapping Creation . . . 154
B.4 Concrete Tests . . . 154
C An Experimental Guide for the Mapping Problem . . . 159
C.1 Goal of the Study . . . 159
C.2 The Mapping Problem . . . 160
C.3 The Structured Test Automation Language (STAL) . . . 160
C.4 Downloading and Installing the Structured Test Automation Language Framework (STALE) . . . 163
C.4.1 Downloading STALE . . . 163
C.4.2 Installing STALE . . . 164
C.5 Running STALE . . . 165
C.5.1 Starting STALE . . . 165
C.5.2 Creating A New Project . . . 166
C.5.3 Adding Mappings . . . 167
C.5.4 Generating Concrete Tests . . . 169
C.6 Experimental Procedure . . . 170
C.6.1 Diagram Transformation . . . 170
C.6.2 Test Generation . . . 172
Bibliography . . . 174

# List of Tables


| Table | | Page |
|---|---|---|
| 1.1 | Variants of the MCTP Problem and Their NP-completeness | 17 |
| 2.1 | A Comparison of Previous Test Oracle Research Papers | 37 |
| 4.1 | Which Elements Need Mappings? | 56 |
| 4.2 | Attributes of Element and Object Mappings | 58 |
| 5.1 | Paths and Nodes for Open Source Methods | 77 |
| 5.2 | Execution Time for Open Source Methods | 79 |
| 5.3 | Paths and Nodes for Modified Methods | 80 |
| 5.4 | Execution Time for Modified Methods | 81 |
| 5.5 | Steps in Automated and Manual Test Generation Processes | 86 |
| 5.6 | Questionnaire | 89 |
| 5.7 | Time for Automatic and Manual Test Generation | 91 |
| 5.8 | A Comparison of Test Oracle Strategies | 96 |
| 5.9 | Experimental Subjects | 102 |
| 5.10 | Numbers of Faults Found by Test Oracle Strategies - Part1 | 104 |
| 5.11 | Numbers of Faults Found by Test Oracle Strategies - Part2 | 105 |
| 5.12 | Effectiveness of Test Oracle Strategies - Part1 | 106 |
| 5.13 | Effectiveness of Test Oracle Strategies - Part2 | 107 |
| 5.14 | Cost of Test Oracle Strategies – Total Number of Distinct Assertions - Part1 | 125 |
| 5.15 | Cost of Test Oracle Strategies – Total Number of Distinct Assertions - Part2 | 126 |
| 5.16 | Cost of Test Oracle Strategies – Total Number of Assertions - Part1 | 127 |
| 5.17 | Cost of Test Oracle Strategies – Total Number of Assertions - Part2 | 128 |
| 5.18 | Cost-effectiveness of Test Oracle Strategies - Part1 | 131 |
| 5.19 | Cost-effectiveness of Test Oracle Strategies - Part2 | 132 |
| A.1 | Which Elements Need Mappings? | 143 |
| A.2 | Attributes of Element and Object Mappings | 145 |
| C.1 | Mappings from old notations to new notations | 164 |
| C.2 | Mappings from States of the UML Diagrams to Nodes of the General Graph | 172 |

# List of Figures


| Figure | | Page |
|---|---|---|
| 1.1 | A Generic Model-based Testing Process | 2 |
| 1.2 | Reachability-Infection-Propagation-Revealability Model | 5 |
| 1.3 | VendingMachine Class | 7 |
| 1.4 | FSM of Vending Machine | 8 |
| 1.5 | A JUnit Test for Class VendingMachine | 12 |
| 2.1 | A Superstring for Two Arbitrary Strings | 29 |
| 4.1 | A UML State Machine Diagram for the Class VendingMachine | 55 |
| 4.2 | Mappings | 71 |
| 4.3 | Precision Relationship among Test Oracle Strategies | 72 |
| 4.4 | The User Interface of the Structured Test Automation Language Framework | 73 |
| 5.1 | Relationship between *% Mapping* and *% Time* | 93 |
| 5.2 | Qqplot for *% Mapping* | 93 |
| 5.3 | Qqplot for *% Time* | 94 |
| 5.4 | Qqplot for NOS of edge coverage | 108 |
| 5.5 | Qqplot for SIOS of edge coverage | 109 |
| 5.6 | Qqplot for OS1 of edge coverage | 109 |
| 5.7 | Qqplot for OS2 of edge coverage | 110 |
| 5.8 | Qqplot for OS3 of edge coverage | 110 |
| 5.9 | Qqplot for OS4 of edge coverage | 111 |
| 5.10 | Qqplot for OS5 of edge coverage | 111 |
| 5.11 | Qqplot for OT1 of edge coverage | 112 |
| 5.12 | Qqplot for OT2 of edge coverage | 112 |
| 5.13 | Qqplot for OT3 of edge coverage | 113 |
| 5.14 | Qqplot for OT4 of edge coverage | 113 |
| 5.15 | Qqplot for OT5 of edge coverage | 114 |
| 5.16 | Qqplot for NOS of edge-pair coverage | 114 |
| 5.17 | Qqplot for SIOS of edge-pair coverage | 115 |

5.18 Qqplot for OS1 of edge-pair coverage . . . 116
5.19 Qqplot for OS2 of edge-pair coverage . . . 117
5.20 Qqplot for OS3 of edge-pair coverage . . . 118
5.21 Qqplot for OS4 of edge-pair coverage . . . 119
5.22 Qqplot for OS5 of edge-pair coverage . . . 120
5.23 Qqplot for OT1 of edge-pair coverage . . . 121
5.24 Qqplot for OT2 of edge-pair coverage . . . 122
5.25 Qqplot for OT3 of edge-pair coverage . . . 123
5.26 Qqplot for OT4 of edge-pair coverage . . . 123
5.27 Qqplot for OT5 of edge-pair coverage . . . 124
5.28 Averages of Cost-effectiveness of Edge-adequate Tests . . . 129
5.29 Averages of Cost-effectiveness of EdgePair-adequate Tests . . . 130
5.30 Averages of Cost-effectiveness of Edge-adequate Tests Below 100 . . . 130
5.31 Averages of Cost-effectiveness of EdgePair-adequate Tests Below 100 . . . 130
B.1 Implementation of VendingMachine Class - Part 1 . . . 155
B.2 Implementation of VendingMachine Class - Part 2 . . . 156
B.3 Mappings for Vending Machine Model - Part1 . . . 157
B.4 Mappings for Vending Machine Model - Part2 . . . 158
C.1 A Overview of STALE Folder Structure . . . 164
C.2 A Correct Java_Home Variable Example . . . 165
C.3 A Correct System Path Variable Example . . . 165
C.4 STALE Started with No Existing Projects . . . 166
C.5 Adding a Model to STALE . . . 167
C.6 A Model is Added to STALE . . . 168
C.7 STALE Folder Structure for the Vending Machine Project . . . 168
C.8 Adding a Mapping for Transition Initialize . . . 169
C.9 Adding a Mapping for Transition Coin . . . 169
C.10 Adding a Mapping for Constraint ConstrinatStockOne . . . 170
C.11 Adding a Mapping for Transition GetChocs . . . 171
C.12 Adding a Mapping for Object StringBuffer . . . 171
C.13 Import Declaration and Test Generation . . . 171
C.14 Time Spent for Test Generation . . . 172

# Abstract

GENERATING COST-EFFECTIVE CRITERIA-BASED TESTS FROM BEHAVIORAL MODELS

Nan Li, Ph.D.

George Mason University, 2014

Dissertation Director: Dr. Jeff Offutt

In software engineering, behavioral models such as finite state machines (FSMs) are used to represent key system behaviors. Coverage criteria are rules to impose test requirements on a test set. Model-based testing (MBT) generates abstract tests in terms of a model (e.g., paths in a FSM) to satisfy a coverage criterion. Then the abstract tests must be mapped to concrete tests, which are expressed in terms of the implementation of the model. The concrete tests should also include test oracles to decide if the tests pass or not.

This research addressed three major sub-problems in model-based testing: the minimum cost test paths problem (MCTP), the mapping problem, and the test oracle problem. MCTP is to find test paths that satisfy all test requirements with minimum cost, which had not been defined before. The mapping problem refers to how to map abstract tests to concrete tests. The previous techniques for solving the mapping problem required lots of supporting diagrams and complicated conformation among the diagrams, which is hard to use. Therefore, many practitioners often solve this mapping problem by hand. The test oracle problem considers how to write test oracles. Until now, there have not been many empirical results. With the same test inputs, test oracles that check more program states have the potential to reveal more faults, but also cost more to design and create.

This research presented two new approximation algorithms to solve the MCTP, the greedy set-covering algorithm and the matching-based prefix graph algorithm, and compared them with a previous breadth-first search based solution. This research designed a structured test automation language (STAL) to partially automate the mapping problem. This research also defined ten new test oracle strategies that check state invariants of models and different outputs and internal state variables of the program state different numbers of times.

The experiments showed that the two new solutions generated fewer test paths than the previous approach. The prefix-graph based solution took much less time than the other two solutions when the number of test requirements is large. Thus, the prefix graph based solution is recommended. The automated test generation method using STAL took a fraction of the time the manual method took, and the manual tests contained numerous errors in which concrete tests did not match their abstract tests. So testers should use STAL to automate the mapping problem. Checking state invariants of models is recommended for testers who do not have enough time due to its low cost-effectiveness; otherwise, testers should check state invariants, outputs and parameter objects. Finally, if testers write test oracles by hand, they should check program states once at the end of tests.

# Chapter 1: Introduction

This chapter first introduces the background of model-based testing, giving a general idea about the three major sub-problems addressed in this research: the minimum cost test paths problem (MCTP), the mapping problem, and the test oracle problem. Then section 1.2 shows the problem description and uses an example to illustrate the three sub-problems. Last, section 1.3 shows the approaches to solve the problems and the hypotheses.

## 1.1 Model-based Testing

Functional testing is an important part of software testing. This testing focuses on the functionality of a system and tries to ensure that every function is implemented correctly and matches system requirements. The system requirements have to be independent from the implementation and can be written in plain text, expressed in specification languages, or represented by models. Software designers often use models such as *Unified Modeling Language* (UML) diagrams to describe the requirement. In *model-based testing* (*MBT*), these models are used to design tests. Although testers can create models from source code for testing during the testing phase, the models may not reflect the requirements correctly because of the program faults. Therefore, this research used behavioral models that reflect functional aspects of the system from the design phase. To be specific, this research focused on UML behavioral diagrams because they are widely used in industry.

Figure 1.1 shows a general process to derive tests from models. Testers choose a coverage criterion to generate test requirements based on a model. Then *abstract tests* that are expressed in terms of the model are generated to satisfy the test requirements. An abstract test is defined using elements and objects from the model, thus cannot be executed on an implementation. For example, if the model is a finite state machine (FSM) and all-transition

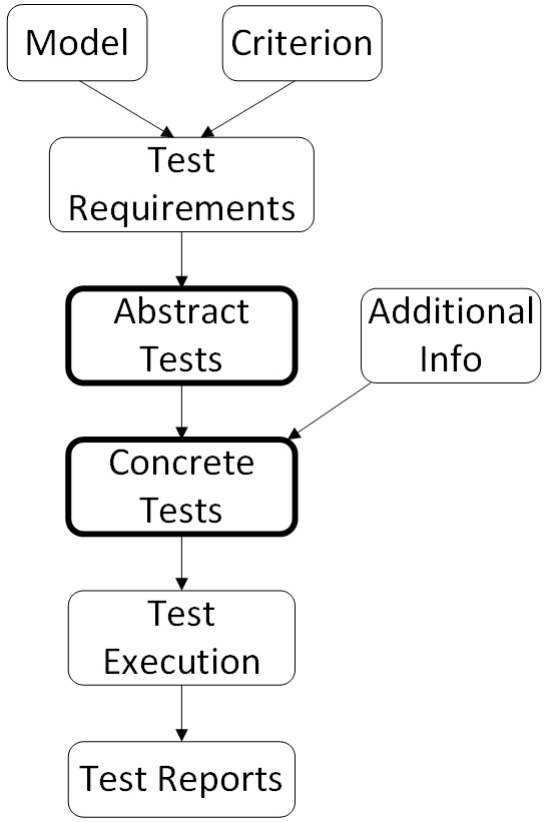


Figure 1.1: A Generic Model-based Testing Process

coverage [1] is chosen, test requirements are all transitions in this model. An abstract test defines a path through that machine. A set of transition-adequate tests should cover all the transitions.

Additional information, including abstract test values and data mappings, is needed to convert abstract tests to concrete tests. *Concrete tests* are expressed in terms of the implementation of the model, and are ready to be run automatically. For example, a *JUnit* test is concrete. In addition, in the concrete tests, *test oracles* that are usually specified in models are written to compare expected results with actual results.

The steps in bold rectangles in Figure 1.1 were addressed in this research to generate cost-effective tests from behavioral models in three aspects. First, graph coverage criteria typically define test requirements ($TR$) as sequences of nodes (a *subpath*) in the graph.

These are then turned into *abstract tests* by creating complete paths. A complete path is a path from an initial node to a final node, called a *test path* (*TP*). Each test path represents a complete execution of the software being modeled. Testers may come up with various sets of tests and each set of tests covers the test requirements with different numbers of tests or different total numbers of nodes in the tests.

Second, abstract tests have to be converted into concrete tests because they cannot be executed directly. However, in many situations, solutions that transform abstract tests to concrete tests automatically are not available. Thus, testers often currently map abstract tests to concrete tests by hand.

Third, concrete tests need to include test oracles. A *test oracle* determines whether a test passes. An example for a test oracle is an assertion in JUnit tests. When tests are executed, a fault may be triggered to produce an error state, which then propagates to be revealed as a failure that can be observed by checking program states (outputs and internal state variables). Testers are likely to observe more failures by checking more program states. But checking more program states requires testers to provide expected values manually in many situations, increasing cost [2].

## 1.2 The Problem

This research addressed how to generate cost-effective criteria-based tests from behavioral models. Section 1.2.1 gives the general problem description. This general problem consists of three major sub-problems: the minimum cost test paths problem (MCTP), the mapping problem, and the test oracle problem. The three problems are illustrated and analyzed in sections 1.2.3, 1.2.4, and 1.2.5 using an example described in section 1.2.2.

### 1.2.1 Problem Description

As stated in section 1.1, model-based testing first generates abstract tests from a model. When generating abstract tests to satisfy a graph coverage criterion, abstract tests (represented as test paths) can be constructed by creating one test path per test requirement,

or by satisfying multiple test requirements in the same path. It turns out that how test requirements are turned into test paths has a major impact on the overall cost of testing, both initially and as the software goes through maintenance. The cost of using abstract tests can be reduced if the number of tests or the total number of nodes (a node in a UML state machine diagram is a state) contained in the tests can be decreased. This problem was generalized as the *minimum cost test paths problem* (MCTP), which refers to the problem of generating abstract tests that cover all test requirements with the minimum cost.

The *mapping problem* refers to the problem of converting abstract test values to concrete test values. Researchers have developed solutions to automate the mapping problem from behavioral models but the previous solutions use lots of supporting diagrams that must be highly consistent, such as the parameters used in one diagram have to be recognized in another diagram. Such complicated requirements are usually not practical in many real-world projects, especially when agile processes are used. Therefore, testers often map abstract tests to concrete tests manually. If an abstract test consists of several events, testers have to write the corresponding code for each event by hand. If one basic event is used multiple times in different abstract tests, testers must write redundant code for the same event. This process is time-consuming, labor-intensive, and error-prone. For instance, to write code in a concrete test for an event "authentication for an account" in an abstract test, testers may have to set up the test environment such as a database connection and account creation, and also write test sequences and oracles. If this event is used in multiple abstract tests, testers have to do the same processes including the setup of test environment, test sequences, and test oracles in each concrete test.

This research also studied a complementary problem: the test oracle problem. Figure 1.2 illustrates how test oracles are used to reveal faults by observing the *program state*. This figure extends the fundamental "RIP" model-"Reachability-Infection-Propagation" [3–6] to the "RIPR" model-"Reachability," "Infection," "Propogation," and "Revealability." The RIP model was originally developed independently by Offutt [3,4] and Morell [5,6] but they used different terms. The current term of the RIP model is defined in Ammann and Offutt's

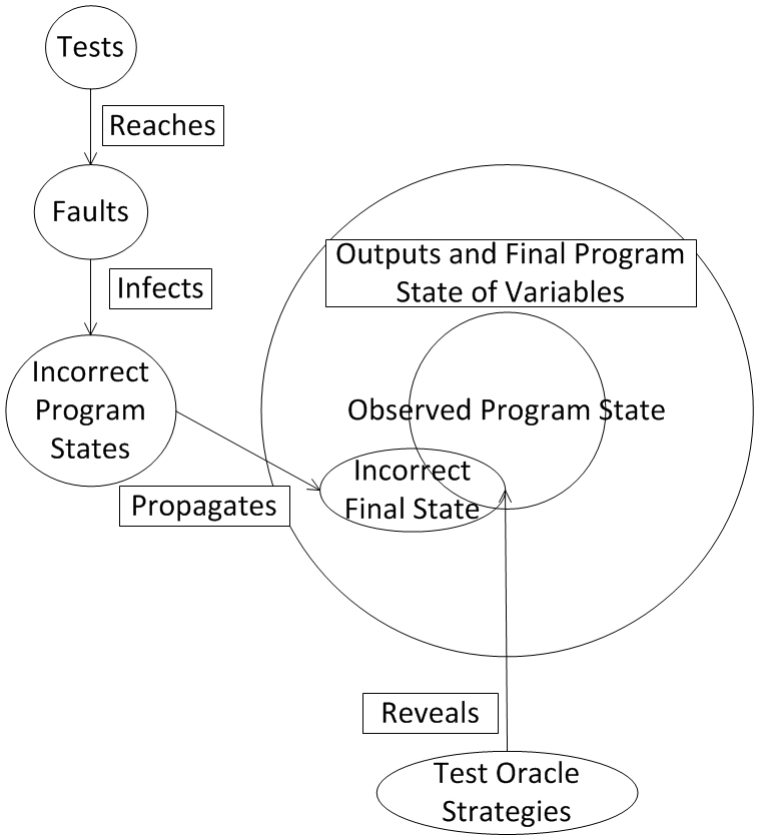


Figure 1.2: Reachability-Infection-Propagation-Revealability Model

book [7].

The program state represents various conditions in a system including the outputs and internal state variables and which and when statements are executed. To detect faults, tests have to *reach* the faults. That is, the statements that have the faults have to be executed. The execution of the faulty statements must cause an incorrect internal program state, such as a *for* loop is executed one few time than is needed. This is called *infecting* the program state is "infected." However, testers usually do not detect internal program states. The incorrect internal program state then must *propagate* to a failure (incorrect outputs or final program state). Then testers write test oracles (e.g., assertions) to monitor the final program state. The failures are revealed only when the incorrect outputs or final program state are observed, such as when actual outputs are found to be not equal to expected

outputs in an assertion. Briand et al. [8] suggested a *test oracle strategy* (abbreviated as OS) to specify which of the program state should be checked. In this research, a *test oracle strategy* is defined as a rule or a set of rules to specify which program states to check and how frequently to check the program states. A common belief is the more program states are checked, the more faults an OS is likely to reveal [8–11].

Briand et al. defined the *precision* of a test oracle strategy as the degree to which internal state variables and outputs are checked by this OS [8]. In this research, the *precision* of a test oracle strategy is refined as how many output values and internal state variables are checked by this OS. The fewer internal state variables and outputs a test oracle strategy checks, the less precise this test oracle strategy is. A more precise test oracle is more expensive, but it can let testers observe more failures. When testing beyond the unit level, testers must decide what parts of the program state to evaluate to determine if the test passes or fails since there are too many variables to check in the system. At the system level, this may simply be outputs to a screen, a database, or messages sent to other systems. This can be such a complicated problem that some practitioners simply take the cheapest possible solution: does the program terminate or not? This is called the *null test oracle* strategy (NOS) [9] and is used in industry. Integration testing may need to evaluate intermediate states from hundreds of objects spread throughout the program, as well as data sent to external destinations (screens, files, databases, sensors, etc.). Therefore, it is usually impractical to measure all states at the level of integration and system testing and testers do not know which states should be evaluated and when to check the states.

**Problem Statement:**

**Currently, when generating automated criteria-based concrete tests from behavioral models at the integration and system level, there are three sub-problems. The first problem is how to generate abstract tests to cover test requirements with the minimum cost. The second problem is that there is a lack of test automation techniques to map abstract tests to concrete tests from only behavioral models because many existing solutions are not applicable in**

**reality. The third problem is that there are too many program states to check and no guidelines for test oracles have been established for testers.**

### 1.2.2 An Example

This section shows a simple example below, which will be used in sections 1.2.3, 1.2.4 and 1.2.5 to illustrate the minimum cost test paths problem, the mapping problem and the test oracle program. The example program simulates the behavior of a vending machine. Some assumptions for this vending machine simplify the program: only chocolates are available for sale; all chocolates cost 90 cents; only dimes, quarters, or dollars are accepted; and the vending machine can contain infinite chocolates. The method specifications for the vending machine class are shown in Figure 1.3.

```
public class VendingMachine
{
    private int credit; // Current credit in the machine
    private LinkedList stock; // Used to store all chocolates

    // Constructor: vending machine starts empty.
    VendingMachine() {}

    // A coin is given to the vendingMachine.
    // Must be a dime (10), quarter (25), or dollar (100).
    public void coin (int coin) {}

    // User asks for a chocolate. Returns the change
    // and sets the parameter StringBuffer variable Choc.
    public int getChoc (StringBuffer choc) {}

    // Adds one new piece of chocolate to the machine.
    public void addChoc (String choc) {}

    // Get the value of the current credit
    public int getCredit () {}

    // Get the number of chocolates
    public int getStock () {}
}
```

Figure 1.3: VendingMachine Class

A simplified finite-state machine (FSM), which is derived from the program in Figure

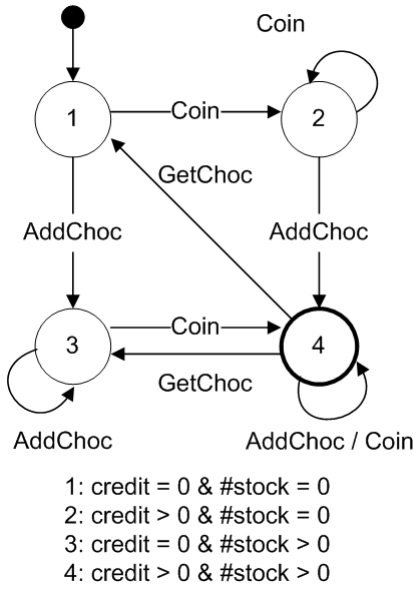


Figure 1.4: FSM of Vending Machine

1.3, is shown in Figure 1.4. Nine transitions and three events are included in this FSM. Three events are "AddChoc," "Coin," and "GetChoc" and an arrow represents a transition when an event is triggered. The FSM consists of four states: 1, 2, 3, and 4. State 1 is the initial state, where the credit is 0 and the number of chocolates is 0. If a customer adds coins, the FSM transitions to state 2, where the credit is greater than 0 but the number of chocolates is still 0. If a service person comes and puts chocolates in the vending machine, the FSM transitions to state 4 (the final state), where the credit is greater than 0 and the number of chocolates is greater than 0. State 4 returns to itself if coins or chocolates are added to the vending machine, and state 4 can transition to state 1 or 3 if any chocolate is taken from the machine. Similarly, when adding chocolates, state 1 can transition to state 3, where the credit is 0 and the number of the chocolates is greater than 0; state 3 returns to itself when adding chocolates; state 3 transitions to state 4 when customers put coins in.

Based on the FSM, testers may generate different tests by applying various coverage criteria. If a tester uses the prime path coverage criterion (a graph coverage criterion that subsumes edge coverage) [7], test paths need to cover 14 prime paths, computed by a graph

coverage web application (support software provided as part of Ammann and Offutt's testing book [12]):

1. $[1, 3, 4, 1]$
2. $[2, 4, 1, 2]$
3. $[1, 2, 4, 1]$
4. $[1, 2, 4, 3]$
5. $[2, 4, 1, 3]$
6. $[4, 1, 2, 4]$
7. $[4, 1, 3, 4]$
8. $[3, 4, 1, 3]$
9. $[3, 4, 1, 2]$
10. $[3, 4, 3]$
11. $[4, 3, 4]$
12. $[3, 3]$
13. $[2, 2]$
14. $[4, 4]$

The next three sections will analyze the three problems using this example.

### 1.2.3 The Minimum Cost Test Paths Problem (MCTP)

How test requirements are turned into test paths could affect the number of tests, the number of total nodes in the tests, etc. For instance, a previous breadth first search-based solution for prime path coverage (algorithm 1 of the graph coverage web application) generates nine test paths to cover 14 prime paths for the VendingMachine example:

1. $[1, 3, 4, 1, 2, 4]$
2. $[1, 2, 4, 1, 2, 4]$
3. $[1, 2, 4, 3, 4]$
4. $[1, 2, 4, 1, 3, 4]$
5. $[1, 3, 4, 1, 3, 4]$
6. $[1, 3, 4, 3, 4]$

7. $[1, 2, 2, 4]$
8. $[1, 3, 3, 4]$
9. $[1, 2, 4, 4]$

However, the same 14 prime paths can be covered by one test path, generated by the prefix graph-based solution developed by this research. The solutions to the MCTP problem will be discussed in section 4.1.

1. $[1, 2, 2, 4, 1, 2, 4, 3, 3, 4, 1, 3, 4, 1, 2, 4, 1, 3, 4, 1, 2, 4, 4, 4, 3, 4, 3, 4]$

The first solution has a total number of 46 nodes in the test paths while the second solution has only 28 nodes. Both of the solutions achieve the same effectiveness in terms of prime path coverage (all the prime paths are covered). But the cost of the second solution is less than than the first with respect to both the number of tests and the total number of nodes in the tests.

Formally, the minimum cost test paths problem is defined below:

**Definition 1** (**Minimum Cost Test Paths Problem (MCTP)**). *Consider a set of test requirements* $TR = \{r_1,\ r_2,\ \cdots,\ r_n\}$. *Each test requirement is presented as a subpath in a graph* $G = (V,\ E)$. *The problem MCTP is to find a set of test paths* $TP = \{t_1, t_2,\ \ldots,\ t_k\}$ *that cover all the test requirements in the graph* $G$ *such that the cost of using the test paths is minimum.*

As suggested above in the example, it turns out that the cost of testing is a multi-faceted problem. In fact, the cost can be reduced in several ways. Following are four possible goals for reducing cost, some of which are complementary and some of which conflict.

1. **Fewer total nodes**. Each TP contains a sequence of nodes in the graph and every execution of every node represents a cost. More nodes in the set of TPs means more values that have to be found, more time executing the tests, and more time evaluating the results of the tests.

2. **Fewer test paths**. Every TP is an abstract test that will be realized into a concrete test, and every test has an associated cost. This includes the initial cost of finding values for the test and a continuing cost of running the test and modifying the test as the software evolves.

3. **Fewer test requirements per test path** (smaller TR to TP ratio). Some TRs are infeasible, but which is often not known until abstract tests are realized into concrete tests. If one TP satisfies several TRs, and one of those TRs is infeasible, the entire TP is lost.

4. **Shorter test paths**. Finding values for TPs becomes more difficult as the TPs get longer. Tests from long TPs are also long, which makes them harder to maintain as the software evolves.

This research defined the MCTP problem for the first time, thus, no one has previously developed efficient solutions or proved there exists optimal polynomial time solutions (MCTP is not NP-complete). The MCTP definition above could be refined to explicitly satisfy any of these goals, resulting different variants of the MCTP problem. Section 1.3.1 will analyze the goals of MCTP and discuss the approaches to solve the MCTP problem.

### 1.2.4 The Mapping Problem

As stated previously, the mapping problem refers to the problem of transforming abstract tests to concrete tests. The mapping problem will be illustrated using the VendingMachine example. Based on Figure 1.4, nine test paths are generated to cover the same 14 prime paths in the last section.

1. $[1, 3, 4, 1, 2, 4]$
2. $[1, 2, 4, 1, 2, 4]$
3. $[1, 2, 4, 3, 4]$
4. $[1, 2, 4, 1, 3, 4]$
5. $[1, 3, 4, 1, 3, 4]$

6. $[1, 3, 4, 3, 4]$
7. $[1, 2, 2, 4]$
8. $[1, 3, 3, 4]$
9. $[1, 2, 4, 4]$

Since each transition is triggered by only one event in the vending machine example, I can use the event names to represent the transitions. The transitions for the first path $[1, 3, 4, 1, 2, 4]$ are: "AddChoc," "Coin," "GetChoc," "Coin," and "AddChoc". Thus, the tester may translate the abstract test to a concrete JUnit test in Java:

```
public class VendingMachineTest
{
    private VendingMachine vm;
    @Before
    public void setUp() throws Exception {
        vm= new VendingMachine();
    }
    @After
    public void tearDown() throws Exception {
        vm= null;
    }
    @Test
    public void testFirstTestPath() {
     1: vm.addChoc("MM");
     2: assertEquals(1,vm.getStock().size());
     3: vm.coin(100);
     4: assertEquals(95,vm.getCredit());
     5: StringBuffer choc = new StringBuffer().append("MM");
     6: vm.getChoc(choc);
     7: assertEquals(0,vm.getStock().size());
     8: vm.coin(10);
     9: vm.coin(25);
    10: vm.coin(25);
    11: vm.coin(25);
    12: vm.coin(25);
    13: vm.addChoc("MM");
    14: vm.addChoc("MM");
    15: assertEquals(100,vm.getCredit());
    16: assertEquals(2,vm.getStock().size());
    }
}
```

Figure 1.5: A JUnit Test for Class VendingMachine

In Figure 1.5, the code in testFirstTestPath() is written to map the abstract test: "AddChoc," "Coin," "GetChoc," "Coin," and "AddChoc." To be specific, "AddChoc" is mapped to line 1; "Coin" is mapped to line 3; "GetChoc" is mapped to lines 5-6; the second "Coin" process is mapped to lines 8-12; and the second "AddChoc" process is mapped to lines 13-14.

For the second test path $[1, 2, 4, 1, 2, 4]$, the transitions are "Coin," "AddChoc," "GetChoc," "Coin," and "AddChoc." The concrete test code that corresponds to "Coin" and "AddChoc" should be very similar to the code from lines 8-14 in testFirstTestPath() above, and the concrete test code that corresponds to "GetChoc" may be the same as the code from lines 5-6. The nine abstract tests have 15 "AddChoc" transitions, 16 "Coin" transitions, and six "getChoc" transitions. Each transition corresponds to at least one method call. So for transitions "AddChoc," "Coin," and "GetChoc," the same or similar code will be written 15 times, 16 times, and 6 times respectively. If the mappings between transitions and concrete code are established, testers will not need to write so much redundant code.

The previous solutions to automate the mapping problem used lots of additional diagrams and required high consistency among these diagrams, which is often impossible for practical use. This research addressed the problem of how to automate the mapping problem without additional diagrams. Section 1.3.2 will discuss the approaches and hypotheses for solving the mapping problem.

### 1.2.5 The Test Oracle Problem

Test oracles have to be included in concrete tests to determine whether the tests pass or fail, for example, with *assertEquals()* statements. When it comes to the test oracle problem, how often to check the states and which internal state variables and outputs to check have to be considered. A test oracle strategy (OS) is more costly if more variables are checked or the variables are checked more frequently. States can be checked after each method call or each transition (more than one method call can be included in a transition) or testers can just check the states once in one test or test suite.

Moreover, which variables should testers check? Should the member variables of all objects be checked? Should the return values of all method calls be checked? The more internal state variables are checked by a test oracle, the more *precise* this test oracle is.

In the example of the vending machine, if testers check the two member variables of the class "VendingMachine" after each method call (there are totally 37 transitions in nine tests and at least one method call is used for one transition), the variables will be checked at least 2 * 37 times. Internal state variables will be checked more times for bigger programs. The goal of this research was to establish OSes that are effective in revealing faults and also cost less in terms of the number of internal state variables and outputs to check and the times the program states are checked.

In this research, tests were generated from models (UML state machine diagrams). If all-transition coverage requires all transitions in a UML state machine diagram to be covered, an abstract test may look like: "transition 1, state invariant 1, ..., transition n, state invariant n." These abstract tests need to be converted into concrete tests. Properties of models such as state invariants in a state machine diagram can be used for OSes. If test oracle data, including expected test values, is very well specified by some specification language and additional information used to transform abstract test oracle data to executable code has been provided, the concrete test oracles can be generated automatically. Such test oracles are called *specified test oracles* [2] because the specification of a system is used as a source to generate test oracles, including expected values.

As pointed out by Harman et al. [2], automated test oracles are not available in many situations. For model-based testing, transformation from abstract tests to concrete tests requires that a model has to be very well specified using additional information. The information may include specification languages such as the object constraint language (OCL) [13] and other additional diagrams and mapping tables to map abstract information to executable code. Moreover, the notions of test inputs and oracles among different diagrams have to be highly consistent. Such complicated requirements are usually not easy to realize in practice. Thus, most practitioners cannot use automated test input and oracle generation

[14]. This is particularly true when agile processes are used. Therefore, most testers have to provide expected values manually for test oracles.

The test oracle problem must address observability. *Observability* is how easy it is to observe a program's internal state variables and outputs [15]. If a test oracle checks more program states, the observability of the program states is increased, and more faults may be revealed. However, writing test oracles can be costly because testers usually must provide expected values manually. Model-based testing can use state invariants from state machine diagrams as test oracles. This research started with that premise and asked the following questions. Is checking only the state invariants good enough? Should testers also check class variables? Should testers check outputs and internal state variables multiple times or only once in tests? What is the cost of checking more of the program state? Section 1.3.3 will discuss the approaches and hypotheses for solving the test oracle problem.

## 1.3 Hypotheses and Approaches

This section presents the approaches and hypotheses for solving the minimum cost test paths problem (MCTP), the mapping problem, and the test oracle problem.

### 1.3.1 Approaches for Solving the Minimum Cost Test Path Problem

Recall that a test requirement (TR) is a subpath (sequences of nodes) in a graph. A test path (TP) is a path from an initial node to a final node. To satisfy a coverage criterion, testers need to generate test paths to cover test requirements. Section 1.2.3 lists four goals to address the minimum cost test path problem: fewer total nodes, fewer test paths, fewer test requirements per test path (TR / TP), and shorter test paths.

Satisfying multiple goals is hard enough, but some of these conflict. The most obvious way to reduce the TR to TP ratio is to have exactly one TP for each TR. However, that conflicts with the **fewer test paths** goal. The **shorter test paths** goal and the **smaller TR to TP ratio** goal are complementary because reducing the TR to TP ratio will result in shorter test paths. The **fewer total nodes** goal has no tradeoffs with the other goals

above, thus is always valid. In practice, the TP length depends on the context, the source of the graph, and the overall difficulty of finding values for the concrete test.

If testers cannot find test inputs to tour a subpath of a test path $p$, this subpath is infeasible, making the whole test path infeasible. Therefore, testers need to regenerate new test paths to cover the test requirements covered by $p$. This problem is usually referred to the *infeasible path problem* [16]. This problem becomes more complex when the TR to TP ratio gets large because if a test path is infeasible, testers need to cover a large number of test requirements toured by this test path. This research did not focus on the infeasible path problem and assumed all test requirements are feasible. Solving the infeasible path problem will be considered in future work.

To balance the other goals above, this research took the following approach: find TPs to satisfy all TRs with the fewest overall number of nodes, by minimizing the maximum TR to TP ratio such that the number of test paths is not "too big," with the fewest test paths such that the maximum TR to TP ratio is not "too big," and with the fewest paths such that the maximum TR to TP ratio is not "too big." Of course, "too big" seems somewhat arbitrary but it depends on the context and the source of the graph, just like the TP length. Specifically five goals are optimized, each of which is a variant of the MCTP.

1. The total number of nodes
2. The total number of test paths
3. The maximum ratio of TR to TP subject to a bounded number of test paths
4. The total number of test paths subject to a bounded ratio of TR to TP
5. The total number of nodes subject to a bounded ratio of TR to TP

Variant (3) maximizes the ratio of TR to TP while generating no more than $m$ (the bounded number) test paths, where $m$ is a positive integer threshold. The bounded ratios in variants (4) and (5) have similar meanings as the bounded number in variant (3). Ideally, I could achieve all five goals with one algorithm. However, it is very hard to design an

Table 1.1: Variants of the MCTP Problem and Their NP-completeness

| **Variants of the MCTP Problem** | **NP-completeness** | **Reduction / Solution** |
|---|---|---|
| 1 | NP-complete | Bin-Packing [17, 18] |
| 2 | P | A modified algorithm of the one solving $CP_1^1$ [19] |
| 3 | NP-complete | 3-Partition [20] |
| 4 | NP-complete | 3-Partition |
| 5 | NP-complete | Bin-Packing |

algorithm to satisfy more than two optimization goals. Table 1.1 shows whether each variant of the MCTP problem are NP-complete and the reduction approaches. The proofs will be shown in chapter 3.

Since variant (2) is polynomial-time solvable, this research focused on approximation algorithms that can solve the other NP-complete variants. The new algorithms proposed in this research form the solid ground to solve variants (1), (3), (4), and (5) in Table 1.1. To solve a specific variant, I can adapt our solutions slightly. For instance, if I want to get the fewest number of nodes in the test paths, after I get the test paths using our algorithms introduced in Section 4.1, I can use dynamic programming to approximate the minimum number of nodes. If I want to ensure that the maximum TR to TP ratio is below a predefined threshold, then I can use a number to restrict the length of each test path generated in the algorithms. The algorithms developed in this research provide a basis to solve the general problem MCTP.

The solutions that solve the shortest superstring problem [17] were developed to solve MCTP. The new algorithms will be shown in section 4.1. Section 5.1 will present the experiments that compared the new algorithms with the previous breadth-first search based solution in terms of the number of test paths and the total number of nodes in the test paths. Note this research did not emphasize time. Time depends on many factors, including the people, tools, test criteria, and software under test. Instead, this research focused on stable, long term costs such as the number of test paths. For completeness, the experiments

in section 5.1 did measure time explicitly.

Since the new algorithms still generate test paths to cover all test requirements with fewer number of test paths or nodes, the effectiveness (in terms of coverage) remains the same as the previous breadth-first search based solution but the cost of the test paths is reduced. The new algorithms are intended to be more cost-effective than the previous solution.

### 1.3.2 Hypotheses and Approaches for Solving the Mapping Problem

In this research, I designed a new structured test automation language (STAL) to automate the mapping from abstract tests to concrete tests. The reason that I could not use an existing language will be discussed in section 2.5.

The mapping problem causes testers to write redundant code, make errors and waste their time. Thus, the goal is to find a way to avoid writing redundant code. One possible solution is that for one basic identifiable element of a model, if we write the corresponding executable code once, the code will be generated automatically the next time when the same element appears in an abstract test. An *identifiable element* is an element of a model that can be used multiple times in abstract tests and can be mapped to one or more lines of code in concrete tests, for example, an event or a transition in a FSM.

STAL is used to create mapping from identifiable elements in the model and automate the transformation from abstract tests to concrete tests. First, programmers or testers use the automation language to create mappings from each basic identifiable element of the model to the executable test code. Once the mappings are generated, testers do not need to write concrete tests line by line for each abstract test; instead, the concrete tests will be generated automatically from the abstract tests by assembling pieces of executable test code based on the mappings. This structured test automation language will be used to improve the efficiency of generating concrete tests from a variety of abstract tests and reduce the potential errors. A test automation tool, the structured test automation framework (STALE) was built and testers can use it to create new mappings between basic elements

and executable code, and view mappings and generate concrete tests automatically.

Section 4.2 will describe STAL and section 4.4 will talk about how STALE works. Experiments were conducted to compare the mapping process using STAL with manual transformation. The hypotheses about STAL are listed below. The experiments in section 5.2 will answer the hypotheses.

- H1: The test automation language is applicable to real programs
- H2: When converting abstract tests to concrete tests, the test automation language can be used to reduce faults and help testers avoid writing redundant code by comparison with doing the same process by hand
- H3: The test automation language can be used for different kinds of programs such as web applications and GUIs

The transformation with STAL is supposed to take less time (cost) than the manual process and maintain the same effectiveness (map abstract tests to concrete tests). So the mapping process with STAL should be more cost-effective than the manual transformation.

### 1.3.3 Hypotheses and Approaches for Solving the Test Oracle Problem

For the test oracle problem, ten test oracle strategies (OSes) were developed, each of which has different precision and frequencies of checking program states. The formal definitions of the OSes are given in section 4.3. Section 5.3 will show the experiments in which the new OSes were compared with two other oracles: the *null test oracle* (NOS) and the *state invariant test oracle* (SIOS), which served as baselines. NOS is implicitly provided by the Java runtime system. It only shows a failure when a program throws an exception or does not respond. SIOS checks every state invariants in models. The new test oracle strategies have different precisions and check different internal state variables and outputs after each transition or at the end of a test.

For each OS (except NOS, which was expected to be much less effective than other OSes), the effectiveness and cost were calculated. The effectiveness of an OS was defined as

the percentage of the number of defects revealed by this test oracle method; the cost was measured in terms of the number of times internal state variables and outputs are checked. The cost-effectiveness was the ratio of the percentage of the faults detected over the number of times the internal state variables and outputs are checked. Four research questions were raised and answered in the experiments:

**RQ1:** With the same test inputs, does a more precise OS reveal more failures than a less precise OS?

**RQ2:** With the same test inputs, does checking outputs and internal state variables multiple times reveal more failures than checking the same outputs and internal state variables once?

**RQ3:** With the same OS, do tests that satisfy a stronger coverage criterion reveal more failures than tests that satisfy a weaker coverage criterion?

**RQ4:** Which OS is recommended when considering both effectiveness and cost?

With the same test inputs, using cost-effective test oracle strategies should result in cost-effective tests.

# Chapter 2: Background and Related Work

This chapter introduces software testing techniques and concepts used in this research, discusses the work related to the three sub-problems for generating tests from behavioral models, and discusses the difference between this research and previous work.

Recall that in model-based testing, testers choose coverage criteria to generate test requirements based on a model. Then abstract tests are generated to satisfy the test requirements. Extra information, including abstract test values and data mappings, is needed to convert abstract tests to concrete tests. Thus, testers need to select models and coverage criteria, then deal with the three sub-problems: the minimum cost test paths problem (MCTP), the mapping problem, and the test oracle problem.

This chapter first talks about the model abstraction and the modeling language and type of model used in this research, then introduces the test graph coverage criteria that were used in this research. Mutation analysis is discussed in section 2.3 because it was used to generate faults and evaluate tests. Section 2.4 talks about two related problems to the MCTP problem: the test suite minimization problem and shortest superstring problem. Section 2.4 discusses the greedy set-covering and matching-based prefix graph algorithms, which are two new solutions to MCTP. Section 2.5 discusses the disadvantages of the current mapping techniques and why the structured test automation language (STAL) is useful. Section 2.6 compares the previous work and my work on the test oracle problem, concluding that my work is the most comprehensive study so far in terms of multiple metrics such as the number of experimental subjects and variety of test oracle strategies.

## 2.1 Model Selection

Model abstractions need to be specified because they determine what artifacts will be used to generate tests. Prenninger and Pretschner [21] concluded that a system can be abstracted for six purposes: (1) getting enriched knowledge of the system and its environment, (2) obtaining the specification of the system, (3) accessing parts of a system, (4) communicating between developers, (5) generating code, and (6) testing systems.

While creating test models, testers can either write *model program*s in a specific language or draw visualization diagrams. Model programs use specification languages such as Spec# [22] or programming languages such as Java or C# to describe model behaviors. ModelJUnit [23], Spec Explorer [24], NModel [25], and Conformiq [26] use Java, Spec#, C#, and Conformiq Modeling Language (QML) [26] to write model programs that can be converted to finite state machines or extended finite state machines.

Many testers prefer to use visual diagrams such as finite state machines, extended finite state machines, and UML behavioral diagrams directly as test models. They can reuse and adapt design models or build new test models [27]. Furthermore, the models from the design phase are more likely to conform to system requirements than those from source code. Researchers have come up with different ways to transform models created for (1) and (2) to test models for (6) [28–35]. Many [28–31, 33–35] used UML models to generate tests. This research applied the same approach and used the UML behavioral diagrams, specifically UML state machine diagrams.

## 2.2 Test Graph Coverage Criteria

Since tests generated from behavioral models have to satisfy some coverage criterion, this section talks about coverage criteria used in this research. Because behavioral models such as UML state machine diagrams can be transformed into general graphs, this research used graph coverage, which are defined on general graphs.

The following definitions are taken from Ammann and Offutt's software testing book

[7]. A graph $G$ is drawn as a collection of circles connected by arrows, where the circles represent nodes and the arrows represent edges. Formally, $G$ is

- a set $N$ of *nodes*, where $N \neq \emptyset$
- a set $N_0$ of *initial nodes*, where $N_0 \subseteq N$ and $N_0 \neq \emptyset$
- a set $N_f$ of *final nodes*, where $N_f \subseteq N$ and $N_f \neq \emptyset$
- a set $E$ of *edges*, where $E$ is a subset of $N \times N$

Thus, a test graph $G$ **requires** at least one initial and one final node, but **allows** more initial and final nodes.

*Edges* are considered to be directed arcs *from* one node $n_i$ and *to* another $n_j$ and written as $(n_i, n_j)$. The initial node $n_i$ is sometimes called the *predecessor* and $n_j$ is called the *successor*. A *path* is a sequence $[n_1,\ n_2,\ ...,\ n_M]$ of nodes, where each pair of adjacent nodes, $(n_i,\ n_{i+1})$, $1 \leq i < M$, is in the set $E$ of edges. The *length* of a path is defined as the number of nodes it contains. A *subpath* of a path $p$ is a subsequence of $p$ (including $p$ itself). A *test path* represents the execution of a test case on a graph. Test paths must start at an initial node and end at a final node. A test path $p$ *covers* a subpath $q$ if $q$ is a subpath of $p$.

*Test requirements* $TR$ are specific elements of software artifacts that must be satisfied or covered. An example test requirement for the test criterion "cover every node" is "cover the initial node." A *test criterion* is a rule or collection of rules that, given a software artifact, imposes test requirements that must be met by tests. For example, the test criterion "cover every node" results in one test requirement for each node in the graph. This test criterion requires that at least one test path is needed to cover all the nodes.

**Edge** coverage (EC) requires the each edge is covered by test paths. The formal definition of the edge coverage criterion on a general graph is given below.

**Definition 2 (Edge Coverage).** *$TR$ contains each edge, inclusive, in $G$.*

The **edge-pair (EP)** coverage (EPC) criterion was originally defined for finite state machines by Pimont and Rault [36] and has also been called *two-trip* [37] and *transition-pair* [38]. The test requirements of EPC are subpaths of length no more than three (the length is measured in terms of the numbers of nodes) in $G$. Thus, edge-pair coverage subsumes edge coverage (test requirements of length of two) and node coverage (test requirements of length of one) in $G$. The formal definition of the edge-pair coverage criterion on a general graph is given below.

