CHANGE-EFFECTS ANALYSIS FOR EFFECTIVE TESTING AND VALIDATION OF EVOLVING SOFTWARE

A Thesis
Presented to
The Academic Faculty

by

Raúl A. Santelices

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
College of Computing

Georgia Institute of Technology
August 2012

CHANGE-EFFECTS ANALYSIS FOR EFFECTIVE TESTING AND VALIDATION OF EVOLVING SOFTWARE

Approved by:

Dr. Mary Jean Harrold, Advisor
College of Computing
Georgia Institute of Technology

Dr. Santosh Pande
College of Computing
Georgia Institute of Technology

Dr. David Notkin
Computer Science and Engineering
University of Washington

Dr. Spencer Rugaber
College of Computing
Georgia Institute of Technology

Dr. Alessandro Orso
College of Computing
Georgia Institute of Technology

Date Approved: 25 April 2012

To my loving family and my great mentors and friends,

for their incredible support and understanding

ACKNOWLEDGEMENTS

The completion of my Ph.D. in Computer Science is a dream made true thanks to the wonderful experiences, lessons, and opportunities I have been so fortunate to receive throughout my entire life. The people responsible for shaping who I am are, first of all, my dear parents, Mónica Ahués and Raúl Santelices II, and my sweet sister Daniela. I owe them my eternal gratitude for their support, love, nourishing, and encouragement.

A very special place in my heart is reserved for my sweet, loving wife, Sara Fuchs. My greatest blessing in my years in Atlanta working on my Ph.D. was to find her. Sara has been an angel with her incredible support and understanding, giving me the strength to make the final push to complete this dissertation—by far the most difficult part. I could not have made it without her. I give Sara much more credit than she wants to take.

During my studies and professional career in Chile, I had the honor and fortune to interact with more classmates, mentors, and colleagues than I can possibly list here. These marvelous people had, in one way or another, a profound impact in my career and eventually in my decision to go for the Ph.D. degree. I want to give special thanks to the following people: Drs. Miguel Nussbaum and Patricio Rodríguez, who gave me the opportunities and support that led me to discover and fulfill my academic interests; professors Jaime Navón and Yadran Eterovic, for their support and encouragement; my classmates, colleagues, and friends who provided me with the most stimulating environment, among whom I count Cristian Rodríguez, José Pablo Zagal, Sandra Suárez, Ximena López, Rodrigo Marchant, Jorge Villalón, Carlo Fabricatore, Amparo Minaya, Marcela Salinas, Patricia Flores, Valeria Valdivia, and Florencia Gómez; and my personal friends Ángel Abusleme, Álvaro Delgado, Pablo Baltera (R.I.P.), Gustavo Olivares, and Álvaro Cortés.

The United States received me in 2005 as my second home. I have met here another

bunch of fantastic people who have taught me and shared with me so much. Among them, the figure of my advisor, professor Mary Jean Harrold, stands out. I simply fall short in words to express how lucky and privileged I am that Mary Jean took me under her wing and led me through all these years with admirable wisdom, patience, generosity, work ethic, and dedication. I cannot imagine having a better advisor and role model.

One of the wonders that Mary Jean has produced is her group at Georgia Tech: the Aristotle Research Group (ARG) that I also know as my "academic family". Among my lab-mates and friends at the ARG who provided me with such a stimulating and fun environment are: James A. Jones, Taweesup (Term) Apiwattanapong, James F. Bowring, Pavan K. Chittimalli, Carsten Goerg, Saswat Anand and his wife Aliva Pattnaik and their son Rrishi, George K. Baah, Hina B. Shah, Sangmin Park, Yanbing Yu, Wanchun (Paul) Li, Sarah Clark, Jake Cobb, Mijung Kim, Chaitanya P. Namburi, Ahmed Sayed Ahmed, Shujuan Jiang, Yuxia (Sabrina) Sun, Rashmitha Chittimalli, Heena Macwan, and Jai Kejriwal.

But not only ARG gave me the tools and lessons that let me complete this dissertation and take off in my academic career. I also owe my gratitude to professor Alessandro (Alex) Orso for his cheerful and complementary visions and advice, and to professor David Notkin, for his unparalleled enthusiasm, encouragement, and generosity, even in the middle of serious health difficulties. I also want to thank professors Santosh Pande, Spencer Rugaber, Andy Podgurski, H. Venkateswaran, Santosh Vempala, and Nate Clarke, and my fellow students and friends James Clause, William G. J. Halfond, Christoph Csallner, Shauvik Roy Choudhary, Chris Parnin, Eli Tilevich, and Arjan Seesing.

Finally, I want to give my special thanks to the University of Notre Dame, its College of Engineering, and the Computer Science and Engineering department for their understanding, support, and encouragement while I was working with them and at the same time completing this dissertation. I am particularly indebted to Kevin Bowyer, Peter Kilpatrick, M. Brian Blake, John Stewman, Greg Madey, Scott Emrich, Aaron Striegel, Jesús Izaguirre, and so many others at this wonderful institution.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

The constant modification of software during its life cycle poses many challenges for developers and testers because changes might not behave as expected or may introduce erroneous side effects. For those reasons, it is of critical importance to analyze, test, and validate software every time it changes.

The most common method for validating modified software is *regression testing*, which identifies differences in the behavior of software caused by changes and determines the correctness of those differences. Most research to this date has focused on the efficiency of regression testing by selecting and prioritizing existing test cases affected by changes. However, little attention has been given to finding whether the test suite adequately tests the *effects* of changes (i.e., behavior differences in the modified software) and which of those effects are missed during testing. In practice, it is necessary to *augment* the test suite to exercise the untested effects.

The thesis of this research is that the effects of changes on software behavior can be computed with enough precision to help testers analyze the consequences of changes and augment their test suites. To demonstrate this thesis, this dissertation uses novel insights to develop a fundamental understanding of how changes affect the behavior of software. Based on these foundations, the dissertation defines and studies new techniques that detect these effects in cost-effective ways. These techniques support test-suite augmentation by (1) identifying the effects of individual changes that should be tested, (2) identifying the combined effects of multiple changes that occur during testing, and (3) optimizing the computation of these effects.

CHAPTER I

INTRODUCTION

The constant modification of software during its life cycle creates serious challenges for developers and testers because changes might not behave as expected or may introduce erroneous side effects. For example, changing the type of a collection of items from a set to a list can lead to undesired duplicate items. For another example, adding minimum-age constraints to a banking system can prevent children from monitoring their accounts even if only parents can perform sensitive operations.

When developers modify a program, they must check the correctness of their changes by identifying and analyzing the effects of those changes on the behavior of the program. To check their changes, developers typically perform *regression testing*, which is the activity of re-testing a program after it is modified (e.g., [12,17,49,53,54,65,66,69,77,88,89,115]). Most research on regression testing has focused on the *efficiency* of this activity through approaches such as (1) selecting a subset of the test cases that need to be re-run (e.g., [25,88]), (2) prioritizing test cases to find errors earlier (e.g., [90, 103]), and (3) reducing the size of the test suite according to some criterion (e.g., [52,114]). However, little attention has been given to the *effectiveness* of regression testing, which requires not only identifying existing test cases affected by changes, but also understanding how changes affect the behavior of a program and what new test cases are needed to exercise new behaviors. Finding as many differences as possible caused by changes in the behavior of a program is a necessity to determine whether those changes are correct or not.

Change-testing approaches in the literature determine new test-coverage requirements for a change by identifying elements, such as statements or branches, that might be affected

by the change and, therefore, need to be covered during testing [17,49,87]. These *coverage-based* techniques, however, consider only the location of the change and ignore how the program state is modified. Because they ignore important information about the effects of a change, these techniques are too conservative and therefore imprecise, as studies of related analyses suggest [18, 23]. Furthermore, these change-testing approaches had not been evaluated empirically prior to this dissertation.

A related class of techniques that explicitly search for effects of a change is *change-impact analysis* (e.g., [5, 7, 13, 20, 21, 46, 68, 70, 75, 76, 81, 83]), which identifies parts of the software that are potentially affected by a change. Developers can use change-impact analysis to decide which existing tests to re-run during regression testing and which new tests to create for impacted components. Unfortunately, existing change-impact analysis techniques are too coarse grained or too imprecise, or both. Coarse-grained impact analyses identify affected classes or methods (e.g., [5,21,83]), which can be useful as a first step for developers to understand the consequences of a change, but do not describe which parts of those classes or methods are affected or how they are affected. In contrast, fine-grained impact analyses that use code dependencies (e.g., [16, 57]) are too imprecise because they either find, in a conservative way, an excessive portion of the program as potentially affected or miss affected code not revealed as impacted by a specific set of executions.

An additional challenge for the analysis of changes stems from the more general and typical case in which multiple changes are made to software. In this context, analyzing each change individually is often inadequate because developers make sets of changes in a program to either accomplish a common goal or perform separate, independent tasks. In the first case, developers need to check that changes interact as expected. In the second case, developers need to ensure that changes do not interfere with each other unexpectedly. This second case is particularly important when multiple developers work in parallel and then have to merge safely their respective changes. Existing approaches for determining the interaction of changes, however, face similar problems to the analysis of single changes:

they either use coarse representations of changes [83, 113] (e.g., modified methods) or rely on imprecise analyses [19, 56] (e.g., all possibly affected statements).

The existing research and its limitations indicate that the problem of analyzing and testing changes presents serious challenges but also unique opportunities. One main observation from this problem is that the analysis of changes can exploit not only the knowledge of the *location* of the changes (i.e., which parts of the program are modified), as most existing techniques do, but also their *semantics* (i.e., how the program state is modified by changes and how those state modifications propagate throughout the program). Because existing research on change testing has not fully exploited this semantic knowledge, the following question arises: Can the semantics—also called *effects*—of the changes be analyzed in enough detail for assessing and testing changes? The thesis statement of this dissertation suggests a positive answer to this question.

1.1 Thesis Statement

The thesis of this dissertation is that the effects that changes have on the behavior of software can be analyzed automatically and with enough precision to help developers discover, exercise, and assess the observable consequences of changes in a cost-effective manner.

1.2 Overview of the Dissertation

To support this thesis, this dissertation presents (1) comprehensive empirical evaluations of existing approaches for testing changes, (2) new foundations for analyzing the effects of changes, (3) new techniques for analyzing and testing changes based on these foundations, and (4) a set of studies that validate the new techniques and indicate that they are significantly more effective than existing approaches for change analysis and testing. Chapter 2 provides the necessary terminology and background to understand these contents.

At the heart of this dissertation lies a novel class of program analyses, called *change-effects analysis* and presented in Chapter 3, that compute with unprecedented precision

the effects of changes—how program states and control flow differ because of changes. Change-effects analysis determines which behaviors of the program are affected and which new behaviors are introduced by changes. Developers can use this information to assess effectively how well a test suite exercises the differences in behavior, which differences are still untested, and how changes interact with each other to produce those differences.

The computation of change effects, however, might come at a considerable cost because a large or even infinite number of program paths might have to be analyzed. To address this problem, Chapter 4 presents a new approach for path-sensitive analysis of software that can alleviate considerably the cost of change-effects analysis and other important software-engineering tasks. This approach exploits dependencies in the program code to improve the cost-effectiveness of path analysis. Although the full computation of change effects remains an intractable problem in general, this new approach can increase dramatically the extent of the analysis that can be performed for practical approximations of change effects presented in the chapters that follow.

A major application of change-effects analysis is *test-suite augmentation* for evolving software[1] [6, 92, 96], which is the two-step process of assessing an existing test suite and adding new test cases for the untested effects of the changes. Figure 1 shows the augmentation process in which program P, which is tested by test suite T, is modified to obtain version P', for which an augmented test suite T' is obtained. Developers first assess how well T and the test cases T^+ that developers add during the modification of P test the effects of the changes in P' (Step 1). Then, the developers add new test cases to $T \cup T^+$ to exercise the untested effects (Step 2). Building on the foundations and techniques from Chapters 3 and 4, this dissertation presents new test-suite augmentation techniques in Chapters 5 and 6. These new techniques are much more effective than existing change-testing approaches, as the experiments described in these chapters show. The techniques work by formulating and monitoring change-testing requirements that developers use to

[1] *Evolving software* refers to software that undergoes modifications over time.

