AUTOMATED REGRESSION TESTING AND VERIFICATION OF COMPLEX CODE CHANGES

DOCTORAL THESIS

MARCEL BÖHME

NATIONAL UNIVERSITY OF SINGAPORE

2014

AUTOMATED REGRESSION TESTING AND VERIFICATION OF COMPLEX CODE CHANGES

MARCEL BÖHME
(*Dipl.-Inf., TU Dresden, Germany*)

A THESIS SUBMITTED
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY
DEPARTMENT OF COMPUTER SCIENCE, SCHOOL OF COMPUTING
NATIONAL UNIVERSITY OF SINGAPORE
2014

To my father.

Declaration

I hereby declare that this thesis is my original work and it has been written by me in its entirety. I have duly acknowledged all the sources of information which have been used in the thesis. This thesis has also not been submitted for any degree in any university previously.

Marcel Böhme (June 30, 2014)

Name	:	Marcel Böhme
Degree	:	Doctor of Philosophy
Supervisor(s)	:	Abhik Roychoudhury
Department	:	Department of Computer Science, School of Computing
Thesis Title	:	Automated Regression Testing and Verification of Complex Code Changes

Abstract

How can we check software changes effectively? During software development and maintenance, the source code of a program is constantly changed. New features are added and bugs are fixed. However, not always are the semantic, behavioral changes that result from the syntactic, source code changes as intended. Existing program functionality that used to work may not work anymore. The result of such unintended semantic changes is software regression. Given the set of syntactic changes, the aim of automated regression test generation is to create a test suite that stresses much of the semantic changes so as to expose any potential software regression.

In this dissertation we put forward the following thesis: *A complex source code change can only be checked effectively by accounting for the interaction among its constitutent changes.* In other words, it is insufficient to exercise each constitutent change individually. This poses a challenge to automated regression test generation techniques as well as to traditional predictors of the effectiveness of regression test suites, such as code coverage. We claim that a regression test suite with a high coverage of individual code elements may not be very effective, *per se*. Instead, it should also have a high coverage of the inter-dependencies among the changed code elements.

We present two automated test generation techniques that can expose realistic regression errors introduced with complex software changes. *Partition-based Regression Verification* directly explores the semantic changes that result from the syntactic changes. By exploring the semantic changes, it also accounts for interaction among the syntactic changes. Specifically, the input space of both program versions can be partitioned into groups of input revealing an output difference and groups of input computing the same output in both versions. Then, these partitions can be explored in an automated fashion, generating one regression test case for each partition. Software regression is observable only for the difference-revealing but never for the equivalence-revealing partitions.

Change-Sequence-Graph-guided Regression Test Generation directly explores the inter-dependencies among the syntactic changes. These inter-dependencies are approximated by a directed graph that reflects the control-flow among the syntactic changes and potential interaction locations. Every statement with data- or control-flow from two or more syntactic changes can serve as potential interaction location. Regression tests are generated by dynamic symbolic execution along the paths in this graph.

For the study of realistic regression errors, we constructed CoREBench consisting of 70 regression errors that were systematically extracted from four well-tested, and -maintained open-source C projects. We establish that the artificial regression errors in existing benchmarks, such as the Siemens Suite and SIR, are significantly less "complex" than those realistic errors in CoREBench. This poses a serious threat to validity of studies based on these benchmarks.

To quantify the complexity of errors and the complexity of changes, we discuss several complexity measures. This allows for the formal discussion about "complex" changes and "simple" errors. The complexity of an error is determined by the complexity of the changes necessary to repair the error. Intuitively, simple errors are characterized by a localized fault that may be repaired by a simple change while more complex errors can be repaired only by more substantial changes at different points in the program. The complexity metric for changes is inspired by McCabe's complexity metric for software and is defined w.r.t. the graph representing the control-flow among the syntactic changes.

In summary, we answer how to determine the semantic impact of a complex change and just how complex a "complex change" really is. We answer whether the interaction of the simple changes constituting the complex change can result in regression errors, what the prevalence and nature of such (change interaction) errors is, and how to expose them. We answer how complex a "complex error" really is and whether regression errors due to change interaction are more complex than other regression errors. We make available an open-source tool, `CyCC`, to measure the complexity of Git source code commits, a test generation tool, `Otter_Graph`, for C programs that exposes change interaction errors, and a regression error subject suite, `CoREBench`, consisting of a large number of genuine regression errors in open-source C programs for the controlled study of regresstion testing, debugging, and repair techniques.

Keywords : Software Evolution, Testing and Verification, Reliability

Acknowledgment

First I would like to thank my advisor, Abhik Roychoudhury, for his wonderful support and guidance during my stay in Singapore. Abhik has taught me all I know of research in the field of software testing and debugging. He has taught me how to think about research problems and helped me make significant progress in skills that are essential for a researcher. Abhik has been a constant inspiration for me in terms of focus, vision, and ideas in research, and precision, rigor, and clarity in exposition. He has always been patient, even very late at night, and has been unconditionally supportive of any enterprise I have undertaken. His influence is present in every page of this thesis and will be in papers that I write in future. I only wish that a small percentage of his brilliance and precision has worn off on me through our constant collaboration these past few years.

I would also like to thank Bruno C.d.S. Oliveira for several collaborative works that appear in this dissertation. It is a pleasure to work with Bruno who was willing to listen to new ideas and contribute generously. Other than helping me in research, Bruno has influenced me a lot to refine and clearly communicate my ideas.

I am thankful to David Rosenblum and Siau Cheng Khoo for agreeing to serve in my thesis committee, in spite of their busy schedules. I would also like to thank Siau Cheng Khoo and Jin Song Dong who readily agreed to serve in my qualifying committee. I am grateful for taking their time off to give most valueable feedback on the improvement of this dissertation.

I thank my friends and lab mates, Dawei Qi, Hoang Duong Thien Nguyen, Jooyong Yi, Sudipta Chattopadhyay, and Abhijeet Banerjee, for the many inspiring discussions on various research topics. Dawei has set an example in terms of research focus, quality, and productivity that will always remain a source of inspiration. Both, Hoang and Dawei, have patiently answered all my technical questions (in my early days of research I surely had plenty for them). Jooyong has helped immensely with his comments on several chapters of this dissertation. Sudipta was there always to listen and help us resolve any problems that we had faced. With Abhijeet I have had countless amazing, deep discussions about the great ideas in physics, literature, philosophy, the life, the universe, and everything.

For the wonderful time in an awesome lab, I thank Konstantin, Sergey, Shin Hwei, Lee Kee, Clement, Thuan, Ming Yuan, and Prakhar who joined Abhik's group within the last year or two, and Lavanya, Liu Shuang, Sandeep, and Tushar who have left the group in the same time to do great things.

I thank all my friends who made my stay in Singapore such a wonderful experience. Thanks are especially due to Yin Xing, who introduced me to research at NUS; Bogdan, Cristina, and Mihai, who took me to the best places in Singapore; Vlad, Mai Lan, and Soumya for the excellent saturday-evenings spent at the Badminton court; Ganesh, Manmohan, Pooja, Nimantha, and Gerisha, for the relaxing afternoon-tea-time-talks; and many more friends who made this journey such a wonderful one.

Finally, I would like to thank my family: my parents, Thomas and Beate, my partner, Kathleen, my sister Manja, and her daughter, Celine-Joelle, who have been an endless source of love, affection, support, and motivation for me. I thank Kathleen for her love, her patience and understanding, her support and encouragement, and for putting up with the many troubles that are due to me following the academic path. My father has taught me to regard things not by their label but by their inner working, to think in the abstract while observing the details, to be constructive and perseverant, and to find my own rather than to follow the established way. I dedicate this dissertation to him.

June 30, 2014

Papers Appeared

Marcel Böhme and Abhik Roychoudhury. CoREBench: Studying Complexity of Regression Errors. In the *Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA) 2014, pp.398-408*

Marcel Böhme and Soumya Paul. On the Efficiency of Automated Testing. In the *Proceedings of ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE) 2014, to appear*

Marcel Böhme, Bruno C.d.S Oliveira, and Abhik Roychoudhury. Test Generation to Expose Change Interaction Errors. In the *Proceedings of 9th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE) 2013, pp.339-349.*

Marcel Böhme, Bruno C.d.S Oliveira, and Abhik Roychoudhury. Partition-based Regression Verification. In the *Proceedings of ACM/IEEE International Conference on Software Engineering (ICSE) 2013, pp.300-309.*

Marcel Böhme, Abhik Roychoudhury, and Bruno C.d.S Oliveira. Regression Testing of Evolving Programs. In *Advances in Computers, Elsevier, 2013, Volume 89, Chapter 2, pp.53-88.*

Marcel Böhme. Software Regression as Change of Input Partitioning. In the *Proceedings of ACM/IEEE International Conference on Software Engineering (ICSE) 2012, pp.1523-1526.*

Contents

List of Figures

Chapter 1

Introduction

„Πάντα ῥεῖ καὶ οὐδὲν μένει."

— Ἡράκλειτος, c. 535 BC – 475 BC

Software changes constantly. There is always this one feature that could be added or that bug that could be fixed. Even after release, common practice involves remotely updating software that is deployed in the field. Patches are made available online and ready for download. For instance, the Linux operating system has been evolving over the last twenty years to a massive 300 *million* lines of code and, last time we looked,[1] *each day* an enormous 16 *thousand* lines of code are changed in the Linux kernel alone!

How can we check these software changes effectively? Even if we are confident that the earlier version works correctly, changes to the software are a definite source of potential incorrectness. The developer translates the intended semantic changes of the program's behavior into syntactic changes of the program's source code and starts implementing the changes. Arguably, as these syntactic changes become more complex, the developer may have more difficulty understanding the semantic impact of these syntactic changes onto the program's behavior and how these changes propagate through the source code. Eventually, the syntactic changes may yield some unintended semantic changes. Existing program functionality that used to work may not anymore. The result of such unintended semantic changes is software regression.

In this dissertation, we develop automated regression test generation and verification techniques that aim to expose software regression effectively. We put forward the thesis that a complex source code change can only be checked effectively by also stressing the interaction among its constituent changes. Thus, an effective test suite must exercise the inter-dependencies among the simple changes that constitute a complex change. We also show how we quantify error and change complexity, and develop a regression error benchmark.

[1] `http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git` Accessed: Feb'14

1.1 Thesis Statement

The thesis statement shall summarize the core contribution of this dissertation in a single sentence. The remainder of this dissertation aims to analytically and empirically test and support this thesis, discuss implications in the context of software evolution and regression testing, and introduce novel regression test generation techniques that build upon this thesis.

Thesis Statement

> *A complex source code change can only be checked cost-effectively by stressing the interaction among its constituent changes.*

In the following, we discuss the different aspects of this statement in more detail.

Firstly, we pursue the problem of cost-effectively *checking code changes*. Changes to a program can introduce errors and break existing functionality. So, we need cost-effective mechanisms to check whether the changes are correct and as intended. Two examples are regression verification as rather effective and regression test generation as rather efficient mechanisms to check source code changes. We discuss techniques that improve the efficiency of regression verification and more importantly the effectiveness of regression test generation.

Secondly, we want to check *complex source code changes*. In this work, we formally introduce a complexity metric for source code changes – the Cyclomatic Change Complexity (CyCC). But for now we can think of a simple change as involving only *one* changed statement while a more complex change is more substantial and involves several statements at different points in the program. It is well-known how to check the semantic impact of a simple source code change onto the program's behavior (e.g., [1, 2]). However, it is still not clearly understood how to check more complex changes effectively.

So, thirdly we claim that the *interaction among the simple changes constituting a complex change* must be considered for the effective checking of complex changes. We argue that the combined semantic impact of several code changes can be different from the isolated semantic impact of each individual change. This change interaction may be subtle and difficult to understand making complex source code changes particularly prone to incorrectness. Indeed, we find that regression errors which result from such change interaction are prevalent in realistic, open-source software projects.

1.2 Overview and Organization

This dissertation is principally positioned in the domain of software testing, debugging, and evolution. Hence, we start with a survey of the existing work on understanding and ensuring the correctness of evolving software. In Chapter 2 we discuss techniques that seek to determine the impact of source code changes onto other syntactic program artifacts and ultimately on the program's behavior. The chapter introduces the required terminology and discusses the background and preliminaries for this dissertation.

In Chapter 3, we introduce a technique that *improves the efficiency of automated regression verification* by allowing gradual and partial verification using dependency analysis and symbolic execution. Given two program versions, regression verification can effectively show the absence of regression for all program inputs. To allow *gradual regression verification*, we devise a strategy to partition the input space of two program as follows: If an input does not reveal an output difference, then every input in the same partition does not reveal a difference. Then, these input partitions are gradually and systematically explored until the exploration is user-interrupted or the complete input space has been explored. Of course, input that does not reveal a difference cannot expose software regression. To allow *partial regression verification*, the partition-based regression verification can be interrupted anytime with the guarantee of the absence of regression for the explored input space. Moreover, partition-based regression verification provides an alternative to regression test generation. Upon allowing the continued exploration even of difference-revealing partitions, the developer may look at the output differences and (in)formally verify the correctness of the observed semantic changes.

In Chapter 4, we introduce a technique that *improves the effectiveness of automated regression test generation* by additionally considering the interaction among several syntactic changes. Given two program versions, regression testing can efficiently show the absence of regression for some program inputs. We define a new class of regression errors, Change Interaction Errors (CIEs), that can only be observed if a critical sequence of changed statements is exercised but not if any of the changes in the sequences is "skipped". Employing two automated test generation techniques, one accounting and one not accounting for interaction, we generated test cases for several "regressing" version pairs in the GNU Coreutils. The test generation technique that *does not* account for potential interaction and instead targets one change at a time exposed only half of the CIEs while our test generation technique that *does* account for interaction and stresses different sequences of changes did expose all CIEs and moreover exposed five previously unknown regression errors.

In Chapter 5, we present complexity metrics for software errors and changes, and CoREBench as benchmark for realistic, complex regression errors. We define the complexity of an error w.r.t. the changes required to repair the error (and only the error). The measure of complexity for these changes is inspired by McCabe's measure of program complexity. Specifically, the complexity of a set of changes directly measures the number of "distinct" sequences of changed statements from program entry to exit. Intuitively, *simple errors* are characterized by a localized fault that may be repaired by changing one statement while *more complex errors* can be repaired only by more substantial changes at different points in the program. We construct CoREBench using a systematic extraction from over four decades of project history and bug reports. For each error, we determined the commit that introduced the error, the commit that fixed it, and a test case that fails throughout the error's lifetime, but passes before and after. Comparing the complexity for the realistic regression errors in CoREBench against the artificial regression errors in the established benchmarks, Siemens Suite and SIR, we observe that benchmark construction using manual fault seeding yields a *bias towards less complex errors* and propose CoREBench for the controlled study of regression testing, debugging, and repair techniques.

We conclude this dissertation with a summary of the contributions and discuss possible future work in Chapter 6.

1.3 Epigraphs

Each chapter in this dissertation starts with an epigraph as a preface to set the context of the chapter. In the following we give the English translations.

- Πάντα ῥεῖ καὶ οὐδὲν μένει (Greek). Everything flows; nothing remains still.
- Nanos gigantium humeris insidentes (Latin). Dwarf standing on the shoulders of giants.
- Divide et Impera (Latin). Divide and Rule.
- Das Ganze ist etwas anderes als die Summe seiner Teile (German). The whole is other than the sum of its parts.
- Simplicity does not precede complexity, but follows it (English).

Chapter 2

Related Work

> „Nanos gigantium humeris insidentes."
>
> — Sir Issac Newton, 1643 – 1727

Software changes, such as bug fixes or feature additions, can introduce software bugs and reduce code quality. As a result tests which passed earlier may not pass anymore – thereby exposing a regression in software behavior. This chapter surveys recent advances in determining the impact of the code changes onto other syntactic program artifacts and the program's behavior. As such, it discusses the background and preliminaries for this thesis.

Static program analysis can help determining change impact in an approximate manner while dynamic analysis determines change impact more precisely but requires a regression test suite. Moreover, as the program is changed, the corresponding test suite may, too. Some tests become obsolete while others are to be augmented, in particular to stress the changes. This chapter discusses existing test generation techniques to stress and propagate program changes. It concludes that a combination of dependency analysis and lightweight symbolic execution show promise in providing powerful techniques for regression test generation.

2.1 Introduction

Software Maintenance is an integral part of the development cycle of a program. In fact, the evolution and maintenance of a program is said to account for 90% of the total cost of a software project – the legacy crisis [3]. The validation of such ever-growing, complex software programs becomes more and more difficult. Manually generated test suites increase in complexity as well. In practice, programmers tend to write test cases only for corner cases or to satisfy specific code coverage criteria. Weyuker [4] goes so far as to speak of non-testable programs

if it is theoretically possible but practically too difficult to determine the correct output for some program input.

Regression testing builds on the assumption that an existing test suite stresses much of the behavior of the existing program P implying that at least one test case fails upon execution on the modified program P' when P is changed and its behavior regresses [5]. Informally, if the developer is confident about the correctness of P, she has to check only whether the changes introduced any regression errors in order to assess the correctness of P'. This implies that the testing of evolving programs can focus primarily on the syntactic (and semantic) entities of the program that are affected by the syntactic changes from one version to the next.

The importance of automatic regression testing strategies is unequivocally increasing. Software regresses when existing functionality stops working upon the change of the program. A recent study [6] suggests that even intended code quality improvements, such as the fixing of bugs, introduces new bugs in 9% of the cases. In fact, at least 14.8∼24.4% of the security patches released by Microsoft over ten years are incorrect [7].

The purpose of this chapter is to provide a survey on the state-of-the-art research in testing of evolving programs. This chapter is structured as follows. In Section 2.2, we present a quick overview of dependency analysis and symbolic execution which can help to determine whether the execution and evaluation of one statement influences the execution and evaluation of another statement. In particular, we discuss program slicing as establishing the relationship between a set of syntactic program elements and units of program behavior. In Section 2.3 we survey the related work of change impact analysis which seeks to reveal the syntactic program elements that may be affected by the changes. In particular, we discuss the problem of semantic change interference, for which the change of one statement may semantically interfere or interact with the change of another statement on some input but not on others. These changes cannot be tested in isolation. Section 2.4 highlights the salient concepts of regression testing. We show that the adequacy of regression test suites can be assessed in terms of code coverage which may approximate the measure of covered program behavior. For instance, a test suite that is 95% statement coverage-adequate exercises exactly 95% of the statements in a program. Section 2.5 investigates the *removal* of test cases from an existing test suite that are considered *irrelevant in some respect*. In many cases, a test case represents an equivalence class of input with similar properties. If two test cases represent the same equivalence class, one can be removed without reducing the current measure of adequacy. For instance, a test case in a test suite that is 95% statement coverage-adequate represents, for each executed statement, the equivalence class of inputs exercising the same

statement. We may be able to remove a few test cases from that test suite without decreasing the coverage below 95%. Similarly, Section 2.6 investigates the *augmentation* of test cases to an existing test suite that are considered *relevant in some respect*. If there is an equivalence class that is not represented, a test case may be added that represents this equivalence class. In the context of evolving programs it may be of interest to generate test cases that expose the behavioral difference exposed be the changes. Only difference-revealing test cases can expose software regression.

2.2 Preliminaries

Dependency analysis and symbolic execution can help to determine whether the execution and evaluation of a statement s_1 influences the execution and evaluation of another statement s_2. In theory, it is generally undecidable whether there exists a feasible path (exercised by a concrete program input) that contains instances of both statements [8]. Static program analysis can approximate the potential existence of such paths for which both statements are executed and one statement "impacts" the other. Yet, this includes infeasible ones. Symbolic execution (SE) facilitates the exploration of all feasible program paths if the exploration terminates. In practice, SE allows to search for input that exercises a path that contains both statements.

2.2.1 Running Example

```
1  input(i,j);
2  a = i;        //ch1 (a=i+1)
3  b = 0;
4  o = 0;
5  if(a > 0){
6    b = j;      //ch2 (b=j+1)
7    o = 1;
8  }
9  if(b > 0)
10   o = 2;      //ch3 (o=o+1)
11 output(o);
```

Original Version P

```
1  input(i,j);
2  a = i + 1;    //ch1 (a=i)
3  b = 0;
4  o = 0;
5  if(a > 0){
6    b = j + 1;  //ch2 (b=j)
7    o = 1;
8  }
9  if(b > 0)
10   o = o + 1;  //ch3 (o=2)
11 output(o);
```

Modified Version P'

Figure 2.1: Running Example

The program P on the left-hand side of Figure 2.1 takes values for the variables i and j as input to compute output o. Program P is changed in three locations to yield the modified program version P' on the righthand side. Change $ch1$ in line 2 is exercised by every input while the other two changes are

guarded by the conditional statements in lines 5 and 9. Every change assigns the old value plus one to the respective variable.

In this survey, we investigate which program elements are affected by the changes, whether they can be tested in isolation, and how to generate test cases that witness the "semantic impact" of these changes onto the program. In other words, in order to test whether the changes introduce any regression errors, we explain how to generate program input that produces different output upon execution on both versions.

2.2.2 Program Dependence Analysis

Static program analysis [9, 10] can approximate the "impact" of s_1 onto s_2. In particular, it can determine that there *does not exist* an input so that the execution and value of s_2 depends on the execution and value of s_1. Otherwise, static analysis can only suggest that there may or may not be such an input.

Statement s_2 *statically control-depends* on s_1 if s_1 is a conditional statement and can influence whether s_2 is executed [10]. Statement s_2 *statically data-depends* on s_1 if there is a sequence of variable assignments[1] that potentially propagate data from s_1 to s_2 [10]. The *Control-Flow Graph* (CFG) models the static control-flow between the statements in the program. Statements are represented as nodes. Arcs pointing away from a node represent possible transfers of control to subsequent nodes. A program's entry and exit points are represented by initial and final vertices. So, a program can potentially be executed along paths leading from an initial to a final vertex. The *Def/Use Graph* extends the CFG and labels every node n by the variables defined and used in n. Another representation of the dependence relationship among the statements in a program is the Program Dependence Graph (PDG) [11]. Every statement s_2 is a node that has an outgoing arc to another statement s_1 if s_2 directly (not transitively) data- or control-depends on s_1. A statement s_2 *syntactically depends* on s_1 if in the PDG s_1 is reachable from s_2.

The program dependence graphs for both program versions in our running example are depicted in Figure 2.2. The nodes are labeled by the line number. The graph is directed as represented by the arrows pointing from one node to the next. It does not distinguish data- or control-dependence. For instance, the node number 7 transitively data- or control-depends on the node number 1 but not on nodes number 6 or 3 in both versions. In the changed program there is a new dependence of the statement in line 10 on those in lines 4 and 7.

[1] A variable defined earlier is used later in the sequence.

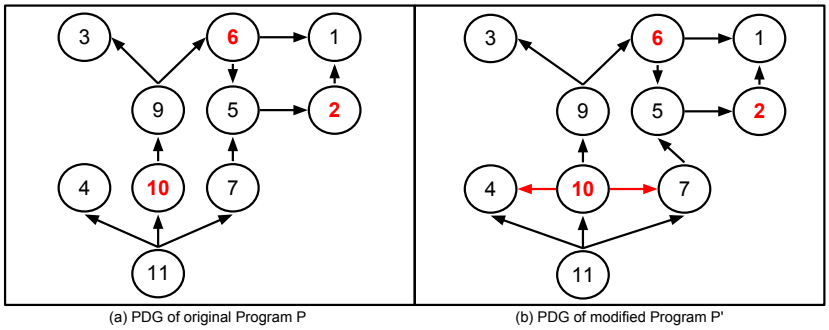


Figure 2.2: Program Dependency Graph of Running Example

2.2.3 Program Slicing

A *program slice* of a program P is a reduced, executable subset of P that computes the same function as P does in a subset of variables at a certain point of interest, referred to as *slicing criterion* [12, 13, 14, 15].

Line	Type	Slice
2	Forward	2, 5, 6, 7, 9, 10, 11
	Backward	1
6	Forward	6, 9, 10, 11
	Backward	1, 2, 5, 6
10	Forward	10, 11
	Backward	1, 2, 3, 5, 6, 9, 10

Original Version P

Line	Type	Slice
2	Forward	2, 5, 6, 7, 9, 10, 11
	Backward	1
6	Forward	6, 9, 10, 11
	Backward	1, 2, 5, 6
10	Forward	10, 11
	Backward	1, 2, 3, 5, 6, **7**, 9, 10

Modified Version P'

Figure 2.3: Static Backward and Forward Slices

A *static backward slice* of a statement s contains all program statements that potentially contribute in computing s. Technically, it contains all statements on which s syntactically depends, starting from the program entry to s. The backward slice can be used in debugging to find all statements that influence the (unanticipated) value of a variable in a certain program location. For example, the static backward slice of the statement in line 6 includes the statements in lines 1, 2, and 5. Similarly, a *static forward slice* of a statement s contains all program statements that are potentially "influenced" by s. Technically, it contains all statements that syntactically depend on s, starting from s to every program exit. A forward slice reveals which information can flow to the output. It might be a security concern if confidential information is visible at the output. As shown in Figure 2.3, for our running example, the static forward slice of the statement in line 6 includes the statements in lines 9, 10, and 11.

If two static program slices are isomorphic, they are behaviorally equivalent [16]. In other words, if every element in one slice corresponds to one element in the other slice, then the programs constituted on both slices compute the same output for the same input. Static slices can be efficiently computed using the PDG (or System Dependence Graph (SDG)) [11, 13]. It possible to test the isomorphism of two slices in linear time [15].

However, while a static slice considers all potential, terminating executions, including infeasible ones, a *dynamic slice* is computed for a given (feasible) execution [14]. A dynamic backward slice can resolve much more precisely which statements *directly* contribute in computing the value of a given slicing criterion. Dynamic slices are computed based on the execution trace of a program input. An execution trace contains the sequence of statement instances exercised by the input. In other words, input exercising the same path produces the same execution trace. For instance, executing program P in Figure 2.3 with input *(0,0)*, the output is computed as $o = 0$ in line 11. The execution trace contains all statements in lines 1, 2, 3, 4, 5, 9, and 11. However, only the statement in line 4 was contributing directly to the value $o = 0$ in line 11.

The *relevant slice* for a slicing criterion s_i contains all statement instances in the execution trace that contribute directly *and indirectly* in computing the value of s_i [17] and is computed as the dynamic backward slice of s_i augmented by potential dependencies [18] of s_i. More specifically, every input exercising the same relevant slice computes the same symbolic values for the variables used in the slicing criterion [19]. For instance, again executing program P in Figure 2.3 with input *(0,0)*, we see that the statements in lines 5, 2, and 1 indirectly contributed to to the value $o = 0$ in line 11. If the conditional statement in line 5 was evaluated differently, the value of o may be different, too. Hence, the output in line 11 *potentially depends* on (the evaluation of) the branch in line 5, which itself transitively data-depends on the statements in lines 2 and 1.

The applications of the relevant slice are manifold. In the context of debugging the developer might be interested in only those executed statements that actually led to the (undesired) value of the variable at a given statement for that particular, failing execution. Furthermore, relevant slices can be utilized for the computation of program summaries. By computing relevant slices w.r.t. the program's output statement, we can derive the symbolic output for a given input. Using path exploration based on symbolic output, we can gradually reveal the transformation function of the analyzed program and group input that computes the same symbolic output [19].

2.2.4 Symbolic Execution

While static analysis may suggest the potential existence of a path that exercises both statements so that one statement influences the other statement, the path may be infeasible. In contrast, *Symbolic Execution* (SE) [20, 21, 22] facilitates the exploration of feasible paths by generating input that each exercises a different path. If the exploration terminates, it can guarantee that there exists (or does not exist) a feasible path and program input, respectively, that exercises both statements. The test generation can be directed towards executing s_1 and s_2 in a goal-oriented manner [23, 24, 25, 26].

SE generates for each test input a condition as first-order logic formula that is satisfied by every input exercising the same program path. This *path condition* is composed of a branch condition for each exercised conditional statement (e.g. `If` or `While`). A conjunction of branch conditions is satisfied by every input evaluating the corresponding conditional statements in the same direction. The negation of these branch conditions one at a time, starting from the last, allows to generate input that exercises the "neighboring" paths. This procedure is called *path exploration*.

	Input	**Output**
P	$i \leq 0$	$o = 0$
	$i > 0 \wedge j \leq 0$	$o = 1$
	$i > 0 \wedge j > 0$	$o = 2$
P'	$i \leq -1$	$o' = 0$
	$i > -1 \wedge j \leq -1$	$o' = 1$
	$i > -1 \wedge j > -1$	$o' = 2$

Figure 2.4: Symbolic Program Summaries

The symbolic execution of our running example can reveal the symbolic program summaries in Figure 2.4. Both versions have two conditional statements. So there are potentially $2^2 = 4$ paths. One is infeasible. The others produce the symbolic output presented in the figure. Input satisfying the condition under *Input* computes the output under *Output* if executed on the respective program version.

Technically, there are static [20] and dynamic [21, 22] approaches to symbolic execution. The former carry a symbolic state for each statement executed. The latter augment the symbolic state with a concrete state for the executed test input. A *symbolic state* expresses variable values in terms of the input variables and subsumes all feasible concrete values for the variable. A *concrete state* assigns concrete values to variables. System and library calls can be modelled as uninterpreted functions for which only dynamic SE can derive concrete output values for concrete input values by actually, concretely executing them [27].

In theory, path exploration can determine all feasible paths if it terminates. Yet, the number of paths grows exponentially due to the number of conditional statements in the explored program. To attack this *path explosion problem*, it is possible to prune a family of infeasible paths when one is encountered [28], group a set of feasible paths to a path family so as to explore only one member of a each family [19, 29, 30], massively parallelize the path exploration [31], and explore components of the program independently so as to compose the fragmented exploration results globally [32]. Further, more scalable approaches are presented in combination with white box fuzz testing [33] and machine learning techniques [34].

2.3 Change Impact Analysis

Change impact analysis [35, 36, 37, 38] can help to check whether and which program entities (including the output) are affected by syntactic program changes. The developer can focus testing efforts on affected program entities in order to more efficiently expose potential regression errors introduced by the changes.

Similar to dependence analysis, it is generally undecidable whether there exists input that exercises even a single changed statement [8] and not to mention that makes any behavioral difference observable. However, static analysis can approximate the potential existence of program paths that reach changes and propagate the semantic effects. Differential symbolic execution [39] allows a more precise analysis of the existence of program paths that can propagate the semantic effects of changes. Dynamic program analysis requires the existence of at least one such program path and can precisely determine the affected program entities and which changes are interacting.

2.3.1 Static Change-Impact Analysis

Statically, we can determine i) which statements are *definitely not affected* by a change [12, 13, 38], ii) which statements are *probably affected* by a change [40], iii) which set of changes do *definitely not semantically interfere* and can thus be tested in isolation [41, 42] and iv) which statements remain, cease to, or begin to syntactically depend on a statement that is changed [43, 44, 45].

There are mainly two different syntactic approaches to statically compute the semantic difference introduced by the changes - text-based and dependency-based differencing. *Text-based differencing* [46, 47, 48] is a technique that given two program versions can expose changed code regions. This includes approaches that compare strings [47], as for instance the Unix utility `diff`, and approaches that compare trees [48]. Text-based differencing tools may efficiently

identify textual differences but they cannot return information on code regions in the program that are affected by the changes.

Dependency-based differencing [43, 44, 45, 49] methods can compute the program entities affected by the changes. Using the static forward slice of the changed statements, we can compute those statements that are potentially affected by the change. Practically, this can be more than 90% of the statements in a program [37]. Still, every statement that is not in the static forward slice of any changed statement is definitely not affected by a change of that statement. Based on empirically justified assumptions, Santelices and Harrold [40] show how to derive the *probability* that the change of one statement has an impact on another given statement. Moreover, it is possible to check whether a set of changes potentially semantically interferes by computing the intersection of the static forward slices for each changed statement [41, 50]. If the static program slices do not intersect, the set of changes can be tested in isolation.

Change Set	**Interference Locations**
$\{ch1, ch2\}$	6, 9, 10, 11
$\{ch1, ch3\}$	10, 11
$\{ch2, ch3\}$	10, 11
$\{ch1, ch2, ch3\}$	10, 11

Figure 2.5: Potentially Semantically Interfering Change Sets

For our running example, the static forward slices of the changes $ch1$ and $ch2$ in lines 2 and 6 are *not intersecting* at line 7 as shown in Figure 2.5. In fact, only $ch1$ may have a semantic effect on line 7. In contrast, the forward slices of both changed statements *are intersecting* at line 9, amongst others. Later in the text we show that $ch1$ and $ch2$ semantically interfere for input $\{0, 0\}$ because removing one change (by replacing the modified code with the original code for the change) alters the semantic effect of the other change on that execution. Therefore, both changes cannot be tested in isolation.

Using program slicing and reconstitution[2], Horwitz [43] presents a technique to compute a program P_C for two program versions P and P' that exhibits all changed behaviors of P' w.r.t. P. The authors note that we cannot always assume to know the correspondence between the elements of the respective PDGs of both versions (P and P') and propose a solution using slice-isomorphism testing which executes in linear time [15]. The explicit (and automatic) tagging of every syntactic element is another solution to establish the correspondence of an element in the PDG in one version to an element in the PDG of another version [42]. Semantic differencing tools based on static dependency analysis were

[2]A program is reconsituted when source code is generated from a dependence graph or program slice [51, 43].

implemented by Jackson and Ladd [44] and more recently by Apiwattanapong et al. [49] and Loh and Kim [45]. However, while syntactic tools are efficient, they are often rather imprecise as the semantics of the programs are ignored. For instance, two syntactically very different pieces of code can always compute the same output for the same input. Yet, dependency-based tools will always report differences.

2.3.2 Dynamic Change Impact Analysis

Dynamically, given an input t, it is possible to determine i) much more precisely which statements are affected by the (exercised) changes [35], ii) whether and how the combined semantic effects of the exercised changes are propagated to the output [52, 53, 17]), and iii) whether two subsets of the exercised changes are interacting [54].

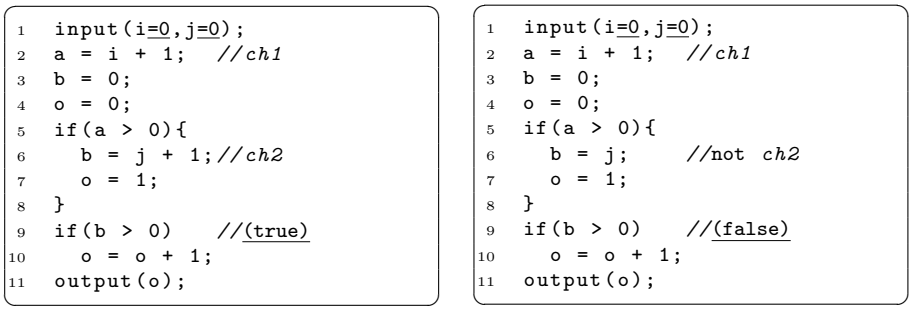
Assume that only the statement c has changed from one program version to the next. To check whether the *semantic effect* of c is propagated to another statement s for an input t, it is sufficient to determine whether s is exercised in one but not in the other version or the values for the variables used in s are different in both versions (cf. [52, 2]). Two changes, c_1 and c_2, *interact* for the execution of t if removing one change (i.e., replacing the modified code with the original code for the change) alters the semantic effect of the other change on that execution. Santelices et al. [54] define and present a technique to compute change interaction. First, given two (sets of) changes c_1 and c_2, four program configurations are constructed - the modified program P', the modified program with c_1 being replaced by the original code ($P'\backslash c_1$), the modified program with c_2 being replaced by the original code ($P'\backslash c_2$), and the modified program with both changes being replaced by the original code ($P'\backslash\{c_1, c_2\}$). Second, the test case t is executed on all configurations to compute the execution traces $\pi(t, P')$, $\pi(t, P'\backslash c_1)$, $\pi(t, P'\backslash c_2)$, and $\pi(t, P'\backslash\{c_1, c_2\})$ augmented by variable values.

$$\mathit{effect}(t, c_1, P') \leftarrow \mathit{diff}(\pi(t, P'), \pi(t, P'\backslash c_1)) \tag{2.1}$$

The semantic effect of c_1 on P' is computed as the difference of the augmented execution traces when executing t on P' and on $P'\backslash c_1$.

$$\begin{aligned}\mathit{interact}(t, c_1, c_2, P') \leftrightarrow ((&\mathit{effect}(t, c_1, P') \neq \mathit{effect}(t, c_2, P'\backslash c_1)) \\ &\vee(\mathit{effect}(t, c_1, P'\backslash c_2) \neq \mathit{effect}(t, c_2, P'))).\end{aligned} \tag{2.2}$$

Both changes c_1 and c_2 are interacting iff the semantic effect of c_1 on P' is different from the semantic effect of c_2 on $P'\backslash c_1$ or the semantic effect of c_2 on P' is different from the semantic effect of c_1 on $P'\backslash c_2$.



Modified Version (P') P' without $ch2$ ($P'\backslash ch2$)

Figure 2.6: Changes $ch1$ and $ch2$ interact for input {0,0}

An example of change interaction for a given test case is depicted in Figure 2.6. It shows two configurations - the modified program P' on the lefthand side and the modified program with $ch2$ being replaced by the original code, $P'\backslash ch2$, on the righthand side. Input $t = \{0, 0\}$ exercises the changes $ch1$ and $ch2$ in lines 2 and 6 in both configurations. The *semantic impact* of $ch2$ on P' is the conditional statement in line 9 being evaluated in different directions in both configurations. As a result, input t produces output $o = 2$ in configuration P' and $o = 1$ in configuration $P'\backslash ch2$. The semantic impact of $ch1$ on $P'\backslash ch2$ is the conditional statement in line 5 being evaluated in different directions in both configurations. As a result, input t produces output $o = 1$ in configuration $P'\backslash ch2$ and $o = 0$ in configuration $P'\backslash\{ch1, ch2\}$. Note, there does not exist any input for which $ch3$ has a semantic impact on any configuration. Both changes, $ch1$ and $ch2$ are *semantically interacting* for input $\{0, 0\}$ because the semantic impact of $ch2$ on P' is different from the semantic impact of $ch1$ on $P'\backslash ch2$ for t. Note, there does not exist any input for which $ch1$ or $ch2$ are interacting with $ch3$. Yet, in general it is undecidable whether there exists such an input t that exercises a changed statement and propagates the semantic effects to another statement (incl. the output), or upon which two (sets of) changes are interacting.

2.3.3 Differential Symbolic Execution

Differential Symbolic Execution [39] can approximate those paths that potentially propagate the semantic effects of a change to the output. Exploiting the fact that the original and changed version of a method are syntactically largely similar, the behaviour of common code fragments is summarized as *uninterpreted functions*. In both versions the behavior of the changed method can be represented as *abstract program summaries*. An abstract summary consists of a set of partition-effect pairs. A partition-effect pair consists of a condition that

is to be satisfied to observe the effect and an effect that computes the output in terms of the method input variables. Both, the condition and the output function can contain uninterpreted functions.

	Input	Output
P	$b(i,j) > 0$	$o = 2$
	$b(i,j) \leq 0$	$o = o(i,j)$
P'	$b(i,j) > 0$	$o' = o(i,j) + 1$
	$b(i,j) \leq 0$	$o' = o(i,j)$

Figure 2.7: Abstract Program Summaries for P and $P'\backslash\{ch1, ch2\}$

In our running example in Figure 2.1 many code fragments are changed. Suppose that only the statement in line 10 is changed in the original program ($P'\backslash\{ch1, ch2\}$). Note, both versions P and $P'\backslash\{ch1, ch2\}$ are semantically equivalent (i.e., compute the same output for the same input). As depicted in Figure 2.7, the behavior of the common code region from lines 2-8 is summarized as uninterpreted functions. In particular, the variable b used in line 9 is defined by the uninterpreted function $b(i, j)$ while o used in lines 11 and 12 is defined by the uninterpreted function $o(i, j)$.

To reveal the differential behavior of the changed version w.r.t. the original version, DSE allows to compute (partition-effects or functional) *deltas* upon both abstract summaries. For instance, if the conditions are the same but the effects are different in both versions and the computed delta does not contain an uninterpreted function, then every input satisfying the condition must expose a difference in program behavior. On the other hand, if the delta contains uninterpreted functions, then the behavior of the common code fragment has to be explored first. For instance, for the abstract summary in Figure 2.7, DSE can show that if $b(i, j) > 0$ is satisfiable, the semantic effects of the changes may propagate to the output. However, in order to find an input that exposes a behavioral difference, first we have to check whether and for which values of i and j the condition $b(i, j) > 0$ can be satisfied. Second, we have to determine a value that satisfies $o \neq o'$ and thus $1 \neq o(i, j)$. There is no such input.

2.3.4 Change Granularity

When a new version of the program's source code is analyzed or tested, we may want to decompose this change from one version to the next into smaller "changes" which can be analyzed and tested in isolation. Syntactic change can be defined on different levels of granularity. For example, we can speak of changed components, features, classes, methods, code regions, statements, or of changed program dependencies.