**Definition 3** (**Edge-pair Coverage**)**.** *$TR$ contains each reachable path of length up to 2, inclusive, in $G$.*

A path from node $n_i$ to $n_j$ is *simple* if no node appears more than once in the path. That is, simple paths cannot have internal loops. However, if the entire path is a loop that has identical first and last nodes, the loop is a simple path. Graphs may have lots of simple paths, many of which are subpaths of other simple paths. To eliminate the redundancy and maintain the coverage strength, *prime path coverage* was invented [7]. A path from $n_i$ to $n_j$ is *prime* if it is simple and does not appear as a proper subpath of any other simple path.

**Definition 4** (**Prime Path Coverage**)**.** *$TR$ tours each reachable prime path in $G$.*

Prime path coverage (PPC) was used in the experiments for the minimum cost test paths problem to generate abstract tests. The experiments for the mapping problem used edge coverage while the experiments for the test oracle problem generated tests for both edge and edge-pair coverage criteria.

## 2.3 Mutation Analysis

Mutation analysis (mutation testing) [39] makes syntactic changes to a program, creating variants of the original program that are called *mutants*. If a test causes a mutant to generate a different result from the original program, this mutant is *killed*. A mutant is called *equivalent* if this mutant cannot be killed by any tests. A *mutation score* is the ratio

of killed mutants over non-equivalent mutants, which reflects the strength of a set of tests. A stronger set of tests usually gets a higher mutation score.

Mutation's strength comes from *mutation operators*, which define rules for changing programs. The quality of the mutation operators determines the effectiveness of the test cases. Mutation operators that modify individual statements (*method-level*) have been designed for many languages, including Fortran [40] and C [41]. Mutation operators that modify connections among classes (class-level) test features like inheritance and polymorphism [42,43].

Mutation analysis systems are used in two general ways. The first is to assess the quality of an existing set of tests in terms of what percentage of mutants it kills (its *mutation score*). This method can be used in practice, and has widely been used to assess the quality of other testing techniques. Andrews et al. [44] presented evidence that mutation-adequate tests can indeed help find "real faults." The second use is to help testers design very high quality tests, by creating mutants and challenging a tester to iteratively design tests to kill all or most mutants. Mutation-based tests can be designed automatically [3,45], or by hand.

This research used a Java mutation testing tool *muJava* [46] since all experimental subjects were in Java. MuJava offers several important features to support testing and experimentation. An important feature is that it lets testers see mutants visually, enhancing their ability to design tests to kill mutants. MuJava also implements both method-level and class-level (OO) mutation operators, and allows testers to choose which specific operators to use. MuJava also provides detailed analysis of test results: including which mutant each test kills, and which test kills each mutant. This information is valuable for research and for designing tests. MuJava is freely available for downloading and has been used quite widely in academia. The latest version of muJava uses JUnit tests and is compatible with J2SE 5.0 and Java SE 6, including generics, annotations, enumerations, varargs, enhanced for-each loops, and static imports. MuJava was used in the experiments for the test oracle problem.

## 2.4 The Minimum Cost Test Paths Problem

In software testing, the minimum cost test paths problem (MCTP) was first defined in this research (published first in ICST 2012 [47]). Thus, there were no directly related papers to this problem. This section first introduces the most related problem to MCTP: the test suite minimization problem, whose solutions were used in the solutions to the MCTP problem. Second, this section talks about the shortest superstring problem and its two solutions: the greedy set-covering and the matching-based prefix graph algorithm, which were adapted to solve MCTP. The definition of the shortest superstring problem, the greedy set-covering algorithm, and matching-based prefix graph algorithms are given in sections 2.4.2, 2.4.3 and 2.4.4. The reason that I chose these two algorithms is that they are the most widely used algorithms used to solve the shortest superstring problem. Many other algorithms were variants of them.

### 2.4.1 The Test Suite Minimization Problem

The nearest related testing problem to the MCTP problem is the test suite minimization problem of reducing the number of tests in a test suite while still maintaining coverage. Yoo and Harman gave the definition of the *test suite minimization problem* in a survey paper [48]:

**Definition 5** (**Test Suite Minimization Problem (TSMP)**). *Consider a set of test case requirements* $\mathcal{R} = \{r_1, \cdots, r_n\}$ *that must be satisfied to provide the desired adequate testing of the program. There is a test suite* $\mathcal{T}$*. For any subset* $T_i \subseteq \mathcal{T}$*,* $\forall i = 1, \ldots, n$*, any test case* $t_j \in T_i$ *can be used to test a subset of* $\mathcal{R}$*. The problem TSMP is to find a representative set* $\mathcal{T}' \subseteq \mathcal{T}$ *such that all the requirements in* $\mathcal{R}$ *can be tested using the test cases in* $\mathcal{T}'$*.*

This test suite minimization problem is related to the set cover problem [49] and several heuristics have been developed [50–53].

This research used a heuristic test suite minimization algorithm to reduce the number

of test paths for each solution to MCTP. The approach is similar to the *GE* and *GRE* heuristics proposed by Chen and Lau [51]. The test suite minimization algorithm used in this research is a simplified version of the *GRE* heuristic, removing all *redundant* test paths, but not running the *GE* heuristic as Chen and Lau did. A test path is *redundant* if it satisfies a proper subset of the test requirements satisfied by another test path.

### 2.4.2 The Shortest Superstring Problem

This research adapted the ideas used in solving the *shortest superstring problem* to solve the MCTP problem. The *shortest superstring problem* has been used in the fields of computational biology and data compression, but not previously in software testing. The MCTP problem can be modeled as a shortest superstring problem, allowing us to adapt existing algorithms for solving the shortest superstring problem. *The shortest superstring problem* is defined as follows.

**Definition 6** (**Shortest Superstring Problem [17]**). *Given a finite alphabet* $\Sigma$*, and a set of* $n$ *strings,* $S = \{s_1, \ldots, s_n\} \subseteq \Sigma^+$*, the shortest superstring problem is to find a shortest string* $s$ *that contains each* $s_i$ *as its substring. Without loss of generality, we assume that no string* $s_i$ *is a substring of another string* $s_j$*,* $\forall j \neq i$*.*

In model-based software testing with graphs, a string is a test requirement as a sequence of nodes such as a prime path. Prime paths were used as test requirements in the experiments for MCTP in section 5.1. For example, for the two prime paths $[1,\ 2,\ 3,\ 1]$ and $[2,\ 3,\ 1,\ 2]$, two possible superstring test paths that cover both are $s_1 = [1,\ 2,\ 3,\ 1,\ 2]$ and $s_2 = [2,\ 3,\ 1,\ 2,\ 3,\ 1]$. The length of the string $s$ is $|s|$, the number of nodes in $s$. Given two different strings $s$ and $t$, the overlap between $s$ and $t$, $|over(s,\ t)|$, is the length of the longest string $x$ such that $s = mx$ and $t = xn$ with non-empty strings $m$ and $n$. $m$, denoted as $|pref(s,\ t)|$ is the prefix of $s$ with respect to $t$. $|pref(s,\ t)|$ is also referred as the *prefix distance* from $s$ to $t$. In the example above, for $s_1$, $[2,\ 3,\ 1]$ is the overlap and $[1]$ is the prefix of $[1,\ 2,\ 3,\ 1,\ 2]$; for $s_2$, $[1,\ 2]$ is the overlap and $[2,\ 3]$ is the prefix of

[2, 3, 1, 2, 3, 1].

*The shortest superstring problem* was proved to be *NP-complete* by Gallant, Maier and Strorer [54]. Tarhio and Ukkonen [55] proposed a greedy algorithm whose *approximation ratio* (the ratio between the polynomial-time approximation algorithm and the hypothetical optimal solution) is $2 \ln n$. Blum et al. [56] presented another greedy algorithm that finds a superstring that is no longer than four times optimal and a modified one that was no more than three times optimal. The best known approximation ratio is 2, given by Weinard and Schnitger [57].

Romero, Brizuela and Tchernykh [58] conducted an experimental comparison of the shortest superstring problem. Their paper compared a 3-approximation algorithm and a 4-approximation algorithm. The results showed the superstrings returned by both approximation algorithms are almost the same. Both solutions were very close to the optimum; at most 1.4% longer than the optimum.

Romero et al. used DNA sequences as examples in their experiment to find a shortest superstring. This research used test requirements (prime paths) as examples. This research presents results on test requirements, precisely prime paths, and different algorithms. Instead of generating only one super-string (super-test requirement), it is necessary to split the results of the superstring into separate test paths. Detailed algorithms will be described in section 4.1.

### 2.4.3 The Greedy Set-Covering Algorithm

The first existing approximation algorithm for the shortest superstring problem to adapt is the set-covering algorithm using a greedy algorithm. Consider the shortest superstring problem. The input to an algorithm is a set of strings, $S$, of size $|S| = n$. The greedy algorithm works as below:

1. Initialize $T$ to be $S$.

2. At each step, select two strings from $T$ that have maximum overlap and replace them

with the string obtained by overlapping them as much as possible. If the remaining strings have no overlap, they are concatenated.

3. After $n-1$ steps, the single string in $T$ will be the superstring returned.

For $s_i, s_j \in S$ and $k > 0$, if the last $k$ symbols of $s_i$ are the same as the first $k$ symbols of $s_j$, let $\sigma_{ijk}$ be the string obtained by overlapping these $k$ positions of $s_i$ and $s_j$. Figure 2.1 shows the superstring covering $s_i$ and $s_j$.

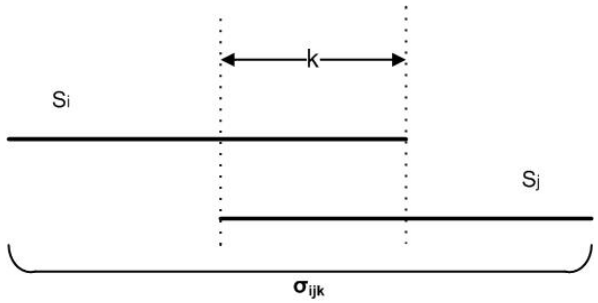


Figure 2.1: A Superstring for Two Arbitrary Strings

Let $M$ be the set that consists of the string $\sigma_{ijk}$ for all valid choices of $i, j, k$. The final set is $P \cup M$, where $P$ is the set of the original strings.

### 2.4.4 The Matching-based Prefix Graph Algorithm

The second approximation algorithm for the shortest superstring problem finds matchings based on a prefix graph and the definition of prefix graph is shown as follows.

**Definition 7** (**Prefix Graph**). *Given a set of strings $S$, a* prefix graph $G = (V,\ E)$ *is constructed as follows: a vertex $v_i \in V$ represents a string $s_i \in S$. For each edge $e_{ij} \in E$, the cost $c_{e_{ij}}$ of the edge is $|pref(s_i,\ s_j)|$ if $s_i$ and $s_j$ overlap or $+\infty$ otherwise.*

The matching-based approximation algorithm for the shortest superstring problem is to find a matching over a prefix graph. One observation is that the minimum weight of a traveling salesman tour of the prefix graph gives a lower bound on the length of the resulting optimal superstring (denoted by OPT) that covers all nodes of the prefix graph.

However the minimum traveling salesman tour OPT cannot be efficiently computed since this problem is NP-complete. Thus, the key idea is to find the minimum weighted paths, using cycle covers, over the prefix graph $G$. A *cycle cover* is a collection of disjoint cycles that cover all vertices. If $s_1,\ s_2,\ \ldots,\ s_n,\ s_1$ is the minimum-weight cycle cover, then

$$\begin{aligned} OPT &\geq \sum_i |prefix(s_i,\ s_{i+1})| + |overlap(s_1,\ s_n)| \\ &= |prefix(s_1,\ s_2)| + |prefix(s_2,\ s_3)| + \\ &\quad \ldots + |prefix(s_n,\ s_1)| + |overlap(s_n,\ s_1)|. \end{aligned}$$

The prefix graph is then transformed into a bipartite graph $G = (U \cup V,\ E)$. $U = \{u_1, ..., u_n\}$ and $V = \{v_1, ..., v_n\}$ are the vertex sets of the two sides of the graph. For each vertex $u_i$ on the left side and each vertex $v_j$ on the right side, add an edge of weight $|prefix(u_i, v_j)|$ if $u_i$ and $v_j$ overlap. Given a graph $G = (U \cup V,\ E)$, a *matching* $M$ in $G$ is a set of edges such that no two edges share a common vertex. A *perfect matching* ensures that every vertex of the graph is incident to exactly one edge. Therefore, each cycle cover of the prefix graph corresponds to a perfect matching of the same weight in $G$. Thus, finding a minimum weight cycle is equivalent to finding a minimum weight perfect matching in $G$.

The bipartite graph $G = (U \cup V,\ E)$ is used to find as many perfect matchings as possible. A maximum matching is found from those. The *Edmonds-Karp* algorithm [59] can find a maximum matching in the bipartite graph. Although the running time of the *Edmonds-Karp* algorithm is $O(VE^2)$, it ran slowly in practice, thus was not used. Instead, a heuristic greedy approach was applied.

## 2.5 The Mapping Problem

This section discusses the previously existing mapping strategies and why I created a new test automation language to automate the mapping problem.

In model-based testing, models only specify key aspects of software's behavior and cannot provide enough information for generating tests. When deriving test cases from UML behavioral models, additional information needs to be specified, such as test values and test oracles. Other UML models (for example, use case diagrams and class diagrams) are often used to provide the missing information.

Briand and Labiche [30] developed the TOTEM system, which uses many artifacts: use case diagrams, use case descriptions, sequence or collaboration diagrams for each use case, class diagrams, and a data dictionary specifying the details of classes. Nebut et al. [31] applied a use case driven approach. This approach extracts additional information from use case models and requires a behavioral model (sequence, state machine, or activity diagram) to specify the sequence ordering of the use cases. Moreover, use cases must have contracts (pre- and post-conditions) to help infer the partial ordering of functionalities. Furthermore, the behavioral models have to be consistent with the use cases. That is, the parameters in the use cases have to be the same as those in the behavioral models. This use case driven approach was validated on one embedded system.

The UML Testing Profile (UTP) [60] reuses some concepts of the UML but adds components for testing such as test context, test case, test component, and verdicts. When creating concrete tests, information in abstract tests have to match attributes of other diagrams such as class and object diagrams.

In summary, when creating tests from UML state machine diagrams, the previous approaches used additional diagrams and structures to provide abstract test values and transform them to concrete test values. The abstract test values have to be consistent with those defined in diagrams such as class diagrams or object diagrams. Thus, abstract tests not only have test sequences, but also lots of abstract test values, which complicates the transformation from abstract to concrete tests. Creating many formalized diagrams for testing may not be difficult for companies and organizations that can reuse models from the design phase and have lots of experts in model-based testing. However, it is very expensive for organizations that do not have such resources.

This research presents the structured test automation language (STAL) to generate tests from UML behavior models, specifically UML state machine diagrams without additional diagrams. Programmers and testers can use STAL to provide missing information by creating mappings from identifiable elements of the model to concrete test code. Previous papers included few empirical studies in comparing concrete MBT solutions with manual approaches. This research includes an empirical comparison of transforming abstract tests to concrete tests by using STAL with the manual method in section 5.2.

I tried to use an existing model-to-test transformation language *Meta-Object Facility Model To Text Transformation Language (MOFM2T)* [61]. Unfortunately, this language has several characteristics that made it impossible to use for this research.

MOFM2T [61] is part of OMG's *Model-driven architecture (MDA)* [62] and was designed to transform models to code for general use. *Acceleo* [63] is the only Eclipse Foundation project that implements MOFM2T. It reads Eclipse Modeling Framework (EMF [64])-based models and transforms them into programs in several languages.

Adapting *MOFM2T* and *Acceleo* to define mappings from abstract to concrete tests posed two problems. First, mapping creation cannot reuse much of the syntax of *MOFM2T*. For example, the *for* loop structure used in *MOFM2T* goes through each component of the same type (e.g., states) in a model and translates them to similar code. However, the *for* loop cannot be used for the mappings because each identifiable element in a model is mapped to different test code.

Second, testers cannot write test code to create mappings with *MOFM2T* directly. Ideally, testers first choose an identifiable element, write down its name, write the code for it, then create the mapping. However, *MOFM2T* and *Acceleo* cannot recognize the element names. Testers would need to write code to look for the identifiable element from the top level to the bottom level of the model structure.

## 2.6 The Test Oracle Problem

A *test oracle* is a verification mechanism to determine whether a test passes or not. Usually testers check the runtime behavior of a system by monitoring its observable outputs to verify whether the system has defects; however, sometimes a program is not testable because either testers do not know what the expected outputs are or they do not find an appropriate test oracle [65].

Test oracles can be generated by hand but it is very expensive. So researchers have developed the *automated oracles* that use assertions that are generated from pre and post-conditions and invariants annotated in specifications, documentations, and models [66, 67].

Recall that a *test oracle strategy* (abbreviated as OS) is defined as a rule or a set of rules to specify which program states to check. The precision of an OS refers to the degree to which the internal state variables and outputs of a program are checked by this OS. The more internal state variables and outputs an OS checks, the more precise the OS is. The precision of an OS also refers to the OS's tolerance for errors [9] because if an OS checks more internal state variables and outputs, it is possible that the OS can reveal more faults. The null OS (NOS) is considered as the least precise OS since it does not check any variable or output explicitly. It only reports a failure when a program terminates abnormally, thus, it tolerates many faults. In contrast, the most precise OS checks all possible internal state variables and outputs whenever they appear during the execution of test cases, so this OS is called *the very precise OS* [8] or *the maximum OS* [10].

How and which internal state variables and outputs should be checked by an OS depends on the cost-effectiveness of OSes. Although a more precise OS can reveal more faults (effectiveness), it needs more assertions (cost) than a less precise OS. Adding more assertions requires testers to provide more expected values and write more assertions manually, and takes longer to execute. Keep in mind that automated test oracle generation is often not available for many situations in industry.

The frequency of checking the program state is a factor when considering the cost-effectiveness of an OS. Given two OSes that check the same internal state variables and outputs, if they find the same faults, the one that checks the states less frequently will be more cost-effective.

To the best of our knowledge, only a few papers have studied the test oracle problem empirically. Briand et al. [8] compared the *very precise OS* with the *state invariant OS* (SIOS) based on the statecharts of three classes, each of which had less than 500 lines of code. The very precise OS checks all the class attributes and outputs after each operation and is considered to be the most accurate verification possible. In contrast, SIOS only checks the invariants of the states that are reached by test cases after each transition. They found that the very precise OS is more effective at revealing failures than SIOS. They also found that the cost of the very precise OS is higher than SIOS in terms of the number of test cases, the CPU execution time, and the lines of code.

Xie and Memon [11] considered what to evaluate and how often to check program states from GUIs. They found that the variations of the two factors affect the fault-detection ability and cost of test cases. They proposed six OSes that check a widget, a window, and all windows after every event and after the last event of a test. They concluded that "weak" OSes reveal fewer failures and a "thorough" OS at the end of a test case yields the best cost-effectiveness ratio in many cases. They evaluated the OSes for GUIs, not general applications.

Staats et al. [10] found that an OS that checks outputs and internal state variables can reveal more failures than an OS that only checks the outputs. They also concluded that the number of variables checked by the *maximum OS* is bigger than that checked by the *output-only OS* and only a small portion of the added internal state variables contributes to improving fault-detection ability. To evaluate how internal state variables affect the fault-detection ability, some *less precise OS*es were compared. A *less precise OS* checks outputs and some internal state variables but not all. In their experiment, internal variables were chosen randomly for these less precise OSes. Therefore, which internal variables contribute

to improving the effectiveness is unclear.

Shrestha and Rutherford [9] empirically compared NOS and the *pre and post-condition OS* (PPCOS) using the *Java Modeling Language* [68]. They found that the latter can find more faults than the former. They suggested that test engineers should move beyond NOS and use more precise OSes.

The *test oracle comparator problem* for web applications is to determine if a web application gets correct outputs automatically given a test case and expected results. Sprenkle et al. [69] developed a suite of 22 automated oracle comparators to check the *HTML* documents, contents, and tags. They found that the best comparator depends on applications' behaviors and the faults.

Yu et al. [70] studied the effectiveness of the *output-only OS* and six other OSes that check more internal state variables to detect special faults that appear in six concurrent programs. They found that these six more precise OSes revealed more failures than the *output-only OS*.

This research conducted a comprehensive experiment to study the effectiveness and cost of OSes and give guidelines about which OS should be used. A comparison between this research and others are in Table 2.1. The first column shows the metrics and the other columns represent others' work.

Shrestha et al. used nine small programs, with the biggest having 263 statements. Others studied no more than six programs. This research used 16 programs with lines of code (LOC) from 52 to 14,155. The subjects included general libraries, GUIs, and web applications. In contrast, Xie and Memon only studied GUIs and Sprenkle et al. studied web applications. Staats et al. worked on synchronous reactive systems [71], which do not have OO classes.

This research considered several OSes: NOS, SIOS, and ten more precise OSes, which is more comprehensive than the other studies. This research studied which internal state variables contribute to the effectiveness of the OSes. This research also studied the frequency of checking program states, which was only studied by Xie and Memon before. Twelve OSes that have different precisions and frequency of checking program states were used in the

experiment while both Shrestha et al. and Briand et al. studied two OSes.

Staats et al. [72] and Mateo and Usaola [73] proposed similar approaches that used mutation analysis to select which program states to check automatically. But this approach could be even more costly because users have to apply mutation analysis before providing test oracle data. Thus, this approach needs further study. Fraser and Zeller [74] also used mutation testing to derive test inputs and test oracles but they did not study the effectiveness or cost of OSes.

Table 2.1: A Comparison of Previous Test Oracle Research Papers

| **Metric** | **This research** | **Briand et al.** | **Xie et al.** | **Staats et al.** | **Shrestha et al.** | **Sprenkle et al.** | **Yu et al.** |
|---|---|---|---|---|---|---|---|
| Number of programs | 16 | 3 | 5 | 4 | 9 | 4 | 6 |
| Types of subjects | General, GUI, Web | General | GUI | Non-OO | General | Web | Concurrent |
| NOS used | Yes | No | No | No | Yes | No | No |
| PPCOS or SIOS used | Yes | Yes | No | No | Yes | No | No |
| Less precise OSes used | Yes | No | Yes | Yes | No | Yes | Yes |
| Internal state variables are used | Yes | No | Yes | Internal state variables picked randomly | No | Yes | Yes |
| Frequency of checking program states | Yes | No | Yes | No | No | No | No |
| The number of OSes used | 12 | 2 | 6 | 3 | 2 | 22 | 7 |

# Chapter 3: Proof of the Hardness of MCTP

Section 1.2.3 defined the minimum cost test paths problem (MCTP): Find a set of test paths $TP = \{t_1, t_2, \ldots, t_k\}$ that cover all test requirements in the graph $G$ such that the cost of using the test paths is minimum, given a set of test requirements $TR = \{r_1, r_2, \cdots, r_n\}$. Each test requirement is presented as a subpath in a graph $G = (V, E)$. The cost can be reduced in the following ways:

1. Fewer total nodes
2. Fewer test paths
3. Fewer test requirements per test path (smaller TR to TP ratio)
4. Shorter test paths

Specifically, the goals to be optimized are:

1. The total number of nodes
2. The total number of test paths
3. The maximum ratio of TR to TP subject to a bounded number of test paths
4. The total number of test paths subject to a bounded ratio of TR to TP
5. The total number of nodes subject to a bounded ratio of TR to TP

Each of these five goals can be instantiated as a variant of the MCTP problem. Variant (2), finding the fewest number of test paths, can be solved in polynomial time using a modified algorithm proposed by Aho and Lee [19]. However, other variants of the MCTP problem are provably NP-complete.

This chapter will show that variants (1), (3), (4), and (5) of MCTP are NP-complete and variant (2) can be solved in polynomial time. If MCTP variants (1), (3), (4), and (5) are NP-complete, then there are no efficient (polynomial-time) exact algorithms to solve them. The proof that MCTP variants (1), (3), (4), and (5) are NP-complete shows that first, MCTP variants (1), (3), (4), and (5) belong to NP and second, NP-complete problems that can be reduced to MCTP variants (1), (3), (4), and (5) exist (Bin-Packing problem and 3-Partition problem). The reductions of MCTP variants (1) and (5) come from a well-known NP-complete problem called the Bin-Packing problem [17,18], as defined below.

**Definition 8** (**Bin-Packing Problem**). *The inputs are a set of n objects, where the size* $s_i \in S$ *of the ith object satisfies* $0 < s_i < 1$. *The size of every bin is 1. The Bin-Packing problem is to find the minimal number of bins such that all objects are packed in the bins.*

The formal language for the Bin-Packing problem is BIN-PACKING = <O, S, z>: the size $s_i$ of object $o_i \in O$ satisfies $0 < s_i < 1$, and finds the number of bins of size 1 no more than z.

The reductions of MCTP variants (3) and (4) come from another NP-complete problem called the 3-Partition problem [20], as defined below.

**Definition 9** (**3-Partition Problem**). *Given a finite set A of* $3m$ *elements, a bound* $B \in Z^+$, *and a 'size'* $s(a) \in Z^+$ *for each* $a \in A$, *such that each s(a) with* $B/4 < s(a) < B/2$ *and* $\sum_{a \in A} s(a) = m * B$, *can A be partitioned into m disjoint sets* $S_1, S_2$, ..., $S_m$ *such that for* $1 \leq i \leq m$, $\sum_{a \in S_i} s(a) = B$*?*

The formal language for the 3-Partition problem is 3-PARTITION = <A, B, m>: a finite set of A has 3m elements. The size of each element $a$, $s(a)$ is between $B/4$ and $B/2$ exclusively and $\sum_{a \in A} s(a) = m * B$. Can m disjoint sets have all 3m elements and the size of each set is no more than B?

The definitions of the MCTP variants (1), (2), (3), (4), and (5) are shown below as well as the proofs.

**Definition 10** (**Minimum Cost Test Paths Problem (MCTP) Variant (1)**). *Consider a set of test requirements* $TR = \{r_1,\ r_2,\ \cdots,\ r_n\}$. *Each test requirement is presented as a subpath in a graph* $G = (V,\ E)$. *The problem MCTP variant 1 is to find a set of test paths* $TP = \{t_1, t_2,\ \ldots,\ t_k\}$ *that cover all test requirements in the graph* $G$ *such that the total number of nodes in the test paths is at most an integer* $d$.

The formal language for the decision problem of the MCTP variant (1) is MCTP-VARIANT1 = <G, TR, d>: graph G finds a set of test paths TP that covers all test requirements TR with the total number of nodes in TP no more than an integer d.

**Theorem 1.** *The problem MCTP variant (1) is NP-complete.*

*Proof.* First, I show that the MCTP variant (1) belongs to NP. That is, for any given solution instance, a polynomial-time verifier exists. Consider a solution instance $TP = \{t_1,\ t_2,\ \ldots,\ t_k\}$. Let the test requirements be $TR = \{r_1,\ r_2,\ \ldots,\ r_n\}$ and check if the total number of nodes in TP $|t_1| + |t_2| + \ \ldots + \ |t_k| \leq$ d. The running time of this part is $\sum_{s=1}^{k} |t_s|$. Then, check whether all $r_i \in TR$ $(i = 1,\ 2,\ \ldots,\ n)$ belong to at least one path in $TP$. For each $r_i$, the string matching algorithms is applied to identify whether $r_i$ belongs to some $t_j$. Using Knuth-Morris-Pratt's algorithm [75], this procedure takes time $O(|r_i| + |t_j|)$. Thus, in total, each requirement $r_i$ takes time $|r_i| \sum_{s=1}^{k} |t_s|$ to identity whether $r_i$ belongs to some $t_s$ in $TP$. The total running time of the verifier is $\sum_{i=1}^{n} |r_i| \sum_{s=1}^{k} |t_s| + \sum_{s=1}^{k} |t_s|$, polynomial in the size of the input.

Second, to prove that MCTP variant (1) is NP-hard, the Bin-Packing problem should be reduced to the MCTP variant (1) in polynomial time, that is BIN-PACKING $\leq_P$ MCTP-VARAINT1. Let <O, S, z> be an instance of BIN-PACKING. An instance of MCTP-VARIANT1 is constructed as follows. Each object $o_i \in O$ represents a test requirement $r_i \in TR$. Let A be the total number of nodes of the most number of test requirements that can be covered by a test path $t_s \in TP$. The size of unit-size bins is A / A = 1. Therefore, the size of $o_i \in O$ that represents $r_i \in TR$ is the number of the nodes of $r_i$ / A, which is less than 1.

I form a graph G = (V + TR, E), where V is a set of distinct nodes that appear in $r_i \in TR$. Each $r_i \in TR$ also represents a node in $G$. A node $n_u \in V$ is connected to another node $n_v \in V$ if edge $(n_u, n_v)$ is a subpath of any $r_i \in TR$. A node $n_u \in V$ is connected to $r_i \in TR$ if edges $(n_u, n_f)$ and $(n_l, n_u)$ are subpaths of any $r_i \in TR$, where $n_f$ is the first node of $r_i$ and $n_l$ is the last node of $r_i$. The size of a node that represents $r_i \in TR$ in $G$ is the total number of nodes in $r_i$. The size of a node $n_u \in V$ is 0. The size of any edges among nodes in V and TR is 0. Thus, the size of a complete path in $G$ only counts the number of nodes in $r_i \in TR$. Since A is the total number of nodes of the most number of test requirements that can be covered by a test path $t_s \in TP$ and z is the number of bins, the total number of nodes in all test paths is no more than z * A. Then the instance of MCTP-VARIANT1 is (G, TR, z * A), which can be created in polynomial time.

Now I show that BIN-PACKING has no more than z bins to fit all objects (test requirements) if and only if graph G has a set of $z$ test paths to cover all the test requirements $TR$ with the size of the test paths no more than z * A. Suppose BIN-PACKING has z bins to pack all objects and each object represents a test requirement $r_i \in TR$. Thus, the size of each object $o_i$ is the ratio of the number of $r_i$ over A. For test requirements in a bin, additional nodes $n_u \in V$ are used to construct a complete test path to cover the test requirements in this bin. Since only the size of $r_i \in TR$ is greater than 0, the size of this test path is no more than A. Thus, the total number of nodes in the test paths that cover all the test requirements is no more than z * A.

Conversely, suppose that graph G has a set of z test paths that covers all test requirements, where the size of the longest test path is A. For each test path, only test requirements remain and other nodes are removed. Each test requirement $r_i$ corresponds to an object $o_i \in O$, where the size of object $o_i$ is the size of $r_i$ over A such that the objects can fit in a bin. Thus, the total number of bins needed to fit all the objects is equal to the number of test paths z. Therefore, BIN-PACKING can be reduced to MCTP-VARIANT1. □

**Definition 11** (**Minimum Cost Test Paths Problem (MCTP) Variant (2)**). *Consider a set of test requirements* $TR = \{r_1,\ r_2,\ \cdots,\ r_n\}$. *Each test requirement is presented*

*as a subpath in a graph $G = (V,\ E)$. The problem MCTP variant 2 is to find a set of test paths $TP = \{t_1, t_2,\ \ldots,\ t_k\}$ that cover all test requirements in the graph $G$ such that the number of the test paths is minimum.*

**Theorem 2.** *The problem MCTP variant (2) is polynomial-time solvable.*

*Proof.* Aho and Lee defined the $CP_1^1$ (minimum-cardinality covering paths) problem [19]: find a set of covering paths for $G = (V,\ E)$ such that the number of paths is minimum. In this problem, the test requirements are nodes in $V$. Aho and Lee proved this problem is polynomial-time solvable. If each node in $G$ of the $CP_1^1$ problem is replaced with a sequence of nodes (test requirements in the MCTP Variant (2)), the MCTP Variant (2) can be solved in polynomial time using the same algorithm for the $CP_1^1$ problem. □

**Definition 12** (**Minimum Cost Test Paths Problem (MCTP) Variant (3)**). *Consider a set of test requirements $TR = \{r_1,\ r_2,\ \cdots,\ r_n\}$. Each test requirement is presented as a subpath in a graph $G = (V,\ E)$. The problem MCTP variant 3 is to find a set of test paths $TP = \{t_1, t_2,\ \ldots,\ t_k\}$ that cover all test requirements in the graph $G$ such that the maximum ratio of TR to TP is less than a number x and the total number of test paths is subjected to a bound p.*

The formal language for the decision problem of the MCTP variant (3) is MCTP-VARIANT3 = <G, TR, x, p>: graph G finds a set of TP that covers TR with the maximum ratio of TR to TP no more than x and the total number of test paths no more than p.

**Theorem 3.** *The problem MCTP variant (3) is NP-complete.*

*Proof.* First, I show that the MCTP variant (3) belongs to NP. That is, for any given solution instance, a polynomial-time verifier exists. Consider a solution instance $TP = \{t_1,\ t_2,\ \ldots,\ t_k\}$. Let the test requirements be $TR = \{r_1,\ r_2,\ \ldots,\ r_n\}$ and check the ratio of TR to TP for each test path. For each test path, $|t_s| = 1$, where $t_s \in TP$. Thus, for each test path $t_s \in TP$, the ratio of TR to TP is the number of test requirements toured by $t_s$.

Each test path $t_s$ takes time $|t_s| \sum_{i=1}^{n} |r_i|$ to identity whether $t_s$ covers some $r_i$ in $TP$ using Knuth-Morris-Pratt's algorithm. The running time of computing all the ratios of TR to TP is $\sum_{s=1}^{k} |t_s| \sum_{i=1}^{n} |r_i|$ and finding the max ratio takes $k-1$. Checking if the number of the test paths is no more than $p$ takes $O(1)$. Thus, the total running time is $\sum_{s=1}^{k} |t_s| \sum_{i=1}^{n} |r_i|$ + $k$, polynomial in the size of the input. Then check whether all $r_i$ ($i = 1,\ 2,\ \ldots,\ n$) belong to at least one path in $TP$, which can be computed in the first step. Thus, no additional computation is needed. The total running time of the verifier is $\sum_{s=1}^{k} |t_s| \sum_{i=1}^{n} |r_i| + k$, polynomial in the size of the input.

Second, to prove that the MCTP variant (3) is NP-hard, the 3-Partition problem should be reduced to the MCTP variant (3) in polynomial time, that is 3-PARTITION $\leq_P$ MCTP-VARAINT3. Let <A, B, m> be an instance of 3-PARTITION. An instance of MCTP-VARIANT3 is constructed as follows. Each element $a \in A$ represents a test requirement $r_i \in TR$. The size of each test requirement $r_i \in TR$ is 1. The ratio TR / TP for each test path is the number of test requirements toured by this test path. Let B be 3. Thus, each element $a \in A$ that represents $r_i \in TR$ (size of 1) is between B/4 and B/2 (3/4 and 3/2).

I form a graph G = (V + TR, E), where V is a set of distinct nodes that appear in $r_i \in TR$. Each $r_i \in TR$ also represents a node in $G$. A node $n_u \in V$ is connected to another node $n_v \in V$ if edge $(n_u, n_v)$ is a subpath of any $r_i \in TR$. A node $n_u \in V$ is connected to $r_i \in TR$ if edges $(n_u, n_f)$ and $(n_l, n_u)$ are subpaths of any $r_i \in TR$, where $n_f$ is the first node of $r_i$ and $n_l$ is the last node of $r_i$. The size of a node that represents $r_i \in TR$ in $G$ is 1. The size of a node $n_u \in V$ is 0. The size of any edges among nodes in V and TR is 0. Thus, the size of a complete path in $G$ only counts the number of $r_i \in TR$ covered by this test path. Each set is mapped to one test path. Therefore, the number of test paths cannot be more than $m$ and each test path has no more than B test requirements. Then the instance of MCTP-VARIANT3 is (G, TR, B, m), which can be created in polynomial time.

Now I show that 3-PARTITION has no more than m sets to fit all elements (test requirements) if and only if graph G has a set of $m$ test paths to cover all the test requirements $TR$ with the max ratio TR over TP no more than B. Suppose 3-PARTITION has m sets to fit all the elements and each element represents a test requirement $r_i \in TR$. Thus, the size of each element $a \in A$ is 1. Each set is mapped to one test path. Thus, the number of test paths is m as well. Since the number of all the elements is m * B, the max ratio of TR / TP for each test path is no more than B. Conversely, suppose that graph G has a set of m test paths that covers all the test requirements and the max ratio of TR / TP is B. For each test path, only the test requirements are kept and other nodes are removed. Each test requirement $r_i$ corresponds to an element $a \in A$, where the size of element $a_i$ is 1 (between B/4 and B/2). The total number of the test requirements in all the test paths is m * B, which is equal to 3m. Thus, 3m elements can be partitioned into m sets with the size of each set no more than B. Therefore, 3-PARTITION can be reduced to MCTP-VARIANT3.

□

**Definition 13** (**Minimum Cost Test Paths Problem (MCTP) Variant (4)**). *Consider a set of test requirements $TR = \{r_1,\ r_2,\ \cdots,\ r_n\}$. Each test requirement is presented as a subpath in a graph $G = (V,\ E)$. The problem MCTP variant 4 is to find a set of test paths $TP = \{t_1, t_2,\ \ldots,\ t_k\}$ that cover all test requirements in the graph $G$ such that the total number of test paths is no more than an integer p and TR to TP for each test path is subjected to a bounded ratio x.*

The formal language for the decision problem of the MCTP variant (4) is MCTP-VARIANT4 = <G, TR, p, x>: graph G finds a set of TP that covers TR with the number of test paths no more than p and the maximum ratio of TR to TP no more than x.

**Theorem 4.** *The problem MCTP variant (4) is NP-complete.*

*Proof.* First, I show that the MCTP variant (4) belongs to NP. That is, for any given solution instance, a polynomial-time verifier exists. Consider a solution instance $TP = \{t_1,\ t_2,\ \ldots,\ t_k\}$. Let the test requirements be $TR = \{r_1,\ r_2,\ \ldots,\ r_n\}$ and check the

ratio of TR to TP for each test path. For each test path, $|t_s| = 1$, where $t_s \in TP$. Thus, for each test path $t_s \in TP$, TR to TP is the number of test requirements toured by $t_s$. Each test path $t_s$ takes time $|t_s| \sum_{i=1}^{n} |r_i|$ to identity whether $t_s$ covers some $r_i$ in $TP$ using Knuth-Morris-Pratt's algorithm. The running time of computing all the ratios of TR to TP is $\sum_{s=1}^{k} |t_s| \sum_{i=1}^{n} |r_i|$ and finding the max ratio takes $k - 1$. Checking if the number of the test paths is no more than $p$ takes $O(1)$. Thus, the total running time is $\sum_{s=1}^{k} |t_s| \sum_{i=1}^{n} |r_i| + k$, polynomial in the size of the input. Next, whether all $r_i$ ($i = 1, 2, \ldots, n$) belong to at least one path in $TP$ needs to be checked, which can be computed in the first step. Thus, no additional computation is needed. The total running time of the verifier is $\sum_{s=1}^{k} |t_s| \sum_{i=1}^{n} |r_i| + k$, polynomial in the size of the input.

Second, to prove that the MCTP variant (4) is NP-hard, the 3-Partition problem is reduced to the MCTP variant (4) in polynomial time, that is 3-PARTITION $\leq_P$ MCTP-VARAINT4. Let <A, B, m> be an instance of 3-PARTITION. An instance of MCTP-VARIANT4 is constructed as follows. Each element $a \in A$ represents a test requirement $r_i \in TR$. The size of each test requirement $r_i \in TR$ is 1. The ratio TR / TP for each test path is the number of test requirements toured by this test path. Let B be 3. Thus, each element $a \in A$ that represents $r_i \in TR$ (size of 1) is between B/4 and B/2 (3/4 and 3/2).

I form a graph G = (V + TR, E), where V is a set of distinct nodes that appear in $r_i \in TR$. Each $r_i \in TR$ also represents a node in $G$. A node $n_u \in V$ is connected to another node $n_v \in V$ if edge $(n_u, n_v)$ is a subpath of any $r_i \in TR$. A node $n_u \in V$ is connected to $r_i \in TR$ if edges $(n_u, n_f)$ and $(n_l, n_u)$ are subpaths of any $r_i \in TR$, where $n_f$ is the first node of $r_i$ and $n_l$ is the last node of $r_i$. The size of a node that represents $r_i \in TR$ in $G$ is 1. The size of a node $n_u \in V$ is 0. The size of any edge among nodes in V and TR is 0. Thus, the size of a complete path in $G$ only counts the number of $r_i \in TR$ covered by this test path. Each set is mapped to one test path. Therefore, the number of test paths cannot be more than $m$ and each test path has no more than B test requirements. Then the instance of MCTP-VARIANT4 is (G, TR, m, B), which can be created in polynomial time.

Now I show that 3-PARTITION has no more than m sets to fit all elements (test requirements) if and only if graph G has a set of $m$ test paths to cover all the test requirements $TR$ with the max ratio TR over TP no more than B. Suppose 3-PARTITION has m sets to fit all elements and each element represents a test requirement $r_i \in TR$. Thus, the size of each element $a \in A$ is 1. Each set has no more than B elements, thus, each test path has no more than B test requirements. Since the number of all the elements is m * B, the total number of the test paths is no more than m. Conversely, suppose that graph G has a set of m test paths that covers all the test requirements and the max ratio of TR / TP is B. For each test path, only the test requirements are kept and other nodes are removed. Each test requirement $r_i$ corresponds to an element $a \in A$, where the size of element $a_i$ is 1 (between B/4 and B/2). The total number of the test requirements in all the test paths is m * B, which is equal to 3m. Thus, 3m elements can be partitioned into m sets with the size of each set no more than B. Therefore, 3-PARTITION can be reduced to MCTP-VARIANT4.

□

**Definition 14** (**Minimum Cost Test Paths Problem (MCTP) Variant (5)**). *Consider a set of test requirements $TR = \{r_1,\ r_2,\ \cdots,\ r_n\}$. Each test requirement is presented as a subpath in a graph $G = (V,\ E)$. The problem MCTP variant 5 is to find a set of test paths $TP = \{t_1, t_2,\ \ldots,\ t_k\}$ that cover all test requirements in the graph $G$ such that the total number of nodes in the test paths is at most an integer d and the ratio TR / TP is no more than m.*

The formal language for the decision problem of the MCTP variant (5) is MCTP-VARIANT5 = <G, TR, d, m>: graph G finds a set of test paths TP that covers all test requirements TR with the total number of nodes in TP no more than d and the ratio of TR / TP no more than m.

**Theorem 5.** *The problem MCTP variant (5) is NP-complete.*

*Proof.* First, I show that the MCTP variant (5) belongs to NP. That is, for any given solution instance, a polynomial-time verifier exists. Consider a solution instance $TP =$

$\{t_1,\ t_2,\ \ldots,\ t_k\}$. Let the test requirements be $TR = \{r_1,\ r_2,\ \ldots,\ r_n\}$ and check the ratio of TR to TP for each test path. For each test path, $|t_s| = 1$, where $t_s \in TP$. Thus, for each test path $t_s \in TP$, TR to TP is the number of test requirements toured by $t_s$. Each test path $t_s$ takes time $|t_s| \sum_{i=1}^{n} |r_i|$ to identity whether $t_s$ covers some $r_i$ in $TP$. The running time for computing all the ratios of TR to TP is $\sum_{s=1}^{k} |t_s| \sum_{i=1}^{n} |r_i|$ and finding the max ratio takes $k$. Checking if the total number of nodes in TP $|t_1| + |t_2| + \ \ldots + \ |t_k| \leq$ d takes $\sum_{s=1}^{k} |t_s|$. Thus, the running time is $\sum_{s=1}^{k} |t_s| \sum_{i=1}^{n} |r_i| + k + \sum_{s=1}^{k} |t_s|$, polynomial in the size of the input. Next, whether all $r_i$ $(i = 1,\ 2,\ \ldots,\ n)$ belong to at least one path in $TP$ needs to be checked, which can be computed in the first step. Thus, no additional computation is needed. The total running time of the verifier is $\sum_{s=1}^{k} |t_s| \sum_{i=1}^{n} |r_i| \ + \ k + \sum_{s=1}^{k} |t_s|$, polynomial in the size of the input.

Second, to prove that the MCTP variant (5) is NP-hard, the Bin-Packing problem should be reduced to the MCTP variant (5) in polynomial time, that is BIN-PACKING $\leq_P$ MCTP-VARAINT5. MCTP-VARIANT3 is a special case of MCTP-VARIANT5, where m is infinite. Because MCTP-VARIANT3 has been proved to be NP-complete through the reduction from the Bin-Packing Problem, MCTP-VARIANT5 is also NP-complete with an enhanced restriction on m.

□

# Chapter 4: Using Behavioral Models to Generate Tests

This chapter contains solutions to each of the three sub-problems of generating tests from behavioral models: the minimum cost test paths problem (MCTP), the mapping problem, and test oracle problem. Section 4.1 introduces a previous breadth-first search based solution and two new solutions that were developed based on the solutions to the *Shortest SuperString Problem* [47]. Section 4.2 describes a structured test automation language (STAL) to automate the mapping problem. To design STAL, how to create mappings, generate test values, transform original diagrams to general graphs, generate abstract tests, solve constraints, and generate concrete tests were considered [14, 76]. Section 4.3 describes ten new test oracle strategies (OSes), compared with two baseline OSes: the null and state invariant OSes [77]. The new OSes as well as the baseline OSes were used to provide the best possible strategies to solve the test oracle problem.

## 4.1 Solutions to the Minimum Cost Test Paths Problem

The previous breadth-first search, greedy set-covering, and prefix-graph solutions include several algorithms. The breadth-first search based solution is composed of Algorithm 1 (covering all nodes), Algorithm 2 (generating a small set of test paths), Algorithm 3 (generating test paths to cover all test requirements), and Algorithm 4 (the test suite minimization algorithm).

The greedy set-covering solution consists of Algorithm 5 (the set-covering algorithm), Algorithm 7 (the splitting algorithm), and the test suite minimization algorithm. The prefix-graph solution is comprised of Algorithm 6 (the matching-based prefix graph algorithm), plus the splitting and test suite minimization algorithms.

The graph coverage web application [12] takes a directed graph $G = (V,\ E)$ and set of test requirements represented by some subpaths in $G$. $G$ has a set of initial nodes $I$ and final nodes $F$. The idea of the breadth-first search solution is to list a set of paths $P$ that cover all the nodes. This step was implemented in Algorithm 1. In Algorithm 2, three steps were needed to make $P$ a set of test paths. First, I kept those paths in $P$ with initial nodes in $I$ and final nodes in $F$. Second, for a path in $P$ with its initial node in $I$ but final node not in $F$, as long as it contains a final node in this path, I took a subpath with its initial node in $I$ and final node in $F$. Last, I extended all the remaining paths (without final nodes in $F$) by making their final nodes $\in F$. In Algorithm 3, I extended all the test requirements that were not covered by the set of test paths generated by Algorithm 2 by making their initial nodes $\in I$ and final nodes $\in F$.

