2013

Directed Test Suite Augmentation

Zhihong Xu
University of Nebraska-Lincoln, xuzhihong.cq@gmail.com

DIRECTED TEST SUITE AUGMENTATION

by

Zhihong Xu

A DISSERTATION

Presented to the Faculty of

The Graduate College at the University of Nebraska

In Partial Fulfilment of Requirements

For the Degree of Doctor of Philosophy

Major: Computer Science

Under the Supervision of Professor Gregg Rothermel

Lincoln, Nebraska

May, 2013

DIRECTED TEST SUITE AUGMENTATION

Zhihong Xu, Ph. D.

University of Nebraska, 2013

Adviser: Gregg Rothermel

Test suite augmentation techniques are used in regression testing to identify code elements affected by changes and to generate test cases to cover those elements. Whereas methods and techniques to find affected elements have been extensively researched in regression testing, the problem of generating new test cases to cover these elements cost-effectively has rarely been studied. We believe that reusing existing test cases will help us achieve this task. This research develops test suite augmentation techniques that reuse existing test cases to automatically generate new test cases to cost-effectively cover affected elements. We begin by using two dynamic test case generation techniques for augmentation, involving concolic testing and genetic algorithms. Then we investigate other factors, which we believe have an impact on the test suite augmentation, with the two techniques both considered. After this, we present a hybrid algorithm for test suite augmentation, that combines multiple approaches while accounting for the effects of other factors. Finally, we apply the test suite augmentation concept to software product line testing to help generate test cases for software product lines.

ACKNOWLEDGMENTS

I would like to thank all those whose helped make this dissertation's existence possible. Thank you for the encouragement, support, and all prompt feedback during these six years.

I would like to thank my advisor Dr. Gregg Rothermel for supporting me, sharing his knowledge, and helping me during the six years. I would like to thank Dr. Myra Cohen, Dr. Witawas Srisa-an and Dr. David Rosenbaum for serving on my committee, reading my dissertation and evaluating my work. Especially, I would like to mention Dr. Myra Cohen, who has been involved in my work and also given me a lot of help and suggestions.

I would like to thank all my friends in the Esquared lab, Tingting, Pingyu, Katie, Wayne, Xiao and Elena. They have been so friendly and given a lot of help. When I need them, they are always there.

I also would like to thank Dr. Moonzoo Kim and his student, Yunho Kim. I have been working with them for three years and they have also given me a lot of help and suggestions for my work.

Lastly, I would like to thank my family for the support and encouragement at all times.

Contents

Contents **iv**

List of Figures **x**

List of Tables **xi**

1 Introduction **1**

2 Background and Related Work **4**

2.1 Test Suite Augmentation 4

2.2 Test Case Generation 6

2.3 Software Product Lines 7

3 Basic Test Suite Augmentation Technique 1 - Using Concolic Testing 11

3.1 Concolic Test Case Generation 11

3.2 Augmentation Algorithm 13

3.3 Example 16

3.4 Extension to Interprocedural 18

3.5 Implementation 18

3.6 Empirical Study 19

3.6.1 Objects of Analysis 19

3.6.2 Variables and Measures 20
3.6.3 Experiment Setup 22
3.7 Threats to Validity 24
3.8 Results and Analysis 25
3.8.1 RQ1: Number of Constraint Solver Calls 25
3.8.2 RQ2: Coverage Criteria 27
3.9 Discussion 28
3.10 Conclusions 29

4 Basic Test Suite Augmentation Technique 2 - Using a Genetic Algorithm 30

4.1 Genetic Test Case Generation 30
4.2 Factors Affecting Augmentation When Using Genetic Algorithms . . . 31
4.3 Case Study 33
4.3.1 Objects of Analysis 34
4.3.2 Genetic Algorithm Implementation 34
4.3.3 Factors, Variables, and Measures 37
4.3.4 Experiment Setup 43
4.4 Study Limitations 43
4.5 Results and Analysis 44
4.5.1 RQ1: Costs of Augmentation 45
4.5.2 RQ2: Effectiveness of Augmentation 46
4.6 Discussion 48
4.7 Conclusions 50

5 A Framework for Test Suite Augmentation 51

5.1 Framework 51

5.2 Augmentation Techniques 52
5.2.1 Augmentation Basics 52
5.2.1.1 Coverage Criterion 52
5.2.1.2 Identifying Affected Elements 53
5.2.1.3 Ordering Affected Elements 53
5.2.1.4 Test Case Reuse Approach 55
5.2.2 Main Augmentation Algorithm 55
5.2.3 Genetic Test Suite Augmentation 57
5.2.4 Concolic Test Suite Augmentation 58
5.3 Empirical Study 1 61
5.3.1 Objects of Analysis 61
5.3.2 Variables and Measures 63
5.3.3 Experiment Setup 64
5.3.4 Experiment Operation 67
5.3.5 Threats to Validity 68
5.3.6 Results and Analysis 69
5.3.6.1 RQ1: Order of Affected Elements 71
5.3.6.2 RQ2: Use of Existing and New Test Cases 73
5.3.6.3 RQ3: Test Case Generation Algorithm 75
5.3.7 Discussion and Implications 77
5.3.7.1 Affected Element Order 77
5.3.7.2 Test Case Reuse Approach 78
5.3.7.3 Test Case Generation Techniques 79
5.3.7.4 Iteration Limits 80
5.3.7.5 Initial Test Suite Characteristics 81
5.3.7.6 The Benefits of Augmentation 83

5.4 Empirical Study 2 84
5.4.1 Experiment Setup 86
5.4.2 Data and Analysis 89
5.4.3 Discussion and Implications 91
5.5 Additional Analysis and Implications 93
5.5.1 Overall Comparison 94
5.5.2 Analysis of Specific Branches 98
5.6 Conclusions and Future Work 105

6 Advanced Test Suite Augmentation Technique - Hybrid Algorithm 106
6.1 Related Work: Combination of Techniques 106
6.2 Direct Hybrid Test Suite Augmentation 107
6.3 Empirical Study 109
6.3.1 Objects of Analysis 109
6.3.2 Variables and Measures 110
6.3.3 Experiment Setup and Operation 111
6.3.4 Threats to Validity 111
6.4 Results 112
6.4.1 RQ1: Hybrid versus Concolic 113
6.4.2 RQ2: Hybrid versus Genetic 115
6.5 Discussion and Implications 116
6.5.1 Masked-out Benefit of Concolic Testing 117
6.5.2 Weakened Diversity of Test Case Population 118
6.5.3 Potential Remedies 119
6.6 Conclusions 120

7 Test Suite Augmentation for SPLs 121

7.1 Software Product Line Testing 121
7.2 CONTESA 122
7.2.1 Identifying Targets 124
7.2.2 Ordering and Selecting a Next Product 125
7.2.2.1 A Static Order 126
7.2.2.2 A Dynamic Order 128
7.2.3 Generating Test Cases 129
7.3 Empirical Study 1 130
7.3.1 Objects of Analysis 130
7.3.2 Variables and Measures 131
7.3.3 Experiment Setup and Operation 131
7.3.4 Threats to Validity 132
7.3.5 Results 133
7.3.5.1 RQ1: Independent Test Case Generation Versus Continuous Test Suite Augmentation 134
7.3.5.2 RQ2: Order Effects in Continuous Test Suite Augmentation 135
7.4 Empirical Study 2 136
7.4.1 Objects of Analysis 137
7.4.2 Variables and Measures 138
7.4.3 Experiment Setup and Operation 138
7.4.4 Threats to Validity 138
7.4.5 Results 139
7.4.5.1 RQ3: Coverage Achieved by Continuous Versus Specification-Based Test Case Generation 139

7.4.5.2 RQ4: Coverage Achieved During Execution Versus Coverage Calculated by CONTESA 140
7.4.5.3 RQ5: Faults Detected by CONTESA Versus Faults Detected by Specification-Based Test Case Generation 140
7.5 Discussion 141
7.5.1 Overall Expense 141
7.5.2 Continuous Effectiveness Change 141
7.6 Conclusions and Future Work 143

8 Conclusion and Future Work **144**

Bibliography **147**

List of Figures

3.1 CFGs for two program versions 17
3.2 Solver calls: DTSA vs Concolic 26

4.1 Partial control flow graphs for two versions of a program 37
4.2 Costs of applying the five treatments, per treatment and version 44
4.3 Coverage obtained in applying the five treatments, per treatment and version 45

5.1 Test Suite Augmentation Framework 52
5.2 Interprocedural control flow graph 55
5.3 Comparison of branch coverage behaviors for concolic and genetic algorithms on two representative cases. 95
5.4 Symbolic execution tree of the example code 103

6.1 Overview of hybrid test suite augmentation approach 107

7.1 Overview of CONTESA 123
7.2 Efficiency and effectiveness for GPL and Bali 133
7.3 Effectiveness achieved by test suites from SB, calculated by C_sE and when run C_sA 139

List of Tables

3.1 Differences in Numbers of Solver Calls 27
3.2 Coverage Results 27

4.1 Disposition of Test Cases Under the Five Treatments for the Example of Figure 1 39
4.2 Results of ANOVA Analysis 46
4.3 Results of Bonferroni Means Test on Cost 46
4.4 Results of Bonferroni Means Test on Coverage 47

5.1 Objects of Analysis 62
5.2 Branch Coverage and Sizes of Initial Test Suites 63
5.3 Coverage Using DFO Order and Existing Test Cases 70
5.4 Coverage Using DFO Order and Existing plus New Test Cases 70
5.5 Coverage Using Random Order and Existing Test Cases 70
5.6 Coverage Using Random Order and Existing plus New Test Cases 70
5.7 Impact of Order in which Affected Elements are Considered on Coverage and Cost. 71
5.8 Impact of Test Case Reuse Approaches on Coverage and Cost. 74
5.9 Comparison of Coverage: Genetic vs Concolic 75
5.10 Impact of Test Reuse in Quartiles 83

5.11 Results of Concolic Testing From Scratch 84
5.12 Initial Coverage Information for `grep` 85
5.13 Coverage and Cost Data for `grep`, per Version and Technique 89
5.14 Branch Coverage Differences – Smaller Programs 93
5.15 Branch Coverage Differences – `grep` 96
5.16 Numbers of Times in which Branches in `grep` were Covered by One, Two, or Three Test Suites, for DFO with Existing and New Test Cases 98
5.17 Summary of Coverage Limitations 101

6.1 Coverage and Cost Data for Printtok1 113
6.2 Coverage and Cost Data for Printtok2 113
6.3 Coverage and Cost Data for Replace 113
6.4 Branches Covered by Both Algorithms over Branched Covered by the Concolic Algorithm 117
6.5 Cost Differences Between Hybrid Algorithms 119

7.1 Cumulative Effectiveness on Bali Products at Each Stage of Augmentation 142

Chapter 1

Introduction

Software engineers use regression testing to validate software as it evolves. To do this cost-effectively, they often begin by running existing test cases. Existing test cases, however, may not be adequate to validate the code or system behaviors that are present in a new version of a system. *Test suite augmentation techniques* (e.g., [2, 78]) address this problem, by identifying where new test cases are needed and then creating them.

Despite the need for test suite augmentation, most research on regression testing has focused on reducing testing effort and increasing efficiency when running existing test cases. There has been research on approaches for *identifying affected elements* (code components potentially affected by changes) (e.g., [2, 74, 78]), but these approaches do not then generate test cases, leaving that task to engineers. There has been research on automatically generating test cases given pre-supplied coverage goals (e.g., [30, 81]), but this research has not attempted to integrate the test case generation task with reuse of existing test cases.

In principle, any test case generation technique could be used to generate test cases for a modified program. We believe, however, that test case generation techniques that

leverage existing test cases hold the greatest promise where test suite augmentation is concerned. This is because existing test cases provide a rich source of data on potential inputs and code reachability, and existing test cases are naturally available as a starting point in the regression testing context. Further, recent research on test case generation has resulted in techniques that rely on dynamic test execution, and such techniques can naturally leverage existing test cases.

Given the foregoing discussion, our research has an overall goal of developing test suite augmentation techniques that support this task cost-effectively for different kinds of programs. We present a set of techniques that not only integrate test case generation techniques with reuse of existing test cases, but also consider important factors that affect the cost-effectiveness of the augmentation process.

It is important to investigate our approach on different types of programs since program characteristics may impact how well various techniques work. Therefore a major element of our work involves empirical investigation of augmentation techniques on real software systems. Our results offer useful suggestions for practical use of augmentation. Our research also offers incentives for researchers who work on test case generation techniques to consider reusing test cases to improve these techniques themselves.

In this dissertation, we provide techniques for cost-effective test suite augmentation. First, we investigate dynamic test case generation techniques that can use existing test cases and are suitable for our goal. We begin by using a concolic test case generation technique in test suite augmentation, in which we bring up the idea of test reuse. We present this work in Chapter 3.

Second, we use a genetic test case generation technique, further investigate the test reuse ideas and find a set of factors that affect the cost-effectiveness of test suite augmentation. This work is presented in Chapter 4.

After investigating basic test suite augmentation techniques, we explore other factors that could affect the cost-effectiveness of test suite augmentation and how these factors affect the test case generation techniques in the test suite augmentation context. This is presented in Chapter 5.

Fourth, we develop a hybrid test suite augmentation technique by considering the identified factors including test case generation techniques. We present this work in Chapter 6.

Finally, we extend our test suite augmentation idea to software product lines, which share similarities with versions of programs, and we build a test suite augmentation framework for software product lines. This work is presented in Chapter 7.

The contributions of this research are: (1) bringing the test reuse notion into test suite augmentation (Chapter 3); (2) identifying factors affecting the test suite augmentation process (Chapter 4); (3) developing cost-effective test suite augmentation techniques (Chapter 5 and 6); (4) extending the test suite augmentation idea into SPLs (Chapter 7); (5) providing insights into the practical use of augmentation for engineers (Chapters 5, 6 and 7).

Chapter 2

Background and Related Work

We provide background and describe related work on test suite augmentation, automated test case generation and software product lines.

2.1 Test Suite Augmentation

Let P be a program, let P' be a modified version of P, and let T be a test suite for P. Regression testing is concerned with validating P'. To facilitate this, engineers often begin by reusing T, and a wide variety of approaches have been developed for rendering such reuse more cost-effective via techniques such as regression test selection (e.g., [21, 63, 72, 76, 77, 103]) and test case prioritization (e.g., [34, 51, 94]).

Test suite augmentation techniques, in contrast, do not focus specifically on the reuse of T. Rather, they are concerned with the tasks of (1) *identifying affected elements* (portions of P' or its specification for which new test cases are needed), and then (2) *creating or guiding the creation of test cases that exercise these elements.*

Various algorithms have been proposed for identifying affected elements in software systems following changes. Some of these [13] operate on levels above the code

such as on models or specifications, but most operate at the level of code, and in this dissertation we focus on these. Code level techniques [11, 41, 74] use various analyses, such as slicing on program dependence graphs, to select existing test cases that should be re-executed, while also identifying portions of the code that are related to changes and should be tested. However, these approaches do not provide methods for generating actual test cases to cover the identified code.

Five recent papers [2, 67, 68, 78, 86] specifically address test suite augmentation. Two of these papers [2, 78] present an approach that combines dependence analysis and symbolic execution to identify chains of data and control dependencies that, if tested, are likely to exercise the effects of changes. A potential advantage of this approach is a fine-grained identification of affected elements; however, the papers do not present or consider any specific algorithms for generating test cases. A third paper [67] presents an approach to program differencing using symbolic execution that can be used to identify affected elements more precisely than the approach in [2, 78], and yields constraints that can be input to a solver to generate test cases for those requirements. A fourth paper [68] uses program analysis techniques to identify the parts of new programs that are affected by changes and apply symbolic execution only on these parts. None of the foregoing approaches, however, are integrated with reuse of existing test cases. Finally, a recent paper [86] presents an approach for using dynamic symbolic execution to reveal execution paths that need to be re-tested, in which existing test cases can be utilized.

In other related work [104], Yoo and Harman present a study of *test data augmentation*. They experiment with the quality of test cases generated from existing test suites using an heuristic search algorithm. While their work presents a technique that is similar to techniques that we consider in this dissertation (because it uses a search algorithm seeded with existing test cases), their goal is to duplicate coverage in

a single release in order to improve fault detection, not to obtain coverage of affected elements in a subsequent release.

2.2 Test Case Generation

While in practice test cases are usually generated manually, there has been a great deal of research on techniques for automated test case generation. For example, there has been work on generating test cases from specifications (e.g., [19, 55, 61]), from formal models (e.g., [6, 42, 92]), from semi-formal models (e.g., [14, 69]) and by random selection of inputs (e.g., [12, 20]).

In this work we focus on code-based test case generation techniques, many of which have been investigated in prior work. Among these, several techniques (e.g., [22, 28, 40]) use symbolic execution to find the constraints, in terms of input variables, that must be satisfied in order to execute a target path, and attempt to solve this system of constraints to obtain a test case for that path.

While the foregoing test case generation techniques are static, other techniques make use of dynamic information. Execution-oriented techniques [49] incorporate dynamic execution information into the search for inputs, using function minimization to solve subgoals that contribute toward an intended coverage goal. Goal-oriented techniques [37] also use function minimization to solve subgoals leading toward an intended coverage goal; however, they focus on the final goal rather than on a specific path, concentrating on executions that can be determined through analysis (e.g., through the use of data dependence information) to possibly influence progress toward that goal.

Several test case generation techniques use evolutionary or search-based approaches (e.g., [7, 30, 58, 66, 93]) such as genetic algorithms, tabu search, and simulated an-

nealing to generate test cases. Other work [18, 23, 39, 80, 81] combines concrete and symbolic test execution to generate test inputs. This second approach is known as *concolic testing* or *dynamic symbolic execution*, and has proven useful for generating test cases for C and Java programs. The approach has been extended to generate test data for database applications [35] and for Web applications using PHP [5, 96].

Implementations of several of the techniques discussed above are available. Java Path Finder (JPF) [46] is a representative symbolic execution tool; it began as a software model checker, but now is provided with various different execution models and extensions including some for generating test cases using symbolic execution. There are several tools (EXE [18], DART [39], CUTE [81], Crest [26] and KLEE [17]) that apply concolic testing to unit testing of C programs. There are also tools that apply search based techniques. For example, AUSTIN [50] is a structural test data generation tool (for unit tests) for the C language that uses search based techniques. AUSTIN is based on the CIL framework and currently supports a random search, as well as a simple hill climber that is augmented with a set of constraint solving rules for pointer type inputs. A second tool is called EvoSuite [38], and uses a hybrid approach for generating test cases for Java programs. EvoSuite generates and optimizes entire test suites with the goal of satisfying a coverage criterion.

2.3 Software Product Lines

Software product line (SPL) engineering has been shown to be a very successful approach to software development that allows engineers to build families of products that share some functionalities in a short time and with high quality [91]. This paradigm has received attention in industry and the software research community as it shows how the development of products can be improved and more importantly how

to respond quickly and effectively to market opportunities. According to Clements et al. [24], a software product line is "a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular segment or mission and that are developed from a common set of core assets in a prescribed way." SPL engineering re-uses and combines individual software building blocks guided by a feature model [24].

While SPL engineering can reduce the time required to develop products compared to the time required by traditional software development methodologies, it actually complicates the process of software testing. First, there are an exponential number of products in a product line with respect to the number of features, and these may all require some form of testing. Second, individual test cases may be valid and/or effective on only a subset of products, making their re-use across products difficult. Third, although products all share some common code, they are often tested as individual programs (rather than as a family of products) with the aim of satisfying the specifications for each product instance.

Software product line development promises to develop a family of products in short time with high quality at lower costs [91]. To achieve this goal, quality assurance becomes an essential part of the development process. Quality attributes such as correctness and reliability have begun to receive attention from industry and the research community as a consequence of the efforts to use more effectively the assets of an SPL throughout the products [36, 59]. There are many research papers concerned with testing and analyzing SPLs. McGregor [45] introduces a set of testing techniques for software product lines including core assets of testing. These techniques are similar to techniques used in software development of single systems. Several authors [9, 60, 62, 73] have proposed the use of use cases to systematically reuse test specifications. Olimpiew et al. [62] introduces CADeT (Customizable Ac-

tivity diagrams, Decision tables and Test specifications). CADeT is a functional test design method that utilizes feature-based test coverage criteria and use cases creating a reusable set of test specifications. Nebut et al. [60] use use cases to generate product-specific functional test objectives, and propose a tool to automate the test case derivation. References [71, 73] present ScenTED (Scenario-based Test case Derivation), a technique that supports the derivation of system test cases from domain requirements. Other research on methods for testing families of products include the PLUTO testing methodology [9], where the feature model is used to develop extended use cases, PLUCS (Product Line Use Cases), that contain variability which can formulate a full set of test cases using category partitioning for the family of products.

Thum et al. [87] categorize SPL analysis techniques into three categories: feature-based (when approaching the system from the individual features), product-based (when analyzing the system from the product perspective), and family-based (when viewing the entire product line as a single program). The same categorization can be applied to testing techniques.

Much of the work on reducing test effort for SPLs has been performed from either the feature or product view. Specification based testing approaches involve the use of the feature model to drive selection of products for testing [16], or to generate test cases that are specific to individual products [10]. Research on model based testing (called delta-oriented testing) has focused on the deltas (or changes) in behavior between products to direct the order of testing subsets of products while increasing re-use between them [52]. Recent trends in software product line testing have also resulted in approaches to reduce the combinatorial space in SPL testing through the use of sampling techniques [25, 64] or static analyses that limit the numbers of

combinations of features that must be tested together (or the number of test cases to run for each product) [82, 47].

Chapter 3

Basic Test Suite Augmentation Technique 1 - Using Concolic Testing

Arguably, the most important factor affecting test suite augmentation is the test case generation technique used. There are many test case generation techniques available, but the techniques we use in this work are dynamic ones that can leverage existing test cases, such as concolic and genetic test case generation techniques. In this chapter, we report work using concolic testing in the test suite augmentation context. (This work has appeared in [102].)

3.1 Concolic Test Case Generation

Concolic testing [18, 39, 81] concretely executes a program while carrying along a symbolic state and simultaneously performing symbolic execution of the path that is being executed. It then uses the symbolic path constraints gathered along the way to

generate new inputs that will drive the program along a different path on a subsequent iteration, by negating a predicate in the path constraints. In this way, concrete execution guides the symbolic execution and replaces complex symbolic expressions with concrete values when needed to mitigate the incompleteness of the constraint solvers [81]. For example, suppose the path constraint collected from one execution is $x > 1 \wedge y > 1 \wedge xy > 8$ with $x = 2$ and $y = 2$ and after negation the new path constraint becomes $x > 1 \wedge y > 1 \wedge xy < 8$. However, this path constraints contain a non-linear formula which is difficult for constrain solvers to solve. Since we have the concrete values for the two variables x, y, we can replace one of them with its concrete value and send the modified constraints to solver again. If we replace y with 2, the constrain becomes $x > 1 \wedge 2 > 1 \wedge 2x > 8$ and it will be easily solved by a solver and the results from the solver are a new input. Conversely, symbolic execution helps to generate concrete inputs for the next execution to increase coverage in the concrete execution scope.

In the traditional application of concolic testing, test case reuse is not considered, and the focus of test generation is on path coverage. First, a random input is applied to the program and the algorithm collects the path condition for this execution. Next, the algorithm negates the last predicate in this path condition and obtains a new path condition. Calling a constraint solver on this path condition yields a new input, and a new iteration then commences, in which the algorithm again attempts to negate the last predicate. If the algorithm discovers that a path condition has been encountered before, it ignores it and negates the second-to-last predicate. This process continues until no more new path conditions can be generated. Ideally, the end result of the process is a set of test cases that cover all paths. (In practice, bounds on path length or algorithm run-time can be applied).

Algorithm 1 DTSA

Require: Set T of test cases for P
CFG_p, P's control flow graph
$CFG_{p'}$, P''s control flow graph
Ensure: Set T' of test cases for P'
1: **Main Procedure**
2: $Goalset$=**RTS**
3: $Goalset$=**RerunAffected**
4: **if** $Goalset \neq \emptyset$ **then**
5: call **Augment**
6: **end if**
7:
8: **Procedure RTS**
9: call `Dejavu` to find affected test cases and update unaffected test cases' trace information and path conditions in P'
10: subtract branches in $CFG_{p'}$ covered by unaffected tests to form $Goalset$
11:
12: **Procedure RerunAffected**
13: rerun all affected test cases and gather their trace information and path conditions
14: subtract branches in $CFG_{p'}$ covered by affected test cases from $Goalset$
15:
16: **Procedure Augment**
17: $Predicatehit$=PickPredicatehit($Goalset$,$CFG_{p'}$)
18: order branches in $Predicatehit$
19: **for** each $e_j \in Predicatehit$ **do**
20: find all test cases covering the source of e_j
21: use their path conditions to do $DelNeg$ at $e_j's$ source
22: **if** path conditions after $DelNeg$ have not been seen before **then**
23: call $ConstraintSolver$ to solve them
24: **if** they are solvable **then**
25: put them into T'
26: run new generated test cases to obtain trace information, path conditions and coverage information
27: **if** they cover any branches in $Goalset$ **then**
28: subtract them from $Goalset$
29: update $Predicatehit$ according to $Goalset$
30: **end if**
31: **end if**
32: **end if**
33: **end for**

3.2 Augmentation Algorithm

Having introduced concolic testing, we now describe how we use concolic testing for test suite augmentation.

When program P evolves into P', coverage of P' by a prior test suite T can be affected in various ways. Some new code in P' may simply not be reached by test cases in T, and some test cases in T may take new paths in P', leaving code that was previously covered in P uncovered. *Regression test selection* (RTS) tech-

niques (e.g., [11, 76], for a survey see [75]) use information about P, P' and T to select a subset T' of T that encounter code changed for P', and thus may take different paths in P'. We can use these techniques to indicate such test cases. In this work we use one particular safe RTS technique, `Dejavu` [76], to help drive test suite augmentation. `DejaVu` performs simultaneous depth-first traversals on control flow graphs (CFGs) for procedures in P and P' to find *dangerous edges* that lead to code that has changed. *Execution traces* of test cases (bit vectors indicating whether edges were taken) on the old version of P are then used to select test cases that traversed dangerous edges in P. We then use information gathered previously for test cases in T to generate test cases that cover uncovered code to form a branch coverage test suite T' for P', using a modified concolic testing approach. We now discuss our approach as applied intraprocedurally (to single procedures).

Algorithm 1 presents our algorithm for directed test suite augmentation (DTSA). The *main procedure* of DTSA (lines 1-6), consists of three steps. Step 1 uses the `Dejavu` RTS technique to partition test suite T into two subsets, one containing *affected* test cases (test cases that reach dangerous edges) and one containing *unaffected* test cases (test cases that do not reach dangerous edges). Step 2 reruns the affected test cases, and calculates a *testing objective* which includes all of the branches in P' that need to be covered. Finally, based on information retrieved from prior executions of unaffected test cases and executions of affected test cases, Step 3 attempts to generate test cases to cover the branches in the testing objective.

Procedure *RTS* (lines 8-10) summarizes Step 1. The algorithm invokes `Dejavu` to find the sets of affected and unaffected test cases. We extend `Dejavu` to also find the corresponding unaffected test cases' trace information, path conditions and covered branches in P' as it synchronously traverses the CFGs, a process that succeeds because the traces and condition information that need updating all exist prior to

code changes and can be found as `Dejavu` traverses the graphs. Next, the algorithm (line 10) subtracts the branches covered by the unaffected test cases from $CFG_{p'}$, placing remaining branches into *Goalset*.

Procedure *RerunAffected* (lines 12-14) summarizes Step 2. The procedure reruns all affected cases that are selected by `Dejavu` to allow engineers to verify their outputs; during this re-execution, trace and path condition information for these test cases are also collected. If an affected test case covers branches in *Goalset*, the branches it covers are subtracted from that set. After all affected test cases have been run, control returns to the *Main Procedure* which then checks whether *Goalset* is empty (line 4). If it is, then our test suite is branch coverage adequate for P' and the algorithm terminates; otherwise, the algorithm continues.

The third and most significant step is procedure *Augment* (lines 16-33). Based on information gathered in the first two steps, the algorithm attempts to augment T using a concolic testing approach. The step begins (line 17) by locating, in *Goalset*, the branches for which the source node is a predicate node that is covered by at least one existing test case; these become the immediate targets for test generation. (These branches are ordered in line 18 for optimization reasons; we explain this later.)

The algorithm next enters a loop in which it selects branches one by one. For a given branch e_j with source (predicate) node p, the algorithm tries all path conditions for test cases whose execution traces reach p. For each such path condition, the algorithm deletes all predicates following p and negates p (the $DelNeg$ operation in line 21) to generate another path condition. If the generated path condition has not been seen before, the algorithm uses it to generate a new test case. Otherwise, the algorithm ignores it and moves on to the next path condition.

By calling a constraint solver to solve a modified path condition, the algorithm may obtain a test case to cover e_j. This test case and its trace and path condition

information are saved. If the test case's trace covers branches in *Goalset*, *Goalset* and *Predicatehit* are updated to indicate the new coverage. If the solver cannot solve the path condition, the algorithm considers other path conditions that cover the predicate. If all path conditions fail the branch may be unreachable, or it is reachable and other methods will need to be found to generate test cases to reach it.

Two aspects of DTSA that differentiate it from existing instantiations of concolic testing bear further discussion. First, the algorithm iterates through all path conditions whose execution traces reach p (line 20) instead of stopping when a test case has been generated for the initial target branch e_j. It does this because doing so may allow it to generate more test cases to reach predicates following e_j, which may control additional branches needing to be covered. This increases the possibility of covering branches that are later in flow.

Similar reasoning motivates the branch ordering that occurs in line 18. Test cases execute CFG edges from predicates that are reached earlier to those that are reached later, and thus, passing through earlier branches is a precondition to reaching later branches; achieving coverage of earlier predicates leads automatically to coverage of certain later ones, and also produces test cases whose path conditions that can be manipulated to generate new test cases to cover later branches. Thus, we order the branches in *Predicatehit* in breadth first order prior to using them.

3.3 Example

We use an example to illustrate how the algorithm works. Suppose we have five test cases for program `foo` in Figure 3.1, t_1=($x = 2$, $y = 2$), t_2=($x = 4$, $y = 4$), t_3=($x = 1$, $y = 0$), t_4=($x = 4$, $y = 3$), t_5=($x = -1$, $y = 0$), which are adequate for branch coverage in `foo` but not in `foo`$'$ due to the change in the second predicate.