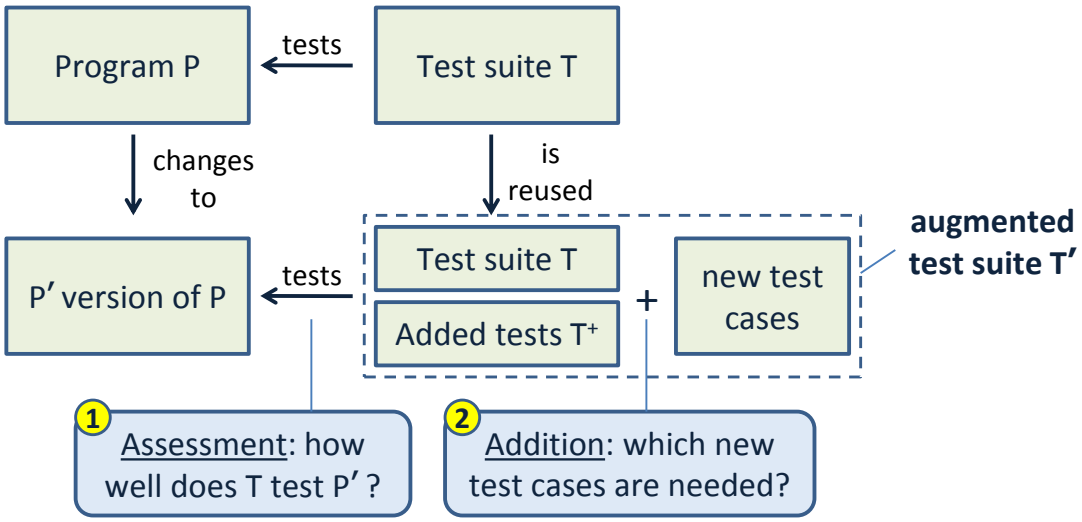


Figure 1: Test-suite augmentation process for evolving software.

augment their test suites. Satisfying these testing requirements guarantees that the effects of changes propagate within the program, and, eventually, produce new differences in its output to help developers assess the correctness of the changes.

Another important application of change-effects analysis is the precise detection of interactions among multiple changes. Chapter 7 presents an exact definition of change interaction based on the fundamental concepts of Chapter 3 and develops a new technique that determines precisely whether and where changes interact with each other. The empirical results presented in that chapter demonstrate not only the effectiveness of the new technique but also reveal that fine-grained approximations obtained by existing analyses, usually regarded as state of the art, are too imprecise for detecting change interactions.

Finally, Chapter 8 provides a conclusion to this dissertation, an analysis of its merits, and a discussion of future research directions that this dissertation opens.

1.3 Contributions

This dissertation makes the following contributions to the fields of software engineering and program analysis:

1. Principled foundations for describing precisely how changes in program code affect the behavior of modified software.

2. A new class of precise analyses, called *change-effects analysis* and based on these foundations, that identify the effects that a change has or can have on the behavior of a program.

3. A new supporting program analysis that speeds up change-effects analysis and potentially other important software-engineering techniques.

4. New and cost-effective test-suite-augmentation approaches that, based on these analyses, identify and monitor testing requirements for changes.

5. A new technique, also based on these analyses, that identifies whether and when changes interact with each other in a program execution. This technique lets developers decide whether to merge changes and shows which interactions are tested.

6. Empirical evidence of the effectiveness of these techniques and the mechanisms for controlling their cost.

CHAPTER II

BACKGROUND

This dissertation builds on the areas of program analysis, fault propagation, and regression testing, and this chapter provides the necessary definitions and background in these areas. Section 2.1 describes program analysis. Program analysis identifies properties of program code needed to calculate the effects of changes. Section 2.2 presents fault propagation models that describe the conditions under which the effects of faults propagate through the code. These models can be directly applied to the effects of changes as well. Finally, Section 2.3 discusses background on regression testing, which is the process within which the analysis of changes usually takes place.

2.1 Program Analysis

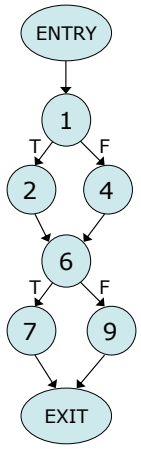
Several program analysis techniques are needed to identify the elements and relationships in program code that can be affected by changes. Control-flow analysis (Section 2.1.1) builds graph representations of all possible sequences of statements in program executions. Control- and data-dependence (Section 2.1.2) analyses identify the dependencies among statements that arise from control decisions and computations in the program. Slicing (Section 2.1.3) uses these dependencies to identify the subset of a program or execution that affects or is affected by a certain control decision or computation. Finally, symbolic execution (Section 2.1.4) produces an algebraic representation of the cumulative effect on the program state of statements in a path or set of paths of the program.

2.1.1 Control Flow Analysis

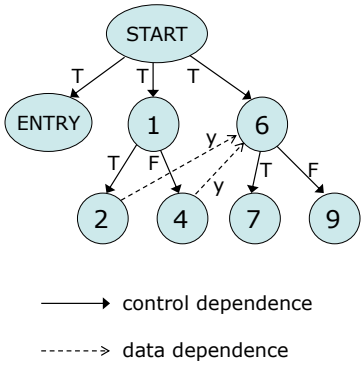
In this research, a statement is a line of *executable* code (i.e., code that can modify a program state or path). Without loss of generality, we assume that a statement contains at most

```
program E(int x, y) // x,y ∈ [1,10]
1.  if (x <= 2) // ch1: if (x > 2)
2.     ++y
3.  else
4.     --y
5.  // ch2: y *= 2
6.  if (y > 2)
7.     print 1
8.  else
9.     print 0
```

(a)



(b)



(c)

Figure 2: Example program E with two changes (a), its CFG (b), and its PDG (c).

one procedure call (i.e., function or method call). A statement that contains a procedure call is a *call site*. The executable code of a program procedure can be represented by its control-flow graph [2], defined next.

DEFINITION 1. A *control-flow graph* (CFG) for a procedure p is a connected directed graph[1] $G{=}(V, E)$ where V is the set of nodes and E is the set of edges. For each statement in p other than a call site, V contains a node. For each call site in p, V contains a call node and a return node that represent the points before and after the call, respectively. Each of these nodes is labeled with a unique identifier (e.g., the line number) for the corresponding statement, followed by a suffix 'a' if it is a call node and 'b' if it is a return node. V also contains two distinguished nodes *EN* (or *ENTRY*) and *EX* (or *EXIT*) for the entry to and exit from p, respectively. In V, only *EN* has no predecessors and only *EX* has no successors. The edges in E represent the flow of control among the elements in p represented by the nodes in V. For each call site in p, E contains an edge from the corresponding call node to the corresponding return node. If p has multiple exit statements, E contains an edge from each node corresponding to an exit statement to node *EX*. An edge e whose source node

[1] A directed graph is connected if, for all pairs of nodes (a, b), there is a path from a to b or from b to a.

has multiple outgoing edges is labeled with the control decision that causes e to be taken.

To illustrate, consider the single-procedure program E in Figure 2(a) and its CFG graph in Figure 2(b). E has nine statements, six of which correspond to executable statements (the empty statement 5 and the `else` statements 3 and 8 are ignored) represented by nodes in the graph. Node 1, for example, is a conditional statement with two outgoing edges: the true branch (1,2) labeled "T", and the false branch (1,4) labeled "F". Node *ENTRY* represents the entry to the procedure and node *EXIT* represents the exit from the procedure.

A CFG is suitable for *intraprocedural* (within procedures) analyses. However, different procedures in a program interact via procedure calls at call sites; such interactions need to be modeled for an *interprocedural* (across procedures) analysis. Therefore, to analyze programs with multiple procedures, researchers have developed graph representations such as *supergraphs* [84] and *interprocedural* control-flow graphs (ICFG) [101]. In this dissertation, we use the ICFG representation.

DEFINITION 2. An *interprocedural control-flow graph* (ICFG) for a program P is a connected directed graph G^P=(V^P, E^P) where V^P is the union of the node sets of the CFGs of all procedures of P and $E^P = (E_{cfgs} - E_{a,b}) \cup E_{inter}$, where

- E_{cfgs} is the union of all edge sets of the CFGs of all procedures of P,
- $E_{a,b}$ is the set of all edges connecting call nodes to return nodes, and
- E_{inter} is the set of all call edges (a, EN_p) and return edges (EX_p, b) connecting the call and return nodes for all call sites in P to the procedures p that they can call.

A call site that uses a function pointer or performs a virtual call might have more than one target procedure. In this research, a call site whose call node has more than one outgoing call edge in E_{inter} is treated as a branching node and each call edge from that node is labeled with the signature of the target procedure. Also, procedures might terminate abnormally by halting or throwing an exception, which creates special control dependencies [101]. These and any other kinds of control flow are represented by CFGs and ICFGs.

Not all paths in the ICFG of a program correspond to paths in that program—only *realizable* paths do.

DEFINITION 3. A *realizable path* in the ICFG of a program is a path for which every call edge (a, EN_p) is matched by its corresponding return edge (EX_p, b).

Henceforth, we use the term *path* to refer only to realizable paths in ICFGs. Two other necessary control-flow concepts are *dominance* and *post-dominance*.

DEFINITION 4. A node m in a CFG or ICFG G *dominates* a node n in G if and only if every path from every entry node of G to node n contains m.

DEFINITION 5. A node m in a CFG or ICFG G *post-dominates* a node n in G if and only if every path from n to every exit node of G contains m.

2.1.2 Program Dependencies

The control decisions and the manipulation of variables in the statements of a program give rise to dependencies among those statements. If a statement s_1 is dependent on a statement s_2, we also say that a node n_1 for s_1 in a CFG or ICFG G is dependent on a node n_2 for s_2 in G. Dependencies created by control decisions are called *control* dependencies [41].

DEFINITION 6. A statement s_1 is *control-dependent* on a statement s_2 with label L if s_2 has a successor u (along an edge labeled L) and a successor v such that s_1 post-dominates u but s_1 does not post-dominate v.

In other words, a statement s_1 is control dependent on a statement s_2 if s_2 has two or more outgoing edges and, for at least one but not all those edges, s_1 necessarily executes after taking that edge. If so, the control (execution) of s_1 depends on the decision at s_2.

A control dependence can be written as (a, b) or (a, b, L) (the latter form if the decision matters), where b is control dependent on a for decision L. To illustrate, consider program E in Figure 2(a) and its CFG in Figure 2(b). In this program, statement 2 is control-dependent on statement 1 for decision T, written as $(1, 2, \mathrm{T})$.

Dependencies created by accesses to program variables are called *data* dependencies [2].

DEFINITION 7. A statement s_1 is *data dependent* on a statement s_2 with label v if: (1) s_2 defines (assigns a value to) a variable v; (2) there is a *definition-clear* path for v from s_2 to s_1 (i.e., a path containing no other definition of v); and (3) s_1 uses (reads) v. The pair of definition and use in a data dependence is called *definition-use pair*, or simply *du-pair*.

A data dependence can be written as (a, b) or (a, b, v), where statement b is data-dependent on statement a for variable v. For example, in Figure 2, statement 6 is data dependent on statement 2 for variable y, represented as $(2, 6, y)$.

Podgurski and Clarke defined *semantic* dependence [79] among statements.