In some cases changes cannot be tested in isolation and yield inconsistent program configurations. Zeller [55] distinguishes *integration failure*, for which one change requires another change that is not included in the configuration, *construction failure*, for which the change configuration cannot be compiled, and *execution failure*, for which the test outcome is unresolved after execution.

```
1 public class Test{
2   public int inc(int b){ //change c1: Add function
3     return b++; //change c2: Add statement
4   }
5 }
```

Figure 2.8: Integration Failure

Ren et al. [38] define change as cluster of changed statements that are required to avoid integration and construction failures. A program configuration can only contain every or no changed statement within a cluster of a selected changes. In Figure 2.8, change c_1 is adding method `inc` to a class. Change c_2 is adding a statement to that method. A configuration that contains c_2 must also contain c_1. The authors define several types of changes, such as adding, deleting, and changing methods or classes.

Jin et al. [56, 57] generate random test cases that are executed on both versions of a changed class. The authors note that the class interface should not change from one version to the next because the same unit test case cannot be executed on both versions simultaneously. Then, the test outcome is unresolved. Korel et al. [58] explain how to find the common input domain when the dimensionality of the input space changes.

As in this thesis, Santelices et al. [54] define a code level change as "a change in the executable code of a program that alters the execution behavior of that program". The configuration $P'\backslash c$ is a syntactically correct version of P' where the original code of a change c replaces the modified code from that change.

2.4 Regression Testing

Regression testing is a technique that checks whether any errors are introduced when the program is changed. While static change impact analysis reveals unaffected program elements, regression testing should exercise those elements which are potentially affected by the changes. In particular, software regression can only be observed for input that exposes a semantic difference in both programs.

Generally, regression testing is based on at least three assumptions: i) the program behaves in a deterministic manner [21], ii) the software tester is routinely able to check the correctness of the program output for any input [4], and

iii) an “adequate” regression test suite stresses much of the program’s behavior, so that, when the program is changed and its behavior regresses, at least one test case fails upon execution on the changed program [5, 59].

2.4.1 Deterministic Program Behavior

A test case is meaningful only if executing the same test upon the same program always produces the same output - the program behavior is deterministic. Only then the output is representative for the test case and can be compared among program versions. Indeterminism can be introduced, for instance, by the program environment, like a file system, or concurrency.

The program environment can introduce indeterminism. Some authors [21] explicitly note that a library function, like an operating-system function or a function defined in the standard C library, is treated as an unknown but deterministic black-box that cannot be analysed but executed. In practice, this may not hold. Suppose, the analyzed program loads a file every time it is executed. At one point the file is changed by a third party. Suddenly, the same test that used to pass now fails on the same program. An approach to model the execution environment is discussed by Qi et al. [60].

The behavior of concurrent programs can be considered indeterministic, as well (cf. race conditions). This can be mitigated by constructing a finite model that considers all feasible schedules within which two or more threads can be executed concurrently and enumerate these schedules to determine for instance the existence of race conditions [61].

2.4.2 Oracle Assumption

In general, a software tester is not routinely able to check the correctness of the program output for any input. A mechanism that determines upon execution whether a test case passes or fails is known as *oracle*. In the context of evolving programs, an oracle further decides whether or not a behavioral difference exposed by a test case is intentional (see change contracts [62]). If the difference is not intentional this test case would be a witness of regression.

The oracle problem [4] postulates that an oracle that decides for every input whether the program computes the correct output is pragmatically unattainable and only approximate. Informally, the oracle problem denotes that even an expert may in some cases not be able to distinguish whether an observed functionality is a bug or a feature. However, there are types of errors that are generally acknowledged as such; for instance, exceptions, buffer overflows, array-out-of-bounds, or system crashes [21, 63, 57, 64, 65]. These are called de-facto or implicit oracles [4, 57]. Otherwise, it is possible to specify errors explicitly

as assertion-, property-, or specification violations [66, 67, 68, 69, 70]. In some cases, the same functionality is implemented more than once to compare the output [71] or the the program is run on "simplified" input data to accurately assess the "simple" output [4]

The oracle problem affects specifically automated test generation, debugging, and bugfixing techniques. For instance, an automated bugfixing technique can correct the (buggy) program only relative to explicitly specified or known errors. In a recent work, Staats et al. [72] point out that empirical software testing research should explicitly consider the definition of oracles when presenting the empirical data in order to better evaluate the efficacy of a testing approach and allow for comparison by subsequent studies.

2.4.3 Code Coverage as Approximation Of Adequacy

The measure of code coverage *approximates* the adequacy of a test suite to cover much of the program behavior [59]. A test suite is 100% *code coverage-adequate* w.r.t. a coverage-criterion if all instances of the criterion are exercised in a program by at least one test case in the test suite [73]. A statement coverage-adequate test suite requires that every statement in the program is exercised by at least one test case in the test suite. Decision coverage requires that the condition in every control structure is evaluated both, to true and false. A path coverage-adequate test suite exercises every feasible path from program entry to exit at least once [73].

The measure of code coverage (excepting path coverage) can often be absolutely computed using syntactic representations of the source code, such as the nodes and edges in a PDG. For instance, a test suite is 50% statement coverage-adequate, if all test cases in the test suite exercise exactly half of the statements in the program. For our running example, the test suite T_{RE} in Equation 2.3 covers every path in both program versions (cf. Fig. 2.4 on page 11).

$$T_{RE} = \left\{ \begin{array}{ccc} \{ & -2, & -2 \ \}, \\ \{ & 2, & -2 \ \}, \\ \{ & 2, & 2 \ \} \end{array} \right\} \tag{2.3}$$

Generally, it is undecidable whether there exists a 100% coverage-adequate test suite for a given program and a given coverage criterion because it is undecidable whether there exists an input that exercises a path containing a given syntactic program artifact [8]. While code-coverage can often be efficiently computed for a test suite w.r.t. a finite amount of syntactic program artifacts, there are other measures to assess the test suite adequacy, such as fault-based

[74, 75, 26], change-based [76], or "behavioral" [59] criteria. The efficacy of the different measures can vary and has been compared [77, 78, 79, 80].

The approximation of the amount of covered behavior by the amount of covered code may not properly quantify the capability of a test suite to reveal regression errors. Specifically, a code coverage-adequate test suite may not inspire confidence in the correctness of the program [81] and may not perform significantly better than random generated test cases in terms of revealing program errors [82, 83, 80, 81]. Weyuker et al. [80] observe that while a test case represents one or more equivalence classes in the input space of a program[3], such an equivalence class may not be homogeneous w.r.t. failure - if one test case fails, every input in the same class fails. For instance, it is not true that if a test case exercises some branch (which it may represent) and exposes an error, then every input exercising the same branch exposes an error.

This leads to our thesis of "semantic" coverage criteria which requires the partitioning of the input space w.r.t. correctness. As for our running example, the regression test suite T_{RE} in Equation 2.3 exercises every path in both versions. However, it does not expose any behavioral difference when comparing the output upon execution in both versions. As software regression is observable only for input that exposes a behavioral difference, we can conclude that even a path coverage-adequate test suite may not expose software regression.

2.5 Reduction of Regression Test Suites

In order to gain confidence that program changes did not introduce any errors, regression test suites are executed recurringly. The number of test cases can greatly influence the execution time of a test suite. When the program is changed, we can choose to execute only relevant test cases that actually execute the changed code regions and are more likely to expose regression errors. Similarly, we can permanently remove test cases that are irrelevant w.r.t. some measure of test suite adequacy.

2.5.1 Selecting Relevant Test Cases

Given a test suite, when the program is changed, only those test cases may be selected that actually stress the changed functionality and can expose software regression [84, 85, 38, 86]. On the other hand, test cases that do not exercise the program changes cannot expose software regression that are introduced by these changes. Ideally, executing only the selected test cases reduces the testing time while preserving the capability to reveal regression errors.

[3] E.g., an input space subdomain represents every input exercising a certain branch.

For example, Ren et al. [38] present a tool that given a test suite can determine test cases that do with certainty not exercise any changed statement. For the analyzed subjects, on average 52% of the test cases were *potentially affected* by the changes; each test case by about 4% of the changes. Furthermore, given a test suite, the tool can ascertain which changed statements are with certainty not executed by any test case. The test suite should be augmented by test cases that exercise these statements to decide whether these changes introduced any regression errors.

Graves et al. [84] empirically compare several test selection techniques. The *minimization technique* chooses only those test cases that cover the modified or affected parts of the program. It produces the smallest and least effective test suite. The *safe technique* selects all test cases in the original test suite that can reveal faults in the program. This technique was shown to find all faults while selecting 60% of the test cases on the median. The *ad-hoc or random technique* selects test cases on a (semi-) random basis. The random technique produced slightly larger test suites than the minimization technique but on average yielded fault detection results equivalent to those of the minimization technique with little analysis costs. Furthermore, randomly selected test suites could be slightly larger than a safely selected test suite but nearly as effective.

2.5.2 Removing Irrelevant Test Cases

Test cases in a large test suite that are redundant in some respect may be removed completely [87, 88, 89]. Ideally, test suite reduction decreases the execution time of recurring regression testing while preserving the capability to reveal regression errors. Considering test cases as representatives of equivalence classes, it is possible to remove those test cases that represent the same equivalence class without reducing the current measure of adequacy. For instance, given a 95% branch coverage-adequate test suite T, test cases are removed from T until the removal of one more test case also reduces the branch-coverage of T to less than 95%. Based on their empirical results, Rothermel et al. [90] conclude that "test suite minimization can provide significant savings in test suite size. These savings can increase as the size of the original test suites increases, and these savings are relatively highly correlated (logarithmically) with test suite size".

However, the reduction of a test suite w.r.t. a code coverage criterion has a negative impact on the capability of a test suite to reveal a fault [91, 92]. Hao et al. [93] observe that the reduction w.r.t. statement coverage incurs a loss in fault-detection capability from 0.157 to 0.592 (with standard deviations from 0.128 to 0.333) for the analyzed subjects. In other words, about 16-60% of the faults

originally detected become unexposed using the reduced test suite. Yu et al. [91] empirically determine that the reduction of a test suite w.r.t. statement coverage increases the fault localization expense by about 5% on average for the analyzed subjects. In other words, given original test suite T and the test suite T' that is reduced w.r.t. statement coverage, if Tarantula[4] were to pinpoint a single statement as probable fault location using T, then Tarantula would require the tester to examine 5% of the source code as probable fault location using T'. In a recent work, Hao et al. [93] propose a test suite reduction technique that removes test cases from the test suite while maintaining the capability to reveal faults above a user-defined threshold.

2.6 Augmentation of Regression Test Suites

In order to gain confidence that program changes did not introduce any errors, existing test suites are augmented by relevant test cases i) to better satisfy a given test suite adequacy criterion, such as code coverage, and ii) to expose behavioral differences which are introduced by changes to the program. Only test cases that reveal a difference upon execution on both program versions can potentially expose software regression.

There are automatic test generation techniques to better satisfy coverage-based [95, 96, 97, 33], fault-based [98, 99, 26], and "behavioral" [59] adequacy criteria. Approaches to generate test cases that expose a behavioral difference in two program versions can be coarsely distinguished into three classes. *Syntactic approaches* [2, 1, 100] aim to generate input that first reaches at least one change, then infects the program state, and thereupon propagates its semantic effect to the output. *Semantic approaches* [39, 19] use a form of program summaries to find input that exposes a difference. *Random approaches* [57, 101] randomly generate test cases that may or may not expose a difference when executed on both versions.

2.6.1 Reaching the Change

Search-based test generation techniques [23, 102] aim to generate test cases that reach specified targets in the program. These targets can be coverage goals to increase code-coverage [95, 96, 33], program changes [1, 2, 25, 26, 99], or specified program faults like assertions [66, 58], exceptions [65, 63], and (functional) properties [67, 68]. Korel and Al-Yami [58] present a technique that given two program version reduces the problem of generating input that exposes a behavioral difference to the problem of reaching an assertion.

[4]Tarantula is an automatic fault-localization technique [94].

It is generally undecidable whether there exists an input that reaches a change [8]. Practically, we can generate test cases to search for such input. If we can assign a given input some measure of *distance* to the change, then we can apply search strategies that reduce this distance. The distance of a test case t to a changed statement c can be defined, for instance, based on the length of the control-dependency chain from c to those branches exercised by t that are not evaluated in favor of the execution of c, that is, have to be negated in order to reach c.

Local search strategies, such as hill climbing [66, 103], monotonically reduce this distance. Random restart procedures [26] can prevent the search strategy to get stuck in a local minimum distance. Ferguson and Korel [103] introduce the Chaining Approach (CA) that leverages data- and control dependencies to generate input that reaches a target by identifying and exercising a necessary sequence of nodes beforehand. Given a target c, CA analyzes the program dependency graph to find program input that exercises c. The target c can be reached only if those nodes upon which c control-depends are evaluated in favor of the execution of c. Given a node p upon which c control-depends is not evaluated in favor of c for some input t, then CA will generate input for which p is negated. If p cannot be negated by input exercising the same path (i.e., the same sequence than t of nodes in the CFG), then p is marked as the *problem node*. "The chaining approach finds a set $LD(p)$ of last definitions of all variables used at problem node p. By requiring that these nodes are executed prior to the execution of problem node b, the chances of altering the flow execution at problem node p may be increased" [103]. Effectively, the nodes in $LD(p)$ become intermediate target nodes. This sequence of (intermediate) target nodes is called *event sequence* (or chain).

```
1  input(i=-2,j=-2);
2  a = i + 1;
3  b = 0;
4  o = 0;
5  if(a > 0){   //(4)(false)
6    b = j + 1;//(3)intermed.
7    o = 1;
8  }
9  if(b > 0)    //(2)problem node
10   o = o + 1;//(1)target
11 output(o);
```

Search state with input $\{-2, -2\}$

```
1  input(i=2,j=-2);
2  a = i + 1;
3  b = 0;
4  o = 0;
5  if(a > 0){   //(true)
6    b = j + 1;
7    o = 1;
8  }
9  if(b > 0)    //(2)(false)
10   o = o + 1;//(1)target
11 output(o);
```

Search state with input $\{2, -2\}$

Figure 2.9: Chaining Approach Explained for Modified Program P'

We explain the chaining approach for our running example in Figure 2.9. Suppose, we want to generate an input for the modified version P' that exercises the changed statement in line 10. The CA may start with random input

$\{-2, -2\}$ as shown on the left-hand side. CA determines the branch in line 9 as problem node. The only variable used in the condition is b which is defined in lines 3 and 6. So, CA designates the statement in line 6 as the intermediate target which is guarded by the branch in line 5. This branch is evaluated to $false$. To negate this branch, CA has to compute an input so that $i + 1 > 0$ (using function minimization). Thus, the next input may be $\{2, -2\}$ as shown on the righthand side. The branch in line 9 guarding the target in line 10 can be negated by input exercising the same path than $\{2, -2\}$. In particular, CA computes an input so that $i + 1 > 0 \wedge j + 1 > 0$ which is satisfied by test $\{2, 2\}$.

Search strategies based on genetic algorithms [104], choose the "fittest" set of inputs from one generation as "seed" for the next generation to find a global minimum distance. Search strategies based on counterexample-guided abstraction refinement [67, 105, 68] try to prove that no such input exists in an abstract theory. If instead a (possibly spurious) counter-example is found, it continues to prove the absence of a counter-example in a refined theory. This repeats until either its absence is proven or a concrete (non-spurious) counter-example is found. A particular kind of search strategies seeks to cover a *set of targets* at once or in a given sequence [96, 24, 26].

To optimize the search it is possible to reduce the search-space in a sound [102, 30, 28, 100] and approximative manner [106, 29], search distinct program components independently and compose the results [32, 68], or execute the search strategy on multiple instances in parallel [31]. Yet, since the problem is undecidable in general, the search for an input that reaches a change may never terminate in some cases [67].

Another practical approach to find input that reaches a change is the random generation of program input [107, 83, 108, 57]. Arcuri et al. [109] analytically determine that the time to reach all of k targets by random test generation is $O(k * log(k))$.

2.6.2 Incremental Test Generation

Given only the changed statements in the changed program P', incremental test generation is concerned with testing the code regions that are affected by the changes. On the one hand, test cases that do not exercise a changed statement cannot reveal a behavioral difference [38]. On the other hand, test cases that do exercise one or more changed statements may or may not yield an observable behavioral difference [2, 110, 25]. In fact, one study [26] finds that only 30% to 53% of the test cases that do exercise a changed statement are difference-revealing for the analyzed whole programs.

In general, every statement in the static forward slice of a changed statement is potentially affected by the change [13]. Hence, one can direct the path exploration of P' explicitly towards the changed statements in order to exercise program paths that are affected by the changes and increase the likelihood to observe a behavioral difference [25]. Vice versa, one can avoid the exploration of paths in P' that will not stress a changed code region and are unlikely to propagate the semantic effect of a change [100].

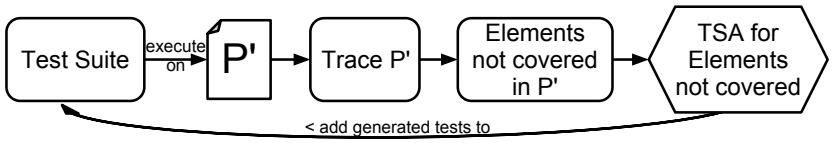


Figure 2.10: Re-establishing Code Coverage

Upon program change, the code coverage of an existing test suite may decrease. As outlined in Figure 2.10, Xu et al. [95, 97] firstly apply a test selection technique to find all test cases that are affected by the changes. Secondly, these test cases are executed on the changed program to determine syntactic program artifacts that are not covered (anymore). Lastly, the authors seek to re-establish the code-coverage by generating test cases that exercise those syntactic program elements that are not covered in P' reusing the selected test cases.

The analysis of only a single version, either P or P', is insufficient to expose all behavioral differences. Even input exercising the same affected path in P' may exercise multiple, different paths in the original version P [111]. As a result, the semantic interaction [54] of a set of changes may or may not be observed at the output, even if every affected path is exercised. As for our running example, the test suite T_{RE} in Equation 2.3 on page 19 exercises every path in both program versions. However, this test suite does not expose any behavioral difference when comparing the output upon execution in both versions.

2.6.3 Propagating a Single Change

One may ask: What is the semantic impact of a change onto the program? Does it introduce a bug? Since it is undecidable whether there exists input that exercises the changed statement [8], it is also undecidable whether there exists an input that reveals a behavioral difference and not to mention software regression. However, given both program versions P and P' we can search for input that 1) reaches the changed statement, 2) infects the program state, and 3) propagates the semantic effect to the output [1, 52, 98].

Santelices et al. [112, 2] describe a technique that derives requirements for new test cases to propagate the semantic effect of the exercised change to a user-specified minimum distance (in terms of static dependence chains starting at the changed statement). The tester can use these requirements to write a test case that is more likely to reveal different behavior in the changed version than a test case that merely executes the change. Using text-based differencing, the algorithm finds the changed statement in the original program P and modified version P'. Then, by means of (partial, dynamic) symbolic execution the path condition and symbolic state for those statements following the changed statement are computed. The path conditions and symbolic states of the corresponding statements are compared for P and P' and requirements derived.

Qi et al. [1] generate a test case t, so that t executes a given change c and the effect of c is observable in the output produced by t. The test case t can be considered a *witness* of the behavioral difference introduced by c in the new program version. The underlying algorithm works as follows.

First, using an efficient hill-climbing search strategy, input that reaches the changed statement is generated. For optimization, all test cases in an existing test suite are executed and respective path conditions are derived. A distance function determines the probability of an input to reach a change and imposes an order over the test inputs. Always taking the input "closest" to the change, the respective path condition is manipulated to generate new input t_{new} that minimizes the distance to the changed statement for the execution of t_{new} on P'. This repeats until the distance is zero and the change is reached.

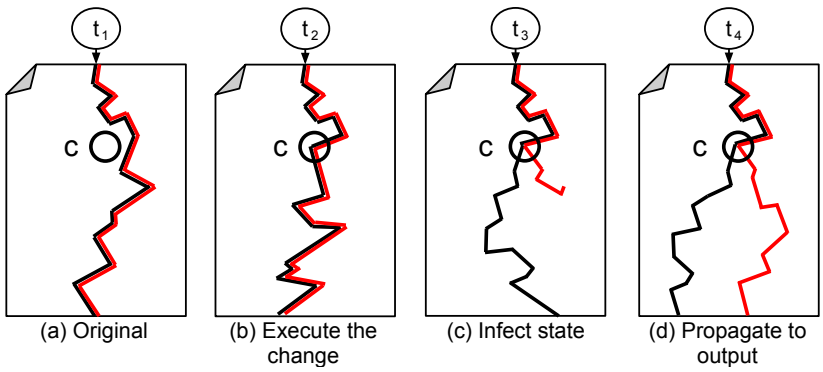


Figure 2.11: Generating input that satisfies the PIE principle

Second, using the Change Effect Propagation Tree (CEPT), the semantic effect of the changed statement is propagated to the output. The semantic effect of a change is observable for an input t in a variable v along the path (and ultimately at the output) if v has a different value for the execution of t

on P than of t on P' (i.e., $execute(t, P', v) \neq execute(t, P'\backslash c, v)$). The CEPT identifies terminating locations of effect propagation for each execution. The CEPT represents why a change cannot be propagated any further. For example, a variable v that carries the semantic effect of the change is redefined without using v anywhere else before. The authors determine three different reasons for propagation termination and handle them accordingly. The path condition is modified to drive the execution along a path that ensures propagation if possible. This repeats until the semantic effect of c is observable in the output.

A simplification of the process is shown in Figure 2.11. For each diagram the left (and black) line depicts the possible augmented execution trace[5] for the original program P. The right (and red) line shows the augmented history for the modified program P'. A deviation of both lines indicates that the same input begins to produce different states in both versions at this point.

Test	**Input**	**P**	**P'**
t_1	{0,−1}	$o = 0$	$o' = 1$
t_2	{0, 0}	$o = 0$	$o' = 2$

Figure 2.12: Behavioral Differences between P and $P'\backslash\{ch1, ch2\}$

The generated concrete test case is only one witness of the changed behavior. The syntactic change could modify the behavior of the program in more than one way. Qi et al. give an approach that shows *some path* that exposes changed behavior due to the change. A regression error may only be exposed on another path leading to the output. Figure 2.12 shows two test cases witnessing a behavioral difference between original version P and a configuration for which only the single change $ch1$ is applied to P. In theory, even for a single change there may be infinitely many paths that exercise the changed statement and produce a different output in both versions (e.g., if a loop condition depends on the input).

2.6.4 Propagation of Multiple Changes

When multiple statements are changed, they may semantically interfere [42] or interact [54] subtly and unintendedly when executed on some input but not on others. Program changes potentially semantically interfere if the static forward slices of the changed statements intersect in the changed program [41, 42]. Then, the changes cannot be tested in isolation. For a given input, the semantic effect of one change onto a statement may be masked or augmented by the semantic effect of another change onto that statement.

[5]The augmented execution trace is the sequence of executed program statements plus respective, relevant program states [54].

Santelices et al. [2] discuss the feasibility of the approach of propagating a single change to a minimal distance [112] in the presence of multiple (non-interfering) changes. For each change there has to exist a path from the program entry to the changed statement that does not contain another changed statement. This ensures that the semantic effect of an "earlier" change is not accidently propagated to a statement that is supposed to be infected by the given change.

Harman et al. [26] call potentially semantically interfering sets of code changes "higher-order mutants" and aim to generate a test case that for a given higher-order mutant produces different output in both versions. First, using the control-dependence graph of the changed program, the technique computes a path that may execute all changes. Then, using this information and a hill climbing algorithm with random restart, the technique generates a test case that exercises every changed statement of the given higher-order mutant. Lastly, the technique searches paths that are more likely to propagate the combined semantic effects to the output.

Given a set of changes C, there are $2^{|C|-1}$ subsets of C that potentially semantically interfere and have to be tested. For example, our running example has three changes yielding four possibly interfering change sets (cf. Fig. 2.5 on page 13). Yet, even for a single subset, the search for a difference-revealing test case may never terminate, which renders this procedure prohibitively expensive. Even if the search yields an input the produces different output on both program versions, this input may not be a witness of software regression.

2.6.5 Semantic Approaches to Change Propagation

While syntactic techniques seek to explicitly reach at least one change and propagate its semantic effect to the output, semantic techniques compute differences based on the transformation functions of original and modified program version [39, 113]. Path exploration based on the symbolic output can reveal the transformation function of a program [19] - the *symbolic program summary*. This summary is an (incomplete) list of input partitions. Each input in the same partition computes the same symbolic output. Given the program summaries of two program versions, a behavioral difference is exposed by input that computes different output. In other words, if for overlapping input partitions the output is computed differently, then every input in this intersection exposes a behavioral difference.

Figure 2.13 lists the symbolic program differences for the two versions in our running example. The respective symbolic summaries are shown in Figure 2.4 on page 11. The intersection is found by conjoining every input condition and

$\{i,\ j\}$	**Input**	**Output**	**Diff**
$\{-1,\ 0\}$	$i \leq -1$	$o = o' = 0$	
$\{0,-1\}$	$i > -1 \wedge i \leq 0 \wedge j \leq -1$	$o = 0 \wedge o' = 1$	x
$\{0,\ 0\}$	$i > -1 \wedge i \leq 0 \wedge j > -1$	$o = 0 \wedge o' = 2$	x
$\{1,-1\}$	$i > 0 \wedge j \leq -1$	$o = o' = 1$	
$\{1,\ 0\}$	$i > 0 \wedge j > -1 \wedge j \leq 0$	$o = 1 \wedge o' = 2$	x
$\{1,\ 1\}$	$i > 0 \wedge j > 0$	$o = o' = 2$	

Figure 2.13: Symbolic Program Difference for P and P'

testing for satisfiability. Note that input exercising the same path in P, $i \leq 0$, can exercise three paths in P'.

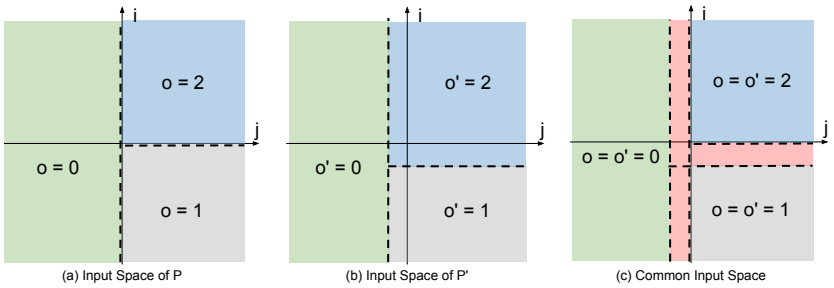


Figure 2.14: Visualization of overlapping Input Space Partitions

The visualization of the input space partitioning is shown in Figure 2.14. The set of all values for input variables i and j forms a 2-dimensional vector space. Diagram 2.14.a) shows the input space of the original program P partitioned in terms of the output values. Diagram 2.14.b) depicts the input space of the modified program P' partitioned in terms of the output values. The overlapping of the partitioning of both input spaces is visualized in Diagram 2.14.c) for the common input space. The red plane in between the gray, blue and green planes represents input that executed on both versions compute different output.

	Input	**Output**
$\Delta_{P,P'\setminus\{ch1,ch2\}} = \Delta_P$	$b(i,j) > 0$	$o = 2$
$\Delta_{P'\setminus\{ch1,ch2\},P} = \Delta_{P'\setminus\{ch1,ch2\}}$	$b(i,j) > 0$	$o' = o(i,j) + 1$

Figure 2.15: Partition-Effect Deltas for P w.r.t. $P'\setminus\{ch1, ch2\}$, and vice versa.

By summarizing the behavior of code blocks that are common in both versions as uninterpreted functions, we can derive the *abstract program summary* for each program [39]. The abstract summaries can be used to compute the partition-effect deltas. Such a delta reflects input partitions and their associ-

ated effects, present in one version that are not present in the other version of a program. Figure 2.15 shows the functional delta computed for the abstract summaries in Figure 2.7 on page 16. If there exists input that renders $b(i,j) > 0$ satisfiable, then there may be a difference in output observable. More specifically, if there exists an assignment to i and j that renders $b(i,j) > 0 \wedge o(i,j) \neq 1$ satisfiable, then this input is a witness of semantic difference. Note, for the versions P and $P' \backslash \{ch1, ch2\}$ there does not exist such an input.

Korel and Al-Yami [58] present a technique that given two program versions reduces the problem of generating input that exposes a behavioral difference to the problem of reaching an assertion. The technique generates a test driver that wraps both program versions and adds the assertion that both versions compute the same output values. Then the technique searches for a witness that violates that assertion using a hill climbing strategy similar to the one presented in Reference [66]. This witness is a difference-revealing input for both program versions.

2.6.6 Random Approaches to Change Propagation

Random test generation techniques can provide test cases that, when executed on both program versions, reveal a difference [57, 101]. The procedure is depicted in Figure 2.16.

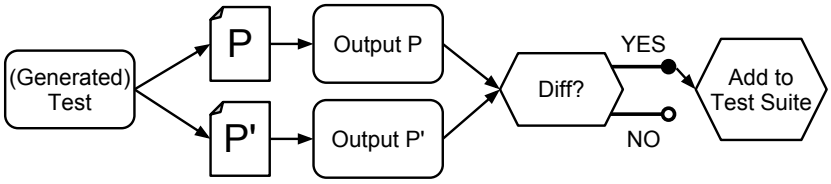


Figure 2.16: Behavioral Regression Testing

Jin et al. [57] present a technique to generate random input, execute it on both versions, and report such cases that yield different output. The proposed technique determines the syntactic difference between two versions through static analysis. Leveraging Randoop [108] as random test generation engine, a large body of test inputs are generated for the set of changed classes. The generated test suite is then run on both versions of those classes, the output compared, and the differences in output reported as behavioral differences. A challenge of the technique is the change of method signatures from one version to the next. The same unit test cases cannot be executed on both versions. While this is a scalable approach, Santelices and Harrold [110] empirically show

that regression errors in low-probability domains are likely to stay undetected when sampling a normal distribution.

Input	**Diff**	**Solutions**	**Probability**
$i \leq -1$		$(2^{31}-1)*(2^{32}-1)$	~ 0.5
$i > -1 \wedge i \leq 0 \wedge j \leq -1$	x	$1*(2^{31}-1)$	2^{-33}
$i > -1 \wedge i \leq 0 \wedge j > -1$	x	$1*2^{31}$	2^{-33}
$i > 0 \wedge j \leq -1$		$(2^{31}-1)*(2^{31}-1)$	~ 0.25
$i > 0 \wedge j > -1 \wedge j \leq 0$	x	$(2^{31}-1)*1$	2^{-33}
$i > 0 \wedge j > 0$		$(2^{31}-1)*(2^{31}-1)$	~ 0.25

Figure 2.17: Random Input reveals a difference with probability $3 * 2^{-33}$

For our running example, Figure 2.17 shows the probability to reveal a difference if we consider i and j to be 32-bit signed integers that are randomly generated. The probability to randomly generate difference revealing test cases is $3 * 2^{-33}$ (about two magnitudes smaller than winning the UK-lottery). In practical terms, setting a bound to -2 and 2 (or -10 and 10), the probability to generate a difference-revealing test case would be 0.28 (or 0.07, respectively) even though every input exercises at least one change.

2.7 Chapter Summary

Software testing remains the most important form of software validation despite advances in program analysis, model checking, and theorem proving via Satisfiability Modulo Theory (SMT) solving. Each of these techniques provide a different perspective of program checking. Program analysis tries to find "bugs" by inferring program properties. Model checking attempts to find problematic test inputs by searching a large search space. Finally, theorem proving is inherently different - it attempts to prove programs correct via deduction, rather than generating potentially problematic test cases.

Despite the huge advances in constraint solving, search space representation and exploration (for model checking) and theorem proving - testing still remains hugely popular. Why? This is because of the concrete outcome from testing as an activity - once the test cases are generated - the activity of testing immediately points us to a potential bug if the test case fails. This aspect of testing is further magnified, when we want to validate a new program version against absence of regressions. Notably while testing a single program version - a notion of "expected output" is needed to validate the observed program output. However, while testing a program version against a previous version to check for regressions - we can often compare the output of the current program version with the previous program version. Thus, testing a program version against

regressions from the previous version can immediately lead us to failing tests which expose potential regressions.

Chapter 3

Partition-based Regression Verification

> „Divide et Impera."
>
> — Julius Caesar, 100 BC – 44 BC

In this chapter, we introduce a technique that can effectively show the absence of regression for all input (i.e., Regression Verification; RV) and improves the efficiency of regression verification by allowing gradual and partial verification using dependency analysis and symbolic execution. The main observation is that software regression is observable only for input that exposes a difference when executed on the original and modified program versions. The complexity of the underlying syntactic changes is irrelevant for this technique, as it directly explores the changed behavior resulting from the syntactic changes.

Partition-based Regression Verification (PRV) is an approach to RV based on the *gradual* exploration of differential input partitions. A differential input partition is a subset of the common input space of two program versions that serves as a unit of verification. Instead of proving the absence of regression for the complete input space at once, PRV verifies differential partitions in a gradual manner. If the exploration is interrupted, PRV retains *partial verification guarantees* at least for the explored differential partitions. This is crucial in practice as verifying the complete input space can be prohibitively expensive.

Experiments show that PRV provides a useful alternative to state-of-the-art regression test generation techniques. During the exploration, PRV generates test cases which can expose different behaviour across two program versions. However, while test cases are generally single points in the common input space, PRV can verify entire partitions and moreover give feedback that allows programmers to relate a behavioral difference to those syntactic changes that contribute to this difference.

3.1 Introduction

Software verification seeks to guarantee the absence of errors in a program, but is rather expensive in practice. There are two main reasons: 1) verification requires specifications, which may be difficult to write and maintain; and 2) the verification process can be very time-consuming.

However, there is some hope for an inexpensive form of *Regression Verification* (RV) [114, 115]. The goal of RV is not to verify the correctness of a program ad absolutum but relative to an earlier version. Thus, RV seeks to guarantee the absence of regression errors. This more modest goal allows RV to avoid separate forms of formal specifications. The previous version serves as sufficient specification for checking whether the changed version is *at least as correct* as the previous version.

Yet, in practice, RV for all inputs is very time-consuming. Godlin and Strichman [114] proposed a decision procedure that takes two program versions and either proves behavioral equivalence (thus the absence of regression) or provides a witness of behavioral difference. The authors report that the verification of non-equivalent versions can take a long time to terminate or run out of memory. In fact, generally proving the equivalence between two programs is an undecidable problem. While the termination of RV provides strong regression guarantees for all inputs, the interruption of the verification procedure (due to time or memory constraints) yields no guarantees at all.

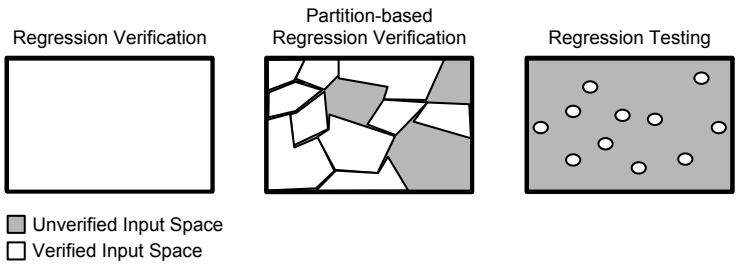


Figure 3.1: PRV versus Regression Verification and Regression Testing

This chapter presents *Partition-based Regression Verification* (PRV), a *gradual* approach to RV based on the exploration of *differential partitions*. A differential partition is a subset of the common input space of two program versions that serves as unit of verification. Instead of verifying the entire input space at once, PRV allows gradually verifying such partitions one-by-one. As illustrated in Figure 3.1, PRV shares the advantages of both, Regression Testing (RT) and RV. *Like RV*, if all differential partitions are shown equivalent, then PRV guarantees the absence of regression errors for all inputs. More importantly, PRV allows a form of *partial verification*: if the verification procedure is

interrupted, PRV guarantees the absence of regression errors for the explored partitions that are shown equivalent. Thus, *like RT*, PRV allows the gradual checking for regression. However, while RT provides verification guarantees only for the concrete, executed sample inputs, PRV seeks to guarantee the absence of regression for entire input partitions. In practice, this partial verification approach is crucial, as verifying the complete input space can be infeasible due to time or other resource constraints.

Technically, differential partitions are computed using a form of symbolic execution [116] and require *deterministic* program execution. In contrast to other input partitioning techniques [80, 21], differential partitioning accounts for the inputs of two programs. A differential partition is characterized by a symbolic condition that defines a range (or subset) of valid input for that partition. Input is grouped according to whether it reaches the same syntactic changes and whether it propagates the same differential state to the output. If an input computes the same output in both versions, the respective partition is said to be *equivalence-revealing*. In such case, both versions are soundly guaranteed to compute the same output for all input satisfying the symbolic condition. Otherwise, the respective partition is said to be *difference-revealing*.

```
1 input(i);
2 a = 0; o = 0;
3 i = i; //i=i+1
4 if(i>0)
5   a++;
6 if(a>0)
7   o=i;
8 output(o);
```

(a) Programs P and P'

	Input	Output
P	$i \leq 0$	$o = 0$
	$i > 0$	$o = i$
P'	$i + 1 \leq 0$	$o' = 0$
	$i + 1 > 0$	$o' = i + 1$
PRV	$i < 0$	$o = o'$
	$i = 0$	$o = 0 \wedge o' = i + 1$
	$i > 0$	$o = i \wedge o' = i + 1$

(b) Differential Partitions

Figure 3.2: Running Example (Incomplete Bugfix)

Figure 3.2 illustrates differential partitions in a concrete example. Program P computes output o based on the values of input i and is changed to P' by substituting line 3 with the commented code. Figure 3.2.b shows the symbolic output that is computed based on the evaluation of the input variables for both programs. The bottom three rows depict one equivalence-revealing and two difference-revealing input partitions. Note that the analysis of only a single version is insufficient to expose all interesting subsets of input. In particular, a test suite $T \leftarrow \{i = -1, i = 1\}$ covers all paths in both programs and even reveals a difference. However, input $i = 0$ is a missing test case that could represent a regression error. Intuitively, it is interesting because the branch in line 4

is evaluated in different directions in both versions. PRV explores a distinct, difference-revealing partition for this input.

PRV provides an alternative to *regression test generation* techniques [58, 1, 25, 100]. Upon allowing the continued exploration even of difference-revealing partitions, the developer may (in)formally verify such partitions. The test cases generated for each difference-revealing partition can be checked against the developer's expectation. The program slice, used to compute the partition, can be inspected to determine the changed statements contributing to the difference. The symbolic conditions and summaries (cf. Fig. 2.b) can be further analyzed by tools.

Our initial experience with PRV is very encouraging. For the studied subjects, PRV efficiently exposes regression errors that are not detected by the considered test generation methods.

In summary the *main contributions* of this paper are:

- *A gradual approach to regression verification* that continuously verifies the input space of a program against another version of that program to find regression errors. If the verification procedure is interrupted, PRV guarantees the absence of regression errors for the explored input space that has been shown equivalence-revealing.
- A *differential partitioning* technique, based on symbolic execution, that soundly partitions the input of two versions. The partitioning technique symbolically groups input of the two programs, and creates partitions which either guarantee behavioral equivalence, or expose differences for a certain subset of inputs.
- *An alternative to regression test generation*. The approach can be used to generate test cases for partitions where differences are found. As illustrated by our experimental evaluation, finding such test cases is competitive with state-of-the-art regression test generation techniques.
- The *implementation* and *experimental evaluation* of PRV.

3.2 Longitudinal Input Space Partitioning w.r.t. Changed Behavior

This section formalizes the concepts and operations for subdomains of the program input space that are disjoint and homogeneous w.r.t. the behavior of an evolving program. We start by providing a background on partitioning the input space w.r.t. the behavior of a program into so-called *behavior partitions*. Then, we introduce the partitioning of the common input space of two successive program versions w.r.t. behavioral difference (and equivalence) into

so-called *differential partitions*. Finally, we extend this approach to the context of multiple successive program versions.

3.2.1 Background: Behavior Partitions

A behavior partition is an input subdomain that is homogeneous w.r.t. the computed symbolic output formula. A *symbolic output formula* is a symbolic expression in terms of the program inputs that is computed along the paths exercised by input in the related subdomain. Formally, a behavior partition is a tuple where the first value is a set of inputs, and the second is the symbolic output formula that is computed for every input in the set of inputs:

$$Behavior = Input \times Output \tag{3.1}$$

Given the program, the set of behavior partitions can be computed by systematic exploration of the program's input space using the function *partition*:

$$partition : Program \rightarrow \{Behavior\} \tag{3.2}$$

The computation of behavior partitions employs symbolic execution and program dependency analysis techniques (i) to group paths that compute the same symbolic output formula and (ii) to enumerate every such group of paths. The symbolic condition associated with a group of paths defines the set of inputs, while the operations computing the output along the exercised paths define the symbolic output formula. A concrete algorithm implementing *partition* is presented in Reference [19].

Example 1

Consider the programs P and P' in Figure 3.2. These are the behavior partitions of the two programs:

$$partition(P) = \{(i \leq 0, 0), (i > 0, i)\}$$
$$partition(P') = \{(i + 1 \leq 0, 0), (i + 1 > 0, i + 1)\}$$

Note that the set of behavior partitions resulting from $partition(P)$ and $partition(P')$ are also shown in a tabular format in Figure 3.2.