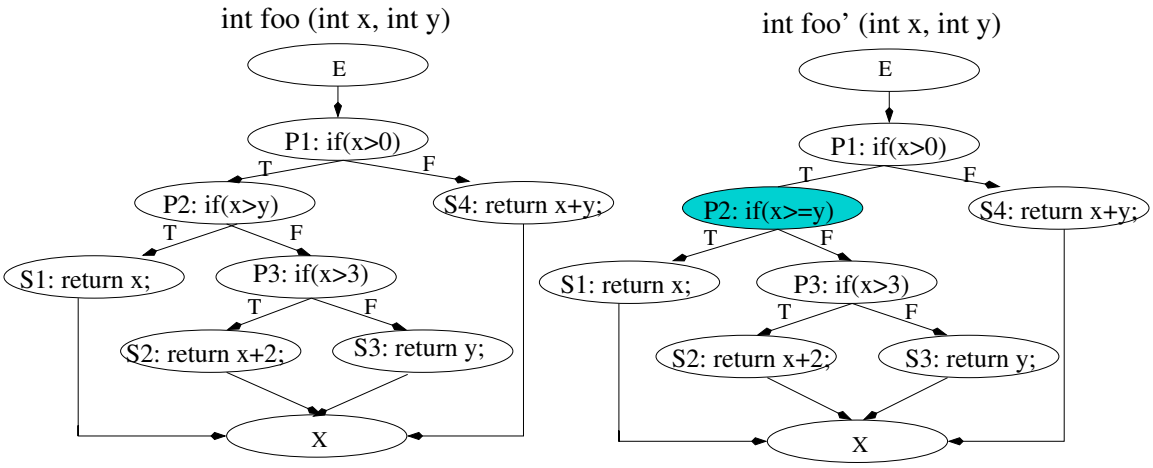


Figure 3.1: CFGs for two program versions

In Procedure *RTS*, t_1, t_2, t_3 and t_4 are selected as affected test cases, since their traces contain the predicate node P2, whose content has changed. Test t_5 is treated as unaffected and it also covers branches (P1, S4). *Goalset* contains (P1, P2), (P2, S1), (P2, P3), (P3, S2) and (P3, S3).

In Procedure *RerunAffected* for P', test cases t_1, t_2, t_3 and t_4 are rerun and their traces are obtained, all of which are (E, P1, P2, S1, X). After subtraction of the branches covered by these, *Goalset* contains (P2, P3), (P3, S2) and (P3, S3). Since P2 is covered by existing test cases, *Predicatehit* includes (P2, P3). Four test cases' executions exercise $P2$, so the algorithm enters line 21 to use their path conditions one by one to attempt to generate new test cases.

First, t_1's path condition, $(x > 0 \wedge x \geq y)$, is selected. *DelNeg* is applied to P2, obtaining another path condition, $(x > 0 \wedge x < y)$. Using the solver to solve it, a new test case is produced, t_6=$(x = 1, y = 2)$, that covers branches (P2, P3) and (P3, S3). At the same time, one more path condition, $(x > 0 \wedge x < y \wedge x \leq 3)$, is collected. Since this path covers some branches in *Goalset*, *Goalset* and *Predicatehit* are updated. Now *Goalset* has one branch left, (P3, S2), and *Predicatehit* contains

one branch, (P3, S2), since P3 is covered by t_6. The algorithm also uses the path conditions for t_2, t_3 and t_4 to generate new test cases. Since these have the same path conditions as t_1 after $DelNeg$ is applied to P2, the algorithm ignores them. Using (P3, S2) from $Predicatehit$ as the next objective, the algorithm enters the next iteration. Running $DelNeg$ on predicate P3 of t_6's path condition, another path condition, $(x > 0 \wedge x < y \wedge x > 3)$, is produced. By solving this, the algorithm obtains an input, $t7$=$(x = 4,\ y = 5)$, to cover branch (P3, S2). After updating $Goalset$, it becomes empty. At this point, the algorithm has generated test data covering all branches in `foo`$'$.

3.4 Extension to Interprocedural

Thus far we have presented our approach at the intraprocedural level, but as mentioned in Section 3.1, concolic testing has also been extended to function interprocedurally. Following similar extensions we extended our technique to the interprocedural level as well. The algorithm remains essentially as presented above, however, in addition to ordering branches within methods (line 18) we use depth first ordering to order methods based on the program's call graph, ensuring that branches in callers are covered first.

3.5 Implementation

We implemented our algorithms within the `Sofya` analysis system [48], which provides utilities for code instrumentation and CFG construction. We used the RTS module, `Dejavu`, provided with `Sofya`, to find affected and unaffected test cases. With the help of the Soot framework [90], we inserted code into P and P''s source code to

obtain the path condition for each execution. With CFGs and trace information, coverage information was obtained. Then we built a concolic testing module to use trace information to target uncovered branches and generate new test cases.

3.6 Empirical Study

To provide initial data on the potential applicability of our DTSA approach we conducted an empirical study. The research questions that we address are:

- **RQ1**: How efficient is DTSA at generating test cases to complete the coverage of P'?
- **RQ2**: How effective is DTSA at generating test cases to complete the coverage of P'?

The remainder of this section describes our objects of analysis, variables and measures, experiment setup, results, and threats to validity.

3.6.1 Objects of Analysis

Since our implementation functions only on programs that utilize arithmetic operations, as objects for our experiment we use 42 versions of one of the Siemens program, `Tcas`, which is available from the SIR repository [31]. `Tcas` includes an original version and 41 revised (faulty) versions, which we denote here as v_0 and v_k ($1 \leq k \leq 41$), respectively. The program is also equipped with a "universe" of 1608 distinct test cases, consisting of black and white box tests, and representing a population of potential test cases. Because `Tcas` was originally written in C and our implementation of DTSA functions on Java programs, we converted all of the versions of `Tcas` to

Java, as was done in [2]. The Java versions of `Tcas` have two classes, 10 methods and about 200 non-comment lines of code.

In practice when programs evolve, some test suites may need to be augmented while others may not. Therefore, in our study we utilize 1000 distinct test suites for v_0. While test suites are available in the SIR repository for the C version of `Tcas`, those suites were not coverage-adequate for the Java version. Thus, we employed the same greedy strategy utilized to produce the test suites for the C version to our Java version to create branch-coverage-adequate suites: randomly and greedily selecting test cases from the universe and adding them to the suite as long as they add coverage, and continuing until all reachable branches are covered.

3.6.2 Variables and Measures

Independent Variables. As independent variables we wish to consider our DTSA technique, and an alternative control technique. One such control technique could be found in existing augmentation techniques; however, as discussed in Section 2.1, all such existing techniques merely identify coverage requirements, leaving the creation of test cases to humans. Studies involving humans are expensive, and before conducting such studies it is reasonable to first determine whether our approach can be applied efficiently and effectively. As a control technique in this case, it makes sense to compare the approach to one in which, given P', concolic testing is reapplied from scratch with a goal of achieving branch coverage. Such a comparison allows us to assess the cost-benefit tradeoffs, in efficiency and effectiveness, that can be achieved by DTSA through its reuse of test cases.

Our independent variable thus consists of two techniques: the DTSA technique described in Section 3.2 and the basic concolic testing technique described in Section 3.1, modified to operate on branch coverage.

In our implementation of concolic testing, when we run a test case we record its associated path condition, and then we apply the *DelNeg* operation for all input-related predicates, attempting to generate modified path conditions that will lead to coverage of as many branches as possible. We use Yices [33] to solve these modified path conditions, yielding new test cases that cover uncovered branches. For each new test case we repeat this process, until we have utilized all test cases. We record all of the path conditions that have been used and ignore duplicates. When we apply the *DelNeg* operation to a predicate, if both branches are already covered, we ignore the modified path condition too. Ultimately, for each new version, this process yields a test suite that covers all reachable branches possible.

Dependent Variables and Measures. We chose two dependent variables and corresponding measures to address our research questions. The first variable relates to costs of the techniques, and the second measures the effectiveness of the techniques in generating test suites. These measures help us understand the general performance of the two techniques, in a manner that provides guidance on their relative strengths and weaknesses.

Technique cost. To measure technique cost, one approach would be to measure execution time. However, with prototype implementations and studies of comparatively small applications this measure is not an appropriate indicator of the costs in practice.

An alternative approach to cost measurement involves tracking the number of invocations, by techniques, of the operations that most directly determine technique

cost. For the techniques that we consider the operation that matters most involves the solution of constraints. Thus, in this study, we measure the number of constraint solver calls made by the techniques.

Technique effectiveness. We have chosen attainment of branch coverage as our test suite generation objective, and both of our techniques target it. For both techniques, however, there are limitations in achieving full branch coverage. When we use DTSA to generate test cases to cover all branches, we are limited by the existing test cases, and using these we may be unable to generate test cases that cover certain branches. In concolic testing, operations focus on predicates and on achieving coverage of these may omit generating additional test cases that could otherwise achieve coverage beneath these.

Given the foregoing, a measure of technique effectiveness involves its ability to generate coverage-adequate test suites, and thus, we track that coverage.

3.6.3 Experiment Setup

There are several issues regarding the setup for the experiment that need to be clarified. First, we conducted our experiments using v1.5.2 of the Java Runtime Environment (JRE) in a Linux environment. For consistency, all measurements for our object program were collected on the same system, a Pentium-M 1600 Mhz system with 1 Gb RAM running SuSE Linux 11.1.

Second, concolic testing can fare differently on different runs, depending on the inputs it randomly chooses initially. DTSA execution can fare differently on given test suites. Thus, it is important to compare data for both techniques on multiple executions, and on DTSA using multiple test suites. Accordingly, to conduct our study, for each version of the object program considered, and for each test suite aug-

mented for that program, we also conducted a run of concolic testing on a randomly generated set of initial inputs. This procedure also ensured equal numbers of runs of the two techniques, facilitating subsequent analysis.

Third, all code-coverage-based testing techniques face issues involving infeasible code components – components (e.g., branches, statements, and so forth) that cannot be reached on any executions and thus cannot be covered. Adequacy criteria are not required to cover infeasible components, and coverage adequacy is measured in terms of percentages of feasible components covered. Each version of our object program has some unreachable branches; we determined these by inspection and we measure coverage in terms of the feasible branches only.

Finally, given the versions of our programs and the test suites created for them, there are numerous cases in which test suites applied to changed versions do *not* leave reachable branches uncovered. These are cases where augmentation is *not* needed. We omit these cases and gather data for just the instances in which augmentation is necessary.

Given the foregoing, to conduct our study we performed the following two steps. First, we instrumented and created the CFG for v_0. We then executed v_0 on each of our 1000 test suites, collecting test trace information and path conditions for each test case in each suite, for use in the next step. Second, for each new version v_k ($0 < k \leq 41$) of v_0, we constructed the CFG for v_k. Then, for each version, for test suite T_k ($0 < k \leq 1000$), we performed the following steps. (1) We executed all test cases in T_k on v_k. (2) We ran algorithm DTSA on the CFGs for v_0 and v_k, together with the saved test trace and path condition data for v_0. If T_k needs to be augmented the algorithm performs the augmentation step, and we save the required data on performance. (3) If T_k needed to be augmented in the prior step we also performed

a run of concolic testing on v_k, starting from randomly generated program inputs, saving the required data on performance.

3.7 Threats to Validity

The primary threat to external validity for this study involves the representativeness of our object program, versions, and associated test suites. We have examined only one software subject, coded in Java, and other systems may exhibit different cost-benefit tradeoffs. We have considered only one set of versions of this subject, all based on changes made to the initial version, and sequences of releases may exhibit different tradeoffs. Subsequent studies are needed to determine the extent to which our results generalize, and the extent to which the approach scales to larger systems.

The primary threat to internal validity for this experiment is possible faults in the implementation of the algorithms and in tools we use to perform evaluation. We controlled for this threat through extensive functional testing of our tools. A second threat involves inconsistent decisions and practices in the implementation of the two techniques studied; for example, variation in the efficiency of implementations of common functionality between techniques could bias data collected. We controlled for this threat by having these two techniques implemented, insofar as this was possible, by the same developer, utilizing consistent implementation decisions and shared code.

Where construct validity is concerned, there are other metrics that could be pertinent to the effects studied, such as the total execution cost of the two techniques. However, in this initial study our subject is not sufficiently complex, and our tools not sufficiently optimized for run-time, to render comparisons of execution times meaningful.

3.8 Results and Analysis

For our object of study, we find 29 versions out of 41 versions needing to be augmented, because with the exception of unreachable branches they have other branches uncovered by old test suites. We analyze our data relative to those 29 versions for each of our research questions in turn.

3.8.1 RQ1: Number of Constraint Solver Calls

To address RQ1 (efficiency of DTSA compared to efficiency of concolic) we compare the number of constraint solver calls made by the two techniques. Figure 3.2 presents boxplots that show the number of solver calls per technique for the 29 versions. The x axis enumerates each version and technique using a suffix of D to denote DTSA and a suffix of C to denote concolic testing.

As the boxplots show, in most cases the number of solver calls made by DTSA is substantially less than the number made by concolic testing. In some cases, however, as in v13 and v37, there is some overlap in the ranges of the data sets.

To formally assess which mean differences are statistically significant we used a paired t-test. Our hypothesis is that the number of constraint solver calls of DTSA will be less than that of concolic testing. We expect to find negative mean differences (that is, DTSA consistently has fewer calls to the solver than concolic testing on average) in our data. Mean differences in which the t-test ρ (rho) value is less than or equal to 0.05 are deemed statistically significant.

Table 3.1 presents the result of our analysis, providing the mean differences in numbers of solver calls between DTSA and concolic testing per version, and ρ-values from the t-tests. Versions for which results are not given are those in which only one test suite needs to be augmented, or, in the case of v15, where two pairs of the values

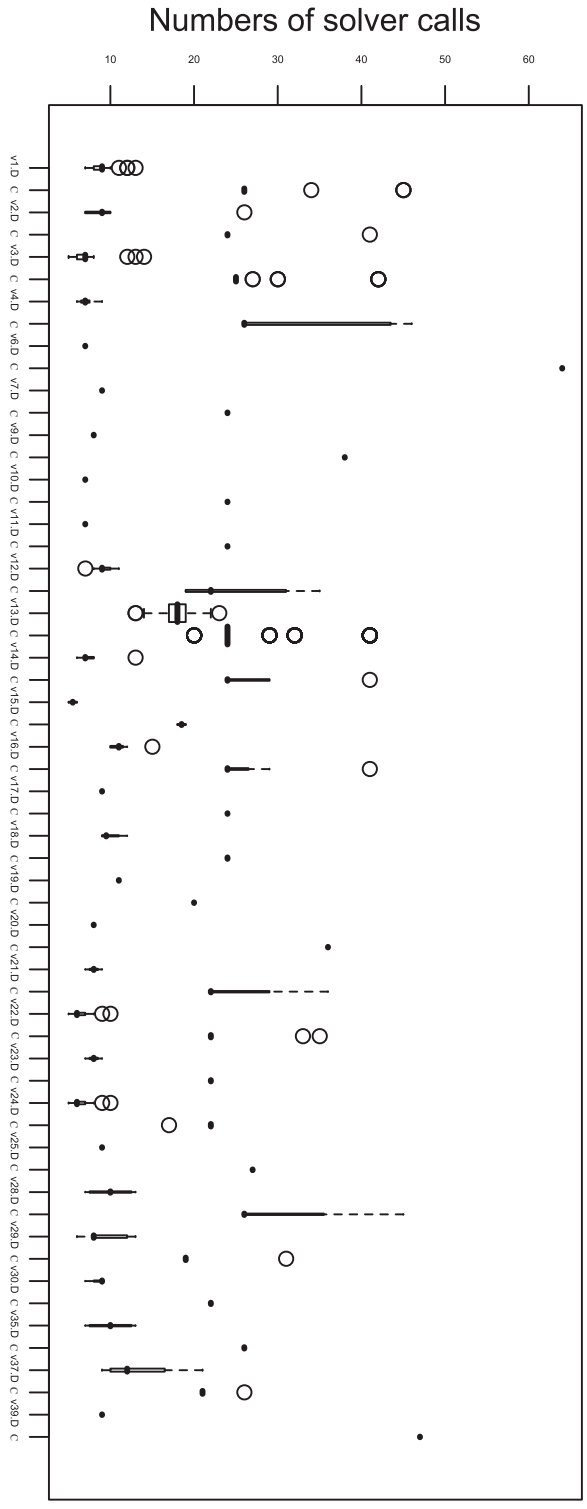


Figure 3.2: Solver calls: DTSA vs Concolic

Table 3.1: Differences in Numbers of Solver Calls

Version	Mean difference	ρ-value	Version	Mean difference	ρ-value	Version	Mean difference	ρ-value
v1	-19.42	< 0.0001	v13	-7.57	< 0.0001	v23	-14.00	< 0.0001
v2	-15.60	< 0.0001	v14	-19.67	0.001	v24	-14.92	< 0.0001
v3	-20.46	< 0.0001	v15	-	-	v25	-	-
v4	-25.64	< 0.0001	v16	-15.86	0.001	v28	-20.75	0.034
v6	-	-	v17	-	-	v29	-10.54	< 0.0001
v7	-	-	v18	-14.00	< 0.0001	v30	-13.50	< 0.0001
v9	-	-	v19	-	-	v35	-16.00	0.002
v10	-	-	v20	-	-	v37	-7.95	< 0.0001
v11	-	-	v21	-17.50	0.02	v39	-	-
v12	-15.22	< 0.0001	v22	-17.53	< 0.0001	total	-10.47	< 0.0001

Table 3.2: Coverage Results

	Branches missed by Concolic	Branches missed by DTSA		Branches missed by Concolic	Branches missed by DTSA		Branches missed by Concolic	Branches missed by DTSA
v1	2.96	0	v13	2.89	0	v23	12	3.00
v2	2.00	2.60	v14	2.83	2.83	v24	3.08	1.00
v3	2.87	2.00	v15	3.00	0	v25	3.00	0
v4	2.91	0	v16	2.86	0	v28	3.00	0
v6	3.00	2.00	v17	3.00	0	v29	2.77	0
v7	3.00	0	v18	3.00	0	v30	2.75	0
v9	6.00	2.00	v19	2.00	0	v35	3.00	0
v10	5.00	5.00	v20	6.00	3.00	v37	2.95	0
v11	6.00	5.00	v21	10.50	3.00	v39	3.00	0
v12	3.00	0	v22	3.15	1.00	total	3.88	1.12

are the same and a t-test cannot return a result. As the table shows, all of the mean differences are less than 0 and all computable ρ-values are less than 0.05, supporting our hypothesis.

3.8.2 RQ2: Coverage Criteria

Next we explore RQ2, which involves the effectiveness of DTSA relative to concolic testing, in terms of achieving adequate branch coverage when augmenting test suites.

Table 3.2 displays the mean numbers of branches not covered by the test suites generated by the two techniques. The total number of reachable branches ranges from 79 to 84 for all versions and all of the branches listed in this table are reachable — infeasible branches were calculated by inspection and excluded. For most versions of our object program, concolic testing left about three branches uncovered, with exception of v9, v10, v11, v21 and v23. On the first three of these, five or six branches were left uncovered, while on the last two, over 10 were left uncovered. DTSA, in contrast, achieved 100% branch coverage on 17 versions, with an average of three on most other versions. On all but versions v10 and v14, however, DTSA achieved better coverage than concolic testing.

3.9 Discussion

Our results show that, for the object program and versions considered, the DTSA technique can be applied effectively, and more efficiently than a full application of concolic testing. In general, when using DTSA to do test suite augmentation, we are able to restrict our attention to a smaller number of testing objectives than full concolic testing, resulting in substantially fewer solver calls.

However, DTSA is not always more efficient than full concolic testing. In full concolic testing, it is possible to generate a single test case that covers several branches. In the final step of DTSA, when the algorithm attempts to cover a branch, it may need to try all of the path conditions that apply, performing the $DelNeg$ operation on a specific predicate several times and calling the solver to solve each modified path condition. This process may ultimately lead to unnecessary solver calls. We believe that this explains the cases (v13 and v37) in which some runs of DTSA utilized more solver calls than some runs of concolic testing.

As Table 3.2 illustrates, an application of full concolic testing is typically less effective than an application of DTSA. This effectiveness gap is particularly strong on versions v21 and v23. We attempted to discern the reasons behind this gap, and we conjecture that the changes to these versions are likely responsible. In both versions a function is replaced by a value that is one of two possible returned values from this function at different positions. The values returned by the function have impacts on subsequent predicates encountered in execution. The changes to return values render it difficult (but not impossible) to cover some branches. However, this difficulty is lessened for DTSA where multiple test cases are available to work from.

The lessons suggested by this analysis are that the modifications made to programs can matter, but also, having multiple inputs (multiple test cases) available that reach code close to changes can facilitate generation of applicable tests. This re-use of prior test cases is not available to an ordinary application of concolic testing "from scratch", and it appears to make a difference in our ability to generate test cases that focus on modifications.

3.10 Conclusions

We presented a new test suite augmentation technique, DTSA, that combines an existing RTS technique with a modified concolic testing approach to generate test cases that reach code that has not been covered by old test cases. The results of our empirical study provide evidence that DTSA was more effective and efficient than using a concolic testing approach from scratch.

Chapter 4

Basic Test Suite Augmentation Technique 2 - Using a Genetic Algorithm

In this Chapter, we describe how we use genetic algorithm for test suite augmentation. This work has appeared in [99]. We begin by explaining the factors that affect performance when using a genetic algorithm for augmentation. Then we describe how we investigate one of the factors through an empirical study.

4.1 Genetic Test Case Generation

Genetic algorithms for structural test case generation begin with an initial (often randomly generated) test case population and evolve the population toward targets that can be blocks, branches or paths in a program [56, 88, 95]. To apply such an algorithm to a program, the test inputs must be represented in the form of a chromosome, and a fitness function must be provided that defines how well a chromosome satisfies the

intended goal. The algorithm proceeds iteratively by evaluating all chromosomes in the population and then selecting a subset of the fittest to mate. These are combined in a crossover stage where information from one half of the chromosomes is exchanged with information from the other half to generate a new population. A small percentage of chromosomes in the new population are mutated to add diversity back into the population. This concludes a single generation of the algorithm. The process is repeated until a stopping criterion has been met.

4.2 Factors Affecting Augmentation When Using Genetic Algorithms

We have identified several factors that are independent of genetic algorithm parameters, but that could potentially affect how well such algorithms perform by impacting both population size and diversity. We now describe these factors.

F1: Algorithm for identifying affected elements. As discussed in Section 2.1, various algorithms have been proposed for identifying affected parts of software systems following changes. We have utilized one such algorithm in our initial work on augmentation (Chapter 3) in the context of a concolic test generation algorithm. Where genetic algorithms are concerned, the numbers and complexity of identified affected elements can clearly impact the cost of subsequent test generation efforts by affecting the numbers of inputs that the algorithms must generate, and the complexity of the paths through code that the algorithms must target.

F2: Characteristics of existing test suites. Test suites can differ in terms of size, composition, and coverage achieved. Such differences in characteristics could potentially affect augmentation processes. For example, the extent to which an existing

suite achieves coverage prior to modifications can affect the number and locations of coverage elements that must be targeted by augmentation. Furthermore, test suite characteristics can impact the size and diversity of the starting populations utilized by genetic algorithms.

F3: Order in which affected elements are considered. For genetic algorithms that utilize existing test cases to generate new ones, the order in which a set of affected elements are considered can affect the overall cost and effectiveness of test generation, and thus, the cost and effectiveness of augmentation. For example, if elements executed earlier in a program's course of execution are targeted first, and if test cases are found to reach them, these may enlarge the size and diversity of the population of test cases reaching other affected elements later in execution, giving the test generation algorithm additional power when it considers those – power that it would not have if elements were considered in some other order.

F4: Manner in which test cases are utilized. Given a set of affected elements, a set of existing test cases, and an augmentation algorithm that uses existing test cases to generate new ones, there are several ways to interleave the use of existing and newly generated test cases in the augmentation process. Consider, for example the following approaches:

1. For each affected element, let the augmentation approach work with all existing test cases.
2. For each affected element, analyze coverage of existing test cases to determine those that are likely to reach it and let the augmentation approach use these.
3. For each affected element, let the augmentation approach use existing test cases which, based on their execution of the modified program, can reach it.

4. For each affected element, let the augmentation approach use existing test cases that can reach it in the modified program (approach 3), together with new test cases that have been generated thus far and reach it.

5. For each affected element, begin with approach 4 but select some subset of those test cases, and let the augmentation approach use these.

Each of these approaches involves different tradeoffs. Approach 1 incurs no analysis costs but may overwhelm a genetic algorithm approach by providing an excessively large population. Approach 2 reduces the test cases used by the genetic algorithm but in relying on prior coverage information may be imprecise. Approach 3 passes a more precise set of test cases on to the genetic algorithm, but requires that these first be executed on the modified program. None of the first three approaches takes advantage of newly generated test cases as they are created, and thus they may experience difficulties generating test cases for new elements due to lack of population diversity. Approaches 4 and 5 do use newly generated test cases along with existing ones, and also use new coverage information, but differ in terms of the number of new test cases used, again affecting size and diversity.

Among these four factors, we believe that F4 is of particular interest, because it provides a range of approaches potentially differing in cost and effectiveness for using genetic algorithms in augmentation tasks. We thus set out to perform a study investigating this factor.

4.3 Case Study

To investigate factor F4, we fix the values of other factors at specific settings as discussed below. The research questions we address are:

RQ1: How does factor F4 affect the cost of augmentation using a genetic algorithm?

RQ2: How does factor F4 affect the effectiveness of augmentation using a genetic algorithm?

4.3.1 Objects of Analysis

For our experiment, we chose a non-trivial Java software application, `Nanoxml`, from the SIR repository [31]. `Nanoxml` has multiple versions and more than 7000 lines of code. `Nanoxml` is an XML parser that reads string and character data as input. It has many individual components which realize different functionality. Drivers are used to execute the various components. We focused on performing augmentation as the system goes through three iterations of evolution, from versions v_0 to v_1, v_1 to v_2, and v_2 to v_3. In other words, we augmented the test suite for v_1 using the suite for v_0, augmented the test suite for v_2 using the suite for v_1 and augmented the test suite for v_3 using the suite for v_2. The test suites for v_0, v_1, and v_2 contain 235, 188, and 234 specification-based test cases applicable to the following versions, respectively. These test cases cover 74.7%, 83.6% and 78.5% of the branches in versions v_0, v_1, and v_2, respectively.

4.3.2 Genetic Algorithm Implementation

To investigate our research questions we required an implementation of a genetic algorithm tailored to fit our object program. We used an approach suitable to the object, that could be modified to work on other string-based programs.

Our chromosome consists of strings containing two parts: test drivers that invoke an application and input files (XML files) that are the target of the application. The

driver is a single gene in the chromosome. The XML files give way to a set of genes; one for each character in the file.

We treat each part of the chromosome differently with respect to crossover and mutation. For the test drivers, we use a pool of drivers that are provided with the application. We do not modify this population, but rather modify how it is combined with the input files that are evolved. We do not perform crossover on the drivers; we use only mutation. When a chromosome's driver gene is selected for mutation, the entire driver is replaced with another (randomly selected) valid driver from our pool of potential drivers. This prevents invalid drivers from being generated.

In the XML part of our chromosome, we perform a one point crossover by randomly selecting a line number that is between 0 and the number of lines of the smaller file. We then swap everything between files starting at that row to the end of the file. We do not test the file for well-formed XML, but rather use it as-is. During mutation, each character in the input file is considered a unit. We randomly select the new character from the set of all ASCII upper and lower case letters combined with the set of special characters found in the pool of input files, such as braces and underscores.

Our search targets are branches in the program, therefore for our fitness function we use the *approach level* described in [97]. For our initial implementation, for the sake of simplicity and due to the instrumentation overhead required, we did not combine this with branch distance. We nonetheless achieved good convergence on these programs; still, research suggests that branch distance is an important part of the fitness function [3] and we intend to consider it in the future.

The approach level is a discrete count measuring how far we were from the predicate controlling the target branch in the CFG when we deviated course. The further away we are from this predicate, the higher the fitness, therefore we are trying to min-

imize this value. If we reach the predicate leading to our target, the approach level is 0. There are other fitness functions, for example, a recent one introduced in [38]. However, in [38], multiple targets are considered at one time, which is different from our approach here.

For selection, we select the best half of the population to generate the next generation; we keep the selected chromosomes in the new generation. We rank the chromosomes and divide them into two parts. The first chromosome in the first half is mated with the first chromosome in the second half, and the second chromosome in the first half with the second chromosome in the second half, etc.

We use a three stage mutation. First we select 30% of the test cases in the population for mutation and mutate the driver for theses test cases. Next we select 30% of the lines in the file part of the chromosome for these test cases, and then select 30% of the genes in these rows for mutation. Our stopping criterion is coverage of the required program branch or ten generations of our genetic algorithm, whichever is reached first.

Note that we manually tuned the parameters used by our algorithm so that we can cover as many branches of a program for a straight test case generation problem before starting our experiments, and this process also led us to choose the values of 30/30/30. However, we performed this tuning for normal test case generation, not augmentation, and we did it on the base version of the program. This is appropriate where augmentation is concerned, because in a regression testing setting, a test generation algorithm can be tuned on prior versions of the system before being utilized to augment test suites for subsequent modified versions.

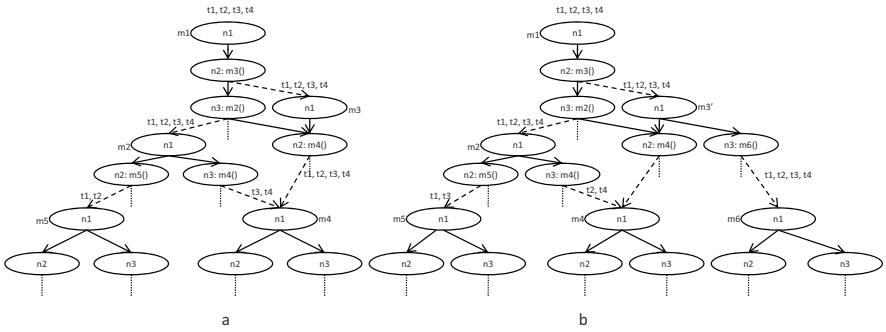


Figure 4.1: Partial control flow graphs for two versions of a program

4.3.3 Factors, Variables, and Measures

We describe our factors, variables and measures next.

Fixed Factors. Our goal being to consider only the effects of factor F4, we selected settings for the other factors described in Section 4.2 and held these constant.

For better understanding, we use the example in Figure 4.1 to explain factors. The figure shows portions of two versions of a program, in a control flow graph notation. The graph on the left corresponds to an initial version a and the graph on the right corresponds to a subsequent version b. Nodes represent statements within methods, and root nodes are indicated by labels $m1$ through $m6$. Solid lines represent control flow within methods and dashed lines represent calls. Labels on dashed lines represent test cases in which the associated method call occurs. From version a to b, changes occur in method $m3$ in which one branch is added to call a new method $m6$. Other methods remain unchanged.

F1: Algorithm for identifying affected elements.

As affected elements we use a set of potentially impacted coverage elements in P'. To calculate these, we use the analysis at the core of the `DejaVu` regression test selection technique [76] as presented in Section 3.2. After using the analysis identi-

fying dangerous edges, we treat methods containing dangerous edges as "dangerous methods", and then apply an algorithm that walks the interprocedural control flow graph for P' to find the set of affected methods that can be reached via control flow paths through one or more of the dangerous methods. All branches contained in affected methods are targets for augmentation.

In our example, $m3'$ contains a dangerous edge, so it is a dangerous method, and $m4$ and $m6$ are reachable via interprocedural control flow from the dangerous edge in $m3'$, so they are affected methods. Further, $m3$'s return value to $m1$ is affected, so $m1$ is also affected. Method $m2$ is called along the path from $m3$ to the exit node of $m1$, so it too is affected. Continuing to propagate impact, $m5$ and $m4$ are called by $m2$, so they are both affected. In this example all methods and all branches contained in them are affected, but in general this may not be the case.