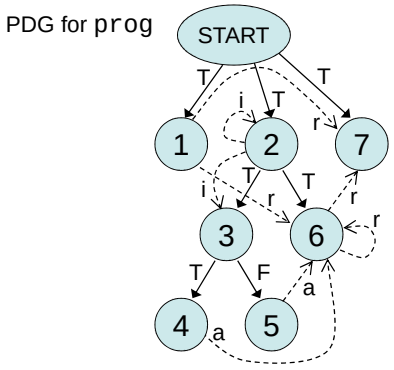
DEFINITION 8. A statement s_1 is *semantically dependent* on a statement s_2 if, for some execution of the program, a change can be made to the value computed at s_2 that causes a change in the values used by s_1 or in the number or location of the occurrences of s_1.

Based on this definition of dependence of s_1 on s_2, we can also say that statement s_1 is semantically dependent on a particular change C in statement s_2 that affects s_1 in at least one execution. Semantic dependencies, however, represent an ideal but not always practical goal of analysis. For instance, finding whether a sequence of dependencies (control or data) is also a semantic dependence is an undecidable problem.

DEFINITION 9. A *program dependence graph* (*PDG*) [41] for a program P is a graph $D{=}(V^D, E^D)$ where V^D contains one node for each node in the ICFG of P and E^D is the set of edges representing the control and data dependencies among the statements of P that the corresponding nodes represent. Each control-dependence edge is labeled with the control decision for which the target node is dependent on the source node. Each data-dependence edge is labeled with the variable that the source node defines and the target node uses. V^D also contains a special node *START*. For every node n in the ICFG of P that is not control dependent on any other node, there is a control-dependence edge with label T (true) from *START* to n.

```
prog(int n, int m[])
int r, i, a
1.  r = n
2.  for (i=1 to 4) {
3.     if (m[i] > 0)
4.        a = 4
      else
5.        a = 2
6.     r += a
   }
7.  print r > 0
```

(a)



(b)

Figure 3: Example program `prog` (a) and its PDG (b).

Figure 2(c) shows the PDG for the example program E, in which solid edges, labeled with the corresponding decisions, represent control dependencies and dashed edges, labeled with the corresponding variables, represent data dependencies.

It is also possible—and useful—to define partial versions of PDGs restricted to one kind of dependence only.

DEFINITION 10. An *interprocedural control-dependence graph* (*ICDG*) [101] for a program P is the PDG for P without the data-dependence edges.

2.1.3 Slicing

This section describes *slicing* from the perspective of the potential effects of changes on the rest of the program. Figure 3(a) shows the example program `prog` used in this section to illustrate slicing. Figure 3(b) shows the PDG for `prog`. This program inputs an integer `n` and an array of integers `m`, and initializes `r` to `n` at statement 1. In the loop in statements 2–6, the program iteratively determines the value of `a`—depending on the value of `m[i]`—and adds `a` to `r`. Finally, at statement 7, the program outputs `1` if `r > 0` or `0` otherwise.

To determine which parts of the program may be affected by a change in a statement, the static forward slice from that statement can be used.

Table 1: Static and dynamic forward slices for sample statements from `prog` in Figure 3.

statement	**static slice**	**dynamic slice (first occurrence)** input: `n=5, m=[1,-1,0,3]`
1	1, 6, 7	1^1,6^1,6^2,6^3,6^4,7^1
4	4, 6, 7, 8	4^1,6^1,6^2,6^3,6^4,7^1,8^1
7	7	7^1

DEFINITION 11. The *static forward slice* [55, 57, 112] from a statement s in a program P is the set containing s and the statements for all nodes directly or transitively reachable along control and data dependencies in the PDG of P from the node for statement s.

To illustrate, consider program `prog` and its PDG in Figure 3. Table 1, in its second column, shows the static forward slices for three different nodes in `prog`. For example, the static forward slice from node 1 consists of $\{1, 6, 7\}$.

To determine for a particular execution which statements may be affected by a change in statement s, a dynamic forward slice can be computed from s in that execution.

DEFINITION 12. The *dynamic forward slice* [16, 55, 57] from a statement s for an execution E of program P is the set containing s and the statements for all nodes directly or transitively reachable from the node for s along control and data dependencies in the PDG of program P for execution E.

In this dissertation, we use the finest-grained form of dynamic slice, which is based on *statement occurrences* [1]: the i^{th} occurrence of statement s in the execution is denoted s^i. The third column in Table 1 shows the dynamic forward slice for the first occurrence of the corresponding statement (first column) in the execution of `prog` on input $\{$`n=5, m=[1,-1,0,3]`$\}$. (Note that, for `prog` and this input, the execution history is 1^1,2^1,3^1,4^1,6^1,2^2,3^2,5^1,6^2,2^3,3^3,5^2,6^3,2^4,3^4,4^2,6^4,7^1,8^1.) For example, the dynamic forward slice for statement occurrence 4^1 consists of the set $\{4^1,6^1,6^2,6^3,6^4,7^1,8^1\}$.

2.1.4 Symbolic Execution

Symbolic execution [27, 63] analyzes a program by executing it with symbolic inputs along some program path. Symbolically executing all paths in a program to a given point, if feasible, provides a description of the semantics of the program up to that point. Symbolic execution represents the values of program variables at any given point in a program path as algebraic expressions by interpreting the operations performed along that path on the symbolic inputs. The *symbolic state* of a program at a given point consists of the set of symbolic values for the variables in scope at that point. The set of constraints that the inputs must satisfy to follow a path is called a *path condition* and is a conjunction of constraints p_i or $\neg p_i$ (depending on the branch taken), one for each predicate traversed along the path. Each p_i is obtained by substituting the variables used in the corresponding predicate with their symbolic values. Symbolic execution on all paths to a program point represents the set of all possible states at that point as a disjunction of clauses, one for each path that reaches the point. These clauses are of the form $PC_i \Rightarrow S_i$, where PC_i is the path condition for path i and S_i is the symbolic state after executing path i.

To illustrate, consider program E in Figure 2. Symbolic execution first assigns symbolic values x_0 and y_0 to inputs x and y, respectively. When statement 1 is executed, the technique computes the path conditions for the *true* and *false* branches, which are $x_0 \leq 2$ and $x_0 > 2$, respectively. These conditions are shown in column *path condition* of Table 2, for statements 2 and 4. The values of variables at the entry of each statement are shown in column *symbolic state*. For example, after evaluating statement 2, the technique updates the value of y to $y_0 + 1$. The execution of the remaining statements is performed analogously. Each row in Table 2 shows the path conditions and the symbolic values at the entry of the corresponding statement, after traversing all paths to that statement. For example, there are two path conditions and symbolic states for statement 7. Each path condition in the second column implies the symbolic state on its right in the third column. Symbolic execution associates with each statement the disjunction of all rows for that statement, where each

Table 2: Symbolic execution for program E in Figure 2

statement	path condition	symbolic state
1	*true*	$x = x_0, y = y_0$
2	$x_0 \leq 2$	$x = x_0, y = y_0$
4	$x_0 > 2$	$x = x_0, y = y_0$
6	$x_0 \leq 2$	$x = x_0, y = y_0 + 1$
	$x_0 > 2$	$x = x_0, y = y_0 - 1$
7	$(x_0 \leq 2) \wedge (y_0 > 3)$	$x = x_0, y = y_0 + 1$
	$(x_0 > 2) \wedge (y_0 > 1)$	$x = x_0, y = y_0 - 1$
9	$(x_0 \leq 2) \wedge (y_0 \leq 3)$	$x = x_0, y = y_0 + 1$
	$(x_0 > 2) \wedge (y_0 \leq 1)$	$x = x_0, y = y_0 - 1$

row represents a path condition and its corresponding symbolic state.

2.2 Fault Propagation Models

A number of researchers have worked on theoretical aspects of fault propagation for *fault-based* testing. Fault-based testing [34, 51] restricts testing to a class of faults and tests the software for the absence of faults of that particular kind. Typically, the faults considered are simple transformations of code at specific locations. There are two main assumptions that justify this testing approach: the *competent programmer hypothesis* [34], which states that developers create programs that are syntactically close to a hypothetical correct version, and the *coupling effect* [34], which states that test data that reveals simple faults is also sensitive enough to uncover more complex faults. The theoretical work on fault-based testing described in this section can be naturally applied to changes. Changes, like faults, can be seen as code transformations; the only difference between changes and faults is that the location and semantics of changes are known.

Morell [71] developed a theory to understand and systematize *fault-based* testing. Fault-based testing restricts testing to a class of faults and tests the software for the absence of these faults. Morell's theory provides a framework where classes of faults are specified as sets of transformations (alternative expressions) on different locations in the code and establishes as the goal of testing the selection of input data that differentiate all alternative

programs produced by those transformations from the original program. To achieve this goal, Morell proposed *symbolic testing*, in which transformations are symbolic (i.e., they represent potentially infinite alternatives), the original program and its alternatives are represented as symbolic functions of the input, and the condition for test inputs to distinguish an alternative from the original program is expressed as a *propagation assertion* in which the symbolic functions are required to differ. Although symbolically evaluating complete programs is impractical, Morell's theory makes symbolic testing an ideal formalism for developing and analyzing less complex, approximate approaches.

Richardson and Thompson [85] proposed the RELAY model of fault detection, which refines Morell's theory by describing in detail the different steps involved in the execution and propagation of a fault to cause an observable failure. These steps include the origination of a potential failure (i.e., an erroneous state) when executing a faulty component, the computational transfer of this failure to the containing expression, and the transfer from computation to computation of faulty states via *information flows* (i.e., chains of control and data dependencies) until a failure is revealed to an external observer. The authors used this model of failure origination and transfer conditions to identify one of the main weaknesses of traditional testing criteria based on coverage of program entities (e.g., branches, data dependencies) [86], which is their inability to address *coincidental correctness*—the case in which a fault executes but the error is not transferred to the output. In Reference [105], the same authors explored in detail the transfer conditions in the RELAY model and acknowledged that symbolic execution is needed to provide accurate transfer conditions, which, in general, makes the process too complex. Nonetheless, the RELAY model gives additional insights to Morell's theory for developing approximations of fault (and change) propagation.

Voas also used the notions of origination and transfer of erroneous states in his PIE model [109]. In this model, the testing criterion for a fault should ensure that the fault is executed (E), that it infects the state (I), and that the infected state propagates to the output

(P). Rather than specify and solve these conditions in detail, however, Voas used this model to construct a technique that estimates the probability that certain program locations cause observable failures if they contain faults. Nevertheless, the PIE model is very convenient to succinctly refer to the main steps of the RELAY model.

The three models described in this section were designed for single faults [71,86,109] or multiple faults that do not interfere with each other [86]. In general, however, developers make multiple changes to software before re-testing it. To the best of our knowledge, no fundamental model exists that describes how the effects of multiple faults or changes interact with each other and propagate in combination to the output.

2.3 Regression Testing and Test-suite Augmentation

Regression testing is an important task in software testing that consists of re-testing software after it is modified [12, 17, 45, 49, 53, 54, 65, 66, 69, 72, 77, 88, 89, 115] to discover any *regressions* (i.e., program behaviors that stop working correctly) introduced by changes. Typically, an existing test suite is re-used to perform regression testing. Also, because changes might introduce new features or exhibit side effects, new test cases might be needed. Within regression testing, we call *test suite augmentation* [6, 92] the two-step process that assesses the existing test suite and adds new test cases as needed. Figure 1 illustrates the augmentation process.

Most research on regression testing has focused on *efficiency* aspects that reduce the cost of re-running the test suite (e.g., [25, 52, 88, 114]) and increase the probability of finding regression errors early in the process (e.g., [90, 103]). Research on efficiency includes the analysis of correlations between deviations in *program spectra* (i.e., profiling information) and regression faults (e.g., [54, 115]) to better identify test cases that need to be re-run.

Regarding the *effectiveness* of regression testing and, in particular, test-suite augmentation, Section 2.3.1 addresses coverage-based testing techniques for whole programs that inspire change-specific testing approaches and Section 2.3.2 describes existing research on

such change-testing approaches.