Depending on the program (and implementation of *partition*), the number of behavior partitions that need to be enumerated can be infinite. In fact, the exploration algorithm presented in Reference [19] may never terminate. However, the exploration can be interrupted at any time, providing at least a partial set of symbolic output formulae for the explored input space.

3.2.2 Differential Partitions

A differential partition is an input subdomain in the common input space of two successive program versions, P and P', that is homogeneous w.r.t. behavioral difference and equivalence, respectively. For instance, if an input yields the same output across both versions, then every input in the same differential partition yields the same output across both versions. Compared to behavior partitions, differential partitions are concerned with the symbolic output formula computed for the subsequent version *relative* to that computed for the previous version.[1]

First, we define the *type* of differential partitions. Formally, a differential partition is a tuple where the first value is a set of inputs and the second is the differential symbolic output that is computed for every input in the set of inputs:

$$DBehavior = Input \times DOutput \tag{3.3}$$

The *differential symbolic output* can be either EQ, which marks equivalent output for the corresponding set of input, or a tuple where the first value is the symbolic output formula computed by the previous program version and the second is the symbolic output formula computed by the subsequent program version:

$$DOutput = EQ + Output \times Output \tag{3.4}$$

Then, we define the *properties* of differential partitions. A differential partition can be equivalence- or difference-revealing.

Definition 1 (*Differential Partition*)

> *Given two successive program versions P and P', inputs in any differential partition $d_{P,P'}$ have the following property: either all inputs in $d_{P,P'}$ produce the same output in P and P' (an equivalence-revealing partition), or all inputs in $d_{P,P'}$ produce different outputs in P and P' (a difference-revealing partition).*

While a *difference-revealing partition* is associated with the concrete tuple of symbolic output formulae computed in both versions, an *equivalence-revealing partition* uses the constant EQ to represent that the computed output is equivalent. In fact, in the context of regression verification the concrete output computed by both version is irrelevant for equivalence-revealing partitions simply because it is guaranteed that the subsequent version is as correct as the previous version for input that does not reveal a difference of behavior. As such, the

[1] Henceforth, we refer to P as previous and P' as subsequent program version.

abstraction EQ for equivalence-revealing partitions allows for important optimizations during the implementation of the exploration algorithm (as discussed in Section 3.2.5).

Now, we define the *operations* required to compose the defined types. Formally, given the symbolic output formulae of two successive program versions, the corresponding differential output is derived as follows:

$$\begin{aligned} \overset{!}{=} &: \mathit{Output} \times \mathit{Output} \to \mathit{DOutput} \\ o_1 \overset{!}{=} o_2 &= \begin{cases} EQ & \textit{only if the values of } o_1 = o_2 \\ (o_1, o_2) & \text{otherwise} \end{cases} \end{aligned} \tag{3.5}$$

Intuitively, differential partitions can be derived by *intersecting* the behavior partitions of the previous and subsequent version to derive the differential output as defined above. One way of checking equivalence (i.e., whether $o_1 = o_2$) is by matching the symbolic expressions that represent the behavior partitions. Formally, the set of differential partitions can be derived from the two sets of behavior partitions as follows:

$$\begin{aligned} \otimes &: \{\mathit{Behavior}\} \times \{\mathit{Behavior}\} \to \{\mathit{DBehavior}\} \\ s_1 \otimes s_2 &= \{((c_1 \cap c_2), (o_1 \overset{!}{=} o_2)) \mid (c_1, o_1) \in s_1 \wedge (c_2, o_2) \in s_2 \wedge (c_1 \cap c_2) \neq \emptyset\} \end{aligned} \tag{3.6}$$

As such, the set of differential partitions can be computed as cross-product of both sets of behavior partitions by intersecting the corresponding input subdomains $(c_1 \cap c_2)$. Note, that a *challenge* of the actual exploration algorithm is to derive much "larger" differential partitions than by the naïve intersection of behavior partitions. If the intersection is non-empty, a differential partition is derived as a tuple with the first value set to the intersection of both input subdomains, and the second value set to the differential output as per Equation 3.5.

Example 2

In Example 1 we have seen the behavior partitions for the programs P and P' shown in Figure 3.2. These are the corresponding differential partitions:

$$\begin{aligned} \mathit{partition}(P) \otimes \mathit{partition}(P') = \{&(i < 0, EQ), \\ &(i = 0, (0, i + 1)), \\ &(i > 0, (i, i + 1))\} \end{aligned}$$

Note that the set of differential partitions are also shown in a tabular format in Figure 3.2.

3.2.3 Multi-Version Differential Partitions

The concept of differential partitions can be generalized to multiple successive program versions, $\mathbb{P} = \{P^0, P^1, \cdots, P^n\}$. For instance, if an input yields the same output across all versions $\mathbb{P}$, then every input in the same multi-version differential partition yields the same output across all versions $\mathbb{P}$. Intuitively, one may think of the intersection of all differential partitions between every successive version pair.

First, we override the earlier definition of the differential symbolic output to account for $n \geq 2$ symbolic outputs (see Eqn. 3.4). The *differential symbolic output* can be either EQ, which marks equivalent output for the corresponding set of input, or an n-tuple where the element at $k-th$ position is the symbolic output formula computed by program version P^k:

$$DOutput_n = EQ + [Output]_n \tag{3.7}$$

Then, we formally define the properties of multi-version differential partitions which can be equivalence- and difference-revealing.

Definition 2 (*Multi-Version Differential Partition*)

Given a set of successive program versions, $\mathbb{P} = \{P^0, P^1, \cdots, P^n\}$, inputs in any multi-version differential partition $d_{\mathbb{P}}$ have the following property: either all inputs in $d_{\mathbb{P}}$ produce the same output for all version $P^k \in \mathbb{P}$, (an equivalence-revealing partition), or all inputs in $d_{\mathbb{P}}$ produce different outputs for at least two versions $\{P^k, P^l\} \subseteq \mathbb{P}$ (a difference-revealing partition).

Similarly, we extend the *operations* required to compose the generalized types, like $\stackrel{!}{=}$ and $\otimes$. Formally, given the symbolic output formulae of a set of successive program versions, $\mathbb{P}$, the corresponding differential symbolic output is derived as follows:

$$\begin{gathered} \langle . \rangle : [Output]_n \rightarrow DOutput_n \\ \langle o_1, \cdots, o_n \rangle = \begin{cases} EQ & \text{if } o_i = o_j \mid 1 \leq i < j \leq n \\ (o_1, \cdots, o_n) & \text{otherwise} \end{cases} \end{gathered} \tag{3.8}$$

Intuitively, multi-version differential partitions can be derived by intersecting all behavior partitions between every successive version pair, $\{P^k, P^{k+1}\} \subseteq \mathbb{P}$, to

derive the differential output as defined above. Formally, the set of multi-version differential partitions can be derived as follows:

$$\begin{aligned} \otimes_n :&[\{Behavior\}]_n \rightarrow \{DBehavior_n\} \\ \otimes(s_1, \cdots, s_n) =&\{(c_1 \cap \cdots \cap c_n, \langle o_1, \cdots, o_n \rangle) \mid (c_i, o_i) \in s_i \wedge (c_1 \cap \ldots \cap c_n) \neq \emptyset\} \end{aligned} \tag{3.9}$$

This generalized form of differential partitions can also be computed as cross-product of behavior partitions by intersecting the corresponding subdomains $(c_1 \cap \ldots \cap c_n)$. The challenge of practical exploration algorithms is to derive much coarser partitions (e.g., using abstractions such as EQ; see Eqn. 3.8).

3.2.4 Deriving the Common Input Space

Differential partitions exist in the *common input space* of successive program versions. The input space of a program P is defined by the program's input variables and can be taken to be a finite, measurable metric space with d dimensions. Note that d corresponds to the number of input variables for the program.

Example 3

Let P be a program with two integer input variables, a and b. The program P has a two-dimensional input space spanned by the orthogonal vectors a and b. Now, every point in that space is a concrete assignment to the input variables. For instance, the point $(1, 3)$ assigns $a = 1$ and $b = 3$.

If the order and number of input variables does not change across two versions, the common input space is trivially the same in both versions. If the *order* of input variables changes across two versions, the common input space can be computed by applying the corresponding transposition to the input space of the changed version. If the *number* of input variables changes across two versions, the dimensionality of the input space does as well. Let the dimension of a given program P be $d = n$. Then the input space of the changed program P' is $d' = n + m - o$, where m is the number of vectors for the input variables added to P and o is the number of vectors for the input variables removed from P. So, the common input space is taken to be with dimensionality $\hat{d} = n + m + o$, containing *all* input variable vectors, where the input spaces of P and P' are transposed accordingly. A practical discussion on the change of input domain can be found in [58].

The input space transpositions have no practical impact on the presented algorithms. A differential partition is a subdomain in the common input space of two program versions. In practice, such a subdomain is defined as a condition

on the input variable vectors identifying each vector by name. The renaming of input variable vectors must be handled.

Example 4

Let P' be the changed version of P of the previous example. If we neither reorder nor remove one of the input variables, a and b, then the input space is trivially the same in P and P'. If in P' the input variables are reordered, a after *b, then the input space of P is that of P'* transposed. *If in P' input variable b is removed, the common input space is that of P, while if in P' an input variable c is added, the common input space is that of P'. Note, if the new input variable c is actually used to compute the output in P', then at least one statement will be changed that exists in both versions.*

3.2.5 Computing Differential Partitions as Composition of Behavior Partitions

Naïvely, differential partitions can be computed by 1) computing the complete set of behavior partitions of each program version as in [19] and 2) intersecting both sets of behavior partitions to compute the differential symbolic output. However, the naïve approach has two important drawbacks:

- **Too fine-grained partitions**. When computing equivalence-revealing partitions, the *particular* symbolic output formula that is produced by both versions is not relevant as long as they are the same. For instance, input that does not execute a changed statement will never compute two different symbolic output formulae for two versions. To find a differential partition with differential symbolic output EQ, it is sufficient to determine groups of input that does not execute a changed statement. This optimization is not available in the naïve approach.

- **Combinatorial Explosion**. In the naïve approach, much time is wasted checking whether the behavior partitions of two versions actually do intersect. Naturally, there are many more behavior partitions that *do not* intersect. Moreover, the number of behavior partitions in one version may be very large, such that one may abort the partition exploration after a time bound is reached and only intersect the explored partitions. There is no guarantee that the explored behavior partitions actually do overlap. A more systematic approach to exploring differential partitions is needed.

In the next section we present an algorithm that does not suffer from these drawbacks and computes differential partions more efficiently and on-the-fly.

3.3 Regression Verification as Exploration of Differential Partitions

Partition-based Regression Verification (PRV) takes two successive program versions and continuously verifies *differential partitions* – to check whether the subsequent version is at least as correct as the previous version. The gradual regression verification can be interrupted at any time. In this case, the regression guarantees are retained for the (partially) verified input space. For every partition, PRV generates a concrete sample input that is added to regression test suite T. In theory, like other regression verification techniques, PRV can terminate with a difference-revealing test case. In practice however, PRV can continue the gradual exploration even of difference-revealing partitions, so as to allow the programmer to check the corresponding difference-revealing test cases in T. After all, the output difference may be intended (e.g., new feature) or unintended (e.g., regression error).

The intuition of partition-based regression verification is presented in Figure 3.3, while the detailed procedure is outlined in Algorithm 1. Later, in Theorem 2, we will claim the *exhaustiveness* of this exploration algorithm.

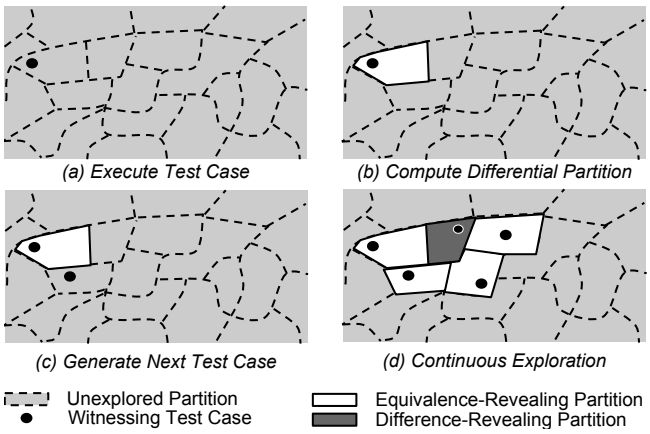


Figure 3.3: Exploration of Differential Partitions

The exploration starts with a random test case in the queue. Depicted as black dot in Figure 3.3.a) this random test case t is taken from the queue and executed upon both versions. Test case t is a point in the common input space[2] of both versions, representing concrete knowledge about the differential behavior. In Figure 3.3.b), input is grouped into a differential partition that yields the same differential behavior as t. This input exercises all those statement in-

[2]The derivation of the common input space for versions with different input spaces is discussed in [58], e.g., the new version has one more input variable.

stances that are "relevant" to the reachability and propagation of the syntactic program changes exercised by t. Hence, in Algorithm 2 the symbolic condition is computed as a conjunction of the pertinent branch conditions. Later, in Theorem 1, we will claim the *soundness* of this generalization from a test case to a differential partition.

Algorithm 1 Partition-based Regression Verification

Input: Versions P and P', Changed stmts in both versions (C, C')

1: let $queue \leftarrow \emptyset$
2: let $T \leftarrow \emptyset$
3: let $V \leftarrow \emptyset$
4: add $randomInput()$ to $queue$
5: **while** $queue \neq \emptyset$ **do**
6: let $t \leftarrow chooseNextTestcase(queue)$
7: let $condition \leftarrow computeDPartition(t, P, P', C, C')$
8: call $generateAdjacentTestcases(condition, queue)$
9: add t to T
10: add $condition$ to V
11: **end while**

Output: Verified Input Space V, Regression Test Suite T

As depicted in Figure 3.3.c), the next test case is executed outside of the explored input space. To generate such "adjacent" test cases, the constituent branch conditions are negated one-by-one (cf. Alg. 4), similar to other path exploration techniques. This yields a number of intermediate constraints. If a constraint solver finds a satisfying witness to one of these constraints, then it is added to the queue waiting to be executed.

As depicted in Figure 3.3.d), after the execution of the next test case from the queue, again, the corresponding differential partition is computed. This procedure repeats until all differential partitions are explored or some (time) budget is exhausted. A *search strategy* would assign some distance or fitness to each constraint and decide the order in which the partitions corresponding to intermediate constraints are explored. This is implemented in the procedure *chooseNextTestcase* (not listed). In particular, PRV takes from the queue in the order they arrive but prioritizes test cases that promise 1) different output,[3] 2) the propagation of already exercised changes and 3) the execution of another set of changes, in that order. Finally, every executed test case is added to the regression test suite T. Each test case is a witness of one differential partition. The set of explored differential partitions V represents the verified input space.

[3] An adjacent test case may witness different output if it is generated by negating $(o = o')$ as constituent of a propagation condition; see Sec. 3.3.3.

3.3.1 Computing Differential Partitions

The computation of the differential partition for a given test case is presented in Algorithm 2. It implements the functionality of procedure *computeDPartition* called in Algorithm 1 and requires determinism - for every execution of the same input on the same program the same output is computed. Also, the deletion of variable assignments (e.g., `x=x++`) in P is represented by dummy-statements (e.g, `x=x`) in P' (cf. [2]).

Upon execution of the test case t on both programs, P and P', the symbolic condition is computed. Input that does not exercise a syntactic change or that does not propagate the differential state to the output is equivalence-revealing. If t does not exercise a changed statement, then PRV employs the *reachability condition* (Def. 5) to group input that does not execute a change for the same "reason". If t exercises at least one changed statement but yields the same output in both versions, then PRV employs the *propagation condition* (Def. 6).

Algorithm 2 - Procedure *computeDPartition*

Input: Input t, Versions P and P', Changed stmts in both versions (C, C')

let trace $\pi \leftarrow execute(t, P)$
let trace $\pi' \leftarrow execute(t, P')$
let $condition \leftarrow false$
if not exist an instance of $c' \in C'$ in π' **then**
    let $condition \leftarrow \bigwedge_{c' \in C'} reach(c', \pi')$
else
    let o_i be the instance of output o in π
    let o'_i be the instance of output o in π'
    if $value(o_i) = value(o'_i)$ **then**
        let $condition \leftarrow prop(o, \pi, \pi', C, C')$
    else
        let $condition \leftarrow \mathit{diff}(o, \pi, \pi', C, C')$
    end if
end if

Output: Condition *condition*

Input that yields different output is difference-revealing. If t yields different output in both program versions, then PRV employs the *difference condition* (Def. 7) to group input that computes different output for the same "reason". These reasons are defined upon the exercised dynamic and static program dependencies, as enunciated in the following.

3.3.2 Computing Reachability Conditions

Intuitively, an input t does not execute a changed statement c because the conditions of the branch instances s_i upon which c statically control-depends are evaluated in the direction that does not favor the execution of c.

```
1  input(i,j);
2  a = 0; b = 0;
3  if(i>0)
4    a=1;
5  for(c=0; c < j; c++)
6    b += c;
7  if(j>0){
8    if(a>0)
9      //change c
```

Figure 3.4: Intuition of Reachability Condition

An example is shown in Figure 3.4. Input $(0, 1)$ does not execute the changed statement in line 9. Why? Because the branch in line 8 is not evaluated to `true`. This is because the condition in line 7 is evaluated to `true` and the condition in line 3 to `false`. The remainder of this section explains the computation of the reachability condition based on the relevant slice of the branch in line 8.

Definition 3 (*Relevant Slice [18, 17]*)

Given an execution trace π and a statement instance s_i in π, the relevant slice *of s_i in π contains all statement instances r_i in π that are in the transitive closure of dynamic data, control- and potential dependence of s_i.*

A statement instance s_i *potentially depends* [18] on conditional statement instance r_i in path π iff. there exists a variable v used in s_i such that (1) v is not defined between r_i and s_i in π but there exists another path σ from r_i to s_i along which v is defined, and (2) evaluating r_i differently may cause this untraversed path σ to be executed. Unlike data- or control-dependence, the potential dependence accounts for the potential difference in value of s_i if the branch (upon which s_i potentially depends) was evaluated differently (see Sec. 2.2).

Note that relevant slices have a desirable property: If two inputs t_0 and t_1 exercise the same relevant slice computed w.r.t. a statement instance s_i, then the variables used in s_i have the same symbolic values for t_0 and t_1 [19]. Relevant slices are used to define the reachability, propagation, and difference conditions. The property of relevant slices is utilized to prove Theorem 1, establishing that these conditions indeed characterize differential partitions as defined in Definition 1.

Definition 4 (*Reachability Slice*)

> *The reachability slice of statement c in trace π is the union of the relevant slices of all instances s_i in π of every statement s that c transitively, statically control-depends on.*

Dually, the statements in the remainder of π are not relevant w.r.t. reaching c. That is, whether or not these statement instances are executed is not relevant to (not) reaching c.

Definition 5 (*Reachability Condition*)

> *The reachability condition, $reach(c, \pi)$, computed over the trace π w.r.t. statement c is the path condition computed over the statement instances of π that are included in the reachability slice of c in π.*

If an input t_0 does not exercise statement c, then every input t_1 satisfying $reach(c, \pi(t_0, P))$ does not exercise c. A *path condition* is a quantifier free first order logic formula on program inputs. Any test input satisfying the path condition of a path π is guaranteed to also exercise all statement instances in path π. The negation of a constituent branch condition in the reachability condition computed w.r.t. statement c may change the reachability of c.

3.3.3 Computing Propagation Conditions

Intuitively, an input t does not propagate the semantic effect of the exercised changes to the output because certain statement instances N_i upon which the output dynamically depends carry the same values in both versions. On a high level, N_i represents the point where the differential program states converge. Any attempt to negate a branch beyond that point to propagate a difference in program state is futile.

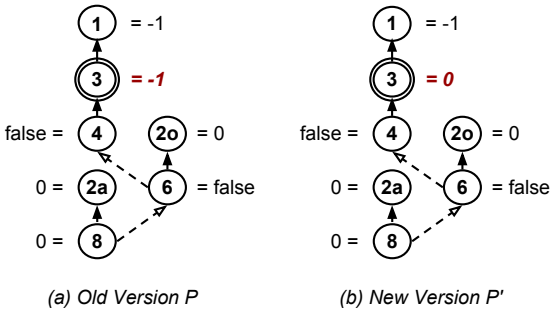


Figure 3.5: Intuition of Propagation Condition

Figure 3.5 shows the dynamic dependency graphs augmented by concrete values and computed for the execution of input (-1) upon the version pairs in

Figure 3.2. The dashed arrows indicate potential dependence while the concrete arrows indicate either dynamic data- or control-dependence. Each node is labeled with the line number of the statement instance it represents. The values for the instance of line 3 are different in both versions. That is, the program state is "infected" after the execution of the change. However, the value of the output in line 8 is the same for both versions. Why?

The semantic effect of the change in line 3 is not propagated to the output in line 8 for the execution of (-1) on P and P' because the branch in line 4 is evaluated in the same direction in both versions even though it dynamically depends on the statement in line 3, which carries different values in both versions. In the remainder of this section, we explain how the instance of line 4 is added to the convergence set N_i and define the propagation condition based on N_i.

As shown in Algorithm 3, both dynamic dependency graphs (DDGs) are computed over the traces π and π' for the execution of input t on both versions P and P'. The DDGs are augmented by potential dependencies and the concrete values for the variables used in every node. Output instances o_i in π and o_i' in π' are aligned and passed into procedure PropAlign to compute N_i recursively.

Algorithm 3 Computing Differential State Convergence N_i

Input: Execution Traces π and π', Output Statement o

1: $aDDG \leftarrow augmentedDDG(\pi)$
2: $aDDG' \leftarrow augmentedDDG(\pi')$
3: $(o_i, o_i') \leftarrow alignableOutput(aDDG, aDDG', o)$
4: $N_i \leftarrow \emptyset$
5: **if** $isChanged(o_i')$ **then** add (o_i, o_i') to N_i
6: **else** call PropAlign(o_i, o_i')
7: **procedure** PropAlign(s_i, s_i')
8: $R_i \leftarrow s_i.getDependsOn()$
9: $R_i' \leftarrow s_i'.getDependsOn()$
10: **for all** $r_i' \in R_i'$ **do**
11: **if** $\neg isChanged(r_i') \wedge \exists r_i \in R_i.\ align(r_i, r_i') \wedge (value(r_i) = value(r_i'))$ **then**
12: call PropAlign(r_i, r_i')
13: **else**
14: add (s_i, s_i') to N_i and **return**
15: **end if**
16: **end for**
17: **end procedure**

Output: Statement instances N_i

Assuming that instances s_i and s_i' can be aligned, the tuple (s_i, s_i') is added to the set N_i if 1) not all of the "subsequent" instances r_i' can be aligned, 2) the values of the variables used in r_i and r_i' are different, or 3) r_i' is a changed statement. This is represented by the intuitively named predicates in line 11.

Note, we do not assume that both DDGs can be aligned completely, which would be rather difficult indeed due to the different number of instances every statement can have in both executions. Instead, the alignment begins from the output statement instances, which we assume to be alignable, and follow the dependence edges recursively. The instance at which alignment fails is added to N_i. In Figure 3.5, the instances in line 4 are added to N_i because they depend on the changed statement in line 3 (which also has different values).

Definition 6 (*Propagation Condition*)

Let statements C in program P be changed to C' yielding P'. Given traces π and π' for the execution of input t on P and P' and Algorithm 3 computes N_i for π and π' and program output statement o, the propagation condition is defined as $prop(o, \pi, \pi', C, C') \stackrel{def}{=} \forall (n_i, n_i') \in N_i . rsc(n_i, \pi) \wedge rsc(n_i', \pi') \wedge value(n_i) = value(n_i') \wedge \bigwedge_{c \in C} reach(c, \pi) \wedge \bigwedge_{c' \in C'} reach(c', \pi')$.

Every input satisfying the same propagation condition does not propagate the effects of the exercised changes for the same reason. To achieve this property, Definition 6 is a conjunction of five necessary conditions. The *necessary conditions 1) and 2)* leverage the property of relevant slices. Note, $rsc(n_i, \pi)$ is the path condition computed over the relevant slice of statement instance n_i in trace π. Every input exercising the same relevant slice w.r.t. n_i, compute the same symbolic value for n_i. The negation of a constituent branch condition may change the computation of n_i and thus enable propagation. The *necessary condition 3)* captures that the symbolic values for the alignable instances in N_i are the same. The negation of such an equivalence condition may enable propagation. The *necessary condition 4) and 5)* captures that those changes (not) exercised by the test case t are also (not) exercised by other input satisfying the same propagation condition. The negation of a constituent branch condition may enable the reachability of other changes.

3.3.4 Computing Difference Conditions

Intuitively, input t computes different output because it exercises a certain set of statement instances in P that contribute to computing the symbolic output of P and another set of statement instances in P' that contribute to computing the symbolic output of P'.

Definition 7 (*Difference Condition*)

Let statements C be changed to C'. Given instances o_i of output statement o in execution trace π and o_i' of o in trace π', the difference condition is defined as $diff(o, \pi, \pi', C, C') \stackrel{def}{=} rsc(o_i, \pi) \wedge rsc(o_i', \pi') \wedge value(o_i) \neq value(o_i') \wedge \bigwedge_{c \in C} reach(c, \pi) \wedge \bigwedge_{c' \in C'} reach(c', \pi')$.

Every input satisfying the same difference condition propagates the semantic effect of the exercised changes for the same reason. To achieve this property, Definition 7 is a conjunction of five necessary conditions. The *necessary conditions 1) and 2)* leverage the property of relevant slices. Every input exercising the same relevant slice w.r.t. o_i, compute the same symbolic value for o_i. The negation of a constituent branch condition may change the computation of o_i and thus disable propagation. Interestingly, *every changed statement in the relevant slice of o_i' contributes in computing o_i' and therefore semantically interferes.* The *necessary condition 3)* captures that the symbolic output values are different in both versions. The negation of this condition may disable propagation. The *necessary conditions 4) and 5)* capture that those changes (not) in π or π' are also (not) exercised by other input satisfying the same difference condition. The negation of a constituent branch condition may enable the reachability of other changes.

A set of changed statements C_t *semantically interferes* for the execution of input t on both program versions, if t yields different output in P and P' and every $c \in C_t$ contributes to computing the output. Thione et al. [42] approximate semantic interference based on static data- and control-dependence. It can be used to understand the origin of regression.

Interestingly, every changed statement in the relevant slice of o_i' contributes in computing o_i' and therefore semantically interferes. This allows the developer to inspect the set of changes responsible for an observed semantic difference.

3.3.5 Generating Adjacent Test Cases

Algorithm 4 generates "adjacent" test cases from the provided symbolic condition and adds those to the *queue*. It implements *generateAdjacentTestcases* called in Algorithm 1.

The symbolic condition is composed of branch conditions $(\psi_0' \wedge \ldots \wedge \psi_m')$ in P', branch conditions $(\psi_0 \wedge \ldots \wedge \psi_n)$ in P, and equivalence conditions υ of the form $value(s_i) = value(s_i')$ or $value(s_i) \neq value(s_i')$ (cf. line 1). First, the constituent equivalence conditions υ_0 to υ_k are negated one-by-one (lines 2-7). If there exists a solution to the computed constraint, it is added to the queue. Second, if some branch conditions are removed from a path condition, the remaining branch conditions have to be *reordered* before negation (lines 8-9). Otherwise, the exploration algorithm ceases to be exhaustive (cf. [19]). Hence, the branch conditions $(\psi_0 \wedge \ldots \wedge \psi_m)$ in P are reordered as follows: If a branch instance b is in the relevant slice of branch instance b_k, then the branch condition of b is placed before the branch condition of b_k. Otherwise, the branch condition

Algorithm 4 - GENERATEADJACENTTESTCASES

Input: Condition *cond*, Queue *queue*

1: let $cond = (\psi'_0 \wedge \ldots \wedge \psi'_m) \wedge (\psi_0 \wedge \ldots \wedge \psi_n) \wedge (\upsilon_0 \wedge \ldots \wedge \upsilon_k)$
2: **for all** υ_i in $[\upsilon_0, .., \upsilon_k]$ **do**
3: $\quad constr \leftarrow (\psi'_0 \wedge \ldots \wedge \psi'_m) \wedge (\psi_0 \wedge \ldots \wedge \psi_n) \wedge \neg\upsilon_i$
4: $\quad$ **if** exists t^+ that satisfies *constr* **then**
5: $\quad\quad$ add t^+ to *queue*
6: $\quad$ **end if**
7: **end for**
8: let $reordered \leftarrow reorder(\psi_0 \wedge \ldots \wedge \psi_n)$
9: let $reordered = (\varphi_0 \wedge \ldots \wedge \varphi_n)$
10: **for all** i from 0 to n **do**
11: $\quad constr \leftarrow (\psi'_0 \wedge \ldots \wedge \psi'_m) \wedge (\varphi_0 \wedge \ldots \wedge \varphi_{i-1} \wedge \neg\varphi_i)$
12: $\quad$ **if** exists t^+ that satisfies *constr* **then**
13: $\quad\quad$ add t^+ to *queue*
14: $\quad$ **end if**
15: **end for**
16: let $reordered' \leftarrow reorder(\psi'_0 \wedge \ldots \wedge \psi'_m)$
17: let $reordered' = (\varphi'_0 \wedge \ldots \wedge \varphi'_m)$
18: **for all** i from 0 to m **do**
19: $\quad constr \leftarrow \varphi'_0 \wedge \ldots \wedge \varphi'_{i-1} \wedge \neg\varphi'_i$
20: $\quad$ **if** exists t^+ that satisfies *constr* **then**
21: $\quad\quad$ add t^+ to *queue*
22: $\quad$ **end if**
23: **end for**

Output: Queue *queue*

of b is placed after the branch condition of b_k. The reordered branch conditions in P are negated one-by-one and conjoined with $(\psi'_0 \wedge \ldots \wedge \psi'_m)$ in P' (line 10-15). If there exists a solution to the computed constraint, it is added to the queue. Lastly, the branch conditions in P' are reordered and negated one-by-one (lines 16-23). Again, if there exists a solution to the computed constraint, it is added to the queue.

3.3.6 Theorems

In the following, we postulate the soundness of Algorithm 2 that computes the differential partition for a given test case and the exhaustiveness of Algorithm 1 that explores differential partitions. The proofs are available in Appendix A.

In practice, the absence of regression errors can be guaranteed for all inputs to the same extent as symbolic execution can guarantee the absence of program errors (see e.g., [67]). Specifically, we assume deterministic program execution.

Theorem 1 (*Sound Generalization*)

Given statements C in program P are changed to C' yielding P', every input satisfying the condition computed by Algorithm 2 for input t is in the same differential partition as t.

Informally, the differential behavior of a point in the common input space is soundly generalized to the set of points in the same differential partition. In particular, let Algorithm 2 compute the symbolic condition Φ for a test case t. If t is equivalence-revealing, then every input satisfying Φ is equivalence-revealing. Similarly, if t is difference-revealing, then every input satisfying Φ is.

Theorem 2 (*Exhaustive Exploration*)

If there exists an input t_0 that computes different values for the output o in versions P and P' and Algorithm 1 terminates with regression test suite T, then there exists a test case $t \in T$ so that t_0 satisfies diff$(o, \pi(t, P), \pi(t, P'), C, C')$.

Informally, if the verification procedure terminates then all differential partitions have been explored. The respective proof leverages the exhaustiveness of the exploration based on relevant slices as shown in [19].

3.4 Empirical Study

Our experiments evaluate the relative efficiency of PRV and discuss practicability based on our experience. The experiments do not prove the scalability of PRV. In fact, PRV suffers from the same limitations as symbolic execution. Similarly, it can benefit from relevant optimizations such as domain reduction [30, 28], parallelization [31], and better search strategies [117, 104].

3.4.1 Setup and Infrastructure

PRV has been implemented into our dynamic backward slicing tool JSlice [118]. The differential partitions are explored in a breadth-first manner starting from the same initial input within the time bound of five minutes, unless stated otherwise. Every version of the same subject uses the same test driver to construct necessary input objects, strings, or arrays from the input integers that come as solution to a first-order logic formula from the Z3-constraint solver [119]. The subject programs are analyzed on a desktop computer with an Intel 3GHz quad-core processor and 4GB of memory.

3.4.2 Subject Programs

The subjects summarized in Figure 3.6 are chosen according to two criteria: 1) they represent a variety of evolving programs and 2) are discussed in related work (which allows the comparison with our own experimental results). There are 83 versions of programs ranging from 20 to almost 5000 lines of code (LoC). Some versions are derived by seeding faults, called mutants, of the original versions. Some are real versions that were committed to a version control system.

Subject	Reference	Classes	Functions	LoC	Versions
Min	[120]	1	1	20	5
Tcas	[114] [2] [26] [121]	1	8	166	21
Replace	[26] [100]	1	21	564	33
Siena	[100]	6	107	1529	7+11
Apache CLI		22	183	4966	6
	Total	*30*	*320*	*7245*	*83*

Figure 3.6: Subject Programs

We compare the empirical results of the references discussing regression verification [114] and regression test generation [26, 121, 100, 2]. Note, there are no empirical results available for the regression test generation techniques [57, 25, 1] and differential symbolic execution [39].

Min [120] is a short function introduced to discuss the problem of equivalent mutants. An equivalent mutant is a simple syntactic change to a program that yields no semantic difference. *Tcas* is the traffic collision avoidance system. This well-studied program is available in the SIR [122] with several versions that contain seeded faults. We chose the first 20 changed versions. *Replace* performs pattern matching and substitution and is available in the SIR with 32 versions that contain seeded faults. *Siena* is an event notification architecture. Note, there are 7 versions available in the SIR and for every version there exist between one and four faulty versions (in total 11 mutants).

Revision	Submission	Developer's Submission Comment
129800	15.08.2002	bug. no 11680 resolved
129803	18.08.2002	bug #11457: implemented fix [..]
129843	14.11.2002	added fix for Rob's problem [..]
129849	19.11.2002	some bug fixes submitted by Rob [..]
538031	15.05.2007	Applying Brian Egge's fix from CLI-13
667565	13.06.2008	Restored CLI 1.0 behavior (CLI-137)

Figure 3.7: Apache CLI Revisions (http://commons.apache.org/cli/)

Apache CLI is an open source command line interpreter. We retrieved the six revisions from the version control system (branches/cli-1.x/src) that are presented in Figure 3.7 along with the submission date and comments and the unique identifiers.

All programs are tested as whole programs, except for Apache CLI. In this case, the the command line component was tested for regression. The first three

programs all have a main method. For Siena, `encode` and `decode` in the class `SENP` serve as main methods. For Apache CLI, `addOption` and `getOptionValue` in the class `CommandLine` serve as testing hooks.

3.4.3 Research Questions

RQ1: How efficiently does PRV find the first input that exposes ***semantic difference****?*
Empirical studies in the discussed related work are concerned with finding the first difference-revealing input as witness of semantic difference. We compare the efficiency of PRV to the efficiency reported in related work.

RQ2: How efficiently does PRV find the first input that exposes ***software regression****?*
Not every difference-revealing input exposes software regression. In fact, after syntactic changes to the program, semantic changes may be anticipated in the form of progression. For instance, when a buggy program is fixed input failing in the buggy version is supposed to pass in the fixed version. To classify a semantic change as regression, we have to define correctness. As often in reality, we assume the *absence* of formal specifications. In this scenario, the developer checks the generated difference-revealing test cases informally against her expectation. If she observes regression, the developer can relate the regression-revealing test cases to the changes that semantically interfere.

RQ3: How practical is PRV in an example usage scenario?
The subject Apache CLI shall be used to evaluate PRV in a practical usage scenario. PRV generates difference-revealing test cases within the bound of 20 minutes for every version pair. We classify the generated test cases (e.g. regression-revealing) and compare the (informal) measure of regression and progression to the submission comments in Figure 3.7.

3.5 Results and Analysis

RQ1: Efficiency - Semantic Difference

We measure two aspects when searching for the first difference-revealing input as shown in Figure 3.8. The first seven rows show the *average time* to find a difference-revealing input per subject. If for a version pair none of the approaches finds a difference-revealing test case within five minutes, then it does not contribute to the calculation of the average time. The *mutation score* depicts the fraction of versions for which a difference-revealing input can be found within five minutes. To gather results for the symbolic execution of the

changed version P', we implemented a DART-like [21] and eXpress-like [100] path-exploration technique (Columns 3-4) into JSlice. The DART-like technique explores all paths in P' while the eXpress-like technique prunes all paths that do not exercise a changed statement in P'. The results for the exploration of the differential behavior of both versions, P and P', are gathered using PRV (Column 2).

	P, P'	only P'	
	PRV	**DART**-like	**eXpress**-like
Average Time *in sec*			
Min (4 Mutants)	0.4	0.3 (-25%)	0.3 (-25%)
Tcas (20 Mutants)	5.6	20.9 (+273%)	20.7 (+270%)
Replace (32 Mutants)	22.8	130.5 (+472%)	60.1 (+164%)
Siena (11 Mutants)	30.7	66.2 (+116%)	40.4 (+32%)
Siena (7 Versions)	14.0	18.3 (+31%)	12.7 (-9%)
Apache CLI (6 Versions)	57.8	38.9 (-33%)	45.1 (-22%)
Mutation Score *- fraction of versions shown semantically different*			
Min (4 Mutants)	0.75	0.50 (-33%)	0.50 (-33%)
Tcas (20 Mutants)	1.00	0.56 (-44%)	0.56 (-44%)
Replace (32 Mutants)	0.76	0.56 (-26%)	0.63 (-17%)
Siena (11 Mutants)	0.82	0.73 (-11%)	0.73 (-11%)
Siena (7 Versions)	0.67	0.67 (±0%)	0.67 (±0%)
Apache CLI (6 Versions)	1.00	1.00 (±0%)	1.00 (±0%)

Figure 3.8: First Witness of Semantic Difference

Answer to RQ1. For the analyzed subjects, *PRV generates a difference-revealing test case on average for 21% more version pairs in 41% less time*, than the eXpress-like approach that analyzes only the changed version P'. For the subtle, seeded faults PRV can find a difference-revealing test case more efficiently. In particular, Tcas is fully analyzed within the time bound by all approaches but only PRV can find a difference-revealing test case for every mutant supporting the motivation illustrated in Figure 3.2. In general, PRV's relative efficiency is better for the first four subjects containing subtle, seeded faults. This efficiency reduces as the changes become more complex in the latter two subjects. This can be attributed to the increased number of changed statements correlating with an increased probability to reveal a difference (for random input). However, not every difference-revealing test case is also regression-revealing as analyzed in RQ2.

Compared to DART, our eXpress-like implementation has a similar relative efficiency than eXpress in [121] and [100]. The authors compare full path exploration (Pex) to pruning paths that do not execute a changed statement

(Pex-eXpress). For Siena, Replace, and the chosen mutants of Tcas, the authors report an improvement in terms of time of 29%, 57%, and 13%, respectively. For these subjects, we see a similar improvement of 37%, 54%, and 16% of the eXpress-like approach over the DART-like approach, respectively.

Mutation Score	Matrix [2]	SHOM [26]	PRV [this]
Tcas	62.7%	62%	100%
Replace	-	72%	76%

Figure 3.9: PRV mutation scores vs SHOM and Matrix

Santelices et al. [2] and Harman et al. [26] report the mutation score for the test generation tools Matrix and SHOM, respectively. Note that many of the subjects used by these authors and us are different. However, Tcas was used to evaluate Matrix and SHOM, while Replace was also used to evaluate SHOM. As shown in Figure 3.9, PRV compares favourably for the commonly evaluated subjects, Tcas and Replace. In contrast to these search-based techniques, PRV avoids searching for difference-revealing test cases within the already explored input space. This may help explaining the observed improvements.

Godlin et al. [114] evaluate the implementation of regression verification using randomly generated programs and Tcas. It takes many hours or the system runs out of memory when analyzing non-equivalent programs. Offutt et al. [120] discuss the problem of equivalent mutants using subject `Min`. PRV guarantees equivalence for Mutant 3 and provides a witness for the other non-equivalent mutants in less than a second.

RQ2: Efficiency - Software Regression

In practice, not every difference-revealing test case reveals software regression. A difference-revealing test case can be checked formally or informally against the programmer's expectation. In the latter case the programmer looks at the output of difference-revealing test cases in both programs and may know whether the test case reveals regression. Figure 3.11 presents the two aspects measured to find the first regression-revealing input. The first seven rows show the *average time* to find a regression-revealing input per subject. If for a version pair none of the approaches finds a regression-revealing test case within the time bound, then it does not contribute to the calculation of the average time. The *mutation score* depicts the fraction of versions for which a regression-revealing input can be found within 20 minutes for Apache CLI and five minutes for the other subjects.

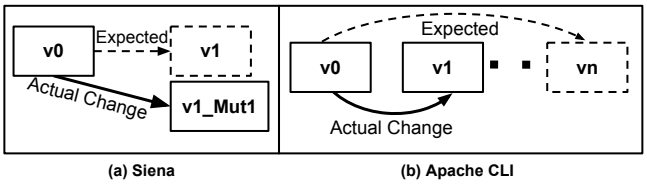


Figure 3.10: How to Measure Regression?