### 4.1.1 The Previous Breadth-first Search Solution

**Algorithm 1** The Heuristic Algorithm to Cover All Nodes

**Input:** A directed graph $G = (V,\ E)$ and a set of initial nodes $I \subseteq V$
**Output:** A set of paths $P$ that cover all the nodes in $V$

```
1: for each initial vertex v_i ∈ I do
2:     run the Breadth-First-Search (BFS) [59] algorithm to cover the vertices in V;
3:     build a BFS-tree with the root v_i;
4:     add the path from v_i to a vertex v ∈ V \ {v_i} into P
5: end for
6: return P
```

The output of Algorithm 1, a set of paths $P$, is an input to Algorithm 2. Then the paths in the set $P$ are extended to reach a final node using Algorithm 2. This results in a small set of complete test paths $TP$, which is an input to Algorithm 3. This set of complete test paths may not cover all the test requirements. The output of Algorithm 3 is a set of complete test paths that cover all the test requirements, which is an input to Algorithm 4. The inputs of Algorithm 4 also include a set of test requirements $TR$. The output of Algorithm 4 is a set of non-redundant test paths $TP'$ that cover all the test requirements in $TR$. Recall that a test path is *redundant* if it satisfies a proper subset of the test requirements satisfied by another test path, mentioned in section 2.4.1.

**Algorithm 2** The Heuristic Algorithm for Generating a Small Set of Test Paths

**Input:** A set of paths $P$ and a set of final nodes $F$
**Output:** A set of test paths $TP$

```
1: for each path p_i ∈ P do
2:   if the last node of p_i ∈ F then
3:     add p_i to TP and remove p_i from P;
4:   else
5:     for each node f_j ∈ F do
6:       if p_i includes f_j then
7:         chop the sub-path p_ij from the beginning to the position where f_j is in p_i
8:         add p_ij to TP and remove p_i from P;
9:         break;
10:      end if
11:    end for
12:    if the last node of p_i ∉ F then
13:      extend p_i to reach a final node;
14:      add p_i to TP and remove p_i from P;
15:    end if
16:  end if
17: end for
18: remove redundant paths in TP;
19: return TP
```

**Algorithm 3** The Breadth-first Search Algorithm for Generating Test Paths to Cover All Requirements

**Input:** A set of test requirements $TR$ and a small set of test paths $TP$
**Output:** A complete set of test paths $CTR$ that covers all the test requirements of $TR$

```
1: for each test requirement tr_i ∈ TR that is not covered by TP do
2:   if the first node of tr_i is not an initial node, nor is the last node a final node then
3:     repeat
4:       for each path p_j ∈ TP do
5:         if p_j includes the first node or last node of tr_i then
6:           extend tr_i using p_j such that tr_i reaches an initial node or final node;
7:         end if
8:       end for
9:     until tr_i is a complete path
10:    add tr_i to CTR;
11:  end if
12: end for
13: return CTR
```

**Algorithm 4** The Test Suite Minimization Algorithm to Remove Redundant Test Paths

**Input:** A set of test requirements $TR$ and a set of test paths $TP$

**Output:** A complete set of non-redundant test paths $TP' \subseteq TP$ that cover all the test requirements of $TR$

```
1: for each test path tp_s ∈ TP do
2:    find a set of the test requirements TR_s ⊆ TR that are covered by tp_s
3:    for each test path tp_t ∈ TP and tp_t ≠ tp_s do
4:       find a set of the test requirements TR_t ⊆ TR that are covered by tp_t
5:       if TR_s ⊆ TR_t then
6:          remove tp_s from TP
7:       end if
8:    end for
9: end for
10: return TP (note this TP (TP') may not the same as the original TP)
```

### 4.1.2 A Set-covering Based Solution

The set-covering based solution uses the set-covering algorithm, the splitting algorithm, and the test suite minimization algorithm. Given a set of test requirements, the set-covering algorithm returns only one super-test requirement, which is concatenated by multiple sub-paths and test requirements. This super-test requirement covers all the test requirements. Then, the splitting algorithm cuts the super-test requirement into separate complete paths. Finally, the test suite minimization algorithm removes any *redundant* test paths.

The set-covering algorithm is described in Algorithm 5. I used the greedy approach to find the set covers for the graph. Section 4.1.4 describes the splitting algorithm (Algorithm 7), which is also used by the matching-based prefix graph solution (Algorithm 6). The set-covering based solution used Algorithm 4 to minimize the size of test suites as well.

The set-covering algorithm (Algorithm 5) has two parts: constructing the sets and picking the set covers using the greedy algorithm. The running-time complexity of constructing the sets is $O(n^2)$ while that of picking the sets is $O(n \lg n)$. Let $\Pi$ denote the super-test requirement that will be the output of set-covering algorithm. Consider a test path $g_i$. The *cost* of $g_i$ is $|g_i|$ (the length of $g_i$) and the *effectiveness* of $g_i$ is the *number* of test requirements that are not covered by the test paths already in $\Pi$ but will be covered by $g_i$ when adding $g_i$ into $\Pi$. Line 8 of Algorithm 5 shows that the cost-effectiveness of a path $g_i$ is calculated when picking set covers greedily during each iteration.

**Algorithm 5** The Set-covering Algorithm

**Input:** A set of test requirements $TR$
**Output:** A concatenation of paths that covers all test requirements $\Pi$

```
1: for each test requirement tr_i ∈ TR and test requirement tr_j ∈ TR, i ≠ j do
2:    if tr_i has overlap with tr_j then
3:       put the super-test requirement s_k of tr_i and tr_j into a set S {refer to Figure 2.1}
4:    end if
5: end for
6: repeat
7:    for each element g_i ∈ TR ⋃ S do
8:       compute the cost-effectiveness of g_i;
9:       find g_i with the minimum cost-effectiveness and extend the super-test requirement
         Π with g_i;
10:      remove the test requirements covered by g_i;
11:   end for
12: until TR = ∅
13: return the super-test requirement Π
```

### 4.1.3 A Matching-based Prefix Graph Solution

This matching-based prefix graph solution is composed of the prefix-graph based algorithm, the splitting algorithm, and the test suite minimization algorithm. Given a set of test requirements, the prefix-graph based algorithm returns a set of super-test requirements that cover all the test requirements. Algorithm 6 constructs a prefix graph and a bipartite graph $G = (V \bigcup U,\ E)$ in the same way as is described in Section 2.4.4.

A perfect matching $M$ over this bipartite graph $G$ is a set of edges that connect left side nodes to right side nodes in the bipartite graph (each node represents a test requirement in the bipartite graph) such that no node is incident to two or more edges. An edge in the perfect matching $M$ represents a super-test requirement of the two nodes to which the edge is connected. Then, with the calculated perfect matching $M$, a cycle cover is constructed by merging super-test requirements represented by the edges in the perfect matching $M$. Here, the greedy algorithm is used to select a cycle cover or a test requirement that has the minimum cost-effectiveness each time until all the test requirements are covered. The cost of a cycle cover is the length of the cycle cover (the number of nodes included in this cycle cover). The effectiveness of the cycle is the number of test requirements that are not covered by the test paths already in $\Pi$ but can be covered by the cycle cover when adding

this cycle into Π. Algorithm 7 then splits the super-test requirement into separate complete test paths. Algorithm 4 is used to remove all *redundant* test paths.

In the bipartite graph, the number of the vertices on the left side may be greater than, less than, or equal to the number of vertices on the right. Therefore, a "perfect matching" cannot guarantee that every vertex is incident to one edge, which means some vertices are not reached by the edges in the "perfect matching." Thus, to ensure all the test requirements are covered, the greedy algorithm (line 7) is applied to both the cycle covers and test requirements.

**Algorithm 6** The Matching-based Prefix Graph Algorithm

**Input:** a set of test requirements $TR$
**Output:** a concatenation of paths that covers all the test requirements Π
1: construct a bipartite graph $G$ from the prefix graph;
2: find a perfect matching $M$ of $G$;
3: **repeat**
4: construct a set of cycle covers $C$ from the elements in $M$;
5: **until** no overlap exist between the cycle covers in $C$ and the remaining matchings in $M$
6: add the remaining matchings in $C$;
7: run the greedy algorithm over $C \cup TR$ to select one element $c \in C$ that has the minimum cost-effectiveness until $TR = \emptyset$; during each iteration, extend Π with the selected $c$ and remove $c$ from $TR$;
8: **return** the super-test requirement Π.

### 4.1.4 Splitting the Super-test Requirement into Test Paths

Both the set-covering and prefix-graph based algorithms return a super-test requirement that covers all test requirements. This "super-test requirement" must then be split into individual complete test paths, as shown in Algorithm 7. A small set of test paths $TP$ generated by Algorithm 2 is one of the inputs of this splitting algorithm. The "super-test requirement" is partitioned into several intermediate paths and they are not connected to each other; in other words, the last node of path $p_i$ is not connected to the first node of path $p_{i+1}$ on the graph $G$. If the resulting shorter paths are not test paths, these intermediate paths are extended to reach an initial node and a final node using $TP$ (the same approach applied in Algorithm 3).

The breadth-first search, set-covering based, matching-based prefix graph solutions were

implemented in the graph coverage web application and compared in the experiments in terms of the number of test paths, total nodes in the test paths, and execution time, which will be shown in section 5.1. The experimental results showed that the matching-based prefix graph solution was better than the other two solutions for large graphs. This solution was finally applied to the graph coverage web application. The public APIs of the graph coverage web application are available for other programs to use. The inputs are a general graph $G = (V,\ E)$ and a coverage criterion and the output will be a set of test paths that cover all the test requirements.

**Algorithm 7** The Algorithm to Split a Super-test Requirement into Test Paths

**Input:** A super-test requirement $\Pi$, a small set of test paths $TP$
**Output:** A set of test paths $CTP$

```
1: for each node n_i in Π do
2:    build an empty path p;
3:    repeat
4:       extend p with n_i;
5:    until n_i is not connected to n_{i+1} on the graph
6:    extend this path p such that p becomes a test path using TP and add p to CTP;
7: end for
8: return CTP
```

## 4.2 The Structured Test Automation Language (STAL)

This section presents a structured test automation language (STAL) to automate the transformation from abstract tests to concrete tests in model-based testing. The vending machine example is used to illustrate how testers use STAL to create mappings. This vending machine example differs slightly from the example in section 1.2.2 by allowing only 10 chocolates. Figure 4.1 is a UML state machine diagram for this vending machine example created with the Eclipse Modeling Framework (*EMF*)-based tool *Papyrus* [78]. Figure 4.1 has one initial state, one final state, nine normal states, and 26 transitions. It also includes six constraints that are used as state invariants for states 1-9[1]. Some states have internal transitions. For example, state 2 has an internal transition on *coin*. The comments next to each state document the status of the states and specify which constraints apply to each

[1]Constraints can be specified to be guards or state invariants.

state. For instance, the comment next to state 1 says "credit = 0 and stock = 0", thus, constraints 3 and 6 apply to state 1. Figure 1.3 in section 1.2.2 shows the implementation of the *VendingMachine* class.

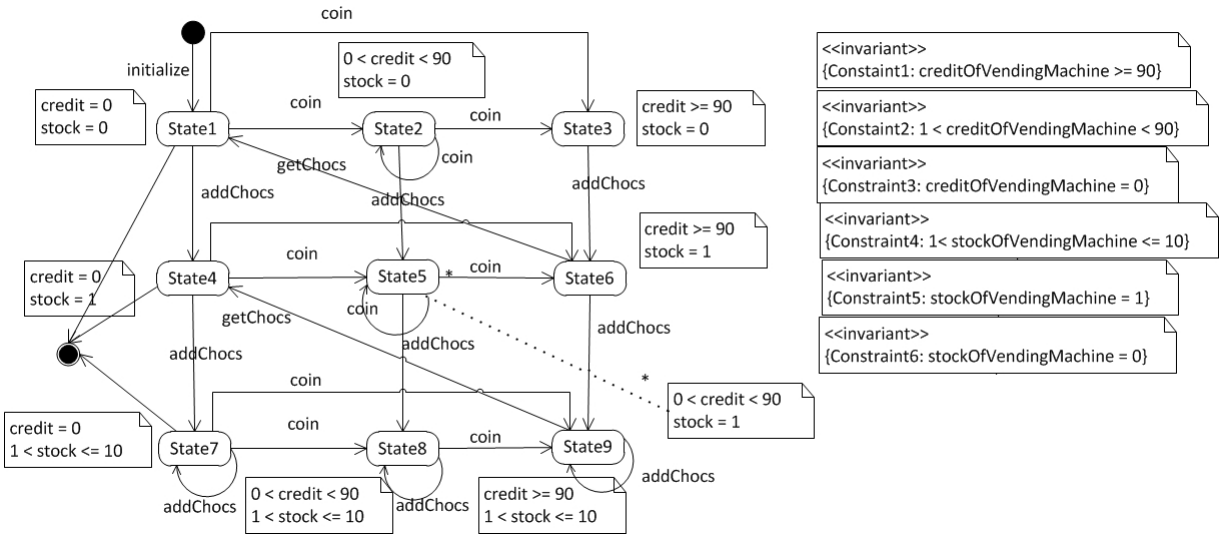


Figure 4.1: A UML State Machine Diagram for the Class VendingMachine

The design of the structured test automation language addressed three key issues: (1) creating mappings and generating test values, (2) graph transformation and test path generation using coverage criteria, and (3) solving constraints and concrete test generation.

The goal of creating mappings is to automate the mapping problem. Once the mappings from identifiable elements in models to executable test code are created, concrete tests can be generated automatically for the same elements in any abstract tests.

Because testers are likely to use UML behavioral diagrams to derive tests, the solution to the mapping problem uses UML state machine diagrams as examples, which were also used in the experiments to evaluate STAL. The UML state machine diagrams need to be transformed to general graphs so that the solutions to the MCTP problem can be applied to them to generate abstract tests.

When abstract tests and mappings are generated, constraints specified in the diagrams have to be satisfied. The constraints are evaluated at run-time. Testers may have to provide

Table 4.1: Which Elements Need Mappings?

| | **Element** |
|---|---|
| Need mappings | Entry Point, Exit Point, and Do Activity of a State, Transition, Constraint |
| Do not need mappings | State Machine, Region, Initial PseudoState, Final State, Fork, Join, Junction, Choice, Simple State, Composite State, Submachine State |
| Not used in test models | Shallow History PseudoState, Deep History PseudoState |

additional mappings to satisfy constraints that cannot be satisfied by the original mappings. Then concrete tests are generated.

To support STAL and new test oracle strategies (solutions to the test oracle problem), a structured test automation language framework (STALE) was developed [79]. This tool can read UML state machine diagrams and transform them into general graphs using the public APIs of the graph coverage web application. Testers can use STALE to create mappings for elements and constraints in models and write test oracles. STALE finally generates concrete tests once the constraints are satisfied by the mappings. Section 4.4 will describe how STALE works. Appendix A gives a complete manual for STAL.

### 4.2.1 Creating Mappings and Generating Test Values

This subsection describes how to create mappings from identifiable elements to executable *Java* code. Table 4.1 summarizes which elements of a UML state machine diagram should be mapped. The first row shows a set of elements that need mappings. The elements of the second row such as *Simple State* and *Composite State* do not need mappings because they can be accessed from *Constraints* and *Transitions* in the first row. The definitions of constraints specify which states the constraints are applied to and the definitions of transitions specify the source states and target states. Although they are not directly mapped to test code, they are used to generate test paths. *Shallow History PseudoState* and *Deep History PseudoState* in the third row do not specify current model behaviors, thus, they are not be used in test models.

STAL defines two kinds of mappings: *element mappings* and *object mappings*. Element mappings directly connect an identifiable element in a UML state machine diagram to test code. For instance, a transition *coin* may be mapped to the test code "*vm.coin (c);*". However, objects and parameters used in this element mapping, such as *vm* (an object of class VendingMachine) and *c* (an int parameter of method *coin (int c)*), also need to be initialized in object mappings, which will be marked as required mappings of this element mapping.

An *element mapping* is formally defined as:

***Mapping*** *mappingName* ***TYPEOFELEMENT*** *nameOfElement*
***Requires*** *objectMappingName ...*
*[**TYPEOFCONSTRAINT** nameOfElement ...] ...*
{*testCode*}

**Mapping** and **Requires** are keywords. The mapping name must be unique. **TYPEOFELEMENT** may vary depending on the actual type of one concrete element. **Transition** and **Constraint** are used in this research. If a mapping uses an object defined in another mapping, the names of additional mappings have to be included in the **Requires** field. The notation "..." means that more than one mapping may be required. If an element is a constraint, the mapping needs to give the type of the constraint (state invariant, guard, etc.) and elements (e.g., states or transitions) in which the constraint is held. Thus, **TYPEOFCONSTRAINT** can be **StateInvariants**, **Guards**, or another type of constraints. StateInvariants and Guards fields are optional and marked by "[ ]" since they are used only for constraints. A constraint may be used as a state invariant in states and guards on transitions at the same time. Test code is required for any mapping and written in curly brackets.

An *object mapping* is formally defined as:

Table 4.2: Attributes of Element and Object Mappings

| **Attributes** | **Element Mapping** | **Object Mapping** |
|---|---|---|
| Element Name | X | |
| Element Type | X | |
| Mapping Name | X | X |
| Test Code | X | X |
| Required Mappings | X | X |
| State Invariants & Guards | X | |
| Object Name | | X |
| Class Name | | X |

***Mapping*** *mappingName* ***Class*** *nameOfClass*

***Object*** *nameOfObject* ***Requires*** *objectMappingName ...*

{*testCode*}

An object mapping includes the class type and name of the object. An object initialization may also need other objects. So an object mapping may require extra object mappings as well. Table A.2 indicates which attributes can be used in element and object mappings.

For the state machine diagram of the vending machine program in Figure 4.1, I need to create mappings for four distinct transitions: *initialize*, *addChocs*, *getChocs*, and *coin*; and define six constraint mappings. The first mapping is an element mapping for the transition *initialize*:

***Mapping*** *vMachineInit* ***Transition*** *initialize*

{ *VendingMachine vm = new VendingMachine();* }

*vMachineInit* is the mapping name. The keyword **Transition** specifies that the mapping *vMachineInit* is created for a transition.

Next is a mapping for the transition *getChocs*. The method *getChoc (StringBuffer)* is used to get chocolates from the vending machine. The *StringBuffer* object "sb" represents a chocolate.

***Mapping*** *getChocolate* ***Transition*** *getChocs*

{

*StringBuffer sb = new StringBuffer ("MM");*

*vm.getChoc (sb);*

}

The mapping *getChocolate* gets only one chocolate from the vending machine. More chocolates can be taken from the vending machine if the method *getChoc (StringBuffer)* is called multiple times. Two objects, *vm* and *sb*, need to be initialized before this transition is taken. Because the transition *initialize* appears at the beginning of every test path, the object *vm* will be initialized before the test code for other mappings is run, thus, other mappings do not need to specify the mapping *vMachineInt* to be required. A *StringBuffer* variable *sb* is initialized directly in the test code of this mapping. Alternatively, the initialization of *sb* could be defined in an object mapping and reused in other mappings. The next example shows another mapping that gets two chocolates. It requires an object mapping *stringBufferInit*.

***Mapping*** *getTwoChocolates* ***Transition*** *getChocs*

***Requires*** *stringBufferInit*

{

*vm.getChoc (sb);*

*vm.getChoc (sb);*

}

The object mapping for *stringBufferInit* is shown below:

***Mapping*** *stringBufferInit* ***Class*** *StringBuffer* ***Object*** *sb*

{ *StringBuffer sb = new StringBuffer ("MM");* }

Note that the initialization of an object should be either embedded in the test code of

an element mapping, defined as an object mapping separately, but not both. Otherwise the object will be defined twice. STALE will report such error to testers. The examples above show that testers could generate multiple mappings with different test values for the same transition (mappings *getChocolate* and *getTwoChocolates* for the transition *getChocs*). Which mapping should be chosen depends on the constraint evaluation, which is discussed in section 4.2.3. The next mapping, *coinOneDollarAndTen*, provides test code for the transition *coin*.

```
Mapping coinOneDollarAndTen () Transition coin
{
  vm.coin (10);
  vm.coin (25);
  vm.coin (25);
  vm.coin (25);
  vm.coin (25);
}
```

This mapping adds $1.10 to the vending machine. The method *coin* is used to add credits in the vending machine. The *int* parameter of the method *coin* represents how much is added. If adding a different amount of money, testers have to create a new mapping. Alternatively, the testers can create a *parameterized mapping* to avoid writing multiple mappings.

```
Mapping coinAnyCredit (int c) Transition coin
    requires cForCoin
{ vm.coin (c); }
```

Testers can provide multiple test values for primitive types and values will be chosen arbitrarily. To provide test values for the parameter *c* in an object mapping, an object mapping *cForCoin* can be written below:

```
Mapping cForCoin Class int Object c
{ 10, 25, 100 }
```

The vending machine only accepts dimes (10), quarters (25), and dollars (100). One of the three *int* values will be selected arbitrarily for the parameter *c*, potentially with a different value each time *cForCoin* is used. Testers can also provide predicates such as {c > 0, c ≤ 100}, separating conditions by commas. Constraint solvers for numeric and string variables [80, 81] used in STALE will return a value that satisfies all constraints. The constraint solvers have a limited language. They accept numeric variables (*int*, *float*, and *double*), arithmetic operators, and regular expressions for *String*s. They do not accept disjuncts or function calls.

A mapping that specifies a constraint to be a state invariant is shown below. In this example, Constraint1 is a state invariant for State3, State6, and State9. The credit of the vending machine has to be equal to or greater than 90 cents in these three states. The test code for this constraint is evaluated when tests go through a state to which the constraint is applied. If the boolean expression evaluates to false, the mapping prior to this state does not satisfy this constraint and an alternative mapping needs to be used. This constraint can also be explicitly expressed as test oracles in concrete tests.

***Mapping*** *constraintForCredit* ***Constraint*** *Constraint1*
***StateInvariants*** *State3, State6, State9*
{ *vm.getCredit() ≥ 90;* }

### 4.2.2 Graph Transformation and Test Path Generation

Each UML state machine diagram can be transformed to a generic graph with initial and final nodes. For a UML state machine diagram, an initial state is mapped to an initial node in the graph, a final state to a final node, and other states to unique nodes. Each transition becomes an edge and an internal transition from one state to itself becomes a self-loop on the corresponding node. Elements including composite states, choices, forks, junctions, and joins need special treatment.

When transforming multiple transitions from a state to another state (that is, more

than one transition has the same source state and target state), there are two general ways. First, transform each transition to a different edge. Thus, each edge represents a different transition. Second, transform all transitions between two states to only one edge, which represents all the transitions. This research chose the latter because when a UML state machine diagram is transformed to a general graph, this graph only uses node numbers to represent edges (such as (1, 2), where 1 and 2 are node numbers). Thus, without using additional syntax, creating different edges for different transitions from the same source to the same target is ambiguous on the graph. The problem of the latter solution is that covering all edges on the graph is not equivalent to covering all transitions on the diagram. Thus, to cover all the transitions, mappings need to be created to map the edges to each transition. Testers need to use *all-transition* coverage [1] to ensure every transition is covered. In this research, the experimental subjects only have single transitions from one state to another. Therefore, I used edge coverage on the graph to cover all transitions on the state machine diagrams.

Each sub-state of a composite state becomes a unique node and the composite state itself will not be transformed to a node. If a composite state has an initial state, an edge will be created for an incoming transition to the initial state of the composite state. If a composite state has $n$ regular sub-states but no initial states, $n$ edges will be created for an incoming transition, one for each node. Likewise, if a composite state has a final state, an edge will be created for an outgoing transition from the final state of the composite state. If a composite state has $n$ sub-states but no final states, $n$ edges will be created for an outgoing transition.

An edge will be created for each outgoing transition of a choice or fork. An edge will be created for each incoming transition of a join or junction.

Once the transformation from a UML state machine diagram to a generic graph is complete, testers can use the algorithms used to solve the MCTP problem from section 4.1 to generate test paths, according to some coverage criterion.

### 4.2.3 Solving Constraints and Concrete Test Generation

After mappings are created, the structured test automation framework (STALE) saves the mappings as *XML*. Figure 4.2 shows the saved mappings *vMachineInit*, *addChocolate*, *coinOneDollar*, *coinAnyCredit*, *intCInit*, *getOneChocolate*, *stringBufferInit*, and *constraintStockOne* in *XML*.

Seven test paths are generated to satisfy edge coverage using the graph web application [12] on the diagram of Figure 4.1. One of the test paths is [*initial*, *state1*, *state4*, *state7*, *state7*, *state9*, *state4*, *final*], whose abstract test is shown below:

*initialize;*
*addChocs;*
*addChocs;*
*addChocs;*
*coin;*
*getChocs;*

Testers can use the mapping *addChocolate* in Figure 4.2 for the transition *addChocs* since only one mapping is created for this transition. If a transition has more than one mapping, the tool has to choose which to use. If the destination state of a transition has a constraint, the constraint has to be satisfied by the selected mapping. If the constraint is not satisfied, another mapping will be selected. If none of the mappings can satisfy the constraint, the tester is informed. This represents an error, and should result in a correction to the model, the program, or the mappings.

An object or element mapping may require more than one object mapping. STALE is able to analyze the dependency relationship among all related object mappings. While generating concrete tests, the test code of object mappings that have no dependencies will be written first, followed by other object mappings that use variables from prior mappings.

When executing the example test path above, the vending machine will reach State9 with three chocolates in stock and *credit* no less than 90. The next step in the abstract test is *getChocs*, which should cause a transition to State4. However, State4 has Constraint5 (specified in Figure 4.2), which says that the vending machine should only have one chocolate

in stock. There is no way to satisfy that constraint (*getChocs* transition can only give one chocolate to customers at one time), so an error will be reported to the tester. The error can be corrected in one of several ways. The model can be changed by modifying the constraint to be $stockOfVendingMachine \geq 1$, adding a transition on *getChocs* to State7, or by changing *getChocs* to allow more than one chocolate to be dispensed. The ability to find errors in the model when generating tests is a major benefit of this approach.

## 4.3 Test Oracle Strategies

This section introduces ten new test oracle strategies (OSes) and two baseline OSes: the *null* test oracle strategy (NOS) and *state invariant* test oracle strategy (SIOS).

NOS only checks for exceptions or abnormal termination, as implicitly provided by Java runtime systems [9]. Many faults do not cause runtime exceptions, so this OS sounds trivial. It is, however, often used in industry.

The second OS is SIOS, which checks the state invariants from the state machine diagram. After testers use STALE to provide proper test mappings from abstract model elements to concrete test code, all state invariants in the state machine diagram should be satisfied. Since all the state invariants can be transformed to executable code using the provided mappings, these state invariants can be added to the tests as assertions automatically. Thus, SIOS is more precise than NOS.

This research considered two dimensions when designing OSes: how often to check states (*frequency*), and how many output values and internal state variables to check (*precision*). Regarding the frequency, testers can check states after each method call, each transition, or they can check states only once. For precision, this research defined four elements of the program state to check:

1. *State invariants*: Check the state invariants in the model
2. *Object members*: Check member variables of objects that call methods in a transition
3. *Return values*: Check return values for each invoked method

4. *Parameter members*: Member variables of objects that are passed to a method call as parameters

"Deep checking" was used for objects, that is, if an object's member was also an object, it was checked recursively until primitive variables were found.

I proposed ten new OSes beyond SIOS. Each new OS satisfies all the state invariants in a model and explicitly writes the satisfied state invariants as assertions. Additionally, they check more program states.

**OS1:** After each transition is executed, check all distinct object members in this transition only once

**OS2:** After each transition is executed, check the return values of the distinct methods only once

**OS3:** After each transition is executed, check all distinct object members in this transition and the return values of the distinct methods only once

**OS4:** After each transition is executed, check all distinct parameter members and the return values of the distinct methods only once

**OS5:** After each transition is executed, check all distinct object and parameter members and the return values of the distinct methods only once

For each $OSi$ $(1 \leq i \leq 5)$, $OTi$ $(1 \leq i \leq 5)$ checks the same outputs and internal state variables (one check per test) but with a lower frequency. $OT$ stands for **O**ne **T**ime checking test oracle strategy. Thus, $OTi$ $(1 \leq i \leq 5)$ is an $OS$, which still represents a test oracle strategy. OSes mentioned elsewhere contain $OTi$ and $OSi$ $(1 \leq i \leq 5)$.

**OT1:** After the last transition, check all distinct object members that appear in all transitions only once

**OT2:** After the last transition, check the return values of the distinct methods that appear in all transitions only once

**OT3:** After the last transition, check all distinct object members and the return values of the distinct methods that appear in all transitions only once

**OT4:** After the last transition, check all distinct parameter members and the return values of the distinct methods that appear in all transitions only once

**OT5:** After the last transition, check all distinct object and parameter members and the return values of the distinct methods that appear in all transitions only once

OS5 is not the most precise test oracle strategy possible because it does not check outputs and variables whenever they appear in the tests. I wanted to see if each OT can be as good as the corresponding OS in terms of finding faults. Please note that each OT checks the object members, return values, and parameter members that appear in all transitions, not just outputs and internal state variables in the last transition.

Checking all object members, return values, and parameter members that appear in all the transitions at the end of tests could cause the $OTi$s to check different program states from the $OSi$s. For instance, if OS2 checks a return value *Object a* after the system initialization in a test ($a$ has an initial value during the system initialization), then OS2 does not check $a$ in the rest of the tests because $a$ is not used as a return value in other transitions. However, $a$'s state could be changed if $a$ is used as a parameter or makes method calls. Therefore, OT2 can check different program states (check $a$ after the last transition) that would not be checked by OS2 (check $a$ after the first transition). Moreover, no matter when $a$'s status is changed, OS5 is able to monitor the change since OS5 checks object members, return values and parameter members for every transition. Thus, OS5 is expected to be no less effective than $OTi$, where $1 \leq i \leq 5$. So $OTi$ is as precise as $OSi$ but may detect faults that cannot be revealed by $OSi$, where $1 \leq i \leq 4$.

Test oracle strategies OT1, OT2, OT3, OT4, and OT5 mimic a common programmer habit of writing assertions (checking program states at the end of tests). The difference is that programmers often only write a few assertions, while OT1, OT2, OT3, OT4, and OT5 check various outputs and internal state variables after the last transition.

Figure 4.3 shows the precision relationships among the OSes. An arrow from one OS to another indicates that the higher OS is more precise.

Section 5.3 presents experiments that compared the new ten OSes (OS1, OS2, OS3, OS4, OS5, OT1, OT2, OT3, OT4, and OT5) with two baseline OSes (NOS and SIOS). The analysis and recommendations is also discussed.

## 4.4 The Structured Test Automation Language Framework (STALE)

The structured test automation language framework (STALE) implemented this research's solutions to the mapping problem and test oracle problem and used the solution to the MCTP problem from the graph coverage web application. The main function of STALE is to generate concrete tests from models. STALE uses the Eclipse Modeling Framework (EMF) library [64] to read EMF-based UML models and transforms the models to generic graphs. STALE uses a prefix-graph based solution [47] from public APIs of the graph coverage web application to reduce the number of tests as well as the number of times transitions appear in the tests, which is considered to be a type of *quantitative human oracle cost reduction* [2]. The quantitative human oracle cost reduction reduces the effort of generating test oracles by decreasing the size of test cases or test suites.

Testers develop the mappings in STAL, which are saved in XML files. Testers also need to provide test oracles for each mapping to generate assertions in the tests. Then testers need to choose a level of test oracle (e.g., OS2) and a test coverage criterion (e.g., edge coverage). Finally, the tool generates concrete tests by choosing mappings that satisfy constraints. If a mapping does not satisfy the constraints, unsatisfied constraints will be reported. Thus, testers need to generate additional mappings.

Figure 4.4 shows the user interface of STALE. If some projects have already been created under STALE, they are shown in the *Available Projects* when users start STALE. The associated models, mappings, and test oracles are populated in their corresponding fields.

If no STALE projects exist, users can create a new project by providing a name and model.

Once a model (only UML state machine diagrams so far) is added, its identifiable elements such as transitions and constraints (only state invariants so far) are placed in the *element* box. Then users should provide mappings from identifiable elements to implementation. Each identifiable element from a diagram can have more than one mapping because testers need to provide as many mappings as possible to satisfy all the state invariants for a specific coverage criterion. Each mapping for an element only needs to be written once. When an element appears again in an abstract test, an appropriate mapping is selected automatically to satisfy the necessary state invariants. The concrete code of a mapping for a transition is a sequence of method calls.

The concrete test code for state invariants can be transformed to JUnit assertions, allowing each assertion to be evaluated at run-time. If an assertion evaluates to false, it means the state invariant is not satisfied by the concrete test sequences of the currently used mapping for a transition between this state and a preceding state. Therefore, the concrete test code of another mapping for the transition will be used and the state invariant is re-evaluated. This process continues until the state invariant is satisfied. If no existing mapping can satisfy a state invariant, STALE reports an error and asks the tester to provide more mappings.

Since the concrete test code of state invariants can be evaluated as JUnit assertions, the assertions can be used directly as test oracles. This is what SIOS evaluates. Additionally, testers can use STALE to write more assertions to check other internal state variables such as class variables and outputs. For instance, if the executable test code of a transition has a method call: "*boolean sign = classObjectA.doActionB();*", testers can write assertions to evaluate the return value of the method call *sign* and *classObjectA*'s class member variables by providing expected test values.

I will show an example below to explain how STALE works. Since a tester only needs to load models and select a coverage criterion to generate abstract tests, this example focuses on how to create mappings and test oracles for generating concrete tests. Recall

the complete vending machine example Figure 4.1 in section 4.2: customers insert coins to purchase chocolates; only dimes, quarters, and dollars are accepted; and the price for all chocolate is 90 cents. Focus on the top three states (state1, state2, and state3) and associated transitions, which describes how the system behaves when customers insert coins. Thus, *Constraint1* is in *State1*, *Constraint2* is in *State2*, and *Constraint3* is in *State3*. This example still uses the same implementation that is described in Figure 1.3.

STALE can read the state machine diagram and generate abstract tests. If testers choose edge coverage (EC), then the test requirements for those top three states are "State1, coin, State2," "State2, coin, State3," and "State1, coin, State3". Thus, they need to be covered by the abstract tests. Testers need to provide mappings so the abstract tests can become executable code. To satisfy state invariants *Constraint1*, *Constraint2*, and *Constraint3*, testers may have to provide multiple mappings for the transition *coin*. The concrete test code of one mapping, *coinTen*, can be "vm.coin(10);", which inserts a dime into the vending machine. *vm* is an object of class *VendingMachine*, and is defined in the mapping *vMachineInit* in Figure 4.2. STALE provides a mechanism to let other mappings use this object. To satisfy the state invariants in *State2* and *State3*, testers can provide another mapping whose test code is "vm.coin(100);", which inserts a dollar.

In addition to the state invariants, testers can also provide more test oracle data to check other fields of class *VendingMachine*. STALE provides ten levels for ten different test oracle strategies. Thus, testers need to provide test oracle data that check different program states for different test oracle levels, for each mapping. When a mapping is used in a test, the corresponding test oracles have to be inserted into the test after the test code of this mapping. Thus, this step is automated. Because testers do not know where the mapping may appear in tests (depend on the coverage criteria, constraints, test values, and algorithms), testers cannot write fixed test values in test oracles. If the test code of a mapping for transition "coin" is *vm.coin(10);*, an assertion after this transition like "assertEquals(10, vm.getCredit());" could return false in many tests since the credit before adding the dime is not 0. Therefore, to automate the generation of test oracles, the test

code of element mappings have to be changed. Figure 4.4 shows an example using transition "coin." The mapping *coinTen* adds two statements "credit = vm.getCredit();" and "stock = vm.getStock().size();" to get the values of *credit* and *stock* before inserting a dime into the vending machine. Note that variables *credit* and *stock* need to be declared in the test initialization to avoid duplicated variable definitions. Then the code of test oracles for the mapping *coinTen* can check whether the assertions return true: "assertEquals(credit + 10, vm.getCredit())" and"assertEquals(stock, vm.getStock().size())".

When generating concrete tests, testers need to choose a test coverage criterion and a test oracle level. Testers also need to add package names and imported package and class declarations if necessary. Once these are prepared, testers press the *Generate tests* button and the tests will be generated in the "test" folder under the project directory.

Figure 4.2 show some example mappings. Appendix B shows the vending machine example completely worked out, including abstract tests, its implementation, and all mappings. The concrete tests would take up to 300 pages, so are available online.

⟨ *mappings* ⟩
⟨ *mapping* ⟩
⟨ *name* ⟩ ***vMachineInit*** ⟨ */name* ⟩
⟨ *transition-name* ⟩ ***i****nitialize* ⟨ */transition-name* ⟩
⟨ *code* ⟩ ***VendingMachine vm = new VendingMachine();*** ⟨ */code* ⟩
⟨ */mapping* ⟩
⟨ *mapping* ⟩
⟨ *name* ⟩ ***addChocolate*** ⟨ */name* ⟩
⟨ *transition-name* ⟩ ***addChocs*** ⟨ */transition-name* ⟩
⟨ *code* ⟩ ***vm.addChoc ("MM");*** ⟨ */code* ⟩
⟨ */mapping* ⟩
⟨ *mapping* ⟩
⟨ *name* ⟩ ***coinOneDollar*** ⟨ */name* ⟩
⟨ *transition-name* ⟩ ***coin*** ⟨ */transition-name* ⟩
⟨ *code* ⟩ ***vm.coin(100);*** ⟨ */code* ⟩
⟨ */mapping* ⟩
⟨ *mapping* ⟩
⟨ *name* ⟩ ***coinOneDime*** ⟨ */name* ⟩
⟨ *transition-name* ⟩ ***coin*** ⟨ */transition-name* ⟩
⟨ *code* ⟩ ***vm.coin(10);*** ⟨ */code* ⟩
⟨ */mapping* ⟩
⟨ *mapping* ⟩
⟨ *name* ⟩ ***getOneChocolate*** ⟨ */name* ⟩
⟨ *transition-name* ⟩ ***getChocs*** ⟨ */transition-name* ⟩
⟨ *code* ⟩ ***vm.getChoc(sb);*** ⟨ */code* ⟩
⟨ *required-mappings* ⟩ ***stringBufferInit*** ⟨ */required-mappings* ⟩
⟨ */mapping* ⟩
⟨ *mapping* ⟩
⟨ *name* ⟩ ***stringBufferInit*** ⟨ */name* ⟩
⟨ *object-name* ⟩ ***sb*** ⟨ */object-name* ⟩
⟨ *class-name* ⟩ ***StringBuffer*** ⟨ */class-name* ⟩
⟨ *code* ⟩ ***StringBuffer sb = new StringBuffer(s);*** ⟨ */code* ⟩
⟨ *required-mappings* ⟩ ***stringInit*** ⟨ */required-mappings* ⟩
⟨ */mapping* ⟩
⟨ *mapping* ⟩
⟨ *name* ⟩ ***stringInit*** ⟨ */name* ⟩
⟨ *object-name* ⟩ ***s*** ⟨ */object-name* ⟩
⟨ *class-name* ⟩ ***String*** ⟨ */class-name* ⟩
⟨ *code* ⟩ ***"MM"*** ⟨ */code* ⟩
⟨ */mapping* ⟩
⟨ *mapping* ⟩
⟨ *name* ⟩ ***constraintStockOne*** ⟨ */name* ⟩
⟨ *constraint-name* ⟩ ***Constraint5*** ⟨ */constraint-name* ⟩
⟨ *code* ⟩ ***vm.getStock().size() == 1;*** ⟨ */code* ⟩
⟨ *required-mappings* ⟩ ***vMachineInit*** ⟨ */required-mappings* ⟩
⟨ *state-invariant* ⟩ ***State4,State5,State6*** ⟨ */state-invariant* ⟩
⟨ */mapping* ⟩
⟨ */mappings* ⟩

Figure 4.2: Mappings

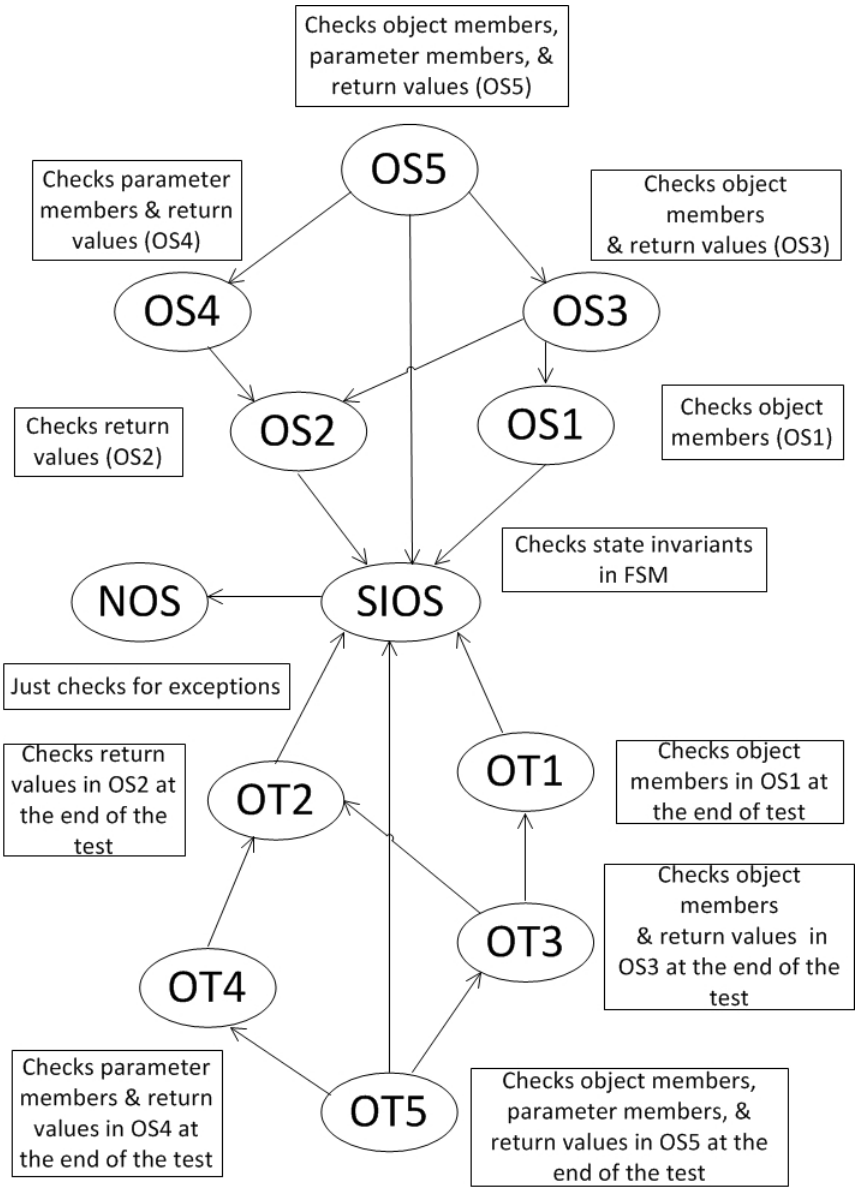


Figure 4.3: Precision Relationship among Test Oracle Strategies

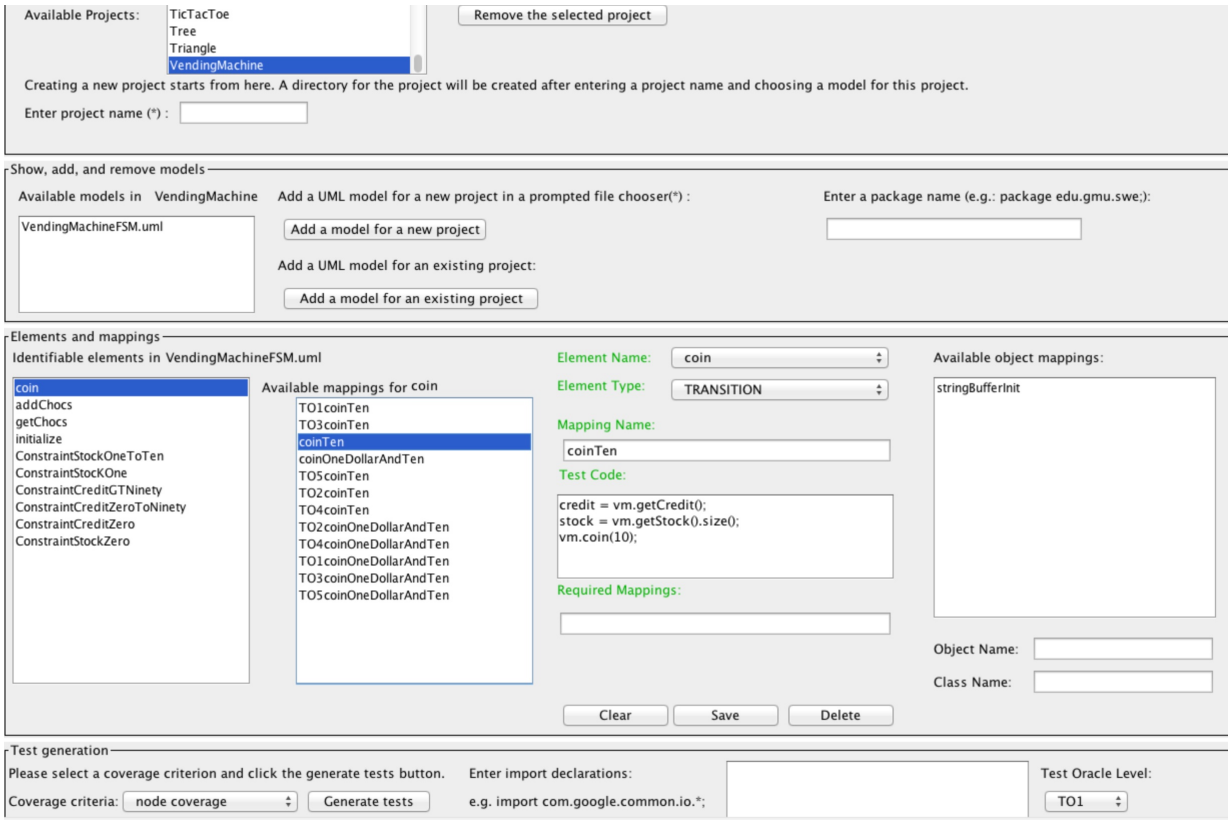


Figure 4.4: The User Interface of the Structured Test Automation Language Framework

# Chapter 5: Experiments

This chapter describes the experiments for the three sub-problems of generating cost-effective criteria-based tests from behavioral models: the minimum cost test paths problem (MCTP), the mapping problem, and the test oracle problem. The experiments in section 5.1 evaluated two new solutions (set-covering and matching based prefix graph) to the MCTP problem against the previously used breadth-first search solution [47]. The experiments in section 5.2 compared the test mapping process with the structured test automation language (STAL) with manual transformation from abstract tests to concrete tests [14, 76]. The experiments in the last section, 5.3, validated the new test oracle strategies (OSes) against two baseline test oracle strategies: the null (NOS) and state invariant (SIOS) test oracle strategies [77].