2.3.1 Coverage-based Testing Criteria for Whole Software

A large body of work has addressed software testing based on white-box coverage criteria aimed at discovering faults hiding anywhere in an implementation. These criteria define which entities in the program must be *covered* (i.e., executed at least once) during testing [12, 45, 72]. A *test suite* (i.e., a collection of test cases) for a program is considered *adequate* for a white-box testing criterion *C* if it covers all entities in that program required by *C*. Some testing criteria require that all control-flow entities of some kind [58, 59, 72] (e.g., statements, branches, paths) are covered, while other criteria require the coverage of all data-flow entities of some kind [12, 45, 67, 82] (e.g., definition-use pairs, chains of data dependencies). Frankl and Weyuker [45] provided a hierarchy of the main control- and data-flow testing criteria that shows the subsumption relationships among these criteria.

2.3.2 Coverage-based Testing Criteria for Modified Software

Whole-program coverage criteria aims at discovering faults whose existence and location are unknown. In the case of modified software, however, the changes and their location are known. Researchers have exploited this information at least partially by proposing to test the effects of changes by conservatively identifying entities, such as branches or data dependencies, that might be affected by the changes and requiring the coverage of these entities by a regression test suite [17, 49, 87]. Affected entities are identified in these techniques via forward traversal of control and data dependencies from the locations affected by changes—the approach corresponding to *forward* program slicing [55, 57, 112]. Thus, these testing techniques use specialized coverage criteria that require the coverage of only affected subsets of program entities instead of all such entities, as whole-program criteria do (Section 2.3.1). The effectiveness of these change-testing techniques, however, has not been evaluated empirically prior to this dissertation.

CHAPTER III

FOUNDATIONS OF CHANGE-EFFECTS ANALYSIS

This chapter provides the foundations for the automatic analysis of the effects of changes. While being a contribution for any research on software changes, these foundations serve in particular as the core components of the change analysis and testing techniques presented in Chapters 5, 6, and 7.

In this chapter, Section 3.1 formally defines the effects of a change, Section 3.2 shows how to compute these effects, and Section 3.3 discusses related work.

3.1 Formal Model of the Effects of Changes

This section presents a formal model of program-code changes and their effects. The guiding principle for this model is that

> *A change C in a program affects an element e of that program if applying C to the program alters in any way the behavior of e.*

Element e can be a statement or even another change. The behavior of a change is the set of differences caused by the change in the behavior of the changed statements.

This section formalizes the concepts of "changes" in program code and the "effects" of those changes. Section 3.1.1 presents the example used to illustrate the model, Section 3.1.2 defines what we mean by a change in code, and Sections 3.1.3 and 3.1.4 present the model for individual and multiple changes, respectively.

3.1.1 Working Example

Figure 4 shows the example program `prog`, originally presented in Figure 3, with four changes shown as comments: `ch1`, `ch2`, `ch3`, and `ch4`. Applying these changes to `prog`

```
prog(int n, int m[]) // → prog'
int r, i, a
1.  r = n       // ch1:  r = n - 1
2.  for (i=1 to 4) {
3.     if (m[i] > 0)
4.        a = 4  // ch2:  a = 5
       else
5.        a = 2
6.     r += a
   }
7.  print r > 0  // ch3: print r > -10
8.  /* nothing */ // ch4:  print a
```

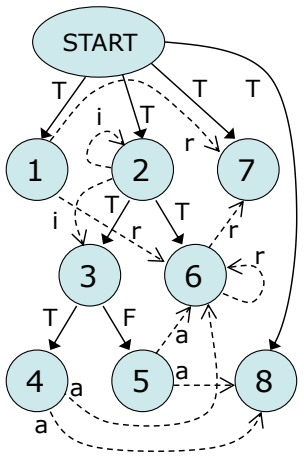


Figure 4: Example program `prog` with four changes that produce version `prog'`, and the PDG for `prog'`.

produces the modified version called `prog'`. To the right of the program, Figure 4 shows the PDG for `prog'`. The PDG for `prog` is identical except for the absence of node 8 and edges (4,8,a) and (START,8). Recall that `prog` inputs an integer `n` and an array of integers `m`, and initializes `r` to `n` at line 1. In the loop of lines 2–6, the program iteratively determines the value of `a`—depending on the value of `m[i]`—and adds `a` to `r`. Finally, at line 7, the program outputs `1` if `r > 0` or `0` otherwise (line 8 is empty in `prog`). In the modified program `prog'`, change `ch1` modifies the expression at line 1 to decrement `n` by 1, `ch2` changes the value assigned to `a` at line 4, `ch3` modifies the expression evaluated and printed at line 7, and `ch4` inserts `print a` at the end.

3.1.2 Definition of Code Change

To define the *effects* of a change, in this dissertation, first we must define the notion of *change* in the executable code of a program.[1] Informally, a *code change* in a program P is a mapping between a set of executable statements in P and a new set of executable statements that replaces the former set to produce the modified program P'. To support the concepts and techniques in the rest of this dissertation, and without loss of generality, we

[1] By *program*, we mean a syntactically-correct (compilable) sequence of declarations and executable code.

assume that the area of the program where a code change is made has a single entry point. Formally, the next two definitions establish what a code change is and how it is specified.

DEFINITION 13. A *sub-ICFG* is a non-empty, connected directed graph $G^S=(V^S, E^S)$ that is a subgraph of an ICFG[2] and where V^S contains exactly one node e that has no predecessors and is the entry of the subgraph. Node e is not necessarily an *EN* node. We say that G^S is a sub-ICFG of an ICFG I if G^S is both a sub-ICFG and a subgraph of I.

DEFINITION 14. A *code change* in a program P is a description of modifications of the executable code of P and is specified by a tuple $<G, G', IN, OUT>$, where G is a sub-ICFG of the ICFG I of P, G' is a sub-ICFG that *differs* from G (i.e., the node sets, edge sets, or corresponding statements differ), and *IN* and *OUT* are partial functions $E \rightarrow V \times L_\varepsilon$ where E is the set of edges in I, V' is the set of nodes in G', and L_ε is the set of control-decision labels, including the empty label ε. The domain of *IN* consists of all edges (m,n) in I such that m is not in G and n is the entry e of G. The image of *IN* is $\{e'\} \times L_\varepsilon$ (i.e., e' maps to e for all edges that enter G). The domain of *OUT* consists of all edges (m,n) in I such that m is in G and n is not in G.

The purpose of the specification $<G, G', IN, OUT>$ for a code change is that sub-ICFG G specifies the part of the ICFG I of P (the original code in P) to replace and sub-ICFG G' specifies the subgraph (new code in P) that replaces G in I. The modified program P' is obtained from the ICFG I' that results from applying this change to ICFG I. Partial functions *IN* and *OUT* specify how G' connects to the rest of I when G' replaces G. Definition 14 indicates that all edges in I that enter G target the entry node e of G. Thus, given an edge (m,e) from the domain of *IN* and *IN*((m,e)) = (e',l), edge (m,e) is replaced by edge (m,e') with label l when applying the change. The edges that leave G in I, however, can leave from any node in G. Each edge (m,n) in the domain of *OUT* is replaced by edge (m',n) with label l, where *OUT*((m,n)) = (m',l).

[2] An ICFG is an interprocedural control-flow graph. See Definition 2 in Section 2.1.1.

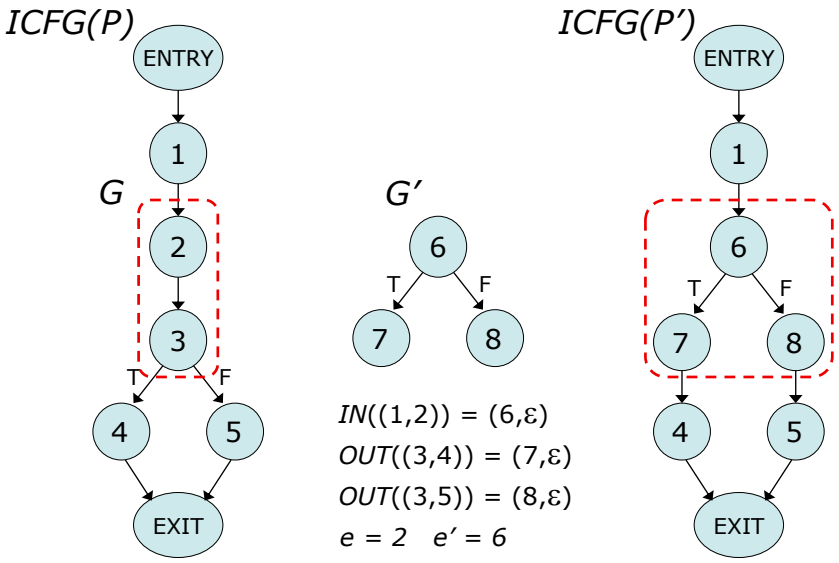


Figure 5: Example of a code change. To the left, the ICFG of program P and its sub-ICFG G (nodes 2 and 3). In the middle, sub-ICFG G', partial functions *IN* and *OUT*, and entry nodes e and e'. To the right, the ICFG of the modified program P'.

The single-entry property of code changes is needed for the definition of the effects of changes presented in Section 3.1.3. This constraint, however, does not affect the ability of Definition 14 to describe changes in code—it simply implies that modified areas in P with multiple entries can be specified either as separate, possibly overlapping code changes, or as one code change whose sub-ICFGs G and G' include all modified nodes as well as nodes e and e' that *dominate* (see Definition 4 in Section 2.1.1) all nodes in G and G', respectively.

To illustrate Definition 14, consider the code change in Figure 5. The figure shows the ICFG of program P on the left, inside which sub-ICFG G is highlighted. G consists of nodes 2 and 3 and edge (2,3). The code corresponding to each node is not important for this example and therefore is not listed. The middle of the figure shows sub-ICFG G', partial functions *IN* and *OUT*, and entry nodes e and e'. The resulting ICFG of the modified program P' is shown on the right. In this ICFG, the modified area is highlighted and corresponds to sub-ICFG G', which replaces sub-ICFG G. *IN* maps the only edge entering the change in G, (1,2), to node 6 (e') of G' with no label. Thus, edge (1,2) is replaced by unlabeled edge (1,6) in the ICFG of P'. *OUT* maps nodes 4 and 5 of G to

nodes 7 and 8 of G', respectively, with empty labels. Thus, edges (3,4) with label T and (3,5) with label F are replaced by edges (7,4) and (7,5) with no labels in the ICFG of P'.

For a more concrete example, consider change `ch1` in `prog` (Figure 4). Both G and G' for `ch1` consist of node 1, whose corresponding statement is modified. Because node 1 is the only node in the change, both e and e' are node 1. In this example, *IN* maps edge (*EN*,1) in G to node 1 in G' with no label and *OUT* maps (1,2) in G to node 1 in G' without label. For another example, `ch4` in Figure 4 represents the insertion of node 8 between nodes 7 and *EX* in `prog`. In this example, because neither G nor G' can be empty, G can be defined as node 7—even though the corresponding statement is not modified—and G' can be specified by nodes 7 and 8 and edge (7,8). *IN* maps edge (2,7) to node 7, thus preserving edge (2,7) in the ICFG of `prog'`, and *OUT* maps edge (7,*EX*) to node 8 to replace edge (7,*EX*) with (8,*EX*). All labels are empty. Node 7 is both e and e' for this change.