How do we measure regression? For Siena, we simulate an incomplete bug fix [7] from one version to the next. An example is shown in Figure 3.10.a). The programmer fixes Siena.v0 which is expected to behave like Siena.v1. Instead, he introduces another bug yielding Siena.v1_Mut1 – a version of Siena.v1 that also contains seeded faults. For Apache CLI, there are no seeded faults available. But we can capture the programmer's idea of the expected behavior to be in the last revision 667565 which remains unchanged for the last four years. This allows us to measure the regression of intermediate revisions w.r.t. the last revision. For Min, Tcas, and Replace every difference-revealing test case also reveals a regression.

	P, P'	only P'	
	PRV	**DART**-like	**eXpress**-like
Average Time *in sec*			
Min (4 Mutants)	0.4	0.3 (-25%)	0.4 (-25%)
Tcas (20 Mutants)	5.6	20.9 (+273%)	20.7 (+270%)
Replace (32 Mutants)	22.8	130.5 (+472%)	60.1 (+164%)
Siena (11 Faulty Versions)	17.6	50.4 (+186%)	44.1 (+151%)
Apache CLI (6 Versions)	141.3	259.6 (+84%)	263.9 (+87%)
Mutation Score *- fraction of versions exposed as regression*			
Min (4 Mutants)	0.75	0.50 (-33%)	0.50 (-33%)
Tcas (20 Mutants)	1.00	0.56 (-44%)	0.56 (-44%)
Replace (32 Mutants)	0.76	0.56 (-26%)	0.63 (-17%)
Siena (11 Faulty Versions)	0.55	0.45 (-17%)	0.45 (-17%)
Apache CLI (6 Versions)	0.40	0.20 (-50%)	0.20 (-50%)

Figure 3.11: First Witness of Software Regression

Answer to RQ2: For the analyzed subjects, *PRV generates a regression-revealing test case on average for 48% more version pairs in 63% less time* than the eXpress-like approach that analyzes only the changed version P'. The improvement of efficiency over finding a single difference-revealing input (cf. Figure 3.8) may be attributed to the subtleness of regression faults. As an instance of this subtleness, consider the program versions in Figure 3.2. The programmer expects that the new version computes output $o = i + 1$ instead of $o = i$ for input $i > 0$. Otherwise, the behavior shall remain unchanged. Thus, a

regression test generation tool may determine progression for almost 50% of the input ($i > 0$) but because a branch is evaluated in different directions for $i = 0$, there exists regression only for one input. Unintendedly, this input computes different output in the changed version, too. Even the generation of an input for every path in the changed program, like for eXpress, may not produce this test case. In contrast, differential partitions can capture such subtle differences.

RQ3: Practicability - Usage Scenario: Apache CLI

Apache CLI is used to evaluate PRV in a practical usage scenario. PRV generates difference-revealing test cases within the bound of 20 minutes for every version pair. A developer checks these test cases for regression and relates the regression-revealing test cases to the changes that semantically interfere. The check is automated in our experiment as illustrated in Figure 3.10.b). The expected behavior of CLI is captured by the last revision (667565) which has not changed in the last four years and is released in CLI1.1. This allows us to measure progression and regression w.r.t. to the expected behavior.

The first column in Figure 3.12 shows the revision pairs, the earlier versus the later revision. The second column presents the total number of tests generated by PRV followed by the number of equivalence- and difference-revealing test cases, respectively. The percentage of difference-revealing test cases (Column 4) witnessing progression (**%Progr**), regression (**%Regr**), and the computation of output that has changed but still does not behave as expected (**%Chan**) are shown in columns 5, 6, and 7, respectively.

Subject and Versions		**#Test**	**#Equ**	**Difference Revealing**			
				#Diff	**%Progr**	**%Regr**	**%Chan**
CLI (20min)	r129800-r129803	788	748	40	0%	0%	100%
	r129803-r129843	835	809	26	65%	0%	35%
	r129843-r129849	721	639	82	82%	1%	17%
	r129849-r538031	509	485	24	0%	***88%***	13%
	r538031-r667565	536	455	81	100%	0%	0%
	Average				**49%**	**18%**	**33%**

Figure 3.12: Exploration of differential behavior in limited time

Answer to RQ3: For the evolution of Apache CLI over six years, tests generated as witnesses of differential behavior of two successive versions suggest an *average progression of 49%, regression of 18% and intermediate semantic changes of 33%* towards the latest revision. The interested reader may compare the results in Figure 3.12 to the developer's notes in Figure 3.7. The behavior of CLI generally experiences progression from version r129800 to r129849 when suddenly the behavior regresses with the change to r538031. In fact, while

trying to fix bug CLI-13[4], the developer introduces bug CLI-137[5]. This is a clear regression bug which is witnessed by 88% of the difference-revealing test cases generated by PRV. However, it takes two months to report and twelve to fix bug CLI-137 and commit it as revision r667565. In contrast, PRV generates the first regression-revealing test case for r538031 in 88 seconds among the first five generated difference-revealing test cases.

3.6 Threats to Validity

The main threat to *internal validity* is the correctness of our implementation of PRV into JSlice. We tried to mitigate this threat by using the same implementation to gather results for the DART-like and eXpress-like approaches. In practice, any implementation of PRV can guarantee the absence of regression errors to the same extent as symbolic execution can guarantee the absence of program errors (see e.g., [67]). Our particular implementation could be faulty, so that it may not report a witness of behavioral difference if one exists. On the other hand, a reported witness of behavioral difference is indeed a witness of behavioral difference. This is inherent to the approach, as the generated test cases are concretely (and symbolically) executed on both programs.

The main threat to *external validity* is the generalization of our results. The limited choice and number of subjects does not suggest generalizability and serve mainly as comparison to related work and give an idea about the practicability of PRV.

3.7 Related Work

Regression Verification (RV) is the problem of deciding whether a changed program is *at least as correct* as a previous version. One line of work takes an earlier version as a program specification of the new version [114, 115, 123]. The authors argue if both versions are semantically equivalent, then there is no software regression. Yet, not every difference is a regression. For instance, a bug-fix yields anticipated behavioral difference. Another line of work requires an explicit specification, and builds on the full verification of an earlier version. Subsequently, only the changed behavior of the following versions need to be checked incrementally [124, 125]. In general, RV can take a long time to terminate. When the search is interrupted, no intermediate guarantees can be reported. In contrast, the interruption of PRV can guarantee the absence

[4] https://issues.apache.org/jira/browse/CLI-13
[5] https://issues.apache.org/jira/browse/CLI-137

of regression at least for the explored equivalence-revealing partitions. Moreover, the difference-revealing test cases can be "informally" checked for further regression by developers.

Conditional regression verification is "partitioning" the input space into two subdomains – verified (equivalence-revealing) and unverified – based on properties that are external to the two program versions, such as a pre-defined bound on the memory, time, or input. The verified subdomain can be bounded *a priori*, e.g., by limiting the number of loop iterations [114], or by pre-defining an input range [126]. The verified subdomain can also be bounded during verification *upon exhausting available resources*, e.g., by setting a time or memory bound [127]).

In contrast, Partition-based Regression Verification (PRV) is partitioning the input space into many disjoint subdomains that are homogeneously equivalence- or difference-revealing and dynamically discovered as inherent property of the two programs versions. A differential partition is defined by the group of paths in both versions that together *explain* the observed output equivalence or difference. The gradual verification of the complete input space is rather an enumeration every differential partition which may serve as units of verification. It remains orthogonal that the verified subdomains can also be bounded a priori or upon exhausting available resources.

Differential Symbolic Execution (DSE) [39] is a general approach to compute program differences while PRV is a specialized approach tailored to RV. Specifically, DSE computes the differences based on two types of program summaries. The *symbolic summaries* in Figure 3.2 on the right (P and P') precisely characterize the behavior of the program versions on the left. The *abstract summaries* in Figure 3.13 over-approximate the behavior for the same versions. Exploiting the syntactic similarity of both versions, the behavior of common code blocks can be represented by uninterpreted functions.

	Input	**Output**	Regression Test Case t
$P\ (= \Delta_{\langle P,P' \rangle})$	*true*	$o = o(0, i)$	Value for i satisfying
$P'\ (= \Delta_{\langle P',P \rangle})$	*true*	$o' = o(0, i+1)$	$o(0, i) \neq o(0, i+1)$

Figure 3.13: Program Deltas (Δ) and Abstract Summaries (cp. Fig.3.2)

RV based on program summaries is either less scalable or infeasible. While symbolic summaries may be used for RV, the differences are computed as an expensive cross-product of (incomplete) summaries. On the other hand, PRV is based on differential partitions that account for the common input space of both versions. Furthermore, PRV yields coarser partitions. For instance, if input does not reach a change already implies both programs compute the same

output. Abstract summaries, on the other hand, remove information required for RV. The interested reader may verify in Figure 3.13 that, if the delta contains uninterpreted functions, a concrete difference-revealing test case cannot be generated. In this example, each delta accounts for a single partition (*true*), while PRV distinguishes two difference-revealing and one equivalence-revealing partition. For each partition, PRV generates a witnessing test case.

Regression Test Generation (RTG) is the problem of constructing sample input that can expose software regression. Classically, test cases are generated towards the coverage of the program's behavior [21, 80]. The hope, when the program is changed and behavior regresses, is that at least one test case fails in the new version. Further, when the program is changed, test cases are generated towards the coverage of program elements that are affected by the syntactic changes [95, 100, 25]. These test cases *augment* an existing test suite that was coverage-adequate for the earlier version. However, the analysis of a single program may be insufficient to generate a regression-revealing test case (cf. Fig. 3.2). Instead, some research directly aims at generating difference-revealing test cases. Syntactic approaches seek to *reach* a change, *infect* the program state, and *propagate* it to the output [2, 1, 26, 58]. However, the number of possibly semantically interfering sets of changes is exponential to the number of overall syntactic changes. Harman et al. [26] note the testing of every subset would be prohibitively expensive. Even for a single subset, the search for a difference-revealing input may not terminate. In contrast, the number of changes is unimportant to PRV, a priori. It groups input, depending on whether it reaches and propagates the same set of changes, on demand during exploration. More importantly, PRV can guarantee the absence of regression not only for a singular point in the common input space, but for an entire partition.

3.8 Chapter Summary

In this chapter, we have proposed the systematic exploration of the semantic changes resulting from the many (potentially interacting) syntactic changes between two program versions. We have formalized the intuition of semantic changes using the notion of differential partitions: Either every input in the same partition computes exactly the same output. Or every input in the same partition computes a certain symbolic output formula in one version but another symbolic output formula in the subsequent version.

Each generated regression test case becomes significant and representative of a larger set of inputs. Once a test case is executed on both versions and exposes a difference, PRV can *soundly* generalize to those inputs that are also difference-revealing. The corresponding test cases and symbolic differences can

be (in-)formally verified by the developer. Conveniently, the absence of regression is already shown for equivalence-revealing partitions and require no further verification. PRV inspires confidence in the absence of regression at least for input in the covered input space and every generated test case directly increases this confidence.

Differential partitions enable a gradual and partial form of regression verification. Differential partitions exist as a unit of verification in the common input space of two program versions and are checked one after another. When the verification process is interrupted, PRV retains regression guarantees for the explored input space. This is crucial as verifying the complete input space is prohibitively expensive.

Experiments have shown that PRV exposes regression errors that are not detected by other regression test generation methods. The proofs of the theorems corresponding to our claims are available in the Appendix A.

Chapter 4

Test Generation to Expose Change Interaction Errors

„Das Ganze ist etwas anderes als die Summe seiner Teile."

— Wolfgang Metzger, 1899 – 1997

Complex changes often introduce program errors, and hence recent software testing literature has focused on generating tests which stress changes. In this chapter, we argue that *the simple changes constituting a complex change cannot be stressed as isolated program artifacts*. Instead, it is the complex dependency across these changes which introduce subtle errors and ensures that such errors remain undiscovered even in well tested and deployed software. We motivate our work based on empirical evidence from a well tested and stable project - Linux GNU Coreutils - where we found that one third of the regressions take more than two (2) years to be fixed, and that two thirds of such long-standing regressions are introduced due to change interactions for the utilities we investigated.

To combat *change interaction errors* (CIE), we first define a notion of change interaction where several changes are found to affect the result of a statement via program dependencies. Based on this notion, we propose a *change sequence graph* (CSG) that captures the control-flow among the changed statements and their interaction locations. The CSG is then used as a guide during directed path exploration via *symbolic execution* – thereby efficiently producing test inputs that witness CIEs. Our experimental infrastructure was deployed on various utilities of GNU Coreutils, which have been distributed with Linux for almost twenty years. Apart from finding five (5) previously unknown errors in the utilities, we found that only one in five generated test cases exercises a sequence that is critical to exposing a CIE, while being an order of magnitude more likely to expose an error. On the other hand, stressing changes in isolation only exposed half of the CIEs. These results demonstrate the importance and difficulty of change dependence aware regression testing.

4.1 Introduction

Changes even to well-tested software projects can introduce subtle bugs of varying severity that may be exposed only years later. Such change-based errors in deployed software come in two forms. First of all, bug fixes may introduce new bugs. For instance, Gu et al. [6] mentions that feature additions or bug fixes, introduce new bugs in 9% of cases. Secondly, a subtle or poorly understood "interaction" among various changes may introduce hard-to-find errors in well-tested code, which then get deployed. In this chapter, we focus on test generation to expose such subtle *change interaction errors* (CIEs).

Evidence of subtle change interaction errors can be found in many well-tested and deployed software projects. In our study on `GNU Coreutils`, we found that *every fifth bug fix actually patches regressions* introduced in an earlier commit. About one third of these regressions take more than two (2) years to find and fix, despite the tool set being rather well tested. Note that ∼21% of the total commits update the comprehensive test suite, while only 30% actually update the utilities (the remaining 49% are related to maintenance, like documentation, the build process, or ambiguous error messages). Thus, it is surprising that on utilities with such well-updated test-suites, errors due to change interaction will remain for two years. In fact, the `GNU Coreutils` have been dispatched with almost every Linux distribution for the last 20 years!! This led us to think that change interaction errors, which stress subtle dependencies across changes, may be hard-to-find due to most regression testing methods being focused on some form of *coverage*.

At this point, we step back and review the recent regression testing research which focus on program changes. A recent work [54] presents criteria and experiments for the interaction among program changes but does not suggest any method for integrating them into regression testing. Among the works achieving change aware test generation, some study only independent program changes [112, 1]. Several of the testing methods attempt to achieve either a structural coverage of changed statements or some other structural coverage (such as branch outcome coverage) in the modified program (e.g., see [95]). Since coverage based methods may not stress the semantic effect of the changes, attempts have been made to take a powerful symbolic execution based path exploration engine, and adapt it to the presence of program changes. Since symbolic execution captures the semantic effect of program changes, the hope is that the semantic effect of a change can be propagated through such methods. On the other hand, since a full-fledged symbolic execution based path exploration can be exceedingly slow, these methods employ various pruning strategies to cull away program paths which cannot reach or propagate the changes (e.g.,

see [100]). Other authors suggest to statically compute the program slices for every change and dynamically employ symbolic execution upon these slices to exercise all paths that are affected by a changed statement (e.g., see [25]). However, in all of these works, the set of changes in a program is treated in an aggregate fashion. The flows/dependencies across changes are not systematically explored/exploited for generating test cases.

In this chapter, we present a test generation method to systematically explore and expose subtle errors arising due to the "interaction" among program changes. Since any such change interaction leading to errors is inherently dynamic, we first statically approximate the relationships among the changes. Our approximation is called *change sequence graph* (CSG) which captures (i) the control-flow across the changed statements and (ii) the control-flow to control locations at which multiple changes may interact, leading to unexpected semantic effects. These interaction locations are computed based on the program dependencies across multiple changed statements. The CSG is then used as a definitive guide to find out the sequence of control locations that need to be visited for exposing potential change interaction errors. These control locations are visited systematically by programming a graph-based search strategy on top of the directed symbolic execution engine, Otter [128].

Experimental results from our approach on `GNU Coreutils` show the prevalence of change interaction errors among regression bugs. We note that the `GNU Coreutils` tool-set is a collection of Linux utilities which have been widely tested. In particular, every fifth commit to the repository updates a comprehensive test suite that exists for more than twenty years, and the tool set was further tested by the authors of `klee` [129] and `test-zesti` [130] (reporting 3 and 2 errors, respectively). Despite such extensive testing, we found and reported five verified, previously unknown regression errors, apart from many known errors. Among other notable findings, we noticed that two in three differential errors can be classified as change interaction errors. We also found that only half of the CIEs were exposed by a testing algorithm that target changes in isolation, but does not account for their interaction. This clearly demonstrates the importance of change-interaction aware regression testing.

In summary, the **contributions** of this chapter are:

- **Change-Interaction Errors:** We identify and formalize change-interaction errors: errors that happen in evolving software, which arise due to the combined semantic impact of multiple changes. We argue for the importance of this class of errors with a study of regression on GNU coreutils over a period of 5 years.

- **Detection Method for CIEs based on CSGs:** We propose a datastructure called a change-sequence graph to capture potential sequences of changed statements and interaction locations in an execution of a program. Using CSGs we show a detection method which stresses sequences in the graph to expose CIEs.
- **Implementation and Empirical Evaluation:** Our CIEs detection method has been implemented and an empirical evaluation using that implementation was conducted to evaluate its effectiveness.

4.2 Regression in `GNU Coreutils`

To study software regression, we looked at the repository of `GNU Coreutils`, which has been actively developed and maintained for more than twenty years. Our results show that within the last five years every fifth bug fix actually patches regressions introduced earlier and that 30% of such regressions take more than 2 years to be fixed. These results are corroborated by the package maintainer.

4.2.1 Statistics of Regression

It is possible to access the history of every change commited to the source code repository of `GNU Coreutils` since Oct'92[1]. Usually, these commits are accompanied by a commit message that describes the relevance and intention of the change. The commits to the repository of `GNU Coreutils` are categorized as changes to particular tools, or as *build*, *tests*, *maint*[enance], amongst others. The developers adopted this commit message labeling about five years ago. This allows us to distinguish code-changing commits[2] from maintenance commits. Parsing the commit messages for keywords, such as "bug", "fix", or "regression", we were able to find how many of the code-changing commits are bug fixes and feature additions. If the commit message contained "introduced" or "regression", we could derive whether a bug fix was actually patching regressions introduced earlier. Often, a *regression-fixing* commit would reference the *regression-introducing* commit. Thus, we can measure the time in-between. As the commits have been nicely categorized in the last five years, we looked at those between Jan'08 and Feb'13. However, *regression-fixing* commits can reference *regression-introducing* commits that were submitted much earlier. Given the X- and the (logarithmic) Y-axis in Figure 4.1, the graph shows that X percent of the *regression-introducing* commits 1) require *more than Y days* to be

[1] http://git.savannah.gnu.org/cgit/coreutils.git
[2] Code-changing commits are labeled by the changed tool.

found and fixed (solid line), and 2) contain *more than Y Changed Lines of Code* (CLoC; dashed line).[3]

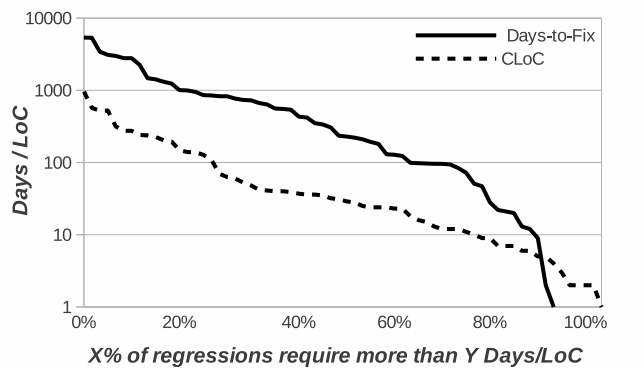


Figure 4.1: Regression Statistics - `GNU Coreutils`

Results. About 30% of the 2.6k commits in the recent 5 years are code changes - feature additions and bug fixes in roughly equal shares. Interestingly, *every fifth bug fix actually patches regressions* introduced in an earlier commit. The following observations are are further corroborated by package co-maintainer, Pádraig Brady, via email-exchange:

O.i 30% of the regressions introduced in earlier commits take more than 2 years to find and fix despite a comprehensive and well-maintained test suite (∼21% of the total commits update the test suite)

O.ii 45% of the regressions are introduced when more than 35 LoC are changed (while only about 25% of the code-changing commits modify 35 LoC or more).

This led us to suspect that the changed behaviour introduced by the syntactic changes to the tools is not properly tested. In particular, we consider the subtle interplay of many code changes as reasons for regressions to be exposed so late. We call this type of errors — *change-interaction errors*. Indeed, as discussed in Section 4.7, we find that 66% of the errors introduced in earlier commits can only be exposed by input exercising certain *critical sequences* of changed statements. It turns out that only one in five tests inputs exercise a critical sequence, while such test cases are 15 times more likely to expose an error. In the remainder of this section, we have a closer look at one of the regression errors.

[3]Days and CLoC computed using the `git stat`-tool.

4.2.2 Buffer Overflow in `cut`

During our investigations, we found and reported a buffer overflow in the tool `cut` of `GNU Coreutils` which was introduced as a regression error in commit `ec48bead` and manifests as SEG_FAULT for the failing test input.[4] Buffer overflows can be exploited maliciously to gain root access to affected computers [131]. This issue is particularly critical for systems that are dispatched with almost every Linux distribution, such as `GNU Coreutils`, which contains well-known command-line tools, such as `cp`, `mv`, `rm`, `echo`, and `cut`. Fortunately, in the five years preceeding this dissertation the package maintainer of `GNU Coreutils` had to fix only 10 SEG_FAULTs.[5] However, a surprising 6 out of 10 are regression errors *introduced in earlier commits*.

The Anatomy of a Regression

In simple terms, the tool `cut` takes a set of number ranges, a file, and an optional output-delimiter as input and prints the content of every line in the specified file within the specified ranges, optionally separated by the specified output-delimiter. For instance, the command in Figure 4.2, uses "hello world" as input to the `cut` utility - which prints the range between the 2nd and 3rd character, and from the 7th character onwards, both ranges separated by "`,`" (comma).

```
$ echo "hello world" | cut -output-del=, -b2-3,7-
el,world
```

Figure 4.2: Linux Terminal - the output of `cut`

Problem. *If there are no finite ranges (e.g., `7-`), then too much memory is unnecessarily allocated.*
Specifically, if `max_range_endpoint` is set in line 504 of Figure 4.3 or earlier, then the array `printable_field` is allocated `max_range_endpoint` of memory (line 509). If `output_delimiter_specified`, then `printable_field` is unnecessarily (but successfully) accessed at `eol_range_start` in line 266. Note, if `eol_range_start>max_range_endpoint`, then `max_range_endpoint` is set to `eol_range_start` in line 504.

Intended Change. *Allocate memory only if necessary.*
Specifically, only if `max_range_endpoint` is set, allocate the array `printable_field` with `max_range_endpoint` of memory. Only if `output_delimiter_specified` and `max_range_endpoint` is set, then the array `printable_field` shall be accessible in line 534.

[4] Report and fix avail. at http://debbugs.gnu.org/13627.
[5] We analysed commit messages in the source repository. The actual number may be greater.

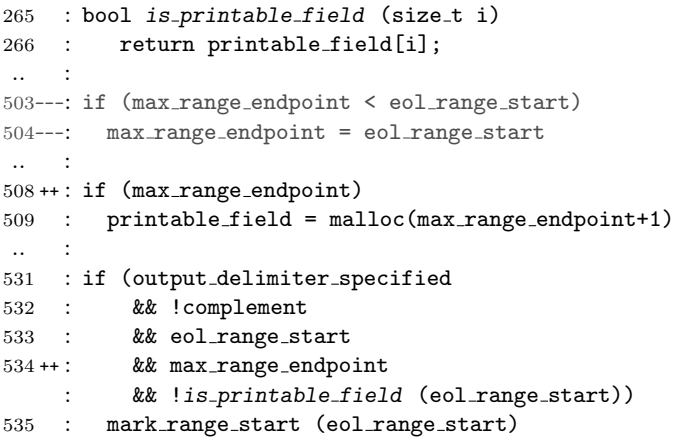


Figure 4.3: SEG_FAULT introduced in `cut`

Actual Changes. *The developer applies three code changes. Every change is essential to fix the memory leak.*
Specifically, the developer *C.1)* adds that `printable_field` is allocated only if `max_range_endpoint` is set (line 508), *C.2)* adds that `printable_field` is accessed only if `max_range_endpoint` is set (line 534), and *C.3)* removes that `max_range_endpoint` is set to `eol_range_start` if `eol_range_start > max_range_endpoint` (lines 503-504). Note, all changes are essential to fix the memory leak. For instance, without change *C.3*, the variable guarding the memory allocation is always set, rendering the additional checks of changes *C.1* and *C.2* redundant.

Regression Error. *If finite ranges are specified, then unallocated memory can be accessed, yielding a SEG_FAULT.* Specifically, if 1) `max_range_endpoint` is set, 2) `max_range_endpoint < eol_range_start`, and 3) `output_delimiter_specified` is set, then the array `printable_field` is accessed out-of-bounds at `eol_range_start` in line 266.

Combined Semantic Impact of Changes

The observation of the regression error depends on the execution of both changes, *C.1* and *C.2*. They have a combined "semantic impact" on the same program location - the memory access. Specifically, the allocation of memory for `printable_field` in line 509 depends on the code added with change *C.1*. The access of memory in `printable_field` in line 266 depends on the code added with change *C.2*. Because the *success* of accessing an array also depends on the memory allocation for this array, both changes have a combined impact at the memory access location. So, the memory access at line 266 is called *interaction location* of *C.1* and *C.2*. The sequences in which the changes can be executed

are depicted in Figure 4.4.[6] Note, *C.3* is not part of the presented graphs since a deletion does not manifest in the changed version P'.

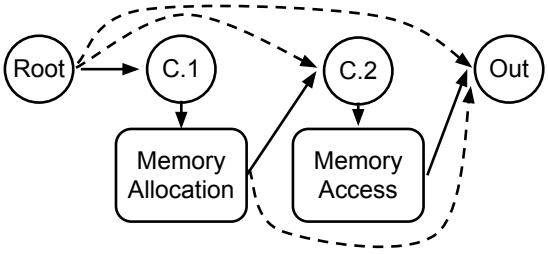


Figure 4.4: Input can exercise these change sequences.

It is insufficient to test both changes in isolation. The regression error is only observable for (some) input that exercises both changes - the sequence following the solid lines in Figure 4.4. The buffer overflow is not observable for input exercising only change *C.1* but not *C.2* and neither for input exercising only change *C.2* but not *C.1* - sequences exercising one of the dashed lines in Figure 4.4. Hence, we call this regression a *change interaction error*.

4.3 Errors in Software Evolution

This section formally describes classes of errors that can occur during software evolution. In particular, we are interested in a class of errors, arising from the interaction of multiple changes, that we call *change interaction errors* (CIE). To establish the context of CIEs, we also define a useful generalization of regression errors, which we call *differential errors*.

4.3.1 Preliminaries

For the definitions in this section, we will assume two successive versions of a program, P and P', and an oracle S. The oracle S specifies the intended behavior for P and P'. As such, it is expected that for all input i executed on P', the output is observationally equivalent[7] to executing i on the oracle S. The explicit oracle can be a specification, the ultimate final version of a program, a validating test suite, or some other artifact that could be used to validate expected behaviour.

[6] For brevity, we removed sequences that contain a change but no memory allocation or access.

[7] Two programs P and P' are observationally equivalent for an input i, $P(i) \equiv P'(i)$, if the *relevant* program output produced by executing i on P and P' is the same.

Using the earlier version P, the changed version P' and intended behavior S of P and P', we can more formally define *regression error*.

Definition 8 (*Regression Error*)

An error is a regression error if for some input i holds:
$P(i) \not\equiv P'(i)$ *and* $P(i) \equiv S(i)$.

In other words, a regression error happens when for some input i the earlier version, P, works as expected but the new version, P', does not work anymore. Note that this definition *does not* prevent P' from exposing the correct behaviour for some other input, which fails in P w.r.t. S. Therefore, our definition of regression error captures the common situation in which the initial version P may have some errors that are intended to be fixed in P', but while P' is fixed for some inputs, it starts behaving incorrectly for some other inputs. An intended software quality improvement turns into a possible deterioration of the software quality.

4.3.2 Differential Errors

In the context of software evolution we often find the need for a notion more general than that of a *regression error*. We call this notion *differential error*.

Definition 9 (*Differential Error*)

An error is a differential error if for some input i holds:
$P(i) \not\equiv P'(i)$ *and* $P'(i) \not\equiv S(i)$.

In other words, a differential error happens when, for some input i, the changed version P' works differently from both, the earlier version P and the intended behavior S. There are two interesting situations. The situation in which the earlier version $P(i)$ worked as expected ($P(i) \equiv S(i)$) is just equivalent to the definition of regression error. On the other hand, the situation in which the earlier version $P(i)$ did not work as expected either ($P(i) \not\equiv S(i)$) cannot be called regression error. So we call it differential error. This captures a situation, e.g., of an incomplete fix. The developer intends to fix the behaviour of P, so that test cases i and j fail on P w.r.t. S. But while i may now pass in the fixed version P' and j produces different output, j may still fail on P' w.r.t. S – the fix was incomplete. In practice, it is helpful to characterize situations in which several intermediate "fixes" are implemented until an ultimate version meets the expectations.

P

```
1
2  while(true){
3   if(isDigit(*fs)){
4    v = rdDigit(*fs );
5
6   else if(*fs == '-')
7    c = v;
8    init = (c)? v : 1;
9   } else if (*fs == EOL) {
10   assert(init != 0);
11   break;
12  }
13  fs++;
14 }
```

P'

```
1  bool lhs = false;
2  while(true){
3   if(isDigit(*fs)){
4    v = rdDigit(*fs);
5    lhs = true;
6   } else if (*fs == '-'){
7    c = lhs;
8    init = (c)? v : 1; //IL
9   } else if (*fs == EOL){
10   assert(init != 0);
11   break;
12  }
13  fs++;
14 }
```

Figure 4.5: Core Utility `cut.v1` changed to `cut.v2`

4.3.3 Change Interaction Errors

A change-interaction error is a special kind of differential error. Informally, a change-interaction error happens when multiple changes are introduced in a program, and those multiple changes interact in unexpected ways. More formally we can define this class of errors as follows.

Definition 10 (*Change interaction error (CIE)*)

A change-interaction error happens when there exists a sequence of changed statements $\vec{C}$, such that both of the following conditions hold:
1) there exists an input i that exercises all changed statements in $\vec{C}$ in order and $S(i) \not\equiv P'(i)$;
2) for every input j that skips the execution of at least one changed statement in the sequence $\vec{C}$, we have that $S(j) \equiv P'(j)$.

We call the sequence $\vec{C}$ the *critical sequence* of the CIE. That is $\vec{C}$ corresponds to a sequence of changed statements that is necessary to expose the error. Any smaller sequence that skips the execution of at least one changed statement in that sequence cannot expose the error.

4.3.4 Running Example

For illustration purposes, we use the two concrete program versions P and P' in Figure 4.5 to explain salient concepts in the remainder of this work. The two programs are simplified extracts of two versions of the Linux core utility `cut` - the behavior of which is explained in Section 4.2.2. The code is related to the parsing of the user-provided number ranges for the tool. As long as `*fs` points to a character of the string, it tests whether the character is a digit (line 1), a dash (line 6) or the end of line (line 9). If the character is a digit, then the number is read into `v`. In the changed version a boolean `lhs` is set to `true` (lines

4-5). If the character is a dash, the variable `init` is computed using `v` (lines 7-8). If the end of line is reached, the bug is observable if `init` is 0 (line 10).

The *changed statements* are highlighted in grey. There are three changed statements in the changed version, which can be identified using the corresponding line numbers: $\{1, 5, 7\}$. We should point out that our notion of change is syntactic, purely textual and corresponds to code changes that manifest in the *changed version* (P'), such as added or modified statements. In other words, changed statements can be determined using textual differencing tools, like `diff`. The use of `diff` has the advantage that it works for any two programs, although it can be quite imprecise. There are other, more precise ways to deal with changes, but these typically assume some form of alignment between the two program versions [112, 1]. Unfortunately, these alignment assumptions do not always hold for the real programs that we are interested in. For this reason we chose the less precise, but unrestricted approach using `diff`. The loss of precision yields CSGs that may be larger than otherwise.

Change Interaction Error. In the program P' in Figure 4.5 a CIE happens when the input string is "0-". In this case the following sequence of changed statements is executed: $\langle 1, 5, 7 \rangle$. Before entering the loop, line 1 is executed. Since the first character is '0', the first iteration of the loop meets the condition at line 3 and the changed statement in line 5 is executed. At this point the variable v is set to 0 and the variable `lhs` is set to `true`. In the second iteration of the loop the condition at line 6 is met and the change in line 7 is executed. Since `lhs` is true, `init` is set to 0 (as `v` is 0). In contrast, for the same input, program P sets the variable `init` to 1. Consequently, in the last iteration of the loop, the assertion in line 10 is violated for P', but not for P.

Note that only input exercising specific (critical) sequences of changed statements triggers this error. The interaction of the changed statements in lines 5 and 7 at the statement in line 8 causes this error. The combined semantic impact of both changes lead to the differential evaluation of the conditional expression `(c)?v:1` in both versions, P and P'. Input exercising a sequence that "skips" any change in $\langle 1, 5, 7 \rangle$, such as $\langle 1, 7, 7 \rangle$ or $\langle 1, 7, 5 \rangle$, will not expose the error.

4.4 Change Sequence Graph

To support detection of change-interaction errors (CIE) we propose a statically computed structure which we call *change-sequence graph* (CSG). A change sequence graph approximates the computation of potential CIEs by using *control-flow information* to derive sequences in which the changed statements can potentially be exercised and *dependence information* to derive locations at which the changes can potentially interact.

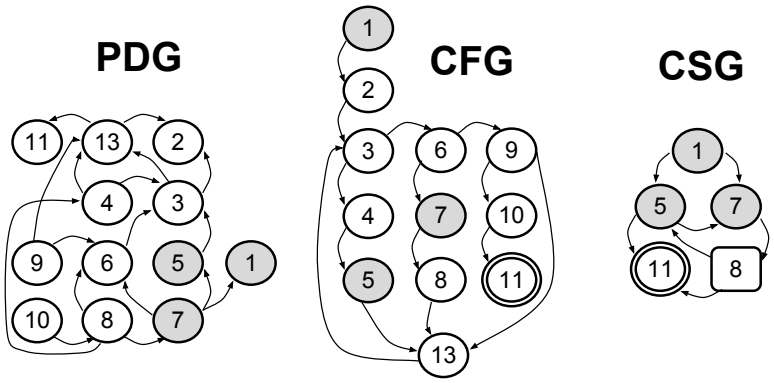


Figure 4.6: PDG, CFG, and CSG for P' in Figure 4.5.

4.4.1 Potential Interaction

To aid detecting CIEs, we can approximate all potential sequences of changed statements in a program using control-flow information from a control-flow graph (CFG). Essentially, a potential sequence of changed statements corresponds to a path in the CFG that contains changed statements. Having information about every potential sequences of changed statements is helpful because all critical sequences will be included in those sequences. In other words, this information will allow us to build a detection method for CIEs that searches potential critical sequences and exposes CIEs.

We are particularly interested in change sequences where the changed statements interact. That is, each executed changed statement has some impact on the output, and not executing one of those statements can lead to a different output. It is in this class of sequences where we can find change interaction errors. To detect such sequences, one useful definition is that of a potential interaction location of a sequence of changed statements.

Definition 11 (*Potential Interaction Location*)

A statement s is a potential interaction location of a sequence of changed statements $\vec{C}$, if s (statically) data- or control-depends on more than 1 changed statements in $\vec{C}$.

Information about potential interaction points can be computed using the program dependency graph (PDG). Essentially, we utilize the backward slice of the statement s to compute the set of changed statements that can have a semantic impact on s. If the set contains more than one different changed statement, then s is a potential interaction location of those statements. Note that an interaction location can coincide with a changed statement.

In Figure 4.6, the graphs on the left and in the middle depict the PDG and the CFG for our running example, respectively. The statement at line 8 is a potential interaction location of the changed statements in lines 5 and 7, since it transitively depends on both the changed statements. As such, both changes can have a combined semantic impact on this control-location, effectively causing the regression error.

The notion of potential interaction location allows us to define an approximation of *change interaction*.

Definition 12 (*Potential Change Interaction*)

> *A sequence of changed statements $\vec{C}$ is potentially interacting if there exists at least one potential interaction location for $\vec{C}$.*

Essentially, if there exists no interaction location for a sequence of changed statements, they are guaranteed not to interact and each of these changes can be tested in isolation. Otherwise, the changes might interact for some input.

The information about all potential sequences of changed statements and potential interaction points can be synthesized in a *change-sequence graph* (CSG). Thus a CSG represents a subset of program paths in a program where change-interaction errors *may* exist. Other program paths, which are not represented in the CSG, cannot have change-interaction errors as they do not contain change sequences.

4.4.2 Computing the Change Sequence Graph

The CSG can be computed using the CFG and the PDG for the changed program P'. Algorithm 5 shows the detailed construction of the CSG. The inputs of the algorithm are two programs P and P' and the output is the change-sequence graph CSG. The first step is to compute the changed statements between P and P' (line 1). As discussed in Section 4.3, this can be done using the `diff` tool. The next step is to compute the annotated versions of the CFG and PDG of P' (lines $2-3$). Both, the CFG and PDG are annotated with information about the changed statements. Initially the CSG contains no edges, only nodes. These nodes are the changed statements and output nodes that are recovered from the CFG using the procedure $markedNodesOf$ (lines $4-5$). The final step of the algorithm is to iterate through all the changed statements in the CFG and, for each change, use the auxiliary function TRAVERSECHANGE to add the relevant edges and interaction locations to the CSG (lines $5-7$).

The recursive function TRAVERSECHANGE takes two arguments *curr* and *c*. The first argument represents the current node in the CFG. The second argument represents the changed statement that edges may have to connect to.

Algorithm 5 Change-Sequence Graph Construction

Input: Programs P and P'

```
 1: let C_Code ← diff(P, P')
 2: let CFG ← markedCFG(P', C_Code)
 3: let PDG ← markedPDG(P', C_Code)
 4: let CSG ← markedNodesOf(CFG)
 5: for all Change c ∈ CFG do
 6:     traverseChange(c, c)
 7: end for
 8:
 9: function traverseChange(curr, c)
10:     for each node that directly follows curr in CFG do
11:         if node is change or output then
12:             add edge from c to node in CSG
13:         else
14:             let C_I ←dependsOnChanges(node, PDG)
15:             if |C_I| > 1 then
16:                 add node to CSG
17:                 for each c ∈ C_I do
18:                     add edge from c to node in CSG
19:                 end for
20:                 traverseChange(node, node)
21:             else
22:                 traverseChange(node, c)
23:             end if
24:         end if
25:     end for
26: end function
```

Output: Change-sequence graph CSG.

For each node in the CFG, which directly follows from the current node, we have three possibilities for the *node*:

Change or output node (lines $11-12$): If we reach some other change node, this indicates that there *may* be a control-flow from the change c to this change. Thus, we add a corresponding edge to the CSG to indicate such potential flow. Similarly, if we reach an output node, we should add an edge between change c and that node to indicate the potential control-flow.

Interaction location (lines $14-20$): If the *node* is an interaction location, it is added to the CSG and connected. Specifically, the function dependsOnChanges(*node*, PDG) computes the changed statements that can have a semantic input on *node* using the PDG. If there is more than one change having a semantic impact on *node*, then *node* is an interaction location and is added to the CSG connected to the changes it depends on. Conceptually, every interaction location can be regarded as a new change. To trace the semantic impact of the interaction location, we keep recursively traversing the CFG by invok-

ing TRAVERSECHANGE with both arguments set to *node*. Naïvely, the function DEPENDSONCHANGES can return all changed statements in the static backward slice of *node*. For optimization purposes, the function may choose a *node* to be an interaction location only if *node* is an "important" interaction location in some respect. For instance, given an interaction location i for change sequence $\vec{C}$ and a statement s that directly depends on i. While s is also an interaction location of $\vec{C}$, it may not be an important one. Alternatively, the interaction locations could be computed by taking the static forward slice for every changed statement and marking their intersection as an interaction location.

Neither of above (line 22): Any other CFG node should be ignored in the CSG. This is achieved by calling TRAVERSECHANGE with *node* as first argument. This sets *curr*, representing the node in the CFG that is currently traversed, to *node* and implies that *node* will not appear in the CSG.

4.5 Search-based Input Generation

Algorithm 6 Search-Based Input Generation

Input: Programs P and P'; Directed Graph CSG

```
 1: let T ← ∅
 2: let symbState.targets ← CSG.startNodes
 3: let symbState.pc ← true
 4: let symbState.next ← P'.firstStmt
 5: let states ← {symbState}
 6: while states ≠ ∅ ∧ ¬isTimeout() do
 7:     let bestState ← chooseBestState(states)
 8:     let s ← symbExec_next(bestState, P')
 9:     if isBranch(s) then
10:         let stateT ← bestState.pc ∧ s.branchCond
11:         let stateF ← bestState.pc ∧ ¬s.branchCond
12:         remove bestState from states
13:         add stateT and stateF to states
14:     else if s ∈ bestState.targets then
15:         if s is an output then
16:             let t ← smt_solve(bestState.pc)
17:             if P(t) ≠ P'(t) then
18:                 add t to T
19:             end if
20:             remove bestState from states
21:         else
22:             bestState.targets ← next targets of s in CSG
23:         end if
24:     end if
25: end while
```

Output: Difference-revealing test cases T.