F2: Characteristics of existing test suites. Our test suites T are those provided with `Nanoxml`. As described above, they are specification-based and operate at the application testing level, and they achieve branch coverage levels ranging from 74.7% to 83.6% on our versions.

F3: Order in which affected elements are considered. As an ordering, we used an approach that causes individual methods to be considered in top-down fashion in control flow, thus approximating the consideration of affected elements in such a fashion. The approach applies a depth first ordering to all affected methods in the call graph for P'. The effect of this approach is to cause augmentation to be applied to a particular method only after its predecessors in the call graph have been considered, which may allow test cases generated earlier to cover methods addressed later. Note, however, that this approach may be imprecise in relation to cycles, and in the order in which it considers individual branches within individual methods. As an

Table 4.1: Disposition of Test Cases Under the Five Treatments for the Example of Figure 1

Treatment	$m1$	$m2$	$m3$	$m4$	$m5$	$m6$
1	$t1$, $t2$, $t3$, $t4$	$t1$, $t2$, $t3$, $t4$	$t1$, $t2$, $t3$, $t4$	$t1$, $t2$, $t3$, $t4$	$t1$, $t2$, $t3$, $t4$	$t1$, $t2$, $t3$, $t4$
2	$t1$, $t2$, $t3$, $t4$	$t1$, $t2$, $t3$, $t4$	$t1$, $t2$, $t3$, $t4$	$t1$, $t2$, $t3$, $t4$	$t1$, $t2$	
3	-	-	$t1$, $t2$, $t3$, $t4$	$t2$,$t4$	-	$t1$, $t2$, $t3$, $t4$
4	-	-	$t1$, $t2$, $t3$, $t4$	$t2$, $t4$, $t1'$	-	$t1$, $t2$, $t3$, $t4$
5	-	-	$t1$, $t2$, $t3$, $t4$	$t2$, $t1'$	-	$t1$, $t2$, $t3$, $t2'$

example, in Figure 4.1, the ordering of methods in version b imposed by our approach is $m1$, $m3'$, $m2$, $m5$, $m4$ and $m6$.

Independent Variables. Our independent variable is factor F4, the "treatment of test cases" factor, and we use the five treatments described in that section, more precisely described here. To facilitate the description, Table 4.1 presents information on the disposition of test cases achieved by the treatments, applied to the example in Figure 4.1.

Treatment 1. For each affected element e in method m, all existing test cases in T are used to compose the initial population for the genetic algorithm. In this case we may have a large population for the genetic algorithm, which may cause it to take a relatively long time to complete the augmentation task for P'. However, this approach does increase the variety in the population which could improve the effectiveness of the search. In Figure 4.1, for all target branches, we use all four test cases $t1$ to $t4$ to compose the initial population.

Treatment 2. For each affected element e in method m, all existing (old) test cases that used to reach m in P, denoted by $T_{old:P}$, are used to compose the initial population for the genetic algorithm. In this case since we are using old coverage information, we avoid running all existing test cases on P' first and focus on the changes from P to P'. However, if we have new methods in P', since there are no existing

test cases available to reach them, we lose opportunities to perform augmentation for them and may lose some coverage.

In Figure 4.1, in this case, for $m1$, $m2$, $m3$ and $m4$ we use all test cases to form the initial population, since all the test cases reach them in version a. For $m5$, we use just $t1$ and $t2$. In this case, since there is no method $m6$ in version a and there are no existing test cases that reach it in that version, we cannot do augmentation for $m6$ directly.

Treatment 3. For each affected element e in method m, all existing test cases that reach m in P', denoted by $T_{old:P'}$, are used to compose the initial population for the genetic algorithm. In this case we need to run all existing test cases on P' first and then we use the new coverage information, which is more precise than in treatment 2 since these test cases are near to our target, and this helps the genetic algorithm in its search. Also, $T_{old:P'} \subseteq T$, so we may lose some variety in the population, but we may save time in the entire process since we have fewer test cases to execute in each iteration.

Considering Figure 4.1, when we run all existing test cases on version b, some of them take new execution paths. Methods $m3$, $m4$ and $m6$ contain uncovered branches after checking the coverage of all existing test cases on b. For $m3$, all existing test cases still reach it in b so they are used in its initial population. Because of the change in $m3$, all test cases that used to reach $m3$ take different paths and reach $m6$ so they are used to form the initial population for $m6$. There are only two test cases, $t2$ and $t4$ from $m2$, that reach $m4$ and they are used to form the initial population for $m4$.

Treatment 4. For each affected element e in method m, all existing test cases that reach m in P' ($T_{old:P'}$) and all newly generated test cases that obtain new coverage in version b, denoted by $T_{new:P'}$, are used to compose the initial population for the genetic algorithm. Here, we also need to run all existing test cases first to obtain

their new coverage information. Adding new test cases brings greater variety to the population, which increases the size of the population but may increase running time.

In Figure 4.1, the same test cases used in treatment 3 are used to form the initial population for $m3$, since when we do augmentation for $m3$ there have not been test cases generated. We generate a test case $t1'$ for $m3$ to cover the branch that calls $m4$, so when we do augmentation for $m4$ we include $t1'$ with $t2$ and $t4$ to form the initial population for it. For $m6$, $t1'$ does not reach it so we still use only the existing test cases that reach it in its initial population.

Treatment 5. For each affected element e in method m, all existing and generated test cases generated that reach m in P' ($T_{old:P'} \cup T_{new:P'}$) are considered applicable, but before being utilized they are considered further. A reasonable size of population is determined (in our case we chose the size that would be required by using treatment 3) and initial test cases are selected from the applicable test cases to compose the population. In this case, a good selection technique should be used to choose test cases that form a population which has the best variety for genetic algorithm. In our case, we chose test cases according to their branch coverage information on P'. More precisely, if we need to pick s test cases, we do the following:

- Find all paths from the root of P's call graph to m.
- Put the methods along these paths, including m, into set M_{pre}.
- Find branches in all methods in M_{pre}.
- Order the candidates on these branches in terms of coverage
- Pick the first s of the ordered candidates.

In Figure 4.1, $m3$ is in the same situation as with treatment 4, so the same test cases are used here. For $m4$, when we perform augmentation for $m3$ we generate thousands of test cases, some that increase coverage such as $t1'$ and others that cover

branches covered by other existing test cases. Next, we order all test cases that reach $m3$ and select two that cover most of the branches in $m1$, $m2$, $m3$ and $m4$. We select $t2$ and $t1'$ here, since they both pass through $m1$, cover one branch in $m2$ and $m3'$ separately, and pass through one of the branches in $m4$. The same procedure is followed on $m6$. For example, $t2'$ and $t3'$ are generated and reach $m6$ and including these with all existing test cases we select $t1$, $t2$, $t3$ and $t2'$ to form the initial population for $m6$.

Dependent Variables and Measures. We chose two dependent variables and corresponding measures to address our research questions. The first variable relates to costs associated with employing the different test case treatments and the second relates to the effectiveness associated with the different treatments.

Cost of employing treatments. To measure the cost of employing treatments, one approach is to measure the execution time of the augmentation algorithm under each treatment. However, measuring time in a manner that facilitates fair comparison requires the use of identical machines, and for the sake of parallelism we ran our experiments on a set of machines and under different system loads.

An alternative approach to cost measurement involves tracking, under each test case treatment, the number of invocations by augmentation techniques of the operations that most directly determine technique cost. For the augmentation technique that we consider the operation that matters most involves the execution of test cases by the genetic algorithm, because if that algorithm finds a target soon it will use fewer iterations, execute fewer test cases and require less running time. Thus, in this study, we use the number of test cases executed by the genetic algorithm as a measure of cost.

Effectiveness of employing treatments. To assess the effectiveness of using different test case treatments, we measure the progress that augmentation can make toward its coverage goal under each treatment in terms of the numbers of branches covered.

4.3.4 Experiment Setup

To conduct our study we performed the following steps. For each v_k ($0 \leq k \leq 2$) we instrumented and created the CFG for v_k using Sofya [84]. We then executed v_k on the test suite T_k for v_k, collecting test coverage for use in the next step. Next, we created the CFG for v_{k+1} and determined the affected methods and target coverage elements (branches) in v_{k+1} using the `Dejavu` algorithm as described in Section 4.3.3. These target elements are the affected elements we attempt to cover with our genetic test case generation algorithm under the different test case treatments. Further, because a genetic algorithm can fare differently on different runs, for each test case treatment we executed the test case generation algorithm fifteen times, and we consider data taken from all of these runs in our subsequent analysis.

4.4 Study Limitations

There are several limitations to our results. The first is the representativeness of our object program, versions, and test suites. We have examined only one system, coded in Java, and other systems may exhibit different cost-benefit tradeoffs. We have considered only three pairs of versions of this subject, and others may exhibit different tradeoffs. A second threat pertains to algorithms; we have utilized only one variant of a genetic test case generation algorithm, hand-tuned, and under particular settings of factors F1, F2, and F3. Subsequent studies are needed to determine the extent to which our results generalize.

Another limitation involves possible faults in the implementation of the algorithms and in tools we use to perform evaluation. We controlled for this through extensive functional testing of our tools.

Finally, there are other metrics that could be pertinent to the effects studied. Given tight implementations and controls over environments, time could be measured. Costs of engineer time in employing methods could also matter.

4.5 Results and Analysis

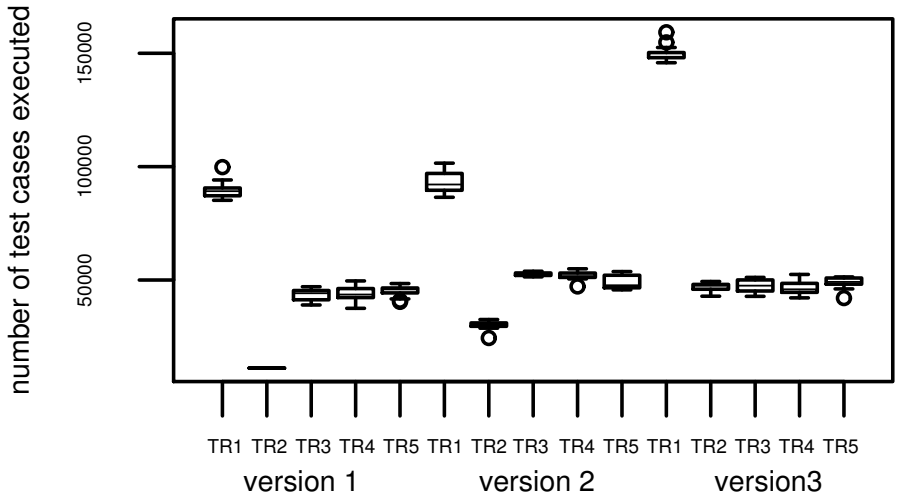


Figure 4.2: Costs of applying the five treatments, per treatment and version

Figures 4.2 and 4.3 present boxplots showing the data gathered for our independent variables. The first figure plots the number of test cases executed (vertical axis) against each treatment (TR1, TR2, TR3, TR4, and TR5) per version (v_1, v_2 and v_3). The second figure plots the number of covered branches against each treatment per version.

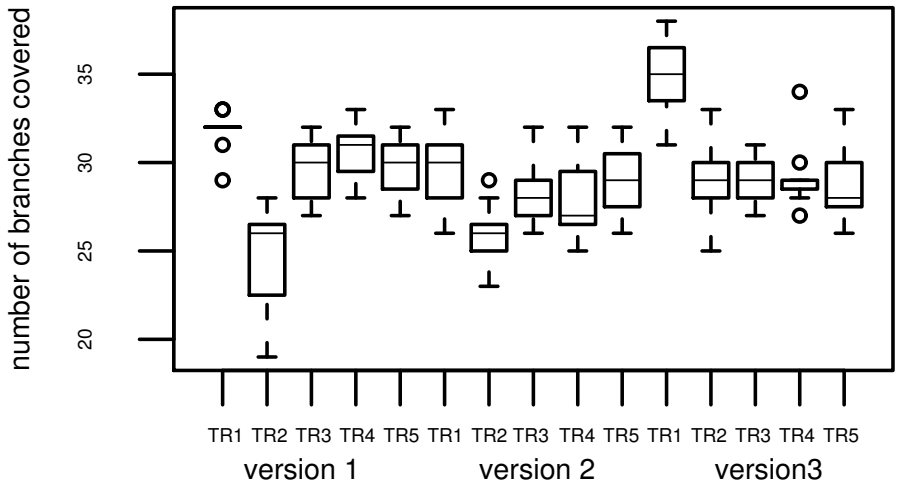


Figure 4.3: Coverage obtained in applying the five treatments, per treatment and version

4.5.1 RQ1: Costs of Augmentation

To address RQ1 (cost of the treatments) we compare the number of test cases executed by the treatments. As the boxplots show, in all cases the number of test cases executed by TR1 is substantially greater than the number executed by the other four treatments. On versions v_1 and v_2, but not v_3, TR2 results in the execution of the fewest test cases. TR5 appears to differ slightly from other treatments on v_2 and v_3, but in other cases treatment results appear similar.

We performed per version ANOVAs on the data for a significance level of 0.05; Table 4.2 reports the results. The first three rows pertain to cost comparisons. As the $p-$values in the rightmost column show, there is enough statistical evidence to reject the null hypothesis on all three versions; that is, the mean costs of the five different treatments are different in each case.

The ANOVA evaluated whether the treatments differ, and a multiple comparison procedure using Bonferroni analysis quantifies how the treatments differ from each

Table 4.2: Results of ANOVA Analysis

	Df	Sum Sq	Mean Sq	F value	Pr
v1_cost	4	4.04e+10	1.01e+10	109.43	<2.2e-16
v2_cost	4	3.18e+10	7.96e+09	1027.30	<2.2e-16
v3_cost	4	6.13e+10	6.13e+10	68.40	<4.4e-12
v1_cov	4	459.15	114.79	36.84	<2.2e-16
v2_cov	4	124.86	31.21	7.83	2.9e-0.5
v3_cov	4	427.20	106.80	35.19	6.6e-16

Table 4.3: Results of Bonferroni Means Test on Cost

(A) cost								
v1			v2			v3		
	Mean	Gr		Mean	Gr		Mean	Gr
TR2	11136	A	TR2	29960	A	TR4	46522	A
TR3	43355	B	TR5	49347	B	TR2	46752	A
TR4	43914	B	TR4	51811	B, C	TR3	47262	A
TR5	45086	B	TR3	52569	C	TR5	48961	A
TR1	84302	C	TR1	93048	D	TR1	149856	B

other. Table 4.3 presents the results of this analysis for the three versions considering treatment cost, ranking the treatments by mean. Grouping letters (in columns with header "Gr") indicate differences: treatments with the same grouping letter were not significantly different. In v_1 the five treatments are classified into three groups: TR1 and TR2 are most and least costly, respectively, while TR3, TR4 and TR5 are in a single group intermediate in cost. In v_2 the treatments are classified into four groups; TR1 remains most costly and TR2 least costly, but TR3, TR4, and TR5 form two overlapping classes in terms of cost, with TR3 significantly more costly than TR5. In v_3, TR1 is most costly and other techniques are classified into a single less costly group.

4.5.2 RQ2: Effectiveness of Augmentation

Next we explore RQ2, which involves the effectiveness of the five treatments in terms of achieving branch coverage when augmenting test suites. As mentioned above, after running all existing test cases we found that 68 branches needed to be covered for v_1, 77 for v_2 and 100 for v_3. Among these, several branches are difficult to cover in

Table 4.4: Results of Bonferroni Means Test on Coverage

(B) coverage								
v1			v2			v3		
	Mean	Gr		Mean	Gr		Mean	Gr
TR1	31.9	A	TR1	29.4	A	TR1	35.0	A
TR4	30.6	A, B	TR5	28.9	A	TR3	29.1	B
TR5	29.9	B	TR3	28.1	A	TR4	29.1	B
TR3	29.7	B	TR4	27.9	A	TR2	29.0	B
TR2	24.7	C	TR2	25.7	B	TR5	28.6	B

each version, since `Nanoxml` is a parser for XML and often requires specific characters at specific positions which can be difficult to satisfy. Also, in v_2 and v_3, since the test drivers we used are for previous versions and we did not mutate them to trigger some methods in the new version that are important for improving coverage, we were unable to cover 13 and 3 branches, respectively.

The boxplots in Figure 4.3 show the numbers of branches covered by each treatment in the fifteen runs for the three versions. On the three versions, TR1 covers the most branches. For v_1 and v_2, TR2 covers the fewest branches and TR3, TR4 and TR5 have similar results, while in v_1, TR4 appears better and in v_2 TR5 appears better. For v_3, the other four treatments return similar results.

Table 4.2 displays the results of ANOVAs on coverage data for the versions (bottom three rows). The $p-$values indicate that the five treatments do have significant effects on coverage for all three versions.

Table 4.4 presents the results of the Bonferroni comparison. The results differ somewhat across versions. In all versions, TR1 is among the most effective treatments, though it shares this with TR4 on v_1 and with all but TR2 on v_2. Similarly, TR2 is always among the least effective treatments, though sharing this with others on v_3. TR3, TR4, and TR5 are always classified together.

4.6 Discussion

Our results show that, for the object program and versions considered, TR1 consistently requires significantly more time to execute but also achieves the best coverage (in terms of means, with significance on one version) than the other treatments. TR2 is also significantly less costly and effective on two of the three versions than other treatments, and on the third version is in the equivalence class of least costly and least effective treatments. The other three treatments behave somewhat differently across versions and we now explore reasons for some of the observed behaviors.

Across all versions, TR4 works comparatively well in terms of cost and coverage according to Table 4.3 and Table 4.4. It uses a smaller population than TR1, and this allows it to save time. Compared to TR3, it has more test cases which does bring greater diversity into its population, since these test cases improve coverage and help the genetic algorithm find targets sooner. However, it is no more costly than TR3, and this is arguably due to the presence of many unreachable branches. When the genetic algorithm tries to cover a branch there are two stopping criteria: either finding a test case to cover the target or reaching the maximum number of iterations without covering the target. For these unreachable branches TR4 may have a larger population than TR3; however, since the branches are unreachable the additional test cases are not useful but require time to run. Therefore the time consumed counteracts the time that is saved by covering other branches sooner.

The data shows that on v_3, all five treatments improve coverage by only 30%, which leaves a lot of branches uncovered. We checked all the uncovered branches. Other than ten determinably unreachable branches, many of the uncovered branches are new in v_3 and no existing test cases reach them. We believe that this relates to factor $F2$, the characteristics of existing test suites. The existing test suite for v_3

covers a relatively small portion of v_3's code, and thus greater effort is required to augment the test suite for that version. At the same time, this relatively poor test suite offers little diversity in terms of coverage of v_3, and this restricts the genetic algorithm's performance. We believe this is the reason that all treatments achieve lower coverage on v_3 than on the other versions.

The foregoing can also can explain why TR2 behaves similar to treatments TR3, TR4, and TR5 on v_3. After updating all the existing test cases' coverage on v_3, many new methods in that version are still unreachable using the existing test cases. In this situation, TR3 is similar to TR2. Since we do not generate many new test cases, when we use TR4, the few new test cases do not add much diversity.

In v_3, TR1 is most effective but is three times more expensive than other techniques, while on the other two versions TR1 is less than two times more expensive than other techniques. We believe this is because the relatively poor starting test suite leaves many affected methods unreachable. In TR1 for all targets in these methods, we use all existing test cases as the base for the genetic algorithm. Since they never reach these methods, our fitness function treats them all equally (the fitness function measures their performance in the method only). This leaves nothing to guide the evolution. For these branches, TR1 just iterates until it reaches the maximum numbers as explained above, which potentially increases its cost. To solve this problem, in addition to a better starting test suite, we may need to find a fitness function that works interprocedually.

Treatment TR5 did not work as expected. We had conjectured that it would have strengths common to both TR3 and TR4, namely, greater diversity in initial population and smaller size. However, its cost and effectiveness are not significantly different than those of TR3 and TR4. We may require a better technique for selecting test cases to compose the initial population for the genetic algorithm. For example,

genetic algorithms require diversity in the chromosome itself, containing all elements required in the application, instead of simply considering its coverage on the code.

4.7 Conclusions

We described four factors that we believe can influence the cost-effectiveness of test suite augmentation using genetic algorithms, providing reasons for this belief. We presented the results of a case study exploring one of those factors, involving the treatment of existing and newly generated test cases, that we believe could be particularly significant in its effects. Our results show that different treatments of test cases can affect the augmentation task when applying a genetic algorithm for test case generation during augmentation.

At present, the primary implications of this work are for researchers. Our results indicate that when attempting to integrate genetic test case generation algorithms into the test suite augmentation task, it is important to consider the treatment of existing and newly generated test cases, and it may also be important to consider the other factors that we have presented.

Chapter 5

A Framework for Test Suite Augmentation

We have investigated two test case generation techniques separately and identified factors that can affect the cost-effectiveness of test suite augmentation. In this chapter, we first present a framework that incorporates all the factors and then evaluate the impact of three factors, the test reuse approach, the order of targets and test case generation techniques. (Part of this work has appeared in [100].)

5.1 Framework

Figure 5.1 shows our test suite augmentation framework. The framework incorporates several factors mentioned in Section 4.2. The most important factor is the test generation techniques. Other factors are the order of targets, the approach of reusing existing tests and the characteristics of existing test suites. Different values chosen for these factors can affect the test suite augmentation process. In the following, we describe how we use the framework to evaluate several important factors. We begin by

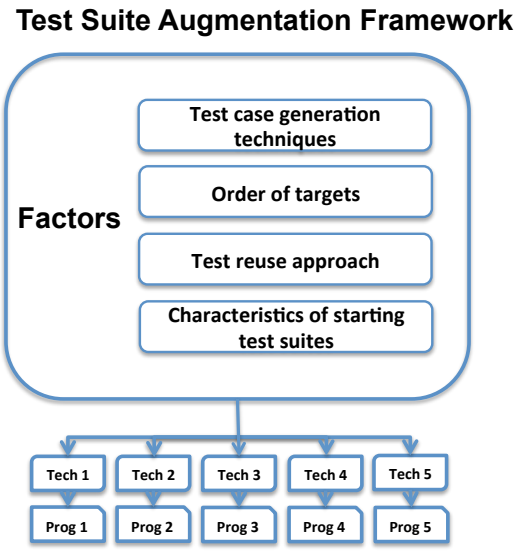


Figure 5.1: Test Suite Augmentation Framework

describing augmentation techniques including augmentation basics, and algorithms used in our experiment, and then we present our experiment results.

5.2 Augmentation Techniques

We now describe the augmentation techniques that we consider. We begin by presenting details relevant to the augmentation task as a whole, and then (Sections 5.2.3 and 5.2.4) we present the two test case generation techniques that we utilize.

5.2.1 Augmentation Basics

5.2.1.1 Coverage Criterion

We are interested in code-based augmentation techniques, and these typically involve specific code coverage criteria. In this work, we continue to focus on code coverage at the level of *branches*; that is, outcomes of predicate statements. While stronger

than statement coverage, branch coverage is more tractable than criteria such as path coverage, and more likely to scale to larger systems.

5.2.1.2 Identifying Affected Elements

As noted in Chapter 1, test suite augmentation consists of two tasks, identifying affected elements and creating test cases that exercise these elements. In Chapters 3 and 4 we presented one approach, based on the use of `Dejavu`, for identifying affected elements. In this work the factors we are studying concern the second of these tasks. Thus, we choose a simple yet practical approach for identifying affected elements. Given program P and its test suite T, and modified version P' of P, to identify affected elements in P' we execute the test cases in T on P' and measure their branch coverage. Any branch in P' that is not covered is an affected element. This approach corresponds to the common "retest-all" regression testing process in which existing test cases are executed on P' first, and then, augmentation is performed where needed.

5.2.1.3 Ordering Affected Elements

Our augmentation techniques operate on lists of affected elements, and as we have stated we believe that the order in which these elements are considered can affect the techniques since test cases covering one element may incidentally cover another. In this work, we investigate the use of a depth-first order of affected elements.

The depth-first order (DFO) of nodes in a control flow graph is the reverse of a postorder traversal of the graph [1][page 660]. In dataflow analysis, considering nodes in DFO causes nodes that are "earlier" in control flow to be considered prior to those that follow them, and can speed up the convergence of the analysis. The same approach can be applied to place branches in depth-first order. We conjecture that by considering affected elements in this order, we may achieve two things. First, we may

be able to speed up the process of generating test cases, because test cases generated for elements that occur earlier in a program's control flow may incidentally cover elements occurring later in control flow, eliminating the need to specifically consider those later elements. Second, we may be able to improve efficiency by considering targets for which path constraints are shorter prior to those for which constraints are longer.

To apply this approach interprocedurally, we calculate DFO in terms of branches in a program's interprocedural control flow graph (ICFG) [65, 83]. We first build the ICFG, then we perform a postorder traversal of that graph recording the nodes visited, and then we reverse the recorded order. Finally, we filter out branches that were not designated as affected to obtain our ordered list of affected elements.

For example, Figure 5.2 shows a simple interprocedural control flow graph. The E and X nodes represent method entry and exit. The numbered nodes represent statements in the program. The nodes with two outgoing edges represent predicate statements, and the labeled edges represent branches out of those predicates and the entry edges of methods (the latter ensures that code in methods containing no branches is also covered). A postorder traversal of the graph visits branches in order b7, b8, b3, b4, b1, b5, b9, b6, b2, b0. The resulting DFO of the branches in the ICFG is thus b0, b2, b6, b9, b5, b1, b4, b3, b8 and b7. Considering branches in this order, we consider b7 only after we have considered b0, b1, b4 and b3. If we begin the augmentation process with just one test case that covers b0, b1, b3, and b8, we first filter out these four covered branches from the ordered list, and then consider the remaining branches in order b2, b6, b9, b5, b4 and b7.

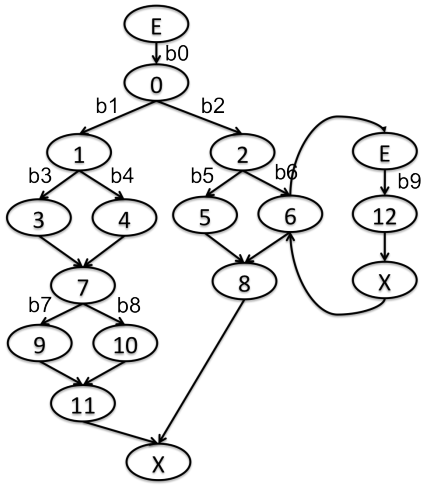


Figure 5.2: Interprocedural control flow graph

5.2.1.4 Test Case Reuse Approach

As mentioned in Section 4.2, existing test cases provide a rich source of data on potential inputs and code reachability, and in the regression testing context, existing test cases are naturally available as a starting point for attempting to cover each affected element. Our results in Section 4.5 have shown that the test case reuse approach is an important factor in the augmentation context that we consider, since it affects both the cost and the effectiveness of the process.

5.2.2 Main Augmentation Algorithm

Algorithm 2 controls the augmentation process, beginning with an initial set of existing test cases, TC, an ordered set of affected elements (target branches), $B_{aff_{ini}}$, and an iteration limit n_{iter}. The algorithm assigns $B_{aff_{ini}}$ to B_{aff} (line 1), which henceforth contains a set of affected elements still needing to be covered. The main loop (lines 3-16) continues until we can no longer increase coverage (which may result due to reaching the iteration limit in the test case generation routines). Within this

loop, for each branch $b_t \in B_{aff}$, if b_t is not covered we call a test case generation algorithm to generate test cases (line 7). If the algorithm successfully generates and returns new test cases this means that at least some new coverage has been achieved in the program (although b_t may or may not have been covered in the process).

Algorithm 2 Main Augmentation Algorithm

Require: set of existing test cases TC, ordered set of affected elements $B_{aff_{ini}}$, and an iteration limit n_{iter}
Ensure: TC augmented with new test cases

```
1: B_aff = B_aff_ini
2: NewCoverage=true;
3: while NewCoverage do
4:     NewCoverage=false
5:     for each b_t ∈ B_aff do
6:         if NotCovered then
7:             NewTests =AUGMENT(TC, B_aff, b_t, n_iter)
8:             if NewTests !=Empty then
9:                 NewCoverage=true
10:            end if
11:            if UseNew then
12:                TC=NewTests ∪ TC
13:            end if
14:        end if
15:    end for
16: end while
```

To accommodate our other factor of concern — the manner in which existing and new test cases are used — we allow for the possibility of adding the newly generated test cases back into our set of available test cases. If the boolean flag $UseNew$ is set to true, this causes the algorithm to combine the newly generated test cases with the original test cases (lines 11-12), and then this newly formed TC is used for the next iteration of our algorithm.

We next describe two different test case generation algorithms that can be invoked at line 7 to generate new test cases.

5.2.3 Genetic Test Suite Augmentation

We have introduced genetic test suite augmentation in Chapter 4. In this section, we present it more formally and describe how the genetic algorithm fits into our main augmentation algorithm.

Algorithm 3 describes the genetic algorithm used in our experiment. The algorithm accepts four parameters: a set of test cases TC, a set of affected elements B_{aff}, an uncovered target branch b_t, and an iteration limit n_{iter}. The algorithm returns a set of new test cases NTC, consisting of all test cases generated that covered any previously uncovered branches in P.

Instead of using random test cases to form an initial population, we take advantage of existing test cases to seed the population. We run this algorithm for each target branch b_t. As the starting population, we select all of the test cases reaching method m_{b_t}, the method that contains b_t; this determines the population size.

Algorithm 3 GENETIC-AUGMENT algorithm

Require: a set of test cases TC, a set of affected elements B_{aff}, an uncovered target branch $b_t \in B_{aff}$, and an iteration limit n_{iter}
Ensure: a set of new test cases NTC
1: $TC_{cur} = TC$ // set of current target test cases
2: $NTC = \emptyset$ // set of new test cases generated
3: TC_{b_t} = {test cases in TC_{cur} that reach method m_{b_t}, the method containing b_t}
4: $Population = TC_{b_t}$
5: $i = 0$
6: **repeat**
7: $Fitness$=CalculateFitness($Population$)
8: $Population$=Select($Population$, $Fitness$)
9: $Population$=Crossover($Population$)
10: $Population$=Mutate($Population$)
11: $i = i + 1$
12: **for** each $tc \in Population$ **do**
13: Execute (tc)
14: **if** *tc covers new branches in* B_{aff} **then**
15: Update B_{aff}
16: $NTC = NTC \cup$ {tc}
17: **end if**
18: **end for**
19: **until** $i \geq n_{iter}$ or b_t *is covered*
20: return NTC

The algorithm repeats for a number of generations (set by the variable n_{iter}) or until b_t is covered. The first step (line 7) is to calculate the fitness of all test cases in the population. Since the fitness of a test case depends on its relationship to the branch we are trying to cover, calculating the fitness requires that we run the test case. (For test cases provided initially we can use coverage information obtained while performing the prior execution of TC, which in our case occurred in conjunction with determining affected elements.) Next a selection is performed (line 8), which orders and chooses the best half of the chromosomes to use in the next step. This population is divided into two halves (retaining the ranking) and the first chromosome in the first half is mated with the first chromosome in the second half and this continues until all have been mated. Next (line 10) a small percentage of the population is mutated, after which all test cases in the current population are executed. If b_t is covered or the iteration limit is met we are finished (line 19), otherwise we iterate.