An example of how Definition 14 handles code areas with multiple entry points is the change resulting from the union of `ch2` and `ch3` in `prog` in Figure 4, which we denote by $\texttt{ch}_{\{2,3\}}$. Both G and G' contain nodes 4 and 7 because the corresponding statements are modified. However, if only those nodes were in G and G', the edges in the domain of *IN* would have to include (3,4) and (2,7), violating the property that the target of those edges is a unique node. Hence, G and G' must also contain nodes e and e' that dominate all other nodes in their respective graphs.[3] Therefore, change $\texttt{ch}_{\{2,3\}}$ can be made consistent with Definition 14 by having both e and e' be node 2, G consist of nodes $\{2, 3, 4, 7\}$ and edges $\{(2,3), (2,7), (3,4)\}$, and G' consist of nodes $\{2, 3, 4, 7, 8\}$ and edges $\{(2,3), (2,7), (3,4), (7,8)\}$. For this change, *IN* maps edge (1,2) to node 2 with no label and *OUT* maps edges (4,6) and (7,*EX*) to nodes 4 and 8, respectively, with no labels.

[3]Because a sub-ICFG is a connected graph, its entry node dominates all other nodes in the sub-ICFG.

3.1.3 Effects of Individual Changes

To facilitate the discussion, hereafter, code changes are simply called *changes* and program executions are considered *reproducible* (i.e., all executions for the same input are identical).[4] Let P and P' be two versions[5] of a program where P' is the result of applying the set of N changes $\{C_i \mid 1 \leq i \leq N\}$ to P. Let $P'\backslash_C$ be the program version that results from applying all changes except C to P.

Changes alter the states of the program and/or the instructions that get executed. The next three definitions formalize the concepts of sequences of instructions and states for programs and changes.

DEFINITION 15. The *execution history of a program* for input I is the sequence of statements executed when that program is run with input I.

DEFINITION 16. The *augmented execution history of a program* for input I is an extension of the execution history of that program for input I, where each statement occurrence s^j is paired with the parts of the program state modified by s^j. The parts of the state modified by a statement occurrence s^j are all variables defined at s^j and the program counter (i.e., the index of the next statement).

For example, consider program `prog'` (Figure 4) and input {`n=5, m=[1,-1,0,3]`}. Let pc denote the program counter after a statement execution. The first five elements in the augmented execution history of `prog'` are:

$<1^1$,{r=4,pc=2}>, $<2^1$,{i=1,pc=3}>, $<3^1$,{pc=4}>, $<4^1$,{a=5,pc=6}>, and
$<6^1$,{r=9,pc=2}>.

A change C may or may not occur (i.e., execute) during a program execution, and if it occurs, it might do so multiple times. C occurs during an execution if at least one of its statements in the modified program is executed.

[4] If all sources of non-determinism in an execution, such as thread interleavings and random values, are captured and made part of the input, the execution is reproducible.

[5] A *version* of a program is that program with a set of changes applied to it.

DEFINITION 17. The *augmented execution history of a change* C on program P for an execution of P is the sequence of occurrences of the location (entry point) of change C in that execution and, for each occurrence, the subset of the augmented execution history of the program containing only statements that are semantically dependent on C in that execution. (See Definition 8 in Section 2.1.2 for the definition of semantic dependence.)

For program `prog'` from Figure 4, executed on input {`n=5, m=[1,-1,0,3]`}, the augmented execution history of change `ch2` contains three occurrences of the change: 4^1, 4^2, and 4^3. The subset of the augmented execution history for occurrence 4^1 of `ch2` in this example is:

$<4^1,\{a{=}5,pc{=}6\}>$, $<6^1,\{r{=}9,pc{=}2\}>$, $<6^2,\{r{=}11,pc{=}2\}>$, $<6^3,\{r{=}13,pc{=}2\}>$,
$<6^4,\{r{=}18,pc{=}2\}>$, and $<7^1,\{out{=}1,pc{=}8\}>$

where *out* denotes the value of the output. Note that the subsets for 4^2 and 4^3 overlap with the subset for 4^1 on all elements after 4^2 and 4^3, respectively.

After a change occurs, the executions of the versions of the program with and without the change potentially diverge in path or state or both, from that point. That divergence continues for part of the respective execution histories afterwards, possibly for the remainder of those two executions. The next definition formalizes this divergence.

DEFINITION 18. Given an execution of modified program P' with input I, the *effects* of change C on that execution are the differences[6] in the augmented execution histories of C on P' and $P'\backslash_C$ for input I.

DEFINITION 19. The *effects* of change C on program P' are the union of the effects of C on all possible executions of P'.

The effects of a change on a modified program reflect the actual meaning of that change

[6]The differences between two sequences can be specified by a set of additions, deletions, and modifications of their elements. Because there might be multiple such sets that specify the differences between two sequences, we use the term *differences* more broadly to represent all possible such sets.

for the behavior of the program. Thus, a change can be viewed not just as a localized syntactic modification of the code, but more precisely as the collection of *behavioral* differences (i.e., differences in states or paths) caused by the syntactic modification. Such differences are initiated at the location of the change and propagate to all affected parts of the program, however far from the location of the change they are.

3.1.4 Effects of Multiple Changes

Changes in programs usually do not occur in isolation. Thus, it is necessary to define what the effects of multiple changes on each other mean—how these effects interact. The guiding principle for multiple changes in our formal model is that

> *A change C_1 affects another change C_2 if the presence of C_1 alters in any way the effects of C_2 on the program.*

This section provides formal definitions of dependence among changes and change interaction, including a proof that change interaction is a symmetric relation.

The context for the effects of a change C is the entire program with all other changes in place, rather than just the syntactic location of C and the original program with change C. Because the meaning of C corresponds to its effects on the entire modified program, any modification of those effects caused by other changes is an interaction with C. This consideration leads to the definition of semantic dependence among changes.

DEFINITION 20. A change C_1 is *semantically dependent* on a change C_2 in program P' if the effects of C_1 on P' differ from the effects of that same change on $P'\backslash_{C_2}$ (i.e., P' without the changes in C_2).

In other words, C_1 semantically depends on C_2 if the presence or absence of C_2 on the modified program P' determines which effects C_1 has on P'. This definition of semantic change-dependence is similar to the semantic dependence among two statements as defined by Podgurski and Clarke [79], except for two differences. First, semantic change-dependence uses the augmented execution history of a change, which includes not only the

syntactically modified statements but also all statements and states affected by that change across the program. Second, instead of considering all possible changes to the computations performed at the source of the dependence, our definition uses the concrete change C_1 between $P'\backslash_{C_2}$ and in P'. This second difference is key for making possible the computation of such dependencies, especially for individual executions.

The semantic dependence between two changes is a symmetric relation. The following theorem formalizes this property.

THEOREM 1. If a change C_1 is semantically dependent on a change C_2, then change C_2 is also semantically dependent on change C_1.

PROOF OF THEOREM 1. The proof is by contradiction. Assume that C_1 is semantically dependent on C_2 but C_2 is not semantically dependent on C_1. Consider an execution in which the set of effects of C_1 is non-empty on either P' or $P'\backslash_{C_2}$. For that execution, let a and b describe the respective parts of the execution histories of P' and $P'\backslash_{C_1}$ that differ (a and b describe the effect of C_1 on P'). Note that a may be equal to b (in which case, a and b would be empty). Also, let c and d describe the respective parts of the execution histories of $P'\backslash_{C_2}$ and $P'\backslash_{C_1,C_2}$ that differ (c and d describe the effects of C_1 on $P'\backslash_{C_2}$). Again, c may be equal to d. However, it cannot be the case that both $a = b$ and $c = d$, because, if so, there would be no difference in the effect of C_1 on P' and $P'\backslash_{C_2}$, and C_1 would not be dependent on C_2 (recall our assumption that C_1 is dependent on C_2). Thus, $a \neq b \vee c \neq d$—there is a non-empty effect of C_1 on either the program pair (P', $P'\backslash_{C_1}$) or the pair ($P'\backslash_{C_2}$, $P'\backslash_{C_1,C_2}$). Thus, because the effect of C_1 is non-empty on at least one of those program pairs, and because the effect is different between those two pairs (by Definition 20), a must differ from c or b must differ from d. In other words, either the execution histories of P' and $P'\backslash_{C_2}$ differ or the execution histories of $P'\backslash_{C_1}$ and $P'\backslash_{C_1,C_2}$ differ, so C_2 has a non-empty effect on at least one of these pairs. For those two effects of C_2 to be equal—we assumed that C_2 is not semantically dependent on C_1—it is necessary that $a = b \wedge c = d$, which contradicts our previous conclusion that $a \neq b \vee c \neq d$. Thus,

if C_1 is semantically dependent on C_2, the effects of C_2 on P' and $P'\backslash_{C_1}$ must be different and, by Definition 20, C_2 is semantically dependent on C_1. □

3.2 Computation of the Effects of Changes

Based on the formal model of change effects (Section 3.1) and the principles of the PIE model (Section 2.2), we can give now a systematic approach for computing the conditions that cause a change to affect the behavior of a program. These conditions are captured and computed by procedure COMPUTEEFFECTS[7] listed in Figure 6. The procedure inputs the original (unmodified) program P and change C (see Definition 14) and, at line 1, computes the modified program P' by applying C to P.

According to definitions 18 and 19, the effects of a change are the differences in the augmented execution history of the change (Definition 17) for P and P'. Thus, the effects of a change cannot be fully specified in terms of either program alone. For example, in Figure 4, the execution history of change `ch4` in `prog` is the history of line 7 (subgraph G of `ch4`). This history is not modified in `prog'` and, therefore, it provides no clue on what program behaviors are affected. Only the inserted code for that change (graph G' of `ch4`) specifies that it is the output that is affected by the change.

To cover all affected behaviors, COMPUTEEFFECTS outputs the effects of change C as a set of PIE conditions for two sets of dependence chains (sequences): one set for G in P and the other for G' (for P'). The computation of effects is thus performed in two steps (two iterations of the loop of lines 2–20), one for P and the other for P'. The conditions computed inside this loop are described next in the next two sections.

3.2.1 Chain Conditions

The augmented execution history of a change (Definition 17 in Section 3.1.3), a key part of the definition of the effects of a change (Definitions 18 and 19 in Section 3.1.3), uses the

[7] We use *procedure* instead of *algorithm* because COMPUTEEFFECTS might not terminate.

semantic dependencies among statements. Computing semantic dependencies for all possible executions is an undecidable problem, but semantic dependencies can be approximated by *syntactic* dependencies (i.e., possible dependencies identified by a program analysis).

DEFINITION 21. A node n is *syntactically dependent* on a node m in the ICFG of a program P and according to a *safe* analysis A,[8] if and only if there is a *chain* of dependencies (control or data or both) identified by A in P that starts at m and ends at n.

DEFINITION 22. A *dependence chain* is a pair $<n$,*seq*$>$ where n is the starting node and *seq* is a possibly-empty sequence of dependencies where the target node of each dependence is the source node of the next dependence in *seq*.

Syntactic dependence is a necessary condition for semantic dependence (see Section 3.1.3). Thus, a natural choice for a procedure that computes the effects of a change is to compute first the dependence chains from the change in both P and P'. For example, in program `prog` of Figure 4 with and without change `ch2`, there are four chains from node 4 to node 7: <4, ((4,6), (6,7))>, <4, ((4,6), (6,6), (6,7))>, <4, ((4,6), (6,6), (6,6), (6,7))>, and <4, ((4,6), (6,6), (6,6), (6,6), (6,7))>.

Lines 4–13 and 16–17 of procedure COMPUTEEFFECTS (Figure 6) compute the *chain conditions* for the change in *program*. The chain conditions are the constraints on the program input for reaching each node in the change and executing each dependence chain from those nodes. The constraints for each chain are the path condition (see Section 2.1.4) of the paths that cover it.[9] Lines 4–7 begin this computation by creating one "empty" chain $<n$, ()$>$ for each definition or branching decision in each node n of the change in *program* and placing this chain in a work list used in the rest of the procedure. Lines 6–8 assign to each empty chain, as the condition for its execution, the path condition for reaching its starting node by calling *reachingCondition*.

[8] A program analysis is safe if it has no false negatives such as missed dependencies.

[9] We use "path condition" not only for single paths but also for the disjunction of path conditions for all paths that reach a node or that cover a dependence.