To expose change interaction errors, and differential errors in general, test cases are generated. The exploration technique uses the Change Sequence Graph as a guide to exercises the structure of inter-dependencies across the changed statements. We employ symbolic execution along these dependencies.

The search-based input generation is depicted in Algorithm 6. The algorithm takes two program versions, P and P', and the CSG (cf. Alg. 5) as input and computes a set of difference-revealing test cases T. We adopted the directed symbolic execution algorithm as discussed by Ma et al. [128]. However, instead of searching for input that exercises any target in a specified (flat) set of targets, we extended the algorithm to search a specified directed graph of targets (i.e., the CSG). The search algorithm is presented independent of the search strategy.

Algorithm 6 is initialized in the first five lines. It starts with an empty test suite T and the first set of changed statements in the CSG (those without incoming edges). These are added as targets for the symbolic state *symbState* which is created in lines 2-4. A symbolic state is essentially an intermediate state of symbolic execution and has three main properties - (i) a statement *next* which is to be executed next, (ii) a partial path condition *pc*, that is satisfied by every input exercising the same program path until s, and (iii) a set of *targets*.

The symbolic execution of a symbolic state can be *resumed* at any time and *pauses* when a branch or a target is reached. The first symbolic state *symbState* is created with $pc = true$, statement *next* is set to the program start, and the *targets* are assigned to the first set of changed statements in line 4. In the following line 5, it is added to the empty list of symbolic *states*.

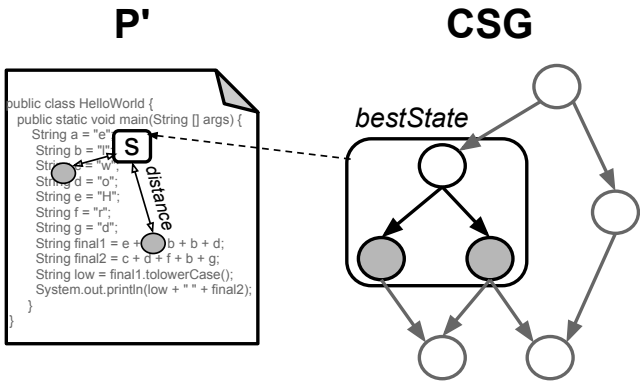


Figure 4.7: The *bestState* is chosen with the shortest distance in the source code of P' from s to the target (left). Once a target is reached, the symbolic state moves to the target's children in the CSG (right).

The search commences in line 6. As long as the list of *states* is non-empty and no timeout occurs, the search works as follows. From the list of *states* the *bestState* is chosen according to a given search strategy, which is implemented in

chooseBestState. For instance, as depicted in Figure 4.5, every symbolic state is assigned a measure of distance to its targets, ranked according to this measure, and chosen if it has the shortest distance. We further prioritize states with a greater proportion of targets that are yet unreached by other symbolic states. In line 8, the *bestState* resumes the symbolic execution of P' until s, the next statement to be executed, becomes either a branch or one of the targets to be reached. If s is a branch (lines 9-12), then two states are created - one following the *true*-branch and the other following the $false$ branch. The path conditions and the list of *states* are updated accordingly. If s is a *target* (lines 13-23), then we further distinguish whether or not s is an output statement. If s is an output statement, then we solve the path condition using a Satisfiability Modulo Theory solver to derive a concrete program input t (line 15). This input is executed on both versions to validate whether t exposes a behavioral difference. If so, t is added to the set of difference-revealing test cases T. Since *bestState* reached the output, it requires no further symbolic execution and can be removed from the list of *states* (line 19). Otherwise, if s is a target of *bestState* and not an output statement (line 21), then we set as new *targets* of *bestState* the nodes following the outgoing edges of the reached node in the CSG. The right-hand side of Figure 4.5 shows the *bestState* searching for two CSG nodes (in grey). If *bestState* finds the node on the left, the next target of *bestState* becomes that bottom left node.

4.6 Empirical Evaluation

4.6.1 Implementation and Setup

We have implemented Algorithm 6 into the directed symbolic execution tool, Otter [128]. The user provides two versions of a C program compiled into the C Intermediate Language (CIL) and a text file with a representation of the CSG.

Otter provides a wide choice of search strategies which implement the function *chooseBestState* in Algorithm 6. For our experiments we used one of the most efficient[8] strategies. The best symbolic state is chosen based on the shortest distance to the targets computed in the interprocedural control-flow graph. Occasionally, the next state is chosen randomly. We extended the search strategy by prioritizing states with a greater proportion of yet unreached targets. Instead of searching for a global set of targets, our implementation extends a symbolic state to have its own set of targets. Once a target is reached, the children of the reached target become the new targets for this state. The execution of a symbolic state terminates only if the output has been reached and thus

[8] `RoundRobin(RandomPath,InterSDSE-efficient)`.

no more further targets are to be reached. We then compute a concrete input satisfying the path condition of a state that reached an output node. This input is executed on both program versions. The information from the standard unix pipes `stdout` and `stderr` describes the program output. If the output differs in more than the program name, the test case and its differing output is reported. If the user *optionally* provides a "golden version", our implementation can classify the observed differential output further as "differential error" (i.e., regression/incomplete fix) or "progression".

We executed our implementation on a desktop computer with an Intel Core2 Quad CPU at 2.83GHz and 4GB of main memory to generate test cases within the time frame of 5 minutes. The same sequence of changed statements can be exercised by multiple generated test cases.

4.6.2 Subjects

We chose the subjects according to the following criteria:

Known Regressions. For every regression, we know i) the earlier version, ii) the regression-introducing version, iii) the bug report(s), and iv) the regressing-fixing version(s). The analysis of known regressions increases the credibility of the subjects and reduces the scope of non-maintenance commits which we need to inspect.

Multiple Changes. In this study, we are not interested in the semantic impact of single changed statements but the interplay of multiple changed statements. Therefore, we consider only regressions involving multiple changed statements.

Deterministic Behavior. The execution of the same input on the same program always yields the same output. Determinism is a prerequesite for many testing techniques including (standard) dynamic symbolic execution.

Version Pair	Fixed in Revision	Commit Date	Bug Report @ http://lists.gnu.org/
seq.v0→seq.v1	seq.v2	09.07.2007	2007-07/msg00055.html
16.06.05→01.07.06	seq.v3	14.02.2009	2009-02/msg00139.html
	seq.v3	14.02.2009	2009-02/msg00139.html
seq.v1 →seq.v2	**seq.v4**	24.11.2012	2012-11/msg00145.html
01.07.06→09.07.07	**seq.v5**	10.01.2013	2013-01/msg00054.html
cut.v0→cut.v1	cut.v4	07.02.2011	2011-02/msg00036.html
02.06.04→04.12.04	**cut.v6**	24.11.2012	2012-11/msg00151.html
cut.v1→cut.v2	cut.v3	22.05.2007	2007-05/msg00195.html
04.12.04→22.05.07	**cut.v5**	18.11.2012	2012-11/msg00114.html
cut.v6→cut.v7	**cut.v8**	05.02.2013	2013-02/msg00011.html
24.11.12→06.12.12			
expr.v0→expr.v1	expr.v2	26.05.2005	2005-05/msg00189.html
16.11.04→14.01.05			

Figure 4.8: Subjects - Version history

We study six version pairs that together introduced 11 regression bugs, five of which are found and reported by our method (in **bold** font). Figure 4.8 shows the considered `Version Pairs` that the latter version of which introduces bugs that are `Fixed in` the a subsequent revision. The fixes are presented with `Commit Date` and `Bug Report`. The bug being fixed with `cut.v4` is further discussed by Marinescu and Cadar [130] and together with `cut.v9` only observable as buffer overflow. We inserted an assertion that states that an array shall never be accessed out of bounds. The tools `cut`, `seq`, and `expr` consist of about 900, 500, and 900 Lines of Code (LoC), respectively. However, these tools utilize monolithic, shared libraries, prompting colleagues to quote between 2k to 3k effective LoC for the smallest tools [129] up to 20k instructions for the largest tool [130] in `GNU Coreutils`.

4.6.3 Research Questions

During the empirical evaluation of the change-interaction guided regression test generation technique, we want to answer the following research questions.

RQ.1 Severity. How many differential errors can be classified as change interaction errors? What is the probability to exercise a sequence critical to exposing a change interaction error compared to sequences that are not?

RQ.2 Efficacy. How many differential errors are exposed by a test generation technique that *does not* stress the inter-dependencies and thus potential interactions among the many changes as compared to one that does?

4.7 Results and Analysis

		RQ1: Change Sequence Graph				RQ2: Individual Changes		
Version Pair	**Fixed in**	**CIE**	**#Tests**	**#Diff**	**#Error**	**#Tests**	**#Diff**	**#Error**
seq.v0 → seq.v1	seq.v2	x	163	43	6	205	65	0
	seq.v3	x	163	43	5	205	65	0
seq.v1 → seq.v2	seq.v3	-	200	26	2	200	21	17
	seq.v4	-	200	26	3	200	21	0
	seq.v5	x	200	26	1	200	21	0
cut.v0 → cut.v1	cut.v4	-	379	42	30	471	42	30
	cut.v6	x	379	42	12	471	42	12
cut.v1 → cut.v2	cut.v3	-	254	228	162	453	201	58
	cut.v5	x	254	228	26	453	201	5
cut.v6 → cut.v7	**cut.v8**	x	324	4	4	342	6	6
expr.v0 → expr.v1	expr.v2	x	42	2	2	82	2	2
Average (per version pair)		*7/11*	*227*	*57.5*	*46.3*	*292.2*	*55.8*	*21.7*

Figure 4.9: Bugs introduced, fixed in later versions, are witnessed by test cases generated within 5 minutes.

4.7.1 Result Presentation

Figure 4.9 shows the bugs introduced when changing the given versions, whether these are change interaction errors and the test cases generated by our CSG-guided test generation technique. The first two columns show the errors introduced by the changes of the `Version Pairs` that are `Fixed in` the versions given in the second column. For instance, when program `seq.v1` was changed to `seq.v2`, errors are introduced that are fixed in versions `seq.v3`, `seq.v4`, and `seq.v4`. Errors highlighted in **bold** face were previously unknown and subsequently reported by us. The subsequent four columns show the results for the generation of test cases exercising the `Change Sequence Graph`, while the latter three columns show the results for a test generation technique that considers sufficient to exercise every changed statement, effectively treating them as `Individual Changes`. Both groups of columns have a similar format. Column `#Tests` depicts the number of test cases generated. Column `#Diff` depicts the number of test cases revealing a difference when executed on both versions. Some of the semantic differences are expected (progression). Column `#Error` depicts the number of test cases that are not expected and expose the respective error. An error, that is exposed only by input exercising a sequence of changed statements but not by input "skipping" statements in that sequence, is classified as change interaction error (Col. `CIE`).

Version Pairs	Sequence	%Test	%Error
seq.v0 → seq.v1	non-critical	19.02%	0.00%
	critical	80.98%	1.39%
seq.v1 → seq.v2	non-critical	99.50%	0.30%
	critical	0.50%	100.00%
cut.v0 → cut.v1	non-critical	96.83%	4.09%
	critical	3.17%	100.00%
cut.v1 → cut.v2	non-critical	87.40%	11.71%
	critical	12.60%	33.33%
cut.v6 → cut.v7	non-critical	95.68%	0.00%
	critical	4.32%	28.57%
expr.v0 → expr.v1	non-critical	71.43%	0.00%
	critical	28.57%	16.67%

Figure 4.10: `%Test` generated test cases exercise a (non-) critical sequence. `%Error` generated test cases exercising a (non-) critical sequence expose an error.

Figure 4.10 shows the percentage of tests exercising critical sequences versus the percentage of tests exercising non-critical sequences. One test case exercises exactly one sequence. A *critical sequence* is a sequence of changed statements that is relevant to expose a change interaction error. The first column depicts the `Version pairs` considered, followed by whether the results refer to `critical` or `non-critical` sequences. The latter two columns are explained by example of the last row: "On average, one quarter of the generated test cases for the version pair expr.v0 and expr.v1 exercise a critical sequence. From those, every sixth exposes an error".

To generate the test suites that stress changes individually (see RQ.2), we generated test cases that cover every changed statement that is also exercised by the approach presented in this chapter. We set as targets the output and such statements that have the greatest depth in the chain of control-dependencies. In other words, instead of a graph of targets, we provided a set of targets. Otherwise, we employed the same tool, search strategy, and time frame.

RQ.1 Change Interaction Errors

Two thirds of the differential errors can be classified as change interaction errors. Only one in five test cases exercise a critical sequence, being 15 time more likely to expose an error.

Using our implementation, we have found and reported four of the seven listed change-interaction errors and one more differential error, that were previously unknown. On average, 227 test cases were generated that *exercise* a change sequence (see Figure 4.9). Every fourth test case *propagates* the combined semantic effect of the exercised changed statements to the output and thus makes a difference observable. While many of these expose expected behavioral changes, every fifth test case exposes a differential error.

Change interaction errors are *subtle*. On average, only 21.7% of the generated test cases exercise a critical sequence (see Figure 4.10). On the other hand, the malicious *effect* of a critical change sequence is much greater than that of a non-critical sequence. Only 3.2% of the test cases exercising a non-critical sequence expose an error versus 50% exercising a critical sequence. Test cases exercising a critical sequence are 15.6 times more likely to expose an error than test cases exercising a non-critical sequence. That suggests that the changes in these critical sequences are interacting in a negative and unintended form.

RQ.2 Comparison to Stressing Changes Individually

Only 57% of the change interaction errors are exposed by test cases generated by a technique disregarding potential change interaction.

To compare, we generated a test suite that covers every changed statement which is also covered by the test suite generated using a change sequence graph. On average, 292 test cases were generated that *exercise* a change sequence (cf. Figure 4.9). Every fifth test case *propagates* the combined semantic effect of the exercised changed statements to the output and thus makes a difference observable. Many of these expose expected behavioral changes, every 15th test case exposes a differential error – significantly less than our CSG-based test generation approach. Within five minutes, using our CSG-based approach every error is witnessed by 25 test cases on average. In contrast, using the other

approach that considers changes individually only seven of the eleven errors are witnessed by, on average, 18 test cases each. In particular, only 57% of the change interaction errors are exposed by test cases generated to stress changes individually as compared to 100% by our technique.

4.8 Threats to Validity

The main threat to *external validity* is the generalization of the results. During our study of `GNU Coreutils` we encountered several regression errors that can only be observed when certain environmental conditions are satisfied. One example is an error that was reported to occur specifically on a Solaris 32-bit maching and could not reproduced on other machines. Depending on the program environment, the same test case may or may not expose an error. In fact, the package co-maintainer of `GNU Coreutils`, Pádraig Brady, noted in an email correspondence that it may be unclear even for the experienced developer, exactly how to write the test cases in the presence of such non-determinism. He suggested to introduce an explicit interface for file operations. This suggests a lack of modelling the environment [60], or concurrency [132] during the testing process. As discussed in Section 4.6.2, our experimental subjects and regression errors are chosen so that the observability of an error does not depend on the program environment but on source code properties. The conclusions should be viewed in the same context.

The main threats to *internal validity* are T.1) the search strategy that was utilized and T.2) the practical absence of assertions that mark an error within symbolic execution. T.1) The experimental results depend on the utilized search strategy. A less efficient search strategy may have exposed less differential errors within the same amount of time. However, the utilized search strategy does not prioritize critical over non-critical sequences. Thus, it does not affect the main conclusion of *RQ.1*. We utilized the same search strategy for the experiments that compares to testing changes individually. Thus, it does not affect the main conclusion of *RQ.2*. T.2) Symbolic execution requires highlighting of error states, for instance, by assertions. In Section 4.2 and Figure 4.8, we list the versions `cut.v4` and `cut.v8` as bug fixes for regressions introduced in an earlier version of `cut`. The regressions are observable as buffer overflows. However, without the explicit assertion stating that an array should never be accessed at an index greater than its size, the symbolic index for this array may often concretize as small number, such as 1 or 0, but never as a number that has more than the nine digits necessary to witness these particular overflows. While our *implementation* is able to find error-exposing test cases in the presence of such assertions, it is unable in their absence for such buffer-overflows.

4.9 Related Work

Test Suite Augmentation aims at generating new test cases that stress the changed behaviour in a program. Typically, this is done by exploiting *knowledge about changes* and using *symbolic execution* techniques - which are also key ideas in our approach. However, the main novelty of our work is the consideration of the inter-dependencies among multiple changes during test generation. Our technique effectively exercises sequences of changed statements and potential interaction locations. Existing techniques either discuss the semantic impact of single changes only [112, 1], or do not systematically consider the interaction and inter-dependencies among multiple code changes [2, 100].

Test Suite Augmentation (TSA) techniques can be distinguished in *semantic approaches* [133, 39], that are based on the program summaries of both versions to compute the semantic changes, and *syntactic approaches*, that are directed by the syntactic changes to exercise paths that may expose semantic changes. The syntactic techniques can be further distinguished into those seeking to re-establish code coverage of a test suite after the program is changed [95], those following the Reach-Infect-Propagate[9] approach [112, 2, 1], and those exercising every program path affected by a change [100, 25].

Techniques, such as eXpress [100] or DiSE [25], that exercise every program path *affected* by changed statements, are finer-grained and less scalable than our approach. The focus on affected code regions makes these techniques more efficient than full path exploration approaches, like DART [21], since less paths are to be explored. However, these techniques may still exercise *many different paths within the same sequence* of changed statements; paths that may or may not contain interaction locations; paths that may all expose the same error. More systematically, our CSG directed TSA approach targets *sequences and interaction locations of changed statements* instead of all *affected paths*. In practice, this means that once a difference revealing test case is found for a sequence, unexplored affected paths that can still realize this sequence do not no have to be explored further.

TSA techniques based on Reach-Infect-Propagate (RIP) [2, 1, 110] follow a motivation similar to our work: Instead of exploring *every path* affected by changes, the RIP approaches deem it sufficient to find *one path* that executes a change, infects the program state, and propagates to the output. However, existing techniques consider the semantic effects of the changes in isolation. For the subjects in our experiments, a technique based on this consideration could expose only half of the change interaction errors. In the presence of multiple

[9] Reach a change, infect the program state, and propagate the infection to the output [52].

changes, the approach of Santelices et al. [2] requires a change-free path from the change to the program start - effectively a change in isolation.

Coverage-based TSA techniques seek to re-establish code coverage when the program is changed [95, 97]. However, to expose change-interaction errors and understand the combined semantic impact of multiple changes, it is insufficient to merely exercise every change, as discussed earlier.

Semantic TSA techniques [133, 39] require the computation of a differential semantic program summary for both versions to determine the semantic changes. While this approach is sound and very precise, it may be less scalable.

Higher-order Mutation-based Testing [134, 26] suggests that a high-quality test suite kills a large percentage of higher-order mutants. Each *higher-order mutant* is an automatically generated version of the program under test that contains several small changes to the program statements. The order of the mutant is determined by the number of changed statements. A mutant is considered *killed* by a test case t if t exposes an output difference when executed on both versions. Conceptually, these mutants represent faulty versions of the correct program and a good test suite differentiates the correct from many faulty versions. Our work in this chapter suggests that higher-order mutants can produce a class of errors – Change Interaction Errors – that is not observable for standard first-order mutants.

Combinatorial Interaction Testing (CIT) [135, 136] is a black box input sampling technique by (randomly) composing potentially interacting program inputs from atomic ones. For instance, a program is tested by sampling the space of all possible program configurations. Or it can be tested by sampling all possible button-clicks and text-field inputs for its graphical user interface. The hypothesis, shared with our work, is that the individual parts (here, atomic program inputs) may potentially interact and stress program behavior that cannot be observed by treating these parts in isolation.

Change Interaction. Santelices et al. [54] propose a formal definition of change interaction: two changes c_1 and c_2 interact in an execution if removing one of the changes alters the semantic effect of the other change on that execution. This notion of change interaction is too precise. For our practical purposes, detecting such changes interactions cannot be done in an efficient manner. Essentially, given a test case t and code changes C that are applied to program P yielding P', there are $2^{|C|}$ program configurations to be analyzed, each with only a subset of C applied to P. Our definition of potential change interaction approximates the above definition and can be computed more efficiently. A set of changed statements C potentially interacts if there exists a statement that syntactically depends on every $c \in C$.

Reachability. In order to explore change sequences our approach builds and extends previous work that deals with reaching statements in a program. However, these tools seek to reach a single statement [1, 137], a set of statements [128], or a sequence of statements [24] instead of a graph. To overcome this problem we have modified the Otter tool [128] to take a graph of statements as input and target multiple statements along this graph structure at once.

4.10 Chapter Summary

In this chapter, we have proposed the systematic exploration of the complex dependency structure among the many (potentially interacting) syntactic changes between two program versions. We applied this technique to realistic regression errors to study the prevalence and nature of regression errors that come into existance only due to the interaction of several syntactic changes – so called Change Interaction Errors (CIEs).

We have argued for the importance and subtleness of such change-interaction errors, which are pervasive even in well-tested and widely used software. Since existing regression test generation techniques do not adequately stress code where change interaction may occur, we have proposed a new regression test generation technique that addresses these limitations. Our recipe for exposing change-interaction errors employs a judicious mix of flows, dependencies and semantic effects across changes. In other words, to witness a change interaction error – multiple changes should be executed (flow information), multiple changes should affect a potential interaction location via data- and control dependencies (dependence information), and the semantic effect of a change should not get masked. In our approach, the control flow between changes is captured in the Change Sequence Graph, dependencies across changes are witnessed in potential interaction locations, and we attempt to exercise these dependencies and propagate their semantic effects via symbolic execution on the changed program. Our experiments on `GNU Coreutils` demonstrate the effectiveness of this approach in hunting down hard-to-find change-interaction errors even in well-tested software.

Chapter 5

On the Complexity of Regression Errors

> "Simplicity does not precede complexity, but follows it."
>
> — Alan Perlis in *Epigrams on Programming* [138], 1922 – 1990

Intuitively we know, some software errors are more complex than others. If the error can be fixed by changing one faulty statement, it is a simple error. The more substantial the fix must be, the more complex we consider the error.

In this work, we formally define and quantify the complexity of an error w.r.t. the complexity of the error's least complex, correct fix. As a concrete measure of complexity for such fixes, we introduce Cyclomatic Change Complexity which is inspired by existing program complexity metrics often used in practice.

Moreover, we introduce CoREBench, a collection of 70 regression errors systematically extracted from several open-source C-projects and compare their complexity with that of the *seeded* errors in the two most popular error benchmarks, SIR and the Siemens Suite. We find that seeded errors are significantly less complex, i.e., require significantly less substantial fixes, compared to actual regression errors. For example, among the seeded errors more than 42% are *simple* compared to 8% among the actual ones. This is a concern for the external validity of studies based on seeded errors and we propose CoREBench for the controlled study of regression testing, debugging, and repair techniques.

5.1 Introduction

Software errors can be arduous. Their fixes can account for half of the code changes even in well-tested software [139]. Before they are fixed, they can remain in the program for many years, causing problems for the software users. When they are fixed, these fixes can introduce even further errors.

Related processes can be automated based on our understanding of the inherent nature of software errors. Testing techniques seek to expose errors; debugging techniques seek to determine the faulty source code for an error; and repair techniques seek to fix the faulty source code.

Two pertinent properties of software errors are complexity and detectability. While the *complexity* of an error is determined by how substantial the error's fix is required to be, the *detectability* of an error is determined by the amount of input exposing the error. Intuitively, an error that is hard to detect may still require only a simple fix. Offutt [140] relates both properties and conjectures: the detectability of simple faults is similar to the detectability of complex faults – the coupling effect hypothesis. He defines simple faults as ones that *can* be fixed by changing one statement while complex faults cannot.

In this dissertation, we are the first to *quantify error complexity* and formally define the term and a metric. The complexity of an error is determined by the complexity of the correct, least complex *fix* of the error. The fix must be *correct* because no other errors should be introduced and *least complex* because even Offutt's simple faults can be fixed in multiple ways, including a complete revision of the program.

To measure the complexity of a fix, we formally define *software change complexity* and introduce a concrete change complexity metric – *Cyclomatic Change Complexity* (CyCC), which is inspired by McCabe's cyclomatic program complexity metric [141]. Program complexity is a measure of the interactions among the various elements of the software. Similarly, we define the change complexity as a measure of the interaction among the various *changed* elements in the *changed* software. We give an efficient algorithm to compute CyCC.

Equipped with our novel error complexity metric we set out to learn about the nature of complex regression errors. The two most popular benchmarks for experimentation with regression errors are the Siemens Suite [142] and SIR [122]. In both cases, most errors were introduced through a process called *fault seeding*. Developers were asked to change the given programs slightly such that they contain errors of varying detectability. However, we were not certain about a varying complexity of the *seeded* errors and constructed our own benchmark to compare to *actual* regression errors.

In this chapter, we introduce CoREBench as a collection of 70 regression errors and compare the complexity of these to the complexity of the seeded ones in the Siemens Suite and SIR. We harvested the regression errors in CoREBench *systematically* from four widely deployed, well-tested open-source software projects. Indeed, we find that the seeded regression errors are significantly less complex, i.e., require significantly less substantial fixes, compared to the actual regression errors in CoREBench. For example, among the seeded errors more than 42% are simple compared to 8% among the actual ones. *This is a concern for the external validity of studies based on such seeded regression errors.*

We apply our error complexity metric to the regression errors in CoREBench in order to experimentally investigate the nature of complex regression errors. Three of our main findings are enumerated in the following:

- Between the complexity of the change introducing an error and of the change fixing it seems to be no correlation. That is, even simple changes can introduce complex errors. One could say that the *cause* of a regression error is already dormant in the code and the change merely *triggers* it. Or, the regression errors may be evolving when the program is and the *complexity of errors may change during evolution.*

- Between the complexity and life span[1] of an error seems to be no correlation. That is, even complex errors may be fixed on the same day when they are introduced or a few years later. This may be *indirect evidence that simple and complex errors are of similar detectability*, i.e., coupled [140].

- Change Interaction Errors (CIEs)[2] require consistently more substantial fixes than other types of regression errors (Non-CIEs). This suggests that *CIEs are not only of less detectability (cf. Chapter 4) but also of greater complexity* than Non-CIEs.

We define change complexity as a measure of interaction among the changed elements and introduce the CyCC as a concrete metric. Yet, there are other metrics, such as number of Changed Lines of Code (CLoC), paths, or hunks. We study CLoC versus CyCC and find: While both rarely agree on the *specific* value or rank of a change's complexity, they strongly correlate *in general*. Basically, both indicate high complexity for substantial change. We believe, CyCC is a precise and practical measure of change complexity.

[1] The life span of an error is the time an error is observable from when it is introduced to when it is fixed.

[2] A regression error is a CIE if a *sequence* of changed statements must be executed in order to expose the error while "skipping" one of them does not expose the error (see Chapter 4).

In summary, this chapter makes the following *contributions*.

1. **Error Complexity Metric**. We formally define and quantify the complexity of an error w.r.t. the complexity of the error's least complex and correct fix. Investigations into error complexity are relevant for software testing, debugging, and repair: What is the root cause of an error that requires a substantial fix? Is a test suite adequate to expose complex errors? How do we correctly and efficiently repair complex errors?
2. **Change Complexity Metric**. We formally define software change complexity, introduce CyCC as a concrete complexity metric, discuss an algorithm to compute the CyCC efficiently based on a graph containing the control-flow among the changed statements, and make available a tool that computes the CyCC of any C source code commit in under one second on average.
3. **Regression Error Benchmark**. We make available CoREBench, a collection of 70 realistically complex regression errors. For each error, we provide the bug report, the error-introducing source code commit, the error-fixing source code commit, and a validating test case that fails for all versions between these commits, but passes before and after.
4. **Empirical Study**. We study the complexity of actual regression errors and establish that seeded errors in existing benchmarks are significantly less complex.

CoREBench and the implementation of CyCC are available at `http://www.comp.nus.edu.sg/~release/corebench`.

The artifact evaluation committee of ISSTA 2014 has found CoREBench and the CyCC tool to exceed expectations.

5.2 An Error Complexity Metric

We define the complexity of an error w.r.t. the complexity of the correct, least complex fix of the error. To measure the complexity of a fix, we formally define software *change* complexity as a measure of the interaction among the *changed* elements in a *changed* program and propose a concrete change complexity metric. Cyclomatic Change Complexity (CyCC) directly measures the number of linearly independent[3] *change sequences* in a changed program and is thus inspired by McCabe's cyclomatic program complexity. Intuitively, CyCC quantifies the amount of *changed* decision logic in the program.

[3] A *linearly independent path* is a complete path through the program that introduces at least one new edge that is not included in any other linearly independent paths.

```
351   : intmax_t value = 0;
352   : int sign = (*valuestring == '-' ?  -1 :  1);
353   : if (sign < 0)
354   :   valuestring++;
355   : do {
356   :   if (ISDIGIT(*valuestring))
357   :     value = 10*value + sign * (*valuestring-'0');
358   : } while (*++valuestring)
359---: return value * sign;
359 ++: return value;
```

Figure 5.1: Fix of simple error `core.6fc0ccf7`

Figure 5.1 shows an example of a simple error in `coreutils`. The simplified code fragment parses a `valuestring` into an integer `value`. However, every string containing a negative number is parsed as a positive number. This error is simple because only one statement (in line 359) needs to be changed in order to repair the error.

```
447++: else if (ent->fts_info == FTS_NS) {
448++:   if (ent->fts_level == 0){
449++:     reportSymlinkLoop();
450++:   } else {
451++:     if (symlink_loop(ent->fts_accpath)){
452++:       reportSymlinkLoop();
453++:     }
454++:   }
456++: }
```

Figure 5.2: Fix of complex error `find.24bf33c0`

Figure 5.2 shows an example of a complex error in `findutils`. The bug report states that "`find` does not report symlink loop when trying to follow symlinks". Hence, the developer adds the presented code fragment to describe conditions under which symlink loops need to be reported. The error is complex because it requires three additional conditional statements and several statements to fix it correctly.

5.2.1 Measuring Change Complexity

Traditional program complexity measures the interaction among the elements in a software system. So, we can define:

Definition 13 (*Change Complexity*)

Change complexity is a measure of the interaction among the changed elements in a changed program.

Note that deleted statements are changed elements nevertheless and can be represented by dummy statements in the changed program (see e.g., [2]).

As a concrete measure of change complexity, we introduce CyCC which is computed based on a graph containing the control-flow among the changed basic blocks – the CSG.

Definition 14 (*Change Sequence Graph (CSG)*)

> *The change sequence graph of a changed program P' is a directed graph containing as vertices the program entry as source, the program exit as sink, and the changed basic blocks in P', with an edge between any two vertices if control may pass from the first to the second without passing through a third.*

The source vertex is connected through an edge to every changed basic block that may be *executed first*, that is, before some other changed basic block is executed. To the sink vertex is connected every changed basic block that may be *executed last*, that is, after any other changed basic block is executed. This simplified definition of CSG accounts for all sequences of changed statements that can be exercised but not for potential interaction locations (see Chapter 4) and can be computed from the changed program's Control Flow Graph.

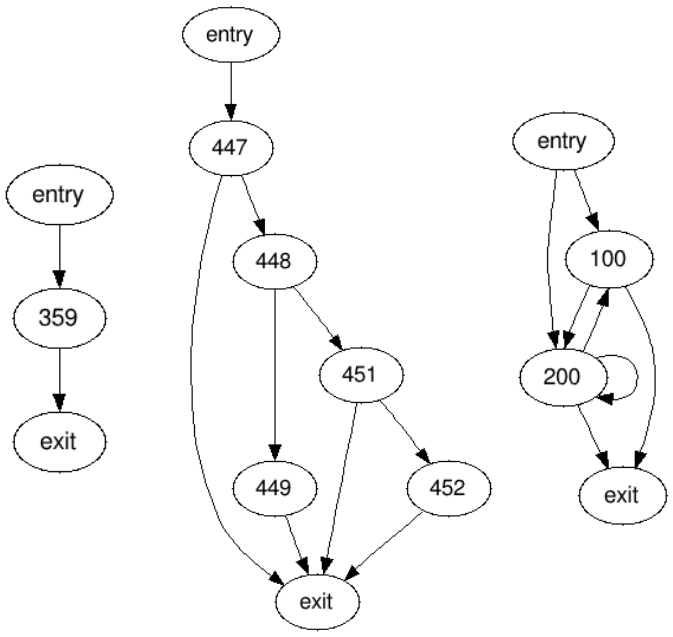


Figure 5.3: Change sequence graphs with linear independent paths `(359)` (left); `(447)`, `(447-448-449)`, `(447-448-451)`, `(447-448-451-452)` (middle); and `(100)`, `(200)`, `(100-200)`, `(200-100)`, `(200-200)` (right).

For example, Figure 5.3 depicts three different CSGs. The paths through a CSG from source to sink represent different sequences of changed statements that may be executed. The CSGs on the left and in the middle are computed for the changed code fragments in Figures 5.1 and 5.2. It is interesting to note that the size of the CSG depends only on the size of the changed code and not on the size of the complete program.

Definition 15 (*Cyclomatic Change Complexity*)

> *The complexity of a set of program changes C is defined with reference to the Change Sequence Graph constructed for C as $CyCC = E - N + 2P$, where*
> *E is the number of edges of the CSG,*
> *N is the number of nodes of the CSG, and*
> *P is the number of connected components in the CSG.*

Cyclomatic Change Complexity (CyCC) measures the number of linearly independent *sequences of changed statements* from entry to exit in a changed program. We argue that the changed statements in each sequence may "interact" differently. In fact, some sequences are critical in exposing so called Change Interaction Errors (see Chapter 4) while others are not. In Figure 5.3, based on the number of linearly independent paths in the CSG, we compute a CyCC=1 (left), CyCC=4 (middle), and CyCC=5 (right), respectively.

5.2.2 Measuring Error Complexity

Before we define and measure the complexity of an error, we quote the IEEE glossary to define what we mean by error.

Definition 16 (*Software Error [143]*)

> *A software error is the difference between a computed, observed, or measured value or condition and the true, specified, or theoretically correct value or condition.*

An error's *detectability* is determined by the proportion of program inputs that expose the error. Such input is said to fail w.r.t. the error. For example, the code fragment in Figure 5.1 parses negative numbers incorrectly. E.g, input setting `valuestring` to "`-2`" fails w.r.t. the error as it produces the output `value` of `2` instead of `-2`. If `valuestring` is directly a program input, then the error has a high detectability.

Definition 17 (*Error Complexity*)

> *The complexity of an error E is the complexity of the least complex change required to pass all input that fails w.r.t. E while the output for all other input remains unchanged.*

Intuitively, we define the complexity of an error based on how substantial its fix must be – without introducing new errors. For example, the error in Figure 5.1 can be correctly repaired with the change of only one statement (line 359). With $CyCC = 1$, it is a *simple error*. The error in Figure 5.2 can be repaired with a change involving three additional conditional statements. Assuming short-circuit evaluation for these conditions (see [144]), the CSG in Fig. 5.3 (middle) might be the least complex. The error is of complexity four.

However, we note that other measures, such as number of changed LoC, paths, or hunks, may assign different *specific* values to an error's complexity. While different measures may disagree on its specific value or rank, they should correlate in general (see RQ1 in Sec. 5.5). For instance, if an error requires a substantial fix involving a *high* number of changed LoC distributed over the code, the values for other measures of complexity should be *high* as well. We believe that CyCC is a precise and practical measure of change complexity as given in Def. 13 and thus of error complexity as in Def. 17.

5.3 Computing Inter-procedural Change Sequence Graphs

We present an algorithm to synthesize the *inter-procedural Change Sequence Graph* (CSG) efficiently from the intra-procedural control-flow graphs of the changed methods and the call graph of the changed program. The *intra-procedural Control-Flow Graphs* (CFGs) of the changed methods are traversed to establish the control-flow among the changed basic blocks in the CSG. The *Call Graph* (CG) of the changed program is traversed to establish whether a basic block transitively calls a changed method.

The inter-procedural CSG is computed *more efficiently* than previously in Algorithm 5 on page 76 because it does not require the entire *inter*-procedural CFG for the complete program. Moreover, it disregards potential interaction locations which require additional analysis of program dependence graphs. Using the CSG for test generation to expose CIEs (see Chapter 4), the potential interaction locations computed in Algorithm 5 serve as targets during directed test generation that can provoke change interaction. However, using the CSG for computing the complexity of a source code change, potential interaction locations are not really required as part of the CSG (cf. Definition 14).

Algorithm 7 depicts the CSG construction process. Given two versions of a program, P and P', the algorithm computes the inter-procedural CSG. After determining which methods and basic blocks have changed, the algorithm follows along the control-flow and method calls from every changed basic block

onwards. If another changed basic block is found, an edge is added to the CSG between the original changed basic block and the found one. Then, we establish whether the original changed basic block can be executed as first or last changed basic block and a corresponding edge is added to program entry and exit, respectively.

Algorithm 7 Inter-procedural Change Sequence Graph

Input: Programs P and P'

```
 1: determine changed methods and basic blocks using diff
 2: let CG ← constructCallGraph(P')
 3: let CSG ← {entry, exit}
 4: for each changed method m ∈ CG do
 5:     let CFG ← constructCFG(m)
 6:     add all changed basic blocks from CFG to CSG
 7:     for each changed basic block c ∈ CFG do
 8:         traverseChange(c, CFG, c)
 9:     end for
10: end for
11: connectEntryExit()
12:
13: function traverseChange(curr, CFG, c)
14:     if curr marked as traversed then return
15:     else mark curr as traversed
16:     for each bb that directly follows curr in CFG do
17:         if bb is a changed basic block then
18:             add an edge from c to bb
19:         else
20:             traverseChange(bb, CFG, c)
21:         end if
22:     end for
23:     for each changed m' that curr may call in CG do
24:         let CFG' ← constructCFG(m')
25:         traverseChange(CFG'.first, CFG', c)
26:     end for
27: end function
```

Output: Inter-procedural CSG

In more detail, Algorithm 7 works as follows. First, a syntactic differencing-tool, such as the Unix `diff`-tool, determines the syntactic differences between both program versions (line 1). These differences are used subsequently to determine in the changed version those basic blocks and methods that have changed. Then, the call graph is constructed for the changed program and the CSG initialized with *entry* and *exit* vertices (lines 2-3). After this initialization, the algorithm computes the intra-procedural CFG for each changed method m, adds the changed basic blocks from the CFG into the CSG, and starts traversing the control-flow recursively from each changed basic block c onwards (lines 4-

10). Since the method TRAVERSECHANGE is a recursive traversal algorithm, we mark the *visited* vertices as such (lines 14-15). If any basic block *bb* transitively following *c* is changed, then add an edge from *c* to *bb* (lines 16-21). If *c* or any transitively following basic block, transitively calls a changed method m', continue traversal from the $first$ basic block in the CFG' of m' (lines 23-26). Finally, the method CONNECTENTRYEXIT computes the edges from the *entry*-vertex to any changed basic block that can be executed first, that is before some other basic block is executed, and the edges to the *exit*-vertex from any changed basic block that can be executed last, that is after any other changed basic block is executed (cf. Def. 14). A complete implementation is discussed in Sec. 5.4.3.

Subject	**Size** in kLoC	**Maturity** 1st commit	**#Commits** total (last year)	**#Tests**
`Coreutils`	83.1	Oct. 1992	27,807 (290)	4772
`Findutils`	18.0	Feb. 1996	2,031 (43)	1054
`Grep`	9.4	Nov. 1989	1,307 (31)	1582
`Make`	35.3	Apr. 1988	2,288 (134)	528

Subject	**#Bug Reports marked fixed**	**Extract. Period** recent 1k commits	**#RErrors extracted**
`Coreutils`	832	08.05.11 – 06.10.13	**22**
`Findutils`	312	01.08.05 – 26.10.13	**15**
`Grep`	66	25.09.01 – 26.10.13	**15**
`Make`	305	01.03.96 – 24.11.13	**18**

Figure 5.4: Subjects of COREBENCH

5.4 Empirical Study

5.4.1 Objects of Empirical Analysis

CoREBench: Complex Regression Errors

COREBENCH is a collection of 70 regression errors that we systematically extracted from the code repositories and bug reports of four open-source software projects: `Make`, `Grep`, `Findutils`, and `Coreutils` (see Fig. 5.4).

We chose these projects because they are well-specified, well-tested, well-maintained, and widely-used open source programs with standardized program interfaces. The version history and all bug reports can be publicly accessed on the GNU homepage.[4] The program interfaces and parameters were specified in POSIX as IEEE standard in 1988 [145].

[4] `http://savannah.gnu.org` and `http://debbugs.gnu.org`

We built the corpus by (1) identifying a regression-fixing commit in the 1,000 most recent revisions and a test that passes after but fails before the fix, and (2) the regression-introducing commit, such that the same test passes before and fails after the commit. Regression errors which could not be reproduced using a test case are not reported. This was the case for some system- or concurrency-related bugs.