## 5.1 The Experiments for the Minimum Cost Test Paths Problem

The experiments compared the set-covering based and matching-based prefix graph solutions with the previous breadth-first search solution. The new solutions were expected to generate fewer tests and use fewer total nodes to cover all test requirements when compared with the previous breadth-first search solution. The solutions developed to solve MCTP in this research apply to arbitrary graphs, without consideration of what software artifact they came from. Model-based testing designs tests from functional requirements or design models, and normally excludes source. Since this research considers graphs in the abstract and the origin of the graphs is irrelevant, the experimental verification chose control flow graphs from source for convenience. The results apply to any test design strategy that uses graphs.

The experiments used prime path coverage as the coverage criterion.

The three solutions described in section 4.1 were studied using 37 methods in two steps. First, 18 methods were taken from several open source projects, including *Joda-Time*[1], *javassist*[2], *graph coverage web application*[3], *Apache Ant*[4], and *jdom*[5]. All methods were written in Java. Methods that had relatively complicated control structures (nested loops) were chosen, because simple methods have fewer prime paths and thus the algorithm used to create test paths is less important. Control flow graphs were generated by hand for each method and the graph coverage web application (support software provided as part of Ammann and Offutt's testing book [12]) was used to generate prime paths (test requirements).

The hypothesis was that the better algorithms would show more benefit for graphs that have more complex structures especially nested loops because such complex structures could result in prime paths that have overlaps among them. So second, another 19 methods from the open source projects were modified structurally to yield control flow graphs that had more complex structures. Modifications included adding a nested loop, adding a *continue* statement, deleting a statement, and adding additional decision structures. Note that the newly changed methods were still valid Java files without compilation errors.

The three solutions described in section 4.1 were implemented (in Java) in the graph coverage web application. Then the prime paths for all 37 methods were the inputs to each of the three solutions, resulting in test paths that toured all prime paths.

### 5.1.1 Experimental Environment and Process

The experiment was conducted on a Dell mini tower, with an Intel Core i5-750 (2.66GHz 8MB L3 cache) CPU. The primary measure of the experimental subjects' size is the number of prime paths, which was determined by the graph coverage web application. The program was executed without interactions with the Internet or interruption from other programs.

[1] http://joda-time.sourceforge.net/
[2] http://www.jboss.org/javassist/
[3] http://cs.gmu.edu:8080/offutt/coverage/GraphCoverage
[4] http://ant.apache.org/
[5] http://www.jdom.org/

The goals of MCTP to be optimized include the total number of test paths, the total number of nodes, and the maximum ratio of TR (number of test requirements) to TP (number of test paths), which has been described in section 1.3.1. Therefore, the three solutions were compared based on the number of test paths, the total number of nodes included in the test paths, and the maximum ratio of TR over TP. In this research, the set-covering based and matching-based prefix graph algorithms are general solutions to the MCTP problem, not to specific MCTP variants. Therefore, specific goals such as the maximum ratio of TR / TP were not be minimized. Thus, the maximum ratio of TR / TP was measured but not used as a metric to evaluate the solutions.

This research did not emphasize time because time depends on many factors such as people, tools, test criteria, and experimental subjects. However, the three solutions were compared in terms of the execution time as well for completeness. For each test path, TP is counted as 1. Thus, the ratio of TR to TP is the number of test requirements covered by the test path.

The maximum ratio of TR over TP finds the most test requirements covered by any test path for each program. Execution time was measured with the Java built-in method *System.nanoTime()*.

To handle variance in measuring execution time, each subject was measured five times for the three solutions and the average (mean) of the measured time was computed.

### 5.1.2 Experimental Results

Table 5.1 shows results from the experiment on natural open source methods. The table shows the number of prime paths (*PPs*), then results in terms of the number of test paths (*Paths*), and the total number of nodes in all test paths (*Nodes*). The *BF* column shows results from the breadth-first search solution, *PG* denotes the prefix-graph based solution, and *SC* represents the set-covering based solution. These methods had from 9 to 1844 prime paths.

Table 5.1: Paths and Nodes for Open Source Methods

| PPs | Paths | | | Nodes | | | Paths Ratio | | | Nodes Ratio | | | TR/TP | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | **BF** | **PG** | **SC** | **BF** | **PG** | **SC** | **BF** | **PG** | **SC** | **BF** | **PG** | **SC** | **BF** | **PG** | **SC** |
| 9 | 7 | 5 | 5 | 46 | 33 | 33 | 1.00 | 0.71 | 0.71 | 1.00 | 0.72 | 0.72 | 2 | 3 | 3 |
| 11 | 9 | 7 | 8 | 54 | 42 | 48 | 1.00 | 0.78 | 0.89 | 1.00 | 0.78 | 0.89 | 2 | 3 | 3 |
| 27 | 14 | 12 | 12 | 210 | 184 | 177 | 1.00 | 0.86 | 0.86 | 1.00 | 0.88 | 0.84 | 9 | 16 | 17 |
| 27 | 17 | 15 | 17 | 160 | 141 | 166 | 1.00 | 0.88 | 1.00 | 1.00 | 0.88 | 1.04 | 3 | 9 | 8 |
| 35 | 25 | 25 | 22 | 229 | 225 | 200 | 1.00 | 0.96 | 0.88 | 1.00 | 0.98 | 0.87 | 6 | 7 | 8 |
| 38 | 18 | 13 | 17 | 196 | 155 | 181 | 1.00 | 0.72 | 0.94 | 1.00 | 0.79 | 0.92 | 8 | 17 | 12 |
| 46 | 30 | 24 | 27 | 366 | 304 | 339 | 1.00 | 0.80 | 0.90 | 1.00 | 0.83 | 0.93 | 6 | 10 | 10 |
| 63 | 36 | 22 | 26 | 576 | 415 | 441 | 1.00 | 0.61 | 0.72 | 1.00 | 0.72 | 0.77 | 9 | 16 | 14 |
| 69 | 49 | 42 | 44 | 536 | 465 | 485 | 1.00 | 0.86 | 0.90 | 1.00 | 0.87 | 0.90 | 7 | 11 | 11 |
| 71 | 37 | 24 | 25 | 699 | 492 | 484 | 1.00 | 0.65 | 0.68 | 1.00 | 0.70 | 0.69 | 12 | 19 | 16 |
| 84 | 70 | 50 | 52 | 792 | 618 | 621 | 1.00 | 0.71 | 0.74 | 1.00 | 0.78 | 0.78 | 7 | 13 | 13 |
| 98 | 65 | 57 | 56 | 1086 | 965 | 947 | 1.00 | 0.87 | 0.86 | 1.00 | 0.89 | 0.87 | 8 | 12 | 12 |
| 101 | 66 | 50 | 52 | 984 | 788 | 771 | 1.00 | 0.76 | 0.79 | 1.00 | 0.80 | 0.78 | 11 | 18 | 11 |
| 122 | 81 | 70 | 63 | 1685 | 1521 | 1393 | 1.00 | 0.86 | 0.78 | 1.00 | 0.90 | 0.83 | 13 | 14 | 14 |
| 170 | 105 | 91 | 93 | 2176 | 1921 | 1927 | 1.00 | 0.87 | 0.89 | 1.00 | 0.88 | 0.89 | 9 | 26 | 13 |
| 1,024 | 913 | 776 | 623 | 26,285 | 24,005 | 21,398 | 1.00 | 0.85 | 0.68 | 1.00 | 0.91 | 0.81 | 15 | 26 | 19 |
| 1,141 | 982 | 913 | 761 | 39,370 | 37,267 | 32,934 | 1.00 | 0.93 | 0.78 | 1.00 | 0.95 | 0.84 | 16 | 29 | 19 |
| 1,844 | 1,265 | 395 | 376 | 28,278 | 12,875 | 11,607 | 1.00 | 0.31 | 0.30 | 1.00 | 0.46 | 0.41 | 14 | 273 | 77 |
| **Total** | **3,789** | **2,590** | **2,279** | **103,728** | **82,416** | **74,152** | | | | | | | | | |
| **Weighted Average** | | | | | | | **1.00** | **0.68** | **0.60** | **1.00** | **0.79** | **0.71** | | | |

The next six columns show the ratio of path length and number of nodes for each solution over the breadth-first search solution. The bottom row shows the total cost of the paths and nodes, and the average ratios. The average ratios were weighted, that is, they were computed from the totals row. The last three columns (TR / TP) display the maximum ratio of TR to TP for the three solutions.

The columns *Paths* and *Nodes* show that the prefix-graph and the set-covering based solutions generated the same or fewer test paths and nodes when compared with the breadth-first search solution, and for most methods, far fewer. The *BF*, *PG*, and *SC* under *Paths* columns show that the prefix-graph based solution generated fewer test paths than the set covering solution for most methods, but the set covering solution generated fewer for the three largest methods. The largest number of test requirements covered by one test path is 273 in the last subject, using the prefix-graph based solution. For the third subject, the set-covering solution covered 17 of the 27 test requirements (63%) with just one test path.

Table 5.2 presents the measured execution time of the three solutions for each subject in the open source programs in seconds. The meanings of *PPs*, *BF*, *PG*, *SC*, *Total*, and *Average* are the same as in Table 5.1. The last three columns show the ratios of the running time of each solution over the breadth-first search solution. The prefix-graph based solution took the least time to generate the test paths for each subject. For the first 15 subjects, the set-covering based solution ran faster than the breadth-first search solution; however, the breadth-first search solution was faster for the last three subjects, which have more than 1000 prime paths. The last subject took almost 10 hours to generate tests with the set-covering based solution. The unweighted and weighted averages were quite different so this table shows both. The unweighted averages were straight averages of the numbers in the column, and the weighted averages were computed from the totals. With execution speed in particular, the weighted averages were heavily influenced by the last subject.

Table 5.3 shows the results of the test paths from the experiment on the 19 modified methods. The results were consistent with those in the first stage.

Table 5.2: Execution Time for Open Source Methods

| PPs | Time (seconds) | | | Time Ratio | | |
|---|---|---|---|---|---|---|
| | **BF** | **PG** | **SC** | **BF** | **PG** | **SC** |
| 9 | 0.002 | 0.0007 | 0.0004 | 1.00 | 0.35 | 0.20 |
| 11 | 0.004 | 0.0007 | 0.0004 | 1.00 | 0.18 | 0.10 |
| 27 | 0.048 | 0.0170 | 0.0160 | 1.00 | 0.35 | 0.33 |
| 27 | 0.023 | 0.0130 | 0.0030 | 1.00 | 0.57 | 0.13 |
| 35 | 0.035 | 0.0160 | 0.0230 | 1.00 | 0.46 | 0.66 |
| 38 | 0.045 | 0.0160 | 0.0190 | 1.00 | 0.36 | 0.42 |
| 46 | 0.064 | 0.0230 | 0.0370 | 1.00 | 0.36 | 0.58 |
| 63 | 0.140 | 0.0360 | 0.1200 | 1.00 | 0.26 | 0.86 |
| 69 | 0.200 | 0.0390 | 0.1300 | 1.00 | 0.20 | 0.65 |
| 71 | 0.230 | 0.0500 | 0.1800 | 1.00 | 0.22 | 0.78 |
| 84 | 0.330 | 0.0660 | 0.2900 | 1.00 | 0.20 | 0.88 |
| 98 | 0.670 | 0.0870 | 0.3600 | 1.00 | 0.13 | 0.54 |
| 101 | 0.330 | 0.0660 | 0.2900 | 1.00 | 0.20 | 0.88 |
| 122 | 1.070 | 0.1600 | 0.9600 | 1.00 | 0.15 | 0.90 |
| 170 | 3.610 | 0.3500 | 2.4400 | 1.00 | 0.10 | 0.68 |
| 1024 | 1340.000 | 142.0000 | 4040.0000 | 1.00 | 0.11 | 3.01 |
| 1141 | 3490.000 | 184.0000 | 5680.0000 | 1.00 | 0.05 | 1.63 |
| 1844 | 1680.000 | 235.0000 | 35,500.0000 | 1.00 | 0.14 | 21.13 |
| **Total** | **6516.801** | **561.9404** | **45,224.8400** | | | |
| | | | **Unweighted Average** | **1.00** | **0.24** | **0.56** |
| | | | **Weighted Average** | **1.00** | **0.09** | **6.94** |

Table 5.3: Paths and Nodes for Modified Methods

| PPs | Paths | | | Nodes | | | Paths Ratio | | | Nodes Ratio | | | TR/TP | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | BF | PG | SC | BF | PG | SC | BF | PG | SC | BF | PG | SC | BF | PG | SC |
| 26 | 10 | 5 | 8 | 150 | 117 | 125 | 1.00 | 0.50 | 0.80 | 1.00 | 0.78 | 0.83 | 10 | 15 | 11 |
| 35 | 15 | 10 | 11 | 198 | 155 | 153 | 1.00 | 0.67 | 0.73 | 1.00 | 0.78 | 0.77 | 9 | 13 | 14 |
| 62 | 35 | 21 | 26 | 476 | 369 | 385 | 1.00 | 0.60 | 0.74 | 1.00 | 0.78 | 0.81 | 9 | 13 | 11 |
| 63 | 41 | 24 | 29 | 625 | 456 | 511 | 1.00 | 0.59 | 0.70 | 1.00 | 0.73 | 0.82 | 7 | 18 | 9 |
| 68 | 37 | 20 | 25 | 559 | 443 | 419 | 1.00 | 0.54 | 0.67 | 1.00 | 0.79 | 0.75 | 11 | 36 | 13 |
| 81 | 55 | 19 | 30 | 674 | 335 | 407 | 1.00 | 0.35 | 0.55 | 1.00 | 0.50 | 0.60 | 7 | 19 | 16 |
| 82 | 58 | 42 | 42 | 1,001 | 829 | 803 | 1.00 | 0.72 | 0.72 | 1.00 | 0.83 | 0.80 | 7 | 15 | 12 |
| 88 | 44 | 23 | 25 | 788 | 574 | 496 | 1.00 | 0.52 | 0.57 | 1.00 | 0.73 | 0.63 | 12 | 34 | 17 |
| 91 | 40 | 21 | 33 | 826 | 589 | 728 | 1.00 | 0.53 | 0.83 | 1.00 | 0.71 | 0.88 | 11 | 28 | 26 |
| 97 | 50 | 30 | 33 | 873 | 591 | 557 | 1.00 | 0.60 | 0.66 | 1.00 | 0.68 | 0.64 | 13 | 33 | 16 |
| 105 | 60 | 29 | 32 | 859 | 623 | 558 | 1.00 | 0.48 | 0.53 | 1.00 | 0.73 | 0.65 | 10 | 44 | 16 |
| 115 | 72 | 53 | 46 | 1,179 | 1,037 | 876 | 1.00 | 0.74 | 0.64 | 1.00 | 0.88 | 0.74 | 10 | 22 | 15 |
| 121 | 85 | 49 | 54 | 1,335 | 1,029 | 1,031 | 1.00 | 0.58 | 0.63 | 1.00 | 0.77 | 0.77 | 9 | 28 | 18 |
| 157 | 94 | 53 | 54 | 2,118 | 1,425 | 1,332 | 1.00 | 0.56 | 0.57 | 1.00 | 0.67 | 0.63 | 14 | 45 | 25 |
| 161 | 98 | 62 | 66 | 1,816 | 1,433 | 1,342 | 1.00 | 0.63 | 0.67 | 1.00 | 0.79 | 0.74 | 14 | 36 | 15 |
| 183 | 103 | 70 | 61 | 2,060 | 1,647 | 1,351 | 1.00 | 0.68 | 0.59 | 1.00 | 0.80 | 0.66 | 12 | 57 | 20 |
| 189 | 113 | 79 | 77 | 2,515 | 2,150 | 1,973 | 1.00 | 0.70 | 0.68 | 1.00 | 0.85 | 0.78 | 7 | 63 | 21 |
| 293 | 150 | 78 | 77 | 2,929 | 2,043 | 1,906 | 1.00 | 0.52 | 0.51 | 1.00 | 0.70 | 0.65 | 13 | 132 | 25 |
| 369 | 244 | 181 | 158 | 4,941 | 4,030 | 3,604 | 1.00 | 0.74 | 0.65 | 1.00 | 0.81 | 0.73 | 13 | 60 | 21 |
| **Total** | **1,404** | **869** | **887** | **25,922** | **19,875** | **18,557** | | | | | | | | | |
| **Weighted Average** | | | | | | | **1.00** | **0.62** | **0.63** | **1.00** | **0.77** | **0.72** | | | |

Table 5.4 shows the measured execution time for the modified methods. The results were similar to these in Table 5.2. The prefix-graph based solution spent the least time to generate the test paths for 17 subjects. However, the set-covering solution took more time than the breadth-first search approach for eight out of the last nine subjects when the number of prime paths got bigger.

Table 5.4: Execution Time for Modified Methods

| **PPs** | **Time (seconds)** | | | **Time Ratio** | | |
|---|---|---|---|---|---|---|
| | **BF** | **PG** | **SC** | **BF** | **PG** | **SC** |
| 26 | 0.016 | 0.0110 | 0.0060 | 1.00 | 0.69 | 0.38 |
| 35 | 0.033 | 0.0180 | 0.0260 | 1.00 | 0.55 | 0.79 |
| 62 | 0.130 | 0.0300 | 0.0800 | 1.00 | 0.23 | 0.62 |
| 63 | 0.004 | 0.0007 | 0.0004 | 1.00 | 0.18 | 0.10 |
| 68 | 0.150 | 0.0400 | 0.1700 | 1.00 | 0.27 | 1.13 |
| 81 | 0.170 | 0.0500 | 0.2300 | 1.00 | 0.29 | 1.35 |
| 82 | 0.300 | 0.0600 | 0.2000 | 1.00 | 0.20 | 0.67 |
| 88 | 0.220 | 0.0560 | 0.3900 | 1.00 | 0.25 | 1.77 |
| 91 | 4.480 | 0.4800 | 3.3100 | 1.00 | 0.11 | 0.74 |
| 97 | 0.710 | 0.0800 | 0.5600 | 1.00 | 0.11 | 0.79 |
| 105 | 0.490 | 0.1100 | 0.7600 | 1.00 | 0.22 | 1.55 |
| 115 | 0.620 | 0.1300 | 0.9100 | 1.00 | 0.21 | 1.47 |
| 121 | 0.700 | 0.1100 | 1.1300 | 1.00 | 0.16 | 1.61 |
| 157 | 2.520 | 0.3200 | 4.6700 | 1.00 | 0.13 | 1.85 |
| 161 | 2.470 | 0.3200 | 4.1300 | 1.00 | 0.13 | 1.67 |
| 183 | 1.790 | 0.3400 | 6.0900 | 1.00 | 0.19 | 3.40 |
| 189 | 3.610 | 0.3500 | 2.4400 | 1.00 | 0.10 | 0.68 |
| 293 | 10.400 | 1.1800 | 33.9000 | 1.00 | 0.11 | 3.26 |
| 369 | 26.200 | 2.7900 | 63.5000 | 1.00 | 0.11 | 2.42 |
| **Total** | **55.013** | **6.4757** | **122.5024** | | | |
| **Unweighted Averages** | | | | **1.00** | **0.22** | **1.06** |
| **Weighted Averages** | | | | **1.00** | **0.12** | **2.23** |

### 5.1.3 Experimental Analysis

The data in Tables 5.1 and 5.3 show that the set-covering based and prefix-graph based solutions generated fewer test paths and nodes than the previous breadth-first search solution. On average, the two new solutions saved 30%-40% of the test paths and 20%-30% of the nodes for the open source methods and the modified methods. The methods with more prime paths had higher savings. For example, the last method in Table 5.1 had a saving of 70% in terms of the number of test paths.

From the experimental results, it seems that the prefix-graph and set-covering based solutions yielded the greatest savings for methods that have "complex" nested loops. Nested loops are likely to lead to more overlaps among the prime paths, resulting fewer number of test paths and nodes. However, it is hard to quantify when to use the new solutions. It is difficult to measure the overlaps among test requirements precisely and determine how graph structures could affect the test requirements and overlaps.

The data in Tables 5.1 and 5.3 also do not definitely say which solution is better. Theoretically, the approximation ratio of the greedy set cover algorithm is $2 \ln n$ while the prefix-graph based algorithm is only 3 [17]. Thus, the prefix-graph based algorithm is expected to yield fewer test paths. However, the experimental results do not always comply with the theoretical worst-case bounds. Besides, each of the new solutions used a test suite minimization algorithm (Algorithm 4) and a super-test requirement splitting algorithm (Algorithm 7). Therefore, I could not estimate the experimental results from the theoretical approximation ratios. The prefix-graph based solution gave better results for 23 subjects, the set-covering based solution for 11 subjects, and they were the same for 3 subjects.

Tables 5.2 and 5.4 show the prefix-graph based solution was faster than the breadth-first search solution for all 37 subjects, and faster than the set-covering based solution for 31 of 37 subjects. The set-covering based solution was faster than the breadth-first search solution for 24 subjects; however, for 14 subjects, the set-covering based solution was slower than the breadth-first search solution.

My observation was that when graphs had few prime paths, the set-covering based solution was faster than the other solutions. When graphs had more prime paths, the set-covering based solution became slower. For example, consider the last subject in Table 5.2. The prefix-graph based solution finished in less than 4 minutes; while the breadth-first search solution took about 25 minutes and the set-covering based solution took almost 10 hours! I concluded that the set-covering solution may be better for small graphs, but may simply be too slow with large graphs.

For some subjects, the maximum ratio of TR to TP was big and the ratio of test requirements that were covered by one test path over all the test requirements was high as well. Other algorithms should be applied to control the maximum ratio of TR to TP. Dynamic programming techniques may solve this problem, which will be discussed in future work, section 6.3.

### 5.1.4 Threats to Validity

As usual with most software engineering studies, there is no way to show that the subjects selected are representative. Thus an external threat to validity is that these results may not hold on other programs. It is important to note that the results do not depend on the behavior of the software, only on the structure of the graph. The key elements that affect the results are the number of prime paths and the amount of overlap among them. An obvious internal threat is that the implementations of these algorithms might be imperfect. The results could also be affected by the use of a different splitting algorithm or a different test suite minimization algorithm.

### 5.1.5 Recommendations

These data strongly indicate that both the prefix-graph and set-covering based solutions generate fewer test paths and nodes than the breadth-first search solution, particularly when methods have nested loops and otherwise complex structure. It seems harder to quantify exactly which and when the solutions should be used. There is probably no controversy

that getting fewer test paths or total nodes is always better, however, the long execution times for the subject with 1844 prime paths is problematic. No tester wants to wait 10 hours for the set-covering based solution to complete, particularly when it only saves 19 test paths over the prefix-graph based solution!

I might wish for a preliminary analysis of the graphs to decide which solution would be the best. In the absence of such an ability, however, I incorporated the prefix-graph based solution in the graph coverage web application. It does not always generate as good solutions as the set-covering solution, but also does not appear to ever take unacceptably long. In the future work, I will adapt the solutions for each variant of the MCTP problem (see Table 1.1).

## 5.2 The Experiments for the Mapping Problem

In this research, the solution to the mapping problem is a structured test automation language (STAL), which is used to automate the transformation from abstract tests to concrete tests. Using STAL could decrease cost and errors made by reducing the amount of repetitive, mechanical work required for manual transformation. Because there are no techniques available to automate the mapping problem from only behavioral models, testers typically do this by hand. Therefore, the experiments compared STAL with the manual approach. Three research questions were proposed:

1. RQ1: Can the structured test automation language (STAL) be used to create automated tests in a practical setting?

2. RQ2: Can using STAL help testers reuse redundant test code and reduce errors when converting abstract tests to concrete tests as compared with doing the same procedure by hand?

3. RQ3: Can STAL be used for different kinds of programs such as web applications and GUIs?

This section presents the experimental design, subjects, procedure, results, threats to validity, and then an analysis of the results.

### 5.2.1 Experimental Design

Test engineers automate tests to reduce the cost of running the same test many times, to reduce the errors inherent in running tests by hand, and to make it easier to modify the test suite when the model, software, or test criterion changes. Evaluating STAL for all of these scenarios would require extensive human resources, so I evaluated the initial development of tests. The scenario is, given a program and its model, testers generate automated tests to satisfy a coverage criterion. The testers did this by hand and with STAL. Any benefits from using STAL during the initial development would also be present when modifying the tests.

Nine testers designed tests for 17 programs. It can take many hours (up to approximate 10 hours without additional help) to design and develop model-based tests by hand, so each program was assigned to one tester at one time. The tests were designed to satisfy edge coverage [7], a widely used and relatively simple test criterion. The testers developed two sets of tests for each program, one by hand and the other using STAL. There were two levels of automation in this study. STALE helps testers **automate** the creation of tests, which were encoded in **automated** JUnit scripts. That is, testers were using STALE to create automated tests automatically. I try to clarify which one I refer to in the following text.

An important decision was which process to use first, manual or automated. Table 5.5 shows the steps for each.

Step 1 is the same both by hand and with STAL. The testers need to understand the software, analyze its controllability and observability, and decide how to implement events from the model in a test. Step 2 is quite different for each process.

For A2, the testers identify the declarations and initializations of objects used in the test code for the elements of the model. Because an element may appear more than once in a

Table 5.5: Steps in Automated and Manual Test Generation Processes

| | **Automated (A)** | **Manual (M)** |
|---|---|---|
| 1 | Understand the model and system under test. Find the test code for each element from a model. (A1) | The same as A1. (M1) |
| 2 | Extract object declarations and initializations from the element mappings. Enter mappings into the structured test automation language framework (STALE) and provide enough mappings to satisfy constraints. (A2) | Write executable tests to map test paths. (M2) |
| 3 | Generate concrete tests and correct errors. (A3) | Correct errors. (M3) |

test path, the corresponding test code will appear multiple times. Object declarations in the test code can result in duplicated object declarations. To avoid errors from this duplication, testers can either put all object declarations and necessary initializations in the mapping for the first transition if all test paths share the same transition, or create object mappings to be required mappings for elements. Testers then enter test code in STALE, satisfying the model constraints. This is the most time consuming part of the automated process.

In M2, the testers first look at the test paths, find matched elements from the model, and write the corresponding test code for the elements. This is the most time consuming part of the manual test generation. Switching among the test paths, the model, and the test code is difficult, takes time, and can result in errors where the test code does not match the test paths.

When testers generate tests manually, they learn how to separate object declarations and how to create enough mappings to satisfy all constraints while writing the concrete tests. If done first, A2 will become much easier and shorter because separating object declarations and creating mappings take most of the time during A2, thus introducing a bias in favor of the automated process. If the automatic process is used first, testers would learn how to write code for each transition and state since they must analyze them thoroughly to create mappings. However, this does not reduce much time for M2 because testers spend most of their time on checking if test code matches the model and if elements in the model are matched to test paths back and forth, writing code for redundant elements, and correcting

errors when mismatched code is found. Besides, the testers do not see the complete tests in A2, therefore, the knowledge gained during step A2 does not make M2 much easier.

Testers may get compilation errors in the automated process if they did not include all the classes or JAR files needed or if the test code in the mappings contain syntax errors. Also, if some constraints are not satisfied, the testers may need to add additional mappings or values. The testers correct these errors in step A3. If M3 is done before A2, the testers will be less likely to make mistakes, so A3 will become easier, again introducing a bias in favor of the automated process. However, doing A3 before M2 does not affect errors in M3 because they are arbitrary syntax errors.

Given these considerations, I concluded that the testers needed to first generate tests using the automated method, then manual. The guide that was given to the participants is online at *http://cs.gmu.edu/∼nli1/experiment/*. A complete experimental guide is also included in Appendix C.

### 5.2.2 Experimental Subjects

Seven of the 17 programs used are open source projects: Calculator[6], Snake[7], TicTacToe[8], CrossLexic[9], Jmines[10], Chess[11], and DynamicParser[12]. Six were from textbooks: VendingMachine [7], ATM [82], Tree [83], BlackJack [84], Triangle [85], and Poly [86]. The other four were taken from the coverage web application for Ammann and Offutt's book [12]. All programs are in Java. To examine RQ3, four types of subjects were selected. Calculator, Snake, CrossLexic, Jmines, Chess, BlackJack, and DynamicParse were GUIs. GraphCoverage, DFCoverage, LogicCoverage, and MinMCCoverage were web applications. TicTacToe was a command-line program. The other five programs are example programs that do not have user interfaces.

[6] http://jcalcadvance.sourceforge.net/
[7] http://sourceforge.net/projects/javasnakebattle/
[8] http://sourceforge.net/projects/tttnsd/
[9] http://crosslexic.sourceforge.net/
[10] http://jmines.sourceforge.net/
[11] http://twoplayerchess.sourceforge.net/
[12] http://dynamic-parser.sourceforge.net/

The UML state machine diagrams of the experimental subjects were drawn using the Eclipse modeling framework (EMF) tool Papyrus [78] by hand, then the diagrams were transformed into generic graphs by STALE. For a few of the more complicated programs, parts of the programs' functionalities were omitted from the diagrams to ensure the testers could complete the program in the allotted two hour time frame. Nine testers (not including me) participated in the experiment. They were part-time and full-time graduate students at George Mason University, all of whom have taken Mason's graduate testing class.

### 5.2.3 Experimental Procedure

The testers were given the experimental guide and asked to understand the process and gain a preliminary familiarity with STALE. This took about two hours apiece. Next the testers entered the software engineering lab at George Mason University at different times and generated tests automatically with STALE, then by hand, in a controlled environment. Thus, subjects did not communicate with each other. All subjects used the same computer and their times were measured. The automated steps were:

1. Create a new project and add the model and program under test to STALE.
2. Create the abstract to concrete mappings. Wall-clock time was measured.
3. Create concrete tests using the tool to satisfy edge coverage. The tool measured the time for this step.

The manual steps were:

1. Write concrete tests by hand. The concrete tests have to be written in the same order as the test paths (abstract tests) to make test comparison easy. The constraints in the states had to be satisfied.
2. Compile the tests and make sure that all tests pass. Wall-clock time was measured.

All subjects finished the experiments in two hours. After completing the experiment, each participant was given an anonymous questionnaire, shown in Table 5.6. Most of the

Table 5.6: Questionnaire

| | **Questions** |
|---|---|
| 1 | Are you working (enter "programmer," "manager," "tester," etc.)? If not, enter "student." |
| 2 | If you have to generate tests from models, would you consider using this automatic test generation / tool? |
| 3 | Please rate the ease of use of the test automation tool on a scale of 1 to 5 (1 being impossible and 5 being trivially easy). |
| 4 | Do you have any suggestions for improving this automatic test generation / tool and other comments? |

participants had taken a graduate course in user interface design and development at George Mason University, so could be expected to be fairly knowledgeable and critical with question 3.

### 5.2.4 Experimental Results

Table 5.7 gives the experimental data. The first four columns give the subject names and statistics about the sizes of the graphs and programs. The nodes and edges are from the generic graphs, not the original models (UML state machine diagrams), and the lines of code were calculated by CLOC [87].

The next three columns show the number of distinct mappings created from the models, the number of times the distinct mappings appear in all the tests, and the ratio of the Mappings column over the All Mappings column. For example, the *VendingMachine* model needed 13 mappings and the seven tests used a total of 132 mappings, 9.8% of which are distinct. The *Test* column shows the number of tests generated for each subject.

The next two columns present the number of seconds used to create mappings and generate tests in the automated process, followed by the time used by the manual process. Last is the ratio of time for the automated process over the time for the manual process. Thus, the automated process for class *VendingMachine* took 34.5% of the time of the manual process.

The first research question asked if STAL could be used in a practical situation. All

nine subjects were able to use STAL to create automatic tests for 16 programs with only a short tutorial, so the answer to RQ1 is yes. The second research question asked if testers could use STAL to reuse redundant test code and reduce errors. Table 5.7 shows that the automated process ranged from 11.7% to 60.8% of the time the manual process took, with an unweighted average of 29.6%. The manually created tests have 48 errors compared with zero for the automatically created tests. The tests were examined for errors by hand. The errors occurred when the participants wrote unmatched test code for a transition in the model or wrote redundant test code for same transitions. The participants made more errors with large programs. Thus, the answer to RQ2 is also yes. Nine subjects were able to generate tests for the programs including GUI applications, web applications, command-line programs, and normal programs. Therefore, the answer to RQ3 is yes as well.

On the questionnaires, all subjects answered "Yes" to the second question, and the average usability rating (third question) was 4.4. The tests created by STALE and by hand for the vending machine example are listed in Appendix B.

Table 5.7: Time for Automatic and Manual Test Generation

| **Programs** | **LOC** | **Nodes** | **Edges** | **Mappings** | **All Mappings** | **% Mapping** | **Tests** | **Automatic (Seconds)** | | **Manual (Seconds)** | **% Time** |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | **Mapping Creation** | **Test Generation** | **Test Generation** | |
| VendingMachine | 52 | 11 | 26 | 13 | 132 | 9.8 | 7 | 630 | 4 | 1,836 | 34.5 |
| ATM | 463 | 8 | 12 | 8 | 27 | 29.6 | 5 | 449 | 1 | 1,267 | 35.5 |
| Calculator | 2,919 | 17 | 76 | 12 | 182 | 6.6 | 14 | 477 | 15 | 3,794 | 13.0 |
| Triangle | 124 | 7 | 31 | 12 | 72 | 16.7 | 6 | 440 | 2 | 1,371 | 32.2 |
| Snake | 1,382 | 18 | 116 | 13 | 132 | 9.8 | 7 | 503 | 46 | 1053 | 52.1 |
| TicTacToe | 665 | 7 | 12 | 9 | 31 | 29.0 | 5 | 640 | 2 | 1,494 | 43.0 |
| CrossLexic | 654 | 13 | 51 | 17 | 305 | 5.6 | 26 | 609 | 123 | 2,539 | 28.8 |
| JMines | 9,486 | 18 | 75 | 10 | 202 | 5.0 | 26 | 445 | 62 | 2,625 | 19.3 |
| Chess | 2,048 | 9 | 17 | 7 | 36 | 19.4 | 6 | 510 | 6 | 904 | 57.1 |
| BlackJack | 403 | 13 | 20 | 12 | 36 | 33.3 | 8 | 300 | 4 | 500 | 60.8 |
| Tree | 234 | 14 | 24 | 11 | 83 | 13.3 | 6 | 685 | 2 | 1671 | 41.1 |
| Poly | 129 | 8 | 21 | 11 | 64 | 17.2 | 5 | 330 | 3 | 1537 | 21.7 |
| DynamicParser | 1,269 | 22 | 73 | 12 | 269 | 4.5 | 21 | 468 | 45 | 4010 | 12.8 |
| GraphCoverage | 4,480 | 20 | 67 | 17 | 253 | 6.7 | 19 | 521 | 14 | 4091 | 13.1 |
| DFCoverage | 4,512 | 15 | 56 | 16 | 147 | 10.9 | 19 | 401 | 7 | 2824 | 14.4 |
| LogicCoverage | 1,808 | 15 | 83 | 15 | 196 | 7.7 | 38 | 522 | 7 | 4512 | 11.7 |
| MinMCCoverage | 3,252 | 14 | 53 | 14 | 150 | 9.3 | 22 | 434 | 1 | 3642 | 11.9 |
| **Total** | 31,832 | 232 | 742 | 196 | 2,325 | | 240 | 8,364 | 344 | 39,670 | |
| **Average** | | | | | | 13.8 | | | | | 29.6 |

### 5.2.5 Experimental Analysis

Figure 5.1 compares the *% Mapping* (on the horizontal axis) and *% Time* (on the vertical axis) for the 17 subject programs. *% Mapping* and *% Time* have the same meaning as in Table 5.7. If *% Mapping* is small, it means that testers do not need to create many distinct mappings, by comparison with all mappings that appear in all tests. Therefore, it is likely that the automatic process takes far less time than the manual process, since the manual test generation for mappings of the repeated transitions can be automated. Thus, I expected a linear correlation relationship between *% Mapping* and *% Time*.

Boddy and Smith [88] suggest using Pearson's correlation coefficient if the data have a normal distribution; otherwise, a non-parametric correlation test [89] such as Spearman's rank correlation coefficient should be used. Qqplots [90] (Figures 5.2 and 5.3) show that the *% Mapping* and *% Time* data deviate from the straight line. Thus, Spearman's correlation coefficient was used.

The *correlation coefficient ($\rho$)* of Spearman's correlation test was 0.72. Cohen [91] suggests that a value of .5 or greater can be considered to be a large correlation. The statistical significance *p-value* was 0.0017, which is normally considered to be highly significant. Therefore, it was concluded that the savings from using the automated process increases as the percentage of distinct mappings in all mappings decreases. This is an encouraging finding for scalability.

### 5.2.6 Threats to Validity

As usual with most software engineering studies, there is no way to show that the selected subjects were representative. This was true both for the programs and the human testers. Another threat to external validity is that I created the UML models from the source code. An internal threat is that STALE's implementation might be imperfect. Another internal threat is that the answers of the participants to the questionnaires might be subjective. Three of the participants are my friends and knew I was going to review the anonymous questionnaires. This might affect the results.

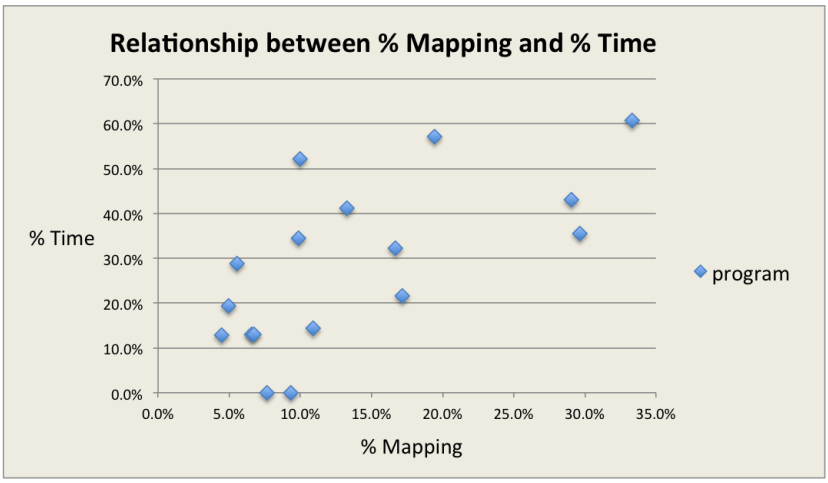


Figure 5.1: Relationship between *% Mapping* and *% Time*

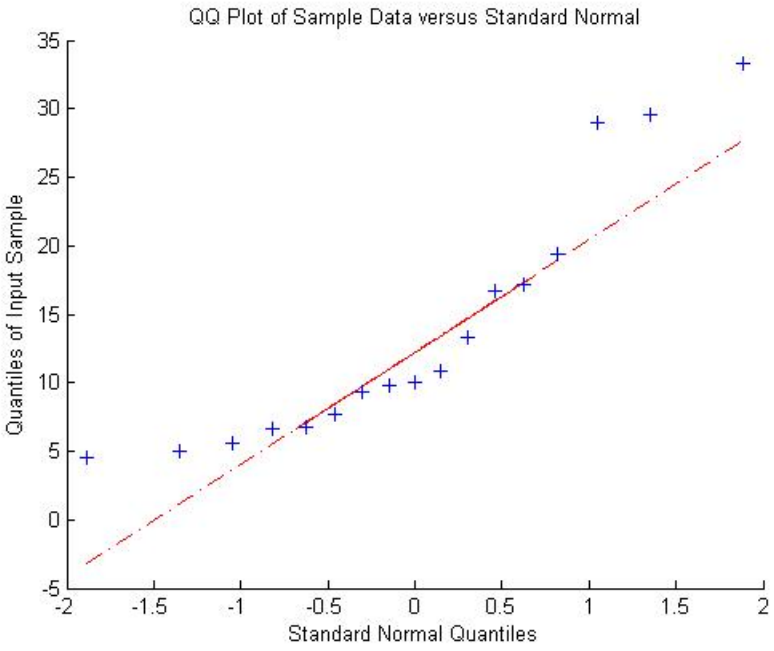


Figure 5.2: Qqplot for *% Mapping*

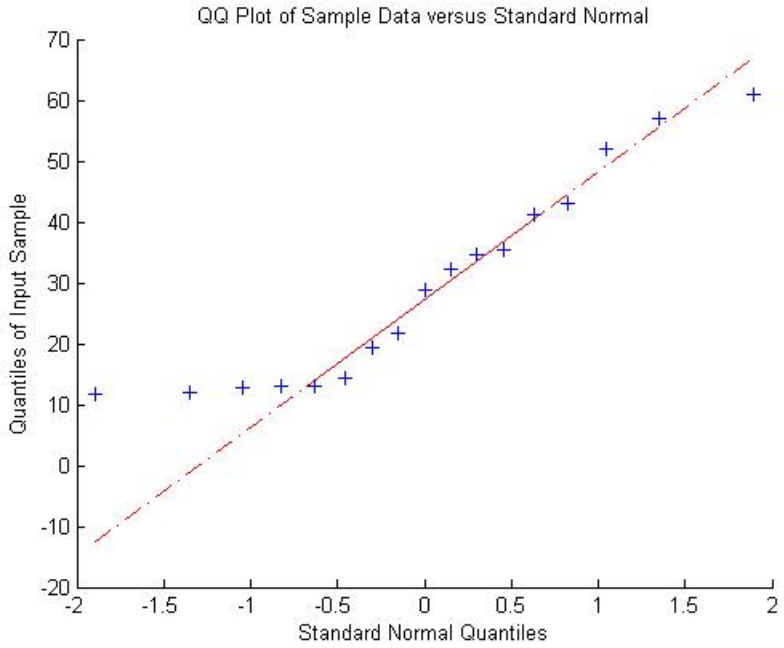


Figure 5.3: Qqplot for *% Time*

### 5.2.7 Discussion

The subjects included four web applications, seven GUIs, one command-line application, and five other programs. STAL worked for all of them with no discernible difference. It was not surprising, since STAL depends on a model and the number of tests, not the kind of program.

## 5.3 The Experiments for the Test Oracle Problem

The test oracle problem refers to how much of program states to check and how frequently to check the program states. The solution to the test oracle problem in this research was to develop new test oracle strategies (OSes) to check various outputs and internal state variables with different frequencies of checking program states. The experiments for the test oracle problem addressed four questions:

**RQ1:** With the same test inputs, does a more precise OS reveal more faults than a less

precise OS?

**RQ2:** With the same test inputs, does checking outputs and internal state variables multiple times reveal more faults than checking the same outputs and internal state variables once?

**RQ3:** With the same OS, do tests that satisfy a stronger coverage criterion reveal more faults than tests that satisfy a weaker coverage criterion?

**RQ4:** Which OS is recommended when considering both effectiveness and cost?

Other researchers [8–11] have studied RQ1, finding that more precise OSes are more effective than less precise OSes at revealing faults. However, they used different test coverage criteria and OSes on different types of programs, as discussed in section 2.6.

RQ2 was evaluated by Xie and Memon [11], who found that checking variables after each event (events used in GUI testing that represent user actions such as button clicks) can detect more faults than checking the same variables once after the last event of the test. However, their study only monitored states of GUIs such as windows. This research checked more outputs and internal state variables of different kinds of programs.

Briand et al. [8] found that tests that satisfy a stronger coverage criterion can find more faults than a weaker criterion (RQ3), with the same test oracle strategy, for one out of three experimental programs. In contrast, my experiments used two different coverage criteria and 16 programs.

A very effective OS may be too costly for practical use. Thus, RQ4 evaluated the cost-effectiveness of the OSes. The rest of this section presents the experimental design, subjects, procedure, results, and threats to validity.

### 5.3.1 Experimental Design

The experiments compared the ten new OSes (OS1, OS2, OS3, OS4, OS5, OT1, OT2, OT3, OT4, and OT5) with the two baseline OSes, NOS and SIOS. The twelve OSes are defined

Table 5.8: A Comparison of Test Oracle Strategies

| **Program State** | **NOS** | **SIOS** | **OS1** | **OS2** | **OS3** | **OS4** | **OS5** |
|---|---|---|---|---|---|---|---|
| Runtime exceptions | X | X | X | X | X | X | X |
| State invariants | | X | X | X | X | X | X |
| Object members | | | X | | X | | X |
| Return values | | | | X | X | X | X |
| Parameter members | | | | | | X | X |

in section 4.3. All OSes were applied to edge-adequate and edge pair (EP)-adequate tests. Then the tests were run against faulty versions of the programs. The faults revealed and the cost of using the OSes were recorded. To remind readers what outputs and internal state variables each OS checks, Table 5.8 shows NOS, SIOS, OS1, OS2, OS3, OS4, and OS5. $OTi$ checks the same outputs and internal state variables as $OSi$ ($1 \leq i \leq 5$), with one check per test (after the last transition). Thus, Table 5.8 does not show OT1, OT2, OT3, OT4, and OT5.

Andrews et al. [44] found that synthetic faults generated using mutation analysis can be used as faults in experiments to predict the real fault detection ability of tests. This research used synthetic faults, which were treated as real faults. The background of mutation testing has been introduced in section 2.3, to better explain how mutation analysis was used in this research, some terms of mutation testing are summarized here. In mutation testing, a *mutant* is a slight syntax change to the original program. A *mutation operator* is a rule or a set of rules that specifies how to generate mutants. If a test causes a mutated program (mutant) to produce different results from the original program, this mutant is said to be "killed." If a mutant cannot be killed by any tests, it is called *equivalent*.

The *mutation score* is the ratio of mutants that are killed over the killable mutants (non-equivalent mutants), which serves as a measure of the effectiveness of a test set. Because programs have different numbers of faults, to compare the effectiveness of the tests in the same scale, percentages of faults (mutation scores) detected by the tests were used to measure the effectiveness of the tests for each program. Furthermore, if tests have the same

test inputs but different OSes, the mutation score of each set of tests can reflect the relative effectiveness of each OS. A higher mutation score indicates the OS is more effective. With the same inputs, a more precise OS can be expected to be at least as effective at revealing faults as a less precise OS. Thus, OS1, OS2, OS3, OS4, OS5, OT1, OT2, OT3, OT4, and OT5 were expected to reveal more faults than NOS and SIOS.

The experiment used muJava [43, 46], a mutation analysis tool for Java, to generate synthetic faults. Each mutated program has only one mutant. Users can use muJava to generate mutants, run tests against mutants, and view mutants, which helps testers recognize equivalent mutants. The latest version of muJava supports JUnit tests and all features of Java 1.6. So the JUnit tests that STALE generated were used in muJava directly. Mutants were generated by using the 15 selective method-level mutation operators of muJava [92].

All OSes were applied with the same sets of edge-adequate and edge-pairs (EP)-adequate tests. In the experimental process, the test oracle generation and execution had three kinds of cost. First, testers entered test oracle data (assertions) by hand. Second, STALE generated tests based on the provided test oracle data. Thus, the concrete tests include the test oracles. Note that the assertions provided by testers in the first step might be used many times in the concrete tests because the program states were checked after each transition in OS1, OS2, OS3, OS4, and OS5. Third, as part of tests, assertions must be executed. The second and third steps were automated, so the humans cost in the first step dominated. Thus, the cost of an OS mainly depends on the cost of creating test oracle data (assertions) by hand.