5.2.4 Concolic Test Suite Augmentation

We have shown how concolic testing can be used in the test suite augmentation context in Chapter 3. In this work, we have improved that concolic algorithm, so we present our improved approach here and also describe how it fits into our main augmentation algorithm.

We use the following notation:

- $CFG_P = (N_P, E_P)$ is a control flow graph of a target program P where N_P is a set of nodes (statements in P) and E_P is a set of edges (branches in P) between N_P.
- A *path condition pc* of a target program P is a conjunction $b_{i_1} \wedge b_{i_2} \wedge ... b_{i_n}$ where $b_{i_1}, ... b_{i_n}$ are edges in E_P and are executed in order. Note that n can be larger

than $|E_P|$, since one branch in a loop body of P may be executed multiple times (i.e., it is possible that $b_{i_k} = b_{i_l}$ for $k \neq l$).

- $DelNeg(pc, j)$ generates a new path condition from a path condition pc by negating a branch occurring at the jth position in pc and removing all subsequent branches. For example, $DelNeg(b_{i_1} \wedge b_{i_2} \wedge b_{i_3}, 2) = b_{i_1} \wedge \neg b_{i_2}$.
- $\overline{b}$ is a paired branch of a branch b (i.e., if b is a `then` branch, $\overline{b}$ is the `else` branch).
- $LastPos(b, pc)$ returns a last position j of a branch b_{i_j} in a path condition pc where $b = b_{i_j}$ (i.e., $\forall j < k \leq n.b_{i_k} \neq b$).
- $Solve(pc)$ returns a test case satisfying the path condition pc if pc is satisfiable; UNSAT otherwise.

Algorithm 4 describes our concolic augmentation algorithm. The algorithm accepts the same four parameters accepted by the genetic algorithm, and returns a set NTC of new test cases. Lines 4-23 detail the main procedure of the algorithm.

Initially, the current target test cases TC_{cur} (from which new test cases are generated) are the old test cases TC (line 1) and NTC is empty (line 2). The start of the main procedure resets the set of newly generated test cases NTC_{cur} (line 4) and selects test cases that can reach $\overline{b_t}$ (the paired branch of b_t) from among the current target test cases TC_{cur} (line 5). If there are no such test cases, the algorithm terminates (lines 6-8). If there are such test cases, the algorithm obtains path conditions by executing the target program with selected test cases (line 9). From each obtained path condition pc, the algorithm generates n_{iter} new path conditions as follows. Suppose the last occurrence of $\overline{b_t}$ is located in the mth branch of pc. Then, the algorithm generates n_{iter} new path conditions (lines 11-19) by negating $b_{i_m}, b_{i_{m-1}}, ..., b_{i_{m-n_{iter}+1}}$ and removing all following branches in pc, respectively (line 13). If a newly generated

Algorithm 4 CONCOLIC−AUGMENT algorithm

Require: a set of test cases TC , a set of affected elements B_{aff}, an uncovered target branch $b_t \in B_{aff}$, and an iteration limit n_{iter}
Ensure: a set of new test cases NTC

```
 1: TC_cur = TC  // a set of the current target test cases
 2: NTC = ∅  // a set of all new test cases generated
 3: repeat
 4:     NTC_cur = ∅ // a set of newly generated test cases in the current execution of line 3 to line 23
 5:     TC_b̄t = { all test cases in TC_cur that reach b̄t }
 6:     if TC_b̄t = ∅ then
 7:         return ∅
 8:         PC_b̄t = { path conditions obtained from executing test cases in TC_b̄t}
 9:         for each pc ∈ PC_b̄t do
10:             for each i = LastPos(b̄t, pc) to i − n_iter+1 do
11:                 if i > 0 then
12:                     pc′ = DelNeg(pc, i)
13:                     tc_new = Solve(pc′)
14:                     if tc_new ≠ UNSAT and tc_new covers uncovered branches in B_aff then
15:                         Update B_aff
16:                         NTC_cur = NTC_cur ∪ {tc_new}
17:                     end if
18:                 end if
19:             end for
20:         end for
21:     end if
22:     TC_cur = NTC_cur
23:     NTC = NTC ∪ NTC_cur
24: until NTC_cur = ∅
25: return NTC
```

path condition pc' has a solution tc_{new} (a new test case) (line 14) and tc_{new} covers uncovered branches in B_{aff} (line 15), B_{aff} is updated to reflect the new status of coverage (line 16), and tc_{new} is added to the set of newly generated test cases NTC_{cur} (line 17).

Note that the iteration limit n_{iter} parameter is a "tuning" parameter that determines how far back in a path condition the augmentation approach will go, and in turn can affect both the efficiency and the effectiveness of the approach.

5.3 Empirical Study 1

Our goal is to investigate augmentation techniques implemented in the context of our framework, focusing on three factors (test case generation algorithm, order of affected elements, and test reuse approach). We thus pose the following research questions.

RQ1: How does the order of consideration of affected elements affect augmentation techniques?

RQ2: How does the manner of use of existing and newly generated test cases affect augmentation techniques?

RQ3: How do genetic and concolic test case generation techniques differ in the augmentation context?

5.3.1 Objects of Analysis

To facilitate technique comparisons, our objects of analysis (programs and test suites) must be suitable for use by both implementations. To select appropriate objects we examined C programs available in the SIR repository [31]. We selected four programs[1] (see Table 5.1), each of which is available with a large "universe" of test cases, representing test cases that could have been created by engineers in practice for these programs to achieve requirements and code coverage [44].[2]

[1]For this study we began by considering the seven Siemens programs, because their size is amenable to study on enormous numbers of test cases. Constraint solvers, however, have limitations. Limitations of the concolic test case generation tool we use (see Section 5.3.3) include difficulties handling non-linear arithmetic and array accesses through symbolic index variables (among others). While these occur in small numbers in the four Siemens programs we selected (as mentioned in Section 6.2) they occur much more frequently in the other three programs. While differences in performance of test case generation techniques across such programs would ultimately be interesting in studying, we considered this a threat to the validity of our attempts to examine the influence of other *factors* on such performance.

[2]Concolic test case generation techniques set limits on the sizes of inputs they generate, and some inputs in the test pools provided with the programs did not conform to reasonable limits. We thus ran several trials with various size limits and selected limits that let us retain at least 60% of

Table 5.1: Objects of Analysis

Program	*Functions*	*LOC*	*Branches*	*Test Cases*
`printtok1`	21	402	174	3052
`printtok2`	20	483	186	3080
`replace`	21	516	206	3174
`tcas`	8	138	76	1608

The programs that we selected do not have actual sequential versions that can be used to model situations in which evolution renders augmentation necessary. We were able, however, to define a process by which a large number of test suites that need augmenting, and that possess a wide range of sizes and levels of coverage adequacy, could be created for the given programs. This lets us model a situation where the given versions have evolved rendering prior test suites inadequate, and require augmentation.

To create such test suites we did the following. First, for each program P we used a greedy algorithm to sample P's associated test universe U, to create test suites that were capable of covering all the branches coverable by test cases in U, and we applied this algorithm 1000 times to P.[3] Next, we measured the minimum size T_{min} and maximum size T_{max} for these suites; this provides estimates of the lower and upper size bounds for coverage-adequate test suites for the programs. Because in practice, programs are often equipped with test suites that are not coverage-adequate, and because we wish to study the effects of augmentation using a wide range of initial test suite sizes and coverage characteristics, we set lower and upper bounds for initial test suites at $T_{min}/2$ and T_{max}, respectively.

Second, we began the test suite construction phase, in which for each test suite to be constructed, we randomly chose a number n such that $T_{min}/2 \leq n \leq T_{max}$,

the inputs in the original test universes. We then removed, from the test universes, test cases that did not conform to these limits. Table 5.1 lists the sizes of the test universes after this reduction.

[3]We chose 1000 because it is a number beyond which (on all programs) further increases fail to lead to changes in observed min and max sizes.

and randomly selected n test cases from U to create a test suite A. We measured the coverage achieved by A on P, and if A was coverage-adequate for P we discarded it. We repeated this step until 100 non-coverage-adequate test suites had been created. Statistics on the sizes and coverages obtained by these test suites are given in Table 5.2.

Table 5.2: Branch Coverage and Sizes of Initial Test Suites

Program	*Branch Coverage*			*Test Suite Size*		
	Avg	Min	Max	Avg	Min	Max
`printtok1`	133.3	110.0	152.0	16.8	9	25
`printtok2`	158.8	129.0	173.0	18.4	8	29
`replace`	165.9	127.0	182.0	17.8	9	28
`tcas`	57.9	30.0	69.0	10.8	5	16

5.3.2 Variables and Measures

Independent Variables Our experiment manipulated three independent variables:

IV1: Order in which affected elements are considered. As orders of affected elements, we use the depth-first order described in Section 5.2, and a baseline approach that orders affected elements randomly.

IV2: Manner in which existing and new test cases are reused. We consider two approaches to reusing test cases; namely, the approach in which a test case generation algorithm attempts to use only existing test cases, and the approach in which it uses existing along with newly generated test cases. From Section 4.5, we know that these two approaches have achieved fairly good coverage and at the same time do not cost much. Also these two approaches are easy to utilize.

IV3: Test case generation technique. We consider two test case generation techniques; namely, the genetic and concolic techniques described in Section 5.2.

Dependent Variables and Measures We wish to measure both the effectiveness and the efficiency of augmentation techniques under each combination of potentially affecting factors. To do this we selected two variables and measures:

DV1: Effectiveness in terms of coverage. The test case augmentation techniques that we consider are intended to work with existing test suites to achieve higher levels of coverage in a modified program P. To measure the effectiveness of our techniques, we track the number of branches in P that can be covered by each augmented test suite.

DV2: Efficiency in terms of time. To track augmentation technique efficiency, for each application of an augmentation technique we measure the *cost* of using the technique in terms of the wall clock time required to apply it. In Section 3.6 and Section 4.3, we used different metrics to measure cost, and the metrics are useful and valid for each technique. However, if we want to make two techniques comparable, we need to find a metric which is valid for them, so we choose wall clock time.

5.3.3 Experiment Setup

Several steps had to be followed to establish the experiment setup needed to conduct our experiment.

Genetic Algorithm Implementation. Even though the core of the genetic algorithm implementation used here is similar to the one used in Section 4.3, we have chosen different object programs. Therefore, we still need to follow some steps to construct our genetic algorithm for the object programs used in this study. First, in our case, each test case is a chromosome where the genes are inputs to the programs, and we customize this for each program. For example, for `printtok1` and `printtok2`, all the characters in the input file form a chromosome and every character is a gene,

while for `tcas`, every integer in the input is a gene in the chromosome. For selection, we select the best half of the population to generate the next generation; we keep the selected chromosomes in the new generation. We rank the chromosomes and divide them into two parts. The first chromosome in the first half is mated with the first chromosome in the second half, and the second chromosome in the first half is mated with the second chromosome in the second half, and so forth.

Second, in crossover, we perform a one point crossover by randomly selecting a position that is between 0 and the number of genes of the smaller chromosome; we then swap everything between chromosomes starting at that position to the end of the chromosome.

Third, our search targets are branches in the program, therefore for our fitness function we use the approach level described in [97]. The approach level is a discrete count measuring how far we were from the predicate controlling the target branch in the CFG when we deviated course; the further away we are from this predicate, the higher the fitness, therefore we are trying to minimize this value. If we reach the predicate leading to our target, the approach level is 0. For different programs, we use different mutation rates: for `printtok1` and `printtok2` we use 0.06, for `replace` we use 0.08 and for `tcas` we use 0.05.[4]

Concolic Algorithm Implementation. The implementation of a concolic algorithm used in Section 3.6 is for Java programs. Since we use C programs in this study, we implemented the concolic test case generation algorithm presented in Section 5.2 based on CREST [15, 26]. CREST transforms a program's source code into an "extended" version in which each original conditional statement with a compound

[4]For our initial implementation, for the sake of simplicity and due to the instrumentation overhead required, we did not combine this with branch distance. We nonetheless achieved good convergence on these programs; still, research suggests that branch distance is an important part of the fitness function [3] and we intend to consider it in the future.

Boolean condition is transformed into multiple conditional statements with atomic conditions without Boolean connectives (i.e., `if(`b_1 `&&` b_2`) f()` is transformed into `if(`b_1`) {if(`b_2`) f()}`).

Extended Programs. To facilitate fair comparisons between concolic and genetic algorithms, we cannot apply the former to extended programs and the latter to non-extended programs. We thus opted to create extended versions of all four programs, and apply both algorithms to those versions.

Iteration Limits. Genetic algorithms iteratively generate test cases, and an iteration limit governs the stopping point for this activity. Similarly, the concolic approach that we use employs an iteration limit that governs the maximum number of path conditions that should be solved to generate useful test cases. These iteration limits can affect both the effectiveness and efficiency of the algorithms. Thus, we cannot run experiments with just one iteration limit per approach, because this would result in a case where our comparisons might reflect iteration limits rather than differences in techniques. For this reason, we chose multiple iteration limits for each test case generation approach, using 1-3-5-7-9 for concolic, and 5-10-15-20-25 for genetic. (The different numbers are due to the different meanings of iterations across the two algorithms, as explained in Sections 5.2.3 and 5.2.4.)

Technique Tuning. Genetic algorithms must be tuned to the programs on which they are to be run. This does not present a problem in a test suite augmentation setting, because tuning can be performed on early system versions, and then the resulting tuned algorithms can be utilized on subsequent versions. For this study, we tuned our genetic algorithms by applying them directly to the extended programs absent any existing suites.

We describe the setup for the parameters in the following. For `printtok1` and `printtok2`, each chromosome is a variable length string containing ASCII characters. For `replace`, each chromosome is split into three parts, each of which contains characters as well. For `tcas`, our chromosome is composed of integers. For the fitness function we use the approach level described in [97]. The approach level is a discrete count that measures how far we were from the predicate controlling the target branch in the CFG when we deviated course during testing. The further we are from this predicate, the higher the fitness, therefore we try to minimize this value. If we reach the predicate leading to our target, the approach level is 0. During the crossover, for all subjects except `replace`, we use a one point crossover by randomly selecting a number between 0 and the length of the shorter chromosome. We then swap everything between chromosomes at that position to the end of the chromosome. For each of the three parts of the `replace` chromosome, we used one point crossover as just described. We chose different mutation rates for the four subjects, 0.05 for `tcas`, 0.06 for `printtok1` and `printtok2`, and 0.08 for `replace`. We chose these based on trial runs performing test case generation (not augmentation) starting with a random population. The mutation pool for `tcas` contains integers mined from predicates in the program, in addition to integers randomly generated from 1800 to -1800. The other subjects use the same pool for mutation; all ASCII characters from 32 to 127, combined with some special characters such as a newline and tab.

5.3.4 Experiment Operation

Given our independent variables, an individual augmentation technique consists of a triple, (G,A,M), where G is one of the two test case generation techniques (Genetic or Concolic), A is one of two affected element orders (Random or Depth-first), and M

is one of the two test case reuse approaches (Existing test cases or New+Existing test cases). An individual *augmentation technique application* consists of an augmentation technique applied at an iteration limit L, and in our case L has five levels.

Our experiment thus employs eight augmentation techniques and 40 augmentation technique applications. Each of these is applied to each of our four programs for each of the 100 test suites that we created for that program. This results in 16,000 augmentation technique applications, for each of which we collect our dependent variables to obtain the data sets needed for our analysis.

Our experiments were run on Linux boxes with Intel Core2duo E8400s at 3.6GHz and with 16GB RAM, running Fedora 9 as OSs. Our processes were the only user processes active on the machines.

5.3.5 Threats to Validity

The primary threat to *external validity* for this study involves the representativeness of our object programs and test suites. We have examined only four relatively small C programs, and the study of other programs, other types of code changes, and other test suites may exhibit different cost-benefit tradeoffs. Furthermore, our programs are chosen to allow application of both genetic and concolic testing, and thus, do not reveal cases in which program characteristics might disable one but not the other of these approaches. A second threat to external validity pertains to our algorithms; we have utilized only one variant of a genetic test case generation algorithm, and one variant of a concolic testing algorithm, and we have applied both to extended versions of the programs, where the genetic approach does not require this and might function differently on the original source code. Further, we have considered only two approaches to handling target branches; other approaches or approaches that handle

sets of target branches rather than single branches (see e.g., [4]) may exhibit different results. Subsequent studies are needed to determine the extent to which our results generalize.

The primary threat to *internal validity* is possible faults in the implementations of the algorithms and tools we use to perform evaluation. We controlled for this threat through extensive functional testing of our implementations. A second threat involves the potential for inconsistent decisions and practices in the implementation of the techniques studied; for example, variation in the efficiency of implementations of techniques could bias data collected.

Where *construct validity* is concerned, there are other metrics that could be pertinent to the effects studied. In particular, our measurements of efficiency consider only technique run-time, and omit costs related to the time spent by engineers employing the approaches. Our time measurements also suffer from the potential biases detailed under internal validity, given the inherent difficulty of obtaining an efficient technique prototype.

Where *conclusion validity* is concerned, our choices of iteration limits for the two test case generation algorithms may have limited our ability to compare the genetic and concolic algorithms fairly in regard to RQ3; it is possible that the addition of additional levels could alter the results of the comparison.

5.3.6 Results and Analysis

As an initial overview of the data, Tables 5.3, 5.4, 5.5 and 5.6 present the average coverage and cost values obtained per program, across all test suites, for each iteration limit, for each combination of order of affected elements and test reuse approach. The coverage is shown as the number of branches covered by augmented test suites and

Table 5.3: Coverage Using DFO Order and Existing Test Cases

	Coverage (number of branches)					Cost (seconds)				
Genetic	*5*	*10*	*15*	*20*	*25*	*5*	*10*	*15*	*20*	*25*
`printtok1`	154.92	155.88	155.98	156.34	156.73	38.66	78.85	117.18	158.39	194.62
`printtok2`	175.89	176.15	176.28	176.35	176.37	26.21	54.93	83.90	113.08	151.67
`replace`	184.55	185.39	186.33	186.67	186.85	65.71	128.39	185.18	247.49	322.67
`tcas`	69.69	70.49	70.73	70.79	70.82	3.06	5.56	8.29	11.08	13.70
Concolic	*1*	*3*	*5*	*7*	*9*	*1*	*3*	*5*	*7*	*9*
`printtok1`	143.97	150.38	151.29	152.19	152.50	1.56	4.35	7.10	9.87	12.62
`printtok2`	165.50	170.84	171.75	172.48	173.16	0.25	0.52	0.80	1.07	1.35
`replace`	176.54	185.61	188.27	189.19	189.58	0.89	2.90	4.95	6.94	9.02
`tcas`	65.12	66.77	68.91	69.51	69.52	0.09	0.19	0.27	0.36	0.44

Table 5.4: Coverage Using DFO Order and Existing plus New Test Cases

	Coverage (number of branches)					Cost (seconds)				
Genetic	*5*	*10*	*15*	*20*	*25*	*5*	*10*	*15*	*20*	*25*
`printtok1`	155.74	156.77	156.96	157.43	157.80	81.22	151.32	239.53	314.83	385.65
`printtok2`	176.37	176.55	176.53	176.57	176.56	54.50	106.33	147.42	229.02	272.56
`replace`	185.21	186.50	186.71	187.31	187.20	92.26	183.31	283.36	365.82	449.64
`tcas`	70.68	70.92	70.86	70.95	70.96	5.12	9.64	14.32	18.65	24.33
Concolic	*1*	*3*	*5*	*7*	*9*	*1*	*3*	*5*	*7*	*9*
`printtok1`	144.16	150.66	151.74	152.49	152.87	1.93	5.73	9.43	13.05	16.61
`printtok2`	165.74	171.30	172.15	172.94	173.70	0.30	0.61	0.93	1.26	1.59
`replace`	176.77	187.54	189.65	190.47	190.75	1.09	3.93	6.65	9.28	11.92
`tcas`	65.63	67.70	70.20	70.85	70.86	0.10	0.20	0.31	0.41	0.51

Table 5.5: Coverage Using Random Order and Existing Test Cases

	Coverage (number of branches)					Cost (seconds)				
Genetic	*5*	*10*	*15*	*20*	*25*	*5*	*10*	*15*	*20*	*25*
`printtok1`	154.93	155.53	155.64	156.19	156.44	39.22	79.07	123.80	164.88	204.94
`printtok2`	175.74	176.34	176.54	176.49	176.37	28.14	59.02	89.97	113.58	153.63
`replace`	184.35	185.94	186.33	186.86	186.99	72.15	130.83	192.52	254.15	311.30
`tcas`	69.80	70.55	70.68	70.76	70.82	2.96	5.41	8.15	11.89	14.50
Concolic	*1*	*3*	*5*	*7*	*9*	*1*	*3*	*5*	*7*	*9*
`printtok1`	143.97	150.38	151.29	152.19	152.50	1.56	4.35	7.09	9.87	12.65
`printtok2`	165.50	170.84	171.75	172.48	173.16	0.25	0.52	0.79	1.07	1.34
`replace`	176.54	185.61	188.27	189.20	189.58	0.88	2.91	4.97	6.98	9.07
`tcas`	65.12	66.77	68.91	69.51	69.52	0.09	0.19	0.27	0.36	0.44

Table 5.6: Coverage Using Random Order and Existing plus New Test Cases

	Coverage (number of branches)					Cost (seconds)				
Genetic	*5*	*10*	*15*	*20*	*25*	*5*	*10*	*15*	*20*	*25*
`printtok1`	155.55	156.20	156.70	157.65	157.31	89.25	171.06	248.59	379.73	428.48
`printtok2`	176.50	176.55	176.54	176.59	176.62	64.89	114.84	165.72	201.23	294.72
`replace`	185.43	186.27	186.95	187.56	187.37	93.53	188.59	285.17	375.81	470.99
`tcas`	70.64	70.86	70.89	70.93	70.97	5.18	9.73	15.09	20.57	25.32
Concolic	*1*	*3*	*5*	*7*	*9*	*1*	*3*	*5*	*7*	*9*
`printtok1`	144.16	150.66	151.68	152.42	152.77	1.92	5.71	9.36	12.92	16.50
`printtok2`	165.74	171.345	172.20	172.94	173.71	0.30	0.61	0.94	1.26	1.59
`replace`	176.79	187.56	189.53	190.55	190.81	1.09	3.96	6.77	9.45	12.23
`tcas`	65.63	67.22	70.20	70.86	70.87	0.10	0.20	0.31	0.42	0.52

Table 5.7: Impact of Order in which Affected Elements are Considered on Coverage and Cost.

	Coverage			
	GDE vs GRE	GDN vs GRN	CDE vs CRE	CDN vs CRN
printtok1	R D D D D	D D D R D	= = = = =	= = D D D
printtok2	D R ***R*** R =	R = R R R	= = = = =	= R R R R
replace	D R = D R	R D R R R	= = = = =	R R D R R
tcas	R R D D =	D D R D R	= = = = =	D ***D*** = = R
	Cost			
printtok1	***D D D D D***	***D D*** D ***D D***	R R R D ***D***	R R R ***R*** R
printtok2	***D D D D D***	***D D D*** R ***D***	D R R R R	R D D R R
replace	***D D D D*** R	***D D D D D***	R ***D D D D***	R D ***D D D***
tcas	R R R ***D D***	***D D D D*** D	= R R D R	D D R D D

the cost is shown in seconds. Each table presents results for concolic and genetic techniques for one combination of the branch order and test case reuse treatments.

We now discuss and analyze this data with respect to our three research questions, in turn.

5.3.6.1 RQ1: Order of Affected Elements

Our first research question pertains to the effects of using different orders of affected elements; in this case, depth-first order versus random. Table 5.7 presents a view of our data that helps us address this question. The table presents results per program, with coverage results in the upper half and cost results in the bottom half. Column headers use mnemonics to indicate techniques: GDE corresponds to (Genetic, DFO, Existing), GDN to (Genetic, DFO, New+Existing), GRE to (Genetic, Random, Existing), GRN to (Genetic, Random, New+Existing), CDE to (Concolic, DFO, Existing), CDN to (Concolic, DFO, New+Existing), CRE to (Concolic, Random, Existing), and CRN to (Concolic, Random, New+Existing). Individual columns correspond to techniques compared; thus, column 2, with header "GDE vs GRE", compares (Genetic, DFO, Existing) to (Genetic, Random, Existing).

Each entry in the table summarizes the differences observed between the two techniques for each of the five iteration limits. "D" indicates that the technique using depth-first order exhibited the better (greater) mean coverage value or better (lesser) cost value, "R" indicates that the technique using random order exhibited the better (greater) mean coverage or better (lesser) cost value, and "=" indicates that techniques exhibited equal mean coverage or cost (through the second decimal place). For example, for `printtok1`, comparing GDE and GRE for coverage, the table contains "R D D D D", indicating that at the lowest iteration limit random order produced greater coverage, and at the other four limits depth first order produced greater coverage. The similar entry for `printtok1` for cost, containing "D D D D D", indicates that at all five iteration limits depth-first order exhibited the lowest cost.

For each pair of techniques compared, for each iteration limit L, we applied a *Wilcoxon-test* [32] to the coverage (and cost) data obtained across all test suites augmented using $\alpha = 0.05$ as the confidence limit, to validate the null hypothesis: there is no significant difference between two orders (DFO and random) in terms of both effectiveness and efficiency when corresponding techniques are compared at iteration limit L. In the table, bold-italicized fonts indicate statistically significant differences. For example, for `printtok2`, comparing GDE and GRE for coverage, the only statistically significant difference between techniques occurred at iteration limit 15. It is these statistical differences that we focus on with respect to our research question.

We begin by considering the results for the genetic algorithm. Where coverage is concerned, no clear advantage resides in either test case order, and results are relatively similar in the instances in which existing, or new and existing, test cases are used. Across all iteration limits and programs, there is only one case in which the two orders result in a statistically significant difference (`printtok2` at iteration

limit 15). Even when considering the non-statistically-significant differences between orders, there is no clear winner.

Where cost results for the genetic algorithm are concerned we see different trends. First, in the GDE vs GRE column there are 16 instances in which order causes statistically significant differences: these include all instances for `printtok1` and `printtok2` and most instances for `replace`. In the GDN vs GRN column there are also 17 instances, again including all instances for `replace` and most instances for `printtok1` and `printtok2`. In all of these instances, DFO is less costly than random.

Turning to the concolic approach, where coverage is concerned, when new and existing test cases are considered, we find only one statistically significant difference between techniques, for `tcas` in the CDN vs CRN case at iteration limit 3. Again, even non-statistically-significant differences show no clear winner. Moreover, when only existing test cases are used, techniques exhibit no differences in coverage at all. Therefore, there are no apparent patterns involving iteration limits or programs to indicate that order potentially influences coverage.

Finally, considering cost results for concolic, unlike the case for the genetic approach, we see only a few statistically significant differences in costs, with five in the CDE vs CRE case and four in the CDN vs CRN case. Seven of these instances are on `replace`, where DFO is less costly than Random, and these are at higher iteration limits, so this may indicate some trend that will emerge as programs become more complex. However, for the other three programs, there is no clear advantage adhering to either Random or DFO orders.

5.3.6.2 RQ2: Use of Existing and New Test Cases

Our second research question pertains to the effects of reusing existing and newly generated test cases. Table 5.8 presents data relevant to this question. The table

Table 5.8: Impact of Test Case Reuse Approaches on Coverage and Cost.

	Coverage			
	GDE vs GDN	GRE vs GRN	CDE vs CDN	CRE vs CRN
printtok1	***N N N N N***	***N N N N N***	***N N N N N***	***N N N N N***
printtok2	***N N N N N***	***N*** N N N ***N***	***N N N N N***	***N N N N N***
replace	***N N*** N ***N N***	***N*** N ***N N N***	***N N N N N***	***N N N N N***
tcas	***N N N N N***	***N N N N N***	***N N N N N***	***N N N N N***
	Cost			
printtok1	***E E E E E***	***E E E E E***	***E E E E E***	***E E E E E***
printtok2	***E E E E E***	***E E E E E***	***E E E E E***	***E E E E E***
replace	***E E E E E***	***E E E E E***	***E E E E E***	***E E E E E***
tcas	***E E E E E***	***E E E E E***	***E E E E E***	***E E E E E***

format is similar to that of Table 5.7, but in keeping with the goal of comparing across test case reuse approaches the differences in terms compared all involve reuse approaches (Existing versus New+Existing). For each pair of techniques compared, for each iteration limit L, we again applied a *Wilcoxon-test* [32] to the coverage (and cost) data obtained across all test suites augmented using $\alpha = 0.05$ as the confidence limit, to validate the null hypothesis: there is no significant difference between two methods of reusing test cases (using existing and using existing and new) in terms of both effectiveness and efficiency when corresponding techniques are compared at iteration limit L. between the two techniques at iteration limit L.

We begin by considering the results for the genetic algorithm. Where coverage is concerned, in all instances, the use of new and existing test cases is superior to reusing only existing test cases, and in most instances the difference is statistically significant. This includes 19 of 20 instances when DFO is used, and 16 of 20 instances in which random order is used.

Where cost results for the genetic algorithm are concerned we observe even stronger effects: in all instances, using existing test cases only is less expensive, and the effect of doing so is statistically significant.

Table 5.9: Comparison of Coverage: Genetic vs Concolic

Program	GDE vs CDE	GDN vs CDN	GRE vs CRE	GRN vs CRN
printtok1	***G***	***G***	***G***	***G***
printtok2	***G***	***G***	***G***	***G***
replace	***C***	***C***	***C***	***C***
tcas	***G***	***G***	***G***	***G***

Turning to the concolic approach, where coverage is concerned, here we see strong evidence that test case reuse matters for coverage, with the use of new test cases always more effective, and in all instances statistically significantly so.

Finally, considering cost results for the concolic approach, we again note statistically significant differences in all instances, again with lower costs adhering to the use of only existing test cases.

5.3.6.3 RQ3: Test Case Generation Algorithm

Our third research question pertains to the effects of using different test case generation algorithms, and we begin by comparing them for effectiveness. One issue to consider in doing this involves inherent differences in the test case generation *algorithms*. In Section 5.3.3 we described the reasoning behind using several iteration limits for each algorithm: we expect concolic and genetic algorithms to respond differently over different limits, and using different limits lets us observe techniques independent of the threat to internal validity that would attend the use of a single iteration limit.

Where comparisons of techniques are concerned, there is no inherent relationship between a given iteration limit for the concolic approach and a given iteration limit for the genetic approach; that is, concolic limits 1, 3, 5, 7, and 9 do not "correspond" in any way to genetic limits 5, 10, 15, 20, and 25. It follows that we cannot validly compare algorithms to each other on a per-iteration-limit basis. Instead, for each

object program P, we locate the iteration limit L_g at which the genetic algorithm operated most effectively on P, and the iteration limit L_c at which the concolic algorithm operated most effectively on P, and we compare the algorithms at these respective optimal iteration limits. To perform these comparisons we again applied a *Wilcoxon-test* [32] to the coverage data at the chosen iteration limits using $\alpha = 0.05$ as the confidence limit, to validate the null hypothesis: there is no significant difference between the two test case generation techniques.