To identify a *regression-fixing commit* ($\overset{Fix}{\rightarrow}$), we parsed the commit messages of the 1,000 most recent commits and a file which highlights recent new features and fixes for keywords, such as "regression", "introduced", and "broken". Except for `Make`, the file and commit messages are sufficiently detailed and may even reference the error-introducing commit. For `Make`, we parsed the bug report referenced in the commit messages. Also for `Make`, we removed seven commits in which the regression fix was tangled[5] with other fixes. Computing the error complexity based on tangled fixes will give wrong results. For all regression errors we ensure that the commit is solely devoted to fixing exactly one error. The error-witnessing test case was always provided with the bug-fixing commit or the bug report.

To identify the *error-introducing commit* ($\overset{Reg}{\rightarrow}$), we used the error-witnessing test case and a binary search on the complete version history of the subject. The binary search is automated using `git bisect`, which conceptually searches all revisions before the error-fixing commit to determine the exact (error-introducing) commit before which the test case passes ($P_{\checkmark}$) and after which the test case fails ($P_{\text{✗}}$). For `Coreutils`, we add five regression errors that we already identified in Reference [139]. Finally, we determined two commits describing the lifetime and a test case exposing the effects of each regression error:

$$\ldots \rightarrow P_{\checkmark} \overset{Reg}{\rightarrow} P_{\text{✗}} \rightarrow \ldots \rightarrow P_{\text{✗}} \overset{Fix}{\rightarrow} P_{\checkmark} \rightarrow \ldots$$

Using this approach, we have identified and validated 70 regression errors (incl. six segmentation faults) that were introduced by 57 different commits. From the time an error was introduced to the time the error was fixed, it took on average 1.7 years. Eleven errors were fixed incorrectly. In these cases the error was indeed removed in the fixed version. Yet, up to three new errors were introduced that required further fixes. About one third of the errors were introduced by changes not to the program's behavior but to non-functional properties such as performance, memory consumption, or APIs. In some cases one error would *supercede* another error such that the superceded was not observable for the duration that the superceding remained unfixed.[6]

[5] See Reference [146].

[6] For instance, `find.66c536bb` supercedes `find.dbcb10e9`.

Base Line: SIR and Siemens Suite

The *Subject Infrastructure Repository* (SIR) [122] and the *Siemens Suite* [142] are arguably the most popular error benchmarks. For every correct program version $P_{\checkmark}$, there are several faulty versions $P_{\text{✗}}$. One may evaluate regression testing and debugging techniques by considering:

$$P_{\checkmark} \overset{Reg}{\rightarrow} P_{\text{✗}} \overset{Fix}{\rightarrow} P_{\checkmark}.$$

The popularity may be due to the provision of test oracles, standardized program interfaces, a large number of test cases, and a uniform format for the materials provided. Program input and output are clearly defined. Each subject consists of a "golden version" as test oracle and several erroneous versions with one fault each. Measuring popularity by the number of citations: In the five years preceding this dissertation, the publications associated with the SIR [122] and Siemens Suite [142] have been cited almost six hundred times.

Figure 5.5 shows the characteristics of the subjects in both benchmarks. The number of tests was derived from the file `universe` while the number of regression errors was derived from `Fault_Seeds.h` that accompanies each subject.

	Subject	**Size** in kLoC	**#Tests**	**#Regression Errors**
Siemens Suite	tcas	0.2	1,608	41
	totinfo	0.6	1,052	23
	printtokens	0.7	4,130	7
	printtokens2	0.6	4,115	10
	replace	0.6	5,542	32
	schedule	0.4	2,650	9
	schedule2	0.4	2,710	10
SIR (C Subjects)	space	6.2	13,585	38
	bash	59.8	1,200	32
	flex	10.5	628	81
	grep	10.1	625	57
	gzip	5.7	214	59
	make	35.5	795	35
	sed	14.4	370	32
	vim	122.2	974	22

Figure 5.5: Subjects of Siemens Suite and SIR

Unfortunately, in both benchmarks almost all errors were created by manual fault seeding[7]. We claim that fault seeding introduces a *bias towards less complex errors*. Our novel measure of error complexity, for the first time, allows us to assess the substance and extent of this bias.

[7]Except for `space`, all errors are manually generated.

5.4.2 Variables and Measures

Our experiment manipulated two independent variables (IV):

- **IV1 Genuineness**: There are two categorical factors of genuineness. *Seeded regression errors* result from faults that were manually seeded. *Actual regression errors* appear in typical evolving software projects.
- **IV2 Regression Cause**: We consider two categorical factors of regression cause. *Change Interaction Errors* (CIEs) can be observed only if a certain sequence of changes is exercised (cf. Chapter 4). *All other errors* (Non-CIEs) are regression errors that are not CIEs.

In our experiment, we measured 3 dependent variables (DV):

- **DV1 Error Complexity**: We consider two measures of error complexity which is defined w.r.t. the error-fixing commit. The *Cyclomatic Change Complexity* (CyCC) is described in Section 5.2.2. The *Changed Lines of Code* (CLoC) corresponds to the number of executable source code lines that were changed. Both are measured for the version just before the error is fixed.
- **DV2 Error Life Span**: We measure the error life span as the number of days between the commit introducing and the commit fixing the error.
- **DV3 Error-Introducing-Commit Complexity**: We measure the error-introducing-commit complexity as CyCC of the commit introducing the error.

5.4.3 Experimental Design

Measuring Error Complexity for CoREBench

To investigate the complexity of actual regression errors, we analyse their actual fixes. But why should the actual fix be that "least complex, correct" fix describing the error complexity (see Def. 17)? In fact, for each error there can be innumerable fixes and not every fix is *correct* such that not only the observed error is fixed but also no new errors are introduced and *least complex* such that no other correct fix is of less complexity.

In practice, we neither have all possible fixes nor do we have all possible test cases that observe that the error (and only the error) is really fixed. Instead, for the analysis of CoREBench we put forward the following hypothesis:

Competent Repair Hypothesis.

Software developers write fixes with a complexity as low as possible and that are close to being correct.

First, the Competent Repair Hypothesis (CRH) states that developers write fixes that are as *simple as possible*. For several errors in CoREBench we found two fixes – the second fixed the error "more efficiently" or repaired "the root cause" of the error even though the first fix was already a *correct* one.[8] Complex fixes are often accompanied by very elaborate explanations why such complex changes were necessary to fix the error.

Then, the CRH states that developers write fixes that are *close to being correct*. Indeed, the fixes for eleven of seventy errors in CoREBench were incorrect such that the repair of one introduced a new error. However, in general we believe that the programmer is likely to fix the error correctly. If this was not the case, we would register an exponential increase of bug reports. This hypothesis is an instance of the Competent Programmer Hypothesis [147] which states that developers "create *programs* that are close to being correct".

Infrastructure and Implementation

We implemented Algorithm 7 based on the C Intermediate Language (`CIL`) program analysis framework [148] and the Unix `diff` tool to compute the Cyclomatic Change Complexity (CyCC) and the executable Changed Lines of Code (CLoC) of a code commit as the two measures of DV1. Both, tool implementation and CoREBench can be downloaded at
`http://www.comp.nus.edu.sg/~release/corebench`.

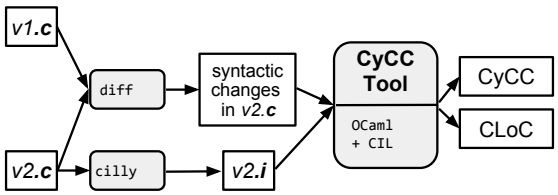


Figure 5.6: CyCC Tool Implementation

As depicted in Figure 5.6, the implementation works as follows. First, the changed version ($v2.c$) is compiled into an intermediate file ($v2.i$) using `cilly`. Then, our script uses the `diff` tool to determine the lines of code that have syntactically changed in $v2.c$. Note that *CLoC* is the number of *executable* changed lines of code while the syntactic changes can also comprise comments. If the program version history is maintained remotely and the changed version is available on the local machine, our script uses the previous verison ($v1.c$) from the repository. Otherwise, its location must be provided to compute the difference.

[8] See commit message of `find.b445af98`

Next, `CIL` can compute the call graph and intra-procedural control-flow graphs (CFG) for the changed program. Using the output of the `diff`-tool, we find the changed methods in the call graph and the changed basic blocks in the CFGs of the changed methods. Note that `diff` detects any line of a multi-line statement that is changed while `CIL` only maintains the first of the potentially multiple lines of a statement. We address this issue for the most common multi-line statement (`if`-conditions) but not for others. Furthermore, top-level variable and method declarations (e.g., `int x;`) are not available in the `CIL` CFGs and macros are readily expanded. Thus, modifications of these program elements, as well as deleted basic blocks, are not reflected in the `CIL`-CFGs and the inter-procedural CSG, respectively.

Once the change sequence graph is synthesized for a source code commit, our implementation computes the CLoC and CyCC according to Definition 15. Note that during our experiments, we ignore errors and code commits that yield "empty" CSGs. For CoREBench, we report the results for all 70 regression errors. However, for SIR and the Siemens Suite, several changes were *only* to variable or method declarations (e.g., change of type) or C macros. While these were ignored, we report the results for the remaining 259 regression errors in SIR and 108 regression errors in Siemens.

The experiments were run on a Linux machine with Intel Core2 Quad CPU at 2.83GHz and 4GB of main memory. On average, it took *less than 1 second* to compute the complexity of an error.

5.4.4 Threats to Validity

Construct validity refers to the degree to which a test measures what it claims, or purports, to be measuring. Three threats to construct validity are the empirical reliability of the competent repair hypothesis, the reliability of CyCC as good measure of error complexity, and the correctness of the implementation of the measure into the `CyCC` tool.

(i) The Competent Repair Hypothesis (CRH) links that theoretical least complex, correct fix specified in Definition 17 to the actual fix of the errors in CoREBench (see Sec. 5.4.3). Assuming the CRH, we measure the complexity of actual regression errors based on the actual fixes of these errors. If the CRH does generally not hold, the actual error complexity may be different from the measured error complexity.

(ii) The CyCC metric may not be a good measure of the complexity of a fix and thus of error complexity. However, we note that Definition 15 of CyCC is inspired by an existing measure of *software* complexity [141]

which itself inspired Definition 13 of change complexity. We study the relationship to another measure of change complexity (see Sec. 5.5).

(iii) The `CyCC` tool may be incorrectly implemented. For instance, some changed elements, like deleted basic blocks, are not represented in the computed CSG from which the CyCC is computed. However, all results are computed using the same tool, subjecting each (compared) measurement to the same potential bias. Furthermore, we make available the source code of the implementation for inspection.

External validity refers to the extent to which the results of a study can be generalized to other objects which are not included in the study. One threat to the external validity is the representativeness of the the chosen objects of empirical analysis. Indeed, our objects are well-maintained, open-source C software projects containing regression errors typical for such projects. However, for instance regression errors in projects written in other languages, like Java, or in commercially developed software may be of different kind and complexity. Hence, the results and conclusion are to be interpreted in this context.

Internal validity refers to the degree to which the independent variable causes the changes seen in the dependent variable being examined within the study. While it is clear that (IV1) the actual regression errors are not seeded and vice versa, it may be that (IV2) regression errors classified as change interaction errors are not actually change interaction errors. However, for each regression error, we attempted to determine the specific sequence of changed statements that need to be exercised to expose the error. In the results we note which errors could thus not be classified.

5.5 Data and Analysis

We investigate the nature of complex regression errors. In our main research hypothesis, we claim that the process of creating errors using manual fault seeding introduces a *bias towards less complex errors*. Formally, we submit a null hypothesis which needs to be rejected in order to empirically prove this claim. We also find out whether actual, more complex regression errors have a longer life span and whether complex errors are introduced by complex commits.

Furthermore, we investigate another measure of change complexity – the number of Changed Lines of Code (CLoC). While we cannot directly compare both measures, we find out whether our Cyclomatic Change Complexity (CyCC) and CLoC agree on the *ranking* of two-hundred commits in terms of their complexity. If so, CLoC and CyCC may be used interchangeably to assess the complexity of a commit.

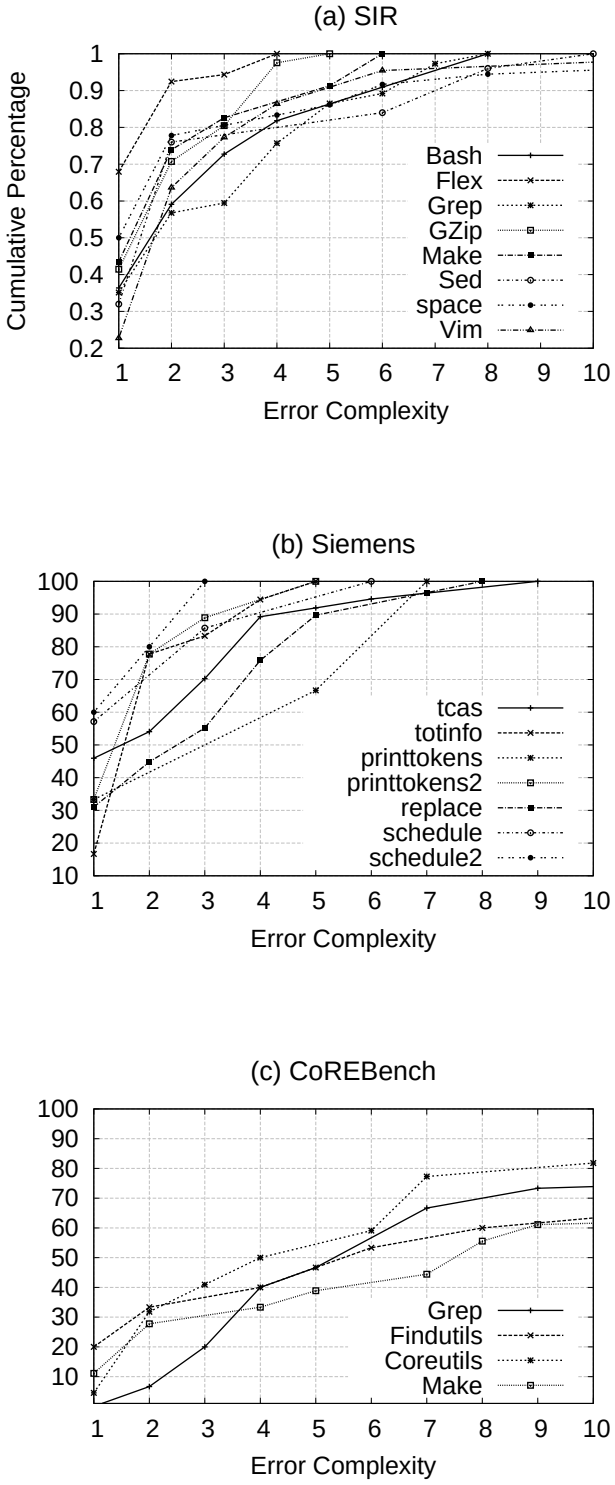


Figure 5.7: Cumulative distribution of error complexity for all subjects in each benchmark

Lastly, we use the actual regression errors to study the prevalence, complexity, and life span of an interesting class of regression errors – Change Interaction Errors (CIEs; cf.[139]). We classify a regression error E as CIE if a sequence of at least two changed statements must be executed in order to expose E while "skipping" one of the changed statements does not expose E. Conservatively, we also require that each change in the sequence *can* potentially be skipped.

5.5.1 Research Questions and Null Hypothesis

In statistical inference, the *null hypothesis*, H_0, states there is no relationship between two measured phenomena. The null hypothesis can be *rejected* based on observed data of a scientific experiment with the conclusion that there is very likely a relationship. The null hypothesis can never be accepted as more data may still reveal a relationship.

To test H_0, we measure either a *difference* or the *strength of the relationship*. In the first case, we subtract the mean of one from the mean of the other dataset. In the latter case, we measure Spearman's rank correlation coefficient [149] which is more robust for non-normal distributions than the common Pearson's product moment correlation. If we fail to reject H_0 with a very low correlation coefficient, we can still conclude that if a relationship exists, it is very weak.

- H_0^a : There is no difference between the complexity of *seeded* and *real regression errors*.
- H_0^b : There is no relationship between the *complexity* and *life span* of a regression error.
- H_0^c : There is no relationship between the complexity of the *error* and the *commit introducing the error*.

Furthermore, we want to answer these research questions:

- **RQ1** Can the *number of Changed Lines of Code* (CLoC) and the *Cyclomatic Change Complexity* (CyCC) be used interchangeably?
- **RQ2** What is the complexity, prevalence, and life span of *Change Interaction Errors*?

H_0^a : Seeded vs. Actual Errors (IV1, DV1)

We compare the error complexity (as CyCC) of the seeded regression errors in the Siemens Suite and SIR with that of the actual regression errors in CoREBench to study the effects of IV1 on DV1 and test H_0^a. For SIR and the

Siemens Suite, we measure the complexity of the errors by considering the non-faulty versions as the fix for the error in the faulty versions. For CoREBench, we measure the complexity of the errors by analyzing the complexity of the regression-fixing commits and assume the Competent Repair Hypothesis. Also, for CoREBench we choose the regression errors such that every regression-fixing commit is designated to fixing exactly one error only.

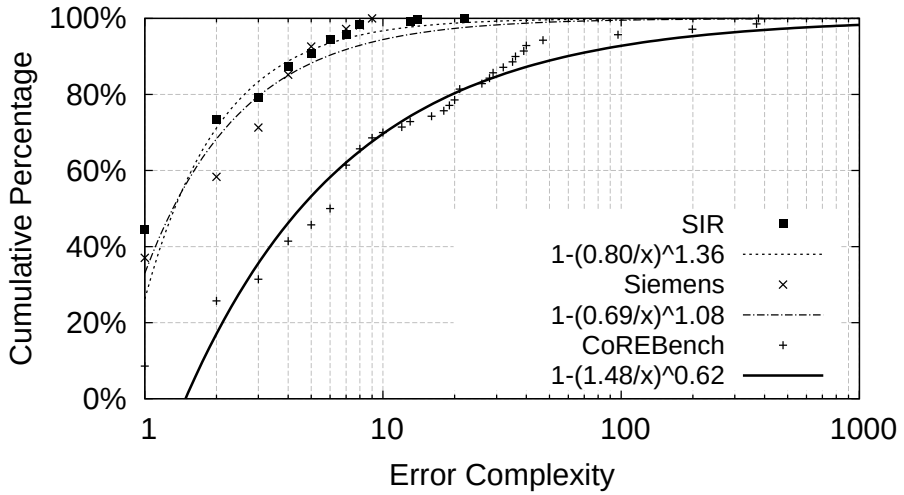


Figure 5.8: Cumulative distribution of error complexity for seeded errors (SIR and Siemens) vs. actual errors (CoREBench)

We reject H_0^a *and conclude: seeded regression errors are significantly less complex than actual regression errors.* The mean error complexity differs by 21.9 for SIR and 21.7 for the Siemens Suite. Fitting the data to a power-law distribution, we compute the cumulative distribution functions shown in Figure 5.8. The complexity distributions for each subject and benchmark are shown in Figure 5.7.

Among the seeded errors, *simple errors* (complexity one) occur five times more often than among the actual errors. Specifically, 42% of the seeded errors are simple while only 8% of the actual errors are. Simple errors are characterized by a localized fault and can often be fixed by changing just one statement. In contrast to actual errors, the complexity of the seeded errors barely exceeds 10. Less than 1% of the seeded errors have a complexity of more than 10 compared to 30% of the actual errors. This means, that actual errors are generally more complex than the errors created through manual fault injection. The most complex error in CoREBench is twenty times more complex than the most complex error in the SIR and the Siemens Suite.

H_0^b : Life Span vs. Complexity (DV1, DV2)

We compare the life span and complexity of actual regression errors to study the correlation between DV1 and DV2 and test H_0^b. Every commit has a timestamp, so we can compute the life span of an error by subtracting the timestamp of the error-introducing from that of the corresponding error-fixing commit. We measure the complexity using CyCC and depict the results in Figure 5.9.

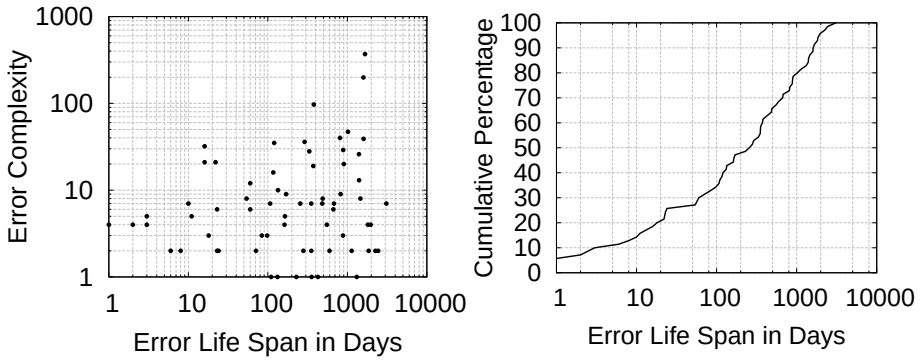


Figure 5.9: Correlation of error life span vs. complexity (left), cumulative distribution of life span (right)

We cannot reject H_0^b *and conclude that if a relationship between the life span and complexity of an error exists, then it is very weak.* We compute a Spearman's rank correlation coefficient of $\rho = 0.0675$ with a two-sided p-value=0.5790. In other words, even simple errors that are *"easy" to fix* can take a very *long time to fix*. Vice versa, even complex errors that are difficult to fix can be fixed on the same day as the error is introduced.

Independent of error complexity, error life span follows a power-law distribution. Once introduced, 12% of the regression errors are fixed within a week while half of them stay undetected and uncorrected for more than 9 months up to 8.5 years. While there is a large number of errors with a small life span, there is a small number of errors with very large life span.

H_0^c : Introducing vs. Fixing Errors (DV1, DV3)

For each actual regression error, we compare the CyCC of the commit introducing and the commit fixing the error to study the correlation between DV1 and DV3 and test H_0^c. The results are presented in Figure 5.10. On the left, we show for each regression error the complexity of the commit introducing the error versus the complexity of the commit fixing the error. On the right, we show the cumulative distribution of error-introducing and error-fixing commits independently.

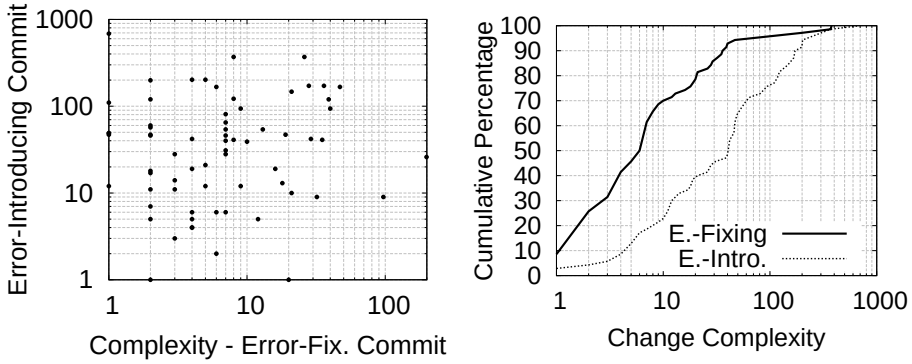


Figure 5.10: Correlation (left) and cumulative distribution (right) of the complexity of the two commits introducing and fixing an error.

We cannot reject H_0^c *and conclude that if there exists a relationship between the complexity of an error and the complexity of the commit which introduces the error, then it is very weak.* We compute a Spearman's rank correlation coefficient of $\rho = 0.1656$ with a two-sided p-value=0.1705. In other words, even complex errors can be introduced by simple changes and vice versa. One interpretation is that sometimes the root cause of some complex regression errors is already dormant in the program and only "unmasked" in the changed code. Then, we should consider these changes as *the trigger instead of the root cause* of an observed error. Another interpretation is that the error itself evolves during its life span due to many other changes to the program. Then, *the complexity of errors may change during evolution.*

On average, error-introducing commits are more complex when compared to error-fixing commits (see Fig. 5.10 – right).

RQ.1 Changed Lines of Code as Proxy Measure

For 200 random code commits[9], we measure the CyCC and Changed Lines of Code (CLoC), to study the concordance and correlation of two measures of DV1 (Error Complexity). *Concordance* describes the degree to which both measures agree on the complexity of a set of changes and is measured using Cohen's kappa [150]. Full agreement ($\kappa = 1$) means that CyCC rates a set of changes C_1 more complex than another set of changes C_2 if and only if CLoC rates C_1 more complex than C_2. In contrast, *correlation* describes the strength of the relationship and is measured using Spearman's ρ. Strong correlation ($\rho = 1$) means that if CyCC is large than CLoC is also likely to be large and vice versa.

[9] We chose the 50 most recent code commits in each of the projects `Coreutils`, `Findutils`, `Grep`, and `Make`.

The results are presented in Figure 5.11. The Bland-Altman plot [151] on the left allows us to compare the *differences* between the measurements with both measures of complexity for each commit. The mean ($\bar{x}$) of these differences is called *bias* and the reference interval ($\bar{x} \pm 1.96 \times$standard deviation) is called *limits of agreement*. If the measures tend to agree, the differences will be plotted near zero. As CLoC and CyCC are not directly comparable and the power-law distribution generates strong outliers, we compare the *ranks* instead of the measurement values. The rank of measurement lies between one and the number of measurements and is greater than the rank of another measurement if and only if the measurement value is greater than that of the other measurement. The plot on the right depicts the (value) correlation of both measures on a logarithmic scale.

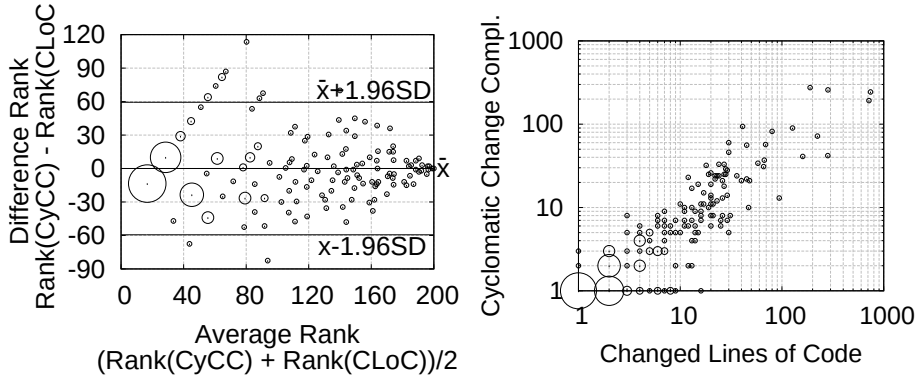


Figure 5.11: Bland-Altman plot of measurement ranks (left) and correlation (right) of CLoC vs. CyCC.

Moderate Agreement. *The Changed Lines of Code and Cyclomatic Change Complexity cannot be used interchangeably to assess the complexity of a set of changes.* The limits of agreement, shown in the Bland-Altman plot, are far apart (±59.4 out of 200 ranks). We also compute a Cohen's kappa of $\kappa = 0.014$ for the measurement ranks ($\kappa = 0.151$ for the values) which indicates only moderate agreement between both measures on the complexity of a code commit.

Two measures that are designed to measure the same property (here, change complexity) may not agree but should have a good correlation. Indeed, we compute Spearman's correlation $\rho = 0.86$ with a two-sided p-value < 0.0001. So, as the CLoC increases, the CyCC increases and vice versa.

RQ.2 Complexity, Life Span, and Prevalence of Change Interaction Errors (IV2, DV1, DV2)

We compare the error complexity (as CyCC) and life span of Change Interaction Errors (CIEs) with the error complexity and life span of actual regression errors that are not Change Interaction errors (Non-CIE) to study the effects of IV2 on DV1 and DV2. We also measure the prevalence of CIEs among actual regression errors.

The results are presented in Figure 5.12. In the table, we show the classification of actual regression errors into CIE, Non-CIE, and Unclassified. For the latter, the regression cause could not be identified. On the left, we show the cumulative distribution of the complexity of CIEs versus Non-CIEs cropped at an error complexity of 50. On the right side, we show the cumulative distribution of the life span of CIE versus Non-CIEs on a logarithmic scale.

	CIE	**Non-CIE**	**Unclassified**
`Coreutils`	7	13	2
`Findutils`	5	7	3
`Grep`	5	7	3
`Make`	5	10	3
Total	22	37	11

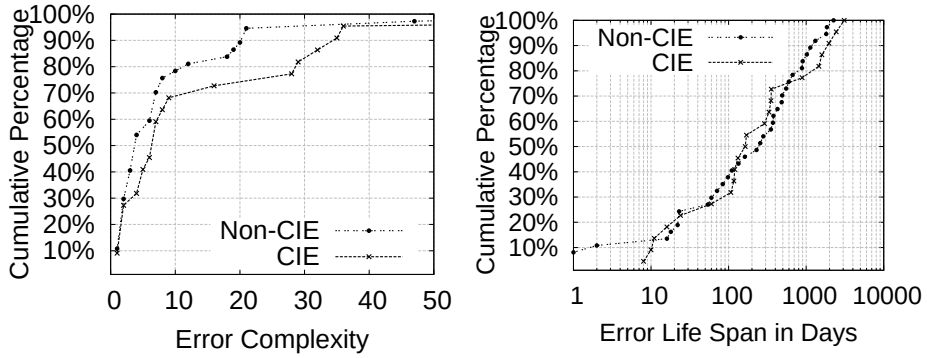


Figure 5.12: Prevalence (top), complexity (left), and life span (right) of Change Interaction Errors

Error Complexity. *CIEs are consistently more complex than Non-CIEs.* The mean complexity of CIEs (20.1) differs from that of Non-CIEs (9.9) by 10.2. On average 10% more CIEs exceed any given complexity than Non-CIEs. For example, while about 32% of the CIEs exceed a complexity of 10, only 22% of the Non-CIEs exceed the same complexity. This means CIEs are *"more difficult to fix"* than other types of regression errors.

Error Life Span. *CIEs and Non-CIEs have a similar life span.* Indeed, the mean life span of CIEs (623 days) differs from that of Non-CIEs (463 days) by 160 days. However, on average only 1% more CIEs exceed any given life span than Non-CIEs. From the chart (Fig. 5.12–right) it seems evident that there is no significant difference between the life span of CIEs and that of Non-CIEs. This means CIEs are manually *"as difficult to find"* as other types of regression errors.

Prevalence. *Change interaction errors are prevalent.*
In fact, 22 of 59 classified actual regression errors can be classified as CIEs. This means that the existence of change interaction errors as a particular type of regression errors must be considered during the testing and debugging of evolving open source C programs. The prevalence and peculiarity of change interaction errors suggests that *CIEs should not be disregarded* during the empirical evaluation of techniques and methodologies in the scientific research of regression testing, debugging, and program repair.

In summary, compared to any other type of regression errors, CIEs are more difficult to expose automatically [139] while it takes the same time to encounter them manually (cf. error life span). Once discovered, CIEs are "more difficult to fix" (cf. error complexity). Since CIEs are prevalent in open-source C programs, they form an important class of regression errors that can be studied in CoREBench.

5.6 Related Work

We first discuss investigations into the relationship of error complexity and detectability, continue with work related to quantifying error complexity, and conclude with an overview of related work on the construction and public provisioning of a benchmark suite with actual regression errors.

Offutt [140] asserts a relationship between the detectability and complexity of software errors. He defines a *simple fault* as one "that can be fixed by making a single change to a source statement" while a *complex fault* is one that can thus not be fixed. In his *coupling effect* hypothesis he conjectures that a "test dataset that detects all simple faults in a program will detect a high percentage of the complex faults" which holds if and only if the detectability[10] of simple errors is somewhat similar to the detectability of complex errors. In the present work, we have extended Offutt's definition of error complexity to be ordinal rather than nominal.

[10]The detectability of an error is determined by the proportion of input exposing the error (see Sec. 5.2.2).

Andrews et al. [152, 153] confirm that the detectability of simple errors resulting from auto-generated faults (i.e., mutants) is similar to the detectability of actual (complex) errors and conclude that the mutation-adequacy of a test suite is a good indicator of its fault-detection capability. Namin et al. [154] caution that this insight is highly sensitive to external threats mentioning several influential factors that must be accounted for. In the present chapter we have investigated not the detectability but the *complexity* of regression errors and found that the complexity of regression errors resulting from seeded faults is different from that of actual regression errors. This raises concerns for the validity of studies based on seeded errors.

While it is intuitively clear that some errors are simple and others certainly more complex, we are not aware of any previous attempt to *quantify* error complexity. However, there has been a great effort to understand how to quantify software complexity [155]. Some established measures of software complexity are McCabe's cyclomatic complexity [141], Henry and Kafura's information flow complexity [156], and Chidamber and Kemerer's object-oriented complexity [157]. To quantify error complexity, we introduce and compare two measures – the cyclomatic change complexity (CyCC) and the number of changed lines of code (CLoC).

A popular technique to extract *actual regression errors* from software repositories is the SZZ-algorithm [158, 159]. First, SZZ identifies the error-fixing commit by parsing the commit messages for relevant keywords. Then, SZZ identifies the error-introducing commit by *blaming* the changed lines in the error-fixing commit. Blaming or annotating is a function of the repository to determine the commits that modified or added any given line of code. Fundamentally, the SZZ-algorithm assumes that the lines changed in the fix contain the fault location and determines which commit changed these lines previously to introduce the error. However, we find that the changed lines in the error-fixing and error-introducing commits in CoREBench do not even overlap for one in every three regression errors.

Three benchmarks that contain *actual program errors* are iBugs [160], BugBench [161], and Marmoset [162]. iBugs consists of a large number of real bug fixes in the version history of two Java projects, AspectJ and Rhino. For some bug fixes, the benchmark also maintains those test cases that were submitted with the fix. BugBench consists of mostly memory-related errors while Marmoset contains errors extracted from student projects and may not contain a representative sample of actual program errors. In contrast to these, our CoREBench allows us to study *regression* testing and regression debugging techniques as well as the evolution of software errors over several program versions for up to eight years from error-introduction to fix.

5.7 Chapter Summary

The research on and development of automated techniques to expose, locate root-causes of, and repair *regression errors* requires an understanding of the inherent nature of such errors. In order to develop automated regression testing, debugging, and repair techniques, we need to be aware of the underlying, general properties of regression errors.

In this chapter, we advertise the study of regression errors with a varying degree of *complexity* and propose the subjects in CoREBench, as a collection of actual regression errors, for such controlled studies. We have analyzed the two most popular benchmarks, the Siemens Suite and SIR, which contain regression errors with a varying degree of *detectability* and found that these errors are often *simple* and generally significantly *less complex* than actual regression errors. In other words, their *fixes* were required to be *less substantial*.

Our novel measure of error complexity enables research and development of regression testing, debugging, and repair techniques *that account for a varying degree of complexity*. We may ask more refined research questions, such as:

- **What is the root-cause of a complex error?** If an error requires a substantial fix, can we assume that there is just one faulty statement causing the error? Are faults of complex errors localizable [163]? The answers may have implications for the performance of (statistical) debugging techniques.

- **Test suite adequacy to expose complex errors?** Some widely used metrics of test suite adequacy, such as statement or branch coverage, are based on the implicit assumption that errors are often simple, i.e., that the fault is localizable within some branch or statement which is covered. Now we may be able to investigate the effectiveness of coverage-adequate test suites w.r.t. a varying degree of error complexity and may develop more sophisticated adequacy-criteria that account for complex errors. Moreover, for the study of the relationship between simple and complex errors (e.g., see coupling effect [140]), we can take error complexity as an *ordinal* rather than a *dichotomous* measure.

- **How do we repair complex errors?** By definition, the fix of complex errors is more substantial than for simple errors. The research community has made significant progress understanding the automated repair of (simple) localizable errors [164, 165]. Now we may be able to evaluate the *efficiency* of such repair techniques w.r.t. a varying complexity of the repaired errors.

The artifact evaluation committee of ISSTA 2014 has found CoREBench and the CyCC tool to exceed expectations. We hope that our novel error complexity metric and the many actual regression errors in CoREBench spur a multitude of studies of regression testing, debugging, and repair techniques and of those assumptions underlying these techniques so as to better understand the nature of complex regression errors.

Chapter 6

Conclusion

6.1 Summary and Contributions

We answer the scientific questions that are most relevant with respect to our thesis: "*A complex source code change can only be checked effectively by stressing the interaction among its constituent changes*", as given in the following.

1. **How can we determine the semantic impact of a complex change?** We have presented a concrete strategy to partition the input space into disjoint, homogeneous subdomains, such that either every input in the same partition produces different output, or every input in the same partition produces the same output when executed on both program versions. We note that only input producing different output can expose software regression. Encoding these differential partitions as symbolic formula over the program inputs, we have presented an algorithm that systematically explores these partitions by negating the constituent branch conditions. We have shown the *soundness* of deriving differential partitions and the *exhaustiveness* of the algorithm to explore all such differential partitions.

2. **Just how complex is a complex change**? In order to study complex source code changes, we wanted to *quantify* the complexity of a source code change formally and uniformly. We have defined the Cyclomatic Change Complexity (CyCC) that directly measures the number of "distinct" sequences of changed statements from program entry to exit. In general, complexity is a property of a system with many parts where those parts interact with each other in multiple ways. In the context of software evolution, a complex source code change consists of many constituent changes that may interact with each other.

3. **Can interaction in a complex change result in regression errors**? Yes, if many constituent changes can interact with each other, there must be a class of regression errors that is only observable when a sequence of changed statements is executed but not if any of the changes in this sequence is skipped. We have defined this class of errors as Change Interaction Errors (CIEs) and found that about *every one in three regression errors* in our benchmark, CoREBench, consisting of seventy genuine regression errors, *are due to change interaction*.

4. **What is the nature of change interaction errors**? To learn about the nature of CIEs, we have generated test cases exercising all the different change sequences for version pairs in several open source C programs (GNU Coreutils) and found that CIEs are "subtle" as only one in five generated test inputs exercises a sequence critical to exposing a CIE. Yet, such input was also an order of magnitude more likely to expose an error. In contrast, tests generated to stress one change at a time exposed only half of the CIEs.

5. **Just how complex is a complex error**? In order to study the complexity of regression errors in general and of CIEs in particular, we have defined an error complexity metric. We wanted to find out what distinguishes a "simple error" from a more "complex error" and assign a value to this complexity. So, we have defined the complexity of an error w.r.t. the changes that are required to repair the error (and only the error). The complexity of these error-repairing changes is measured using the CyCC. Intuitively, *simple errors* are characterized by a localized fault that may be repaired by changing one statement while more *complex errors* can be repaired only by more substantial changes at different points in the program.

   For the seventy regression errors in CoREBench, we have found that CIEs are generally more complex than other regression errors that cannot be classified as CIEs. In other words, the repair of a CIE is "more difficult" because it involves more substantial changes.

6. **How can we expose change interaction errors**? We have discussed and implemented a change-sequence-guided test generation technique that systematically explores the complex dependency structure among the composite changes to generate test cases that exercise many different sequences of changed statements. The tool also effectively exposed five previously unknown errors in the GNU Coreutils and found interest from the developers who have maintained the Coreutils for more than a decade.

The following are the main technical contributions of this dissertation.

- **Efficient Regression Verification**. We have presented a technique that can effectively show the absence of regression for all input (i.e., regression verification) and improves the efficiency of regression verification by allowing gradual and partial verification using dependency analysis and symbolic execution. To allow *gradual* regression verification, differential partitions are explored gradually and systematically until the exploration is user-interrupted or the complete input space has been explored. Input that does not reveal a difference cannot expose software regression. To allow *partial* regression verification, the partition-based verification can be interrupted at anytime with the guarantee of the absence of regression for the explored input space. Upon allowing the continued exploration even of difference-revealing partitions, the developer may look at the output differences and verify the correctness of the observed semantic changes.

- **Effective Regression Test Generation**. We have discussed a technique that can efficiently show the absence of regression for some input (i.e., regression test generation) and improves the effectiveness of regression test generation by additionally considering the interaction among several syntactic changes. Using directed symbolic execution, test input is generated that exercises the complex dependency structure among the composite changes. This change dependency is modelled by the Change Sequence Graph (CSG) which captures the control-flow among the composite changes and potential interaction locations of these changes. The tool is available at: `http://www.comp.nus.edu.sg/~mboehme/otter_graph.zip`.

- **Change Complexity Measure and Error Complexity Metric**. We have defined a measure of change complexity that can be compared to other measures of program complexity and a metric of error complexity w.r.t. the changes necessary to repair the error (and only the error). We have implemented the `CyCC` tool which measures the complexity of a GIT commit in under a second, on average, and made it available at: `http://www.comp.nus.edu.sg/~mboehme/corebench/cycc.tar.gz`

- **Regression Error Benchmark**. As there are no established benchmarks containing *genuine* regression errors, we have constructed a benchmark of 70 genuine regression errors, called CoREBench, using a systematic extraction from over four decades of project history and bug reports. For each error, we determined the commit that introduced the error, the commit that fixed it, and a test case that fails throughout the error's lifetime, but passes before and after. Comparing CoREBench to the the two

established benchmarks containing *artificial regression errors*, SIR and Siemens Suite, we found that these are biased containtaining less complex errors and propose CoREBench for the controlled study of regression testing, debugging, and repair techniques. CoREBench is available at `http://www.comp.nus.edu.sg/~mboehme/corebench/corebench.tar.gz`

6.2 Future Work

Reducing Differential to Conventional Program Analysis. Given a complex change, we want to introduce a technique that generates a meta-program which represents the set of program configuration where no, some, or all of the constituent changes are applied. The set of represented configuration includes the program versions before and after the complex change is applied. The meta-program contains *common statements* that are common to all program configurations and represent the common behavior, and a set of *change hooks*. Each change hook is a function pair where one function contains the statements unique to the old version and the corresponding function contains the statements unique to the new version. During (conventional) program analysis, each change hook may be bound to the one member of the corresponding function pair or it may be used as an uninterpreted function.