Procedure COMPUTEEFFECTS

Inputs: P: program; C: change
Output: *effects*: map *program* $\rightarrow$ *dependence chain* $\rightarrow$ *conditions*

```
(1)   P' := applyChange (P, C)
(2)   foreach program ∈ { P, P' }
(3)     altProgram := P' if program is P ; P otherwise
(4)     chainWorklist := createEmptyChains (program, change)
(5)     foreach emptyChain ∈ chainWorklist
(6)       effects[program][emptyChain] := reachingCondition (program, start(emptyChain) )
(7)     endfor
        // incremental computation of conditions
(8)     while chainWorklist ≠ ∅
(9)       prefixChain := pop (chainWorklist)
(10)      foreach dep ∈ nextDependencies (program, prefixChain)
(11)        chain := append (prefixChain, dep)
            // chain conditions
(12)        depCond := coverCondition (program, prefixChain, dep)
(13)        chainCond := effects[program][prefixChain] ∧ depCond
            // state conditions
(14)        (S, S_alt) := PSE_Dep (program, altProgram, dep)
(15)        stateCond := live(S) ≠ live(S_alt) ∨ (pc(S) ∧ ¬ pc(S_alt))
(16)        effects[program][chain] := chainCond ∧ stateCond
(17)        push (chainWorklist, chain)
(18)      endfor
(19)    endwhile
(20)  endfor
(21)  return effects
```

Figure 6: Procedure that computes the effects of a change.

The *while* loop in lines 8–19 iterates over the work list of chains until the list is empty, extracting one chain at a time at line 9 and placing it in *prefixChain*. For each *prefixChain*, the *for* loop of lines 10–18 extends it with every control or data dependence that starts where the prefix chain ends (line 11). For each dependence, the loop computes its coverage conditions (line 12) in terms of the program variables at the source of the dependence and then computes the chain conditions for the extended chain as the conjunction of the chain conditions of the prefix chain and the conditions for the new dependence (line 13). The auxiliary procedure *coverCondition* takes not only the program and the dependence but also the prefix chain to ensure that only *realizable* paths (i.e., paths whose procedure call

and return edges match) are considered. Line 16 stores this chain condition for the extended chain along with the state condition (see Section 3.2.2) for the chain. Finally, line 17 adds the chain to the work list to be further extended in later iterations of the *while* loop.

3.2.2 State Conditions

Syntactic dependence, although necessary, is not a sufficient condition for semantic dependence of a node s on a change C—the infection created by a change might not propagate through any chain connecting C and s. For example, node 7 in program `prog'` is connected to change `ch1` via node 6 by a chain of data dependencies on variable `r`, but the value printed by node 7 is not affected by `ch1` unless the value of `r` that reaches that point is 0 with `ch1` and thus 1 without `ch1`. Hence, to obtain the effects of a change C on a point in the program, additional conditions on the program state are needed.

Despite the undecidability of computing semantic dependencies, it is possible to add conditions on the program state for such dependencies incrementally from the change, as shown in procedure COMPUTEEFFECTS of Figure 6. Recall that the *while* loop in the procedure extends each chain one control or data dependence at a time. Although the computation of effects for a particular program point might not terminate due to cycles in control- or data-flow, an incremental procedure for computing these effects can be the basis for useful approximations such as those presented in the next chapters.

The program state can be logically divided in two parts: the program variables and the *program counter* (i.e., the pointer to the next instruction to execute). Thus, there are two cases in which an infection propagates throughout a dependence chain:

- Case 1: the infection is a difference in the value of at least one program variable.
- Case 2: the infection is a difference in the program counter that causes a divergence in the paths taken by the program.

To compute the two cases of *state conditions*, line 14 invokes *PSE_Dep* to perform *partial symbolic execution* [6, 92] defined next.

DEFINITION 23. *Partial symbolic execution*, or *PSE* for short, is a form of symbolic execution that starts and ends at arbitrary points in the program and uses as symbolic input the program variables at the start point. The result of PSE consists of the state of the program at the end point and the path condition[9] for reaching that point from the start point. Both results are in terms of the symbols for the variables at the start point.

PSE_Dep applies PSE to the paths that cover dependence *dep* in *program* and its alternate *altProgram* (determined at line 3) and returns the respective symbolic states S and S_{alt} at the end of those paths. The symbols in these states are the program variables at the source node of the dependence. The symbol for each of these variables, however, is equated to the symbolic value of the variable already available *effects* when adding the state condition at line 16. Similarly, *PSE_Dep* uses placeholder conditions p and p_{alt} stating that the execution in the respective program reaches the source node of the dependence; those placeholders are replaced at line 16 by the actual path conditions for reaching the end of the prefix chain.

Condition p_{alt} is not satisfiable if the source of dependence *dep* does not exist in *altProgram* or is modified so that the last dependence in *prefixChain* does not exist or does not end in that node. Also, the path condition in S_{alt} is not satisfiable if *dep* does not exist in *altProgram* (i.e., in *altProgram*, the source or target node do not exist or there is no dependence between those nodes).

For Case 1, line 15 of the procedure specifies in the first term of the disjunction that the value of some *live* variable[10] at the target of the dependence differs in the two programs. Only live variables are of interest because any differences in the values of *dead* variables[11] do not affect the behavior of program.

For Case 2, the procedure at line 15 specifies in the second term of the disjunction that the path condition in S must hold while the path condition in S_{alt} must not hold. One way

[10] A variable v is *l*ive at a program point n if there is a definition-clear path for v from n to a use of v.

[11] A variable v is *dead* at a program point n if there is no definition-clear path for v from n to any use of v.

in which the negation of the path condition in S_{alt} can be satisfied is that *altProgram* simply does not reach the source node of *dep*.

3.3 Related Work

In References [6] and [92], we introduced partial symbolic execution (PSE) and symbolic state differencing as mechanisms for testing the effects of changes. In this chapter, we defined the foundations for that work by developing a comprehensive formal model of the effects of changes (Section 3.1) that underlie and justify those mechanisms and by specifying in full how those effects can be obtained programmatically (Section 3.2). We presented earlier versions of the work in this chapter in References [92] and [97].

A number of theories and models of fault propagation have been proposed in the literature. These models target fault-based testing, a form of testing that focuses on specific classes of faults to improve the chances of discovering faults in those classes. Although these models address faults instead of changes, the same principles for the propagation of faults can be applied to the propagation of the effects of changes.

Morell [71] presented a theory that, given a class of faults, defines the conditions under which fault-based testing for individual faults in that class succeeds at reaching the output. These conditions are formulated as a set of fault-propagation equations that can be solved using symbolic evaluation. Our formal model of the effects of changes (Section 3.1) is consistent with Morell's theory. Unlike this theory, however, our model also provides precise definitions of what a change in program code is, how the effects of such a change manifest in individual executions, and how changes affect a program at each point and for all possible executions. Also unlike Morell's model, our model addresses multiple changes and their possibly-interacting effects. Furthermore, the definitions in our model lead directly to the formulation of a programmatic approach for obtaining the effects of changes (Section 3.2)—an approach from which practical and effective approximations can be derived, as shown in the next chapters of this dissertation.

Richardson and Thompson [85], inspired by Morell's theory, defined a framework called RELAY that identifies the different phases involved in executing a fault and propagating it to the output to produce an observable failure. In this framework, users can specify, for a class of faults, the conditions under which faults in that class transfer from instruction to instruction to the output. Although RELAY does not specify concrete transfer conditions for faults, the authors later defined the transfer conditions for specific kinds of instructions to illustrate how the framework is used [105]. These conditions, however, unlike our model and computational approach presented in this chapter, are insufficient to specify in general the effects of a change and their propagation throughout the program.

Voas defined the PIE model [109] to describe in a simpler way the same notions of origination and transfer of erroneous states present in the RELAY framework. In this model, a fault is revealed by the output if it executes (E), infects the program state (I), and the infected state propagates to the output (P). Like, RELAY, however, the PIE model only identifies these phases without giving the conditions that carry out these phases. Although Voas defined the PIE model for a technique that estimates the testability of program code, the PIE model also provides a convenient and succinct way to refer to the main phases of the RELAY framework. In this dissertation, we use the PIE model to justify and illustrate the different aspects of the effects of changes and their computation.

Other techniques exist that compute approximations of the effects of changes for testing and impact analysis. For testing changes, Binkley [17], Rothermel and Harrold [87], and Gupta and colleagues [50] present techniques that use slicing [57, 112] to obtain testing requirements corresponding to individual data- and control-dependencies affected by a change. These techniques, however, are quite limited for representing the effects of changes accurately. A dependence that is potentially affected by a change is not necessarily executed exclusively as part of the propagation of the effects of the change. Also, individual dependencies do not reflect the different and complex ways in which dependencies interact to propagate infections. Furthermore, these techniques do not account for the differences

in the values of variables in the program. Meanwhile, change-impact analysis techniques (e.g., [20, 21, 68, 76, 83]) identify elements such as methods or statements that are possibly affected by a change and therefore need special attention. These techniques, however, are no better than change-testing approaches for modeling the effects of changes at a fine-grained level, for similar reasons. Such techniques either rely on slicing only or identify coarse-grained entities such as methods executed after a change.

CHAPTER IV

SCALABILITY OF CHANGE-EFFECTS ANALYSIS

Symbolic execution [28, 63] is a key analysis needed for computing the effects of changes and many other applications in software engineering. Unfortunately, however, this technique has serious scalability problems because the number of paths in a program grows exponentially with the size of the program (or, for PSE, with the distance between the start and end nodes). The number of paths to analyze can, in fact, be infinite. For this reason, techniques that use symbolic execution in practice, such as the one presented in the next chapter, must limit in some way the extent of the analysis performed. Therefore, any increase in the number of paths that can be symbolically executed for the same amount of time can have a great, positive impact in the cost-effectiveness of practical techniques.

This chapter presents a new technique, called *Symbolic Program Decomposition* (or SPD), for symbolic execution of multiple paths that is more scalable than existing techniques, which symbolically execute control-flow paths individually. SPD exploits control and data dependencies to avoid analyzing unnecessary combinations of subpaths. SPD can also compute a safe approximation of symbolic execution by abstracting away symbolic subterms arbitrarily to further scale the analysis at the cost of precision.

In this chapter, Section 4.1 gives a detailed motivation for this new technique in the context of existing research in symbolic execution. Then, Section 4.2 presents and illustrates the SPD technique and Section 4.3 discusses its implementation. Section 4.4 presents a study that uses this implementation to assess the ability of SPD to save path-exploration costs. Finally, Section 4.5 discusses related work in detail.

4.1 Motivation

Approaches for symbolic execution of individual paths have been presented that heuristically select new paths to explore in ways that get increasingly "closer" to target program points and, thus, reduce the number of paths analyzed [47, 99, 116]. Further research on modularity has made symbolic execution more efficient by computing reusable method summaries [3, 24]. However, despite these advances, the scalability of symbolic execution on all paths is still compromised by the path-explosion problem. For relatively large programs and components, only a small set of paths can be symbolically executed for a reasonable computational budget.

In the related field of model checking [107], which exhaustively explores sequences of states, researchers address the scalability problem by using state abstractions and on-demand refinement [26]. However, most software model checkers (e.g., [9, 14]) do not offer control-flow (path) abstractions; instead, they operate directly on the control-flow graph of the program. Although some techniques use demand-driven expansion of method calls into constituent paths [8, 24], simply abstracting paths as methods does not prevent the explosion of the path space when methods are expanded. Another abstraction, called path slicing [61], simplifies a path to better refine state abstractions, but its result is limited to the features of the original path and represents only a fraction of the path space.