We begin by considering the results for the genetic algorithm. Where coverage is concerned, in all instances, the use of new and existing test cases is superior to reusing only existing test cases.

Table 5.9 presents data relevant to RQ3 with respect to algorithm effectiveness following the analysis procedure just described. The table provides data for each program and for each of the four combinations of affected element ordering and test reuse strategies studied. An individual table entry indicates which technique achieved greater coverage, and bold-italicized fonts indicate instances in which the difference was statistically significant.

As the table shows, on every program but `replace`, the genetic algorithm outperforms the concolic algorithm, in each category in which they were compared. On `replace` the advantage goes to concolic. All differences were statistically significant.

Turning to efficiency, note that this comparison is complicated by the inherent differences in our two implementations. In fact, it is quite difficult to fairly compare techniques for efficiency because their implementations are derived from different sources, and cannot be said to represent "optimal" implementations of the two algorithms. Thus we restrict ourselves to observing efficiency differences in a qualitative fashion. As data presented in Tables 5.3-5.6 shows, costs for the genetic algorithm range from times in the tenths of seconds to times above 400 seconds, while costs

for the concolic algorithm range from times in the tenths of seconds to times near 20 seconds. With our current implementations this represents a very large difference in favor of the concolic approach.

A further issue involves the effects that increasing iteration limits have on the respective algorithms. Here, as remarked earlier, increases in limits seem to correspond to roughly similar increases, proportionally, in costs. This provides some post-hoc justification for our choice of particular iteration limits, in that they seem somewhat comparable in terms of their effects on relative effort.

5.3.7 Discussion and Implications

We now discuss the results presented in the prior section and provide further analysis.

5.3.7.1 Affected Element Order

One might argue that in principle, the order of affected elements is not likely to significantly affect algorithm effectiveness in terms of coverage achieved, because the same elements will ultimately be considered under any order. This is what we observed in the results of our study.

Where efficiency is concerned, in contrast, we do see differences: our results show that DFO can provide savings in costs when using the genetic algorithm. This can be explained by observing that with the genetic algorithm, if we work with higher-level branches first we can incidentally cover additional branches. Also, test cases that cover branches higher in dependency chains could have inputs that are close to those used to reach lower branches, thereby seeding the population with inputs that help the algorithm cover those more quickly.

With the concolic algorithm, in contrast, cost saving results are mixed. We suspect this is because test cases generated to cover a given branch b_t (lines 11-19 of Algorithm 4) may not cover other uncovered branches unless these uncovered branches share a common ancestor branch at a short distance from b_t (less than n_{iter}) in an execution tree. In such cases, the ordering of affected elements is not likely to affect cost.

All things considered, based on our results we can argue that DFO has the potential to be more efficient than random ordering when using genetic algorithms, since we observed this in almost all cases considered. There seems to be no clear benefit to using either order, however, where the concolic approach is concerned. Still, these results do not preclude finding some other orderings that are more predictably cost-effective for that approach.

5.3.7.2 Test Case Reuse Approach

Our results show that the use of new test cases in addition to existing test cases always significantly increased the cost of test generation by both techniques. This result can be explained by the correlation between technique effort and the number of test cases used to seed the technique. Having additional test cases affects the population size for the genetic algorithm, while the concolic technique must consider each test case supplied to it. Note that in Section 4.5, these two approaches have similar costs when using the genetic algorithm, because for the subject program `NanoXML` not many new test cases are generated compared to the size of existing test cases. We needed to run more test cases but at the same time the new test cases brought diversity which led to fewer iterations. Therefore, these two approaches did not exhibit much difference in terms of cost in that case.

The use of new test cases also significantly increased test generation technique effectiveness in all cases in which the concolic approach was used, and in most cases where the genetic approach was used. The difference across techniques can be explained as follows. With the genetic algorithm, having additional test cases to work with can increase population diversity and improve the chances that crossover will generate chromosomes that cover previously uncovered branches; however, changes due to the increase might not be substantial when just a few test cases are added to those that have been used previously. This is the same as we see in Section 4.5, two approaches have achieved similar coverage when the genetic algorithm is used for `NanoXML`. The concolic approach, in contrast, utilizes each new test case independently and can potentially gain from each as such.

If these results generalize we have an important cost-benefit tradeoff. With both techniques there is a potential payoff for incurring the additional costs involved in reusing test cases, and this effect is greater for the concolic technique than for the genetic technique. In practice, whether any effectiveness gain is worth the additional cost must be assessed relative to the actual costs of generating test cases versus the actual benefits of obtaining better coverage on the particular systems being verified. Such assessments, however, are quite viable in the context of software evolution, where systems are expected to be re-tested many times, and long-term cost-benefit gains make assessments more worthwhile.

5.3.7.3 Test Case Generation Techniques

As mentioned in our discussion of threats to validity, we are working with particular variants and implementations of test case generation algorithms. Genetic algorithms require tuning in terms of fitness function, selection method, and mutation mechanism. We have tuned our algorithms on our object versions, independent of existing

test suites, and in practice this approach could be used on early versions of systems in order to support regression testing of later versions. Still, alternative tunings might have allowed the genetic algorithms to perform better. Similarly, we have used just one concolic algorithm and implementation, and alternative algorithms or implementations might allow it to perform better. Finally, as we have also mentioned, efficiency differences between the implementations cannot be compared in any rigorously quantitative sense.

These cautions noted, in our experiment, concolic and genetic test case generation techniques did perform statistically significantly differently. The genetic algorithm exhibited greater effectiveness than the concolic algorithm on `printtok1`, `printtok2`, and `tcas` under all combinations of other factors. It appears that the genetic algorithm is more costly (potentially by two orders of magnitude) than the concolic algorithm in doing this, although again this comparison must be made cautiously due to the foregoing factors. These observations do prompt us, however, to further explore the reasons for differences. We postpone discussion of that exploration to Section 5.5, however, when we can present it together with further input from the results of our second study.

5.3.7.4 Iteration Limits

We did not consider iteration limit to be an independent variable; rather, we blocked our analyses per iteration limit level, since this is our stopping criterion. We did examine our data, however, to assess iteration limit effects.

First, there does appear to be an increasing trend in coverage values as iteration limits increase. Beginning with the genetic algorithm, and considering the 16 cases in which limits increase (i.e., four increases per program, progressing from 5 to 10, 10 to 15, 15 to 20, and 20 to 25), coverage values for GDE increase as limits increase in all 16

cases, coverage values for GRE increase as limits increase in 14 of 16 cases, coverage values for GDN increase as limits increase in all 16 cases, and coverage values for GRN increase as limits increase in all 16 cases. The coverage increases, however, are small overall — never more than two — and only 24 of 64 are statistically significant, which indicates that our genetic algorithm is converging.

Iteration trends occur for the concolic algorithm as well, with values generally increasing by small amounts in all 64 cases. In this case, all of these increases are statistically significant, suggesting that iteration plays a more measurable role for the concolic approach than for the genetic approach, and that further increases may provide opportunities to increase effectiveness.

Where algorithm efficiency is concerned iteration limits have larger effects. For the genetic algorithm, costs differ across iteration limits by relatively substantial amounts (i.e., by factors ranging from four to six from iteration limits 5 to 25). Where the concolic algorithm is concerned we also see increases in costs as iteration limits increase. The increases are smaller numerically than those observed with the genetic algorithm, but they are similar in terms of the factors involved (i.e., they increase by factors ranging from five to ten from iteration limits 1 to 9).

5.3.7.5 Initial Test Suite Characteristics

Test suites can differ in terms of size, composition, and coverage achieved. Such differences in test suite characteristics could potentially affect augmentation processes. For example, the extent to which an existing test suite achieves coverage prior to modifications can affect the number and locations of coverage elements that must be targeted by augmentation. Furthermore, test suite characteristics can impact the size and diversity of the starting populations utilized by test case generation techniques.

For these reasons, we chose to additionally examine our results in terms of four different fixed levels of coverage achieved by test suites. To do this, for each object program, we considered the total branch coverage achieved by the 100 test suites for that program, ranked them in terms of coverage, and partitioned them into four equal size quartiles, denoted Q1, Q2, Q3, and Q4, respectively, where Q1 contains the 25 test suites achieving the lowest levels of coverage, Q2 contains the 25 suites achieving the next highest levels, and so forth. We then conducted the same statistical tests on the resulting data that were conducted in examining our first and second research questions, on a per quartile basis.

In three of the resulting comparisons, namely, (1) the impact of test order on coverage, (2) the impact of test order on cost, and (3) the impact of test reuse on cost, we observed no differences in results across quartiles. That is, test suite characteristics did not impact the associated effects. In one of the resulting comparisons, however, namely, (4) the impact of test reuse on coverage, we did observe effects.

Table 5.10 presents results relevant to this assessment. The table is similar to Table 5.8, but in this case, we provide separate results per program for each of the four quartiles in the rows labeled "Q1", "Q2", "Q3" and "Q4". Where Table 5.8 revealed statistically significant coverage differences between approaches using existing test cases and approaches using existing plus new test cases in all but five cases, the per-quartile assessment exhibits many more cases in which differences are not statistically significant. This may be caused, in part, by the fact that these comparisons employ data sets which, being smaller, do not provide enough data to provide sufficient power to statistical tests. There does appear to be a tendency, however, for lower quartiles to exhibit significance more frequently than higher quartiles. In other words, the coverage benefits of using new test cases in addition to existing ones may dissipate as the degree of coverage achieved by initial test suites increases. Further, on the most

Table 5.10: Impact of Test Reuse in Quartiles

		Coverage			
		GDE vs GRN	GRE vs GRN	CDE vs CDN	CRE vs CRN
printtok1	Q1	N ***N*** ***N*** ***N*** ***N***	***N*** N ***N*** ***N*** ***N***	N ***N*** ***N*** ***N*** ***N***	N ***N*** ***N*** ***N*** ***N***
	Q2	***N*** N N N N	N N ***N*** ***N*** N	N N N N N	N N N N N
	Q3	N N ***N*** ***N*** N	N N N ***N*** N	N N N N N	N N N N N
	Q4	***N*** N N ***N*** E	N N N N E	= N N N N	= N N N N
printtok2	Q1	***N*** ***N*** N N ***N***	***N*** ***N*** N ***N*** ***N***	N ***N*** ***N*** ***N*** ***N***	N ***N*** ***N*** ***N*** ***N***
	Q2	***N*** N N N N	***N*** N N N N	N ***N*** ***N*** ***N*** ***N***	N ***N*** ***N*** ***N*** ***N***
	Q3	N N N N N	N N N N N	N = N = N	N = = = =
	Q4	N N E E E	N E E E E	N N = N N	N N N N N
replace	Q1	N N N N N	***N*** N ***N*** ***N*** ***N***	***N*** ***N*** ***N*** ***N*** ***N***	***N*** ***N*** ***N*** ***N*** ***N***
	Q2	N ***N*** N N ***N***	***N*** N N N N	N ***N*** ***N*** ***N*** ***N***	N ***N*** ***N*** ***N*** ***N***
	Q3	N N N N N	N N N N N	N ***N*** ***N*** ***N*** ***N***	N ***N*** ***N*** ***N*** ***N***
	Q4	N ***N*** N N N	***N*** N N ***N*** N	N N N ***N*** ***N***	= N ***N*** ***N*** ***N***
tcas	Q1	***N*** ***N*** ***N*** ***N*** ***N***	***N*** ***N*** ***N*** N ***N***	***N*** ***N*** ***N*** ***N*** ***N***	***N*** ***N*** ***N*** ***N*** ***N***
	Q2	***N*** N N N N	***N*** N N N N	N ***N*** ***N*** ***N*** ***N***	N N ***N*** ***N*** ***N***
	Q3	N ***N*** N N N	N N N N N	= N N = =	= = = N N
	Q4	N = N = =	N N N N N	= = = = =	= = = = =

complex of the programs, `replace`, the efficacy of using new test cases dissipates more slowly for the concolic algorithm than for the genetic algorithm. This may indicate the potential for the algorithms to be differently influenced by initial test suite characteristics on programs of different characteristics, a suggestion that we return to in Section 5.5 following presentation of the results of our second study.

5.3.7.6 The Benefits of Augmentation

In Section 1, we conjectured that augmentation techniques working with existing test suites can perform better than augmentation techniques working without existing suites. To further consider this claim, we applied the concolic testing tool CREST from scratch on our programs, working without the benefit of test cases (the approach under which these algorithms have been traditionally been studied to date).[5]

[5]An equivalent investigation of the genetic algorithm would be complicated by the fact that that the algorithm does require test cases to begin with, and the only relevant approach to compare

Table 5.11 displays the results, listing the cost in seconds and the final coverage reached in branches on each program, per iteration level IL (left column). Entries of the form '-' under `tcas` indicate cases where larger iteration limits are not needed. Comparing results with those for augmentation techniques reveals substantially poorer coverage on all programs but `tcas`, at costs that are relatively similar. The benefit of allowing the concolic approach to reuse test cases in the augmentation task is quite clear.

Table 5.11: Results of Concolic Testing From Scratch

	`printtok1`		`printtok2`		`replace`		`tcas`	
IL	Cost	Cov.	Cost	Cov.	Cost	Cov.	Cost	Cov.
1	1.4	111	0.5	87	1.5	56	0.2	71
3	3.4	111	0.9	92	2.5	78	-	-
5	5.3	111	1.3	101	3.8	78	-	-
7	7.8	111	1.9	110	6.2	78	-	-
9	9.6	111	2.3	115	8.8	78	-	-

5.4 Empirical Study 2

The results of Study 1 suggest that affected element order and test case reuse approach can indeed have different impacts in the context of different augmentation techniques, and that the two underlying test case generation techniques that we consider have different strengths on different programs. However, as we discussed in Section 5.3.5, the programs we used in that study are relatively small and simple. We wish to see whether these results generalize to larger, more complex programs. Thus, we replicated Study 1 on a considerably more complex open-source program, `grep`, for which a sequence of six versions was available.

against would be one in which those test cases were randomly generated. Randomly generating applicable test cases for the object programs is a non-trivial task, and the process of then applying the genetic algorithm to generate test cases with these is expensive. We judge the knowledge that might be gained from such an attempt to not be worth this effort. Thus we have not performed this comparison with respect to the genetic algorithm.

For this study, we again consider the same research questions considered in Study 1, and for completeness we repeat these here, designated RQ1′, RQ2′ and RQ3′ in recognition of the different experimental context being considered.

RQ1′: How does the order of consideration of affected elements affect augmentation techniques?

RQ2′: How does the use of existing and newly generated test cases affect augmentation techniques?

RQ3′: How do genetic and concolic test case generation techniques differ in the augmentation context?

As noted, this study utilizes the `grep` program provided in the SIR [31]. The `grep` program is a command-line text-search utility originally written for Unix. It searches files or standard input globally for lines matching a given regular expression, and prints the lines to the program's standard output. It contains about 10,000 lines of C code. As mentioned above, `grep` is available with six sequential versions. However, the program does not have an enormous test universe of test cases offering complete coverage of the code; rather, it comes with a single test suite containing 792 test cases. We augment this test suite for each of the five versions after the base version. Table 5.12 provides details on the numbers of the branches for each of these subsequent versions, as well as the coverage achieved on each of those versions by the test suite prior to augmentation.

Table 5.12: Initial Coverage Information for `grep`

Version	Total number of branches	Initial coverage
V1	3934	2151
V2	4146	2245
V3	4234	2271
V4	4262	2284
V5	4264	2284

This study utilizes the same variables and measures as Study 1. It also possesses the same threats to validity as Study 1 with the exception of those specifically addressed in this study (size and representativeness of the object programs). We thus do not repeat discussion of these here. Instead, we describe only the differences between this study and Study 1. We then present data and analysis and discussion of results.

5.4.1 Experiment Setup

The `grep` program is quite different from the programs used in Study 1, in terms of size and initial test suite, and this required us to make some adjustments to the experiment process. First, one test case for `grep` has three parts: option, pattern and file. The option part includes command-line arguments that change many of the program's behaviors. For example, the option flag "-i" enables case-insensitive search (ignore case). The pattern part is the regular expression that the user wishes to find in files. Therefore, both option and pattern parts are strings. The file part specifies where the user wishes to search for the pattern and is usually a path. We did not limit option and pattern lengths as we did for the programs in our first study, since both lengths in the existing test cases are less than 30 which does not cause any problem for our test case generation techniques. For the file parameter, the existing test cases make use of five different files of which the largest contains 10,965 lines.

A second set of changes involved the settings used for the genetic algorithm, the first of which involves the test suite reuse approach. With the genetic algorithm, if all test cases are used to form the initial population for a target, the test case generation process may take an inordinately long (and practically unreasonable) amount of time. In such cases, it is common to use a subset of the population [85]. To determine a reasonable subset size to use, we ran trials on the base version (V0) using initial

sizes 25, 50, 100, 150, and 792. They covered 540, 613, 577, 634 and 623 branches separately in 5.6, 4.4, 4.7, 6.1 and 6.7 days, respectively. We determined that size 50 presented the best ratio of coverage to efficiency when applied to version V0. For a target, if there are more than 50 test cases reaching the method that contains it, we select the 50 fittest test cases as the initial population when we just use existing test cases. When we consider existing plus new test cases in the genetic technique, in addition to the 50 we chose, we add the newly generated test cases that reach the method containing the target into the initial population for the target. In this case, the existing plus new approach has more test cases to use for each target. In our experiment runs, we use that population size on subsequent versions (and we do not use data from V0). Note that this approach is practically reasonable in the context of evolving software, because engineers can tune a testing approach on an initial version and then use that tuned approach on subsequent versions.

We also altered the genetic algorithm process somewhat for use on `grep`. Every character in the option and pattern arguments to the program is treated as a gene in the chromosome. The whole file is also a gene – this is different from the approach used for the smaller programs, but is necessary since the files used for `grep` are very large and if we consider mutating the files, it would be difficult for the concolic technique to do so. To be fair, however, in both techniques we treat the file as a manipulable input. For the genetic approach, this means that the file is treated as a gene, and we can switch the file in the chromosome with other files in the file pool. We used the same strategies for fitness function, selection and crossover as the smaller programs. We use a mutation rate of 0.05.

Finally, because the test case generation process takes much longer on `grep` than on the programs used in Study 1, rather than use five different iteration levels we used just one. To make an informed decision as to an iteration level, we applied the

following process to version V0. (Again, this is a process that engineers could apply on an initial version in practice in order to tune an approach for use on subsequent versions). We reasoned that an iteration level should be chosen based on the tradeoff it presents with respect to costs and benefits. We used the following formula to examine these tradeoffs:

$$\frac{(C(I_{k+3}) - C(I_k))/C(I_k)}{(T(I_{k+3}) - T(I_k))} \tag{5.1}$$

Here, $C(I_k)$ is the number of covered branches in the target program at the kth iteration level and $T(I_k)$ is the execution time required to augment the test suite at the kth iteration level, measured in hours for the concolic algorithm and days for the genetic algorithm. The formula calculates the cost-benefit increase across the subsequent three versions to avoid local minima or maxima that may exist in calculating it across a single iteration level.

To choose an iteration level for the genetic and concolic algorithms, we applied each algorithm to version V0 of `grep` at increasingly higher iteration levels, applying the equation to each level as the data required for that level (from applications at subsequent levels) became available. We continued this process until the difference in ratios between two successive iterations fell below 0.01. In other words, after this point, it takes more than one hour for the concolic approach and 24 hours for the genetic approach to increase coverage by 1% when we run the experiment at the third higher level. This process ultimately led us to choose iteration level 11 for the concolic approach, and iteration level 15 for the genetic approach.

Having selected the foregoing parameters we proceeded with the experiment runs, in which we applied each augmentation technique to each of the five subsequent versions of `grep`. Because the algorithms do include non-deterministic behavior, we

Table 5.13: Coverage and Cost Data for `grep`, per Version and Technique

Coverage													
		V1		V2		V3		V4		V5		Avg	
		D	R	D	R	D	R	D	R	D	R	D	R
GA	E	584	575	557	592	607	590	594	636	656	593	599.6	597.2
	N	587	570	584	615	594	583	631	640	635	621	606.2	605.8
CT	E	390	390	405	405	423	423	448	448	448	448	422.8	422.8
	N	604	622	621	621	644	626	668	676	668	676	641.0	644.2
Cost (hours)													
		V1		V2		V3		V4		V5		Avg	
		D	R	D	R	D	R	D	R	D	R	D	R
GA	E	93.6	93.6	88.8	98.4	110.4	84.0	79.2	96.0	91.2	88.8	92.6	92.2
	N	160.80	163.2	146.4	184.8	208.8	182.4	132.0	163.2	180.0	213.6	165.6	181.4
CT	E	7.6	8.2	11.6	12.3	13.9	13.7	12.5	12.3	12.4	12.7	11.6	11.8
	N	28.3h	28.2h	40.4	41.1	46.3	43.2	34.9	39.7	35.7	39.9	37.1	38.4

applied each algorithm three times for each version. We thus obtained 60 data points on the program for each algorithm, in total (i.e., 2 test reuse approaches $*$ 2 target orders $*$ 5 versions $*$ 3 runs).

5.4.2 Data and Analysis

Table 5.13 presents the data gathered for `grep`. The upper half of the table provides coverage data and the lower half provides cost data. In each half of the table, the first two rows present the data for the genetic algorithm and the last two rows present the data for the concolic algorithm. Coverage data is presented in terms of the numbers of previously uncovered branches (total number of branches - initial coverage in Table 5.12) that the approach covered. Cost data is presented in hours for both algorithms. For each version and algorithm, four numbers are shown, corresponding to measurements gathered for the four combinations of affected element orders ("D" and "R") and test case reuse approaches ("E" and "N"). Each cell in the table shows the mean value across the three runs performed for the given combination.

Where coverage data is concerned, for the genetic algorithm, on average across all versions, using DFO and existing test cases covered 599.6 new branches while using

existing plus new test cases added 606.2, just a 1.1% increase. Using random ordering, existing test cases covered 597.2 branches while existing plus new added 605.8, a 1.4% increase. Results varied across versions, however, with the use of existing plus new cases outperforming the use of just existing test cases on only three of five versions for each ordering (V1, V2, and V4 for DFO; V2, V4, and V5, for random). For DFO, the largest increase was 6.2% on V4 and the smallest was -3.2% on V5, while for random orders the largest increase was 4.7% on V5 and the smallest was -1.2% on V3. Differences associated with test case orders were also small on average (less than one branch), with no test case order being predominantly better.

For the concolic algorithm, differences associated with different test case reuse methods are greater. On average across all versions, using DFO and existing test cases covered 422.8 new branches while using DFO and existing plus new test cases added 641.0, a 51.6% increase. Using random orders and existing test cases covered 422.8 new branches while using random orders and existing plus new test cases covered 644.2 branches, a 52.4% increase. Improvements in results were consistent across versions and fell within relatively similar ranges, with the largest increase being 54.9% on V1 and the smallest 49.1% on V4 and V5 for DFO, and the largest increase being 59.5% on V1 and the smallest 48% on V3 for random orders. Differences associated with test case orders, however, continued to be small or none on average.

Where cost data is concerned, for the genetic algorithm, on average across all versions, using DFO and existing test cases cost 92.6 hours while using existing plus new test cases cost 165.6 hours, a 78.8% increase. Using random ordering and existing test cases cost 9221 hours while using random ordering and existing plus new test cases cost 181.4 hours, a 96.9% increase. Results were consistent in direction across versions, varying in magnitude from 97.4% on V5 to 64.9% on V2 for DFO and from 117.1% on V3 to 70.0% on V4 for random orders. Differences between test case

orders, in contrast, were less consistent across versions. When using just existing test cases there was no average difference (and no clear winner) between DFO and random orders. When using existing plus new test cases there was a 15.84 hours average difference favoring DFO, with DFO outperforming random on all but V3.

For concolic testing, on average across all versions, using DFO and existing test cases cost 11.6 hours, while using DFO and existing plus new test cases cost 37.1 hours, a 220.0% increase. Using random ordering and existing test cases cost 11.8 hours while using random ordering and existing plus new test cases cost 38.4 hours, a 224.5% increase. Results were again consistent in direction across versions, varying in magnitude from 272.4% on V1 to 179.2% on V4 for DFO and from 243.9% on V1 to 214.2% on V5 for random. Differences between test case orders, however, were inconsistent across versions and relatively small on average (e.g., 1.3 hours when using existing plus new test cases and 0.2 hours when using just existing test cases).

Finally, where comparisons of the test case generation algorithms are concerned, we note that when using just existing test cases, the genetic algorithm attains substantially higher coverage (from 37.5% to 49.7%) across the five versions than the concolic algorithm. When using existing plus new test cases, however, the concolic algorithm outperforms the genetic algorithm, from amounts ranging from 1.0% to 9.1% across versions. Also, in all cases, the concolic algorithm is substantially faster than the genetic algorithm.

5.4.3 Discussion and Implications

We begin by summarizing the results for `grep`, as follows:

- DFO and random orders had little effect on coverage differences, for both the genetic and concolic approaches and under both test case reuse approaches.

- DFO and random orders had inconsistent and varying effects on the costs of genetic and concolic approaches in general. The one combination of treatments in which order could be seen to have an impact occurred when using both existing and new test cases with the genetic algorithm.

- The concolic algorithm benefitted substantially in terms of coverage when using existing plus new test cases rather than just existing test cases, and this benefit occurred for both test case orders. The genetic algorithm benefitted only mildly and less consistently.

- In all cases, using existing plus new test cases added substantial costs to the test case generation process.

- The concolic approach outperformed the genetic approach in terms of coverage when using existing plus new test cases, while the genetic approach was better when using just existing test cases.

The foregoing results are similar in their overall trends to those seen in Study 1, with the exception of the last. We believe that the differences observed for the concolic approach are primarily due to the fact that initial test suites achieve much lower levels of coverage on `grep` than did the initial test suites used in Study 1; thus, new test cases that are generated have larger potential to lead to additional coverage simply due to the fact that more targets are available. The fact that the genetic algorithm does not achieve a similar level of improvement, on the other hand, is likely due to the fact that the newly generated test cases do not provide better power than the existing test cases, which is consistent with what we observed on the smaller programs.

The data also prompts additional observations. On this much larger program, the costs associated with the test generation task are much greater than on the

Table 5.14: Branch Coverage Differences – Smaller Programs

	DFO/EXISTING				DFO/NEW			
Program	GA	CT	GA ∩ CT	GA∪ CT	GA	CT	GA ∩ CT	GA∪ CT
printtok1	5.25	0.27	152.23	157.75	5.65	0.11	152.76	158.52
printtok2	4.39	1.17	171.90	177.54	4.02	1.13	172.50	177.71
replace	4.23	5.70	183.90	193.84	3.35	5.29	185.46	194.09
tcas	1.37	0.10	69.44	70.91	0.13	0.01	70.86	71.00
	RAND/EXISTING				RAND/NEW			
Program	GA	CT	GA ∩ CT	GA∪ CT	GA	CT	GA ∩ CT	GA∪ CT
printtok1	5.32	0.15	152.35	157.82	5.21	0.13	152.64	157.98
printtok2	4.35	1.14	172.02	177.51	4.02	1.08	172.63	177.7
replace	4.28	5.67	183.93	193.88	3.28	5.22	185.58	194.08
tcas	1.39	0.04	69.50	70.93	0.13	0.02	70.80	71.00

smaller programs, measuring in hours for the concolic approach and days for the genetic approach. There is still a cost-benefit tradeoff involved, for both techniques, in choosing to use existing or existing plus new test cases, but the benefit to cost ratio for the genetic algorithm is much smaller here than with the first four programs, and the benefit to cost ratio for the concolic algorithm is much larger. Thus, the cases in which using existing plus new test cases would be worthwhile are likely to occur much less often for the genetic algorithm than for the concolic algorithm.

5.5 Additional Analysis and Implications

Our two studies revealed overall performance differences between augmentation techniques utilizing different test case generation algorithms and suggested several reasons for those differences. To obtain further insights we analyzed the differences in coverage results between the techniques in greater detail, and we present the results of that analysis here.

5.5.1 Overall Comparison

We begin by considering results of Study 1. Table 5.14 shows the differences in branch coverage achieved by concolic and genetic test case generation techniques in that study. The table shows, for each of the four programs, for each of the four techniques applied, the average numbers of branches across 100 test suites such that (1) (GA) test suites generated by the genetic algorithm covered that branch while no test suites generated by the concolic algorithm covered it; (2) (CT) test suites generated by the concolic algorithm covered that branch while no test suites generated by the genetic algorithm covered it; (3) (GA $\int$ CT) each algorithm succeeded in generating at least one test suite that covered the branch. (4) (GA $\cup$ CT) one or both algorithms succeeded in generating at least one test suite that covered the branch. As the table shows, for all techniques and programs, each of the two algorithms (concolic and genetic) is able to cover at least some branches that cannot be covered by the other algorithm. On `tcas`, the smallest of the four programs, the numbers are small (between 0.01 and 1.39 branches). On the other three programs, larger ranges of branch coverage differences occur, with the genetic algorithm accounting for more differences on `printtok1` and `printtok2`, and the concolic algorithm accounting for more on `replace`.

We provide further details on two of the programs in Figure 5.3. The figure focuses on the two object programs on which the techniques exhibited the greatest range of differences, `replace` and `printtok1`, and on the case in which DFO and new plus existing test cases are utilized. For each of these two cases the figure displays a graph. The x-axes in these graphs correspond to branches (branch identifier numbers) in the program. The y-axes indicate the numbers of test suites (from among the 100 suites used) in which each branch was covered, with the bar extending upward from the

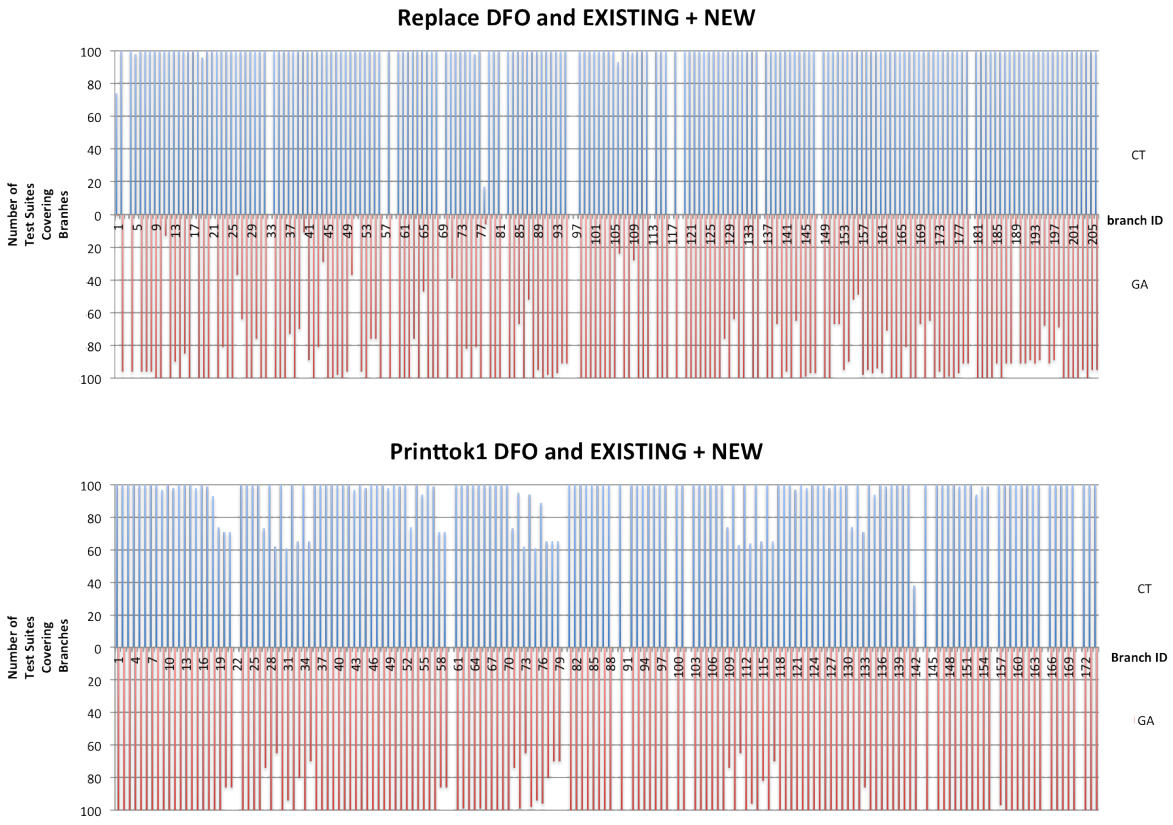


Figure 5.3: Comparison of branch coverage behaviors for concolic and genetic algorithms on two representative cases.

line labeled "0" showing results for the concolic algorithm, and the bar extending downward from that line showing coverage for the genetic algorithm.