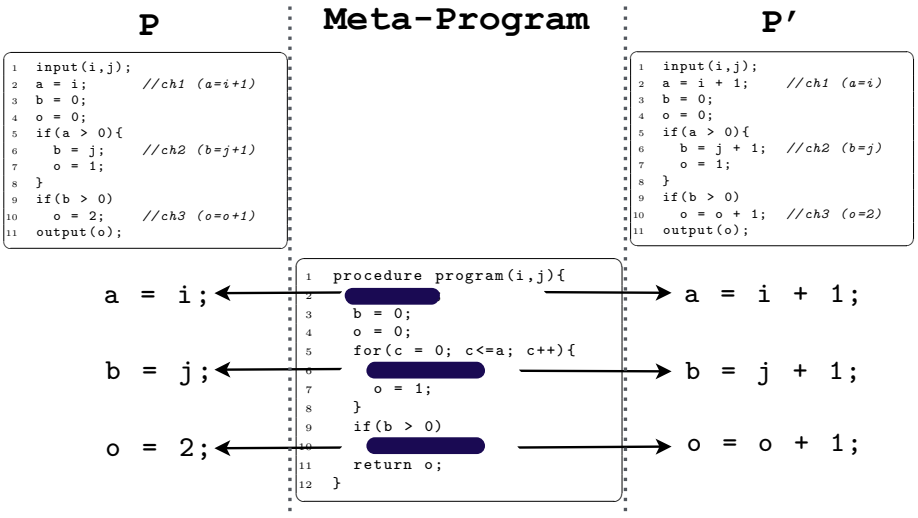


Figure 6.1: Meta-program representing all configurations between two versions

Figure 6.1 shows an example of two program versions and the corresponding meta-program. The shaded code regions represent the change hooks. Depending on the relevant program configuration, a convential program analysis technique may interpret a hook as per the old or as per the new version.

Input Condition	Symbolic Output
$(0 > a(i)) \wedge (0 \leq 0)$	$program(i,j) = 0$
$(0 > a(i)) \wedge (0 > 0)$	$program(i,j) = o(0)$
$(0 \leq a(i) < 1) \wedge (b(j,0) \leq 0)$	$program(i,j) = 1$
$(0 \leq a(i) < 1) \wedge (b(j,0) > 0)$	$program(i,j) = o(1)$
$(1 \leq a(i) < 2) \wedge (b(j,0) \leq 0)$	$program(i,j) = 1$
$(1 \leq a(i) < 2) \wedge (b(j,0) > 0)$	$program(i,j) = o(1)$
$\vdots$	$\vdots$

Figure 6.2: Symbolic output of a meta-program

Compared to other differential program analysis techniques, our technique allows to abstract the changed program behavior during analysis. For instance, one could derive the symbolic output of the meta-program. The symbolic output is not only given in terms of the program input but also in terms of the uninterpreted functions representing the change hooks. These uninterpreted functions can be interpreted according to one of the corresponding function pairs. Figure 6.2 gives the symbolic output of the meta-program in Fig. 6.1. There are three change hooks, $a(i)$, $b(j,b)$, and $o(o)$. The input conditions and the symbolic output are given in terms of these change hooks.

Compared to syntactic differencing, we solve two challenges i) of aligning two corresponging statements and syntactic changes in two versions, and ii) of generating compilable, executable, intermediate program configurations (see Sec. 2.3.4).

The state space of the meta-program can be explored by extended symbolic execution. Intuitively, the state space is extended by one dimension spanning the change configurations. The path exploration proceeds similarly when reaching a conditional statement with two branches. When exercising an uninterpreted function, the symbolic state can be forked, executing the original behavior in one, and the changed behavior in the other symbolic state. Symbolic program summaries, such as in Figure 2.4 on page 11, can contain uninterpreted functions to indicate change impact on the output. Note, while the Differential Symbolic Execution approach [39] summarizes "similar behavior" as uninterpreted functions, we suggest to summarize the changed behavior across two versions.

Exposing Vulnerability Regressions. If we can empirically determine the severity and prevalence of vulnerability regression errors, we want to present an efficient and effective technique to generate input that exposes the vulnerabilty in the changed version. In particular, the problem can be stated as follows: Given an access-regulating code region and a sensitive code region, generate input that can bypass the access-regulating code region to access the sensitive code region when the program is changed. The user login or autho-

rization checks are examples of access-regulating code regions. Unauthorized access to sensible databases, private files, or OS-specific data are examples of vulnerabilities which we seek to witness via test generation.

Appendix A

Theorems – Partition-based Regression Verification

In the following, we postulate the soundness of Algorithm 2 that computes the differential partition for a given test case (cf. Theorem 1) and the exhaustiveness of Algorithm 1 that explores differential partitions (cf. Theorem 2 on page 125). In practice, the absence of regression errors can be guaranteed for all inputs to the same extent as symbolic execution can guarantee the absence of program errors (cf. [67, 68]). Specifically, we assume deterministic program execution. The predicate $s_i \rightsquigarrow r_i$ denotes that r_i is in the relevant slice of s_i. In other words, $s_i \rightsquigarrow r_i$ holds if s_i transitively, dynamically data-, control- or potentially depends on r_i or $s_i = r_i$.

A.1 Soundness

Theorem 1 (*Sound Generalization*)

Given statements C in program P are changed to C' yielding P', every input satisfying the condition computed by Algorithm 2 for input t is in the same differential partition as t.

Informally, the differential behavior of a point in the common input space is soundly generalized to the set of points in the same differential partition. In particular, let Algorithm 2 compute the symbolic condition Φ for a test case t. If t is equivalence-revealing, then every input satisfying Φ is equivalence-revealing. Similarly, if t is difference-revealing, then every input satisfying Φ is difference-revealing. The respective proof is based on the property of relevant slices.

Lemma 1 (*Property of Relevant Slices [19]*)

If two inputs t_0 and t_1 exercise the same relevant slice computed w.r.t. a statement instance s_i, then the variables used in s_i have the same symbolic values for t_0 and t_1.

The property of relevant slices is at the center of the proofs.

Lemma 2 (*Homogenity - Reachability Condition*)

Let c be a statement in program P. Let π_0 and π_1 be the traces for the execution of inputs t_0 and t_1 on P. If t_1 satisfies $reach(c, \pi_0)$, then i) all instances c_i of c executed in π_0 are also executed in π_1 and vice versa, and ii) $reach(c, \pi_0) \leftrightarrow reach(c, \pi_1)$.

Collorary: Given statements C in program P are changed to C' yielding P', if input t does not execute any $c' \in C'$, then every input satisfying $\bigwedge_{c' \in C'} reach(c', \pi(t, P'))$ does neither execute any $c \in C$ in P nor any $c' \in C'$ in P'.

Proof: Assume, t_1 satisfies $reach(c, \pi_0)$. By Definition 5 and this assumption, t_1 exercises the same statement instances that are included in the reachability slice of c in π_0. We prove

- i) every instance c_i of c executed by t_0 is also executed by t_1, and vice versa:
- i.a) If c does not statically control-depend on any statement s, then every instance of c, including c_i, is in all paths, including π_0 and π_1.
- i.b) If c does statically control-depend on a conditional statement s and π_0 contains an instance s_i of s, then every instance c_i of c in π_0 dynamically control-depends on s_i. By Definition 4 and assumption, t_0 and t_1 exercise the relevant slice of s_i. By Lemma 1, the variables used in s_i have the same symbolic values in π_0 and π_1. Thus, if c_i is executed in π_0, it is also in π_1, and vice versa.
- i.c) The case that for every statement s that c statically control-depends on, there exists no instance s_i of s in π_0, is unsatisfiable by the theorems of transitive, static control-dependence. This can easily be shown. Assuming above, then the (non-)execution of s transitively depends on the evaluation of another instance r_i of some statement r in π_0 that is exactly evaluated in the direction that does not favor the execution of s. This is a contradiction because there exists an instance r_i of statement r in π that c transitively control-depends on. Concluding cases i.a), i.b), and i.c), all instances c_i of c executed in π_0 are also executed in π_1, and vice versa.
- ii) The application of Definition 5, Lemma 1, and i) onto $reach(c, \pi_0)$ derives $reach(c, \pi_1)$, and vice versa. Thus, $reach(c, \pi_0)$ is exactly the same as $reach(c, \pi_1)$. ■

Lemma 3 (*Alignment 1*)

Let N_i be computed by Algorithm 3 for the traces $\pi(t, P)$, $\pi(t, P')$ and output statement o. Given alignable output instances (o_i, o'_i) that have the same values in P and P', for every $(n_i, n'_i) \in N_i$ and for every (s_i, s'_i) that satisfies $o_i \rightsquigarrow s_i \rightsquigarrow n_i$ in $\pi(t, P)$ and $o'_i \rightsquigarrow s'_i \rightsquigarrow n'_i$ in $\pi(t, P')$ holds $value(s_i) = value(s'_i)$ and $align(s_i, s'_i)$.

Informally, all elements in the relevant slice of o_i' until n_i' in $\pi(t, P')$ can be aligned and have the same values in both programs.

Proof: We show by induction over every input (s_i, s_i') of procedure PROPALIGN that (s_i, s_i') satisfies $value(s_i) = value(s_i')$ and $align(s_i, s_i')$. Base Case: Initially, (o_i, o_i') is assigned (s_i, s_i'). Clearly, the hypothesis holds. Inductive Step: The current assignment to (s_i, s_i') is (q_i, q_i'). Assume, (q_i, q_i') satisfies $value(q_i) = value(q_i')$ and $align(q_i, q_i')$. The procedure PROPALIGN is called only for the tuple (r_i, r_i') if q_i' directly depends on r_i' and there exists r_i upon which q_i directly depends and satisfies $value(r_i) = value(r_i')$ and $align(r_i, r_i')$ (line 10). Hence, the next assignment to (s_i, s_i') does also satisfy $value(s_i) = value(s_i')$ and $align(s_i, s_i')$. ■

Lemma 4 (*Alignment 2*)

Let statements C in program P be changed to C' yielding P'. Let N_i be computed by Algorithm 3 for the traces $\pi(t, P)$, $\pi(t, P')$ and output statement o. If c_i is an instance of $c \in C$ and $o_i \rightsquigarrow c_i$ in $\pi(t, P)$, then there exists $n_i | (n_i, n_i') \in N_i$ so that $o_i \rightsquigarrow n_i \rightsquigarrow c_i$ in $\pi(t, P)$. Similarly, if c_i' is an instance of $c' \in C'$ and $o_i' \rightsquigarrow c_i'$ in $\pi(t, P')$, then there exists $n_i' | (n_i, n_i') \in N_i$ so that $o_i' \rightsquigarrow n_i' \rightsquigarrow c_i'$ in $\pi(t, P')$.

Informally, for every transitive dynamic dependence of the output onto a change there exists an element in N_i so that the output depends on that element and the element depends on that change.

Proof: Assume, $o_i \rightsquigarrow c_i$ and $o_i' \rightsquigarrow c_i'$. As every $(n_i, n_i') \in N_i$ is previously assigned to some (s_i, s_i') it is sufficient to show by induction over every input (s_i, s_i') of procedure PROPALIGN that $o_i \rightsquigarrow s_i \rightsquigarrow c_i$ and $o_i' \rightsquigarrow s_i' \rightsquigarrow c_i'$. Base Case: Initially, (o_i, o_i') is assigned (s_i, s_i'). Clearly, the hypothesis holds. Inductive Step: The current assignment to (s_i, s_i') is (q_i, q_i'). Assume, $o_i \rightsquigarrow q_i \rightsquigarrow c_i$ and $o_i' \rightsquigarrow q_i' \rightsquigarrow c_i'$. The method PROPALIGN is called only if there exists r_i upon which q_i depends in $\pi(t, P)$ for every r_i', upon which q_i' depends in $\pi(t, P')$, so that $align(r_i, r_i')$, $value(r_i) = value(r_i')$, and r_i' is not c_i'. Thus, for (r_i, r_i') holds $o_i \rightsquigarrow r_i \rightsquigarrow c_i$ and $o_i' \rightsquigarrow r_i' \rightsquigarrow c_i'$ and is the next assignment to (s_i, s_i'). ■

Lemma 5 (*Homogenity - Propagation Condition*)

Given statements C in program P are changed to C' yielding P', if input t_0 computes the same values for output o in P and P', then every input t_1 satisfying $prop(o, \pi(t_0, P), \pi(t_0, P'))$ i) exercises exactly the same instances of $c \in C$ in P and of $c' \in C'$ in P', ii) computes the same values for output o in P and P'.

Proof: Assume, t_0 is equivalence-revealing, t_1 satisfies $prop(o, \pi(t_0, P), \pi(t_0, P'))$, and Algorithm 3 computes N_i for program output statement o and the exe-

cution of t_0 upon P and P'. We prove i) t_0 and t_1 exercise exactly the same instances of $c \in C$ in P and of $c' \in C'$ in P' and ii) t_1 is also equivalence-revealing.

• i) By Definition 6 and assumption t_0 and t_1 satisfy $\bigwedge_{c \in C} reach(c, \pi)$ and $\bigwedge_{c' \in C'} reach(c', \pi')$. By Lemma 2, all instances c_i of every $c \in C$ that are executed in $\pi(t_0, P)$ are also executed in $\pi(t_1, P)$ and vice versa. Similarly, all instances c'_i of every $c' \in C'$ that are executed in $\pi(t_0, P')$ are also executed in $\pi(t_1, P')$ and vice versa.

• ii) By Definition 6 and assumption t_0 and t_1 satisfy $\forall (n_i, n'_i) \in N_i.\, rsc(n_i, \pi(t_0, P)) \wedge rsc(n'_i, \pi(t_0, P)) \wedge value(n_i) = value(n'_i)$. By Lemma 1, t_0 and t_1 compute the same symbolic values for the variables used in n_i. Similarly, t_0 and t_1 compute the same symbolic values for the variables used in n'_i. As $value(n_i) = value(n'_i)$, t_0 and t_1 also compute the same symbolic values for the variables used in n_i and n'_i accross both versions. Thus, by Lemma 3, by Lemma 4, and by i) t_1 computes the same values for output o in P and P'. ■

Lemma 6 (*Homogenity - Difference Condition*)

Given statements C in program P are changed to C' yielding P', if input t_0 is difference-revealing for P and P', then every input t_1 satisfying $diff(o, \pi(t_0, P), \pi(t_0, P'))$ i) is difference-revealing, ii) computes the same symbolic values for o_i in P and the same symbolic values for o'_i in P', and iii) exercises exactly the same instances of $c \in C$ in P and of $c' \in C'$ in P'.

Proof: Assume, t_0 is difference-revealing and t_1 satisfies $diff(o, \pi(t_0, P), \pi(t_0, P'))$. We prove i) t_1 is also difference-revealing and ii) t_0 and t_1 exercise exactly the same instances of $c \in C$ in P and of $c' \in C'$ in P'.

• i+ii) By Definition 7 and assumption, t_1 and t_0 satisfy $rsc(o_i, \pi)$, $rsc(o'_i, \pi')$, and $value(o_i) \neq value(o'_i)$. By Lemma 1, the variables used in o_i have the same symbolic values in $\pi(t_0, P)$ and $\pi(t_1, P)$. By the same lemma, the variables used in o'_i have the same symbolic values in $\pi(t_0, P')$ and $\pi(t_1, P')$. As $value(o_i) \neq value(o'_i)$, for t_0 and t_1 the symbolic output is different across P and P'. Hence, t_1 is also difference-revealing.

• iii) By Definition 7, assumption, and Lemma 2, exactly those instances of every $c \in C$ in $\pi(t_0, P)$ and every $c' \in C'$ in $\pi(t_0, P')$ are also executed in $\pi(t_1, P)$ and $\pi(t_1, P')$, respectively. Hence, t_0 and t_1 exercise exactly the same instances of $c \in C$ in P and of $c' \in C'$ in P'. ■

Theorem 1 *(Sound Generalization)*

Proof: Assume, Algorithm 2 computes condition Φ_0 for the execution of t_0 on both program versions, P and P', and input t_1 satisfies Φ_0. We prove that t_0 and t_1 are in the same differential partition by showing i) if t_0 is equivalence-revealing, then t_1 is equivalence-revealing and ii) if t_0 is difference-revealing, then t_1 is difference-revealing.

• i) Assume, t_0 is equivalence-revealing. According to Algorithm 2 we distinguish two cases a) there does not exists an instance of $c' \in C'$ in $\pi(t_0, P')$ and b) otherwise. If a), then Φ_0 is $\bigwedge_{c' \in C'} reach(c', \pi(t_0, P'))$. As t_1 satisfies Φ_0, by the collorary of Lemma 2, t_1 does neither execute any $c \in C$ in P nor any $c' \in C'$ in P'. Thus, t_1 is equivalence-revealing. If b), then Φ_0 is $prop(o, \pi(t_0, P), \pi(t_0, P'))$ because t_0 is equivalence-revealing and $value(o_i) = value(o'_i)$. As t_1 satisfies Φ_0, by Lemma 5, t_1 is equivalence-revealing.
• ii) Assume, t_0 is difference-revealing. Thus, at least one changed statement is executed and $value(o_i) \neq value(o'_i)$. According to Algorithm 2, Φ_0 is $diff(o, \pi(t_0, P), \pi(t_0, P'))$. As t_1 satisfies Φ_0, by Lemma 6, t_1 is difference-revealing. ■

A.2 Exhaustiveness

In the following, we postulate and proof the exhaustiveness of Algorithm 1 that explores differential partitions (cf. Theorem 2).

Theorem 2 (*Exhaustive Exploration*)

If there exists an input t that computes different values for the output o in versions P and P' and Algorithm 1 terminates with regression test suite T, then there exists a test case $t^+ \in T$ so that t satisfies $diff(o, \pi(t^+, P), \pi(t^+, P'))$.

Informally, if the verification procedure terminates then all differential partitions have been explored.

The respective proof leverages the exhaustiveness of the exploration based on relevant slices. The applicable lemmas and the definition of distance are repeated in the following.

Lemma 7 (*Branch Negation in reordered RSC [19]*)

Let s_i be a statement instance in the traces $\pi(t_0, P)$ and $\pi(t_1, P)$ for the execution of inputs t_0 and t_1 on program P. Let f and g be the reordered $rsc(s_i, \pi(t_0, P))$ and the reordered $rsc(s_i, \pi(t_1, P))$, respectively. Suppose, f is $\varphi_1 \wedge \ldots \wedge \varphi_{j-1} \wedge \varphi_j$ and g is $\psi_1 \wedge \ldots \wedge \psi_{i-1} \wedge \psi_i$. If the first different branch condition between f and g is at location k, then $\varphi_k = \neg\psi_k$.

Notation: The *first different branch condition* is at location k for conditions $\varphi_1 \wedge \ldots \wedge \varphi_{j-1} \wedge \varphi_j$ and $\psi_1 \wedge \ldots \wedge \psi_{i-1} \wedge \psi_i$ if and only if for all $m < k$, $\varphi_m = \psi_m$ and $\varphi_k \neq \psi_k$.

Lemma 8 (*Same Prefix in reordered RSC [19]*)

Let s_i be a statement instance in the traces $\pi(t_0, P)$ and $\pi(t_1, P)$ for the execution of inputs t_0 and t_1 on program P. Suppose, $\psi_1 \wedge \ldots \wedge \psi_{i-1} \wedge \psi_i$ is a prefix of the reordered $rsc(s_i, \pi(t_0, P))$. If t_1 satisfies $\psi_1 \wedge \ldots \wedge \psi_{i-1} \wedge \neg\psi_i$,

then the reordered $rsc(s_i, \pi(t_1, P))$ *must contain* $\psi_1 \wedge \ldots \wedge \psi_{i-1} \wedge \neg\psi_i$ *as a prefix.*

Notation: The function $rs\_dist(f, g)$ denotes the distance of the reordered relevant slice condition f to the reordered relevant slice condition g. Suppose, $f = \varphi_1 \wedge \ldots \wedge \varphi_{j-1} \wedge \varphi_j$ and $g = \psi_1 \wedge \ldots \wedge \psi_{i-1} \wedge \psi_i$. Let k be the first different branch condition if there exists one and $min(i+1, j+1)$ otherwise. We define $rs\_dist(f, g) \stackrel{def}{=} 1 - \frac{k}{i+1}$. When f and g are the same, the distance of f to g is zero.

Lemma 9 (*Exhaustive Relevant Slice Exploration - P'*)

Suppose, in line 6 of Algorithm 1 the input t_0 *is chosen from the queue and the condition computed in line 7 implies* $rsc(s'_i, \pi(t_0, P'))$ *for statement instance* s'_i *in execution trace* $\pi(t_0, P')$*. If there exists an input* t *that (i) exercises the same relevant slices of all branch instances upon which* s'_i *dynamically control-depends than* t_0 *but computes a different value for* s'_i*, (ii) the condition computed by Algorithm 2 for* t *implies* $rsc(s'_i, \pi(t, P'))$*, and (iii) Algorithm 1 terminates with regression test suite* T*, then there exists a test case* $t^+ \in T$ *that satisfies* $rsc(s'_i, \pi(t, P'))$*.*

Note, by Lemma 1 necessary condition 9.i) requires that t evaluates all branch instances upon which s'_i dynamically control-depends in the same direction than t_0 but computes a different value for s'_i.

Proof: Assume, *condition* in line 7 of Algorithm 1 implies $rsc(s'_i, \pi(t_0, P'))$ for statement instance s'_i in trace $\pi(t_0, P')$. Further assume, A.i) input t exercises the same relevant slices of all branch instances upon which s'_i dynamically control-depends than t_0, A.ii) the *condition* computed by Algorithm 2 for t implies $rsc(s'_i, \pi(t, P'))$, and A.iii) Algorithm 1 terminates with regression test suite T. We prove that there exists a test case $t^+ \in T$ that satisfies $rsc(s'_i, \pi(t, P'))$.

In line 16 of Algorithm 4 the branch conditions in P' within *condition* are reordered. Because every order-preserving subset of a sorted set is also sorted, $rsc(s'_i, \pi(t_0, P'))$ is also reordered within *reordered'*. Let $f = \sigma_1 \wedge \ldots \wedge \sigma_{j-1} \wedge \sigma_j$ be the reordered $rsc(s'_i, \pi(t_0, P'))$. Let $g = \phi_1 \wedge \ldots \wedge \phi_{i-1} \wedge \phi_i$ be the reordered $rsc(s'_i, \pi(t, P')$. Suppose, the first different branch condition between f and g is at location k. In lines 19-22 of Algorithm 4 $constr = \sigma_1 \wedge \ldots \wedge \sigma_{k-1} \wedge \neg\sigma_k$ is constructed and solved. According to Lemma 7, $\neg\sigma_k = \phi_k$ and thus $\sigma_1 \wedge \ldots \wedge \sigma_{k-1} \wedge \neg\sigma_k$ is the same as $\phi_1 \wedge \ldots \wedge \phi_{k-1} \wedge \phi_k$. Note that g and thus $\phi_1 \wedge \ldots \wedge \phi_{k-1} \wedge \phi_k$ are satisfiable, as g is the relevant slice condition for the feasible path $\pi(t, P')$. Therefore, $\sigma_1 \wedge \ldots \wedge \sigma_{k-1} \wedge \neg\sigma_k$ is satisfiable. In line 21, the input t_1, as solution to this formula, is added to the *queue*.

We show that the *condition* computed for t_1 in line 7 of Algorithm 1 implies $\sigma_1 \wedge \ldots \wedge \sigma_{k-1} \wedge \neg\sigma_k$ and the relevant slice distance to g is strictly decreasing.

By A.i) and the definition of the reorder-function, the branch conditions of all branch instances upon which s'_i dynamically control-depends are placed before the first different branch condition. In other words, A.i) for t_0 is preserved in $\sigma_1 \wedge \ldots \wedge \sigma_k$ so that input t_1 does also exercise s'_i. By A.ii) and Algorithm 2, the *condition* computed by Algorithm 2 for t_1 implies $rsc(s'_i, \pi(t_1, P'))$. Let h be the reordered $rsc(s'_i, \pi(t_1, P'))$. By Lemma 8, h has $\sigma_1 \wedge \ldots \wedge \sigma_{k-1} \wedge \neg\sigma_k$ as prefix. The index of the first different branch for h and g is greater than the index k of the first different branch for f and g. Thus, $rs\_dist(h, g) < rs\_dist(f, g)$. The distance is strictly decreasing. ■

Lemma 10 (*Exhaustive Relevant Slice Exploration - P*)

Suppose, in line 6 of Algorithm 1 the input t_0 is chosen from the queue and the condition computed in line 7 implies $rsc(s'_i, \pi(t_0, P'))$ for statement instance s'_i in execution trace $\pi(t_0, P')$ and $rsc(s_i, \pi(t_0, P))$ for statement instance s_i in execution trace $\pi(t_0, P)$. If there exists an input t that (i) exercises the same relevant slice of s_i than t_0 in P', (ii) exercises the same relevant slices of all branch instances upon which s_i dynamically control-depends than t_0 but computes a different value for s_i in P, (iii) the condition computed by Algorithm 2 for t implies $rsc(s_i, \pi(t, P))$, and (iv) Algorithm 1 terminates with regression test suite T, then there exists a test case $t^+ \in T$ that satisfies $rsc(s'_i, \pi(t_0, P')) \wedge rsc(s_i, \pi(t, P))$.

Proof: The proof is analogous to the proof of Lemma 9. However, the constraint that t^+ has to satisfy in lines 8-15 of Algorithm 4 dictates that all the branch conditions $(\psi'_0 \wedge \ldots \wedge \psi'_m)$ for P' in *condition* remain satisfied for *constr*. Thus, $rsc(s'_i, \pi(t_0, P'))$ remains satisfied for every generated test case t^+. ■

Lemma 11 (*Enabling Reachability 1*)

Let C' be changed statements in program P' and input $t_0 \in$ queue does not exercise any $c' \in C'$. If there exists an input t that exercises an instance of $c' \in C'$ and Algorithm 1 terminates with regression test suite T, then there exists a test case $t^+ \in T$ that satisfies $reach(c', \pi(t, P'))$.

Notation: Let the distance function $dist(s_i, c, \pi)$ be zero if s_i is an instance of statement c in trace π, infinite if there is no transitive control-dependence of c onto statement s of which s_i is an instance in π, and the number of static control-dependence edges of the shortest path from statement c' to s, otherwise.

Proof: Assume, t exercises an instance of changed statement $c' \in C'$ and Algorithm 1 terminates with regression test suite T. We prove that there exists $t^+ \in T$ that satisfies $reach(c', \pi(t, P'))$.

After t_0 is chosen in line 6, its *condition* is computed in line 7 by Algorithm 2. Because t_0 does not exercise $c' \in C'$, the *condition* becomes

$reach(c', \pi(t_0, P'))$. The condition is passed into Algorithm 4 and reordered to $(\varphi'_0 \wedge \ldots \wedge \varphi'_m)$ (line 16). The constituent branch conditions φ' are negated one-by-one and checked for satisfiability (lines 18-23). Let t_1 be a satisfying solution that is added to the *queue*.

Let b'_i be the first different instance of a branch upon which c' control-depends in traces $\pi(t_0, P')$ and $\pi(t, P')$. Clearly, there exists an input t_1 that evaluates b'_i in the other direction than t_0 (and the same direction than t). By Definition 5, *condition* implies the relevant slice condition of b'_i. Since b'_i is the first different instance of a branch upon which c' control-depends, inputs t and t_1 exercise the same relevant slices of all branch instances upon which b'_i dynamically control-depends than t_0. By Lemma 9 and because Algorithm 1 terminates with regression test suite T, there exists a test case $t_1^+ \in T$ that satisfies $rsc(b'_i, \pi(t_1, P'))$. By Lemma 1, t_1, t_1^+, and t evaluate b'_i in the same direction. Let d'_i be the first different instance of a branch upon which c' control-depends in traces $\pi(t_1^+, P')$ and $\pi(t, P')$. Clearly, $dist(d'_i, c', \pi(t_1^+, P')) < dist(b'_i, c', \pi(t_0, P'))$. The distance is strictly decreasing until there is a test case t^+ generated that evaluates all branch instances upon which c' control-depends in the same direction than t. Thus by Definition 5 and Lemma2, t^+ satisfies $reach(c', \pi(t, P'))$. As Algorithm 1 terminates, t^+ is generated eventually. ■

Lemma 12 (*Enabling Reachability 2*)

Let statements C in program P be changed to C' yielding P'. Let input $t_0 \in$ queue exercise in P' at least one instance of a changed statement. If there exists an input t that exercises C_i of C in P and C'_i of C' in P' and Algorithm 1 terminates with regression test suite T, then there exists a test case $t^+ \in T$ that satisfies $\bigwedge_{c' \in C'} reach(c', \pi(t, P')) \wedge \bigwedge_{c \in C} reach(c, \pi(t, P))$.

Proof: Assume, t exercises instance c_i of original statement $c \in C$ in P and c'_i of changed statement $c' \in C'$ in P'. Further assume Algorithm 1 terminates with regression test suite T. By Definition 5 and Lemma 2, we prove that there exists $t^+ \in T$ that exercises change instances c'_i in P' and c_i in P.

After t_0 is chosen in line 6, its *condition* is computed in line 7. Because t_0 exercises in P' some instances of changed statements, the *condition* becomes either $prop(o, \pi(t_0, P), \pi(t_0, P'))$ or $diff(o, \pi(t_0, P), \pi(t_0, P'))$. This condition is passed into Algorithm 4 and in both cases by definitions 6 and 7, $condition \rightarrow reach(c, \pi(t_0, P))$ and $condition \rightarrow reach(c', \pi(t_0, P'))$. We distinguish three cases: i) t_0 does not exercise c'_i in $\pi(t_0, P'))$, ii) t_0 exercises c'_i in $\pi(t_0, P'))$ but not c_i in $\pi(t_0, P))$, and iii) t_0 exercises c'_i in $\pi(t_0, P'))$ and c_i in $\pi(t_0, P))$.

• i) Assume, $t_0 \in$ *queue* does not exercise c'_i in $\pi(t_0, P'))$.

Let b'_i be the first different instance of a branch upon which c' control-depends in traces $\pi(t_0, P')$ and $\pi(t, P')$. Clearly, there exists an input t_1 that eval-

uates b'_i in the other direction than t_0 (and the same direction than t). By Definition 5, *condition* implies the relevant slice condition of b'_i. Since b'_i is the first different instance of a branch upon which c' control-depends, inputs t and t_1 exercise the same relevant slices of all branch instances upon which b'_i dynamically control-depends than t_0. By Lemma 9 and because Algorithm 1 terminates with regression test suite T, there exists a test case $t_1^+ \in T$ that satisfies $rsc(b'_i, \pi(t_1, P'))$. By Lemma 1, t_1, t_1^+, and t evaluate b'_i in the same direction. Let d'_i be the first different instance of a branch upon which c' control-depends in traces $\pi(t_1^+, P')$ and $\pi(t, P')$. Clearly, $dist(d'_i, c', \pi(t_1^+, P')) < dist(b'_i, c', \pi(t_0, P'))$. The distance is strictly decreasing until there is a test case generated that evaluates all branch instances upon which c'_i control-depends in the same direction than t and thus exercises c'_i in P'. As Algorithm 1 terminates, this test case is generated eventually and added to the *queue* to be evaluated in the next case.

• ii) Assume, $t_0 \in$ *queue* exercises c'_i in $\pi(t_0, P'))$ but not c_i in $\pi(t_0, P))$. Let b_i be the first different instance of a branch upon which c control-depends in traces $\pi(t_0, P)$ and $\pi(t, P)$. Clearly, there exists an input t_1 that exercises the same relevant slices of c'_i in P' but evaluates b_i in the other direction than t_0 in P (and the same direction than t). By Definition 5, *condition* implies the relevant slice condition of b_i. Since b_i is the first different instance of a branch upon which c control-depends, inputs t and t_1 exercise the same relevant slices of all branch instances upon which b_i dynamically control-depends than t_0. By Lemma 10 and because Algorithm 1 terminates with regression test suite T, there exists a test case $t_1^+ \in T$ that satisfies $rsc(s'_i, \pi(t_0, P')) \wedge rsc(b_i, \pi(t_1, P))$. By Lemma 1, t_1, t_1^+, and t evaluate b_i in the same direction in P. Let d_i be the first different instance of a branch upon which c control-depends in traces $\pi(t_1^+, P)$ and $\pi(t, P)$. Clearly, $dist(d_i, c, \pi(t_1^+, P)) < dist(b_i, c, \pi(t_0, P))$. The distance is strictly decreasing until there is a test case generated that evaluates all branch instances upon which c_i control-depends in the same direction than t and thus exercises c_i in P and c'_i in P'. As Algorithm 1 terminates, this test case is generated eventually and added to the *queue* to be evaluated in the next case.

• iii) If $t_0 \in$ *queue* exercises c'_i in $\pi(t_0, P'))$ and c_i in in $\pi(t_0, P))$, then t_0 exercises c'_i in P' and c_i in P. ■

Lemma 13 (*Enabling Propagation to the Output 1*)

Let statements C in program P be changed to C' yielding P'. Let input $t_0 \in$ queue exercise instances C_i of C in P and C'_i of C' in P' and compute the same values for instance o_i of output o in trace $\pi(t_0, P)$ and o'_i in trace $\pi(t_0, P')$. If there exists an input t that exercises C_i and C'_i, t computes the same values for o_i and o'_i, Algorithm 3 computes N_i for $\pi(t, P)$ and $\pi(t, P')$, and Algorithm 1 terminates with regression test suite T, then there

exists a test case $t^+ \in T$ *that exercises* C_i *and* C'_i, t^+ *computes the same values for* o_i *and* o'_i, *Algorithm 3 computes* M_i *for* $\pi(t^+, P)$ *and* $\pi(t^+, P')$, *and* $\forall(n_i, n'_i) \in N_i.\exists(m_i, m'_i) \in M_i.(m_i \rightsquigarrow n_i)$ *in* $\pi(t^+, P)$ *and* $(m'_i \rightsquigarrow n'_i)$ *in* $\pi(t^+, P')$.

Notation: Let the distance function $ddist(s_i, r_i, \pi)$ be zero if statement instances $s_i = r_i$ in trace π, the number of dynamic control-, data-, and potential dependence edges of the shortest path from r_i to s_i if $s_i \rightsquigarrow r_i$ in π, and infinite, otherwise.

Proof: Assume, t exercises instance c_i of original statement $c \in C$ in P and c'_i of changed statement $c' \in C'$ in P', t computes the same values for o_i and o'_i, and Algorithm 1 terminates with regression test suite T. Further assume, Algorithm 3 computes (n_i, n'_i) for $\pi(t, P)$ and $\pi(t, P')$. Let n and n' be the statements corresponding to the instances n_i and n'_i, respectively. We proof that there exists $t^+ \in T$ that exercises c_i and c'_i, t^+ computes the same values for o_i and o'_i, Algorithm 3 computes M_i^+ for $\pi(t^+, P)$ and $\pi(t^+, P')$ and there exists $(m_i^+, m_i'^+) \in M_i^+$ so that $(m_i^+ \rightsquigarrow n_i)$ in $\pi(t^+, P)$ and $(m_i'^+ \rightsquigarrow n'_i)$ in $\pi(t^+, P')$.

After t_0 is chosen in line 6, its *condition* is computed in line 7. Because t_0 exercises some instances of changed statements and computes the same values for o_i and o'_i, the *condition* becomes $prop(o, \pi(t_0, P), \pi(t_0, P'))$. By Definition 6, Algorithm 3 computes M_i for $\pi(t_0, P)$ and $\pi(t_0, P')$ and by all-quantor instantiation of M_i to (m_i, m'_i), the *condition* implies $rsc(m_i, \pi(t_0, P)) \wedge rsc(m'_i, \pi(t_0, P')) \wedge value(m_i) = value(m'_i)$.

We distinguish three cases: i) $(m'_i \not\rightsquigarrow n'_i) \wedge (n'_i \not\rightsquigarrow m'_i)$, ii) $((m'_i \rightsquigarrow n'_i) \vee (n'_i \rightsquigarrow m'_i))$ and $(m_i \not\rightsquigarrow n_i) \wedge (n_i \not\rightsquigarrow m_i)$, and iii) $((m'_i \rightsquigarrow n'_i) \vee (n'_i \rightsquigarrow m'_i))$ and $((m_i \rightsquigarrow n_i) \vee (n_i \rightsquigarrow m_i))$.

• i) Assume, $m'_i \not\rightsquigarrow n'_i$ in $\pi(t_0, P')$ and $n'_i \not\rightsquigarrow m'_i$ in $\pi(t, P')$. Let b'_i be the first different branch instance that satisfies $m'_i \rightsquigarrow b'_i$ in $\pi(t_0, P')$ and $n'_i \rightsquigarrow b'_i$ in $\pi(t, P')$. Clearly, there exists an input t_1 that exercises c_i in P and c'_i in P' and evaluates b'_i in the other direction than t_0 in P' (and the same direction than t). As the *condition* implies $rsc(m'_i, \pi(t_0, P')$ and $m'_i \rightsquigarrow b'_i$, the *condition* also implies the relevant slice condition of b'_i in $\pi(t_0, P')$. Since b'_i is the first different instance that satisfies $m'_i \rightsquigarrow b'_i$ in $\pi(t_0, P')$ and $n'_i \rightsquigarrow b'_i$ in $\pi(t, P')$, inputs t and t_1 exercise the same relevant slices of all branch instances upon which b'_i dynamically control-depends than t_0. By Lemma 9 and because Algorithm 1 terminates with regression test suite T, there exists a test case $t_1^+ \in T$ that satisfies $rsc(b'_i, \pi(t_1, P'))$. By Lemma 1, t_1, t_1^+, and t evaluate b'_i in the same direction in P'. Let d'_i be the first different branch instance that satisfies $m'_i \rightsquigarrow d'_i$ in $\pi(t_1^+, P')$ and $n'_i \rightsquigarrow d'_i$ in $\pi(t, P')$. Clearly, $dist(d'_i, n', \pi(t_1^+, P')) < dist(b'_i, n', \pi(t_0, P'))$. The distance is strictly decreasing until there is a test case t_2^+ generated that evaluates all branch instances a'_i in the same direction than t that satisfy $m'_i \rightsquigarrow a'_i$ in $\pi(t_2^+, P')$ and $n'_i \rightsquigarrow a'_i$

in $\pi(t, P')$. Thus, for the execution of t_2^+ on P' either $n_i' \rightsquigarrow m_i'$ or $m_i' \rightsquigarrow n_i'$ in $\pi(t_2^+, P')$. By Lemma 4, t_2^+ also exercises c_i' in P' and c in P. As Algorithm 1 terminates, t_2^+ is generated eventually and added to the *queue* to be evaluated in the next case.

• ii) Assume, either $m_i' \rightsquigarrow n_i'$ in $\pi(t_0, P')$ or $n_i' \rightsquigarrow m_i'$ in $\pi(t, P')$. Assume further, $m_i \not\rightsquigarrow n_i$ in $\pi(t_0, P)$ and $n_i \not\rightsquigarrow m_i$ in $\pi(t, P)$. Let b_i be the first different branch instance that satisfies $m_i \rightsquigarrow b_i$ in $\pi(t_0, P)$ and $n_i \rightsquigarrow b_i$ in $\pi(t, P)$. Clearly, there exists an input t_1 that exercises c_i in P and c_i' in P' and evaluates b_i in the other direction than t_0 in P (and the same direction than t). As the *condition* implies $rsc(m_i, \pi(t_0, P)$ and $m_i \rightsquigarrow b_i$, the *condition* also implies the relevant slice condition of b_i in $\pi(t_0, P)$. Since b_i is the first different instance that satisfies $m_i \rightsquigarrow b_i$ in $\pi(t_0, P)$ and $n_i \rightsquigarrow b_i$ in $\pi(t, P)$, inputs t and t_1 exercise the same relevant slices of all branch instances upon which b_i dynamically control-depends than t_0. By Lemma 10 and because Algorithm 1 terminates with regression test suite T, there exists a test case $t_1^+ \in T$ that satisfies $rsc(m_i', \pi(t_0, P')) \wedge rsc(b_i, \pi(t_1, P))$. By Lemma 1, t_1, t_1^+, and t evaluate b_i in the same direction in P. Let d_i be the first different branch instance that satisfies $m_i \rightsquigarrow d_i$ in $\pi(t_1^+, P)$ and $n_i \rightsquigarrow d_i$ in $\pi(t, P)$. Clearly, $dist(d_i, n, \pi(t_1^+, P)) < dist(b_i, n, \pi(t_0, P))$. The distance is strictly decreasing until there is a test case t_2^+ generated that evaluates all branch instances a_i in the same direction than t that satisfy $m_i \rightsquigarrow a_i$ in $\pi(t_2^+, P)$ and $n_i \rightsquigarrow a_i$ in $\pi(t, P)$. Thus, for the execution of t_2^+ on P either $n_i \rightsquigarrow m_i$ or $m_i \rightsquigarrow n_i$ in $\pi(t_0, P)$. By Lemma 4, t_2^+ also exercises c_i' in P' and c in P. Because t_2^+ (still) satisfies $rsc(m_i', \pi(t_0, P'))$, for the execution of t_2^+ on P' either $n_i' \rightsquigarrow m_i'$ or $m_i' \rightsquigarrow n_i'$ in $\pi(t_2^+, P')$. As Algorithm 1 terminates, t_2^+ is generated eventually and added to the *queue* to be evaluated in the next case.