Each assertion was given equal weight, and the cost of creating test oracles (assertions) was the sum of costs of creating each assertion. The assertions include state invariants and normal assertions such as checking a member variable of a class. There were four reasons that I gave both state invariants and normal assertions the same weight. First, writing an assertion needs testers to understand the program. Thus, writing any assertion takes the same amount of time for understanding the program. Second, this research required testers

to write test oracles for each transition. This step required test oracles can be generated automatically no matter where transitions appear in tests. Thus, testers have to change the test code of the transitions, as shown in section 4.4. My experience told me that the first two steps took the most time for creating an assertion. Third, designing state invariants is only part of designing a state machine diagram and the diagram was mainly used for generating tests. Moreover, a group of assertions (state invariants in different states) was created within the diagram. Thus, the time for each state invariant only takes a fraction of the total time. Fourth, when designing normal assertions, testers may have to spend extra time to look for assertions. For instance, when checking member variables of a class, if a member variable is also an object, testers have to check its member variables of this object until all member variables are primitive. Therefore, the cost of each assertion was treated equally and I used the number of distinct assertions as an approximation for the cost.

The cost-effectiveness of an OS is the ratio of the number of assertions over the percentage of faults detected by the OS. The assertions were provided by hand to check the internal state variables and outputs. A smaller cost-effectiveness is better. The cost-effectiveness ratio can be interpreted as: To sacrifice one percent of mutation score (percentage of faults detected), how many fewer assertions should testers write? This cost-effectiveness ratio was applied to SIOS, OS1, OS2, OS3, OS4, OS5, OT1, OT2, OT3, OT4, and OT5 but not to NOS. The cost of NOS is zero, so is incompatible with the cost-effectiveness ratio.

$$\textit{Cost-effectiveness} = \frac{\#assertionsCreatedByHand}{\%FaultsDetected} \tag{5.1}$$

To better specify how to measure the goals of the experiments, three groups of hypotheses were extracted from the first three research questions. The first group of hypotheses ($Hypotheses_A$) compared all pairs of OSes, $OS_A$ and $OS_B$, where $OS_B$ is more precise than $OS_A$. The null and alternative hypotheses are listed below.

**Null hypothesis ($H_0$):**

There is no difference between the percentage of failures revealed by $OS_A$ and $OS_B$

with the same test inputs.

**Alternative hypothesis ($H_1$):**

The percentage of failures revealed by $OS_B$ is greater than $OS_A$ with the same test inputs.

The test oracles strategy pairs that were applied to $Hypotheses_A$ are: {NOS, SIOS}, {SIOS, OS1}, {SIOS, OS3}, {SIOS, OS5}, {OS1, OS3}, {OS3, OS5}, {OS1, OS5}, {OS2, OS5}, {OS4, OS5}, {SIOS, OS2}, {SIOS, OS4}, {OS2, OS3}, {OS2, OS4}, {SIOS, OT1}, {SIOS, OT3}, {SIOS, OT5}, {OT1, OT3}, {OT3, OT5}, {OT1, OT5}, {OT2, OT5}, {OT4, OT5}, {SIOS, OT2}, {SIOS, OT4}, {OT2, OT3}, and {OT2, OT4} for both edge coverage (EC) and edge-pair coverage (EPC). This research did not compare NOS with other OSes (OS1, OS2, OS3, OS4, OS5, OT1, OT2, OT3, OT4, and OT5) because NOS was expected to be much less effective than other OSes. The comparison between the effectiveness of NOS and other OSes is shown in section 5.3.4.

RQ2 asks if checking outputs and internal state variables multiple times is more effective at finding faults than checking the same outputs and internal state variables once. $OTi$ checks the same object members, return values, and parameter members that $OSi$ checks, but $OSi$ checks after each transition and $OTi$ only checks once after the last transition, where $1 \leq i \leq 5$. Thus, the second group of hypotheses ($Hypotheses_B$) for RQ2 were:

**Null hypothesis ($H_0$):**

There is no difference between the percentage of failures revealed by $OTi$ and $OSi$ with the same test inputs, where $1 \leq i \leq 5$.

**Alternative hypothesis ($H_1$):**

The percentage of failures revealed by $OSi$ is greater than $OTi$ with the same test inputs, where $1 \leq i \leq 5$.

The test oracles strategy pairs that were applied to $Hypotheses_B$ are: {OT1, OS1}, {OT2, OS2}, {OT3, OS3}, {OT4, OS4}, and {OT5, OS5}.

The third group of hypotheses ($Hypotheses_C$) for RQ3 took two test coverage criteria $CC_A$ and $CC_B$ into consideration ($CC_B$ subsumes $CC_A$).

**Null hypothesis ($H_0$):**

There is no difference between the percentage of failures revealed by criterion $CC_A$ and $CC_B$ if both use the same test oracle strategy.

**Alternative hypothesis ($H_1$):**

The percentage of failures revealed by criterion $CC_B$ is greater than $CC_A$ if both use the same test oracle strategy.

$Hypotheses_C$ were applied to edge-adequate and EP-adequate tests for any of the twelve OSes used in this research.

### 5.3.2 Experimental Subjects

I evaluated 16 Java programs and used STALE to generate tests from their UML state machine diagrams. First, I generated test inputs to satisfy both EC and EPC. Then I entered test oracle data for ten OSes (NOS did not need test oracle data, and the state invariant test oracle data were provided by STALE while generating test inputs). Finally, the twelve OSes were applied to the two sets of tests that satisfy EC and EPC, resulting in 24 sets of tests for each program.

The programs and UML state machine diagrams that were used to derive tests in this experiment are almost the same as were used in section 5.2. The only difference is that this experiment did not use the program Chess because its tests could not be run under muJava.

Table 5.9 shows some properties of the programs and tests. The column *LOC* shows the lines of code for each program. The columns *Tests* show the number of tests for edge-adequate and EP-adequate tests. The columns *Trans* represent the number of transitions that appeared in the tests and the columns *SI* provide the number of appearances of state invariants that were satisfied and also used as test oracles. The columns *D-Trans* and *D-SI* represent the number of distinct mappings of transitions and state invariants provided by

hand.

Section 4.4 (the structured test automation framework) talks about how users provide mappings for transitions and state invariants so that abstract tests can be transformed to concrete tests. Since transitions and state invariants appear many times, the numbers of the columns *Trans* and *SI* are far more than those of the columns *D-Trans* and *D-SI*. By comparing the columns *D-Trans* for EC and EPC, I only needed to provide more mappings for EPC than EC for three programs. That is, the mappings required to satisfy state invariants for EC can also satisfy most of the state invariants for EPC.

### 5.3.3 Experimental Procedure

The experiment was carried out in the following steps:

1. For each program, I used STALE to create test inputs by hand to satisfy EC, and then generated additional test inputs to satisfy EPC.

2. STALE was used to enter expected results for the OSes. Then 24 test sets were generated for each pair of combination for the two coverage criteria and twelve OSes.

3. MuJava was used to generate faults for each program. Equivalent mutants were identified by hand and then removed.

4. Each set of tests was run against the faults for each program. The number of faults revealed as failures detected and the number of times the internal state variables and outputs are checked for each set of tests were recorded.

5. The cost-effectiveness of each OS was calculated and analyzed.

For efficiency, tests were only run against faults that appeared in methods called when the tests were run.

Table 5.9: Experimental Subjects

| **Programs** | **LOC** | **Properties of the Tests** | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | **Edge** | | | | | **Edge Pair** | | | | |
| | | **Tests** | **Trans** | **SI** | **D – Trans** | **D – SI** | **Tests** | **Trans** | **SI** | **D – Trans** | **D – SI** |
| ATM | 463 | 5 | 18 | 22 | 6 | 6 | 6 | 30 | 39 | 6 | 6 |
| BlackJack | 403 | 8 | 27 | 27 | 11 | 3 | 9 | 51 | 51 | 11 | 3 |
| Calculator | 2,919 | 14 | 167 | 16 | 11 | 9 | 39 | 893 | 893 | 11 | 9 |
| CorssLexic | 654 | 26 | 113 | 209 | 11 | 7 | 63 | 404 | 756 | 11 | 7 |
| DFGraphCoverage | 4,512 | 8 | 49 | 78 | 10 | 7 | 45 | 390 | 643 | 10 | 7 |
| DynamicParser | 1,269 | 20 | 116 | 408 | 13 | 15 | 26 | 385 | 1,233 | 14 | 15 |
| GraphCoverage | 4,480 | 16 | 122 | 207 | 14 | 11 | 23 | 359 | 605 | 14 | 11 |
| JMines | 9,486 | 9 | 60 | 6 | 7 | 1 | 22 | 201 | 21 | 7 | 1 |
| LogicCoverage | 1,808 | 30 | 115 | 94 | 12 | 8 | 94 | 561 | 483 | 12 | 8 |
| MMCoverage | 3,252 | 78 | 273 | 228 | 20 | 16 | 142 | 699 | 570 | 20 | 16 |
| Poly | 129 | 5 | 32 | 57 | 11 | 6 | 12 | 129 | 237 | 18 | 6 |
| Snake | 1,382 | 7 | 70 | 120 | 10 | 8 | 8 | 194 | 341 | 10 | 8 |
| TicTacToe | 665 | 5 | 24 | 7 | 6 | 3 | 7 | 46 | 16 | 6 | 3 |
| Tree | 234 | 6 | 35 | 48 | 6 | 3 | 8 | 99 | 146 | 6 | 3 |
| Triangle | 124 | 6 | 36 | 36 | 7 | 5 | 27 | 271 | 271 | 7 | 5 |
| VendingMachine | 52 | 7 | 44 | 88 | 6 | 6 | 9 | 105 | 210 | 7 | 6 |
| **Total** | **33,983** | **250** | **1,301** | **1,802** | **161** | **114** | **540** | **4,817** | **6,515** | **170** | **114** |

### 5.3.4 Experimental Results

Tables 5.10 and 5.11 show the failures revealed by each OS for each program with both sets of tests. Both Tables 5.10 and 5.11 show the total number of faults of each program and the faults that are revealed as failures by OSes for EC and EPC. Table 5.10 shows the results for $OSi$ while table 5.11 focuses on the results for $OTi$, where $1 \leq i \leq 5$. Columns *NOS* and *SIOS* in both Tables 5.10 and 5.11 are the same. Tables 5.12 and 5.13 have similar columns. They show the number of faults that are revealed as failures by each OS divided by the total number of faults for each program, producing percentages of failures revealed by each OS for each program. The total number of the faults ("# Faults", non-equivalent mutants) is 9,627. So a total of 91,263,960 tests were executed ((12 OSes * 250 edge-adequate tests) + (12 OSes * 540 EP-adequate tests)) * 9,627). The JUnit tests for EC and EPC with twelve OSes for the vending machine example are listed in Appendix B.

Note that OT1 found more faults than OS1 for the subjects "TicTacToe" and "Triangle," and OT3 found more faults than OS3 for the subject "TicTacToe." As discussed in section 4.3, $OTi$ may check different program states from $OSi$, where $i \leq i \leq 5$. Thus, $OTi$ may reveal more failures than $OSi$ for some subjects. Tables 5.12 and 5.13 show that NOS revealed far fewer failures than the other OSes on average, indicating NOS is much less effective at finding faults. Since a more precise OS checks more program states than a less precise OS, I expected the more precise OS to find more faults, and checking outputs and internal state variables more frequently can find more faults than checking the same program states less frequently. However, Tables 5.12 and 5.13 show that the percentages of the faults detected by OS1, OS2, OS3, OS4, OS5, OT1, OT2, OT3, OT4 and OT5 were very close, and 4 to 7% higher than that of SIOS.

Table 5.10: Numbers of Faults Found by Test Oracle Strategies - Part1

| **Programs** | **#Faults** | **# Faults Found by Test Oracle Strategies** | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | **Edge** | | | | | | | **EdgePair** | | | | | | |
| | | **NOS** | **SIOS** | **OS1** | **OS2** | **OS3** | **OS4** | **OS5** | **NOS** | **SIOS** | **OS1** | **OS2** | **OS3** | **OS4** | **OS5** |
| ATM | 257 | 49 | 76 | 182 | 172 | 182 | 201 | 206 | 53 | 81 | 184 | 175 | 184 | 201 | 206 |
| BlackJack | 56 | 15 | 19 | 22 | 19 | 22 | 19 | 22 | 15 | 19 | 22 | 19 | 22 | 19 | 22 |
| Calculator | 494 | 94 | 205 | 228 | 205 | 228 | 205 | 228 | 205 | 228 | 246 | 223 | 246 | 223 | 246 |
| CrossLexic | 470 | 176 | 206 | 209 | 209 | 209 | 209 | 209 | 182 | 211 | 214 | 214 | 214 | 214 | 214 |
| DFGraph Coverage | 683 | 274 | 274 | 415 | 274 | 415 | 274 | 415 | 274 | 274 | 415 | 274 | 415 | 274 | 415 |
| Dynamic Parser | 3,378 | 1,723 | 2,300 | 2,300 | 2,300 | 2,300 | 2,300 | 2,300 | 1,724 | 2,301 | 2,301 | 2,301 | 2,301 | 2,301 | 2,301 |
| Graph Coverage | 385 | 187 | 210 | 301 | 210 | 301 | 210 | 301 | 187 | 210 | 301 | 210 | 301 | 210 | 301 |
| JMines | 263 | 66 | 66 | 79 | 66 | 79 | 66 | 79 | 202 | 202 | 205 | 202 | 205 | 202 | 205 |
| Logic Coverage | 436 | 218 | 375 | 381 | 375 | 381 | 375 | 381 | 217 | 375 | 381 | 375 | 381 | 375 | 381 |
| MM Coverage | 845 | 143 | 251 | 262 | 251 | 262 | 251 | 262 | 143 | 251 | 262 | 251 | 262 | 251 | 262 |
| Poly | 259 | 128 | 249 | 250 | 250 | 250 | 250 | 250 | 131 | 250 | 250 | 250 | 250 | 250 | 250 |
| Snake | 572 | 164 | 225 | 421 | 400 | 421 | 400 | 421 | 216 | 226 | 422 | 401 | 422 | 401 | 422 |
| TicTacToe | 1,045 | 56 | 464 | 464 | 486 | 486 | 509 | 509 | 56 | 507 | 507 | 507 | 507 | 523 | 523 |
| Tree | 113 | 24 | 60 | 70 | 64 | 70 | 64 | 70 | 33 | 67 | 70 | 70 | 70 | 70 | 70 |
| Triangle | 263 | 4 | 128 | 140 | 166 | 168 | 166 | 168 | 4 | 128 | 140 | 171 | 172 | 171 | 172 |
| Vending Machine | 108 | 0 | 65 | 75 | 90 | 90 | 90 | 90 | 0 | 66 | 77 | 91 | 91 | 91 | 91 |
| **Total** | **9,627** | **3,321** | **5,173** | **5,799** | **5,537** | **5,864** | **5,589** | **5,911** | **3,556** | **5,391** | **5,997** | **5,734** | **6,043** | **5,776** | **6,081** |

Table 5.11: Numbers of Faults Found by Test Oracle Strategies - Part2

| **Programs** | **#Faults** | **# Faults Found by Test Oracle Strategies** | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | **Edge** | | | | | | | **EdgePair** | | | | | | |
| | | **NOS** | **SIOS** | **OT1** | **OT2** | **OT3** | **OT4** | **OT5** | **NOS** | **SIOS** | **OT1** | **OT2** | **OT3** | **OT4** | **OT5** |
| ATM | 257 | 49 | 76 | 175 | 171 | 178 | 198 | 206 | 53 | 81 | 177 | 173 | 180 | 201 | 206 |
| BlackJack | 56 | 15 | 19 | 21 | 19 | 21 | 19 | 22 | 15 | 19 | 21 | 19 | 21 | 19 | 22 |
| Calculator | 494 | 94 | 224 | 205 | 224 | 205 | 224 | 240 | 119 | 223 | 240 | 223 | 240 | 223 | 240 |
| CrossLexic | 470 | 176 | 206 | 209 | 206 | 209 | 206 | 209 | 182 | 211 | 214 | 211 | 214 | 211 | 214 |
| DFGraph Coverage | 683 | 274 | 274 | 415 | 274 | 415 | 274 | 415 | 274 | 274 | 415 | 274 | 415 | 274 | 415 |
| Dynamic Parser | 3,378 | 1,723 | 2,300 | 2,300 | 2,300 | 2,300 | 2,300 | 2,300 | 1,724 | 2,301 | 2,301 | 2,301 | 2,301 | 2,301 | 2,301 |
| Graph Coverage | 385 | 187 | 210 | 275 | 210 | 275 | 210 | 275 | 187 | 210 | 275 | 210 | 275 | 210 | 275 |
| JMines | 263 | 66 | 66 | 68 | 66 | 68 | 66 | 68 | 202 | 202 | 204 | 202 | 204 | 202 | 204 |
| Logic Coverage | 436 | 218 | 375 | 381 | 375 | 381 | 375 | 381 | 217 | 375 | 381 | 375 | 381 | 375 | 381 |
| MM Coverage | 845 | 143 | 251 | 260 | 251 | 260 | 251 | 260 | 143 | 251 | 260 | 251 | 260 | 251 | 260 |
| Poly | 259 | 128 | 249 | 249 | 249 | 249 | 249 | 249 | 131 | 250 | 250 | 250 | 250 | 250 | 250 |
| Snake | 572 | 164 | 225 | 286 | 225 | 286 | 225 | 286 | 216 | 226 | 286 | 226 | 286 | 226 | 286 |
| TicTacToe | 1,045 | 56 | 464 | 489 | 486 | 489 | 512 | 512 | 56 | 507 | 510 | 508 | 510 | 527 | 527 |
| Tree | 113 | 24 | 60 | 70 | 64 | 70 | 64 | 70 | 33 | 67 | 70 | 70 | 70 | 70 | 70 |
| Triangle | 263 | 4 | 128 | 150 | 148 | 158 | 148 | 158 | 4 | 128 | 158 | 154 | 166 | 154 | 166 |
| Vending Machine | 108 | 0 | 65 | 65 | 83 | 83 | 83 | 83 | 0 | 66 | 66 | 84 | 84 | 84 | 84 |
| **Total** | **9,627** | **3,321** | **5,173** | **5,637** | **5,332** | **5,666** | **5,385** | **5,718** | **3,556** | **5,391** | **5,828** | **5,531** | **5,857** | **5,578** | **5,901** |

Table 5.12: Effectiveness of Test Oracle Strategies - Part1

| **Programs** | **#Faults** | **Percentage of Faults detected by Test Oracle Strategies** | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | **Edge** | | | | | | | **EdgePair** | | | | | | |
| | | **NOS** | **SIOS** | **OS1** | **OS2** | **OS3** | **OS4** | **OS5** | **NOS** | **SIOS** | **OS1** | **OS2** | **OS3** | **OS4** | **OS5** |
| ATM | 257 | 0.19 | 0.30 | 0.71 | 0.67 | 0.71 | 0.78 | 0.80 | 0.21 | 0.32 | 0.72 | 0.68 | 0.72 | 0.78 | 0.80 |
| BlackJack | 56 | 0.27 | 0.34 | 0.39 | 0.34 | 0.39 | 0.34 | 0.39 | 0.27 | 0.34 | 0.39 | 0.34 | 0.39 | 0.34 | 0.39 |
| Calculator | 494 | 0.19 | 0.41 | 0.46 | 0.41 | 0.46 | 0.41 | 0.46 | 0.24 | 0.45 | 0.50 | 0.45 | 0.50 | 0.45 | 0.50 |
| CorssLexic | 470 | 0.37 | 0.44 | 0.44 | 0.44 | 0.44 | 0.44 | 0.44 | 0.39 | 0.45 | 0.46 | 0.46 | 0.46 | 0.46 | 0.46 |
| DFGraph Coverage | 683 | 0.40 | 0.40 | 0.61 | 0.40 | 0.61 | 0.40 | 0.61 | 0.40 | 0.40 | 0.61 | 0.40 | 0.61 | 0.40 | 0.61 |
| Dynamic Parser | 3,378 | 0.51 | 0.68 | 0.68 | 0.68 | 0.68 | 0.68 | 0.68 | 0.51 | 0.68 | 0.68 | 0.68 | 0.68 | 0.68 | 0.68 |
| Graph Coverage | 385 | 0.49 | 0.55 | 0.78 | 0.55 | 0.78 | 0.55 | 0.78 | 0.49 | 0.55 | 0.78 | 0.55 | 0.78 | 0.55 | 0.78 |
| JMines | 263 | 0.25 | 0.25 | 0.30 | 0.25 | 0.30 | 0.25 | 0.30 | 0.77 | 0.77 | 0.78 | 0.77 | 0.78 | 0.77 | 0.78 |
| Logic Coverage | 436 | 0.50 | 0.86 | 0.87 | 0.86 | 0.87 | 0.86 | 0.87 | 0.50 | 0.86 | 0.87 | 0.86 | 0.87 | 0.86 | 0.87 |
| MM Coverage | 845 | 0.17 | 0.30 | 0.31 | 0.30 | 0.31 | 0.30 | 0.31 | 0.17 | 0.30 | 0.31 | 0.30 | 0.31 | 0.30 | 0.31 |
| Poly | 259 | 0.49 | 0.96 | 0.97 | 0.97 | 0.97 | 0.97 | 0.97 | 0.51 | 0.97 | 0.97 | 0.97 | 0.97 | 0.97 | 0.97 |
| Snake | 572 | 0.29 | 0.39 | 0.74 | 0.70 | 0.74 | 0.70 | 0.74 | 0.38 | 0.40 | 0.74 | 0.70 | 0.74 | 0.70 | 0.74 |
| TicTacToe | 1,045 | 0.05 | 0.44 | 0.44 | 0.47 | 0.47 | 0.49 | 0.49 | 0.05 | 0.49 | 0.49 | 0.49 | 0.49 | 0.50 | 0.50 |
| Tree | 113 | 0.21 | 0.53 | 0.62 | 0.57 | 0.62 | 0.57 | 0.62 | 0.29 | 0.59 | 0.62 | 0.62 | 0.62 | 0.62 | 0.62 |
| Triangle | 263 | 0.02 | 0.49 | 0.53 | 0.63 | 0.64 | 0.63 | 0.64 | 0.02 | 0.49 | 0.53 | 0.65 | 0.65 | 0.65 | 0.65 |
| Vending Machine | 108 | 0.00 | 0.60 | 0.69 | 0.83 | 0.83 | 0.83 | 0.83 | 0.00 | 0.61 | 0.71 | 0.84 | 0.84 | 0.84 | 0.84 |
| **Average** | **9,627** | **0.34** | **0.54** | **0.60** | **0.58** | **0.61** | **0.58** | **0.61** | **0.37** | **0.56** | **0.62** | **0.60** | **0.63** | **0.60** | **0.63** |

Table 5.13: Effectiveness of Test Oracle Strategies - Part2

| **Programs** | **#Faults** | **Percentage of Faults detected by Test Oracle Strategies** | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | **Edge** | | | | | | | **EdgePair** | | | | | | |
| | | **NOS** | **SIOS** | **OT1** | **OT2** | **OT3** | **OT4** | **OT5** | **NOS** | **SIOS** | **OT1** | **OT2** | **OT3** | **OT4** | **OT5** |
| ATM | 257 | 0.19 | 0.30 | 0.68 | 0.67 | 0.69 | 0.77 | 0.80 | 0.21 | 0.32 | 0.69 | 0.67 | 0.70 | 0.78 | 0.80 |
| BlackJack | 56 | 0.27 | 0.34 | 0.38 | 0.34 | 0.38 | 0.34 | 0.39 | 0.27 | 0.34 | 0.38 | 0.34 | 0.38 | 0.34 | 0.39 |
| Calculator | 494 | 0.19 | 0.41 | 0.45 | 0.41 | 0.45 | 0.41 | 0.45 | 0.24 | 0.45 | 0.49 | 0.45 | 0.49 | 0.45 | 0.49 |
| CorssLexic | 470 | 0.37 | 0.44 | 0.44 | 0.44 | 0.44 | 0.44 | 0.44 | 0.39 | 0.45 | 0.46 | 0.45 | 0.46 | 0.45 | 0.46 |
| DFGraph Coverage | 683 | 0.40 | 0.40 | 0.61 | 0.40 | 0.61 | 0.40 | 0.61 | 0.40 | 0.40 | 0.61 | 0.40 | 0.61 | 0.40 | 0.61 |
| Dynamic Parser | 3,378 | 0.51 | 0.68 | 0.68 | 0.68 | 0.68 | 0.68 | 0.68 | 0.51 | 0.68 | 0.68 | 0.68 | 0.68 | 0.68 | 0.68 |
| Graph Coverage | 385 | 0.49 | 0.55 | 0.71 | 0.55 | 0.71 | 0.55 | 0.71 | 0.49 | 0.55 | 0.71 | 0.55 | 0.71 | 0.55 | 0.71 |
| JMines | 263 | 0.25 | 0.25 | 0.26 | 0.25 | 0.26 | 0.25 | 0.26 | 0.77 | 0.77 | 0.78 | 0.77 | 0.78 | 0.77 | 0.78 |
| Logic Coverage | 436 | 0.50 | 0.86 | 0.87 | 0.86 | 0.87 | 0.86 | 0.87 | 0.50 | 0.86 | 0.87 | 0.86 | 0.87 | 0.86 | 0.87 |
| MM Coverage | 845 | 0.17 | 0.30 | 0.31 | 0.30 | 0.31 | 0.30 | 0.31 | 0.17 | 0.30 | 0.31 | 0.30 | 0.31 | 0.30 | 0.31 |
| Poly | 259 | 0.49 | 0.96 | 0.96 | 0.96 | 0.96 | 0.96 | 0.96 | 0.51 | 0.97 | 0.97 | 0.97 | 0.97 | 0.97 | 0.97 |
| Snake | 572 | 0.29 | 0.39 | 0.50 | 0.39 | 0.50 | 0.39 | 0.50 | 0.38 | 0.40 | 0.50 | 0.40 | 0.50 | 0.40 | 0.50 |
| TicTacToe | 1,045 | 0.05 | 0.44 | 0.47 | 0.47 | 0.47 | 0.49 | 0.49 | 0.05 | 0.49 | 0.49 | 0.49 | 0.49 | 0.50 | 0.50 |
| Tree | 113 | 0.21 | 0.53 | 0.62 | 0.57 | 0.62 | 0.57 | 0.62 | 0.29 | 0.59 | 0.62 | 0.62 | 0.62 | 0.62 | 0.62 |
| Triangle | 263 | 0.02 | 0.49 | 0.57 | 0.56 | 0.60 | 0.56 | 0.60 | 0.02 | 0.49 | 0.60 | 0.59 | 0.63 | 0.59 | 0.63 |
| Vending Machine | 108 | 0.00 | 0.60 | 0.60 | 0.77 | 0.77 | 0.77 | 0.77 | 0.00 | 0.61 | 0.61 | 0.78 | 0.78 | 0.78 | 0.78 |
| **Average** | **9,627** | **0.34** | **0.54** | **0.59** | **0.55** | **0.59** | **0.56** | **0.59** | **0.37** | **0.56** | **0.61** | **0.57** | **0.61** | **0.58** | **0.61** |

To analyze the RQs statistically, I used Qqplots [90] to determine that the percentages of faults detected by the OSes for both EC and EPC were not normally distributed. The Qqplots are shown in Figures 5.4 through 5.27. Because these data deviate from a straight line, the percentages of faults detected by the OSes for EC and EPC were not normally distributed.

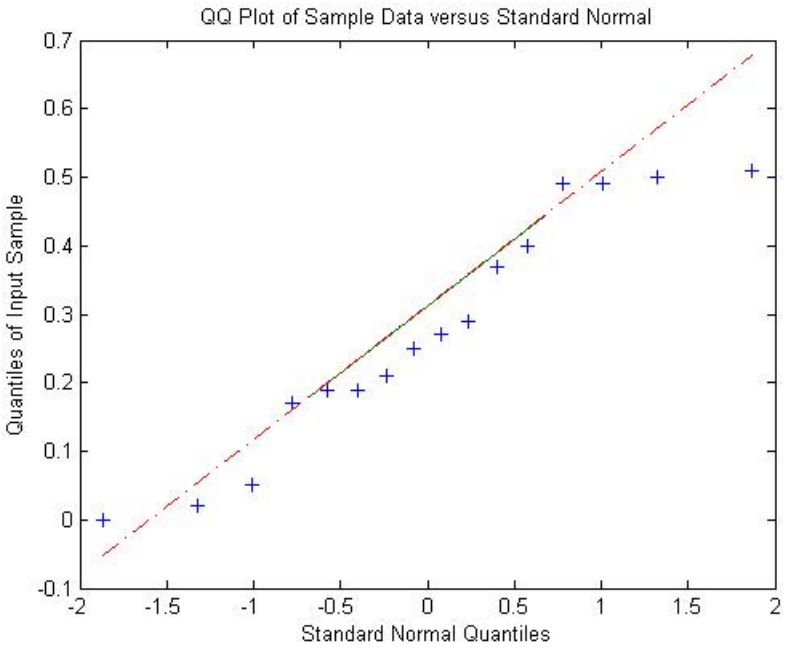


Figure 5.4: Qqplot for NOS of edge coverage

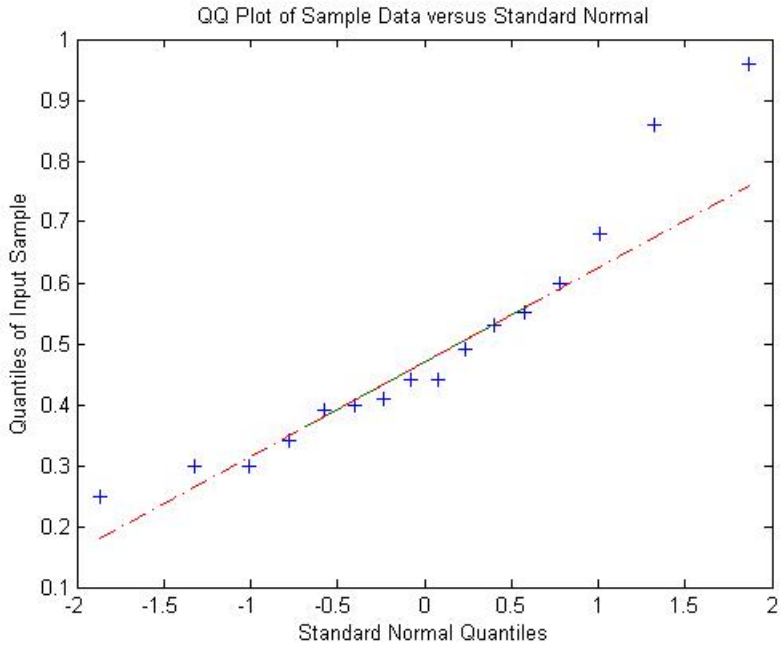


Figure 5.5: Qqplot for SIOS of edge coverage

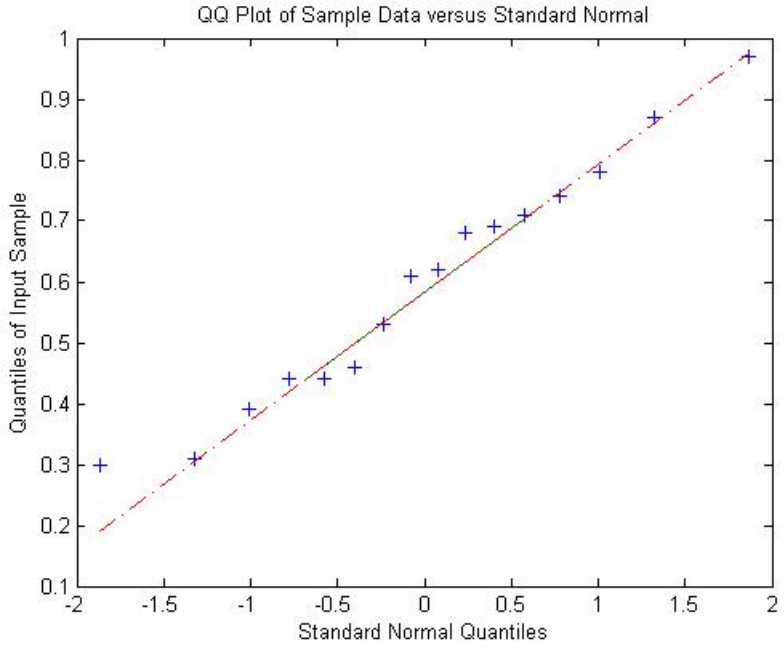


Figure 5.6: Qqplot for OS1 of edge coverage

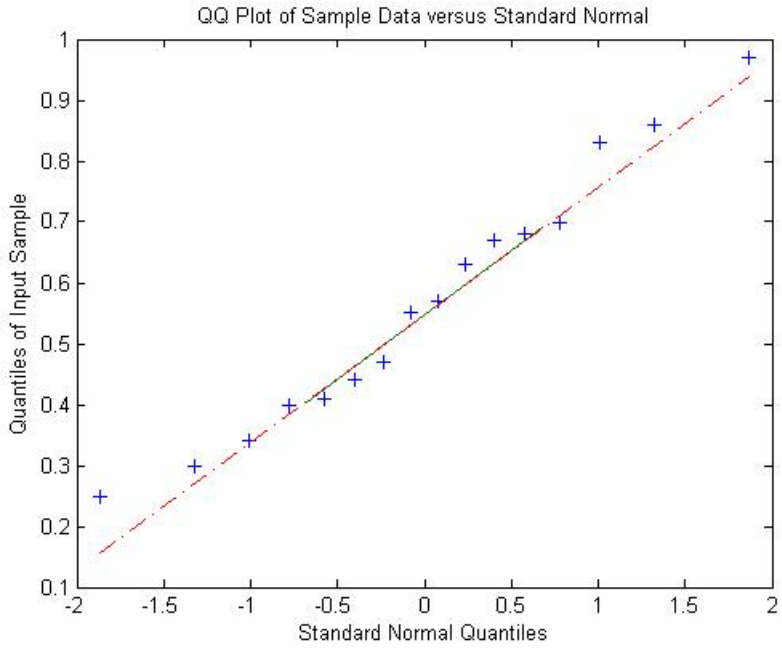


Figure 5.7: Qqplot for OS2 of edge coverage

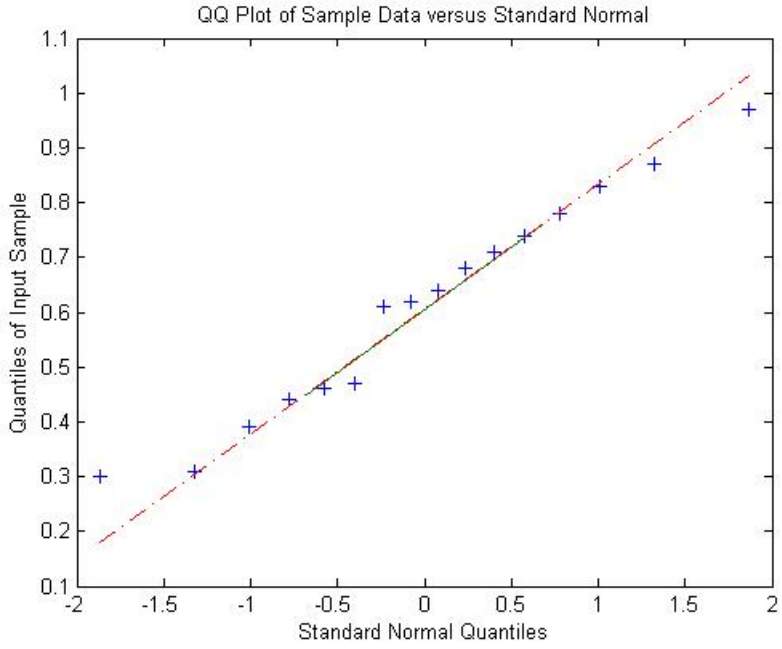


Figure 5.8: Qqplot for OS3 of edge coverage

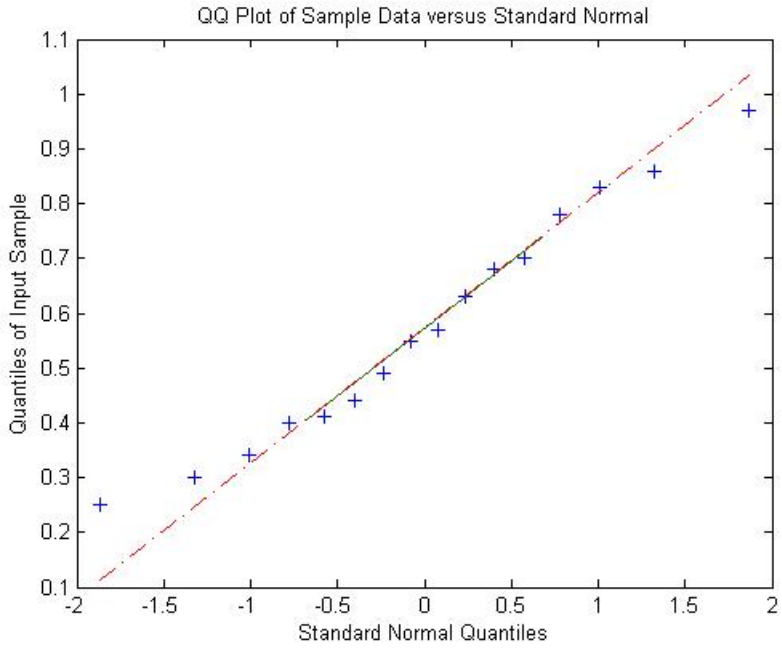


Figure 5.9: Qqplot for OS4 of edge coverage

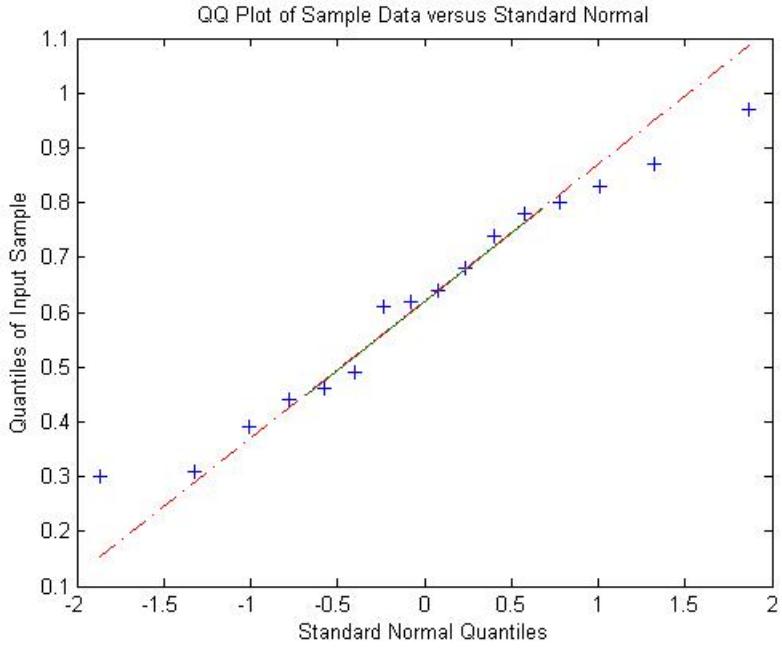


Figure 5.10: Qqplot for OS5 of edge coverage

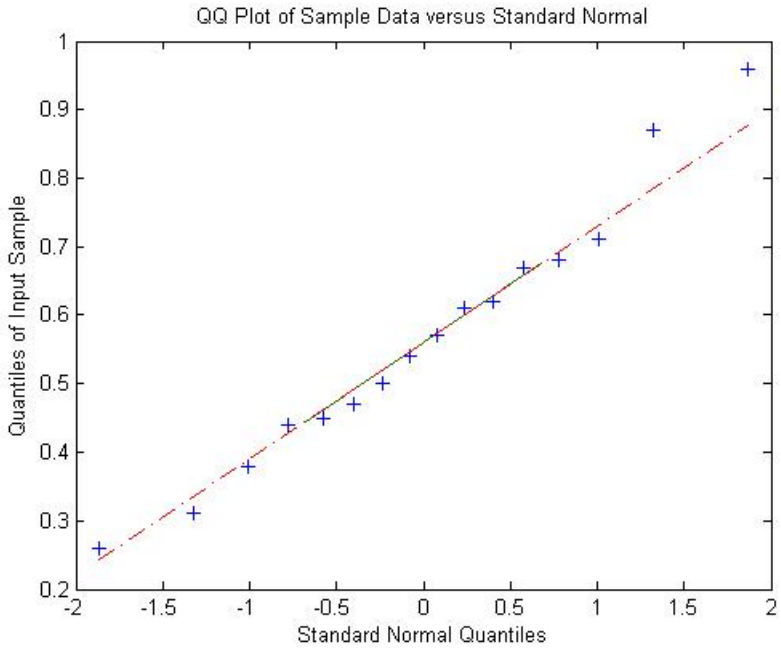


Figure 5.11: Qqplot for OT1 of edge coverage

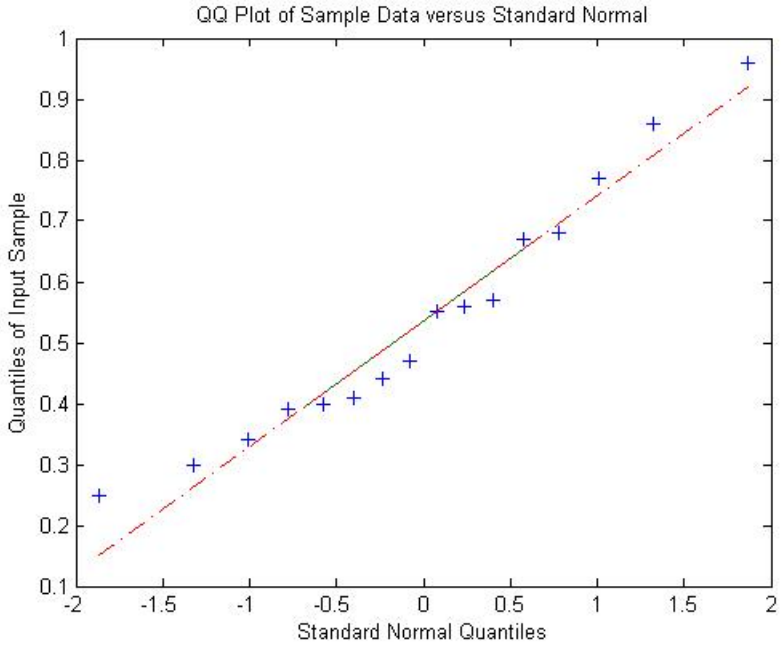


Figure 5.12: Qqplot for OT2 of edge coverage

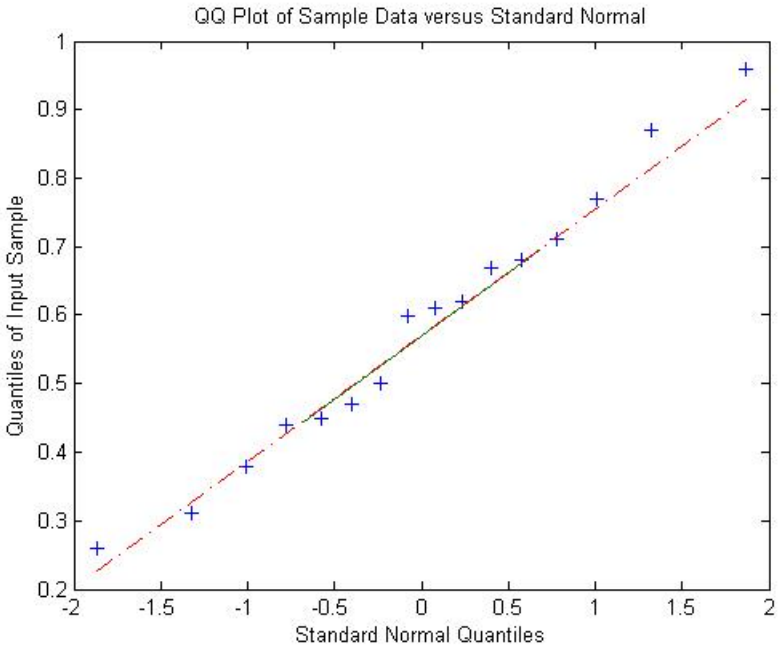


Figure 5.13: Qqplot for OT3 of edge coverage

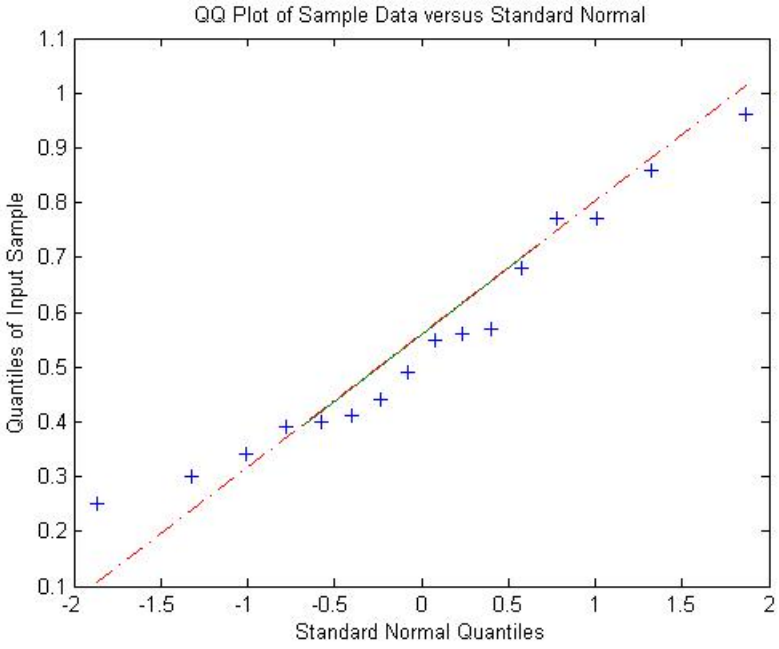


Figure 5.14: Qqplot for OT4 of edge coverage

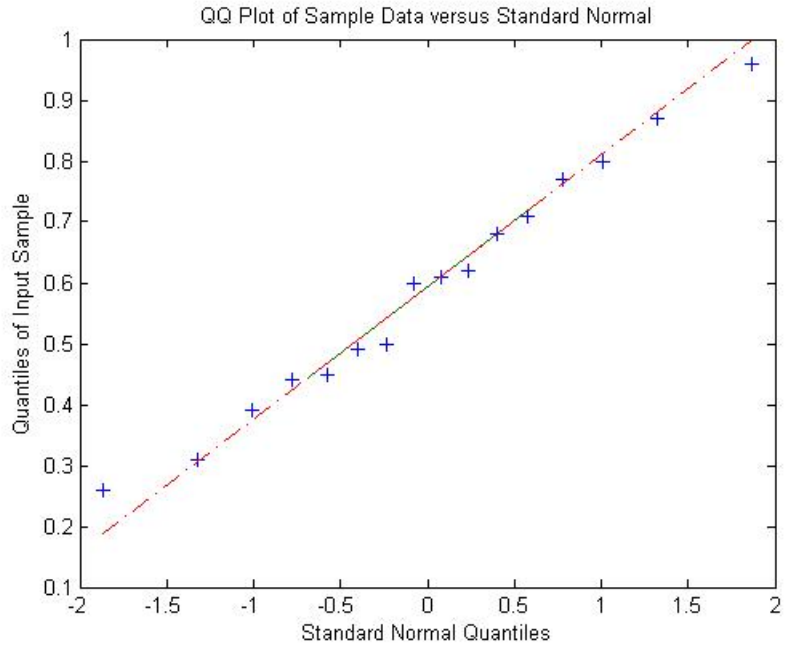


Figure 5.15: Qqplot for OT5 of edge coverage

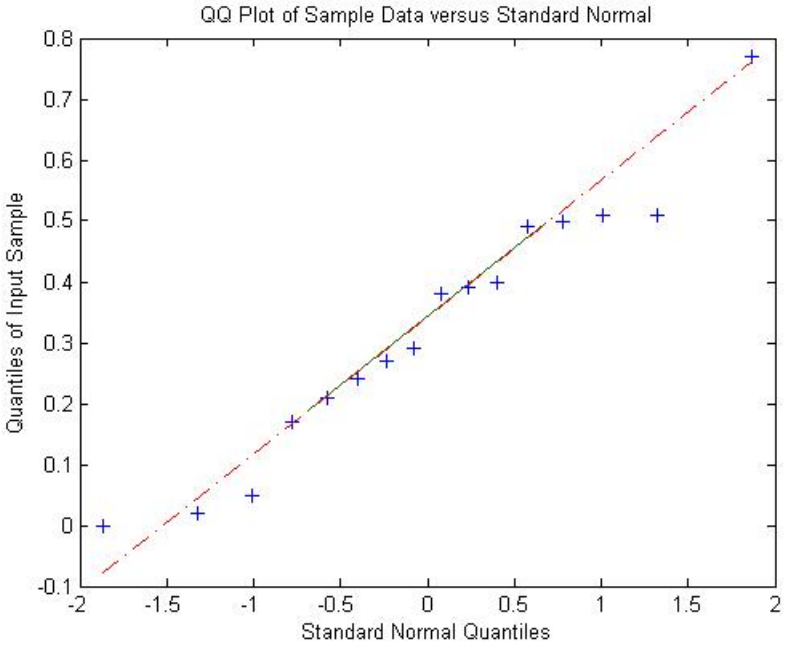


Figure 5.16: Qqplot for NOS of edge-pair coverage

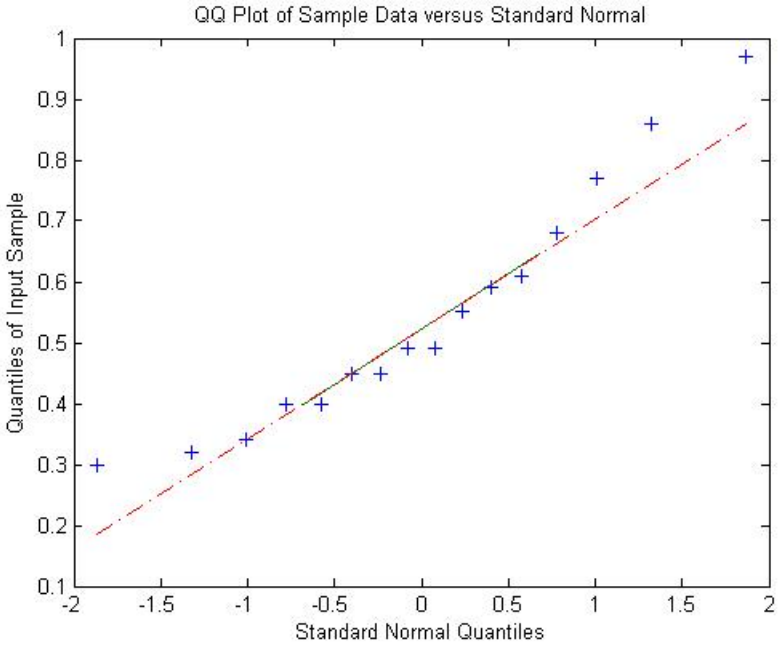


Figure 5.17: Qqplot for SIOS of edge-pair coverage

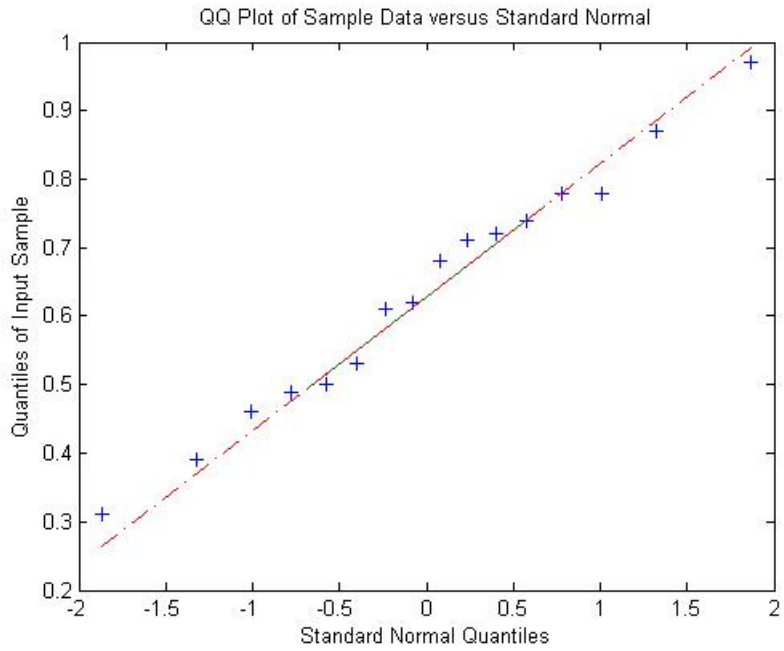


Figure 5.18: Qqplot for OS1 of edge-pair coverage

To get statistical evidence of the effectiveness difference between a less precise OS and a more precise OS, I used the one-tailed Wilcoxon signed-rank test (statistical significance level $\alpha = 0.05$) [93] to compare the paired percentages of the faults detected by two different OSes for both EC and EPC. The reason that I used the one-tailed Wilcoxon signed-rank test was that the compared data were paired and came from the same tests (EC or EPC tests). For instance, SIOS for EC was compared with OS1 for EC. This is a non-parametric test to assess whether two population means differ when data are not normally distributed. This test first finds the absolute difference for each pair and gets the number of the pairs that are different, N. Then this test ranks the pairs and calculates the test statistic W. If N is greater than 9, the sampling distribution of W is a reasonably close approximation of the normal distribution and the one-tail probability p can be calculated. If N is less than or equal to 9, but greater than 4, the calculated W value is compared to a $W_{critical}$ from the separate table of critical values of W.