In the case of `replace`, we see that a relatively small number of branches (13 to be precise) are not covered by any of the 100 test suites, for either technique. A much larger number (101 to be precise) are covered by all 100 test suites, for both techniques. The remaining branches are missed for at least some test suites by one or both algorithms. For the concolic algorithm, only a few such branches (7 to be precise) are missed by between 1 and 99 test suites, while for the genetic algorithm far more (86 to be precise) are missed by between 1 and 99 test suites. In other words,

Table 5.15: Branch Coverage Differences – `grep`

	DFO/EXISTING				DFO/NEW			
Version	GA	CT	GA ∩ CT	GA∪ CT	GA	CT	GA ∩ CT	GA∪ CT
v1	541	236	154	931	375	302	302	979
v2	553	293	112	958	403	334	287	1024
v3	531	270	153	954	383	349	295	1027
v4	546	293	155	994	394	341	327	1062
v5	565	285	163	1013	382	334	334	1050
	RAND/EXISTING				RAND/NEW			
Version	GA	CT	GA ∩ CT	GA∪ CT	GA	CT	GA ∩ CT	GA∪ CT
v1	521	243	147	911	342	323	299	964
v2	559	278	127	964	411	355	266	1032
v3	559	272	151	982	398	356	270	1024
v4	548	276	172	996	374	319	357	1050
v5	518	292	156	966	369	343	333	1045

the concolic algorithm achieves much higher rates of success in covering branches than the genetic algorithm on a large number of branches.

The `printok1` object presents a different picture. Here again, several branches are left uncovered by both techniques, but the genetic technique is 100% successful on a few more branches (22 to be precise) than the concolic approach, and the genetic approach has somewhat higher success at covering branches that are not always covered. The differences between the two algorithms on this object program, however, are not as large as those seen on `replace`.

We next turn our attention to Study 2 and `grep`. In this case, because the executions per version involve independent runs of techniques, we cannot compare differences per run; instead we choose a different approach. Table 5.15 shows, for each of the five versions of `grep`, for each of the four techniques applied, the numbers of branches such that (1) (GA) at least one of the three test suites generated by the genetic algorithm covered that branch while no test suites generated by the concolic algorithm covered it; (2) (CT) at least one of the three test suites generated by the concolic algorithm covered that branch while no test suites generated by the genetic

algorithm covered it; (3) (GA $\int$ CT) each algorithm succeeded in generating at least one test suite that covered the branch. (4) (GA $\int$ CT) one or both algorithms succeeded in generating at least one test suite that covered the branch.

As the table shows, on the larger `grep` object, the genetic and concolic algorithms exhibited large disparities in their abilities to cover specific branches. For example, for the scenario in which DFO and existing test cases only were used, both algorithms jointly were able to cover between 112 and 163 branches across the five versions, but the numbers of branches covered only by the concolic algorithm exceeds these numbers by factors of between 0.5 and 0.8, and the number of branches covered only by the genetic algorithm exceeds these numbers by a factor of between 2.5 and 3.9. Similar trends (though with different increase factors) are seen in the other scenarios. Clearly, in this more complex program, the differences in coverage abilities of the two algorithms are larger than those seen on the smaller, less complex programs.

Table 5.16 considers these differences further for the case in which DFO and existing plus new test cases are used. For each version of `grep`, the table displays data about just those branches that are covered only by genetic testing, or only by concolic testing. The data denotes the numbers of times these branches were covered by only one of the test suites generated, only two of the test suites generated, or all three of the test suites generated. The table shows a trend observed generally (across all four augmentation techniques) on the program: the concolic approach either succeeded or failed in all cases (on all test suites created), whereas the genetic algorithm often encountered branches that are covered only probabilistically, i.e., on some test suites generated but not on others.

Table 5.16: Numbers of Times in which Branches in `grep` were Covered by One, Two, or Three Test Suites, for DFO with Existing and New Test Cases

version	GA Only			CT Only		
	1	2	3	1	2	3
v1	25	11	339	0	0	302
v2	13	16	374	0	0	334
v3	16	11	356	0	0	349
v4	13	21	360	0	0	341
v5	6	10	366	0	0	334

5.5.2 Analysis of Specific Branches

To further understand the differences in technique performance, we selected several branches from `replace`, `printtok1` and `grep` on which such differences occurred and analyzed them to determine causes of the differences. On `replace` we selected the seven branches that exhibited the most extreme differences in results, on which the concolic algorithm greatly outperformed the genetic algorithm. On `printtok1` we selected the seven branches that exhibited the most extreme differences in results, on which the genetic algorithm greatly outperformed the concolic algorithm. On `grep`, where we have only three test suites, we could not locate branches that were outliers, so instead we randomly sampled four branches that were easy for the concolic approach to cover but not for the genetic approach to cover, and four branches in which this situation was reversed.

Considering `replace` first, we were able to classify the seven branches on which the concolic algorithm outperformed the genetic algorithm into three groups based on three overall observed causes of problems in coverage.

The first group (G1) of branches relates to limitations in the mutation pool settings chosen for the genetic algorithm. One of the seven branches falls into this group. In `replace`, there is a predicate that checks the number of input arguments provided to the program, and the program needs to be given fewer than two arguments to cover

the "true" branch out of this predicate. In the initial population of test cases provided to the algorithm, however, all test cases have two or three arguments, and we did not include the choice of mutating the number of inputs as part of our mutation pool. Thus, the genetic algorithm can never cover the branch. To cover this branch with the genetic approach, we would need to have sufficient knowledge of the program internals to cause us to change this behavior, perhaps via a pre-processing static analysis. In our study we treated the programs as black boxes for the genetic algorithm, and tuning is done based on program specifications, inputs, and environment conditions. In contrast, the concolic approach treats program as white boxes, and applying it requires testers to consider program internals. Thus, for the concolic approach, we specified the number of arguments as a symbolic value and this let us cover the branch in question on every run.

The second group (G2) of branches also involves mutation pool settings, but of a different type, and three of the seven branches belong to it. There are several branches in `replace` such that, for those branches to be taken, characters in specific strings must equal the NULL character. Because we did not include this character in our mutation pool, the only way in which it would occur in a test case would be if it occurred in the initial test case population, and this is infrequent. Thus, it is difficult for the genetic algorithm to cover such branches. Including all possible characters in the mutation pool could remedy this, but would increase the search space and cost of the approach substantially. Further analysis of the program could also remedy this, at the cost of such analysis. In contrast, the concolic approach does not exclude the character.

The third group (G3) of branches involves the presence of deeply nested if branches, and three branches belong to it. Predicates in deeply nested branches pose a well-known problem for genetic algorithms, although the algorithms can be helped through

specific program transformations [57]. For example, in `replace`, there is one branch in a function named `in_set_2`. This is in the first `if` statement in that function, but this function is called at the tenth level of its callee function `makepat`. Above `makepat` there are two other functions. To cover this branch a test case must satisfy several conditions. The genetic algorithm has no "knowledge" of these conditions and simply attempts to proceed in a general search direction; thus it is difficult for the algorithm to satisfy all the conditions at once. Here too, the concolic approach, by design, has no problem.

Considering `printok1`, we were able to classify the seven branches considered into two groups. The first group (G4) contains one branch, and the failure of the concolic algorithm to cover it is related to the limitations of CREST on pointer arithmetic and non-linear arithmetic. More specifically, `printtok1` contains a predicate `check_delimiter()` that contains the `isalpha()` and `isdigit()` C standard macro functions. Both of these functions use the bit-wise `&` operator and pointer arithmetic. To cover this branch using concolic testing, we would need to use an implementation that supports bitwise operators by employing bit-vector logic, and handle pointer arithmetic by providing a memory model. In contrast, the genetic approach is not affected by complex expressions such as this because it does not attempt to solve path constraints.

The failure of the concolic approach to cover the second group (G5) of branches, including the other six, is due to iteration limits. The `printok1` program includes a `next_state()` function that uses a symbolic input character as an index into an array of characters. Since CREST does not support accesses to array elements through a symbolic index variable, it transforms the process to use `if-then-else` statements to handle all possible values of the symbolic index variable one by one. For example, for a symbolic `unsigned char` variable `i`, `int next_state(int i) { ... if(a[i]==C)`

Table 5.17: Summary of Coverage Limitations

group	program	number of branches	weak algorithm	specific cause	classification
G1	`replace`	1	GA	limitations in mutation pool setting (arguments)	tuning
G2	`replace`	3	GA	limitations in mutation pool setting (NULL char)	tuning
G3	`replace`	3	GA	deeply nested ifs not reached	algorithmic
G4	`printtok1`	1	CT	limitations handling arithmetic constructs	implementation
G5	`printtok1`	6	CT	iteration limits and loops	algorithmic
G6	`grep`	1	GA	deeply nested ifs not reached	algorithmic
G7	`grep`	3	GA	malloc failures not covered	algorithmic
G8	`grep`	2	CT	external libraries not analyzable	algorithmic
G9	`grep`	2	CT	dynamic memory management not controlled	algorithmic

`f(b[i]);...`} is transformed into the following code where `a` is an array of characters, `b` is an array of integers (suppose that `b[i]=i+10`; i.e., `b[0]=10`, `b[1]=11`, ...), and `C` is a character constant:

```
01:void f(int x){
02: if (x == 10){ ... }
03: else if (x == 20){ ... }
04: else if (x == 30){ ... } ...  }
05:
06:int next_state(int i){...
07: // Transformation of
08: // if(a[i]==C) f(b[i]);
09: if(i==0 && a[0]==C)         f(b[0]);
10: else if(i==1 && a[1]==C)    f(b[1]);
11:  ...
12: else if(i==255 && a[255]==C)f(b[255]);
13: ... }
```

However, this transformation still does not solve the problem completely. Suppose that the concolic approach tries to cover the branches in `f()` (lines 2-4). The concolic approach controls the symbolic variable `i` that is passed to `next_state()` as a parameter (line 6) and controls the parameter to `f()` indirectly (lines 9-12). In other words, to cover the branches in `f()`, the concolic approach has to try corresponding different branches in `next_state()` (i.e., a maximum of 256 different values for symbolic variable `i`). Given an iteration limit less than 10, there is little chance for the approach to reach all branches in `f()`. For example, suppose that a target branch b_t is the `then` branch of `f()` at line 2 (i.e., `x==10`). Also suppose that an initial value of `i` is 255, which makes the first symbolic execution path be $\neg(i = 0 \wedge a[0] = C) \wedge \neg(i = 1 \wedge a[1] = C) ... \wedge \neg(i = 254 \wedge a[254] = C) \wedge (i = 255 \wedge a[255] = C) \wedge \neg(x = 10) ...$ (see the rightmost execution path in Figure 5.4). To cover b_t, Algorithm 4 has to iterate through lines 8-16 255 more times, since b_t can be covered by only the leftmost execution path in Figure 5.4. However, this is not possible since $n_{iter} < 10$ in our experiments (see line 8 in Algorithm 4). In contrast, the genetic approach may reach any of the branches if it succeeds in choosing appropriate inputs.

Finally we turn to `grep`. Of the four branches on which the genetic approach had difficulties, one (group G6) was a deeply nested branch, similar to the case discussed above with respect to `replace`. The other three branches (group G7) are all incident on `malloc` attempts, and taken when that routine fails due to the exhaustion of memory. It is virtually impossible for the genetic approach to generate test cases for `grep` that consume enough memory to trigger coverage of these branches. The concolic approach, however, covers them, but this is actually a side effect rather than a direct effect. This is because the concolic algorithm saves execution path information

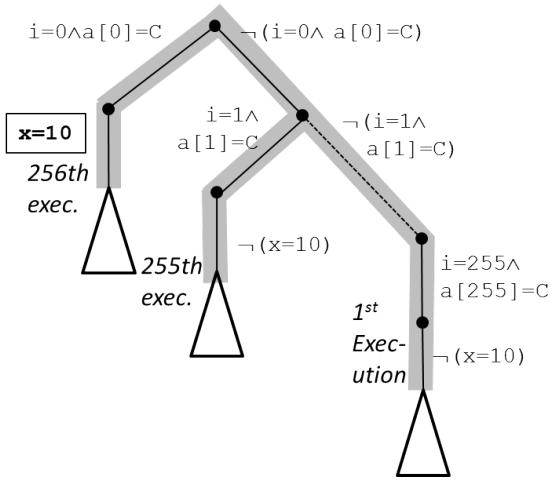


Figure 5.4: Symbolic execution tree of the example code

for test cases, and eventually this path information can consume enough memory to cause `malloc` failures.

For the four branches on which the concolic approach had difficulties, we identified two groups, each relevant to two of the branches. The first group (G8) is related to external binary library functions such as `strcmp()` and `strlen()`. Branches belonging to this group are taken based on results of these binary library functions. These functions cannot be analyzed by the concolic algorithm, and thus it fails to generate test cases that cover them. The genetic approach does not need to analyze the functions and does select inputs that cover the branches.

The second group (G9) of branches are related to dynamic memory management. For example, `grep` transforms a given regular expression pattern into a deterministic finite automaton (DFA) and stores the DFA in a buffer. Before `grep` stores the DFA into the buffer, it should check whether the size of the buffer is large enough to contain the DFA. If not, `grep` extends the buffer. Since the concolic approach cannot control the size of the DFA directly via path conditions, it is difficult for it to cover branches that compare the size of the buffer and the size of the DFA. The genetic approach,

however, due to the diversity created through crossover and mutation, can by chance end up with test cases that vary the DFA size as needed.

Table 5.17 summarizes the foregoing results. For each of the groups identified, the table lists the program(s) that group occurred in, the number of branches, the algorithm that exhibited the weakness in achieving coverage, and the cause of the weakness. The rightmost column in the table classifies the observed weaknesses into three categories, as follows.

The first broad category of weaknesses (groups G1 and G2, four branches) involve tuning limitations (mutation pool settings), and occurred only for the genetic algorithm. Such weaknesses will necessarily occur for that algorithm due to the way in which the algorithm must be applied; however, in practice they could be partly addressed by tuning the algorithm better, which is particularly possible in the context of an evolving program as test suites are reused and improved on subsequent versions.

The second broad category of weaknesses (group G4, one branch) involve effects related to implementations, and occurred only for the concolic algorithm. In this case, the failure of the technique is not algorithmic, but rather, is due to the specific implementation of the algorithm, and could be addressed through improvements in implementations. For example, the concolic approach *could* be implemented to better handle non-linear arithmetic.

The third broad category of weaknesses (groups G3, G5, G6, G7, G8, and G9, 18 branches) involve neither tuning problems nor implementation problems, but rather, lie in the natures of the algorithms themselves. Genetic algorithms are simply not likely to handle deeply nested ifs (groups G3 and G6), whereas concolic algorithms can. Concolic algorithms are simply not able to handle non-analyzable external libraries or dynamic memory management issues (groups G8 and G9). We also place group G5 in this category. While we selected iteration limits and thus, they might

be seen as a matter of tuning, at the core of the concolic approach some limit will be needed as an algorithmic matter, and there could exist programs such that, for any limit selected, that limit is not sufficient to allow certain branches to be reached. Finally, regarding group G7, the fact that the concolic implementation could cover branches incident on malloc failures is related to the algorithm's need to collect data that can exceed available memory.

5.6 Conclusions and Future Work

In this work we have focused on test suite augmentation, and our results have several implications for the creation and further study of augmentation techniques. Perhaps the most intriguing result stems from the observed complimentariness of the concolic and genetic test case generation approaches, and the consequent implications this raises for the prospects of hybrid approaches. The results also have implications, however, for engineers creating initial test suites for programs. This is because such engineers often begin, at least at the system test level, with black box requirements-based test cases. It has long been recommended that such test suites be extended to provide some level of coverage. The techniques we have presented can conceivably serve in this context too, working with initial black-box test cases and augmenting these.

There are additional factors that influence augmentation that we have not examined directly in this work. Program characteristics certainly play a role, because they can impact the ability of test case generation techniques to function cost-effectively, as described in Sections 5.2.3 and 5.2.4. Characteristics of program modifications also matter. More formal studies of these factors could be helpful.

Chapter 6

Advanced Test Suite Augmentation Technique - Hybrid Algorithm

Since we have seen the benefits of combining test suite augmentation techniques, we begin building advanced test suite augmentation techniques by considering them together. As our first step toward advanced test suite augmentation techniques, we have created a hybrid test suite augmentation technique by combining a concolic testing and a genetic algorithm. (This work has appeared in [101].)

6.1 Related Work: Combination of Techniques

Recently, other researchers have combined different techniques to help generate test cases. Hybrid concolic testing [54] combines random and concolic testing to generate test cases. In contrast, our technique combines genetic and concolic techniques, and we focus on the test suite augmentation context, in which there are many other factors to be considered that are not discussed in [54]. Inkumsah et al. [45] combine a genetic algorithm and concolic testing to generate test cases for programs. They focus on unit

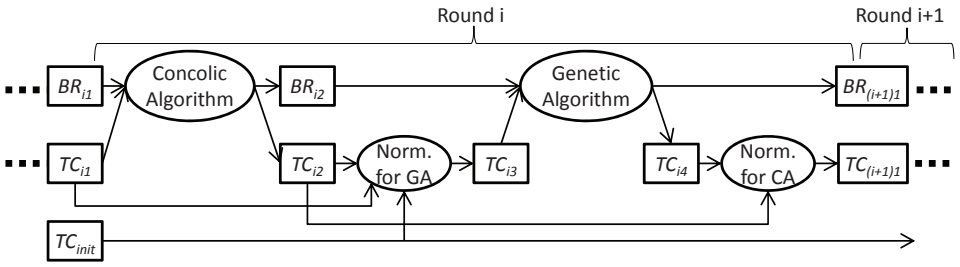


Figure 6.1: Overview of hybrid test suite augmentation approach

testing of objected-oriented programs, whereas we focus on system testing. Further, they use evolutionary testing to find method sequences and concolic testing to cover branches, whereas our hybrid approach uses the two generation methods together to enhance branch coverage. Finally, their approach does not reuse existing test cases, which is central to our approach.

6.2 Direct Hybrid Test Suite Augmentation

The results of Chapter 5 suggest that a hybrid test suite augmentation technique should be created keeping the following requirements in mind:

1. Concolic test case augmentation is much more efficient than genetic test case augmentation. Thus, a hybrid technique should begin by using a concolic test case generation algorithm and attempt to cover as many branches as possible before passing control to a genetic test case generation algorithm.
2. Processing targets in depth-first order can improve the efficiency of the genetic algorithm but has no effect on the concolic algorithm. Thus, we can order the targets to improve the former without harming the latter.
3. Test reuse approach has an impact on the effectiveness of the concolic algorithm. When using that algorithm we should utilize new test cases as they are created.

Our hybrid test suite augmentation technique is summarized in Figure 6.1. This hybrid technique incorporates multiple *rounds* of test case generation, where one round consists of an application of a concolic test case generation algorithm followed by an application of a genetic test case generation algorithm. We focus on branch coverage rather than path coverage for issues of scalability; rounds continue until no new branches are covered. In the ith round, the concolic algorithm receives a list of target branches BR_{i1} and a set of test cases TC_{i1} from the (i-1)th round, where BR_{11} is a list of all target branches sorted in depth-first order and $TC_{11} = TC_{init}$ is a set of initial test cases.[1] For each round i:

1. The concolic algorithm generates a set of new test cases TC_{i2}, each of which covers at least one new branch. After this step, $BR_{i2} = BR_{i1} - cov(TC_{i2})$, where $cov(TC)$ indicates a set of branches covered by TC_{i2}.
2. TC_{init}, TC_{i1} and TC_{i2} are *normalized/modified* to form a test case population TC_{i3} for genetic testing. Currently, the genetic algorithm employed by our hybrid augmentation technique fixes the size of a test case population at $|TC_{init}|$ for all rounds (i.e.,$\forall i \geq 1, |TC_{i3}| = |TC_{init}|$). This normalization process randomly selects $|TC_{init}|$ test cases from $TC_{init} \cup TC_{i1} \cup TC_{i2}$.
3. The genetic algorithm generates a set of test cases TC_{i4}, each of which covers at least one new branch After this step, $BR_{(i+1)1} = BR_{i2} - cov(TC_{i4})$.
4. TC_{i2} and TC_{i4} are normalized to form $TC_{(i+1)1}$, a set of test cases that is used by the concolic algorithm in the $(i+1)$th round. Currently, this step sets $TC_{(i+1)1}$ to $TC_{i2} \cup TC_{i4}$, which are new test cases. This step enables the concolic algorithm to utilize the "old+new test case reuse strategy" (requirement 3).

[1] In the first round, any set of branches determined to need coverage can be passed to the algorithm; in this work we assume that a regression test suite has been executed on the program, and that the initial set of branches is the set of branches not covered by the test cases in that suite.

The precise algorithms used for concolic and genetic test case generation in the foregoing hybrid augmentation technique are similar to those described in Section 5.2. To avoid redundancy, we do not repeat them here.

6.3 Empirical Study

Our goal is to compare the use of our hybrid directed test suite augmentation technique to non-hybrid techniques. We thus pose the following research questions.

RQ1: How does hybrid test suite augmentation compare, in terms of cost and effectiveness, to augmentation using a straightforward concolic test case generation technique?

RQ2: How does hybrid test suite augmentation compare, in terms of cost and effectiveness, to augmentation using a straightforward genetic test case generation technique?

6.3.1 Objects of Analysis

To facilitate augmentation technique comparisons, programs must be suitable for use by all techniques. Also, programs must be provided with test suites that need to be augmented. In our prior work (Chapter 5) we selected several programs from the SIR repository [31] that meet the needs of such comparisons. Here we utilize three of these programs, printtok1, printtok2 and replace. We reuse the test suites generated for them in that work here also.

6.3.2 Variables and Measures

The comparison of hybrid and non-hybrid techniques is complicated by the fact that they inherently involve different amounts of effort. One could certainly run the two types of techniques for the same amount of time and compare their relative effectiveness, but we expect that in practice, engineers would run the techniques until the techniques cease to achieve sufficient new coverage, and then stop. It thus seems more appropriate to run the techniques to some reasonable stopping points, and then compare their relative effectiveness and efficiency. We choose independent and dependent variables keeping this approach in mind. Further, as discussed below, we use different iteration limits to investigate the variance that might be seen in performance if techniques are allowed to run longer times.

Independent Variable. Our experiment manipulates one independent variable: the augmentation technique used. Three treatments were chosen for this variable: (1) the hybrid test suite augmentation technique described in Section 6.2, (2) an augmentation technique using just concolic test case generation, and (3) an augmentation technique using just genetic test case generation.

Dependent Variable. We wish to measure both the effectiveness and the efficiency of augmentation techniques under each combination of potentially affecting factors. To do this we selected two variables and measures:

DV1: Effectiveness in terms of coverage. The test case augmentation techniques that we consider are intended to work with existing test suites to achieve higher levels of

coverage in a modified program P'. To measure the effectiveness of techniques, we track the number of branches in P' that can be covered by each augmented test suite.

DV2: Efficiency in terms of time. To track augmentation technique efficiency, for each application of an augmentation technique we measure the *cost* of using the technique in terms of the wall clock time required to apply the technique.

6.3.3 Experiment Setup and Operation

We followed the same steps described in Section 5.3.3 to set up the experiments. We continue to use the extended program and we apply several iteration limits on both techniques. However, in these experiments we use only three iteration limits for each test case generation algorithm, choosing 1-5-9 for concolic and 5-15-25 for genetic, because prior studies showed that these represented lower and upper bounds outside of which technique effectiveness ceased to vary by more than small amounts. We use the same concolic testing and genetic algorithm implementations described in Section 5.3.3.

6.3.4 Threats to Validity

The primary threat to *external validity* for this study involves the representativeness of our object programs and test suites. We have examined only three relatively small C programs using simulated versions, and the study of other objects, other types of versions, and other test suites may exhibit different cost-benefit tradeoffs. However, if results on smaller programs show that our approach is beneficial, then arguably, programs with more complex features should enable a hybrid approach to function even better. A second threat to external validity pertains to our algorithms; we have utilized only one variant of a genetic test case generation algorithm, and one variant

of a concolic testing algorithm, and we have applied both to extended versions of the object programs, where the genetic approach does not require this and might function differently on the original source code. Subsequent studies are needed to determine the extent to which our results generalize.

The primary threat to *internal validity* is possible faults in the implementation of the algorithms and in tools we use to perform evaluation. We controlled for this threat through extensive functional testing of our tools. A second threat involves inconsistent decisions and practices in the implementation of the techniques studied; for example, variation in the efficiency of implementations of techniques could bias data collected.

Where *construct validity* is concerned, there are other metrics that could be pertinent to the effects studied. In particular, our measurements of efficiency consider only technique run-time, and omit costs related to the time spent by engineers employing the approaches. Our time measurements also suffer from the potential biases detailed under internal validity, given the inherent difficulty of obtaining an efficient technique prototype.

6.4 Results

Tables 6.1, 6.2, and 6.3 present the data obtained in our study for the three object programs, respectively. Each table shows cost and coverage data. Data is shown per iteration limit, with CA1, CA5, and CA9 representing limits for the concolic test case generation algorithm, and GA5, GA15, and GA25 representing limits for the genetic test case generation algorithm. A given cell in the table represents a comparison between the techniques indicated by the label at the top of the column containing that cell.

Table 6.1: Coverage and Cost Data for Printtok1

COST (seconds)						
	CA1		CA5		CA9	
	CA HY	GA HY	CA HY	GA HY	CA HY	GA HY
GA5	***1.54 57.51***	56.38 57.51	***6.94 67.11***	***56.38 67.11***	***12.27 75.06***	***56.38 75.06***
GA15	***1.54 190.37***	***210.56 190.37***	***6.94 200.15***	***210.56 200.15***	***12.27 192.33***	***210.56 192.33***
GA25	***1.54 351.87***	339.19 351.87	***6.94 405.57***	***339.19 405.57***	***12.27 414.33***	***339.19 414.33***
COVERAGE (branches)						
	CA HY	GA HY	CA HY	GA HY	CA HY	GA HY
GA5	***143.97 155.58***	***154.89 155.58***	***151.29 155.65***	***154.89 155.65***	***152.50 155.78***	***154.89 155.78***
GA15	***143.97 156.11***	155.88 156.11	***151.29 156.23***	155.88 156.23	***152.50 156.02***	155.88 156.02
GA25	***143.97 156.62***	156.54 156.62	***151.29 156.51***	156.54 156.51	***152.50 156.51***	156.54 156.51

Table 6.2: Coverage and Cost Data for Printtok2

COST (seconds)						
	CA1		CA5		CA9	
	CA HY	GA HY	CA HY	GA HY	CA HY	GA HY
GA5	***0.25 35.67***	***32.21 35.67***	***0.84 35.02***	***32.21 35.02***	***1.43 31.86***	32.21 31.86
GA15	***0.25 153.88***	***131.25 153.88***	***0.84 154.71***	***131.25 154.71***	***1.43 159.42***	***131.25 159.42***
GA25	***0.25 275.49***	***248.64 275.49***	***0.84 291.97***	***248.64 291.97***	***1.43 296.92***	***248.64 296.92***
COVERAGE (branches)						
	CA HY	GA HY	CA HY	GA HY	CA HY	GA HY
GA5	***165.34 176.06***	***175.85 176.06***	***171.59 176.42***	***175.85 176.42***	***173.00 176.41***	***175.85 176.41***
GA15	***165.34 176.58***	***176.34 176.58***	***171.59 176.54***	***176.34 176.54***	***173.00 176.60***	***176.34 176.60***
GA25	***165.34 176.62***	***176.40 176.62***	***171.59 176.67***	***176.40 176.67***	***173.00 176.65***	***176.40 176.65***

Table 6.3: Coverage and Cost Data for Replace

COST (seconds)						
	CA1		CA5		CA9	
	CA HY	GA HY	CA HY	GA HY	CA HY	GA HY
GA5	***0.74 84.66***	***90.49 84.66***	***4.40 75.99***	***90.49 75.99***	***8.04 82.55***	***90.49 82.55***
GA15	***0.74 341.95***	***320.71 341.95***	***4.40 322.79***	320.71 322.79	***8.04 322.19***	320.71 322.19
GA25	***0.74 570.88***	***618.83 570.88***	***4.40 552.71***	***618.83 552.71***	***8.04 576.28***	***618.83 576.28***
COVERAGE (branches)						
	CA HY	GA HY	CA HY	GA HY	CA HY	GA HY
GA5	***176.43 186.94***	***185.80 186.94***	***187.24 190.02***	***185.80 190.02***	***188.59 190.53***	***185.80 190.53***
GA15	***176.43 188.58***	***187.83 188.58***	***187.24 190.51***	***187.83 190.51***	***188.59 190.75***	***187.83 190.75***
GA25	***176.43 189.18***	188.81 189.18	***187.24 190.66***	***188.81 190.66***	***188.59 190.88***	***188.81 190.88***

Next we analyze our results, per research question.