• iii) Assume, either $m_i' \rightsquigarrow n_i'$ in $\pi(t_0, P')$ or $n_i' \rightsquigarrow m_i'$ in $\pi(t, P')$. Assume further, either $m_i \rightsquigarrow n_i$ in $\pi(t_0, P)$ or $n_i \rightsquigarrow m_i$ in $\pi(t, P)$. Because $align(m_i, m_i')$ and $align(n_i, n_i')$, the assumption reduces to two cases: iii.a) $m_i' \rightsquigarrow n_i'$ in $\pi(t_0, P')$ and $m_i \rightsquigarrow n_i$ in $\pi(t_0, P)$, and iii.b) $n_i' \rightsquigarrow m_i'$ in $\pi(t_0, P')$ and $n_i \rightsquigarrow m_i$ in $\pi(t_0, P)$.

The case iii.a) proves that there exists $t^+ \in T$ that exercises c_i and c_i', t^+ computes the same values for o_i and o_i', Algorithm 3 computes M_i^+ for $\pi(t^+, P)$ and $\pi(t^+, P')$ and there exists $(m_i^+, m_i'^+) \in M_i^+$ so that $(m_i^+ \rightsquigarrow n_i)$ in $\pi(t^+, P)$ and $(m_i'^+ \rightsquigarrow n_i')$ in $\pi(t^+, P')$. The remainder elaborates on case iii.b) if not stated otherwise. The case iii.b) can only occur if $value(m_i) = value(m_i')$ for t_0 because by assumption $align(m_i, m_i')$ and $\neg isChanged(m_i')$ holds. By the properties of transitive dynamic data- and potential dependence, the value of m_i' and m_i control the value of n_i' and n_i, respectively.

Clearly, there exists an input t_1 that computes for m_i and m_i' the same symbolic values than t and satisfy $value(m_i) \neq value(m_i')$. By assumption, the *condition* implies $rsc(m_i', \pi(t_0, P'))$, $rsc(m_i, \pi(t_0, P))$, and $(value(m_i) = value(m_i')$. We distinguish two cases, iii.b.1) t_0 computes for m_i and m_i' dif-

ferent symbolic values than t and iii.b.2) t_0 computes for m_i and m_i' the same symbolic values than t but $value(m_i) \neq value(m_i')$.
iii.b.1) Assume t_0 computes different values for m_i and m_i' than t. Since by case iii.b, $n_i \rightsquigarrow m_i$ in $\pi(t, P)$, the inputs t and t_1 exercise the same relevant slices of all branch instances upon which m_i dynamically control-depends than t_0. By Lemma 10 and because Algorithm 1 terminates with regression test suite T, there exists a test case $t_1^+ \in T$ that satisfies $rsc(m_i', \pi(t_0, P')) \wedge rsc(m_i, \pi(t_1, P))$. By Lemma 1, t_1, t_1^+, and t compute the same symbolic value for m_i in P. Input t_1^+ does clearly propagate beyond (m_i, m_i'). In other words, given we compute (l_i, l_i') for t_1^+ like we compute (m_i, m_i') for t_0, then $ddist(l_i', n_i', \pi(t_1^+, P')) + ddist(l_i, n_i, \pi(t_1^+, P)) < ddist(m_i', n_i', \pi(t_0, P')) + ddist(m_i, n_i, \pi(t_0, P))$.
iii.b.2) Assume t_0 already computes for m_i and m_i' the same symbolic values than t but $value(m_i) \neq value(m_i')$. In lines 2-7 of Algorithm 4, the equivalence condition $value(m_i) = value(m_i')$ is negated and conjoined with the branch conditions in *condition* yielding the constraint *constr*. A solution t_1^+ to *constr* does clearly propagate beyond (m_i, m_i'). In other words, given we compute (l_i, l_i') for t_1^+ like we compute (m_i, m_i') for t_0, then $ddist(l_i', n_i', \pi(t_1^+, P')) + ddist(l_i, n_i, \pi(t_1^+, P)) < ddist(m_i', n_i', \pi(t_0, P')) + ddist(m_i, n_i, \pi(t_0, P))$.
Using cases i), ii) and iii), for the generated test cases, the distance is strictly reduced until a test case is generated that yields $m_i' \rightsquigarrow n_i'$. By Lemma 4, every generated test case remains exercising c_i' in P' and c in P. ■

Lemma 14 (*Enabling Propagation to the Output 2*)

Let statements C in program P be changed to C' yielding P'. Let input $t_0 \in$ queue exercise instances C_i of C in P and C_i' of C' in P' and compute the same values for instance o_i of output o in trace $\pi(t_0, P)$ and o_i' in trace $\pi(t_0, P')$. If there exists an input t that exercises C_i and C_i' and computes different values for o_i and o_i' and Algorithm 1 terminates with regression test suite T, then there exists a test case $t^+ \in T$ that satisfies $diff(o, \pi(t, P), \pi(t, P'))$.

Proof: Assume, t exercises instance c_i of original statement $c \in C$ in P and c_i' of changed statement $c' \in C'$ in P', t computes different values for o_i and o_i' and Algorithm 1 terminates with regression test suite T. By Definition 7, we prove that there exists $t^+ \in T$ that satisfies $rsc(o_i', \pi(t, P'))$, $rsc(o_i, \pi(t, P))$, and $value(o_i) \neq value(o_i')$.

After t_0 is chosen in line 6, its *condition* is computed in line 7. Because t_0 exercises some instances of changed statements and computes the same values for o_i and o_i', the *condition* becomes $prop(o, \pi(t_0, P), \pi(t_0, P'))$.

Clearly, there exists a test case t_1 that satisfies the following. Input t_1 exercises instance c_i in $\pi(t_1, P)$ and c_i' in $\pi(t_1, P')$ and computes the same values for o_i and o_i'. Algorithm 3 computes N_i for $\pi(t_1, P)$ and $\pi(t_1, P')$ and $(n_i, n_i') \in N_i$.

Importantly, t_1 satisfies $rsc(o'_i, \pi(t, P'))$, and for every $(n_i, n'_i) \in N_i$ the distance $ddist(o_i, n_i, \pi(t_1, P))$ is minimal, and o_i transitively, dynamically *data*-depends on n_i in $\pi(t_1, P)$. By Lemma 13 and because Algorithm 1 terminates with regression test suite T, a test case $t_1^+ \in T$ is generated eventually that satisfies $prop(o, \pi(t_1, P), \pi(t_1, P'))$.

After t_1^+ is chosen in line 6, its *condition* is computed in line 7. Because t_1^+ exercises some instances of changed statements and computes the same values for o_i and o'_i, the *condition* becomes $prop(o, \pi(t_1^+, P), \pi(t_1^+, P'))$. By Definition 6, Algorithm 3 computes N_i for $\pi(t_1^+, P)$ and $\pi(t_1^+, P')$ and by all-quantor instantiation of N_i to (n_i, n'_i), the *condition* implies $rsc(n_i, \pi(t_0, P)) \wedge rsc(n'_i, \pi(t_0, P')) \wedge value(n_i) = value(n'_i)$. As specified earlier, t_1^+ satisfies $rsc(o'_i, \pi(t, P'))$, and o_i transitively, dynamically *data*-depends on n_i in $\pi(t_1^+, P)$. Thus, the value of n_i directly influences the value of o_i.

Clearly, there exists an input t_2 that also satisfies $rsc(o'_i, \pi(t, P))$ and computes for the variables used in n_i the same symbolic values as t and satisfies $value(o_i) \neq value(o'_i)$. By assumption, the *condition* implies $rsc(o'_i, \pi(t_1^+, P'))$, $rsc(n_i, \pi(t_1^+, P))$, and $(value(o_i) = value(o'_i)$. We distinguish two cases, i) $n_i \neq o_i$ in $\pi(t_1^+)$ and ii) $n_i = o_i$ but $value(o_i) = value(o'_i)$ in $\pi(t_1^+)$.

• i) Assume, $n_i \neq o_i$ in $\pi(t_1^+)$. Then, t_1^+ computes different symbolic values for n_i than t. Since $o_i \rightsquigarrow n_i$ in $\pi(t, P)$, the inputs t and t_2 exercise the same relevant slices of all branch instances upon which n_i dynamically control-depends than t_1^+. By Lemma 10 and because Algorithm 1 terminates with regression test suite T, there exists a test case $t_2^+ \in T$ that satisfies $rsc(o'_i, \pi(t_1^+, P')) \wedge rsc(o_i, \pi(t_2, P))$. By Lemma 1, t_2, t_2^+, and t compute the same symbolic value for o_i in P and o'_i in P'. Case $value(o_i) = value(o'_i)$ is evaluated next.

• ii) Assume, $n_i = o_i$ but $value(o_i) = value(o'_i)$ in $\pi(t_1^+)$. In lines 2-7 of Algorithm 4, the equivalence condition $(value(o_i) = value(o'_i)$ is negated and conjoined with the branch conditions in *condition* yielding the constraint *constr*. A solution t_2^+ clearly satisfies $rsc(o'_i, \pi(t, P'))$, $rsc(o_i, \pi(t, P))$, and $value(o_i) \neq value(o'_i)$.

By cases i), ii) there exists $t^+ \in T$ that satisfies $rsc(o'_i, \pi(t, P'))$, $rsc(o_i, \pi(t, P))$, and $value(o_i) \neq value(o'_i)$. By Lemma 4, every generated test case remains exercising c'_i in P' and c in P. ■

Theorem 2 ***(Exhaustive Exploration)***

Proof: Assume, t exercises instance c_i of original statement $c \in C$ in P and c'_i of changed statement $c' \in C'$ in P', t computes different values for o_i and o'_i and Algorithm 1 terminates with regression test suite T. We prove that there exists $t^+ \in T$ so that t satisfies $diff(o, \pi(t^+, P), \pi(t^+, P'))$.

If $t_0 \in queue$ does not exercise c'_i in P', by Lemma 11, there exists a test case in T that exercises c'_i in P'. Let t_0 be that test case. If $t_0 \in queue$ exercises c'_i in P' but not c in P, by Lemma 12, there exists a test case in T that exercises

c_i' in P' and c_i in P. Let t_0 be that test case. If $t_0 \in queue$ exercises c_i' in P' and c_i in P and computes the same values for o_i and o_i', by Lemma 14, there exists a test case in $t^+ \in T$ that exercises c_i' in P' and c_i in P and t satisfies $diff(o, \pi(t^+, P), \pi(t^+, P'))$. ■

Bibliography

[1] Dawei Qi, Abhik Roychoudhury, and Zhenkai Liang. Test generation to expose changes in evolving programs. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, ASE '10, pages 397–406, New York, NY, USA, 2010. ACM.

[2] R. Santelices, P. K. Chittimalli, T. Apiwattanapong, A. Orso, and M. J. Harrold. Test-suite augmentation for evolving software. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, pages 218–227, Washington, DC, USA, 2008. IEEE Computer Society.

[3] Robert C. Seacord, Daniel Plakosh, and Grace A. Lewis. *Modernizing Legacy Systems: Software Technologies, Engineering Process and Business Practices*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[4] Elaine J. Weyuker. On Testing Non-Testable Programs. *The Computer Journal*, 25(4):465–470, November 1982.

[5] Glenford J. Myers and Corey Sandler. *The Art of Software Testing*. John Wiley & Sons, 2004.

[6] Zhongxian Gu, Earl T. Barr, David J. Hamilton, and Zhendong Su. Has the bug really been fixed? In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 55–64, New York, NY, USA, 2010. ACM.

[7] Zuoning Yin, Ding Yuan, Yuanyuan Zhou, Shankar Pasupathy, and Lakshmi Bairavasundaram. How do fixes become bugs? In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, ESEC/FSE '11, pages 26–36, New York, NY, USA, 2011. ACM.

[8] Allen Goldberg, T. C. Wang, and David Zimmerman. Applications of feasible path analysis to program testing. In *Proceedings of the 1994 ACM SIGSOFT international symposium on Software testing and analysis*, ISSTA '94, pages 80–94, New York, NY, USA, 1994. ACM.

[9] Margaret C. Thompson, Debra J. Richardson, and Lori A. Clarke. An information flow model of fault detection. In *Proceedings of the 1993 ACM SIGSOFT international symposium on Software testing and analysis*, ISSTA '93, pages 182–192, New York, NY, USA, 1993. ACM.

[10] A. Podgurski and L. A. Clarke. A formal model of program dependences and its implications for software testing, debugging, and maintenance. *IEEE Trans. Softw. Eng.*, 16:965–979, September 1990.

[11] Karl J. Ottenstein and Linda M. Ottenstein. The program dependence graph in a software development environment. In *Proceedings of the first ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments*, SDE 1, pages 177–184, New York, NY, USA, 1984. ACM.

[12] Mark Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, ICSE '81, pages 439–449, 1981.

[13] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. Program. Lang. Syst.*, 12:26–60, January 1990.

[14] B. Korel and J. Laski. Dynamic program slicing. *Inf. Process. Lett.*, 29:155–163, October 1988.

[15] Susan Horwitz and Thomas Reps. Efficient comparison of program slices. *Acta Inf.*, 28(9):713–732, November 1991.

[16] G. A. Venkatesh. The semantic approach to program slicing. In *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, PLDI '91, pages 107–119, New York, NY, USA, 1991. ACM.

[17] Tibor Gyimóthy, Árpád Beszédes, and Istán Forgács. An efficient relevant slicing method for debugging. In *Proceedings of the 7th European software engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC/FSE-7, pages 303–321, London, UK, UK, 1999. Springer-Verlag.

[18] Hiralal Agrawal, Joseph R. Horgan, Edward W. Krauser, and Saul London. Incremental regression testing. In *ICSM*, pages 348 – 357, 1993.

[19] Dawei Qi, Hoang D.T. Nguyen, and Abhik Roychoudhury. Path exploration based on symbolic output. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, ESEC/FSE '11, pages 278–288, New York, NY, USA, 2011. ACM.

[20] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19:385–394, July 1976.

[21] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: directed automated random testing. *SIGPLAN Not.*, 40:213–223, June 2005.

[22] Koushik Sen, Darko Marinov, and Gul Agha. Cute: a concolic unit testing engine for c. *SIGSOFT Softw. Eng. Notes*, 30:263–272, September 2005.

[23] Phil McMinn. Search-based software test data generation: a survey: Research articles. *Softw. Test. Verif. Reliab.*, 14(2):105–156, June 2004.

[24] Wei Jin and Alessandro Orso. Bugredux: reproducing field failures for in-house debugging. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 474–484, Piscataway, NJ, USA, 2012.

[25] Suzette Person, Guowei Yang, Neha Rungta, and Sarfraz Khurshid. Directed incremental symbolic execution. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 504–515, 2011.

[26] Mark Harman, Yue Jia, and William B. Langdon. Strong higher order mutation-based test data generation. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, ESEC/FSE '11, pages 212–222, New York, NY, USA, 2011. ACM.

[27] Patrice Godefroid. Higher-order test generation. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 258–269, New York, NY, USA, 2011. ACM.

[28] Mickaël Delahaye, Bernard Botella, and Arnaud Gotlieb. Explanation-based generalization of infeasible path. In *Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation*,

ICST '10, pages 215–224, Washington, DC, USA, 2010. IEEE Computer Society.

[29] Raul Santelices and Mary Jean Harrold. Exploiting program dependencies for scalable multiple-path symbolic execution. In *Proceedings of the 19th international symposium on Software testing and analysis*, ISSTA '10, pages 195–206, New York, NY, USA, 2010. ACM.

[30] Peter Boonstoppel, Cristian Cadar, and Dawson Engler. Rwset: attacking path explosion in constraint-based test generation. In *Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems*, TACAS'08/ETAPS'08, pages 351–366, Berlin, Heidelberg, 2008. Springer-Verlag.

[31] Matt Staats and Corina Păsăreanu. Parallel symbolic execution for structural test generation. In *Proceedings of the 19th international symposium on Software testing and analysis*, ISSTA '10, pages 183–194, New York, NY, USA, 2010. ACM.

[32] Saswat Anand, Patrice Godefroid, and Nikolai Tillmann. Demand-driven compositional symbolic execution. In *Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems*, TACAS'08/ETAPS'08, pages 367–381, Berlin, Heidelberg, 2008. Springer-Verlag.

[33] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. Automated whitebox fuzz testing. In *Proceedings of the Network and Distributed System Security Symposium*, NDSS '08. The Internet Society, 2008.

[34] Misty Davies, Corina Pasareanu, and Vishwanath Raman. Symbolic execution enhanced system testing. In Rajeev Joshi, Peter Müller, and Andreas Podelski, editors, *Verified Software: Theories, Tools, Experiments*, volume 7152 of *Lecture Notes in Computer Science*, pages 294–309. Springer Berlin / Heidelberg, 2012.

[35] Steffen Lehnert. A taxonomy for software change impact analysis. In *Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th annual ERCIM Workshop on Software Evolution*, IWPSE-EVOL '11, pages 41–50, New York, NY, USA, 2011. ACM.

[36] Taweesup Apiwattanapong, Alessandro Orso, and Mary Jean Harrold. Efficient and precise dynamic impact analysis using execute-after sequences.

In *Proceedings of the 27th international conference on Software engineering*, ICSE '05, pages 432–441, New York, NY, USA, 2005. ACM.

[37] Alessandro Orso, Taweesup Apiwattanapong, and Mary Jean Harrold. Leveraging field data for impact analysis and regression testing. *SIGSOFT Softw. Eng. Notes*, 28(5):128–137, September 2003.

[38] Xiaoxia Ren, Fenil Shah, Frank Tip, Barbara G. Ryder, and Ophelia Chesley. Chianti: A tool for change impact analysis of java programs. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 432–448. ACM Press, 2004.

[39] Suzette Person, Matthew B. Dwyer, Sebastian Elbaum, and Corina S. Păsăreanu. Differential symbolic execution. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, SIGSOFT '08/FSE-16, pages 226–237, New York, NY, USA, 2008. ACM.

[40] Raul Santelices and Mary Jean Harrold. Probabilistic slicing for predictive impact analysis. Technical Report CERCS, GIT-CERCS-10-10, College of Computing, Georgia Institute of Technology, 2010.

[41] Susan Horwitz, Jan Prins, and Thomas Reps. Integrating noninterfering versions of programs. *ACM Trans. Program. Lang. Syst.*, 11(3):345–387, July 1989.

[42] Dewayne E. Perry, Harvey P. Siy, and Lawrence G. Votta. Parallel changes in large-scale software development: an observational case study. *ACM Trans. Softw. Eng. Methodol.*, 10(3):308–337, July 2001.

[43] Susan Horwitz. Identifying the semantic and textual differences between two versions of a program. In *Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, PLDI '90, pages 234–245, New York, NY, USA, 1990. ACM.

[44] Daniel Jackson and David A. Ladd. Semantic diff: A tool for summarizing the effects of modifications. In *Proceedings of the International Conference on Software Maintenance*, ICSM '94, pages 243–252, Washington, DC, USA, 1994. IEEE Computer Society.

[45] Alex Loh and Miryung Kim. Lsdiff: a program differencing tool to identify systematic structural differences. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ICSE '10, pages 263–266, New York, NY, USA, 2010. ACM.

[46] F. I. Vokolos and P. G. Frankl. Empirical evaluation of the textual differencing regression testing technique. In *Proceedings of the International Conference on Software Maintenance*, ICSM '98, pages 44–, Washington, DC, USA, 1998. IEEE Computer Society.

[47] Webb Miller and Eugene W. Myers. A file comparison program. *Softw., Pract. Exper.*, 15(11):1025–1040, 1985.

[48] K. Zhang and D. Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM J. Comput.*, 18(6):1245–1262, December 1989.

[49] Taweesup Apiwattanapong, Alessandro Orso, and Mary Jean Harrold. Jdiff: A differencing technique and tool for object-oriented programs. *Automated Software Engg.*, 14(1):3–36, March 2007.

[50] David Binkley, Susan Horwitz, and Thomas Reps. Program integration for languages with procedure calls. *ACM Trans. Softw. Eng. Methodol.*, 4:3–35, January 1995.

[51] S. Horwitz, J. Prins, and T. Reps. On the adequacy of program dependence graphs for representing programs. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '88, pages 146–157, New York, NY, USA, 1988. ACM.

[52] Jeffrey M. Voas. Pie: A dynamic failure-based technique. *IEEE Transactions on Software Engineering*, 18:717–727, 1992.

[53] Shuvendu K. Lahiri, Chris Hawblitzel, Ming Kawaguchi, and Henrique Rebêlo. Symdiff: a language-agnostic semantic diff tool for imperative programs. In *Proceedings of the 24th international conference on Computer Aided Verification*, CAV'12, pages 712–717, Berlin, Heidelberg, 2012. Springer-Verlag.

[54] Raul Santelices, Mary Jean Harrold, and Alessandro Orso. Precisely detecting runtime change interactions for evolving software. In *Int'l Conf. on Software Testing, Verification and Validation (ICST)*. IEEE, 2010.

[55] Andreas Zeller. Yesterday, my program worked. today, it does not. why? In *Proceedings of the ESEC/FSE'99, 7th European Software Engineering Conference*, volume 1687 of *Lecture Notes in Computer Science*, pages 253–267. Springer, September 1999.

[56] Wei Jin, Alex Orso, and Tao Xie. Bert: A tool for behavioral regression testing. In *Proc. the 18th ACM SIGSOFT Symposium on the Foundations*

of Software Engineering (FSE 2010), Research Demonstration, pages 361–362, November 2010.

[57] Wei Jin, Alessandro Orso, and Tao Xie. Automated behavioral regression testing. In *Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation*, ICST '10, pages 137–146, Washington, DC, USA, 2010. IEEE Computer Society.

[58] Bogdan Korel and Ali M. Al-Yami. Automated regression test generation. In *Proceedings of the 1998 ACM SIGSOFT international symposium on Software testing and analysis*, ISSTA '98, pages 143–152, New York, NY, USA, 1998. ACM.

[59] Gordon Fraser and Neil Walkinshaw. Behaviourally adequate software testing. *Software Testing, Verification, and Validation, 2008 International Conference on*, 0:300–309, 2012.

[60] Dawei Qi, William Sumner, Feng Qin, Mai Zheng, Xiangyu Zhang, and Abhik Roychoudhury. Modeling software execution environment. In *19th IEEE Working Conference on Reverse Engineering*, WCRE'12, 2012.

[61] Neha Rungta, Eric G. Mercer, and Willem Visser. Efficient testing of concurrent programs with abstraction-guided symbolic execution. In *Proceedings of the 16th International SPIN Workshop on Model Checking Software*, pages 174–191, Berlin, Heidelberg, 2009. Springer-Verlag.

[62] Dawei Qi, Jooyong Yi, and Abhik Roychoudhury. Software change contracts. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 22:1–22:4, 2012.

[63] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. Exe: Automatically generating inputs of death. *ACM Trans. Inf. Syst. Secur.*, 12:10:1–10:38, December 2008.

[64] Mike Papadakis and Nicos Malevris. An empirical evaluation of the first and second order mutation testing strategies. *Software Testing Verification and Validation Workshop, IEEE International Conference on*, 0:90–99, 2010.

[65] N. Tracey, J. Clark, K. Mander, and J. McDermid. Automated test-data generation for exception conditions. *Softw. Pract. Exper.*, 30(1):61–79, January 2000.

[66] B. Korel. Automated software test data generation. *IEEE Trans. Softw. Eng.*, 16(8):870–879, August 1990.

[67] Nels E. Beckman, Aditya V. Nori, Sriram K. Rajamani, and Robert J. Simmons. Proofs from tests. In *Proceedings of the 2008 international symposium on Software testing and analysis*, ISSTA '08, pages 3–14, 2008.

[68] Patrice Godefroid, Aditya V. Nori, Sriram K. Rajamani, and Sai Deep Tetali. Compositional may-must program analysis: unleashing the power of alternation. In *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '10, pages 43–56, 2010.

[69] Bogdan Korel and Ali M. Al-Yami. Assertion-oriented automated test data generation. In *Proceedings of the 18th international conference on Software engineering*, ICSE '96, pages 71–80, Washington, DC, USA, 1996. IEEE Computer Society.

[70] Yves Le Traon, Benoit Baudry, and Jean-Marc Jezequel. Design by contract to improve software vigilance. *IEEE Trans. Softw. Eng.*, 32:571–586, August 2006.

[71] Ansuman Banerjee, Abhik Roychoudhury, Johannes A. Harlie, and Zhenkai Liang. Golden implementation driven software debugging. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, FSE '10, pages 177–186, New York, NY, USA, 2010. ACM.

[72] Matt Staats, Michael W. Whalen, and Mats P.E. Heimdahl. Programs, tests, and oracles: the foundations of testing revisited. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 391–400, 2011.

[73] Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29:366–427, December 1997.

[74] W. E. Howden. Weak mutation testing and completeness of test sets. *IEEE Trans. Softw. Eng.*, 8(4):371–379, July 1982.

[75] L.J. Morell. A theory of fault-based testing. *IEEE Transactions on Software Engineering*, 16:844–857, 1990.

[76] Marc Fisher, II, Jan Wloka, Frank Tip, Barbara G. Ryder, and Alexander Luchansky. An evaluation of change-based coverage criteria. In *Proceedings of the 10th ACM SIGPLAN-SIGSOFT workshop on Program analysis*

for software tools, PASTE '11, pages 21–28, New York, NY, USA, 2011. ACM.

[77] P.G. Frankl and E.J. Weyuker. A formal analysis of the fault-detecting ability of testing methods. *IEEE Transactions on Software Engineering*, 19(3):202 –213, mar 1993.

[78] S.C. Ntafos. A comparison of some structural testing strategies. *IEEE Transactions on Software Engineering*, 14(6):868 –874, jun 1988.

[79] Hong Zhu. A formal analysis of the subsume relation between software test adequacy criteria. *IEEE Transactions on Software Engineering*, 22(4):248 –255, apr 1996.

[80] Elaine J. Weyuker and Bingchiang Jeng. Analyzing partition testing strategies. *IEEE Trans. Softw. Eng.*, 17:703–711, July 1991.

[81] D. Hamlet and R. Taylor. Partition testing does not inspire confidence (program testing). *IEEE Transactions on Software Engineering*, 16:1402–1411, 1990.

[82] Joe W. Duran and Simeon C. Ntafos. An evaluation of random testing. *IEEE Trans. Softw. Eng.*, 10(4):438–444, July 1984.

[83] Simeon Ntafos. On random and partition testing. In *Proceedings of the 1998 ACM SIGSOFT international symposium on Software testing and analysis*, ISSTA '98, pages 42–48, New York, NY, USA, 1998. ACM.

[84] Todd L. Graves, Mary Jean Harrold, Jung-Min Kim, Adam Porter, and Gregg Rothermel. An empirical study of regression test selection techniques. *ACM Trans. Softw. Eng. Methodol.*, 10:184–208, April 2001.

[85] Rajiv Gupta, Mary Jean, Mary Jean Harrold, and Mary Lou Soffa. An approach to regression testing using slicing. In *In Proceedings of the Conference on Software Maintenance*, pages 299–308. IEEE Computer Society Press, 1992.

[86] Yih-Farn Chen, David S. Rosenblum, and Kiem-Phong Vo. Testtube: a system for selective regression testing. In *Proceedings of the 16th international conference on Software engineering*, ICSE '94, pages 211–220, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[87] M. Jean Harrold, Rajiv Gupta, and Mary Lou Soffa. A methodology for controlling the size of a test suite. *ACM Trans. Softw. Eng. Meth.*, 2:270–285, July 1993.

[88] J.A. Jones and M.J. Harrold. Test-suite reduction and prioritization for modified condition/decision coverage. *IEEE Trans. Softw. Eng.*, 29(3):195 – 209, march 2003.

[89] Gordon Fraser and Franz Wotawa. Redundancy based test-suite reduction. In *Proceedings of the 10th international conference on Fundamental approaches to software engineering*, FASE'07, pages 291–305, Berlin, Heidelberg, 2007. Springer-Verlag.

[90] Gregg Rothermel, Mary Jean Harrold, Jeffery Ostrin, and Christie Hong. An empirical study of the effects of minimization on the fault detection capabilities of test suites. In *Proceedings of the International Conference on Software Maintenance*, ICSM '98, pages 34–, Washington, DC, USA, 1998. IEEE Computer Society.

[91] Yanbing Yu, James A. Jones, and Mary Jean Harrold. An empirical study of the effects of test-suite reduction on fault localization. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 201–210, New York, NY, USA, 2008. ACM.

[92] Scott McMaster and Atif M. Memon. Fault detection probability analysis for coverage-based test suite reduction. In *ICSM*, pages 335–344. IEEE, 2007.

[93] Dan Hao, Lu Zhang, Xingxia Wu, Hong Mei, and Gregg Rothermel. On-demand test suite reduction. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 738–748, Piscataway, NJ, USA, 2012. IEEE Press.

[94] James A. Jones and Mary Jean Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, ASE '05, pages 273–282, New York, NY, USA, 2005. ACM.

[95] Zhihong Xu, Yunho Kim, Moonzoo Kim, Gregg Rothermel, and Myra B. Cohen. Directed test suite augmentation: techniques and tradeoffs. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, FSE '10, pages 257–266, New York, NY, USA, 2010. ACM.

[96] Gordon Fraser and Andrea Arcuri. Whole test suite generation. *IEEE Transactions on Software Engineering*, 99(PrePrints), 2012.

[97] Zhihong Xu and Gregg Rothermel. Directed test suite augmentation. In *Proceedings of the 2009 16th Asia-Pacific Software Engineering Conference*, APSEC '09, pages 406–413, Washington, DC, USA, 2009. IEEE Computer Society.

[98] Richard A. DeMillo and A. Jefferson Offutt. Constraint-based automatic test data generation. *IEEE Trans. Softw. Eng.*, 17(9):900–910, September 1991.

[99] Gordon Fraser and Andreas Zeller. Mutation-driven generation of unit tests and oracles. *IEEE Transactions on Software Engineering*, 38:278–292, 2012.

[100] Kunal Taneja, Tao Xie, Nikolai Tillmann, and Jonathan de Halleux. eXpress: guided path exploration for efficient regression test generation. In *ISSTA*, pages 1–11. ACM, 2011.

[101] G. Soares, R. Gheyi, and T. Massoni. Automated behavioral testing of refactoring engines. *Software Engineering, IEEE Transactions on*, PP(99):19, April 2012.

[102] Phil McMinn, Mark Harman, Kiran Lakhotia, Youssef Hassoun, and Joachim Wegener. Input domain reduction through irrelevant variable removal and its effect on local, global, and hybrid search-based structural test data generation. *IEEE Transactions on Software Engineering*, 38:453–477, 2012.

[103] Roger Ferguson and Bogdan Korel. The chaining approach for software test data generation. *ACM Trans. Softw. Eng. Methodol.*, 5(1):63–86, January 1996.

[104] Phil McMinn, Mark Harman, David Binkley, and Paolo Tonella. The species per path approach to searchbased test data generation. In *Proceedings of the 2006 international symposium on Software testing and analysis*, ISSTA '06, pages 13–24, New York, NY, USA, 2006. ACM.

[105] Daniel Kroening, Alex Groce, and Edmund Clarke. Counterexample guided abstraction refinement via program execution. In *Formal Methods and Software Engineering: 6th International Conference on Formal Engineering Methods*, pages 224–238. Springer, 2004.

[106] Jan Strejček and Marek Trtík. Abstracting path conditions. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, pages 155–165, New York, NY, USA, 2012. ACM.

[107] Joe W. Duran and Simeon C. Ntafos. An evaluation of random testing. *IEEE Trans. Software Eng.*, 10(4):438–444, 1984.

[108] Carlos Pacheco and Michael D. Ernst. Randoop: feedback-directed random testing for java. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, OOPSLA '07, pages 815–816, New York, NY, USA, 2007. ACM.

[109] Andrea Arcuri, Muhammad Zohaib Z. Iqbal, and Lionel C. Briand. Random testing: Theoretical results and practical implications. *IEEE Trans. Software Eng.*, 38(2):258–277, 2012.

[110] Raul Santelices and Mary Jean Harrold. Applying aggressive propagation-based strategies for testing changes. In *Proceedings of the 2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*, ICST '11, pages 11–20, Washington, DC, USA, 2011. IEEE Computer Society.

[111] Dawei Qi, Abhik Roychoudhury, Zhenkai Liang, and Kapil Vaswani. Darwin: an approach for debugging evolving programs. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC/FSE '09, pages 33–42, New York, NY, USA, 2009. ACM.

[112] Taweesup Apiwattanapong, Raul Andres Santelices, Pavan Kumar Chittimalli, Alessandro Orso, and Mary Jean Harrold. Matrix: Maintenance-oriented testing requirement identifier and examiner. In *Proceedings of theTesting and Academic Industrial Conference Practice and Research Techniques (TAIC PART 2006)*, pages 137–146, Windsor, UK, August 2006.

[113] Marcel Böhme. Software regression as change of input partitioning. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 1523–1526, Piscataway, NJ, USA, 2012. IEEE Press.

[114] Benny Godlin and Ofer Strichman. Regression verification: proving the equivalence of similar programs. *Softw. Test., Verif. Reliab.*, pages 1–18, March 2012.

[115] Sagar Chaki, Arie Gurfinkel, and Ofer Strichman. Regression verification for multi-threaded programs. In *VMCAI*, pages 119–135, 2012.

[116] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976.

[117] Jan Malburg and Gordon Fraser. Combining search-based and constraint-based testing. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ASE '11, pages 436–439, Washington, DC, USA, 2011. IEEE Computer Society.

[118] Tao Wang and Abhik Roychoudhury. Using compressed bytecode traces for slicing Java programs. In *ICSE*, pages 512–521, 2004.

[119] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *TACAS*, pages 337–340, 2008.

[120] A. Jefferson Offutt and Jie Pan. Automatically detecting equivalent mutants and infeasible paths. *Softw. Test., Verif. Reliab.*, 7:165–192, 1997.

[121] Kunal Taneja, Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Wolfram Schulte. Guided path exploration for regression test generation. In *ICSE Companion*, pages 311–314, 2009.

[122] Hyunsook Do, Sebastian G. Elbaum, and Gregg Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empir. Softw. Eng.*, 10(4):405–435, 2005.

[123] Shuvendu K. Lahiri, Chris Hawblitzel, Ming Kawaguchi, and Henrique Rebêlo. Symdiff: A language-agnostic semantic diff tool for imperative programs. In *CAV*, pages 712–717, 2012.

[124] Guowei Yang, Matthew B. Dwyer, and Gregg Rothermel. Regression model checking. In *ICSM*, pages 115–124, 2009.

[125] Steven Lauterburg, Ahmed Sobeih, Darko Marinov, and Mahesh Viswanathan. Incremental state-space exploration for programs with dynamically allocated data. In *ICSE*, pages 291–300, 2008.

[126] Junaid Haroon Siddiqui and Sarfraz Khurshid. Scaling symbolic execution using ranged analysis. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '12, pages 523–536, New York, NY, USA, 2012. ACM.

[127] Dirk Beyer, Thomas A. Henzinger, M. Erkan Keremoglu, and Philipp Wendler. Conditional model checking: a technique to pass information between verifiers. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 57:1–57:11, New York, NY, USA, 2012. ACM.

[128] Kin-Keung Ma, Khoo Yit Phang, Jeffrey S. Foster, and Michael Hicks. Directed symbolic execution. In *Proceedings of the 18th International Conference on Static Analysis*, SAS'11, 2011.

[129] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.

[130] Paul Dan Marinescu and Cristian Cadar. make test-zesti: a symbolic execution solution for improving regression testing. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 716–726, Piscataway, NJ, USA, 2012. IEEE Press.

[131] Crispin Cowan, Calton Pu, Dave Maier, Heather Hintony, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th conference on USENIX Security Symposium - Volume 7*, SSYM'98, pages 5–5, 1998.

[132] Chao Wang, Mahmoud Said, and Aarti Gupta. Coverage guided systematic concurrency testing. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 221–230, New York, NY, USA, 2011. ACM.

[133] Marcel Böhme, Bruno C.d.S. Oliveira, and Abhik Roychoudhury. Partition-based regression verification. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2013, pages 301–310, 2013.

[134] Yue Jia and Mark Harman. Higher order mutation testing. *Information and Software Technology*, 51(10):1379–1393, October 2009.

[135] C. Yilmaz. Test case-aware combinatorial interaction testing. *IEEE Transactions on Software Engineering*, 39(5):684–706, May 2013.

[136] Emine Dumlu, Cemal Yilmaz, Myra B. Cohen, and Adam Porter. Feedback driven adaptive combinatorial testing. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 243–253, 2011.

[137] Cristian Zamfir and George Candea. Execution synthesis: a technique for automated software debugging. In *Proceedings of the 5th European conference on Computer systems*, EuroSys '10, 2010.

[138] Alan J. Perlis. Special feature: Epigrams on programming. *SIGPLAN Not.*, 17(9):7–13, September 1982.

[139] Marcel Böhme, Bruno C. d. S. Oliveira, and Abhik Roychoudhury. Regression tests to expose change interaction errors. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 334–344, 2013.

[140] A. Jefferson Offutt. Investigations of the software testing coupling effect. *ACM Trans. Softw. Eng. Methodol.*, 1(1):5–20, January 1992.

[141] T.J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, 1976.

[142] Monica Hutchins, Herb Foster, Tarak Goradia, and Thomas Ostrand. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proceedings of the 16th International Conference on Software Engineering*, ICSE '94, pages 191–200, 1994.

[143] IEEE. Standard Glossary of Software Engineering Terminology. *IEEE Std 610.12-1990*, pages 1–84, 1990.

[144] David Hovemeyer and William Pugh. Finding bugs is easy. *SIGPLAN Not.*, 39(12):92–106, December 2004.

[145] IEEE. *1003.1-1988 INT/1992 Edition, IEEE Standard Interpretations of IEEE Standard Portable Operating System Interface for Computer Environments (IEEE Std 1003.1-1988)*. IEEE, New York, NY, USA, 1988.

[146] Kim Herzig and Andreas Zeller. The impact of tangled code changes. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 121–130, 2013.

[147] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, April 1978.

[148] George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. In *Proceedings of the 11th International Conference on Compiler Construction*, CC '02, pages 213–228, 2002.

[149] Charles Spearman. The proof and measurement of association between two things. *American Journal of Psychology*, 15:72–101, 1904.

[150] Jacob Cohen. A coefficient of agreement for nominal scales. *Educational and Psychological Measurement*, 20(1):37–46, 1960.

[151] J. M. Bland and D. G. Altman. Measuring agreement in method comparison studies. *Statistical Methods in Medical Research*, 8(2):135–160, 1999.

[152] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 402–411, 2005.

[153] James H. Andrews, Lionel C. Briand, Yvan Labiche, and Akbar Siami Namin. Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE Trans. Software Eng.*, 32(8):608–624, 2006.

[154] Akbar Siami Namin and Sahitya Kakarla. The use of mutation in testing experiments and its sensitivity to external threats. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 342–352, 2011.

[155] E.J. Weyuker. Evaluating software complexity measures. *IEEE Transactions on Software Engineering*, 14(9):1357–1365, 1988.

[156] S. Henry and D. Kafura. Software structure metrics based on information flow. *IEEE Transactions on Software Engineering*, SE-7(5):510–518, 1981.

[157] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, June 1994.

[158] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? In *Proceedings of the 2005 International Workshop on Mining Software Repositories*, MSR '05, pages 1–5, 2005.

[159] Sunghun Kim, Thomas Zimmermann, Kai Pan, and E. James Jr. Whitehead. Automatic identification of bug-introducing changes. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, ASE '06, pages 81–90, 2006.

[160] Valentin Dallmeier and Thomas Zimmermann. Extraction of bug localization benchmarks from history. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 433–436, 2007.

[161] Shan Lu, Zhenmin Li, Feng Qin, Lin Tan, Pin Zhou, and Yuanyuan Zhou. Bugbench: Benchmarks for evaluating bug detection tools. In *In Workshop on the Evaluation of Software Defect Detection Tools*, 2005.

[162] Jaime Spacco, Jaymie Strecker, David Hovemeyer, and William Pugh. Software repository mining with marmoset: An automated programming project snapshot and testing system. In *Proceedings of the 2005 International Workshop on Mining Software Repositories*, MSR '05, pages 1–5, 2005.

[163] Lucia, F. Thung, D. Lo, and Lingxiao Jiang. Are faults localizable? In *2012 9th IEEE Working Conference on Mining Software Repositories (MSR)*, pages 74–77, 2012.

[164] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. Semfix: Program repair via semantic analysis. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 772–781, 2013.

[165] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 3–13, 2012.