For $Hypotheses_A$, the OS pairs {NOS, SIOS}, {SIOS, OS1}, {SIOS, OS3}, {SIOS, OS5},

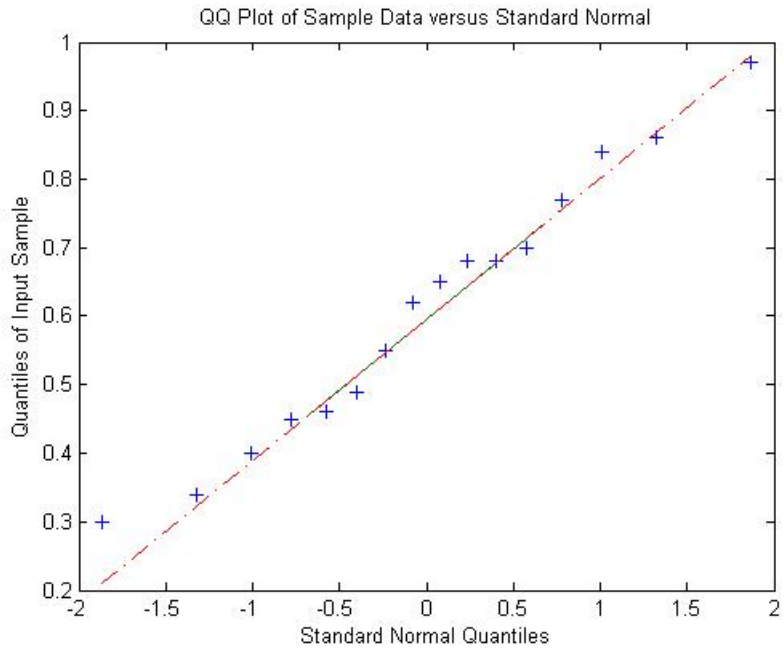


Figure 5.19: Qqplot for OS2 of edge-pair coverage

{OS1, OS3}, {OS3, OS5}, {OS1, OS5}, {OS2, OS5}, {OS4, OS5}, {SIOS, OS2}, {SIOS, OS4}, {OS2, OS3}, {OS2, OS4}, {SIOS, OT1}, {SIOS, OT3}, {SIOS, OT5}, {OT1, OT3}, {OT3, OT5}, {OT1, OT5}, {OT2, OT5}, {OT4, OT5}, {SIOS, OT2}, {SIOS, OT4}, {OT2, OT3}, and {OT2, OT4} were compared for EC and EPC using the Wilcoxon signed rank test. I got the p-values from 0.0002 - 0.0038 for {NOS, SIOS}, {SIOS, OS1}, {SIOS, OS3}, {SIOS, OS5}, {OS2, OS5}, {OS4, OS5}, {OS2, OS3}, {SIOS, OT1}, {SIOS, OT3}, {SIOS, OT5}, {OT2, OT5}, {OT4, OT5}, and {OT2, OT3} for EC and EPC because the N values of these pairs were greater than 10. So I can reject $H_0$. For these pairs, the effectiveness of a more precise OS is significantly greater than that of a less precise OS. Pairs {SIOS, OS2} and {SIOS, OS4} for EC and EPC had N less than 10 but greater than 4, thus, the table of critical values of W was used, finding that the differences between the OSes in these pairs were not due to chance.

I found statistical evidence that there were no significant differences between {OT1,

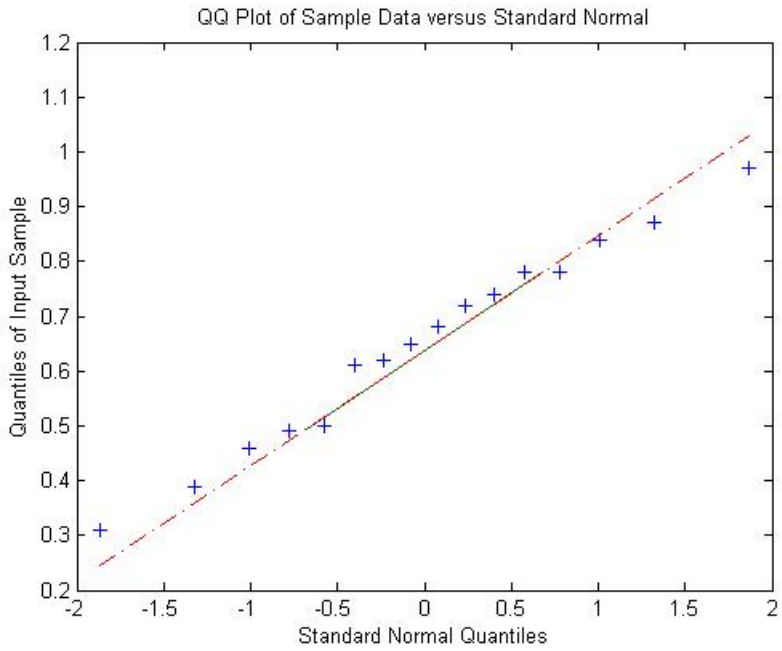


Figure 5.20: Qqplot for OS3 of edge-pair coverage

OT5} and {SIOS, OT4} for EC and EPC. For pairs {OS1, OS3}, {OS3, OS5}, {OS1, OS5}, {OS2, OS4}, {OT1, OT3}, {OT3, OT5}, {SIOS, OT2}, and {OT2, OT4} for EC and EPC, the N values of OSes were fewer than five, so I could not perform the Wilcoxon signed-rank test. This also implied that there was no significant difference between these pairs.

Some OSes pairs had inconsistent results for EC and EPC. For pair {SIOS, OT2}, the EC tests showed that OT2 is as effective as SIOS using the Wilcoxon signed-rank test but the EPC tests showed that N for this pair was less than 5, thus, the Wilcoxon signed rank test could not be applied to this pair for the EPC tests.

In summary, for $Hypotheses_A$, I reject $H_0$ for the pairs {NOS, SIOS}, {SIOS, OS1}, {SIOS, OS3}, {SIOS, OS5}, {OS2, OS5}, {OS4, OS5}, {OS2, OS3}, {SIOS, OT1}, {SIOS, OT3}, {SIOS, OT5}, {OT2, OT5}, {OT4, OT5}, {OT2, OT3}, {SIOS, OS2}, and {SIOS, OS4}.

Although Tables 5.12 and 5.13 show that a more precise OS can detect a higher average

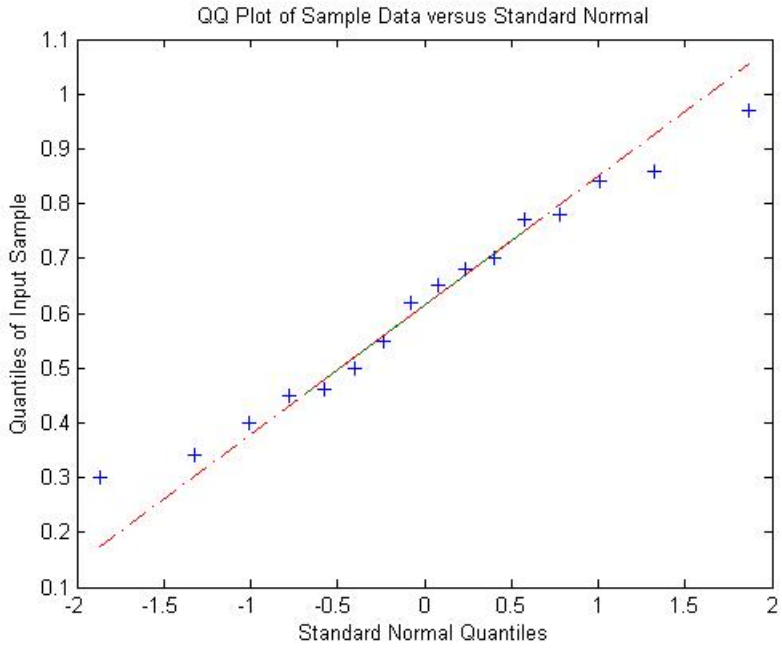


Figure 5.21: Qqplot for OS4 of edge-pair coverage

percentage of faults than a less precise OS, the results of the Wilcoxon signed-rank test showed that the percentage of faults detected by a more precise OS might not be significantly different from that of a less precise OS (such as {OT1, OT5}). Therefore, the answer to RQ1 is: for any two OSes that have different precision, the more precise OS is not necessarily more effective than the less precise OS.

I also used the Wilcoxon signed rank test for $Hypotheses_B$ to decide if the frequency of checking variables impacts the effectiveness of OSes for 10 pairs {OT1, OS1}, {OT2, OS2}, {OT3, OS3}, {OT4, OS4}, and {OT5, OS5} for EC and EPC. The results showed that I can reject $H_0$ for four pairs: {OT1, OS1}, {OT3, OS3}, {OT5, OS5} for EC and {OT3, OS3} for EPC. I was not able to reject $H_0$ for the other four pairs: {OT4, OS4} for EC and {OT1, OS1}, {OT2, OS2}, and {OT5, OS5} for EPC. The numbers of different pairs of {OT2, OSs} for EC and {OT4, OS4} for EPC were less than 5, thus, the Wilcoxon signed rank test could not be applied to these two pairs. From Tables 5.12 and 5.13, the difference between the average percentages of faults detected by $OT_i$ and $OS_i$ $(1 \leq i \leq 5)$ for both

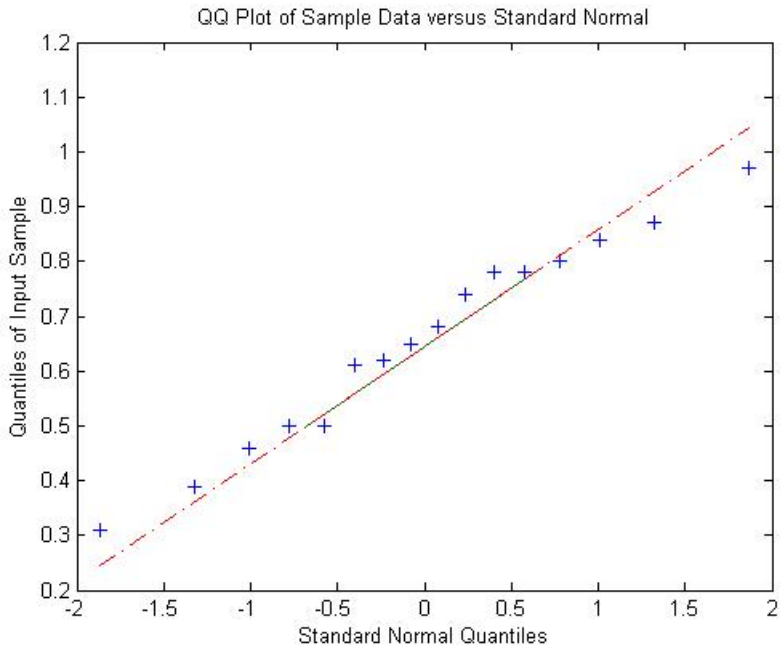


Figure 5.22: Qqplot for OS5 of edge-pair coverage

EC and EPC is very small. With regard to RQ2, checking program states multiple times was not significantly more effective than checking the same program states once.

Tables 5.12 and 5.13 show that each OS for the edge-adequate tests reveals almost the same number of faults as the same OS for the EP-adequate tests. I used the one-tailed Mann-Whitney test (statistical significance level $\alpha = 0.05$) [93] to look for statistical evidence that EPC is more effective than EC. The reason that I used Mann-Whitney test was that the comparison was between two independent tests: EC and EPC tests, with the same OS. I applied each OS to the edge-adequate and EP-adequate tests for each program and then compared the percentages of the faults detected by the two paired sets of tests that have the same OS. The p-values were between 0.1977 and 0.5823 for all the OSes, so I cannot reject $H_0$ for $Hypotheses_C$. Therefore, the answer to RQ3 is that a stronger coverage criterion (EPC) was not found to be more effective than a weaker criterion (EC) with the same OS.

Tables 5.14 and 5.15 show how many distinct assertions were created by hand for each

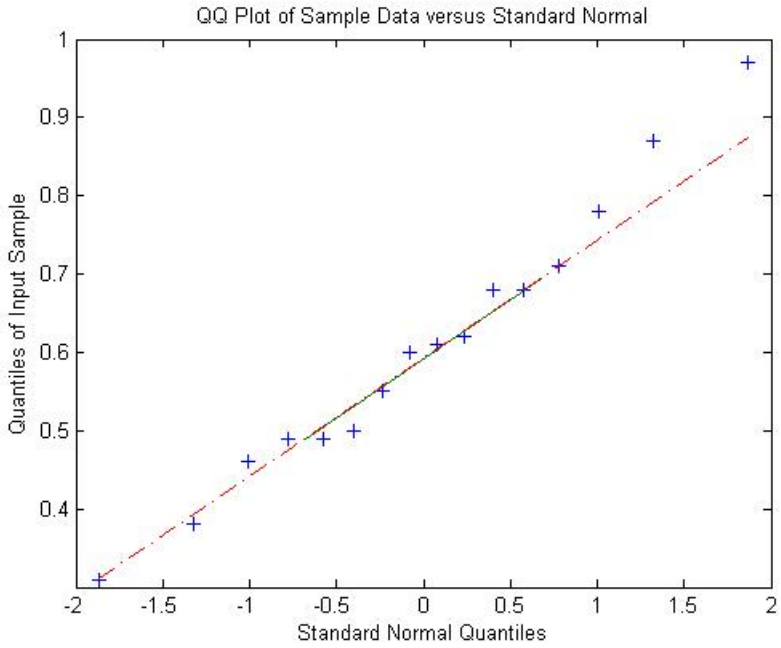


Figure 5.23: Qqplot for OT1 of edge-pair coverage

OS. Since a more precise OS checks more outputs and internal state variables than a less precise OS, more distinct assertions were generated for the more precise OSes. Thus, the cost of OS5 was greater than that of any other OS except OT5. Although OT5 checks program states less frequently than OS5, it uses the same number of distinct assertions as OS5. For each $OTi$, it uses the same number of distinct assertions as $OSi$, where $1 \leq i \leq 5$ in this research.

Since testers need to write test oracles for all outputs and internal state variables that appear in all transitions for each OT test oracle strategy, testers have to analyze the effects of each transition by the end of each test. For the experimental subjects, I had to analyze each transition and write the same test oracles as for OSi $1 \leq i \leq 5$, because each distinct transition produced different program states by the end of the tests. Thus, the cost of $OTi$ is equivalent to that of $OSi$. Furthermore, OS1, OS3, OS5, OT1, OT3, and OT5 required similar numbers of distinct assertions but have far more assertions than OS2, OS4, OT2, and OT4. This was because object members checked by OS1, OS3, OS5, OT1, OT3, and

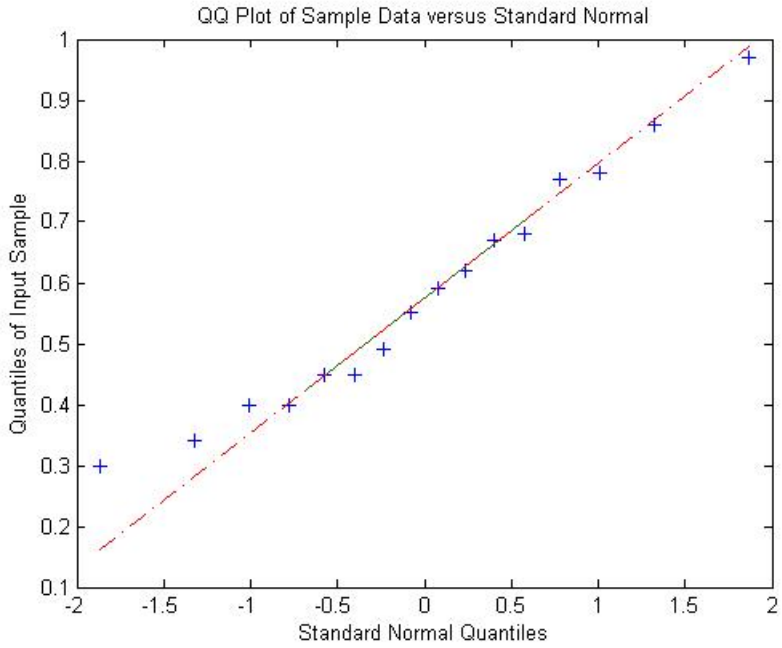


Figure 5.24: Qqplot for OT2 of edge-pair coverage

OT5 produced lots of assertions since I checked the member variables of objects recursively. For the same OS, the EP-adequate tests did not have many more distinct assertions than the edge-adequate tests. This was because I did not need to create many more mappings to satisfy EPC than the mappings created for EC, as discussed in section 5.3.2.

For completeness, Tables 5.16 and 5.17 show the total numbers of assertions used for each OS. NOS and SIOS are shown in both Tables 5.16 and 5.17 to make comparisons among the total numbers of assertions used for test oracle strategies easier.

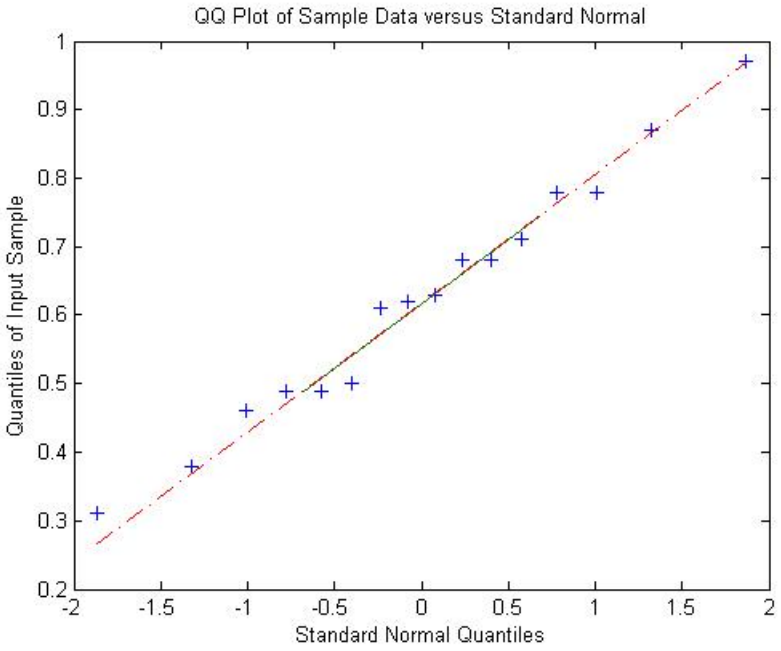


Figure 5.25: Qqplot for OT3 of edge-pair coverage

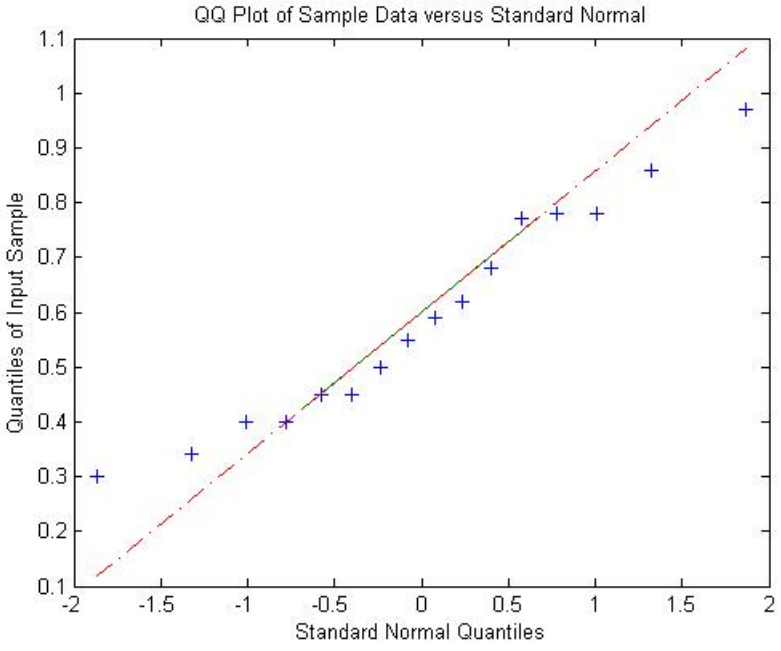


Figure 5.26: Qqplot for OT4 of edge-pair coverage

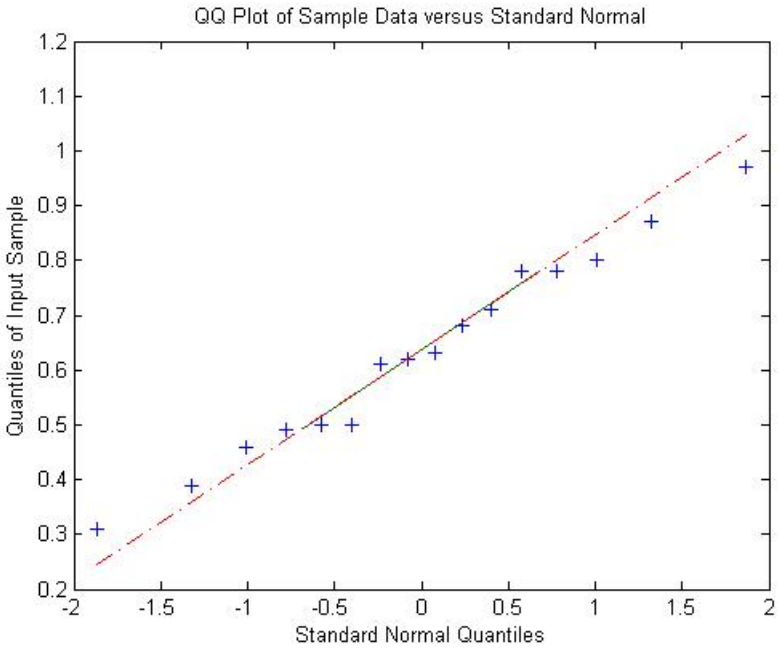


Figure 5.27: Qqplot for OT5 of edge-pair coverage

Table 5.14: Cost of Test Oracle Strategies – Total Number of Distinct Assertions - Part1

| **Programs** | **Cost of Test Oracle Strategies** | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | **Edge** | | | | | | | **EdgePair** | | | | | | |
| | **NOS** | **SIOS** | **OS1** | **OS2** | **OS3** | **OS4** | **OS5** | **NOS** | **SIOS** | **OS1** | **OS2** | **OS3** | **OS4** | **OS5** |
| ATM | 0 | 6 | 49 | 34 | 49 | 50 | 74 | 0 | 6 | 49 | 34 | 49 | 50 | 74 |
| BlackJack | 0 | 3 | 552 | 3 | 552 | 3 | 552 | 0 | 3 | 552 | 3 | 552 | 0 | 552 |
| Calculator | 0 | 9 | 123 | 9 | 123 | 9 | 123 | 0 | 9 | 123 | 9 | 123 | 9 | 123 |
| CorssLexic | 0 | 7 | 127 | 7 | 127 | 7 | 127 | 0 | 7 | 127 | 7 | 127 | 7 | 127 |
| DFGraph Coverage | 0 | 7 | 163 | 7 | 163 | 7 | 163 | 0 | 7 | 163 | 7 | 163 | 7 | 163 |
| Dynamic Parser | 0 | 15 | 23 | 15 | 23 | 15 | 23 | 0 | 15 | 23 | 15 | 23 | 15 | 23 |
| Graph Coverage | 0 | 11 | 156 | 11 | 156 | 11 | 156 | 0 | 11 | 156 | 11 | 156 | 11 | 156 |
| JMines | 0 | 1 | 792 | 82 | 792 | 82 | 792 | 0 | 1 | 792 | 82 | 792 | 82 | 792 |
| Logic Coverage | 0 | 8 | 181 | 8 | 181 | 8 | 181 | 0 | 8 | 181 | 8 | 181 | 8 | 181 |
| MM Coverage | 0 | 16 | 587 | 16 | 587 | 16 | 587 | 0 | 16 | 584 | 16 | 587 | 16 | 587 |
| Poly | 0 | 6 | 12 | 10 | 12 | 10 | 12 | 0 | 6 | 14 | 12 | 14 | 12 | 14 |
| Snake | 0 | 8 | 463 | 8 | 463 | 8 | 463 | 0 | 8 | 463 | 8 | 463 | 8 | 463 |
| TicTacToe | 0 | 3 | 33 | 6 | 36 | 60 | 90 | 0 | 3 | 33 | 6 | 36 | 60 | 90 |
| Tree | 0 | 3 | 23 | 14 | 34 | 14 | 34 | 0 | 3 | 23 | 14 | 34 | 14 | 34 |
| Triangle | 0 | 5 | 42 | 6 | 51 | 9 | 51 | 0 | 5 | 42 | 6 | 51 | 6 | 51 |
| Vending Machine | 0 | 6 | 17 | 9 | 17 | 9 | 18 | 0 | 6 | 19 | 9 | 20 | 9 | 21 |
| **Total** | **0** | **114** | **3,343** | **164** | **3,366** | **234** | **3,446** | **0** | **114** | **3,347** | **166** | **3,371** | **236** | **3,451** |

Table 5.15: Cost of Test Oracle Strategies – Total Number of Distinct Assertions - Part2

| **Programs** | **Cost of Test Oracle Strategies** | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | **Edge** | | | | | | | **EdgePair** | | | | | | |
| | **NOS** | **SIOS** | **OT1** | **OT2** | **OT3** | **OT4** | **OT5** | **NOS** | **SIOS** | **OT1** | **OT2** | **OT3** | **OT4** | **OT5** |
| ATM | 0 | 6 | 49 | 34 | 49 | 50 | 74 | 0 | 6 | 49 | 34 | 49 | 50 | 74 |
| BlackJack | 0 | 3 | 552 | 3 | 552 | 3 | 552 | 0 | 3 | 552 | 3 | 552 | 0 | 552 |
| Calculator | 0 | 9 | 123 | 9 | 123 | 9 | 123 | 0 | 9 | 123 | 9 | 123 | 9 | 123 |
| CorssLexic | 0 | 7 | 127 | 7 | 127 | 7 | 127 | 0 | 7 | 127 | 7 | 127 | 7 | 127 |
| DFGraph Coverage | 0 | 7 | 163 | 7 | 163 | 7 | 163 | 0 | 7 | 163 | 7 | 163 | 7 | 163 |
| Dynamic Parser | 0 | 15 | 23 | 15 | 23 | 15 | 23 | 0 | 15 | 23 | 15 | 23 | 15 | 23 |
| Graph Coverage | 0 | 11 | 156 | 11 | 156 | 11 | 156 | 0 | 11 | 156 | 11 | 156 | 11 | 156 |
| JMines | 0 | 1 | 792 | 82 | 792 | 82 | 792 | 0 | 1 | 792 | 82 | 792 | 82 | 792 |
| Logic Coverage | 0 | 8 | 181 | 8 | 181 | 8 | 181 | 0 | 8 | 181 | 8 | 181 | 8 | 181 |
| MM Coverage | 0 | 16 | 587 | 16 | 587 | 16 | 587 | 0 | 16 | 584 | 16 | 587 | 16 | 587 |
| Poly | 0 | 6 | 12 | 10 | 12 | 10 | 12 | 0 | 6 | 14 | 12 | 14 | 12 | 14 |
| Snake | 0 | 8 | 463 | 8 | 463 | 8 | 463 | 0 | 8 | 463 | 8 | 463 | 8 | 463 |
| TicTacToe | 0 | 3 | 33 | 6 | 36 | 60 | 90 | 0 | 3 | 33 | 6 | 36 | 60 | 90 |
| Tree | 0 | 3 | 23 | 14 | 34 | 14 | 34 | 0 | 3 | 23 | 14 | 34 | 14 | 34 |
| Triangle | 0 | 5 | 42 | 6 | 51 | 9 | 51 | 0 | 5 | 42 | 6 | 51 | 6 | 51 |
| Vending Machine | 0 | 6 | 17 | 9 | 17 | 9 | 18 | 0 | 6 | 19 | 9 | 20 | 9 | 21 |
| **Total** | **0** | **114** | **3,343** | **164** | **3,366** | **234** | **3,446** | **0** | **114** | **3,347** | **166** | **3,371** | **236** | **3,451** |

Table 5.16: Cost of Test Oracle Strategies – Total Number of Assertions - Part1

| **Programs** | **Cost of Test Oracle Strategies** | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | **Edge** | | | | | | | **EdgePair** | | | | | | |
| | **NOS** | **SIOS** | **OS1** | **OS2** | **OS3** | **OS4** | **OS5** | **NOS** | **SIOS** | **OS1** | **OS2** | **OS3** | **OS4** | **OS5** |
| ATM | 0 | 23 | 156 | 51 | 156 | 83 | 188 | 0 | 40 | 250 | 103 | 250 | 175 | 322 |
| BlackJack | 0 | 27 | 1,313 | 27 | 1,313 | 27 | 1,313 | 0 | 52 | 2,473 | 52 | 2,585 | 52 | 2,585 |
| Calculator | 0 | 168 | 1,908 | 168 | 1,908 | 168 | 1,908 | 0 | 894 | 10,047 | 1,313 | 10,047 | 1,323 | 10,047 |
| CorssLexic | 0 | 210 | 1,304 | 410 | 1,304 | 410 | 1,304 | 0 | 757 | 4,852 | 1,261 | 4,852 | 1,261 | 4,852 |
| DFGraph Coverage | 0 | 79 | 1,091 | 167 | 1,091 | 167 | 1,091 | 0 | 644 | 9,016 | 1,139 | 9,016 | 1,139 | 9,016 |
| Dynamic Parser | 0 | 409 | 456 | 409 | 456 | 409 | 456 | 0 | 1,234 | 1,417 | 1,234 | 1,417 | 1,234 | 1,417 |
| Graph Coverage | 0 | 208 | 1,796 | 448 | 1,796 | 448 | 1,796 | 0 | 606 | 4,904 | 951 | 4,904 | 951 | 4,904 |
| JMines | 0 | 42 | 6,428 | 805 | 6,428 | 805 | 6,428 | 0 | 152 | 21,553 | 1,911 | 21,553 | 1,911 | 21,553 |
| Logic Coverage | 0 | 95 | 1,471 | 155 | 1,471 | 155 | 1,471 | 0 | 484 | 7,766 | 674 | 7,766 | 674 | 7,766 |
| MM Coverage | 0 | 229 | 4,911 | 541 | 5,208 | 541 | 5,208 | 0 | 571 | 12,746 | 1,139 | 13,226 | 1,139 | 13,226 |
| Poly | 0 | 58 | 81 | 91 | 91 | 91 | 91 | 0 | 238 | 333 | 357 | 357 | 357 | 357 |
| Snake | 0 | 121 | 3,229 | 450 | 3,229 | 450 | 3,229 | 0 | 342 | 9,128 | 718 | 9,128 | 718 | 9,128 |
| TicTacToe | 0 | 8 | 160 | 16 | 167 | 267 | 437 | 0 | 17 | 236 | 33 | 252 | 609 | 828 |
| Tree | 0 | 49 | 209 | 87 | 247 | 87 | 247 | 0 | 147 | 587 | 329 | 769 | 329 | 769 |
| Triangle | 0 | 37 | 310 | 90 | 315 | 90 | 315 | 0 | 272 | 2,290 | 538 | 2,340 | 538 | 2,340 |
| Vending Machine | 0 | 89 | 156 | 96 | 163 | 96 | 170 | 0 | 211 | 380 | 229 | 398 | 229 | 421 |
| **Total** | **0** | **1,852** | **24,979** | **4,011** | **25,343** | **4,294** | **25,652** | **0** | **6,661** | **87,978** | **11,991** | **88,860** | **12,639** | **89,531** |

Table 5.17: Cost of Test Oracle Strategies – Total Number of Assertions - Part2

| **Programs** | **Cost of Test Oracle Strategies** | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | **Edge** | | | | | | | **EdgePair** | | | | | | |
| | **NOS** | **SIOS** | **OT1** | **OT2** | **OT3** | **OT4** | **OT5** | **NOS** | **SIOS** | **OT1** | **OT2** | **OT3** | **OT4** | **OT5** |
| ATM | 0 | 23 | 68 | 42 | 69 | 80 | 92 | 0 | 40 | 85 | 56 | 87 | 99 | 119 |
| BlackJack | 0 | 27 | 475 | 27 | 475 | 27 | 475 | 0 | 52 | 556 | 52 | 556 | 52 | 556 |
| Calculator | 0 | 168 | 322 | 322 | 322 | 322 | 322 | 0 | 894 | 1,323 | 1,313 | 1,323 | 1,323 | 1,323 |
| CorssLexic | 0 | 210 | 420 | 210 | 420 | 210 | 420 | 0 | 757 | 1,288 | 757 | 1,288 | 757 | 1,288 |
| DFGraph Coverage | 0 | 79 | 262 | 79 | 262 | 79 | 262 | 0 | 644 | 1,675 | 644 | 1,675 | 644 | 1,675 |
| Dynamic Parser | 0 | 409 | 428 | 409 | 428 | 409 | 428 | 0 | 1,234 | 1,260 | 1,234 | 1,260 | 1,234 | 1,260 |
| Graph Coverage | 0 | 208 | 423 | 208 | 423 | 208 | 423 | 0 | 606 | 920 | 606 | 920 | 606 | 920 |
| JMines | 0 | 42 | 792 | 42 | 792 | 42 | 792 | 0 | 152 | 792 | 42 | 792 | 42 | 1,802 |
| Logic Coverage | 0 | 95 | 801 | 95 | 801 | 95 | 801 | 0 | 484 | 1,870 | 484 | 1,870 | 484 | 1,870 |
| MM Coverage | 0 | 229 | 3,188 | 229 | 3,188 | 229 | 3,188 | 0 | 571 | 4,832 | 571 | 4,832 | 571 | 4,832 |
| Poly | 0 | 58 | 63 | 68 | 68 | 68 | 68 | 0 | 238 | 250 | 250 | 250 | 250 | 250 |
| Snake | 0 | 121 | 359 | 121 | 359 | 121 | 359 | 0 | 342 | 595 | 342 | 595 | 342 | 595 |
| TicTacToe | 0 | 8 | 80 | 12 | 80 | 80 | 85 | 0 | 17 | 111 | 23 | 112 | 118 | 118 |
| Tree | 0 | 49 | 111 | 62 | 111 | 62 | 111 | 0 | 147 | 211 | 194 | 211 | 194 | 211 |
| Triangle | 0 | 37 | 85 | 42 | 85 | 42 | 90 | 0 | 272 | 488 | 295 | 488 | 295 | 511 |
| Vending Machine | 0 | 89 | 90 | 94 | 94 | 94 | 94 | 0 | 211 | 212 | 219 | 219 | 219 | 219 |
| **Total** | **0** | **1,852** | **7,967** | **2,062** | **7,977** | **2,168** | **8,010** | **0** | **6,661** | **16,468** | **7,092** | **16,478** | **7,230** | **17,559** |

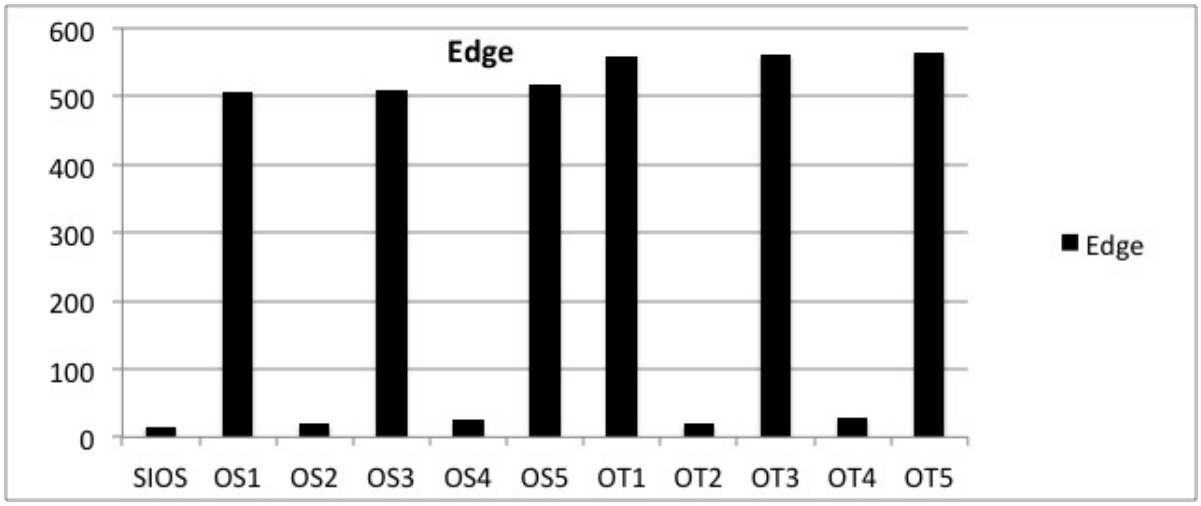


Figure 5.28: Averages of Cost-effectiveness of Edge-adequate Tests

Figures 5.28 and 5.29 give the average cost-effectiveness (cost / the percentage of faults detected by OSes) from formula 1 in section 5.3.1 over all the programs for EC and EPC. Figures 5.30 and 5.31 show the differences for OSes whose cost-effectiveness values are below 100. Since the cost of generating test oracles for NOS is 0, the cost-effectiveness was not used for NOS. Figures 5.28 and 5.29 show that SIOS is the most cost-effective, followed by OS2, OS4, OT2, and OT4. The details about the cost-effectiveness of OSes for EC and EPC are shown in Tables 5.18 and 5.19. Note that the figures reflect the cost-effectiveness of OSes when they were generated by STALE, not by hand. More discussion will be in section 5.3.5.