6.4.1 RQ1: Hybrid versus Concolic

The columns labeled "CA HY" in Tables 6.1, 6.2, and 6.3 present data relevant to this question. Each entry in these columns shows the comparison between the hybrid test suite augmentation technique and the concolic test suite augmentation technique

in terms of cost or coverage. The numbers represent the average cost of, or coverage obtained by, the two techniques across all 100 test suites. For example, the first entry in Table 6.1 contains 1.54 and 57.51. Here, 1.54 represents the average cost in seconds to perform test suite augmentation across 100 test suites with the concolic augmentation technique run at iteration limit 1, while 57.51 represents the average cost in seconds when the hybrid augmentation technique is used with its concolic algorithm component run at iteration limit 1 and its genetic algorithm component run at iteration limit 5. For each pair of data sets (each cell in the tables), we applied a *Wilcoxon test* to determine whether there is a statistically significant difference between the two techniques, using $\alpha = 0.05$ as the confidence level. In the table, bold-italicized fonts indicate statistically significant differences. For example, for the first entry of Table 6.1, comparing the costs of the hybrid augmentation technique and the concolic augmentation technique, there is a statistically significant difference between these two, and the concolic technique cost less than the hybrid technique.

We begin by considering comparisons in terms of cost. The concolic technique cost less than the hybrid technique on all programs, and the differences in cost were statistically significant in all cases. On `printtok1`, the hybrid technique cost up to 350 times more than the concolic technique; on `printtok2`, the hybrid technique cost up to 110 times more than the concolic technique; and on `replace`, the hybrid technique cost up to 771 times more than the concolic technique. (All of these maximal differences occurred when the concolic technique was run at iteration limit 1 and the genetic component of the hybrid technique was run at iteration limit 25.)

Where effectiveness is concerned, the hybrid technique has advantages. In all entries related to coverage comparisons between the hybrid technique and the concolic technique, the hybrid technique covers more branches than the concolic technique, and the differences are statistically significant in all cases. On `printtok1`, the hy-

brid technique covered up to 13 branches more than the concolic technique; and on `printtok2` and `replace`, the hybrid technique covered up to almost 13 branches more than the concolic technique. Maximal differences occurred when the concolic technique was run at iteration limit 1 and the genetic component of the hybrid technique was run at iteration limit 25.

To summarize, comparing the concolic test case augmentation technique to the hybrid technique, the hybrid technique was more effective but less efficient.

6.4.2 RQ2: Hybrid versus Genetic

The columns labeled "GA HY" in Tables 6.1, 6.2, and 6.3 present data relevant to this question. We again begin with cost comparisons. Here, results varied more widely than in the case of RQ1. On `printtok1`, the hybrid augmentation technique cost more (by up to 33%) than the genetic augmentation technique in six of nine cases, of which four involve statistically significant differences. The genetic technique cost more (by up to 11%) than the hybrid technique in three cases, all of them statistically significant differences occurring when the genetic component of the hybrid technique was run at iteration limit 15. On `printtok2`, the hybrid technique cost more (by up to 21%) than the genetic technique in eight of nine cases, all of which involve statistically significant differences. The only exception occurred when the concolic component of the hybrid algorithm was run at iteration limit 9 and the genetic component was run at iteration limit 5, in which case the two did not differ significantly. On `replace`, the genetic technique cost more (by up to 19%) than the hybrid technique in six cases, all of which involved statistically significant differences. The genetic technique cost less in the other three cases, only one of which involved a statically significant difference.

In terms of coverage, on `printtok1` the hybrid technique achieved higher coverage than the genetic technique in seven cases, of which three involved statistically significant differences. The genetic technique had better coverage in the other two cases but with no statistically significant differences, and in both situations the differences were smaller than one branch. On `printtok2` and `replace`, the hybrid technique achieved higher coverage in all cases in which there are statistically significant differences. On `printtok2` the differences were less than one branch while on `replace`, the differences ranged from less than one branch up to almost five branches.

Overall, comparing the genetic test suite augmentation technique and the hybrid test suite augmentation technique, the hybrid technique achieved greater coverage than the genetic technique and sometimes (but not always) cost less.

6.5 Discussion and Implications

We now discuss the results presented in the prior section, and comment on their implications.

The hybrid test case augmentation technique outperformed both the concolic and genetic augmentation techniques in terms of effectiveness in most cases. If our results generalize, then when effectiveness has the highest priority, the hybrid technique is the best choice. In this respect, the results of our study met our expectations.

Where the cost of augmentation techniques is concerned, however, the results presented some surprises. On one hand, it is obvious that the hybrid technique should cost more than the concolic technique, because the hybrid technique includes a genetic algorithm component, which itself requires much more time than the concolic technique. On the other hand, we had expected the hybrid augmentation technique to cost less than the genetic augmentation technique, because the hybrid technique

Table 6.4: Branches Covered by Both Algorithms over Branched Covered by the Concolic Algorithm

	CA1	CA5	CA9
`printtok1`	53.26%	56.92%	55.15%
`printtok2`	79.49%	72.88%	69.52 %
`replace`	35.15%	32.83%	32.47%

begins with a concolic test case generation step, which should cover some targets in a relatively short time, leaving fewer targets for the genetic algorithm to work on. We did observe this result in most cases on `replace`. On `printtok1` and `printtok2`, however, the hybrid technique usually did *not* save time with respect to the genetic technique. We inspected our results further and found that there are two reasons that can account for this difference.

6.5.1 Masked-out Benefit of Concolic Testing

The first reason for the performance difference is that the branches covered by the concolic algorithm component of the hybrid technique are easily covered by the genetic algorithm component of the hybrid technique, in the first few iterations of the genetic algorithm component. This means that the benefits of concolic testing (i.e., coverage of target branches in a relatively short time compared to the genetic algorithm) can be "masked out" at the beginning of the genetic algorithm. To further investigate this, we identified branches covered by the concolic algorithm and branches covered by the genetic algorithm in the first five iterations (note that in this case we applied both algorithms separately, not in the hybrid framework). Then, we calculated the percentage of branches that are covered by both algorithms over branches covered by the concolic algorithm. Table 6.4 shows these percentage numbers. For example, the entry 53.26% in column CA1 for `printtok1` means that the straightforward genetic algorithm covers 53.26% of the branches in five iterations that are covered by the

concolic algorithm with iteration limit 1. As the table shows, on `printtok1`, the genetic algorithm covers more than 53% of the branches covered by the concolic algorithm across all levels. On `printtok2`, the genetic algorithm covers even more branches: up to 79% of those covered by the concolic algorithm. Thus, this can explain why the hybrid algorithm is slower than the genetic algorithm on `printtok2`, since even more benefits of the concolic algorithm are masked out in this case. On `replace`, in contrast, the genetic algorithm covers fewer branches, so the benefits of using the concolic algorithm first are realized to a larger extent, and the hybrid technique saves time compared to the genetic technique.

6.5.2 Weakened Diversity of Test Case Population

The second reason for the performance difference involves the diversity of the test case population. In the hybrid technique we randomly select test cases from the existing test cases and the test cases newly generated by the concolic algorithm to form an initial population of test cases for use by the genetic algorithm. The test cases generated by the concolic algorithm, however, tend to be only slightly different from existing test cases, due to the manner in which the concolic algorithm operates. Thus, when drawing from these newly generated test cases it is more likely that an initial population of test cases will lack diversity, and this can reduce the efficiency of the genetic algorithm.

To further investigate this issue, we performed an additional set of runs using a version of the hybrid technique in which the genetic algorithm uses only test cases from the initial test suite TC_{init} to form the initial population for targets. When we compare the coverage data from these runs to the coverage data reported in Section 6.4, there are no statistically significant differences. When we compare the cost

Table 6.5: Cost Differences Between Hybrid Algorithms

		printtok1			printtok2			replace		
		GA			GA			GA		
		5	15	25	5	15	25	5	15	25
CA	1	***H2***	***H2***	***H2***	H1	***H2***	***H2***	H1	***H2***	***H2***
	5	***H2***	H2	***H2***	H1	***H2***	***H2***	H1	***H2***	***H2***
	9	***H2***	H1	***H2***	H1	***H2***	***H2***	H1	***H2***	***H2***

data from these runs, however, in most cases this new version of the hybrid algorithm (H2) outperformed the initial one (H1). Table 6.5 shows the cost comparison between the two approaches. Table entries of "H1" indicate that the first hybrid algorithm cost less than the second, while entries of "H2" indicate that the second algorithm cost less than the first. Bold-italicized entires indicate that there is a statistically significant difference between the techniques. As the table shows, in most cases H2 cost significantly less than H1. This confirms our conjecture that the newly generated test cases affect the diversity of the population for the genetic algorithm, since this is the only differences between the two hybrid techniques. Nevertheless, H2 continues to have the shortcoming mentioned earlier (masked-out benefits of concolic testing) and does not significantly improve efficiency.

6.5.3 Potential Remedies

The foregoing discussion reveals several ways in which our basic hybrid algorithm could be improved. One method for overcoming the masked-out benefit of concolic testing (Section 6.5.1) is to customize a concolic algorithm to attempt to reach branches that are difficult for a genetic algorithm to reach first. For example, it is well known that deeply nested branches are difficult for genetic algorithms to cover. We can modify a concolic algorithm to focus on such branches first. We can also modify the genetic algorithm to target branches that are difficult for the concolic algorithm to cover due to the presence of external libraries or floating point arithmetic.

Regarding the weakened diversity problem (Section 6.5.2), we can select only new test cases generated by concolic testing that are largely different from each other as an initial population for genetic testing. Alternatively, we can enhance symbolic path formulas to generate a solution that is much different from the previous one by inserting additional constraints on the solution space. Last, we can fully utilize the randomized capability of an underlying SMT solver to obtain more diverse solutions.

6.6 Conclusions

We have presented a hybrid technique for performing test suite augmentation, that utilizes both concolic and genetic test case generation algorithms in an attempt to harness the different strengths of both. Our empirical study of this technique shows that it can improve augmentation effectiveness, but as initially configured, it does not consistently save time in comparison to the genetic and concolic test suite augmentation techniques. Our analysis of these results uncovers reasons for this effect, and supports suggestions on how to improve the hybrid technique.

In this work we have focused on test suite augmentation. Our results also have implications, however, for engineers creating initial test suites for programs. Engineers often begin, at least at the system test level, with black box requirements-based test cases. The techniques we have presented can conceivably help these engineers extend initial black-box test cases to achieve better code coverage.

Chapter 7

Test Suite Augmentation for SPLs

As mentioned in Chapter 1, the products of a software product line share some similarities with the versions of a traditional program. In this chapter, we apply the test suite augmentation idea to software product lines. We begin by introducing software product line testing. Then we focus on our methodology. Finally we discuss our experimental results. (This work will appear in [98].)

7.1 Software Product Line Testing

Testing software product line has focused on blackbox or specification-based testing approaches, performing from a product-based view. For instance, feature models have been used to represent the product space for instantiating and sampling products for testing [10, 25, 64, 89]. The work of Uzuncaova et al. [89] transforms a feature model into an Alloy specification and uses this to generate test cases, while the work of Cohen et al. [25] and Oster et al. [64] uses the feature model to define samples of products that should be included in testing. Similarly, the PLUTO methodology [10] uses the feature model to develop extended use cases that contain variability which

can formulate a full set of test cases for the family of products. Schürr et al. [79] use a classification tree method for testing from the feature model and Reis et al. [70] perform integration of features based on UML activity diagrams from which they derive feature dependencies. Cabral et al. [16] use a graph derived from a feature model (a feature inclusion graph) to select subsets of products and test cases that can be run on them.

Denger et al. [29] present an empirical study to evaluate the difficulty of detecting faults in the common versus variable portions of an SPL code base, concluding that the types of faults found in these two portions of the code differ. In other code-based approaches, Kim et al. [47] use a dependency analysis to determine which features are relevant for each test case within a test suite, reducing the number of products tested per test case (again a product-based view). Shi et al. [82] use a dataflow analysis to reduce the number of combinations of features that should be tested together and compositional symbolic execution to integrate features.

Techniques that leverage ideas from regression testing in SPL contexts also exist [27, 43] and these involve examining changes to the feature model or architecture, not the products themselves.

In this chapter, we present a hybrid technique for testing SPLs. It involves a hybrid of family and product based approaches, is white-box (aims to cover more common code at each step), and transforms the testing problem into one of regression testing.

7.2 CONTESA

We call the system we have developed for testing SPLs "CONTESA". In this section we describe how CONTESA works. The notion behind the approach is that since products within a product line are not independent (we expect them to share

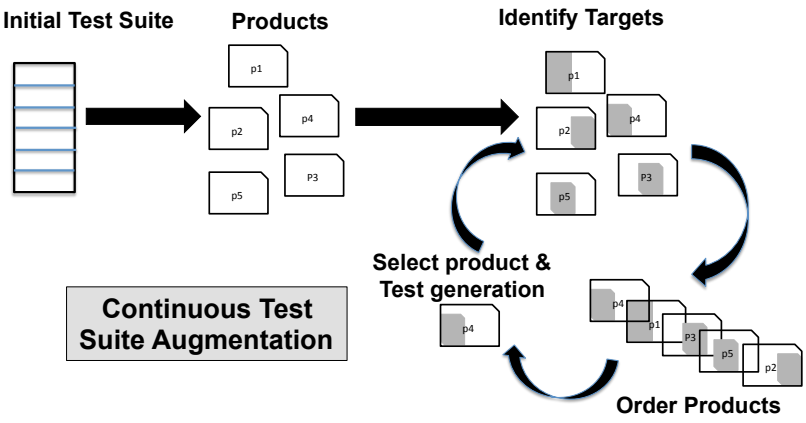


Figure 7.1: Overview of CONTESA

large portions of code), we can approach test suite augmentation for product lines in a continuous instead of an independent fashion, applying augmentation to products iteratively, taking advantage of test cases and testing information derived while addressing prior products.

Figure 7.1 provides an overview of CONTESA. The algorithm begins with an initial (possibly empty) set of test cases. These are the base test cases for the product line, and could have been created following any applicable approach. CONTESA runs each product's test suite on that product and identifies affected elements in the product – these form an initial set of *targets* (and in the rest of this paper we refer to them as such) that need to be covered by test cases. CONTESA then begins to iterate over the products, selecting a next product for augmentation based on heuristics that we describe below. Given a product, CONTESA uses a test case generation technique to attempt to cover targets in that product, and applies this technique until all targets have been covered or some stopping criterion is reached. CONTESA then recomputes the targets for products not yet tested; this computation informs the selection of the next product for testing. CONTESA then moves on to the next iteration.

In the following sections we describe the three overall activities performed by the approach (identifying targets, ordering products and selecting one, and generating test cases) in turn.

7.2.1 Identifying Targets

During augmentation, a target may be a statement, branch, path or method in a program [100]. In this work we use branches as targets, but CONTESA could function on other types of code components. There are several methods for identifying target branches in a product. The easiest method involves considering all branches in each product as targets. Given a set of initial test cases for a product line, we could run those test cases and then treat the uncovered branches in each product as our targets. We refer to this set of uncovered branches for a given product p_i as UC_{p_i}.

Attempting to cover all uncovered branches in each product in a product line can be expensive, and it is this expense that we wish to reduce via our approach. We observe that, as we attempt to generate test cases for a product p_i, we likely have already covered many branches in products previously tested that also exist in p_i, and that are not affected by differences between those prior products and p_i. We call these branches *common* branches. Rather than attempt to re-generate test cases for such common branches, CONTESA considers them to be already covered and removes them from the target set.

To determine common branches for p_i, one approach is to run all test cases generated for prior products on p_i and see which branches remain uncovered. This may not always work, however, on product lines, because often, test case syntax changes across products, and test cases for one product cannot be directly run on others. An alternative approach involves applying a static analysis technique to determine common

branches, and then determining which of these common branches has already been covered during the testing of prior products. The first approach is a product-based approach, the alternative is a family-based approach. Because the second approach can apply whether or not test cases are applicable across products, and because it can save testing costs, this is the approach we investigate in this work.

In CONTESA, we employ a static analysis also based on `Dejavu` [76] to identify *common* code and non-common code (hereafter referred to as *variable* code) in products. For each pair of products p_i and p_j, this analysis takes each pair of methods that is common to the two products and traverses the two control flow graphs of these methods synchronously in depth-first order. If this traversal encounters a pair of nodes in the two graphs for which the associated code differs, it halts and returns the nodes' position. We consider any branches not reached in either graph (i.e, branches that follow differing nodes in control flow) to be *variable* branches, while branches that are reached in both graphs are *common*. We save all *common* branches for a pair of products p_i and p_j in a set $Common_{i,j}$. All branches in methods that are not common to the pair of products are necessarily treated as variable.

Given information on common branches, after we generate test cases for a product p_i, we update the coverage information for each other product p_j by excluding branches that are both *common* and covered in p_i from the uncovered branch set UC_{p_j}. This gives us a new set of uncovered branches, UC'_{p_j}, for each product p_j.

7.2.2 Ordering and Selecting a Next Product

After identifying targets for each product, we need to select a next product to work on. There are several ways in which we could do this. In the simplest approach, we could randomly select the next product, or we can select the product with the most or

the fewest features. In our approach, however, we wanted to leverage the information collected while "Identifying Targets". We have created two different techniques for doing this, one that creates a static order and one that operates dynamically. The following subsections present these two techniques.

7.2.2.1 A Static Order

Our static approach involves an order that is calculated statically before we perform any test case generation. In this approach, when selecting each next product, we wish to choose the product that has the most *common* uncovered branches with respect to products that have already been considered. Algorithm 5 presents an algorithm for computing this order. The algorithm uses several variables in addition to those already defined above, as follows:

- $P_{remaining}$ is a set that initially contains all products.
- $CommonMap$ is a map containing information about the branches that are common between each pair of products p_i and p_j.
- $InitialUncovMap$ is a map denoting the initial set of uncovered branches for each product.
- $P_{considered}$ is a set used to denote products that have already been considered.
- $Ordered$ is a list in which a prioritized list of products is returned.
- $AllUC$ is used to save information for each remaining product, and later on for comparison.
- AUC_{p_i} is a set of uncovered branches for all products in $P_{considered}$ that are also common between p_i and those products.
- UCC_{p_i} is a set of uncovered branches in p_i that are common between p_i and one of the products in $P_{considered}$, and are not covered in any product in $P_{considered}$.

Algorithm 5 StaticOrderCalculation ($P_{remaining}$, $CommonMap$, $InitialUncovMap$)

```
P_considered={p_j}
P_remaining=P_remaining-p_j
Ordered= []
while P_remaining is not empty do
  for each product p_i in P_remaining do
    AllUC=[]
    AUC_{p_i}={}
    for each product p_j in P_considered do
      UC_{p_j}=InitialUncovMap.get(p_j)
      Common_{i,j}=CommonMap.get(i, j)
      AUC_{p_i}=AUC_{p_i} ∪ (UC_{p_j} ∩ Common_{i,j})
    end for
    UC_{p_i}=InitialUncovMap.get(p_i)
    UCC_{p_i}= UC_{p_i} ∩ AUC_{p_i}
    AllUC[p_i]=UCC_{p_i}
  end for
  p_n=Select the product with largest size of UC in AllUC
  Ordered.add(p_n)
  P_considered=P_considered ∪ p_n
  P_remaining=P_remaining - p_n
end while
return Ordered
```

The algorithm begins with a randomly selected product p_j, inserts it into $P_{considered}$, and removes it from $P_{remaining}$. For each product p_i in $P_{remaining}$, the inner `for` loop is used to calculate all the branches that (1) are common between product p_i and products in $P_{considered}$ and (2) are not covered by initial test suites for products in $P_{considered}$, and save them in set AUC_{p_i}. All branches in AUC_{p_i} that are also not covered in product p_i are then placed in set UCC_{p_i}. After this has been done for all products in $P_{remaining}$, the algorithm selects the product that has the largest size of UCC and appends it to the $Ordered$ list. Then this product is removed from $P_{remaining}$ and added to $P_{c}onsidered$, and the loop iterates.

By selecting this order, we hope to cover common branches as quickly as possible, since this is where we can achieve savings.

Algorithm 6 DynamicOrderCalculation ($P_{remaining}$, $P_{considered}$, $CommonMap$, $CovMap$, $InitialUncovMap$)

```
AllU=[]
for each product p_i in P_remaining do
  ACC_{p_i}={}
  for each product p_j in P_considered do
    C_{p_j}=CovMap.get(p_j)
    Common_{i,j}=CommonMap.get(i,j)
    ACC_{p_i}=ACC_{p_i} ∪ (C_{p_j} ∩ Common_{i,j})
  end for
  UC_{p_i}=InitialUncovMap.get(p_i)
  UC'_{p_i}= U_{p_i} ∩ ACC_{p_i}
  AllU[p_i]=UC'_{p_i}
end for
p_n=Find the product with smallest size of UC' in AllU
return p_n
```

7.2.2.2 A Dynamic Order

Our dynamic algorithm, which uses a dynamic order, is called after each attempt at test case generation, rather than just once initially. In this case, our goal is to select a next product to consider, from the remaining products, that has the smallest size of UC'. Algorithm 6 presents the algorithm for computing this order. The algorithm uses several variables that are not defined above and we define them as follows:

- $CovMap$ is a map containing covered branches of each product in $P_{considered}$.
- $AllU$ is a list used to save all the computed information for products in $P_{considered}$ in order to select the next product.
- ACC_{p_i} is a set of branches covered by products in $P_{considered}$ and also common between p_i and those products.
- C_{p_j} is a set of branches covered in p_j after test case generation has been performed for p_j.

Given a set of products that have been considered, the dynamic ordering algorithm iterates through the remaining products to update their current coverage information. The inner `for` loop first finds the branches covered by p_j and unions these with the branches common between p_i and p_j. This information is saved to ACC_{p_i}. After this calculation has been completed for each product in $P_{considered}$, the algorithm excludes the branches in ACC_{p_i} from UC_{p_i} to form UC'_{p_i}. Then, it selects the product that has the smallest size of UC' as the next product. Since we call this algorithm after each attempt at test case generation, the algorithm can be simplified to let $P_{considered}$ contain only the product that just has been considered.

This order lets us always select the product that has the fewest uncovered branches so far. In this manner, we gradually build up the coverage for the entire product line.

The calculation of the ordering requires us to evaluate all pair-wise products. Note, however, that much of this calculation can be done in the preliminary period of testing, prior to the time at which the product nears release and time for testing is critical. Furthermore, the algorithm operates quickly; an earlier implementation was capable of processing 50,000 lines of code in under two minutes [76]. Still, the approach may be combinatorially infeasible on larger systems. We believe that we can relax our analysis by sampling the products with techniques such as pairwise testing [25, 64], but leave this extension as future work.

7.2.3 Generating Test Cases

After CONTESA has identified targets and chosen a next product P_j to test, it uses a test case generation technique to augment the test suite for P_j by generating test cases for targets in P_j. There are many test case generation techniques that could be employed for such a purpose. For example, we can use a random approach, but

this may not work well for large and complex programs since it does not target specific branches. More sophisticated techniques include dynamic symbolic execution (e.g., [18, 39, 81]), and evolutionary or search-based approaches (e.g., [7, 66]) such as genetic algorithms. In this work we use a genetic test case generation algorithm (described in the background), that we have used in prior augmentation studies [100, 99].

7.3 Empirical Study 1

To evaluate CONTESA we conducted two empirical studies. In the first study, we are interested in whether our continuous test suite augmentation approach, CONTESA which is a family-based approach to testing product lines is more effective and efficient than a product-based process in which test cases are generated independently for each product. We also wish to determine whether the order used in our continuous test suite augmentation approach matters. We thus pose the following research questions.

RQ1: Is continuous test suite augmentation more effective and efficient than generating test cases independently for each product?

RQ2: Does the order used in continuous test suite augmentation matter in terms of effectiveness and efficiency?

7.3.1 Objects of Analysis

To investigate these questions we selected two software product lines developed by other researchers and used in prior studies. The first SPL is a Graph Product Line (GPL) created by Lopez-Herrejon and Batory [53]; it is built using the AHEAD methodology and implemented as a series of .jak files [8]. GPL has 38 products and

1435 lines of code across all of its .jak files. The second SPL is a portion of the AHEAD tool suite called Bali; it is also built using the AHEAD methodology and implemented as a series of .jak files. Bali has 8 products, but we excluded two that involve using a GUI. This left six products containing 11811 lines of code across all of the .jak files.

7.3.2 Variables and Measures

Independent Variable. Our independent variable is the testing technique utilized. We considered three techniques: generating test cases for products independently (B), continuous test suite augmentation using the static order discussed in Section 7.2.2.1 (C_s), and continuous test suite augmentation using the dynamic order discussed in Section 7.2.2.2 (C_d). Note that C_d and C_d differ only in the order used.

Dependent Variables. We measure the efficiency and effectiveness of each of the techniques.

DV1: Efficiency. To track technique efficiency, for each application of a technique we measure the amount of wall clock time required to apply the technique to each product, in seconds.

DV2: Effectiveness. The techniques that we consider are intended to achieve higher levels of coverage of each product. To measure the effectiveness of techniques, we track the number of branches designated as covered in each product after the technique has run to completion.

7.3.3 Experiment Setup and Operation

To establish the experiment setup needed to conduct our experiment we first tuned the genetic algorithm. We used the same genetic algorithm described in Section 4.3.2

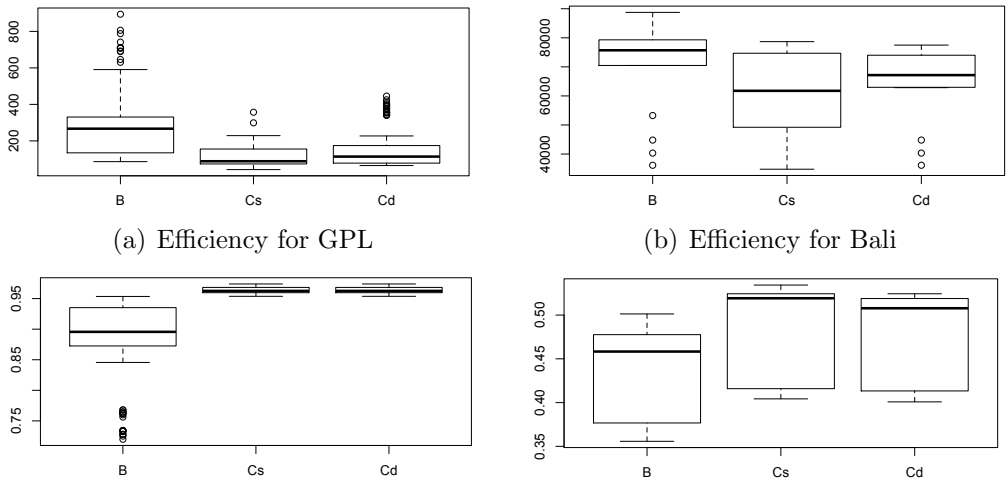
in terms of the fitness function, the selection approach and the crossover approach. As a mutation rate we used 0.3 for GPL and 0.5 for Bali. We set the iteration limit to 5 for GPL and 15 for Bali. The chromosome (representation of inputs) for GPL is an object, with a number of nodes, edges and weights for each of the edges. We allow 0..9 as an integer range for nodes, and we can have 0 to 30 edges. The weights can be any value from 0 to 50. We do not use crossover for GPL. During mutation if we delete nodes, we then delete any associated edges for those nodes during a repair phase.

For Bali the chromosome is an array of 0 to 4 options (this is variable) followed by all of the characters from 0 to 2 input files. These are grammar input files that are used by Bali. During mutation we randomly modify characters or flip options. During mutation and crossover we do not check to ensure that we maintain a valid grammar.

Since there is some randomness inherent in the use of genetic algorithms, we applied each technique to each of our software product lines three times. All of our data was gathered on a parallel cluster with AMD 6128 2GHz, Quad-Processors (8-Core) and 128GB RAM on each core.

7.3.4 Threats to Validity

The primary threat to *external validity* for this study involves the representativeness of our software product lines. We have examined only two product lines and the study of others may exhibit different cost-benefit tradeoffs. However, the product lines we used do present two different variants of product lines, with GPL being smaller but widely used, and Bali larger and more complex. A second threat to external validity pertains to our algorithms; we have utilized only one variant of a genetic test case



(a) Efficiency for GPL

(b) Efficiency for Bali

(c) Effectiveness for GPL

(d) Effectiveness for Bali

Figure 7.2: Efficiency and effectiveness for GPL and Bali

generation algorithm. Subsequent studies are needed to determine the extent to which our results generalize.

The primary threat to *internal validity* is possible faults in the implementation of the algorithms and in tools we use to perform evaluation. We controlled for this through functional testing. A second threat involves decisions and practices in the implementation of the techniques studied; for example, variation in the efficiency of implementations of techniques could bias data collected.

Where *construct validity* is concerned, there are other metrics that could be pertinent to the effects studied. In particular, our measurement of efficiency considers only technique run-time, and omits costs related to the time spent by engineers employing the approaches.

7.3.5 Results

Figure 7.2 present the results obtained in our study for both software product lines with respect to efficiency and effectiveness, respectively. The two sets of boxplots on

the left, Figures 7.2(a) and 7.2(c) show data pertaining to GPL while the two sets of boxplots on the right, Figures 7.2(b) and 7.2(d), show data pertaining to Bali. Each set of boxplots contains three individual boxplots corresponding to the three techniques, B, C_s and C_d. As mentioned in Section 7.3.2, B represents generating test cases for products independently, C_s represents continuous test suite augmentation using the static order discussed in Section 7.2.2.1, and C_d represents continuous test suite augmentation using the dynamic order discussed in Section 7.2.2.2. The upper two sets of boxplots show data on efficiency while the bottom two sets show data on effectiveness. For efficiency, each boxplot denotes the data obtained on each of the three runs for each product in the associated software product line. For effectiveness, each boxplot denotes the final coverage obtained on each of the three runs for each product in the associated software product line.

7.3.5.1 RQ1: Independent Test Case Generation Versus Continuous Test Suite Augmentation

We begin by comparing C_s and C_d with B for efficiency. Considering the upper two sets of boxplots, it appears that on both SPLs, C_s (static continuous augmentation) and C_d (dynamic continuous augmentation) were more efficient than B (product based augmentation). On GPL, B required 276.9 seconds per product on average, whereas C_s required only 114.1 seconds and C_d required only 151.9 seconds per product on average. On Bali, B required 70,008.2 seconds (19.4 hours) per product on average, whereas C_s required only 60,291.4 seconds (16.7 hours) and C_d required only 64,973.2 seconds (18 hours) per product on average.

To determine whether these observed differences were statistically significant, we applied a Wilcoxon signed rank test to the data at confidence level .05. In all cases, the differences were statistically significant. (On GPL, for B versus C_s the p-value

was 2.2e-16, and for B versus C_d the p-value was 2.245e-15. On Bali, for B versus C_s the p-value was 2.91e-11 and for B versus C_d the p-value was 2.2e-16).

Next, we compare C_s and C_d to B with respect to effectiveness. Considering the two bottom sets of boxplots, it appears that on both software product lines, C_s and C_d were more effective than (achieved greater coverage than) B. On GPL, B achieved 88.3% coverage per product on average, while C_s and C_d each achieved 96.3% coverage per product on average. On Bali, B achieved 43.5% coverage per product on average, while C_s achieved 48.7% and C_d achieved 48.0% coverage per product on average. We note that we are measuring per product coverage (rather than family-based coverage) since we cannot compose the entire code-base to evaluate this.

We again applied a Wilcoxon signed rank test to the data, at confidence level .05. Again, in all cases the differences were both statistically significant. (On GPL, p-values when comparing both C_s and C_d to B were both 2.2e-16. On Bali, for B versus C_s the p-value was 7.629e-06, and for B versus C_d the p-value was 0.001289.)