To alleviate the path-explosion problem of symbolic execution for change-effects analysis and many other techniques, we developed *Symbolic Program Decomposition* (SPD). SPD is a new interprocedural technique for symbolically executing multiple paths that exploits the dependence structure of programs to obtain the same result as *traditional symbolic execution* (TRADSE)[1] (i.e., symbolic execution of individual control-flow paths, one by one) but more efficiently. Instead of analyzing control-flow paths one by one, SPD performs symbolic execution on groups of paths that share common control dependencies,

[1] Hereafter, we denote traditional symbolic execution as TRADSE.

called *path families*. SPD decomposes (partitions) path families iteratively into constituent path families according to the data dependencies found within the path families. Because each of the resulting path families represents, in general, multiple control-flow paths that do not have to be analyzed individually, SPD can achieve significant savings over TRADSE while obtaining the same final result.

This section also presents a feature of SPD that lets it trade precision for even more scalability by "dropping" symbolic conditions in a safe manner to produce an over-approximating abstraction (i.e., an under-constrained description) of the results of symbolic execution. In this way, by sacrificing some precision, SPD can analyze larger sets of paths than precise symbolic execution. SPD can use any abstraction strategy provided by the user and designed specifically for the task at hand, such as change analysis. A good abstraction strategy for a given task makes SPD produce a final set of path families that is small enough for symbolic execution in practice and that remains effective for that task. For example, a good strategy for analyzing and testing a change captures the initial propagation conditions for the change while freeing up resources that help identify additional conditions from code located beyond the limits of a precise analysis.

SPD has a number of potential applications in program analysis and software engineering, in addition to analyzing changes for regression testing [78, 92]. For example, SPD can be used to approach test-input generation [47, 99, 116] from a path-family perspective. For other examples, it can be used for static invariant-discovery [31, 39, 104], bug finding [8, 37, 48], and modular analysis [3, 24]. The symbolic expressions computed by SPD model, precisely or approximately, the effects of a program module in terms of its input. Such models can serve as invariants that hold for all possible behaviors of the module and can be checked against a specification. In addition, over-approximate results from SPD can be refined iteratively to produce increasingly more precise abstractions.

The main benefit of this new technique is that it can symbolically analyze, for the same effort, many more paths than TRADSE, improving the cost-effectiveness of many

software-engineering applications. By showing how to analyze paths in groups, rather than individually, using the control- and data-dependence structure of programs, this section shows that dependencies are a more effective representation of programs for path-sensitive analysis than simple control-flow. Another benefit of this technique is that, using safe approximations, it offers further gains in scalability in exchange for some loss in precision.

4.2 The SPD *Technique*

This section presents the new SPD technique. First, Section 4.2.1 provides the working example used to illustrate the technique. Then, Section 4.2.2 gives an overview of SPD and its core concepts, while Section 4.2.3 describes the abstraction feature of SPD. Sections 4.2.4 and 4.2.5 describe how path families are constructed and partitioned, respectively, in an iterative fashion by SPD. Finally, Section 4.2.6 presents and explains in detail the SPD algorithm.

4.2.1 Working Example

Figure 7 shows the example used in the rest of this section. Function `addElem`, listed on the left, inputs three numbers and a map reference. In statements 1–3, `addElem` creates a new map if it is not initialized and retrieves the size of the map. In statements 4–10, it retrieves the list `q` associated with `a` (if `q` is empty, it is assigned one element), or, if there is no such entry, creates a new list `q` for `a` and adds entry `a`→`q` to the map. In statements 11–14, the function adds `b` to `q` if `q` is longer than a certain value and prints `"succeeded"` or `"failed"` accordingly. Finally, the function returns the map size.

Figure 7 also shows a partial ICFG of `addElem` on the top right and the ICDG[2] of this program on the bottom right. For simplicity, in these graphs, the call-site nodes are not split into call and return nodes. Also, the figure highlights each set of nodes in the ICDG that share the same control dependence by surrounding that set in a box.

[2]For the definition of interprocedural control-dependence graph (ICDG), see Definition 10 in Section 2.1.2.

```
uint addElem(uint a,b,c; Map m)
    List q, int sz

 1.  if m == null
 2.     m = new Map
 3.  sz = m.size

 4.  if hasKey(m, a)
 5.     q = getVal(m, a)
 6.     if q.length == 0
 7.        q = <a>
    else
 8.     q = <a>
 9.     m = insert(m, a→q)
10.     sz++

11.  if q.length >= sz + c
12.     print "failed"
    else
13.     q = append(q, b)
14.     print "succeeded"

15.  return sz
```

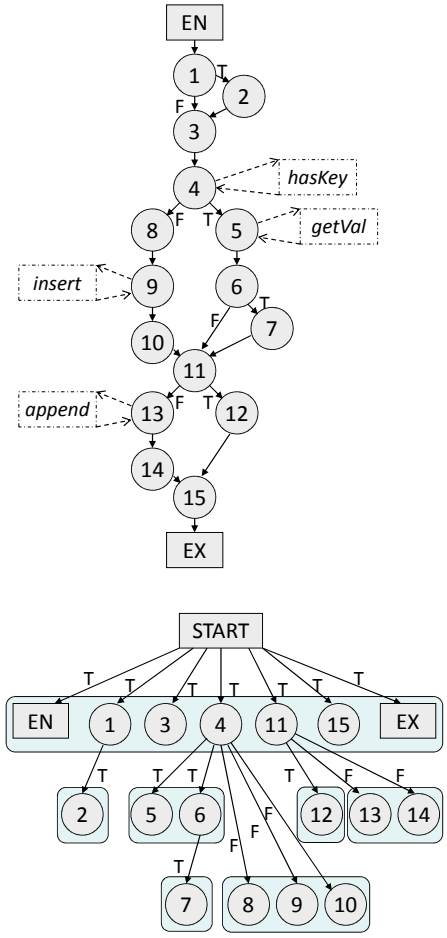


Figure 7: Example program `addElem` (left), its control-flow graph (CFG) (top right), and its control-dependence graph (ICDG) (bottom right).

4.2.2 Overview of the SPD Technique

Consider the example in Figure 7 and suppose that we want to obtain the symbolic value of `sz` at statement 15 for all paths of `addElem`.[3] Assume for now that all called functions (e.g., `hasKey`) are treated as primitive operations—the analysis does not enter those functions. The value of use $\texttt{sz}_{15}$ depends on which path we consider. For example, for path <1F,4F,11F>, $\texttt{sz}_{15}$ is $\texttt{m}_0\texttt{.size} + 1$, because `sz` is first assigned $\texttt{m}_0\texttt{.size}$ at statement 3

[3] We denote the use of variable `v` at statement n by $\texttt{v}_n$; the symbol $\texttt{v}_0$ denotes the input value for `v`. Thus, `sz` at statement 15 is $\texttt{sz}_{15}$.

and is then incremented by 1 at statement 10. Similarly, for all other paths, we obtain the respective symbolic value of sz_{15}. Using traditional symbolic execution (TRADSE), path by path, the value of sz_{15} can be written as an expression with 12 cases:

```
{ case <1F,4F,11F>:  m0.size + 1;
  case <1F,4F,11T>:  m0.size + 1;
  case <1F,4T,6F,11F>:  m0.size;
  ...  }
```

For simplicity, we do not symbolically evaluate branch conditions such as 1F and 4F for now. Although TRADSE explores 12 paths, there are paths for which sz_{15} has the same value. For example, paths <1F,4F,11F> and <1F,4F,11T> both yield the value $m_0.size + 1$ because the decision at statement 11 does not affect sz_{15}. Also, the decision at statement 6 does not affect sz_{15}. Hence, these 12 paths can be grouped as Figure 8 shows, so that the value of sz_{15} is simplified to four cases:

```
{ case <1F,4F>:  m0.size + 1;
  case <1F,4T>:  m0.size;
  case <1T,4F>:  (new Map).size + 1;
  case <1T,4T>:  (new Map).size;  }
```

In contrast with TRADSE, SPD identifies these four groups of paths without having to enumerate the 12 paths and simplify the results later. SPD achieves this by initially grouping all paths between the start and end points into one *path family* (set of paths) and then partitioning this path family iteratively until no more partitions are required to distinguish the cases in which the symbolic result differs. Like TRADSE for paths, SPD reuses prefixes shared by path families, such as <1F> for path families <1F,4F> and <1F,4T>. The savings achieved by SPD are the difference between the number of paths that TRADSE explores and the number of path families that SPD finds. In this example, the difference is eight.

A path family is described by an ordered list of control dependencies (e.g., branches) shared by its constituent paths. For example, path family <1F,4T> in program `addElem` represents the four paths that share branches 1F and 4T. By grouping paths into path families, SPD can compute symbolic values based on shared control dependencies without analyzing paths individually, which can be infinite if the code contains loops, while obtaining the same result as TRADSE. The symbolic condition corresponding to the list of control dependencies is called *path-family condition* (PFC)—the multi-path version of a path condition. The PFC is the condition for traversing *any* of the paths of the family; any distinction among these paths is irrelevant for the result. For example, the PFC for path family <1T> is $\texttt{m}_0\texttt{=null}$. PFCs also include aliasing conditions, as described in Section 4.2.6.

SPD partitions each path family independently from each other to avoid unnecessary combinations of conditions that affect different parts of the result.[4] For example, if we want the PFC to statement 12 in `addElem`, we need the symbolic expression for branch 11T which includes uses $\texttt{q}_{11}$, $\texttt{sz}_{11}$, and $\texttt{c}_{11}$ (always equal to $\texttt{c}_0$). The value of $\texttt{sz}_{11}$ is the same as $\texttt{sz}_{15}$, but $\texttt{q}_{11}$ is affected by the outcome of the condition at statement 6, so this condition cannot be ignored in this example. Thus, it would appear that all combinations of outcomes of the conditions at 1, 4, and 6 are needed because each such combination results in a different symbolic expression for 11T. However, SPD recognizes that not all such combinations are necessary because uses $\texttt{q}_{11}$ and $\texttt{s}_{11}$ can be computed independently. For $\texttt{q}_{11}$, only path families <4T,6F>, <4T,6T>, and <4F> are needed, and for $\texttt{sz}_{11}$, only path families <1F,4F>, <1F,4T>, <1T,4F>, and <1T,4T> are required.

To quantify the savings achieved by SPD interprocedurally in this last example, let H, G, and I be the number of paths in `hasKey`, `getVal`, and `insert`, respectively. Instead of analyzing the $2 \times H \times (G \times 2 + I)$ interprocedural paths between statements 1 and 12, SPD requires, at most, $H \times (G \times 2 + 1)$ path families for $\texttt{q}_{11}$ and $2 \times H$ path families for

[4] Partitions are also path-sensitive to distinguish multiple occurrences of a statement.

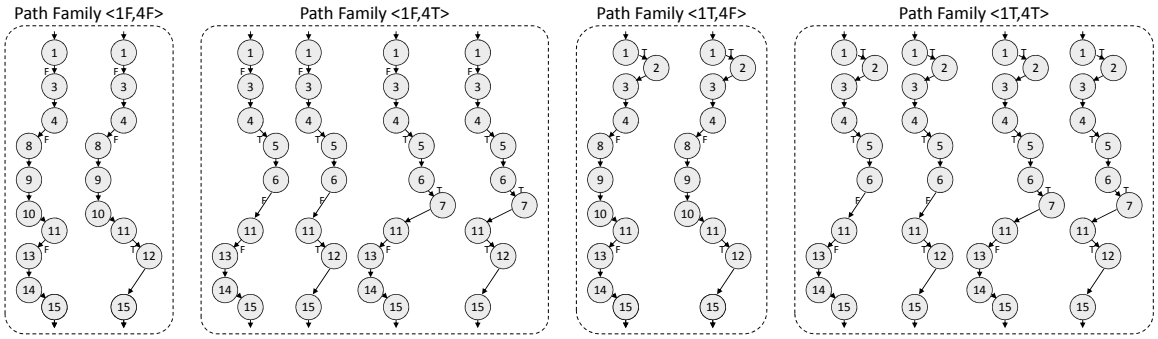


Figure 8: The 12 paths in Figure 7 form four groups, one for each value of $\texttt{sz}_{15}$. SPD finds these groups without enumerating all paths.