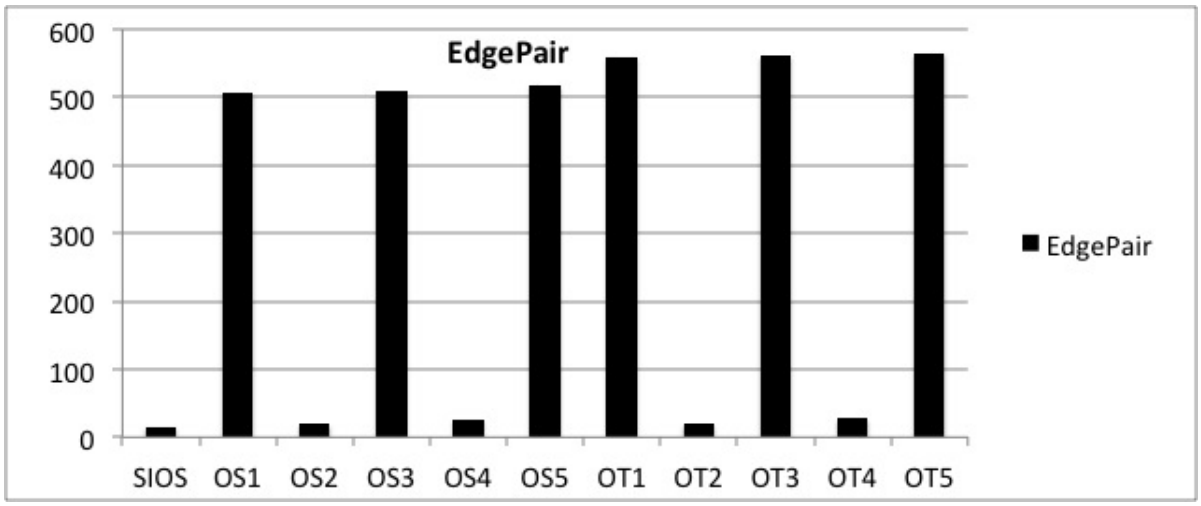


Figure 5.29: Averages of Cost-effectiveness of EdgePair-adequate Tests

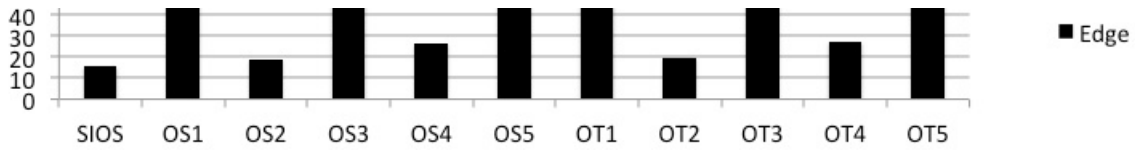


Figure 5.30: Averages of Cost-effectiveness of Edge-adequate Tests Below 100

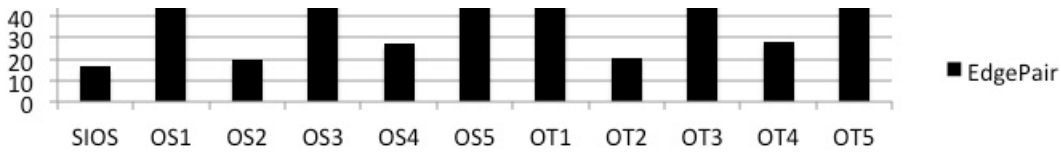


Figure 5.31: Averages of Cost-effectiveness of EdgePair-adequate Tests Below 100

Table 5.18: Cost-effectiveness of Test Oracle Strategies - Part1

| **Programs** | **Cost-effectiveness of Test Oracle Strategies** | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | **Edge** | | | | | | | **EdgePair** | | | | | | |
| | **NOS** | **SIOS** | **OS1** | **OS2** | **OS3** | **OS4** | **OS5** | **NOS** | **SIOS** | **OS1** | **OS2** | **OS3** | **OS4** | **OS5** |
| ATM | 0.0 | 20.3 | 69.2 | 50.8 | 69.2 | 63.9 | 92.3 | 0.0 | 19.0 | 68.4 | 49.9 | 68.4 | 63.9 | 92.3 |
| BlackJack | 0.0 | 8.8 | 1405.1 | 8.8 | 1405.1 | 8.8 | 1405.1 | 0.0 | 8.8 | 1405.1 | 8.8 | 1405.1 | 8.8 | 1405.1 |
| Calculator | 0.0 | 21.7 | 266.5 | 21.7 | 266.5 | 21.7 | 266.5 | 0.0 | 19.9 | 247.0 | 19.9 | 247.0 | 19.9 | 247.0 |
| CorssLexic | 0.0 | 16.0 | 285.6 | 15.7 | 285.6 | 15.7 | 285.6 | 0.0 | 15.6 | 278.9 | 15.4 | 278.9 | 15.4 | 278.9 |
| DFGraph Coverage | 0.0 | 17.4 | 268.3 | 17.4 | 268.3 | 17.4 | 268.3 | 0.0 | 17.4 | 268.3 | 17.4 | 268.3 | 17.4 | 268.3 |
| Dynamic Parser | 0.0 | 22.0 | 33.8 | 22.0 | 33.8 | 22.0 | 33.8 | 0.0 | 22.0 | 33.8 | 22.0 | 33.8 | 22.0 | 33.8 |
| Graph Coverage | 0.0 | 20.2 | 199.5 | 20.2 | 199.5 | 20.2 | 199.5 | 0.0 | 20.2 | 199.5 | 20.2 | 199.5 | 20.2 | 199.5 |
| JMines | 0.0 | 4.0 | 2636.7 | 4.0 | 2636.7 | 4.0 | 2636.7 | 0.0 | 1.3 | 1016.1 | 1.3 | 1016.1 | 1.3 | 1016.1 |
| Logic Coverage | 0.0 | 9.3 | 207.1 | 9.3 | 207.1 | 9.3 | 207.1 | 0.0 | 9.3 | 207.1 | 9.3 | 207.1 | 9.3 | 207.1 |
| MM Coverage | 0.0 | 53.9 | 1893.2 | 53.9 | 1893.2 | 53.9 | 1893.2 | 0.0 | 53.9 | 1893.2 | 53.9 | 1893.2 | 53.9 | 1893.2 |
| Poly | 0.0 | 6.2 | 12.4 | 10.4 | 12.4 | 10.4 | 12.4 | 0.0 | 6.2 | 14.5 | 12.4 | 14.5 | 12.4 | 14.5 |
| Snake | 0.0 | 20.3 | 629.1 | 11.4 | 629.1 | 11.4 | 629.1 | 0.0 | 20.3 | 627.6 | 11.4 | 627.6 | 11.4 | 627.6 |
| TicTacToe | 0.0 | 6.8 | 74.3 | 12.9 | 77.4 | 123.2 | 184.8 | 0.0 | 6.2 | 68.0 | 12.4 | 74.2 | 119.9 | 179.8 |
| Tree | 0.0 | 5.7 | 37.1 | 24.7 | 54.9 | 24.7 | 54.9 | 0.0 | 5.1 | 37.1 | 22.6 | 54.9 | 22.6 | 54.9 |
| Triangle | 0.0 | 10.3 | 78.9 | 9.5 | 79.8 | 9.5 | 79.8 | 0.0 | 10.3 | 78.9 | 9.2 | 78.0 | 9.2 | 78.0 |
| Vending Machine | 0.0 | 10.0 | 24.5 | 10.8 | 20.4 | 10.8 | 21.6 | 0.0 | 9.8 | 26.6 | 10.7 | 23.7 | 10.7 | 24.9 |
| **Average** | *N/A* | 15.8 | 507.6 | 19.0 | 508.7 | 26.7 | 516.9 | *N/A* | 15.3 | 404.4 | 18.6 | 405.6 | 26.2 | 413.8 |

Table 5.19: Cost-effectiveness of Test Oracle Strategies - Part2

| **Programs** | **Cost-effectiveness of Test Oracle Strategies** | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | **Edge** | | | | | | | **EdgePair** | | | | | | |
| | **NOS** | **SIOS** | **OT1** | **OT2** | **OT3** | **OT4** | **OT5** | **NOS** | **SIOS** | **OT1** | **OT2** | **OT3** | **OT4** | **OT5** |
| ATM | 0.0 | 20.3 | 72.0 | 51.1 | 70.7 | 63.9 | 92.3 | 0.0 | 19.0 | 71.1 | 50.5 | 70.0 | 63.9 | 92.3 |
| BlackJack | 0.0 | 8.8 | 1472.0 | 8.8 | 1472.0 | 8.8 | 1405.1 | 0.0 | 8.8 | 1472.0 | 8.8 | 1472.0 | 8.8 | 1405.1 |
| Calculator | 0.0 | 21.7 | 271.3 | 21.7 | 271.3 | 21.7 | 271.3 | 0.0 | 19.9 | 253.2 | 19.9 | 253.2 | 19.9 | 253.2 |
| CorssLexic | 0.0 | 16.0 | 285.6 | 16.0 | 285.6 | 16.0 | 285.6 | 0.0 | 15.6 | 278.9 | 15.6 | 278.9 | 15.6 | 278.9 |
| DFGraph Coverage | 0.0 | 17.4 | 268.3 | 17.4 | 268.3 | 17.4 | 268.3 | 0.0 | 17.4 | 268.3 | 17.4 | 268.3 | 17.4 | 268.3 |
| Dynamic Parser | 0.0 | 22.0 | 33.8 | 22.0 | 33.8 | 22.0 | 33.8 | 0.0 | 22.0 | 33.8 | 22.0 | 33.8 | 22.0 | 33.8 |
| Graph Coverage | 0.0 | 20.2 | 218.4 | 20.2 | 218.4 | 20.2 | 218.4 | 0.0 | 20.2 | 218.4 | 20.2 | 218.4 | 20.2 | 218.4 |
| JMines | 0.0 | 4.0 | 3063.2 | 4.0 | 3063.2 | 4.0 | 3063.2 | 0.0 | 1.3 | 1021.1 | 1.3 | 1021.1 | 1.3 | 1021.1 |
| Logic Coverage | 0.0 | 9.3 | 207.1 | 9.3 | 207.1 | 9.3 | 207.1 | 0.0 | 9.3 | 207.1 | 9.3 | 207.1 | 9.3 | 207.1 |
| MM Coverage | 0.0 | 53.9 | 1907.8 | 53.9 | 1907.8 | 53.9 | 1907.8 | 0.0 | 53.9 | 1907.8 | 53.9 | 1907.8 | 53.9 | 1907.8 |
| Poly | 0.0 | 6.2 | 12.5 | 10.4 | 12.5 | 10.4 | 12.5 | 0.0 | 6.2 | 14.5 | 12.4 | 14.5 | 12.4 | 14.5 |
| Snake | 0.0 | 20.3 | 926.0 | 20.3 | 926.0 | 20.3 | 926.0 | 0.0 | 20.3 | 926.0 | 20.2 | 926.0 | 20.2 | 926.0 |
| TicTacToe | 0.0 | 6.8 | 70.5 | 12.9 | 76.9 | 122.5 | 184.8 | 0.0 | 6.2 | 67.6 | 12.3 | 73.8 | 119.9 | 178.5 |
| Tree | 0.0 | 5.7 | 37.1 | 24.7 | 54.9 | 24.7 | 54.9 | 0.0 | 5.1 | 37.1 | 22.6 | 54.9 | 22.6 | 54.9 |
| Triangle | 0.0 | 10.3 | 73.6 | 10.7 | 84.9 | 10.7 | 84.9 | 0.0 | 10.3 | 69.9 | 10.2 | 80.8 | 10.2 | 80.8 |
| Vending Machine | 0.0 | 10.0 | 28.2 | 11.7 | 22.1 | 11.7 | 23.4 | 0.0 | 9.8 | 31.1 | 11.6 | 25.7 | 11.6 | 27.0 |
| **Average** | *N/A* | 15.8 | 559.2 | 19.7 | 561.0 | 27.4 | 564.9 | *N/A* | 15.3 | 429.9 | 19.3 | 431.6 | 26.8 | 435.5 |

### 5.3.5 Discussion and Recommendations

I found statistical evidence that the more precise OS was more effective in terms of the percentage of faults detected than the less precise OS for the pairs {NOS, SIOS}, {SIOS, OS1}, {SIOS, OS3}, {SIOS, OS5}, {OS2, OS5}, {OS4, OS5}, {OS2, OS3}, {SIOS, OT1}, {SIOS, OT3}, {SIOS, OT5}, {OT2, OT5}, {OT4, OT5}, {OT2, OT3}, {SIOS, OS2}, {SIOS, OS4}, {OT1, OS3}, {OT1, OS5}, {OT2, OS3}, {OT2, OS4}, {OT3, OS5}, and {OT4, OS5} but not for {OS1, OS3}, {OS3, OS5}, {OS1, OS5}, {OS2, OS4}, {OT1, OT3}, {OT3, OT5}, {SIOS, OT2}, and {OT2, OT4}, {OT1, OT5} and {SIOS, OT4}. This shows that if the precision difference of two OSes is not large enough, there is no difference between their effectiveness.

In Table 5.12, the percentage of faults detected (mutation score) by the EP-adequate tests for the most precise OS (OS5) in this paper is only 0.63. This is a low score considering that 90% mutation is considered a good test set [7]. The test inputs were generated to satisfy state invariants in the state machine diagrams while mutation-adequate tests usually require more test inputs. This implies that mutation coverage is generally more effective at finding faults than EC and EPC on the model. A previous paper [94] found that mutation can find more faults than EPC at the unit testing level. Furthermore, the system tests generated in this paper could only call methods that are mapped to the models at a high level.

Tables 5.12 and 5.13 show that NOS was not very good at revealing faults. This implies that checking runtime exceptions is not enough. I also noticed that SIOS can reveal more than 80% of the faults detected by OS5 but with many fewer assertions. Test inputs were generated to satisfy state invariants, thus checking the limited number of outputs and internal state variables used in the state invariants can reveal many faults. In contrast, checking more program states (as OS5 does) that are not affected by the test inputs are not likely to reveal more faults. If more program states have to be checked, checking return values or parameter members (OS2 and OS4) gets better cost-effectiveness while checking object members (OS1, OS3, OS5, OT1, OT3, and OT5) requires a lot more cost but gains

little improvement in effectiveness.

The results show that checking outputs and internal state variables multiple times was not significantly more effective than checking the same outputs and internal state variables once for both EC and EPC. My belief is that program states are changed when execution enters a different state of a state machine diagram. Therefore, most faults should be revealed if all outputs and internal state variables are checked for each distinct diagram state only once, as done by $OTi$, where $1 \leq i \leq 5$. Checking the same outputs and internal state variables multiple times does not seem to help much. Therefore, if testers use a tool such as STALE to generate test oracles automatically, they should generate test oracles after each transition to achieve higher effectiveness; otherwise, the testers should check outputs and internal state variables at the end of tests to avoid writing redundant test oracles manually for transitions that appear multiple times in tests.

Moreover, I found statistical evidence that EP-adequate tests were not significantly more effective than edge-adequate tests. This may be because EPC did not require many more mappings on the state machine diagrams, even though EPC had more tests. The models could affect the results. If a model has lots of nodes that have multiple incoming and outgoing edges, the edge-pairs could be very different from edges. Then EPC could result in stronger tests than EC. Furthermore, if a state machine diagram is designed with multiple variables, then each state represents multiple variable states (constraints). EPC abstract tests are usually more complex than EC abstract tests, and tour the same states multiple times. Satisfying the constraints (finding appropriate test values) in complex tests is harder because it causes more program states, when compared with less complex tests. In this case, EPC may need more mappings than EC.

Checking program states frequently could require too many assertions in tests. OT5 resulted in 8,010 assertions for EC (17,559 for EPC) but OS5 produced 25,652 (89,531 for EPC), as shown in Tables 5.16 and 5.17. Checking lots of program states could cause the size of a JUnit test method to exceed 65,536 bytes, resulting in a compiler error. Since testers must split these methods by hand, this adds an additional hidden costs for the

precision.

Figures 5.28 and 5.29 show that SIOS was the most cost-effective for both EC and EPC. So if testers need to save time, SIOS is a good choice. Otherwise, testers should choose OS2 or OS4 because they are almost as effective as OS5 but require fewer assertions than OS5.

### 5.3.6 Threats to Validity

As in most software engineering studies, I could not be sure that the subject programs are representative. The results may differ with other programs and models. Another threat to external validity is that I generated test values by hand to satisfy EC and EPC. The results such as the answers to RQ3 may differ if I used different tests. One construct validity threat is that I used muJava to generate synthetic faults. Using real faults or another mutation tool may yield different results. Another internal threat is that I identified equivalent mutants by hand, thus, mistakes could affect the results in small ways.

# Chapter 6: Conclusions and Thoughts

This chapter concludes the research, specifies the contributions, and talks about the future work.

## 6.1 Conclusions

This research studied three sub-problems for generating cost-effective criteria-based tests from behavioral models: the minimum cost test paths problem (MCTP), the mapping problem, and the test oracle problem. This research also presented solutions to each problem and results from experiments to evaluate the solutions. Three groups of conclusions for each problem are summarized below.

First, this research defined a new problem, the MCTP problem, and proved its hardness. Four variants were proved to be NP-complete and one variant can be solved in polynomial time. This research also presented and compared three solutions to the MCTP problem: the breadth-first search based, the prefix-graph based, and the set-covering based solutions. They were compared on 37 methods, 18 from open source programs and 19 constructed by hand. Graphs were created by hand, and prime paths were generated automatically as test requirements. Then all three solutions were used to generate test paths for the same set of prime paths. Both the prefix-graph based and set-covering based solutions generated fewer test paths than the breadth-first search based solution, however neither the prefix-graph nor the set-covering solution was always better than the other and I could not quantify which solution is preferable or when. The set-covering based solution took an extremely long time with the largest sets of prime paths, so the prefix-graph based solution seems preferable. Thus, this solution was implemented in the graph coverage web application [12]. This web app is used by thousands of students and professors who use Ammann and Offutt's book

[7].

Second, this research created a general, practical solution to the mapping problem when behavioral models are used: a structured test automation language (STAL). Testers use STAL to define mappings between identifiable elements in the abstract tests to specific sequences of code that will be part of the concrete executable tests. STAL was implemented in a structured test automation language framework, STALE, which reads models and automatically creates fully executable concrete tests.

The test automation language can be used whenever abstract tests include the same elements many times, resulting in duplicate components of concrete tests. This research explained STAL in the context of using graph-based test criteria defined on graphs that were derived from UML state machine diagrams, but it can also be used with other models and other techniques for designing model-based tests.

This research also compared test generation using STAL with manual test generation. The results, based on 17 programs, showed that automatic test generation used 29.6% of the time that manual test generation used. The manual tests also contained 48 errors in which concrete tests did not map abstract tests correctly.

Third, this research proposed ten new test oracle strategies (OSes) to help find more faults with less cost in model-based testing. These new OSes were compared with two baseline OSes: the null and state invariant test oracle strategies (NOS and SIOS). Test inputs were generated to satisfy edge coverage (EC) and edge-pair coverage (EPC) from UML state machine diagrams for 16 programs. Then the twelve OSes were applied to the edge-adequate and EP-adequate tests, resulting in 24 sets of tests for each program. These sets of tests were run against faults. The effectiveness and cost of each OS were recorded and the cost-effectiveness was calculated in section 5.3.

I had several findings from the experiments for the test oracle problem. First, NOS was not very effective and testers need to check program states, not just runtime exceptions. Second, the more precise OSes did not always detect more failures than the less precise OSes. Third, an OS that checks outputs and internal state variables multiple times was not

more effective than OSes that check the same outputs and internal state variables just once. Fourth, with the same OS, a specific stronger test coverage criterion (EPC) was not more effective at finding faults than a weaker criterion (EC). In summary, checking only runtime exceptions will result in many failures not being noticed. Checking only state invariants (SIOS) is recommended for testers who are willing to sacrifice a little quality for time; otherwise, testers should check state invariants, outputs and parameter objects. If testers use a tool such as STALE to generate test oracles automatically, they should generate test oracles after each transition to achieve higher effectiveness; otherwise, the testers should check outputs and internal state variables at the end of tests to avoid writing redundant test oracles manually for transitions that appear multiple times in tests.

In summary, the solutions to the three sub-problems result in more cost-effective tests than previous solutions. The prefix-graph and set-covering based solutions can generate abstract tests with less cost in terms of the number of tests and total number of nodes in the tests. STAL can be used to transform abstract tests to concrete tests with less time and fewer errors. Cost-effective OSes were also developed for the concrete tests.

## 6.2 Contributions and Recommendations

This research gave a complete solution to the problem of generating cost-effective criteria-based tests from behavioral models, (including three sub-problems: the minimum cost test paths problem, the mapping problem, and the test oracle problem). These solutions include theoretical proofs, algorithms, a new language, new test oracle strategies, and practical implementation. Specifically, the contributions and recommendations are listed below:

1. Defined the minimum cost test paths problem (MCTP), which generalizes common objectives for reducing costs when generating tests from graphs. Five variants of MCTP were also developed.

2. Proved four variants of MCTP to be NP-complete and the optimal solution of the other to be solvable in polynomial time

3. Developed the set-covering and prefix-graph based solutions to MCTP and implemented the solutions in the graph coverage web application

4. Experimentally evaluated the new solutions to MCTP and recommended the prefix-graph based solution

5. Invented a structured test automation language (STAL) to automate the mapping problem when only behavioral models are used

6. Implemented the structured test automation language (STAL) in the structured test automation language framework (STALE)

7. Experimentally compared STAL with manual transformation method from abstract tests to concrete tests and recommended STAL

8. Proposed ten new test oracle strategies (OSes)

9. Compared the new test oracle strategies with two baseline test oracle strategies. Checking only state invariants (SIOS) is recommended for testers who are willing to sacrifice a little quality for time; otherwise, testers should check state invariants, outputs and parameter objects. Testers should check program states after each transition if test oracles can be generated automatically using a tool; otherwise, check the program states after the last transition.

10. Extended "Reachability-Infection-Propagation (RIP)" model to "Reachability-Infection-Propogation-Revealability (RIPR)" model.

This research has three significant general contributions. First, this research gives a complete solution for generating tests from behavioral models without any additional supporting models. Second, this solution greatly reduces the cost in comparison with previous solutions, both theoretically and experimentally. Third, a structured test automation language framework (STALE) was developed for practical use.

The list below shows the publications related to my dissertation:

1. Nan Li and Jeff Offutt. An Empirical Analysis of Test Oracle Strategies for Model-based Testing. 7th IEEE International Conference on Software Testing, Verification and Validation. Cleveland, Ohio, USA. April 2014.

2. Nan Li and Jeff Offutt. A Test Automation Language for Behavioral Models. Submitted to MODELS 2014.

3. Nan Li, Fei Li, and Jeff Offutt. Better Algorithms to Minimize the Cost of Test Paths. 5th IEEE International Conference on Software Testing, Verification and Validation. Montreal, Quebec, Canada. April 2012.

4. Nan Li. A Smart Structured Test Automation Language (SSTAL). The Ph.D. Symposium of 5th IEEE International Conference on Software Testing, Verification and Validation. Montreal, Quebec, Canada. April 2012.

The list below shows my other publications:

1. Marcio Eduardo Delamaro, Lin Deng, Nan Li, Vinicius Durelli and Jeff Offutt. Experimental Evaluation of SDL and One-Op Mutation for C. 7th IEEE International Conference on Software Testing, Verification and Validation. Cleveland, Ohio, USA. April 2014.

2. Vinicius Durelli, Jeff Offutt, Nan Li, and Marcio Delamaro. Predicates in the Wild: Gauging the Cost-Effectiveness of Applying Active Clause Coverage Criteria to Open-source Java Programs. Under revision for the Journal of Systems and Software.

3. Nan Li, Xin Meng, Jeff Offutt, and Lin Deng. Is Bytecode Instrumentation as Good as Source Code Instrumentation: An Empirical Study With Industrial Tools (Experience Report). The 24th IEEE International Symposium on Software Reliability Engineering. Pasadena, CA, USA. November 2013.

4. Lin Deng, Jeff Offutt, and Nan Li. Empirical Evaluation of the Statement Deletion Mutation Operator. 6th IEEE International Conference on Software Testing, Verification and Validation. Luxembourg, Luxembourg. March 2013.

5. William Shelton, Nan Li, Paul Ammann, and Jeff Offutt. Adding Criteria-based Tests to TDD. The Testing: Academic and Industrial Conference - Practice and Research Techniques (TAIC-PART 2012). Montreal, Quebec, Canada. April 2012. (Best Student Paper Award).

6. Jeff Offutt, Nan Li, Paul Ammann and Wuzhi Xu. Using Abstraction and Web Applications to Teach Criteria-Based Test Design. 24th IEEE-CS Conference on Software Engineering Education and Training. Hawaii, USA. May 2011.

7. Jingyu Hu, Nan Li, and Jeff Offutt. An Analysis of OO Mutation Operators. Mutation 2011: 6th International Workshop on Mutation Analysis. Berlin, Germany. March 2011.

8. Nan Li, Upsorn Praphamontripong, and Jeff Offutt. An Experimental Comparison of Four Unit Test Criteria: Mutation, Edge-Pair, All-uses and Prime Path Coverage. Mutation 2009: 4th International Workshop on Mutation Analysis. Denver, Colorado. April 2009.

## 6.3 Future Work

For the minimum cost test paths problem, it would be helpful to adapt other algorithms from the shortest superstring literature. It would also be helpful to quantify properties of methods so that testers could pre-determine which algorithm would work better. The number of test requirements (prime paths) is an easy measure, but does not seem to be accurate. Checking overlaps among the test requirements may work better. Further experimentation could also measure other factors, such as the number of nodes and the structural complexity. It is also possible that using dynamic programming and different test suite minimization and splitting algorithms could yield different solutions. Furthermore, I plan to adapt the prefix-graph based solution to solve other MCTP variants, including the fewest total nodes, the minimizing maximum ratio of TR to TP subject to a number of test paths, the fewest test paths subject to a bounded ratio of TR to TP, and the fewest total nodes subject to a

bounded TR to TP ratio problems. Additionally, I also would like to modify the algorithms to solve the infeasible paths problem (discussed in section 1.3.1).

In the future, I would like to extend the STALE to accept more diagrams such as UML activity diagrams and use more programming languages such as C++. Moreover, I will be looking for the possiblities to use this framework in the real world.

With regard to the test oracle problem, in the future, I would like to develop new OSes and study their effectiveness and cost. For the research questions raised in this research, I want to try more coverage criteria to check if the results differ. Using mutation analysis to select which program states to check seems promising [72, 73], but could be costly because testers have to run mutation analysis before providing test oracle data. I intend to seek a way to select OSes that are effective but less costly.

I generated faults using mutation analysis in the experiments for the test oracle problem. I would like to see if failures revealed by OSes could change if I use the *minimal mutants* [95]. I also would like to use *one-op mutation* [96] to generate faults. I propose a new way to generate mutants based on one-op mutation, called *adaptive one-op mutation*. One-op mutation uses only one fixed mutation operator to all statements of a program. Adaptive one-op mutation analyzes the structure of statements first, and then apply different mutation operators to statements that have different structures (one mutation operator for one structure). For instance, if a structure has a predicate with multiple clauses, using relational operator replacement (ROR) mutation operator [97] may detect more faults than using statement deletion (SDL) mutation operator [98] with not much more costs.

Table A.1: Which Elements Need Mappings?

| | **Element** |
|---|---|
| Need mappings | Entry Point, Exit Point, and Do Activity of a State, Transition, Constraint |
| Do not need mappings | State Machine, Region, Initial PseudoState, Final State, Fork, Join, Junction, Choice, Simple State, Composite State, Submachine State |
| Not used in test models | Shallow History PseudoState, Deep History PseudoState |

# Appendix A: A Manual for the Structured Test Automation Language

This appendix is a manual for the structured test automation language (STAL). STAL can be used for UML behavioral diagrams (so far only state machine diagrams) to transform abstract tests to concrete tests. Thus, this manual focuses on UML state machine diagrams. The example concrete tests are written in Java. This manual still uses the vending machine example used in section 4.2.1.

Table A.1 summarizes which elements of a UML state machine diagram should be mapped. The first row shows a set of elements that need mappings. The elements of the second row such as *Simple State* and *Composite State* do not need mappings because they can be accessed from *Constraints* and *Transitions* in the first row. The definitions of constraints in UML specify which states the constraints are applied to and the definitions of transitions specify the source states and target states. Although they are not directly mapped to test code, they are used to generate test paths in state machine diagrams. *Shallow History PseudoState* and *Deep History PseudoState* in the third row do not specify the current model behaviors, thus, they are not be used in test models.

STAL defines two kinds of mappings: *element mappings* and *object mappings*. Element mappings directly connect an identifiable element in a UML state machine diagram to test code. For instance, a transition *coin* may be mapped to the test code "*vm.coin (c);*". However, objects and parameters used in this element mapping, such as *vm* (an object of

class VendingMachine) and *c* (an int parameter of method *coin (int c)*), also need to be initialized in object mappings, which will be marked as required mappings of this element mapping.

An element mapping is formally defined as:

***Mapping*** *mappingName* ***TYPEOFELEMENT*** *nameOfElement*
***Requires*** *objectMappingName ...*
*[****TYPEOFCONSTRAINT*** *nameOfElement ...] ...*
{*testCode*}

**Mapping** and **Requires** are keywords. The mapping name must be unique. **TYPEOFELEMENT** may vary depending on the actual type of one concrete element. Based on Table A.1, **TYPEOFELEMENT** can be **Transition**, **Constraint**, **Entry Point of a State**, **Exit Point of a State**, and **Do Activity of a State**. If a mapping uses an object defined in another mapping, the names of additional mappings have to be included in the **Requires** field. The notation "..." means that more than one mapping may be required.

If an element is a constraint, the mapping needs to give the type of the constraint and elements in which the constraint is held. This manual considers three types of constraints: state invariants, guards (pre-condition of transitions), and post-condition of transitions. State invariants are used in states. Guards and post-conditions are used for transitions. Thus, **TYPEOFCONSTRAINT** can be **StateInvariants**, **Guards**, and **PostCondition**. StateInvariants, guards, and post-conditions fields are optional and marked by "[ ]" since they are used only for constraints. A constraint may be used as a state invariant in states and guards and post-conditions on transitions at the same time. Test code is required for any mapping and written in curly brackets.

An object mapping is formally defined as:

Table A.2: Attributes of Element and Object Mappings

| **Attributes** | **Element Mapping** | **Object Mapping** |
|---|---|---|
| Element Name | X | |
| Element Type | X | |
| Mapping Name | X | X |
| Test Code | X | X |
| Required Mappings | X | X |
| State Invariants & Guards | X | |
| Object Name | | X |
| Class Name | | X |

***Mapping*** *mappingName* ***Class*** *nameOfClass*

***Object*** *nameOfObject* ***Requires*** *objectMappingName ...*

{*testCode*}

An object mapping includes the class type and name of the object. An object initialization may also need other objects. So an object mapping may require extra object mappings as well. Table A.2 indicates which attributes can be used in element and object mappings.

For the state machine diagram of the vending machine program in Figure 4.1, I need to create mappings for four distinct transitions: *initialize*, *addChocs*, *getChocs*, and *coin*; and define six constraint mappings. The first mapping is an element mapping for the transition *initialize*:

***Mapping*** *vMachineInit* ***Transition*** *initialize*

{ *VendingMachine vm = new VendingMachine();* }

*vMachineInit* is the mapping name. The keyword **Transition** specifies that the mapping *vMachineInit* is created for a transition.

Next is a mapping for the transition *getChocs*. The method *getChoc (StringBuffer)* is used to get chocolates from the vending machine. The *StringBuffer* object "sb" represents a chocolate.

***Mapping*** *getChocolate* ***Transition*** *getChocs*

{

*StringBuffer sb = new StringBuffer ("MM");*

*vm.getChoc (sb);*

}

The mapping *getChocolate* gets only one chocolate from the vending machine. More chocolates can be taken from the vending machine if the method *getChoc (StringBuffer)* is called multiple times. Two objects, *vm* and *sb*, need to be initialized before this transition is taken. Because the transition *initialize* appears at the beginning of every test path, the object *vm* will be initialized before the test code for other mappings is run, thus, other mappings do not need to specify the mapping *vMachineInt* to be required. A *StringBuffer* variable *sb* is initialized directly in the test code of this mapping. Alternatively, the initialization of *sb* could be defined in an object mapping and reused in other mappings. The next example shows another mapping that gets two chocolates. It requires an object mapping *stringBufferInit*.

***Mapping*** *getTwoChocolates* ***Transition*** *getChocs*

***Requires*** *stringBufferInit*

{

*vm.getChoc (sb);*

*vm.getChoc (sb);*

}

The object mapping for *stringBufferInit* is shown below:

***Mapping*** *stringBufferInit* ***Class*** *StringBuffer* ***Object*** *sb*

{ *StringBuffer sb = new StringBuffer ("MM");* }

Note that the initialization of an object should be either embedded in the test code of

an element mapping, defined as an object mapping separately, but not both. Otherwise the object will be defined twice. STALE will report to testers if such errors happen. The examples above show that testers could generate multiple mappings with different test values for the same transition (mappings *getChocolate* and *getTwoChocolates* for the transition *getChocs*). Which mapping should be chosen depends on the constraint evaluation, which is discussed in section 4.2.3. The next mapping, *coinOneDollarAndTen*, provides test code for transition *coin*.

```
Mapping coinOneDollarAndTen () Transition coin
{
  vm.coin (10);
  vm.coin (25);
  vm.coin (25);
  vm.coin (25);
  vm.coin (25);
}
```

This mapping adds $1.10 to the vending machine. The method *coin* is used to add credits in the vending machine. The *int* parameter of the method *coin* represents how much is added. If adding a different amount of money, testers have to create a new mapping. Alternatively, the testers can create a *parameterized mapping* to avoid writing multiple mappings.

```
Mapping coinAnyCredit (int c) Transition coin
    requires cForCoin
{ vm.coin (c); }
```

Testers can provide multiple test values for primitive types and values will be chosen arbitrarily. To provide test values for the parameter *c* in an object mapping, an object mapping *cForCoin* can be written below:

```
Mapping cForCoin Class int Object c
{ 10, 25, 100 }
```

The vending machine only accepts dimes (10), quarters (25), and dollars (100). One of the three *int* values will be selected arbitrarily for the parameter *c*, potentially with a different value each time *cForCoin* is used. Testers can also provide predicates such as {c $>$ 0, c $\leq$ 100}, separating conditions by commas.

***Mapping*** *anyValueBetween0And100* ***Class*** *int* ***Object c***
{ *c > 0, c ≤ 100*}

The constraint solver *Choco* for numeric variables [80] used in STALE will return a value such as 1 that satisfies the two boolean expressions. *Choco* has a limited language, and only accepts numeric variables (*int*, *float*, and *double*) and arithmetic operators. Another constraint solver *Xeger* [81] can return *String* values to satisfy regular expressions. Both of the constraint solvers do not accept disjuncts or function calls. The string value "MM" of the test code above *StringBuffer sb = new StringBuffer ("MM");* can be generated using *Xeger*.

***Mapping*** *sForMM* ***Class*** *String* ***Object s***
{"*[M]{2,2}*"}

A mapping that specifies a constraint to be a state invariant is shown below. In this example, Constraint1 is a state invariant for State3, State6, and State9. The credit of the vending machine has to be equal to or greater than 90 cents in these three states. The test code of this constraint is evaluated when tests go through a state to which the constraint is applied. If the boolean expression is evaluated to false, the mapping prior to this state does not satisfy this constraint and an alternative mapping needs to be used. This constraint can also be explicitly expressed as test oracles in concrete tests.

***Mapping*** *constraintForCredit* ***Constraint*** *Constraint1*

***StateInvariants*** *State3, State6, State9*

{ *vm.getCredit() ≥ 90;* }

# Appendix B: A Complete Example using the Vending Machine System

This appendix gives all software artifacts for the vending machine system used in the experiments for the mapping and test oracle problems in sections 5.2 and 5.3. The artifacts include the implementation of the vending machine class, the UML state machine diagram, the abstract tests, the mappings created for transforming the abstract tests to concrete tests, and the concrete tests with various test oracle strategies (OSes).

## B.1 The Implementation of Vending Machine System

The vending machine system was introduced in section 4.2. The vending machine system has one class *VendingMachine*. Figure 1.3 shows the specification of this class. The actual implementation is shown in Figures B.1 and B.2 below.

Figure 4.1 shows a UML state machine diagram for this vending machine example created with the Eclipse Modeling Framework (*EMF*)-based tool *Papyrus* [78].

## B.2 Abstract Tests

The UML state machine diagram was transformed to a general graph. Thus, the general graph has 11 nodes and 26 edges. The initial node is 1 and the final node is 5. The edges are:

1. $[4, 7]$
2. $[3, 4]$
3. $[2, 6]$
4. $[6, 8]$
5. $[2, 9]$
6. $[9, 10]$
7. $[10, 11]$

8. [6, 10]
9. [8, 11]
10. [3, 2]
11. [11, 2]
12. [7, 8]
13. [10, 10]
14. [6, 6]
15. [2, 8]
16. [9, 11]
17. [9, 5]
18. [2, 5]
19. [3, 5]
20. [1, 3]
21. [4, 6]
22. [4, 4]
23. [3, 7]
24. [9, 9]
25. [11, 11]
26. [8, 3]

The abstract tests for edge coverage are:

1. [1, 3, 2, 9, 9, 11, 2, 5]
2. [1, 3, 4, 4, 6, 6, 10, 10, 11, 2, 9, 10, 11, 2, 5]
3. [1, 3, 4, 7, 8, 3, 2, 6, 8, 11, 11, 2, 8, 3, 2, 5]
4. [1, 3, 2, 9, 5]
5. [1, 3, 2, 5]
6. [1, 3, 5]
7. [1, 3, 7, 8, 3, 2, 5]

The edge-pairs are:

1. [4, 7, 8]
2. [3, 4, 7]

3. $[3, 4, 6]$
4. $[3, 4, 4]$
5. $[2, 6, 8]$
6. $[2, 6, 10]$
7. $[2, 6, 6]$
8. $[6, 8, 11]$
9. $[6, 8, 3]$
10. $[2, 9, 10]$
11. $[2, 9, 11]$
12. $[2, 9, 5]$
13. $[2, 9, 9]$
14. $[9, 10, 11]$
15. $[9, 10, 10]$
16. $[10, 11, 2]$
17. $[10, 11, 11]$
18. $[6, 10, 11]$
19. $[6, 10, 10]$
20. $[8, 11, 2]$
21. $[8, 11, 11]$
22. $[3, 2, 6]$
23. $[3, 2, 9]$
24. $[3, 2, 8]$
25. $[3, 2, 5]$
26. $[11, 2, 6]$
27. $[11, 2, 9]$
28. $[11, 2, 8]$
29. $[11, 2, 5]$
30. $[7, 8, 11]$
31. $[7, 8, 3]$
32. $[10, 10, 11]$
33. $[10, 10, 10]$
34. $[6, 6, 8]$
35. $[6, 6, 10]$

36. [6, 6, 6]
37. [2, 8, 11]
38. [2, 8, 3]
39. [9, 11, 2]
40. [9, 11, 11]
41. [1, 3, 4]
42. [1, 3, 2]
43. [1, 3, 5]
44. [1, 3, 7]
45. [4, 6, 8]
46. [4, 6, 10]
47. [4, 6, 6]
48. [4, 4, 7]
49. [4, 4, 6]
50. [4, 4, 4]
51. [3, 7, 8]
52. [9, 9, 10]
53. [9, 9, 11]
54. [9, 9, 5]
55. [9, 9, 9]
56. [11, 11, 2]
57. [11, 11, 11]
58. [8, 3, 4]
59. [8, 3, 2]
60. [8, 3, 5]
61. [8, 3, 7]

The tests for edge-pair coverage are shown below:

1. [1, 3, 2, 9, 9, 5]
2. [1, 3, 4, 6, 10, 10, 10, 11, 2, 5]
3. [1, 3, 2, 9, 9, 10, 10, 11, 11, 11, 2, 5]
4. [1, 3, 4, 4, 4, 7, 8, 11, 11, 2, 9, 10, 11, 11, 2, 9, 11, 2, 9, 5]
5. [1, 3, 2, 8, 11, 2, 6, 10, 11, 2, 6, 8, 3, 7, 8, 3, 2, 5]

6. $[1, 3, 7, 8, 3, 4, 6, 6, 8, 11, 11, 2, 5]$
7. $[1, 3, 2, 9, 9, 9, 11, 11, 2, 6, 6, 6, 10, 11, 2, 8, 3, 4, 4, 6, 8, 3, 2, 6, 6, 8, 3, 4, 4, 6, 8, 3, 4, 7, 8, 3, 5]$
8. $[1, 3, 2, 5]$
9. $[1, 3, 5]$

## B.3 Mapping Creation

This section displays the mappings created for the model in Figures B.3 and B.4 for the mapping problem experiment.

## B.4 Concrete Tests

Because the concrete tests for the experiments take too much space, they are online at *http://cs.gmu.edu/∼nli1/dissertation/*.

```
import java.util.*;
import java.io.*;
import java.lang.reflect.*;

public class VendingMachine
{
    private int credit; // Current credit in the machine
    private LinkedList stock; // Used to store all chocolates

    // Constructor: vending machine starts empty.
    public VendingMachine()
    {
        credit = 0;
        stock = new LinkedList();
    }

    // A coin is given to the vendingMachine.
    // Must be a dime (10), quarter (25), or dollar (100).
    public void coin (int coin)
    {
        if (coin != 10 && coin != 25 && coin != 100)
            return;
        if (credit >= 90)
            return;
        credit = credit + coin;
        return;
    }

    // User asks for a chocolate. Returns the change
    // and sets the parameter StringBuffer variable Choc.
    public int getChoc (StringBuffer choc)
    {
        int change;
        if (credit < 90 || stock.size() <= 0)
        {
            change = 0;
            choc.replace (0, choc.length(), "");
            return (change);
        }

        change = credit - 90;
        credit = 0;
        choc.replace (0, choc.length(), (String) stock.removeFirst());
        return (change);
    }
```

Figure B.1: Implementation of VendingMachine Class - Part 1

```
    // Adds one new piece of chocolate to the machine.
    public void addChoc (String choc)
    {
        if (stock.size() >= MAX)
            return;
        stock.add (choc);
        return;
    }

    // Get the value of the current credit
    public int getCredit ()
    {
        return credit;
    }

    // Get the number of chocolates
    public int getStock ()
    {
        return stock;
    }
}
```

Figure B.2: Implementation of VendingMachine Class - Part 2

⟨ *mappings* ⟩
⟨ *mapping* ⟩
⟨ *name* ⟩ ***vMachineInit*** ⟨ */name* ⟩
⟨ *transition-name* ⟩ ***i***initialize ⟨ */transition-name* ⟩
⟨ *code* ⟩ ***VendingMachine vm = new VendingMachine();*** ⟨ */code* ⟩
⟨ */mapping* ⟩
⟨ *mapping* ⟩
⟨ *name* ⟩ ***getOneChocolate*** ⟨ */name* ⟩
⟨ *transition-name* ⟩ ***getChocs*** ⟨ */transition-name* ⟩
⟨ *code* ⟩ ***vm.getChoc(sb);*** ⟨ */code* ⟩
⟨ *required-mappings* ⟩ ***stringBufferInit*** ⟨ */required-mappings* ⟩
⟨ */mapping* ⟩
⟨ *mapping* ⟩
⟨ *name* ⟩ ***getTwoChocolates*** ⟨ */name* ⟩
⟨ *transition-name* ⟩ ***getChocs*** ⟨ */transition-name* ⟩
⟨ *code* ⟩
***vm.getChoc(sb);***
***vm.coin(100);***
***vm.getChoc(sb);***
⟨ */code* ⟩
⟨ *required-mappings* ⟩ ***stringBufferInit*** ⟨ */required-mappings* ⟩
⟨ */mapping* ⟩
⟨ *mapping* ⟩
⟨ *name* ⟩ ***addChocolate*** ⟨ */name* ⟩
⟨ *transition-name* ⟩ ***addChocs*** ⟨ */transition-name* ⟩
⟨ *code* ⟩ ***vm.addChoc ("MM");*** ⟨ */code* ⟩
⟨ */mapping* ⟩
⟨ *mapping* ⟩
⟨ *name* ⟩ ***coinOneDime*** ⟨ */name* ⟩
⟨ *transition-name* ⟩ ***coin*** ⟨ */transition-name* ⟩
⟨ *code* ⟩ ***vm.coin(10);*** ⟨ */code* ⟩
⟨ */mapping* ⟩
⟨ *mapping* ⟩
⟨ *name* ⟩ ***coinOneDollarAndTen*** ⟨ */name* ⟩
⟨ *transition-name* ⟩ ***coin*** ⟨ */transition-name* ⟩
⟨ *code* ⟩
***vm.coin(10);***
***vm.coin(25);***
***vm.coin(25);***
***vm.coin(25);***
***vm.coin(25);***
⟨ */code* ⟩
⟨ */mapping* ⟩
⟨ *mapping* ⟩
⟨ *name* ⟩ ***stringBufferInit*** ⟨ */name* ⟩
⟨ *object-name* ⟩ ***sb*** ⟨ */object-name* ⟩
⟨ *class-name* ⟩ ***StringBuffer*** ⟨ */class-name* ⟩
⟨ *code* ⟩ ***StringBuffer sb = new StringBuffer(s);*** ⟨ */code* ⟩
⟨ */mapping* ⟩

Figure B.3: Mappings for Vending Machine Model - Part1

⟨ *mapping* ⟩
⟨ *name* ⟩ ***constraintStockOne*** ⟨ */name* ⟩
⟨ *constraint-name* ⟩ ***ConstraintStocKOne*** ⟨ */constraint-name* ⟩
⟨ *code* ⟩ ***vm.getStock().size() == 1;*** ⟨ */code* ⟩
⟨ *state-invariant* ⟩ ***Credit0Stock1,Credit0To90Stock1,Credit90Stock1*** ⟨ */state-invariant* ⟩
⟨ */mapping* ⟩
⟨ *mapping* ⟩
⟨ *name* ⟩ ***constraintStockOneToTenMapping*** ⟨ */name* ⟩
⟨ *constraint-name* ⟩ ***ConstraintStockOneToTen*** ⟨ */constraint-name* ⟩
⟨ *code* ⟩ ***vm.getStock().size() >= 1*** && ***vm.getStock().size() <= 10;*** ⟨ */code* ⟩
⟨ *state-invariant* ⟩
***Credit0Stock1To10,Credit0To90Stock1To10,Credit90Stock1To10***
⟨ */state-invariant* ⟩
⟨ */mapping* ⟩
⟨ *mapping* ⟩
⟨ *name* ⟩ ***constraintCreditAtLeastNinty*** ⟨ */name* ⟩
⟨ *constraint-name* ⟩ ***ConstraintCreditGTNinety*** ⟨ */constraint-name* ⟩
⟨ *code* ⟩ ***vm.getCredit() >= 90;*** ⟨ */code* ⟩
⟨ *state-invariant* ⟩
***Credit90Stock0,Credit90Stock1,Credit90Stock1To10***
⟨ */state-invariant* ⟩
⟨ */mapping* ⟩
⟨ *mapping* ⟩
⟨ *name* ⟩ ***constraintCreditZeroToNinety*** ⟨ */name* ⟩
⟨ *constraint-name* ⟩ ***ConstraintCreditZeroToNinety*** ⟨ */constraint-name* ⟩
⟨ *code* ⟩ ***vm.getCredit() > 0*** && ***vm.getCredit() < 90;*** ⟨ */code* ⟩
⟨ *state-invariant* ⟩
***Credit0To90Stock0,Credit0To90Stock1,Credit0To90Stock1To10***
⟨ */state-invariant* ⟩
⟨ */mapping* ⟩
⟨ *mapping* ⟩
⟨ *name* ⟩ ***constraintCreditEqualsZero*** ⟨ */name* ⟩
⟨ *constraint-name* ⟩ ***ConstraintCreditZero*** ⟨ */constraint-name* ⟩
⟨ *code* ⟩ ***vm.getCredit() == 0;*** ⟨ */code* ⟩
⟨ *state-invariant* ⟩ ***Credit0Stock0,Credit0Stock1,Credit0Stock1To10*** ⟨ */state-invariant* ⟩
⟨ */mapping* ⟩
⟨ *mapping* ⟩
⟨ *name* ⟩ ***constraintStockEqualsZero*** ⟨ */name* ⟩
⟨ *constraint-name* ⟩ ***ConstraintStockZero*** ⟨ */constraint-name* ⟩
⟨ *code* ⟩ ***vm.getStock().size() == 0;*** ⟨ */code* ⟩
⟨ *state-invariant* ⟩ ***Credit0Stock0,Credit0To90Stock0,Credit90Stock0*** ⟨ */state-invariant* ⟩
⟨ */mapping* ⟩
⟨ */mappings* ⟩

Figure B.4: Mappings for Vending Machine Model - Part2

# Appendix C: An Experimental Guide for the Mapping Problem

This experimental guide was used for the experimental participants to understand the experiment for the mapping problem. The guide compares the automatic test generation using the structured test automation language (STAL) with manual test generation through a vending machine example.

## C.1 Goal of the Study

The goal of this experiment is to study the automatic test generation using STAL in model-based testing by comparison with manual test generation at the system level. STAL is implemented in an structured test automation language framework (STALE). The effectiveness and cost of the automatic test generation will be measured as well. All tests have to be written in **JUnit** and **Eclipse** must be used.

This study requires participants to automate the mapping process from abstract tests to concrete tests from UML models using STALE (state machine diagrams used in this study). Then the participants create tests by hand from the same models. During the test generation phase, the time for both the automatic approach and the manual procedure will be measured. After the manual tests are generated, they will be examined to see if they match the abstract tests, which are the test paths generated from the models.

An example project will be given to each participant for learning how to use STALE to generate tests automatically. Each participant will receive a unique package that contains the experimental guide, program under test, its model, example tests. Thus, each participant will be working on a different program.

Before generating tests, all participants have to read the program, model, and example tests. By running the program example tests and studying the model, they should understand the behaviors of the system and how to test the program.