In summary, on both product lines, continuous test suite augmentation was statistically significantly more efficient and effective than independent test case generation.

7.3.5.2 RQ2: Order Effects in Continuous Test Suite Augmentation

To address this research question, we compare the results of C_s and C_d, which use different product ordering approaches static and dynamic, respectively). First we compare them in terms of efficiency. In Figure 7.2(a), on GPL, C_s appears to be slightly more efficient than C_d, requiring 37.7 seconds less than C_d in average. In Figure 7.2(b), on Bali, C_s also appears to be more efficient than C_d, requiring 4,681.8 seconds less on average. Wilcoxon signed rank tests show that the observed differences are statistically significant, yielding p-values of 4.465e-06 and 2.910e-11 for GPL and Bali, respectively.

Next we compare the two techniques in terms of effectiveness. In the boxplots for GPL (Figure 7.2(c)) we cannot see differences between the two techniques; in fact they each achieved 96.5% average coverage and contain identical sets of data points (hence obviating the need for statistical analysis). On Bali ((Figure 7.2(d))), however, it appears that C_s yields better coverage than C_d, achieving 0.68% more final coverage. More concretely, C_s covers between 3 and 29 more branches than C_d across all products in all three runs. We applied the Wilcoxon signed rank test to the Bali coverage data, obtaining a p-value of 0.0004824, indicating that the difference between the techniques on this SPL was statistically significant.

In summary, Order 1 was statistically significantly more efficient than Order 2 on both product lines, and Order 1 had a statistically significant effectiveness advantage on one of the two product lines.

7.4 Empirical Study 2

In prior work [16] investigating a product-based approach to testing software product lines, we used the Graph Product Line (GPL) utilized in our first empirical study, and used a specification-based test case generation approach to create a test suite for the system based on its feature model, consisting of 18 test cases. These test cases were generated from use cases that followed the specifications for each product. The existence of this test suite and the use of this object in our first study provides a way for us to investigate three additional interesting questions about our approach.

First, we can investigate whether the test cases generated by CONTESA are better than specification-based test cases generated manually in terms of effectiveness. Second, we can compare the coverage results observed for CONTESA in the first study, which are there reported from a family-based perspective, to the results that

would be achieved if the test cases generated by CONTESA were actually all executed (a product-based approach) on each of the products. This second question is interesting because we do not retest already covered code in new products. This question helps us to evaluate the difference between family-based and product-based coverage. Third, because GPL was seeded with a set of 60 faults for our prior study, which are present in various GPL products, we can compare the fault-detection ability of the test cases created by CONTESA with that of the test cases generated manually.

Thus we pose the following research questions:

RQ3 Do the test cases generated by continuous test suite augmentation achieve better branch coverage on each product than specification-based test cases generated manually?

RQ4 How do the coverage values computed by CONTESA compare to coverage values achieved by execution of all test cases on all products?

RQ5 How effective are the test cases generated by CONTESA in terms of fault detection and compared to specification-based test cases generated manually?

7.4.1 Objects of Analysis

Again, we use the Graph Product Line (GPL) SPL. In this case we also utilize the 60 faults available for GPL (described in [16]). Note that to investigate RQ4 Bali will not work, because its test cases function only per product as the different products have different input syntax; thus, we cannot execute all test cases on all products. Further, investigating RQ3 and RQ5 with Bali would require us to have specification-based test cases, and we do not.

7.4.2 Variables and Measures

Independent Variable. Our independent variable is the approach used to generate test cases for GPL: specification-based test case generation (SB), continuous test suite augmentation using the static order mentioned in Section7.2.2.1 (C_sE), and continuous test suite augmentation using the static order, while also using coverage information gathered from actually running test cases from C_sE (C_sA). For SB we use the previously existing test cases.

Dependent Variables. For RQ3 and RQ4, We measure the effectiveness of the test cases generated by SB and C_sA. To do this we track the number of covered branches in each product after running the test cases generated by the two approaches. We also track (as in Study 1) the number of branches reported as covered by the CONTESA approach, using results for C_sE generated for the previous study. For RQ5, we measure the faults detected by test cases generated by SB and C_sA.

7.4.3 Experiment Setup and Operation

We used the same experiment setup and operation parameters utilized in the first study (see Section 7.3.3).

7.4.4 Threats to Validity

This study shares threats to validity with the first study (Section 7.3.4); in addition it utilizes only one software product line as a subject, which further limits external validity.

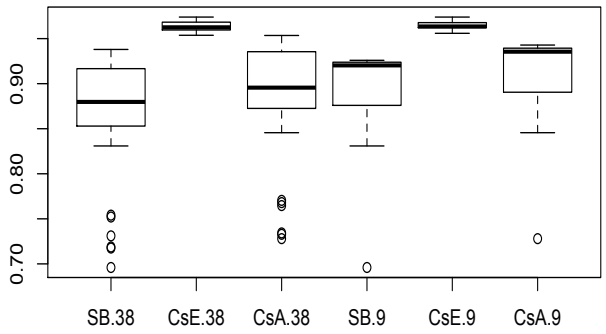


Figure 7.3: Effectiveness achieved by test suites from SB, calculated by C_sE and when run C_sA

7.4.5 Results

Figure 7.3 presents the results obtained in our study, with respect to research questions RQ3 and RQ4. The figure displays six boxplots. The leftmost contains data obtained by executing test cases from specication-based test case generation on all 38 products. The next one shows the coverage that is accumulated via the CONTESA process (data repeated from Figure 7.2). The third from left is the result of running all of the CONTESA test cases on the 38 GPL products. In [16], we used a feature model based approach called Fig Basis Path to reduce the number of products to be tested to 9; the right most three boxplots show the data obtained by the three testing approaches on just those nine products.

7.4.5.1 RQ3: Coverage Achieved by Continuous Versus Specification-Based Test Case Generation

Figure 7.3 shows that in both cases (with 38 or 9 products), the test cases created by CONTESA achieved greater actual coverage when executed on all software product lines. When running all test cases on all 38 products, SB achieved 86.7% coverage

while running all test cases C_sA on them achieved 88.5%. When running only 9 products, SB achieved 88.1% coverage while C_sA achieved 90.0%.

7.4.5.2 RQ4: Coverage Achieved During Execution Versus Coverage Calculated by CONTESA

Figure 7.3 also shows that in both cases (38 or 9 products), C_sE achieved higher levels of coverage than C_sA. Average coverage was 96.4% for C_sE and 88.5% for C_sA on all products and average coverage was 96.5% for C_sE and 88.5% for C_sA on 9 products. The coverage calculated by CONTESA, using its family-based approach, is greater than that achieved when test cases are run on each product using the product-based approach. We believe that this reflects differences in the coverage that can be attained in different products due to code that is non-executable under particular product configurations, or code whose execution is masked by the presence of certain features. As such this is indicative of differences between family-based and product-based testing approaches at the code coverage level.

7.4.5.3 RQ5: Faults Detected by CONTESA Versus Faults Detected by Specification-Based Test Case Generation

The test cases generated by CONTESA detected 41 faults while the test suite used in [16] was reported to detect 54 faults. We have investigated that result further and found that the two studies use different definitions of faults. There are 12“ faults” detected by the test cases used in [16] that are simple syntax errors. If we count those as faults, our test suite would detect 53 faults, one fewer. In other words, the test cases from CONTESA are almost as effective as the test cases generated by humans in fault detection.

7.5 Discussion

We now provide additional insights into the results of our studies.

7.5.1 Overall Expense

The boxplots presented in this paper represent data points relative to single products. Continuous test suite augmentation is a process that is applied to an entire set of products, so for practical purposes we are also interested in how expensive this overall process is. To compare this approach to the baseline approach (B) that independently generates test cases for all products, we also measured the time used to finish considering all products. In the case of our study, for GPL, an end-to-end application of continuous test suite augmentation across the entire set of products on average across our three runs required 4337.6 seconds (1.2 hours) using the static ordering technique (C_s) and 5771.3 seconds (1.6 hours) using the dynamic ordering technique (C_d), while technique B required 10523.5 seconds (2.9 hours). For Bali, an end-to-end application of continuous test suite augmentation across the entire set of products on average across our three runs required 361,748.7 seconds (4.2 days) using technique C_s and 389,839.3 seconds (4.5 days) using technique C_d, while technique B required 393,443.7 seconds (4.6 days).

7.5.2 Continuous Effectiveness Change

In continuous test suite augmentation, after finishing test case generation for a given product, we update the targets for remaining products, and update the coverage achieved. Thus, coverage grows cumulatively, and it is interesting to see the effects of that growth. Table 7.1 presents coverage data from Bali after each application of augmentation to a specific product for techniques C_s (static) and C_d (dynamic)

Table 7.1: Cumulative Effectiveness on Bali Products at Each Stage of Augmentation

Stage	P1		P2		P3		P4		P5		P6	
	C_s	C_d	C_s	C_d	C_s	C_d	C_s	C_d	C_s	C_d	C_s	C_d
1	42.2	42.2	42.6	42.6	45.4	45.4	38.0	38.0	46.5	46.5	40.0	40.0
2	42.5	42.5	43.0	43.0	46.2	46.2	39.5	39.5	46.6	46.6	40.5	40.5
3	**49.0**	47.0	**49.3**	47.5	**50.2**	49.7	39.6	40.3	**50.4**	50.0	40.6	41.4
4	**50.6**	48.4	**51.0**	48.8	**51.6**	50.6	39.8	40.4	**51.8**	50.8	40.8	41.4
5	**51.2**	50.3	**51.6**	50.7	**52.6**	51.7	**40.6**	40.4	**52.7**	51.9	**41.6**	41.4
6	**51.6**	50.6	**52.0**	51.0	**52.9**	52.0	**40.7**	40.5	**53.1**	52.2	**41.7**	41.5

on Bali. Each row represents the (cumulative) coverage achieved across all products after a run of augmentation on a particular product. (The particular product is listed in the leftmost column, denoted by "Stage", coverage data achieved for each other product following that stage appears in the twelve columns that follow, per product and technique). The data in each cell is the average coverage (percentage of branches covered relative to the number of total branches) achieved across the three runs of our techniques. Bold numbers indicate, at each stage for each product, cases in which one technique has achieved better cumulative coverage than the other. There are 20 cases in which C_s has achieved better coverage, and once that advantage is achieved it is retained; there are no cases in which C_d achieves higher coverage. We applied the Wilcoxon signed rank test to this data to compare C_s to C_d; results showed that the two were statistically significantly different (p-value 2.2e-16).

For GPL, there are 38*38 = 1444 data points with each point representing the average coverage of three runs, so for reasons of space we do not show a similar table for it. However, we note that among those data points, C_s was more effective than C_d in 312 cases while C_d was more effective than C_s in 104 cases. We also applied the Wilcoxon signed rank test to the data to compare C_s to C_d; again the two were statistically significantly different (p-value 7.636e-12).

7.6 Conclusions and Future Work

We have presented CONTESA, a continuous test suite augmentation process for testing software product lines. CONTESA approaches testing from a product-family perspective, generating and running tests on products only to covered previously uncovered code (determined through a regression testing static analysis). We evaluated CONTESA on two software product lines and observed that a family-based approach is more effective and efficient than a per-product approach, yielding both higher coverage and a shorter run time. When comparing CONTESA with a product-based testing approach that uses specification based tests, we see higher code coverage, but lower fault detection. We also observed that the coverage obtained if we were to actually run the test cases was lower than the coverage that is estimated through our analysis.

Chapter 8

Conclusion and Future Work

In this thesis, we have presented several test suite augmentation techniques for use in regression testing. We have also presented a test suite augmentation framework and studied factors that affect the cost-effectiveness of the test suite augmentation process, including test case generation techniques, the order of targets and test reuse approaches. We have shown that the last two factors have different impacts on the first factor. We have developed a hybrid test suite augmentation technique by combining test case generation techniques to further improve performance. Finally, extending the generality of the work, we have developed a test suite augmentation technique for use on software product lines, CONTESA.

This research has made the following contributions, for both researchers and practitioners:

1. Brought the notion of test reuse into test suite augmentation for regression testing.

2. Identified several factors that impact the cost-effectiveness of the augmentation process.

3. Provided researchers with new insights into the test suite augmentation process.

4. Developed a framework for instantiating test suite augmentation techniques.

5. Used our framework to instantiate several techniques, including a hybrid technique, and enabled them to work effectively and efficiently.

6. Introduced the test suite augmentation idea into the SPL testing process and identified effective orders for augmenting test suites for products in an SPL.

In this work we have focused on test suite augmentation, and our results have several implications for the creation and further study of augmentation techniques. The results also have implications, however, for engineers creating initial test suites for programs. This is because such engineers often begin, at least at the system test level, with black box requirements-based test cases. It has long been recommended that such test suites be extended to provide some level of coverage. The techniques we have presented can conceivably serve in this context too, working with initial black-box test cases and augmenting these.

In future work we intend to improve our hybrid technique by following lessons learned from our results, and study its application to additional and larger object programs. As further potential improvements we will also seek ways in which the individual test case generation algorithms used by the hybrid technique can make use of additional information gathered by the other algorithms to generate test cases more cost-effectively. On SPL testing, future work includes running CONTESA on larger product lines, and evaluating a sampling approach to reduce the combinatorics of our pair-wise static analysis. We will also further investigate why we see lower code coverage when tests are actually run on products (in a product-based fashion) than

when coverage is calculated by CONTESA (in a family-based fashion). Finally, we plan to extend CONTESA to work with alternative test case generation techniques.

Bibliography

[1] A.V. Aho, M. Lam, R. Sethi, and J.D. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley, Boston, MA, 2nd edition, 2007.

[2] T. Apiwattanapong, R. Santelices, P. Chittimalli, A. Orso, and M. Harrold. Matrix: Maintenance-oriented testing requirements identifier and examiner. In *Proceedings of the Testing: Academic Industrial Conference on Practice And Research Techniques*, pages 137–146, 2006.

[3] A. Arcuri. It does matter how you normalise the branch distance in search based software testing. In *International Conference on Software Testing, Verification, and Validation*, pages 205–214, 2010.

[4] A. Arcuri, Z. Iqbal, and L. Briand. Formal analysis of the effectiveness and predictability of random testing. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 219–229, July 2010.

[5] S. Artzi, A. Kiezun, J. Dolby, F. Tip, D. Dig, A. Paradkar, and M. Ernst. Finding bugs in dynamic web applications. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 261–272, 2008.

[6] A. Avritzer and E. Weyuker. The automatic generation of load test suites and the assessment of the resulting software. *IEEE Transaction Software Engineering*, 21(9):705–716, September 1995.

[7] A. Baresel, D. Binkley, M. Harman, and B. Korel. Evolutionary testing in the presence of loop-assigned flags: a testability transformation approach. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 108–118, 2004.

[8] D. Batory, J. Neal Sarvela, and A. Rauschmayer. Scaling step-wise refinement. In *Proceedings of the International Conference on Software Engineering*, pages 187–197, 2003.

[9] A. Bertolino, A. Fantechi, S. Gnesi, and G. Lami. Product line use cases: Scenario-based specification and testing of requirements. In *Software Product Lines - Research Issues in Engineering and Management*, pages 425–445. Springer-Verlag, 2006.

[10] A. Bertolino and S. Gnesi. PLUTO: A test methodology for product families. In *Lecture Notes in Computer Science. 3014*, pages 181–197, 2004.

[11] D. Binkley. Semantics guided regression test cost reduction. *IEEE Transaction on Software Engineering*, 23(8):498–516, August 1997.

[12] D. Bird and C. Munoz. Automatic generation of random self-checking test cases. *IBM Systems Journal*, 22(3):229–245, 1983.

[13] S. Bohner and R. Arnold. *Software Change Impact Analysis*. IEEE Computer Society Press, Los Alamitos, CA, 1996.

[14] L. C. Briand, Y. Labiche, and S. He. Automating regression test selection based on UML designs. *Information and Software Technology*, 51:16–30, January 2009.

[15] J. Burnim and K. Sen. Heuristics for scalable dynamic test generation. In *Proceedings of the International Conference on Automated Software Engineering*, pages 443–446, September 2008.

[16] I. Cabral, M. B. Cohen, and G. Rothermel. Improving the testing and testability of software product lines. In *International Conference on Software Product Lines*, pages 241–255, 2010.

[17] C. Cadar, D. Dunbar, and D. Engler. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the USENIX conference on Operating systems design and implementation*, pages 209–224, 2008.

[18] C. Cadar, V. Ganesh, P. Pawlowski, D. Dill, and D. Engler. Exe: Automatically generating inputs of death. In *Proceedings of the ACM conference on Computer and Communications Security*, pages 322–335, Oct 2006.

[19] J. Chang and D. Richardson. Structural specification-based testing: Automated support and experimental evaluation. In *Proceedings of the International Symposium on Foundations of Software Engineering*, pages 285–302, September 1999.

[20] T. Y. Chen and R. Merkel. Quasi-random testing. *IEEE Transactions on Reliability*, 56(3):562–568, 2007.

[21] Y. Chen, D. Rosenblum, and K. Vo. TestTube: A system for selective regression testing. In *Proceedings of the International Conference on Software Engineering*, pages 211–220, May 1994.

[22] L. Clarke. A system to generate test data and symbolically execute programs. *IEEE Transactions on Software Engineering*, 2(3):215–222, September 1976.

[23] L. Clarke and D. Richardson. Applications of symbolic evaluation. *Journal of Systems and Software*, 5(1):15–35, January 1985.

[24] P. Clements and L. Northrup. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2002.

[25] M. B. Cohen, M. B. Dwyer, and J. Shi. Coverage and adequacy in software product line testing. In *Workshop on the Role of Architecture for Testing and Analysis*, pages 53–63, July 2006.

[26] CREST - automatic test generation tool for C. `http://code.google.com/p/crest/`.

[27] P. A. da Mota Silveira N., I. Do. Carmo M., Y. C Cavalcanti, E. S. de Almeida, V. C. Garcia, and S. R. de Lemos Meira. A regression testing approach for software product lines architectures. In *Brazilian Symposium on Software Components, Architectures and Reuse*, pages 41–50, September 2010.

[28] R. DeMillo and A. Offutt. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, September 1991.

[29] C. Denger and R. Kolb. Testing and inspecting reusable product line components: First empirical results. In *International Symposium on Empirical Software Engineering*, pages 184–193, 2006.

[30] E. Díaz, J. Tuya, R. Blanco, and J. J. Dolado. A tabu search algorithm for structural software testing. *Journal of Computers and Operations Research*, 35(10):3052–3072, 2008.

[31] H. Do, S. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering*, 10(4):405–435, October 2005.

[32] S. Dowdy, S. Wearden, and D. Chilko. *Statistics for Research, 3rd Edition*. Wiley, 2004.

[33] B. Dutertre and L. de Moura. The Yices SMT solver. http://yices.csl.sri.com/tool-paper.pdf, Aug 2006.

[34] S. Elbaum, A.G. Malishevsky, and G. Rothermel. Test case prioritization: A family of empirical studies. *IEEE Transactions on Software Engineering*, 28(2):159–182, 2002.

[35] M. Emmi and K. Majumdar, R.and Sen. Dynamic test input generation for database applications. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 151–162, 2007.

[36] L. Etxeberria and G. Sagardui. Quality assessment in software product lines. In *Proceedings of the International Conference on Software Reuse: High Confidence Software Reuse in Large Systems*, pages 178–181, 2008.

[37] R. Ferguson and B. Korel. The chaining approach for software test data generation. *ACM Transactions on Software Engineering and Methodology*, 5(1):63–86, January 1996.

[38] G. Fraser and A. Arcuri. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the International Symposium on Foundations of Software Engineering*, pages 416–419, 2011.

[39] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 213–223, June 2005.

[40] A. Gotlieb, B. Botella, and M. Reuher. Automatic test data generation using constraint solving techniques. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 53–62, March 1998.

[41] R. Gupta, M. J. Harrold, and M. L. Soffa. Program slicing-based regression testing techniques. *Journal of Software Testing, Verification and Reliability*, 6:83–112, 1996.

[42] A. Hartman and K. Nagin. Model driven testing - agedis architecture interfaces and tools. In *Proceedings of the European Conference on Model Driven Software Engineering*, pages 1–11, December 2003.

[43] W. Heider, P.l Rabiser, R.and Grünbacher, and D. Lettner. Using regression testing to analyze the impact of changes to variability models on products. In *International Conference on Software Product Lines*, pages 196–205, 2012.

[44] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proceedings of the International Conference on Software Engineering*, pages 191–200, May 1994.

[45] K. Inkumsah and T. Xie. Improving structural testing of object-oriented programs via integrating evolutionary testing and symbolic execution. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, pages 297–306, 2008.

[46] http://javapathfinder.sourceforge.net/.

[47] C. Kim, D. S. Batory, and S. Khurshid. Reducing combinatorics in testing product lines. In *International Conference on Aspect-Oriented Software Development*, pages 57–68, 2011.

[48] A. Kinneer, M. Dwyer, and G. Rothermel. Sofya: A flexible framework for development of dynamic program analysis for Java software. Technical Report TR-UNL-CSE-2006-0006, University of Nebraska - Lincoln, April 2006.

[49] B. Korel. Automated software test data generation. *IEEE Transactions on Software Engineering*, 16(8):870–897, August 1990.

[50] K. Lakhotia, M. Harman, and H. Gross. Austin: A tool for search based software testing for the c language and its evaluation on deployed automotive systems. In *Proceedings of the International Symposium on Search Based Software Engineering*, pages 101–110, 2010.

[51] Z. Li, M. Harman, and R. Hierons. Search algorithms for regression test case prioritization. *IEEE Transactions on Software Engineering*, 33(4):225–237, April 2007.

[52] M. Lochau, I. Schaefer, J. Kamischke, and S. Lity. Incremental model-based testing of delta-oriented software product lines. In *Tests and Proofs*, volume

7305 of *Lecture Notes in Computer Science*, pages 67–82. Springer Berlin Heidelberg, 2012.

[53] R. E. Lopez-Herrejon and D. S. Batory. A standard problem for evaluating product-line methodologies. In *Proceedings of the International Conference on Generative and Component-Based Software Engineering*, pages 10–24, 2001.

[54] R Majumdar and K. Sen. Hybrid concolic testing. In *Proceedings of the International Conference on Software Engineering*, pages 416–426, 2007.

[55] D. Marinov and S. Khurshid. TestEra: A novel framework for automated testing of Java programs. In *Proceedings of the International Conference on Automated Software Engineering*, pages 22–31, November 2001.

[56] P. McMinn. Search-based software test data generation: a survey: Research articles. *Journal of Software Testing, Verification, and Reliability*, 14(2):105–156, 2004.

[57] P. McMinn, D. Binkley, and M. Harman. Empirical evaluation of a nesting testability transformation for evolutionary testing. *ACM Transactions on Software Engineering and Methodology*, 18:11:1–11:27, June 2009.

[58] C. Michael, G. McGraw, and M. Shatz. Generating software test data by evolution. *IEEE Transactions on Software Engineering*, 27(12):1085–1110, December 2001.

[59] S. Montagud and S. Abrahão. Gathering current knowledge about quality evaluation in software product lines. In *Proceedings of the International Software Product Line Conference*, pages 91–100, 2009.

[60] C. Nebut, F. Fleurey, Y. L. Traon, and J. Jzquel. A requirement-based approach to test product families. In *Proceedings of the Workshop Product Families Engineering*, pages 198–210, 2003.

[61] J. Offutt and A. Abdurazik. Generating tests from UML specifications. In *Proceedings of the International Conference on the Unified Modeling Language*, October 1999.

[62] E. M. Olimpiew and H. Gomaa. Reusable model-based testing. In *Proceedings of the International Conference on Software Reuse: Formal Foundations of Reuse and Domain Engineering*, pages 76–85, 2009.

[63] A. Orso, N. Shi, and M. J. Harrold. Scaling regression testing to large software systems. In *Proceedings of the International Symposium on Foundations of Software Engineering*, November 2004.

[64] S. Oster, M. Zink, M. Lochau, and M. Grechanik. Pairwise feature-interaction testing for SPLs: potentials and limitations. In *International Conference on Software Product Lines*, pages 6:1–6:8, 2011.

[65] H. Pande, W. Landi, and B.G. Ryder. Interprocedural def-use associations in C programs. *IEEE Transactions on Software Engineering*, 20(5):385–403, May 1994.

[66] R. P. Pargas, M. J. Harrold, and R. R. Peck. Test-data generation using genetic algorithms. *Journal of Software Testing, Verification and Reliability*, 9:263–282, September 1999.

[67] S. Person, M. Dwyer, S. Elbaum, and C. Păsăreanu. Differential symbolic execution. In *Proceedings of the International Symposium on Foundations of Software Engineering*, pages 226–237, November 2008.

[68] S. Person, G. Yang, N. Rungta, and S. Khurshid. Directed incremental symbolic execution. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2011.

[69] O. Pilskalns, G. Uyan, and A. Andrews. Regression testing UML designs. In *Proceedings of the 22nd IEEE International Conference on Software Maintenance*, pages 254–264, 2006.

[70] S. Reis, A. Metzger, , and K. Pohl. Integration testing in software product line engineering: A model-based technique. In *Proceedings of the International Conference on Fundamental Approaches to Software Engineering*, pages 321–335, 2007.

[71] S. Reis, A. Metzger, and K. Pohl. A reuse technique for performance testing of software product lines. In *Proceedings of the International Workshop on Software Product Line Testing*, 2006.

[72] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley. Chianti: A tool for change impact analysis of Java programs. In *Proceedings of the ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 432–448, October 2004.

[73] A. Reuys, E. Kamsties, K. Pohl, and S. Reis. Model-based system testing of software product families. In *Proceedings of the International Conference on Advanced Information Systems Engineering*, pages 519–534, 2005.

[74] G. Rothermel and M. J. Harrold. Selecting tests and identifying test coverage requirements for modified software. In *Proceedings of the International Symposium on Software Testing and Analysis*, Aug 1994.

[75] G. Rothermel and M. J. Harrold. Analyzing regression test selection techniques. *IEEE Transactions on Software Engineering*, 22(8):529–551, August 1996.

[76] G. Rothermel and M. J. Harrold. A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering and Methodology*, 6(2):173–210, April 1997.

[77] M. Ruth, S. Oh, A. Loup, B. Horton, O. Gallet, M Mata, and S. Tu. Towards automatic regression test selection for web services. In *Proceedings of the International Computer Software and Applications Conference*, pages 729–736, August 2007.

[78] R. Santelices, P. K. Chittimalli, T. Apiwattanapong, A. Orso, and M. J. Harrold. Test-suite augmentation for evolving software. In *Proceedings of the International Conference on Automated Software Engineering*, September 2008.

[79] A. Schürr, S. Oster, and F. Markert. Model-driven software product line testing: An integrated approach. In *Theory and Practice of Computer Science*, pages 112–131, 2010.

[80] K. Sen and G. Agha. CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools. In *Proceedings of the International conference on Computer Aided Verification*, pages 419–423, Aug 2006.

[81] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *Proceedings of the International Symposium on Foundations of Software Engineering*, pages 263–272, Sept. 2005.

[82] J. Shi, M. Cohen, and M. Dwyer. Integration testing of software product lines using compositional symbolic execution. In *International Conference on Fundamental Approaches to Software Engineering*, pages 270–284. Springer, 2012.

[83] S. Sinha, M. J. Harrold, and G. Rothermel. Computation of interprocedural control dependencies. *ACM Transactions on Software Engineering and Methodology*, 10(2):209–254, April 2001.

[84] http://csce.unl.edu/∼galileo/pub/sofya.

[85] H. Sthamer. *The Automatic Generation of Software Test Data Using Genetic Algorithms*. PhD thesis, University of Glamorgan, Pontyprid, Wales, UK, April 1996.

[86] K. Taneja, T. Xie, N. Tillmann, J. Halleux, and W. Schulte. eXpress: Guided path exploration for regression test generation. In *Proceedings of the International Symposium on Software Testing and Analysis*, July 2011.

[87] T. Thum, S. Apel, C. Kästner, M. Kuhlemann, I Schaefer, and G. Saake. Analysis strategies for software product lines. Technical report, Faukulät für Informatik, Otto-von-Guericke-Univerität Magdeburg, April 2012.

[88] P. Tonella. Evolutionary testing of classes. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 119–128, 2004.

[89] E. Uzuncaova, D. Garcia, S. Khurshid, and D. Batory. Testing software product lines using incremental test generation. In *Proceedings of the International Symposium on Software Reliability Engineering*, pages 249–258, 2008.

[90] R. Vallée-Rai. Soot: A Java Bytecode Optimization Framework. Master's thesis, McGill University, 2000.

[91] F. van der Linden. Software product families in europe: the esaps and cafe projects. *IEEE Software*, 19(4):41–49, July 2002.

[92] W. Visser, C. Pasareanu, and S. Khurshid. Test input generation with Java Pathfinder. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 97–107, July 2004.

[93] H. Waeselynck, P. Thévenod-Fosse, and O. Abdellatif-Kaddour. Simulated annealing applied to test generation: Landscape characterization and stopping criteria. *Empirical Software Engineering*, 12(1):35–63, 2007.

[94] A. Walcott, M. L. Soffa, G. M. Kapfhammer, and R. S. Roos. Time-aware test suite prioritization. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 1–12, July 2006.

[95] S. Wappler and F. Lammermann. Using evolutionary algorithms for the unit testing of object-oriented software. In *Proceedings of the Conference on Genetic and Evolutionary Computation*, pages 1053–1060, 2005.

[96] G. Wassermann, D. Yu, D. Chander, A.and Dhurjati, H. Inamura, and Z. Su. Dynamic test input generation for web applications. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 249–260, 2008.

[97] J. Wegener, A. Baresel, and H. Sthamer. Evolutionary test environment for automatic structural testing. *Information and Software Technology*, 43(14):841 – 854, 2001.

[98] Z. Xu, M. B. Cohen, W. Motycka, and G. Rothermel. Continuous test suite augmentation in software product lines. In *Proceedings of the International Software Product Line Conference*, 2013.

[99] Z. Xu, M. B. Cohen, and G. Rothermel. Factors affectnig the use of genetic algorithms in test suite augmentation. In *Proceedings of the Genetic and Evolutionary Computation Conference*, July 2010.

[100] Z. Xu, Y. Kim, M. Kim, G. Rothermel, and M. Cohen. Directed test suite augmentation: Techniques and tradeoffs. In *Proceedings of the ACM SIGSOFT Symposium on Foundations of Software Engineering*, November 2010.

[101] Z. Xu, Y. Kim, Kim. M, and G. Rothermel. A hybrid directed test suite augmentation technique. In *International Symposium on Software Reliability Engineering*, pages 150–159, 2011.

[102] Z. Xu and G. Rothermel. Directed test suite augmentation. In *Proceedings of the Asia-Pacific Software Engineering Conference*, pages 406–413, December 2009.

[103] S. Yoo and M. Harman. Pareto efficient multi-objective test case selection. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 140–150, July 2007.

[104] S. Yoo and M. Harman. Test data augmentation: Generating new test data from existing test data. Technical Report TR-08-04, Dept. of Computer Science, King's College London, July 2008.