## C.2 The Mapping Problem

In *model-based testing* (MBT), a model is an abstract partial description of a program that usually reflects functional aspects of the system. Tests expressed in terms of a model are called *abstract tests*. An abstract test can be a path in a state machine diagram starting from an initial state, going through transitions, and ending with a final state and. *Concrete tests* are expressed in terms of the implementation of the model. Thus, a JUnit test is a concrete test. Abstract tests must be transformed to concrete tests since abstract tests cannot be applied directly to the actual program. The mapping problem refers to the problem of converting abstract tests to concrete tests.

Testers currently map abstract tests to concrete tests by hand. If an abstract test consists of several transitions, testers have to write the code for each transition by hand. If one transition is used multiple times in different abstract tests, testers must write redundant code for the same transition. Furthermore, transitions may have the same name and that can be mapped to the same code. Thus, testers have to write redundant code for these transitions as well. This process is time-consuming, labor-intensive, and error-prone. For example, to write a concrete test for a transition authentication for an account, testers may have to set up the test environment, including making a database connection and creating an account, and then write test sequences and an oracle. If this event is used in multiple abstract tests, testers have to do the same process for all concrete tests.

## C.3 The Structured Test Automation Language (STAL)

One solution to solve the mapping problem is to automate the mapping process. My approach directly creates mappings from identifiable elements (e.g., transitions in a state machine diagram) of a behavioral diagram to concrete test code without other supporting diagrams. When an element appears in an abstract test, the corresponding test code will be generated automatically based on the mappings. An abstract test may look like: [transition

1, ..., transition i, ..., finalTransition]. This abstract test will be transformed to [mappingForTransition1, ..., mappingForTransitionI, ..., mappingForFinalTransition]. Then the mapping sequence will be converted to test code [testCode1, ..., testCodeI, ..., finalTestCode].

The vending machine example is the same as what is used in section 4.2. Figure 4.1 is a UML state machine diagram for this vending machine example created with the Eclipse Modeling Framework (*EMF*)-based tool *Papyrus* [78]. Figure 4.1 has one initial state, one final state, nine normal states, and 26 transitions. It also includes six constraints that are used as state invariants for states. Figure 1.3 in section 1.2.2 shows the implementation of the *VendingMachine* class.

The test automation language defines the rules about how to write mappings. Three are two kinds of mappings: are *element mappings* and *object mappings*. An element mapping directly connects an element to test code. For instance, a transition *coin* may be mapped to the test code *vm.coin (c);*. However, objects and parameters used in this element mapping such as *vm* (an object of class VendingMachine) and *c* (an int parameter of method *coin (int)*) may also need object mappings for the object initializations. Object mappings used in an element mapping will be marked as the required mappings of this element mapping.

An element mapping for an element such as a transition has five main fields: element name, element type, mapping name, test code, and required mappings. Since this experiment uses state machine diagrams, element mappings are used only for transitions and constraints. If an element mapping is a constraint mapping, this mapping has to specify the constraint. This study only uses state invariants as constraints. A state invariant has to be satisfied in specified states. Thus users have to provide proper mappings to satisfy the constraints. An object mapping has five fields: mapping name, object name, class name, test code, and required mappings. Object mappings are used for initializing objects used in the test code of element mappings. Table A.2 shows the fields of element and object mappings.

Users may not have to use object mappings because the object initializations can be

done in the very first transition *initialize* if there is one. In this case, the following element mappings do not need to put a mapping for the transition *initialize* in the required-mappings field. Since the mapping for *initialize* is executed ahead of mappings for other transitions, the objects used in those mappings could be initialized before the use in an executable test.

Below is an example of creating new mappings for transitions: Suppose that an abstract test consists of two transitions (*initialize* and *coin*) and one constraint (*Constraint1*). For each transition, users can create as many mappings as possible. STALE will automatically pick one mapping that satisfies the next constraint. The transition *initialize* means initializing a vending machine and *coin* means inserting coins in the vending machine. The constraint Constraint1 says that after the transition *coin*, the credit of the vending machine has to be greater than ninety cents. Below is a mapping created for the transition *initialize*. This mapping does not depend on any other mapping so the field of required mappings is empty.

***Element name**: initialize*
***Element type**: transition*
***Mapping name**: vendingMachineInit*
***Test code**: VendingMachine vm = new VendingMachine();*
***Required mappings**:*

A mapping for the transition *coin* is created below:

***Element name**: coin*
***Element type**: transition*
***Mapping name**: coinADollar*
***Test code**: vm.coin (100);*
***Required mappings**:*

This mapping requires the initialization of the object *vm* but does not need to specify the mapping *vendingMachineInit* in the required mappings field because it will be executed

before the mapping *coinADollar*.

For the constraint *Constraint1*, the state invariants field shows in which states the constraint is held. In this case, in the state 3, credit has to be greater than 90 cents. The test code for the constraint has to be a Boolean expression.

***Element name**: Constraint1*
***Element type**: constraint*
***Mapping name**: constriant1Mapping*
***Test code**: vm.getCredit() > 90;*
***Required mappings**:*
***State invariants**: State 3*

The rest of Appendix A and Appendix B use different names for the states and constraints in the UML state machine diagram for the vending machine class. Those names were used originally for the experiments. Then a new Figure 4.1 was created for readers to better understand the diagram. Table C.1 shows the mapping from old notations to new notations in Figure 4.1.

## C.4 Downloading and Installing the Structured Test Automation Language Framework (STALE)

This section gives users the links to STALE and instructions for the installation.

### C.4.1 Downloading STALE

The link to download STALE for Mac users is http://cs.gmu.edu/∼nli1/experiment/taf.zip. The link to download STALE for Windows users is http://cs.gmu.edu/∼nli1/experiment/taf_Windows.zip.

Table C.1: Mappings from old notations to new notations

| **Old Notations** | **New Notations in Figure 4.1** |
|---|---|
| Credit0Stock1 | State4 |
| Credit0Stock0 | State1 |
| Credit0To90Stock0 | State2 |
| Credit0To90Stock1 | State5 |
| Credit90Stock0 | State3 |
| Credit90Stock1 | State6 |
| Credit0Stock1To10 | State7 |
| Credit0To90Stock1To10 | State8 |
| Credit90Stock1To10 | State9 |
| ConstraintCreditZero | Constraint3 |
| ConstraintStockZero | Constraint6 |
| ConstraintCreditZeroToNine | Constraint2 |
| ConstraintStockOne | Constraint5 |
| ConstraintCreditGTNinety | Constraint1 |
| ConstraintStockOneToTen | Constraint4 |

### C.4.2 Installing STALE

After downloading and unzipping the zip file, you should be able to see two files: taf.jar (taf_Windows.jar for Windows users) and a directory called "project", as shown in Figure C.1.


Figure C.1: A Overview of STALE Folder Structure

Because STALE calls the Java compiler to compile tests, the environment variable JAVA_HOME has to point to a JDK 7 directory and the environment variable **PATH** has to include the $JAVA_HOME/bin. Do not include any JRE in the PATH. For Mac users: Open the Terminal, type "echo $JAVA_HOME" and "echo $PATH" and make sure the environment variables JAVA_HOME and PATH are correctly set. Figures C.2 and C.3

show examples for "$JAVA_HOME" and "$PATH" variables.

```
lis-macbook-pro:~ nli$ echo $JAVA_HOME
/Library/Java/JavaVirtualMachines/jdk1.7.0_10.jdk/Contents/Home
```

Figure C.2: A Correct Java_Home Variable Example

```
lis-macbook-pro:~ nli$ echo $PATH
/Library/Java/JavaVirtualMachines/jdk1.7.0_10.jdk/Contents/Home/bin
```

Figure C.3: A Correct System Path Variable Example

If users do not know how to install JDK 7 on a Mac, read the article (http://docs.oracle.com/javase/7/docs/webnotes/install/mac/mac-jdk.html) about how to install JDK 7, and how to set up the JAVA_HOME and PATH environment variables (http://www.cybercitiz.biz/faq/linux-unix-set-java_home-path-variable/).

For Windows users: Open the command line, type "echo %JAVA_HOME%" and "echo %PATH%" to make sure the environment variables JAVA_HOME and PATH are correctly set. If users do not know how to install JDK 7 on a Windows machine, read the articles (http://docs.oracle.com/javase/7/docs/webnotes/install/windows/jdk-installation-windows.html) about how to install JDK 7 and update the PATH environment variable. An important note for users is that %JAVA_HOME%/bin has to be put at the beginning of the environment variable "PATH" so that STALE can find the Java compiler. Otherwise, a Null Pointer Exception will occur when STALE is run.

## C.5 Running STALE

This section discusses how to run STALE with no existing projects.

### C.5.1 Starting STALE

For Mac users, type "java -jar taf.jar" in the Terminal. For Windows users, type the same command "java -jar taf.jar" in the command line. Then users should be able to see a GUI,

as shown in Figure C.4. Since there are no projects under the root directory "project", STALE does not show any projects.

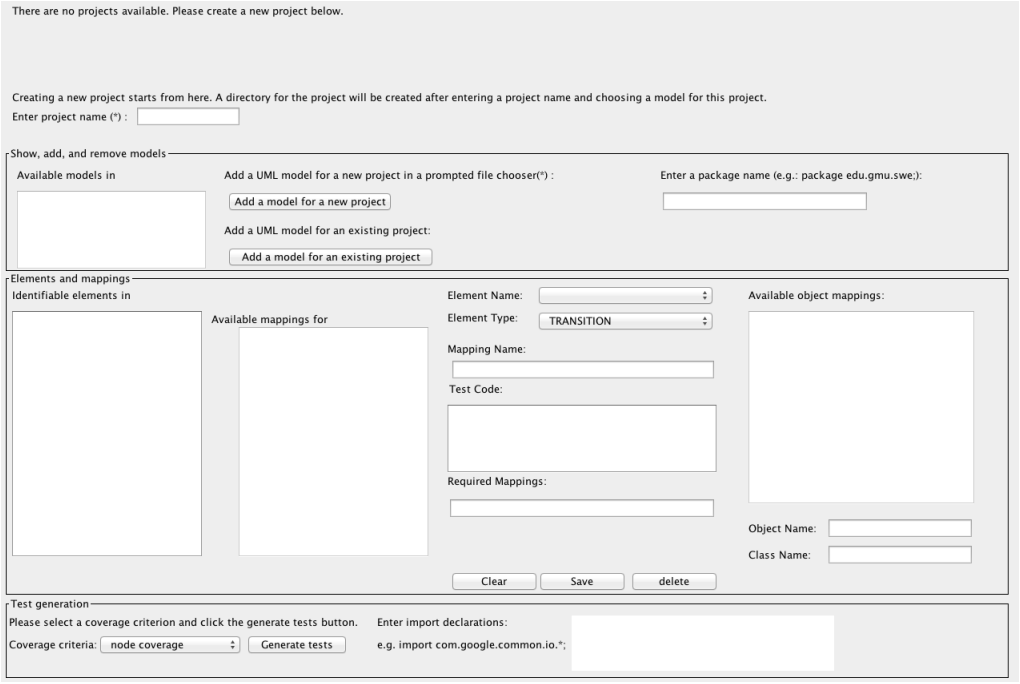


Figure C.4: STALE Started with No Existing Projects

### C.5.2 Creating A New Project

Users can enter "VendingMachine" for the project name and press the button "Add a model for a new project" for adding models to STALE. Figure C.5 shows that a file chooser will pop up and then the users need to select a UML model.

After clicking on the "load" button, the model will be added to the project. Then the elements of the model will be populated automatically, as shown in Figure C.6.

A directory called "VendingMachine" will be created under the root directory "project" and four sub-directories "class", "model", 'xml", and "test" will be created as well. The selected model has been put in the "model" folder. The "class" folder keeps the program class files and dependent Jar files. But users have to put classes under test and dependent Jar files (VendingMachine.class for this case) under the sub-directory "class" by hand. In

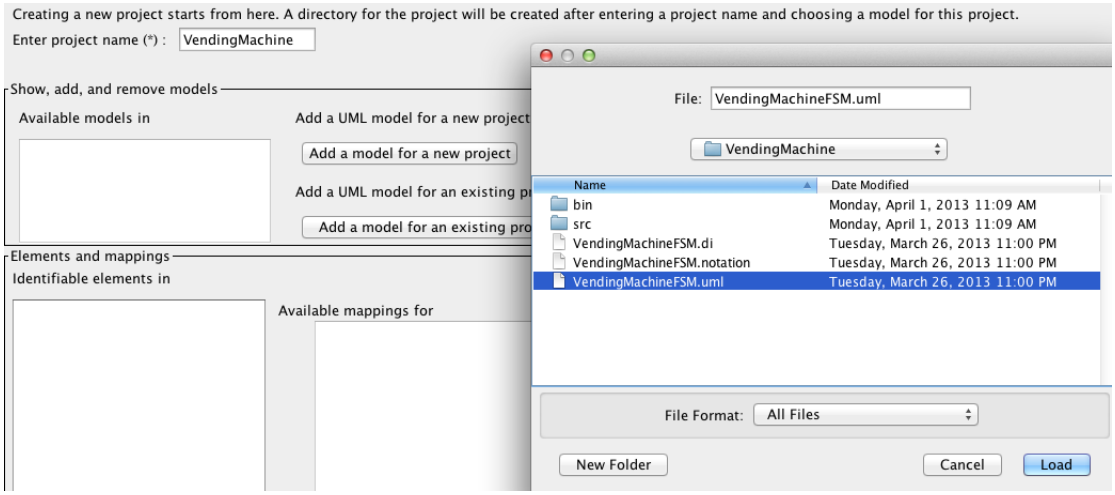


Figure C.5: Adding a Model to STALE

the "xml" subdirectory, mappings are saved in an Xml file (e.g., projectNameFSM.xml) that has the same name as the model (e.g., projectNameFSM.uml). Automatically generated tests are in the "test" folder. There is another folder in the "test" folder called "temp" which has intermediate test files that are used to evaluate constraints. Figure C.7 shows the folder structure for the vending machine project.

### C.5.3 Adding Mappings

Next step, users need to create mappings for elements in the model. Double clicking on an element in the element list such as "initialize". The name and type of the element will be filled automatically in the comboBoxes "Element Name" and "Element Type". Users need to type a mapping name, test code or required mappings if necessary. Then click the "save" button.

Figure C.8 shows a mapping for the transition "initialize".

This initialization mapping will be required in other mappings because the object *vm* is also used in other mappings.

Figure C.9 shows a mapping for the transition *coin*. When generating the test code from this mapping, the object *vm* needs to be initialized. So users need to add the mapping

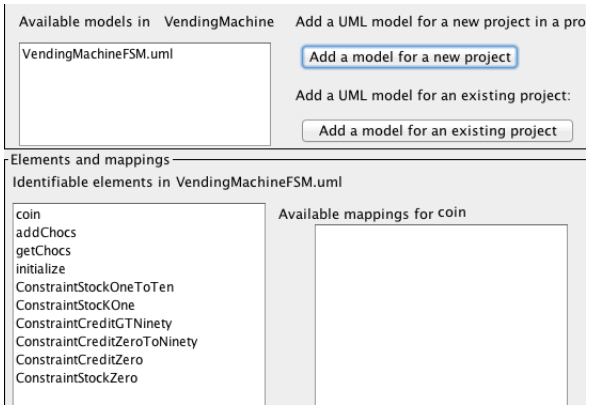


Figure C.6: A Model is Added to STALE

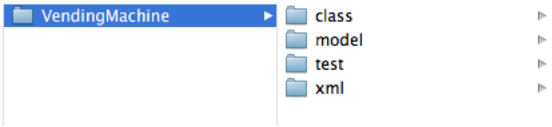


Figure C.7: STALE Folder Structure for the Vending Machine Project

*initialize VendingMachine* defined above in the field of required mappings for the mapping created for the transition *coin*. Thus, the test code for the transition *coin* will be executed following the *vm* initialization. However, if in every test path, the transition *initialize* is executed before the transition *coin*, users do not need to put the initialization mapping in the required mappings field since the test code for the object initialization will be executed before any other mappings.

Figure C.10 shows a mapping for the constraint *ConstraintStockOne*. When double clicking on the constraint *ConstraintStockOne*, its name, type, and constrained elements will be filled automatically as well. In this case, constraints are only used for state invariants. Thus, the field of state invariants only shows the states that have the constraint.

Figure C.11 is a mapping for the transition *getChocs* that uses object mappings. Because the parameter *sb* that is an object of class *StringBuffer* needs to be initialized, the parameter will be mapped to test code for *sb* initialization.

Users can select "OBJECT" in the comboBox of "Element Type" and then fill in the

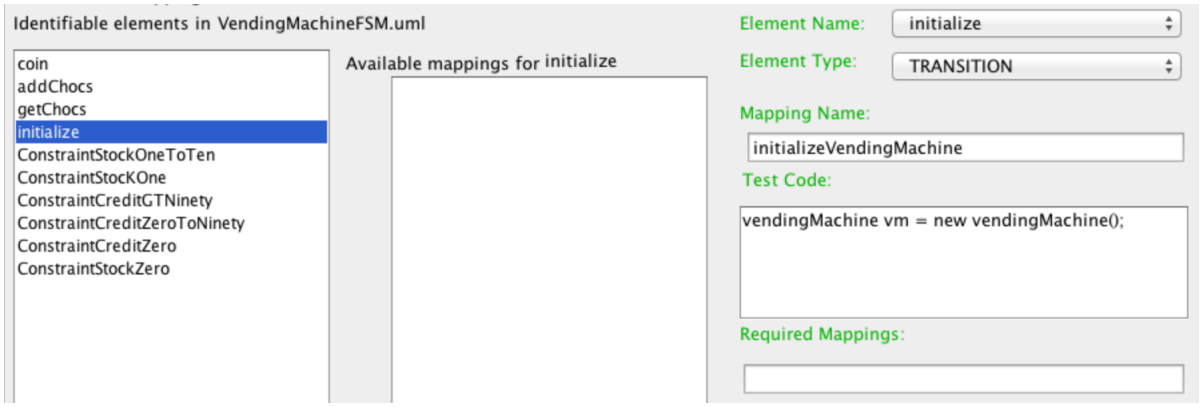


Figure C.8: Adding a Mapping for Transition Initialize

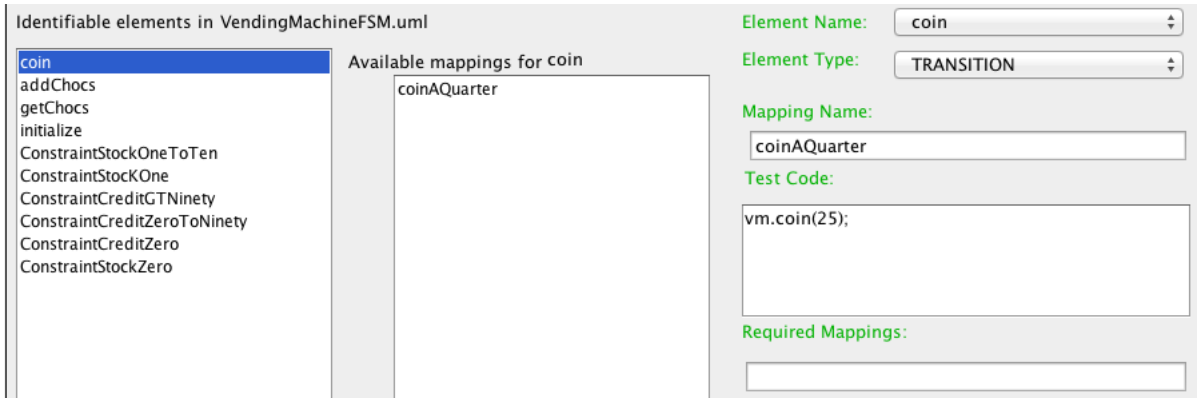


Figure C.9: Adding a Mapping for Transition Coin

fields for adding the object mapping "stringBufferInit" for the parameter "sb", as shown in Figure C.12.

### C.5.4 Generating Concrete Tests

Users may have to import declarations if necessary. The format is exactly as in Java. The default import declarations in the tool are: "import java.io.*; import org.junit.*; and import static org.junit.Assert.*;". Besides the default import declarations, users need to add others if required. Figure C.13 shows the import declaration field and test generation button.

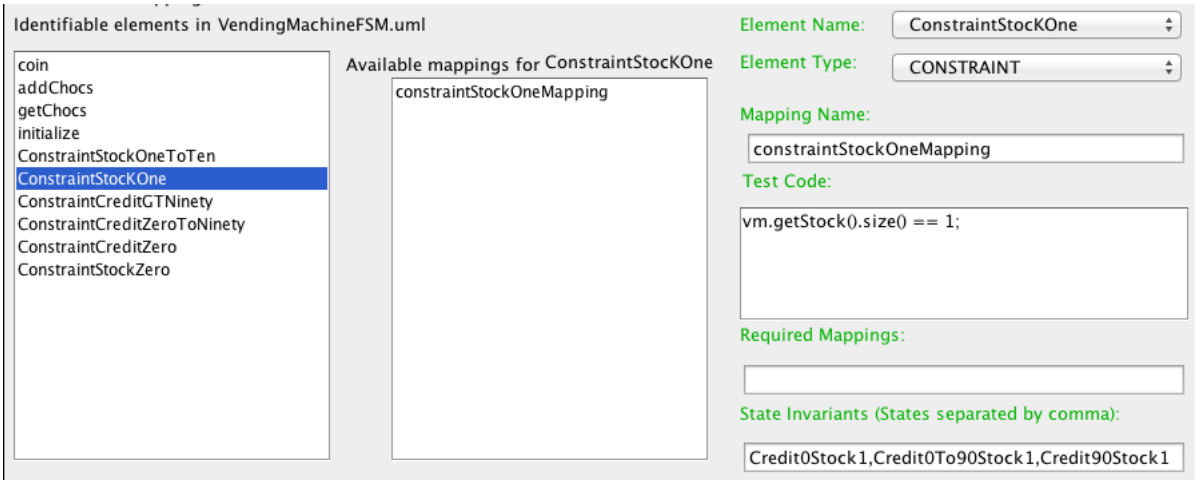


Figure C.10: Adding a Mapping for Constraint ConstrinatStockOne

Once all necessary mappings are generated, users should select the edge coverage criterion and press the button "generate tests". The generated tests will be under the subdirectory "test". The Terminal or command-line will show the time used for test generation, as shown in Figure C.14.

## C.6 Experimental Procedure

Before reading this section, participants should have read the experiment guide, known how to use STALE, and understood program under test and its model. This section describes the steps about how to generate tests by hand and by STALE. The diagram transformation for the manual test generation will be introduced first.

### C.6.1 Diagram Transformation

When generating tests by hand, a state machine diagram should be transformed to a general graph. Table C.2 shows the transformation from states of UML state machine diagram to nodes of a general graph. Both the automatic approach and manual process will use the same test paths, which is shown below.

Test paths for edge coverage are generated using the graph coverage web application:

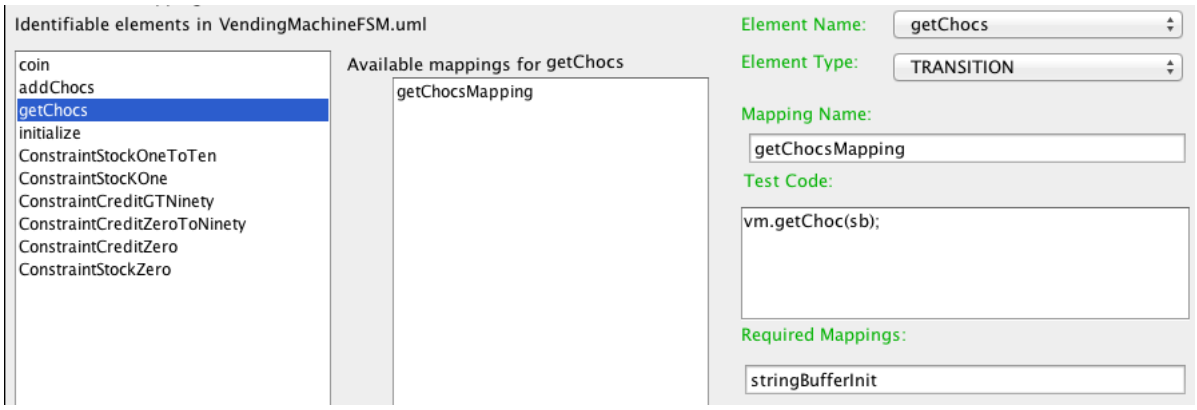


Figure C.11: Adding a Mapping for Transition GetChocs

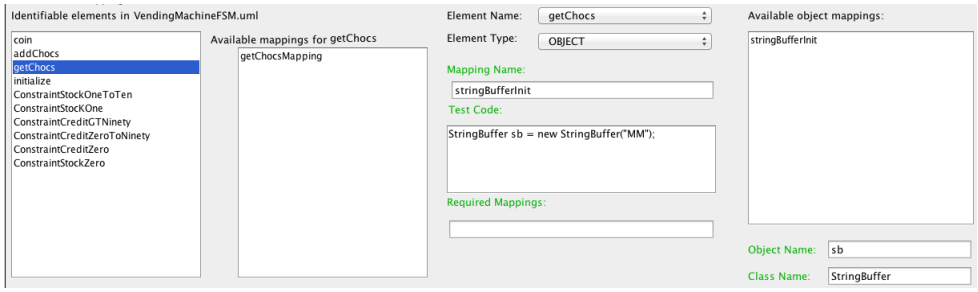


Figure C.12: Adding a Mapping for Object StringBuffer

1. $[1, 3, 2, 9, 9, 11, 2, 5]$
2. $[1, 3, 4, 4, 6, 6, 10, 10, 11, 2, 9, 10, 11, 2, 5]$
3. $[1, 3, 4, 7, 8, 3, 2, 6, 8, 11, 11, 2, 8, 3, 2, 5]$
4. $[1, 3, 2, 9, 5]$
5. $[1, 3, 2, 5]$
6. $[1, 3, 5]$


Figure C.13: Import Declaration and Test Generation

```
Time for generating tests = 5602318000 nano seconds
Time for generating tests = 5 seconds
```

Figure C.14: Time Spent for Test Generation

Table C.2: Mappings from States of the UML Diagrams to Nodes of the General Graph

| **States of UML Diagrams** | **Nodes of General Graphs** |
|---|---|
| Initial | 1 |
| Credit0Stock1 | 2 |
| Credit0Stock0 | 3 |
| Credit0To90Stock0 | 4 |
| Final | 5 |
| Credit0To90Stock1 | 6 |
| Credit90Stock0 | 7 |
| Credit90Stock1 | 8 |
| Credit0Stock1To10 | 9 |
| Credit0To90Stock1To10 | 10 |
| Credit90Stock1To10 | 11 |

7. $[1, 3, 7, 8, 3, 2, 5]$

### C.6.2 Test Generation

Participants will first generate tests automatically, and then generate tests manually.

1. Download and install the test automation tool. The time used in this step will not be measured.

2. Create a new project "VendingMachine" and add the state machine diagram to the project. Then put the program under test and dependent Jar files in test by hand. Time will be measured.

3. Once the model is added, all elements are shown in the element list. Users need to create mappings for transitions and constraints using the tool. Time will be measured.

4. Create concrete tests using the tool to satisfy the edge coverage criterion and the tool will measure the time for this step. The time used for this step will be shown in the

console.

The manual steps were:

1. Transform the FSM to a general graph. The test paths have been provided above in the diagram transformation section. So this step does not need participants to do anything.

2. Generate concrete tests from the test paths by hand. Create a new Java project using Eclipse and put the class under test (VendingMachine.java) in this project. Then create a JUnit test case in the same project and all tests in this test case. Measure the time for this step of the test preparation.

3. For each test path, write a concrete test that matches it. The concrete tests have to be written in the same order as the test paths. Be aware that students do not have to write test oracles (assertions) but the constraints specified in states have to be satisfied. At the last step, compile the test and make sure that all tests pass. Measure the time for this step.

# Bibliography

# Bibliography


[1] J. Offutt and A. Abdurazik, "Generating tests from UML specifications," in *Proceedings of the Second IEEE International Conference on the Unified Modeling Language (UML99)*. Fort Collins, CO: Springer-Verlag Lecture Notes in Computer Science Volume 1723, October 1999, pp. 416–429.

[2] M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "A comprehensive survey of trends in oracles for software testing," University of Sheffield, Department of Computer Science, Tech. Rep. CS-13-01, 2013. [Online]. Available: http://philmcminn.staff.shef.ac.uk/publications/pdfs/2013-techrep.pdf

[3] R. A. DeMillo and J. Offutt, "Constraint-based automatic test data generation," *IEEE Transaction on Software Engineering*, vol. 17, no. 9, pp. 900–910, September 1991.

[4] J. Offutt, "Automatic test data generation," Ph.D. dissertation, Georgia Institute of Technology, Atlanta, GA, USA, 1988, Technical report GIT-ICS 88/28.

[5] L. J. Morell, "A theory of error-based testing," *IEEE Transactions on Software Engineering*, vol. 16, no. 8, pp. 844–857, August 1990.

[6] ——, "A theory of error-based testing," Ph.D. dissertation, University of Maryland, College Park, MD, USA, 1984, Technical report TR-1395.

[7] P. Ammann and J. Offutt, *Introduction to Software Testing*. Cambridge, UK: Cambridge University Press, 2008, iSBN 0-52188-038-1.

[8] L. C. Briand, M. D. Penta, and Y. Labiche, "Assessing and improving state-based class testing: A series of experiments," *IEEE Transaction on Software Engineering*, vol. 30, no. 11, pp. 770–793, November 2004.

[9] K. Shrestha and M. Rutherford, "An empirical evaluation of assertions as oracles," in *Proceedings of the 2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*, ser. ICST '11. Berlin, Germany: IEEE Computer Society, March 2011, pp. 110–119.

[10] M. Staats, M. W. Whalen, and M. P. Heimdahl, "Better testing through oracle selection (NIER track)," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11. Waikiki, Honolulu, HI, USA: ACM, May 2011, pp. 892–895.

[11] Q. Xie and A. Memon, "Designing and comparing automated test oracles for GUI-based software applications," *ACM Transaction on Software Engineering and Methodology*, vol. 16, no. 1, February 2007.

[12] P. Ammann, J. Offutt, W. Xu, and N. Li, "Graph coverage web applications," Online, 2008, http://cs.gmu.edu:8080/offutt/coverage/GraphCoverage, last access October 2013.

[13] O. M. Group, "Object constraint language," Online, 2006, http://www.omg.org/spec/OCL/2.0/, last access December 2013.

[14] N. Li and J. Offutt, "A test automation language for behavioral models," Available at http://cs.gmu.edu, Department of Computer Science, George Mason University, Fairfax, VA, USA, Tech. Rep. GMU-CS-TR-2013-7, 2013.

[15] R. S. Freedman, "Testability of software components," *IEEE Transactions on Software Engineering*, vol. 17, no. 6, pp. 553–564, 1991.

[16] J. Offutt and J. Pan, "Detecting equivalent mutants and the feasible path problem," vol. 7, no. 3, pp. 165–192, September 1997.

[17] V. V. Vazirani, *Approximation Algorithms*. Springer, 2000.

[18] D. Hochbaum, *Approximation Algorithms for NP-Hard Problems*, 1st ed. Course Technology, July 1996.

[19] A. V. Aho and D. Lee, "Efficient algorithms for constructing testing sets, covering paths, and minimizing flows," *AT&T Bell Laboratories Tech. Memo*, vol. 159, 1987.

[20] M. Garey and D. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York, NY, USA: W. H. Freeman & Co., 1979.

[21] W. Prenninger and A. Pretschner, "Abstractions for model-based testing," *Electronic Notes in Theoretical Computer Science*, vol. 116, pp. 59–71, January 2005.

[22] M. Barnett, K. R. M. Leino, and W. Schulte, "The spec# programming system: An overview," in *Proceedings of the 2004 International Conference on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, ser. CASSIS'04. Marseille, France: Springer-Verlag, 2005, pp. 49–69.

[23] M. Utting, G. Perrone, J. Winchester, S. Thompson, R. Yang, and P. Douangsavanh, "The ModelJUnit model-based testing tool," Online, 2007, http://www.cs.waikato.ac.nz/∼marku/mbt/modeljunit/, last access April 2013.

[24] M. Veanes, C. Campbell, W. Grieskamp, W. Schulte, and N. Tillmann, "Microsoft SpecExplorer," Online, 2002, http://research.microsoft.com/en-us/projects/specexplorer/, last access April 2013.

[25] J. Jacky and M. Veanes, "NModel," Online, 2006, http://nmodel.codeplex.com/, last access April 2013.

[26] C. Inc., "CONFORMIQ Automated Test Design," Online, 2011, http://www.conformiq.com/, last access April 2013.

[27] M. Utting and B. Legeard, *Practical Model-Based Testing: A Tools Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007.

[28] P. Fröhlich and J. Link, "Automated test case generation from dynamic models," in *Proceedings of the 14th European Conference on Object-Oriented Programming*, ser. ECOOP '00. London, UK: Springer-Verlag, 2000, pp. 472–492.

[29] J. Ryser and M. Glinz, "A scenario-based approach to validating and testing software systems using statecharts," in *Proceedings of the 12th International Conference on Software and Systems Engineering and their Applications*, ser. ICSSEA '99, Paris, France, 1999.

[30] L. Briand and Y. Labiche, "A UML-based approach to system testing," in *Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools*, ser. UML '99. London, UK: Springer-Verlag, 2001, pp. 194–208.

[31] C. Nebut, F. Fleurey, Y. L. Traon, and J.-M. Jézéquel, "Automatic test generation: a use case driven approach," *IEEE Transaction on Software Engineering*, vol. 32, no. 3, pp. 140–155, March 2006.

[32] S. Liu and S. Nakajima, "A framework for automatic functional testing based on formal specifications," in *Proceedings of the 2011 ACM International Workshop on Automation of Software Test*, ser. AST '11. Waikiki, Honolulu, USA: ACM, May 2011, pp. 107–108.

[33] A. Ulrich, E.-H. Alikacem, H. H. Hallal, and S. Boroday, "From scenarios to test implementations via Promela," in *Proceedings of the 22nd IFIP WG 6.1 International Conference on Testing Software and Systems*, ser. ICTSS '10. Natal, Brazil: Springer-Verlag, 2010, pp. 236–249.

[34] D. Lugato, C. Bigot, and Y. Valot, "Validation and automatic test generation on UML models: The AGATHA approach," *Electronic Notes in Theoretical Computer Science*, vol. 66, no. 2, pp. 33–49, 2002.

[35] Y. Kim, H. Hong, D. Bae, and S. Cha, "Test cases generation from UML state diagrams," *IEE Proceedings. Software*, vol. 146, no. 4, pp. 187–192, August 1999.

[36] S. Pimont and J. C. Rault, "A software reliability assessment based on a structural behavioral analysis of programs," in *Proceedings of the Second International Conference on Software Engineering*, San Francisco, CA, October 1976, pp. 486–491.

[37] S. I. G. i. S. T. British Computer Society, *Standard for Software Component Testing, Working Draft 3.3.* British Computer Society, 1997, http://www.rmcs.cranfield.ac.uk/∼cised/sreid/BCS_SIG/.

[38] J. Offutt, S. Liu, A. Abdurazik, and P. Ammann, "Generating test data from state-based specifications," *Software Testing, Verification, and Reliability*, vol. 13, no. 1, pp. 25–53, March 2003.

[39] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *IEEE Computer*, vol. 11, no. 4, pp. 34–41, April 1978.

[40] K. N. King and J. Offutt, "A Fortran language system for mutation-based software testing," *Software–Practice and Experience*, vol. 21, no. 7, pp. 685–718, July 1991.

[41] M. E. Delamaro and J. C. Maldonado, "Proteum-A tool for the assessment of test adequacy for C programs," in *Proceedings of the Conference on Performability in Computing Systems (PCS 96)*, New Brunswick, NJ, July 1996, pp. 79–95.

[42] S. Kim, J. A. Clark, and J. A. McDermid, "Investigating the effectiveness of object-oriented strategies with the mutation method," in *Proceedings of Mutation 2000: Mutation Testing in the Twentieth and the Twenty First Centuries*, San Jose, CA, October 2000, pp. 4–100.

[43] Y.-S. Ma, J. Offutt, and Y.-R. Kwon, "MuJava : An automated class mutation system," *Wiley's Journal of Software Testing, Verification, and Reliability*, vol. 15, no. 2, pp. 97–133, June 2005.

[44] J. H. Andrews, L. C. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?" in *Proceedings of the 27th International Conference on Software Engineering, (ICSE 2005)*. St. Louis, Missouri: IEEE Computer Society, May 2005, pp. 402–411.

[45] J. Offutt, Z. Jin, and J. Pan, "The dynamic domain reduction approach to test data generation," *Software-Practice and Experience*, vol. 29, no. 2, pp. 167–193, January 1999.

[46] Y.-S. Ma, J. Offutt, Y.-R. Kwon, and N. Li, "muJava home page," Online, 2013, http://cs.gmu.edu/∼offutt/mujava/, last access August 2013.

[47] N. Li, F. Li, and J. Offutt, "Better algorithms to minimize the cost of test paths," in *Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, ser. ICST '12. Montreal, Quebec, Canada: IEEE Computer Society, April 2012, pp. 280–289.

[48] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: A survey," *Software Testing, Verification and Reliability*, vol. 22, no. 2, pp. 67–120, 2012.

[49] R. Garfinkel and G. L. Nemhauser, *Integer Programming*. New York, USA: John Wiley & Sons, 1972.

[50] M. J. Harrold, R. Gupta, and M. L. Soffa, "A methodology for controlling the size of a test suite," *ACM Transactions on Software Engineering Methodology*, vol. 2, pp. 270–285, July 1993.

[51] T. Y. Chen and M. F. Lau, "Dividing strategies for the optimization of a test suite," *Information Processing Letters*, vol. 60, no. 3, pp. 135–141, 1996.

[52] J. R. Horgan and S. London, "A data flow coverage testing tool for C," in *Proceedings of the Symposium of Quality Software Development Tools*, May 1992, pp. 2–10.

[53] A. J. Offutt, J. Pan, and J. M. Voas, "Procedures for reducing the size of coverage-based test sets," in *Proceedings of the Twelfth International Conference on Testing Computer Software*, Washington, DC, June 1995, pp. 111–123.

[54] J. Gallant, D. Maier, and J. Storer, "On finding minimal length superstrings," *Journal of Computer and System Sciences*, vol. 20, pp. 50–58, 1980.

[55] J. Tarhio and E. Ukkonen, "A greedy approximation algorithm for constructing shortest common superstrings," *Theoretical Computer Science-International Symposium on Mathematical Foundations of Computer Science*, vol. 57, no. 1, pp. 486–491, April 1988.

[56] A. Blum, T. Jiang, M. Li, J. Tromp, and M. Yannakakis, "Linear approximation of shortest superstrings," *Journal of the ACM*, vol. 41, no. 4, pp. 630–647, July 1994.

[57] M. Weinard and G. Schnitger, "On the greedy superstring conjecture." *SIAM Journal on Discrete Mathematics*, vol. 20, no. 2, pp. 502–522, 2006.

[58] H. J. Romero, C. A. Brizuela, and A. Tchernykh, "An experimental comparison of approximation algorithms for the shortest common superstring problem," in *Proceedings of the Fifth Mexican International Conference in Computer Science*. Colima, Mexico: IEEE Computer Society, 2004, pp. 27–34.

[59] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, *Introduction to Algorithms*. The MIT Press, Cambridge, Massachusetts, 2009.

[60] P. Baker, Z. R. Dai, J. Grabowski, O. Haugen, E. Samuelsson, I. Schieferdecker, and C. E. Williams, "The UML 2.0 testing profile," in *Proceedings of the '8th Conference on Quality Engineering in Software Technology 2004*, ser. CONQUEST 2004. Nuremberg, Germany: ASQF e.V., Erlangen, September 2004, pp. 181–189.

[61] O. M. Group, "MOF model to text transformation language," Online, 2008, http://www.omg.org/spec/MOFM2T/1.0/, last access Sept 2012.

[62] ——, "OMG model driven architecture," Online, 2003, http://www.omg.org/mda/, last access Sept 2012.

[63] E. Foundation, "Acceleo - transforming models into code," Online, 2009, http://www.eclipse.org/acceleo/, last access Sept 2012.

[64] ——, "Eclipse modeling framework," Online, 2008, http://www.eclipse.org/modeling/emf/, last access Sept 2012.

[65] E. Weyuker, "On testing non-testable programs," *The Computer Journal*, vol. 25, no. 4, pp. 465–470, 1982.

[66] D. J. Richardson, S. L. Aha, and T. O. O'Malley, "Specification-based test oracle for reactive systems," in *Proceedings of the 14th International Conference on Software Engineering*, ser. ICSE '92, 1992.

[67] L. Baresi and M. Young, "Test oracles," Department of Computer and Information Science, University of Oregon, Technical Report CIS-TR-01-02, 2001.

[68] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll, "An overview of JML tools and applications," *International Journal on Software Tools for Technology Transfer*, vol. 7, pp. 212–232, June 2005.

[69] S. Sprenkle, L. Pollock, H. Esquivel, B. Hazelwood, and S. Ecott, "Automated oracle comparators for testing web applications," in *18th IEEE International Symposium on Software Reliability Engineering*, ser. ISSRE '07, Trollhattan, Sweden, November 2007, pp. 117–126.

[70] T. Yu, W. Srisa-an, and G. Rothermel, "An empirical comparison of the fault-detection capabilities of internal oracles," in *The 24th IEEE International Symposium on Software Reliability Engineering*, ser. ISSRE '13, Pasadena, CA, USA, November 2013.

[71] N. Halbwachs, "Synchronous programming of reactive systems - a tutorial and commented bibliography," in *Tenth International Conference on Computer-Aided Verification, CAV98, Vancouver (B.C.), LNCS 1427*. Springer Verlag, 1998, pp. 1–16.

[72] M. Staats, G. Gay, and M. P. E. Heimdahl, "Automated oracle creation support, or: how I learned to stop worrying about fault propagation and love mutation testing," in *Proceedings of the 2012 International Conference on Software Engineering*, ser. ICSE 2012. Zurich, Switzerland: IEEE Press, 2012, pp. 870–880.

[73] P. R. Mateo and M. P. Usaola, "$Bacterio^{ORACLE}$: An oracle suggester tool," in *Proceedings of the 25th International Conference on Software Engineering and Knowledge Engineering*, ser. SEKE 2013, Boston, USA, June 2013.

[74] G. Fraser and A. Zeller, "Mutation-driven generation of unit tests and oracles," *IEEE Transactions on Software Engineering*, vol. 38, no. 2, pp. 278–292, 2012.

[75] D. E. Knuth, J. H. M. Jr., and V. R. Pratt, "Fast pattern matching in strings," *SIAM Journal on Computing*, vol. 6, no. 2, June 1977.

[76] N. Li, "A smart structured test automation language (SSTAL)," in *The Ph.D. Symposium of 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST)*, ser. ICST '12, Montreal, Quebec, Canada, April 2012, pp. 471–474.

[77] N. Li and J. Offutt, "An empirical analysis of test oracle strategies for model-based testing," in *Proceedings of the 2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*, ser. ICST '14, Cleveland, Ohio, USA, 2014.

[78] E. Foundation, "Papyrus," Online, 2008, www.eclipse.org/papyrus/, last access Sept 2012.

[79] N. Li, "The structured test automation language framework," Online, 2013, http://cs.gmu.edu/∼nli1/stale/, last access May 2013.

[80] T. C. D. Team, "Choco constraint solver," Online, 2004, http://www.emn.fr/z-info/choco-solver/, last access May 2013.

[81] X. Team, "Xeger string generator," Online, 2009, https://code.google.com/p/xeger/, last access May 2013.

[82] H. Deitel and P. Deitel, *Java: How to program*, 6th ed. Pearson Education, Inc., 2005.

[83] Anonymous, "Class of tree," Online, 2008, http://homepage.cs.uiowa.edu/∼sriram/21/fall08/code/tree.java, last access May 2013.

[84] Lewis, Chase, and Coleman, "Class of blackjack," Online, 2004, http://faculty.washington.edu/moishe/javademos/blackjack/, last access May 2013.

[85] M. Rusma, "Class of triangle," Online, 2004, http://www.cs.du.edu/∼snarayan/sada/teaching/COMP3705/FilesFromCD/Exercises/Lab4_WhiteBox/Triangle.java, last access May 2013.

[86] B. Liskov and J. Guttag, *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*, 1st ed. Addison-Wesley Professional, 2000.

[87] A. Danial, "CLOC," Online, 2006, http://cloc.sourceforge.net, last access Sept 2012.

[88] R. Boddy and G. Smith, *Effective Experimentation: For Scientists and Technologists.* Wiley, 2010.

[89] J. Miles and M. Shevlin, *Applying Regression and Correlation: A Guide for Students and Researchers, Sage Publications*, 1st ed. SAGE Publications Ltd, 2000.

[90] D. Lilja, *Measuring Computer Performance: A Practitioner's Guide.* New York, NY, USA: Cambridge University Press, 2005.

[91] J. Cohen, *Statistical Power Analysis for the Behavioral Sciences*, 2nd ed. Hillsdale, New Jersey, USA: Lawrence Erlbaum Associates, Inc., 1988.

[92] Y.-S. Ma and J. Offutt, "Description of method-level mutation operators for Java," Online, 2005, http://cs.gmu.edu/∼offutt/mujava/mutopsMethod.pdf, last access August 2013.

[93] R. Lowry, *Concepts and Applications of Inferential Statistics.* Cambridge, UK: Cambridge University Press, 2008.

[94] N. Li, U. Praphamontripong, and J. Offutt, "An experimental comparison of four unit test criteria: Mutation, edge-pair, all-uses and prime path coverage," in *Fifth Workshop on Mutation Analysis (Mutation 2009)*, Denver CO, April 2009.

[95] P. Ammann, M. E. Delamaro, and J. Offutt, "Establishing theoretical minimal sets of mutants," in *Proceedings of the 2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*, ser. ICST '14, Cleveland, Ohio, USA, 2014.

[96] M. E. Delamaro, L. Deng, V. Durelli, N. Li, and J. Offutt, "Experimental evaluation of sdl and one-op mutation for C," in *Proceedings of the 2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*, ser. ICST '14, Cleveland, Ohio, USA, 2014.

[97] Y.-S. Ma and J. Offutt, "Description of method-level mutation operators for Java," Online, 2005, http://cs.gmu.edu/ offutt/mujava/mutopsMethod.pdf, last access March 2014.

[98] L. Deng, N. Li, and J. Offutt, "Empirical evaluation of the statement deletion mutation operator," in *Proceedings of the 2013 IEEE Seventh International Conference on Software Testing, Verification and Validation*, ser. ICST '13, Luxembourg, Luxembourg, 2013.

# Biography

Nan Li is a Ph.D. candidate in the Department of Computer Science of Volgenau School of Engineering at George Mason University. He received his M.S. in Computer Science from Fairleigh Dickinson University in New Jersey in 2008. Before that, he received a B.E. in Software Engineering from Beihang University in 2006. His current research interests include model-based testing, test generation, test automation, test oracles, and mutation testing. His advisor is Dr. Jeff Offutt.