$\texttt{sz}_{11}$. (SPD finds that `insert` does not affect the result.) SPD reuses the analysis of the H path families, so the savings are $H \times (G \times 2 + I - 3)$ paths.

4.2.3 Abstraction of Loops and Complex Code

If parts of the code in the example program from Figure 7, such as functions `hasKey` or `getVal`, are complex or contain loops, and that code affects the symbolic value that we need to compute, then the number of path families to identify can be too large or infinite. To address this problem, SPD provides a mechanism for abstracting intermediate conditions and values (picked by SPD or the user) in a safe manner, which results in an over-approximate but simpler result. Abstractions can be used to avoid analyzing all iterations in a loop; typically, SPD limits the number of iterations to analyze, so the remaining iterations can be safely abstracted.

For example, in `addElem`, computing the PFC to statement 12 (i.e., the symbolic value of 11T) involves finding the length of list `q` at node 11. To that end, SPD looks for the values of `q` that can reach 11: `getVal(m,a)`, {a}, and {a}, assigned at statements 5, 7, and 8, respectively. To reach statement 11, these values require branch lists <4T,6F>, <4T,6T>, and <4F>, respectively. The assignment to `q` at 5 depends on the return value of `getVal`; conditions 4F and 4T depend on the return value of `hasKey`. Therefore, SPD has two alternatives: (1) enter these functions and analyze their contents (a complex endeavor),

or (2) abstract variable q at 5 and conditions 4F and 4T as $\star$, which represents the *top* value (i.e., the set of all possible values for a given type).[5] In this example, suppose that SPD chooses the second alternative. The value $\star$ for 4F and 4T means that both conditions have the value set {true, false} (i.e., both values simultaneously); because one of the values is true, both 4F and 4T are treated as satisfied, for any input.

The expression for condition 11T is q_{11}.length$\geq sz_{11}+c_0$. (Because a, b, and c are never modified, any use of these variables can be replaced by inputs a_0, b_0, and c_0, respectively.) If we replace q_{11} with its reaching definitions and their respective reaching (case) conditions, we obtain the following abstract expression for 11T:

{ case 4T,6F: $\star$.length $\geq$ sz_{11} + c_0
  case 4T,6T: {a_0}.length $\geq$ sz_{11} + c_0
  case 4F: {a_0}.length $\geq$ sz_{11} + c_0 }

which, because {a_0} is a list of one element, simplifies to

{ case 4T,6F: $\star$.length $\geq$ sz_{11} + c_0
  case 4T,6T: 1 $\geq$ sz_{11} + c_0
  case 4F: 1 $\geq$ sz_{11} + c_0 }

Meanwhile, sz_{11} requires 1F and 1T, which are m_0=null and $m_0\neq$null, respectively. Thus, sz_{11} in path family <4T> is {case m_0=null: 0; case $m_0\neq$null: m_0.size;} because the length of new Map is 0. We rewrite sz_{11} in<4T> as (m_0=null)?0:m_0.size. Similarly, in path family <4F>, sz_{11} is (m_0=null)?1:m_0.size+1.

We already saw that 4F and 4T are $\star$. 6F and 6T are also $\star$ because statement 5 assigns $\star$ to q and, thus, condition $\star$.length=0 is satisfied (the empty list, which belongs to set $\star$, has length 0). After replacing these conditions and the value of sz_{11} for each path family, the final expression for 11T is the value set

[5] $\star$ is a safe over-approximation of any symbolic expression, so an expression containing $\star$ is also a safe over-approximation of the actual symbolic expression.

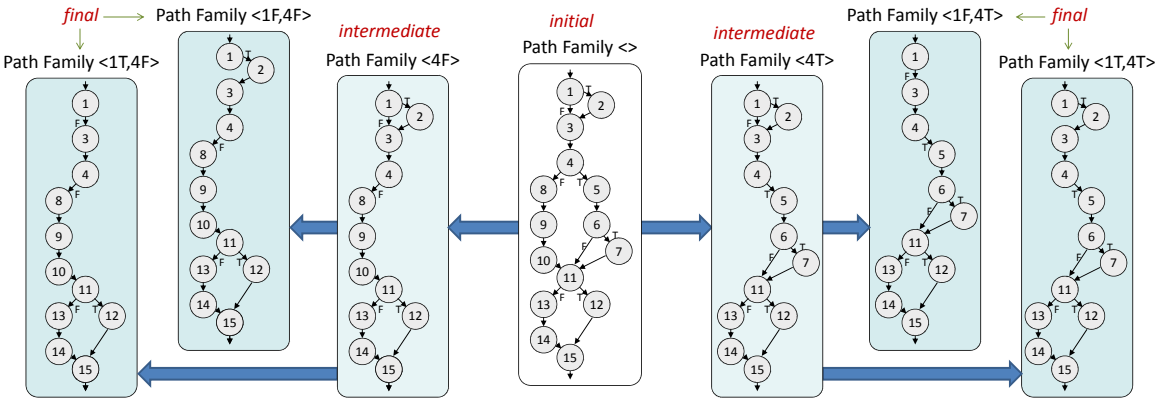


Figure 9: Partition of path families for $\mathtt{sz}_{15}$ in Figure 7. Here, path families are represented by graphs. The initial path family <> is partitioned into path families <4F> and <4T>, which are then partitioned into the final four path families from Figure 8.

$$\{\ \star\texttt{.length} \geq (\texttt{(}m_0\texttt{=null)? 0 :}\ m_0\texttt{.size)} + c_0\texttt{,}$$

$$1 \geq (\texttt{(}m_0\texttt{=null)? 0 :}\ m_0\texttt{.size)} + c_0\texttt{,}$$

$$1 \geq (\texttt{(}m_0\texttt{=null)? 1 :}\ m_0\texttt{.size + 1)} + c_0\ \}$$

where the condition `case` $\star$ for each element is omitted—it is trivially `true`.

The resulting abstract PFC is `true` if any of its three values is `true`; the abstractions applied by SPD make all three symbolic values possible at the same time.[6] An input generator that solves this condition in a way that maximizes the chances of covering statement 12 should pick an input that satisfies all three values. If the solver knows that the fields `length` and `size` of lists and maps are never negative, then it can strengthen this expression to $0 \geq (\texttt{(}m_0\texttt{=null)?0:}m_0\texttt{.size)} + c_0$, and, therefore, pick `null` for m_0 and `0` for c_0. This input executes path <1,2,3,4,8,9,10,11,12,15>, which covers the target statement 12.

4.2.4 Construction of Path Families

SPD computes the edge lists for path families by exploring the ICDG. We assume that, for each ICDG node, the order of execution of its successors within each cd-region[7] is known.

[6] $\star$ includes the `null` list, so the first element of the PFC can be an error, which we interpret as "not satisfied" (i.e., `false`)—although, in this case, this error is spurious.

[7] A cd-region is a region that groups all nodes that have the same control dependencies [41].

For example, in `addElem` (Figure 7), to obtain the edge list <11T> describing the family of all paths from EN to 12, an algorithm can use the knowledge that EN executes before node 11 in cd-region START-T to determine that node 11 is reachable from EN and then find the ICDG edge 11T to node 12. In general, however, the ICDG traversal can be more complex because some path families need to include or avoid intermediate points. For example, consider finding the edge list for the family of paths between nodes 2 and 13 that also cover node 7 in Figure 7. Starting at node 2, there is no ICDG path to node 7 and there is no ICDG path from node 7 to node 13. However, after node 2, the program returns to the cd-region START-T just after node 1. Similarly, after node 7, the program returns to START-T just after node 4. Within this cd-region, the program then reaches node 4 from node 1 and node 11 from node 4. From node 4, edges 4T and 6T lead to node 7, and from node 11, edge 11F leads to node 13. Thus, this path family is described by edge list <4T,6T,11F>. Note that not all edges in the list are consecutive in the ICDG.

Informally, the algorithm that finds the edge list from node u to v first looks for a path in the ICDG from u to v. If this fails, v may still be reachable in the ICDG from some other node w that executes after u. For example, in Figure 7, there is no ICDG path from node 7 to 13, but there is node 11, which executes after 7, and from which there is an ICDG path to 13. Thus, the algorithm looks for an ICDG path to v from either u or another node w that executes after u. Node w can be located after u (in execution order) in the cd-region of u, or located after an ICDG ancestor u' of u in the cd-region of u'. In our example, if u is node 7 and v is node 13, then w is node 11 and u' is node 4; both u' and w are in cd-region START-T, and w is located after u' in that region.

The rules for the construction of path families are formalized in the next three definitions. We first define *executes after*.

DEFINITION 24. Node w *executes after* node u if w is u or (1) there is a cd-region R that contains u' and w, where u' is u or an ICDG ancestor of u, and (2) w is located after u' in R.

Using this definition, we now define *valid* ICDG-edge lists.

DEFINITION 25. A list of ICDG edges for nodes s and t is *valid* if (1) for each pair (e_1, e_2) of consecutive edges in that list, the source node v of e_2 executes after the first node u of cd-region e_1, (2) the source node s' of the first edge in the list executes after s, and (3) t executes after the first node t' of the cd-region of the last edge of the list. An empty list is valid if t executes after s.

Valid lists of ICDG edges help define *path families* precisely.

DEFINITION 26. A *path family* is a triple $<s, t, E>$, where s is the starting point, t is the ending point, and E is a non-empty set of valid ICDG edge lists for node pair (s, t).

To illustrate these definitions, the family of all paths in Figure 7 from EN to 13 that include 7 is <EN,13,{<4T,6T,11F>}>. The edge list <4T,6T,11F> is valid because (1) for pairs (4T,6T) and (6T,11F), the source node of the second edge on each pair executes after nodes 5 (the first node of region 4T) and 7 (the first node of region 6T), respectively; (2) node 4 executes after EN; and (3) node 13 is the first node of region 11F. In this example, E contains one edge list, but, in general, multiple edge lists might be needed to describe the paths between s and t. If, for example, an ICDG has loops, then E might contain an infinite number of edge lists. In such a case, E can be described finitely using regular expressions or, to consider only realizable paths for the interprocedural case, context-free grammars.

4.2.5 Partition of Path Families

SPD starts with an *initial path family* that represents all paths to analyze between two points. SPD then partitions the initial path family into *intermediate* path families, which are subsets of the paths of the original family. The partitioning is determined by the data dependencies (reaching definitions) for variables used in the PFC of the initial family and the expression to symbolically evaluate. The lists of ICDG edges describing the partitioned path families "refine" the original list with additional edges inserted at any point. SPD continues this process iteratively on the intermediate families until it obtains the set of *final*

path families that cause the desired symbolic expression to contain only input symbols.

For the example of Figure 7 in which $\texttt{sz}_{15}$ is computed intraprocedurally, Figure 9 shows the partitions performed, where each path family is depicted as a control-flow graph. SPD partitions the initial path family <> (Figure 9, middle) using the reaching definitions for $\texttt{sz}_{15}$ at statements 3 and 10 and the respective conditions, <4T> and <4F>, under which these definitions of `sz` reach $\texttt{sz}_{11}$ (the condition for the definition of `sz` at 3 is <4T> to avoid the re-definition at 10). The result of splitting the path family <> is the pair of intermediate path families <4F> and <4T> in Figure 9. In addition, for each intermediate path family, SPD substitutes for $\texttt{sz}_{15}$ the right-hand side expression of the corresponding definition. Thus, SPD transforms the initial { `case <>:` $\texttt{sz}_{15}$`;` } into

{ `case <4F>:` $\texttt{sz}_{10}$ `+ 1;  case <4T>:` $\texttt{m}_3$`.size; }`

SPD continues by finding one reaching definition at 3 for use $\texttt{sz}_{10}$ within path family <4F> and replacing $\texttt{sz}_{10}$ with $\texttt{m}_3$`.size`. Because there is only one definition for $\texttt{sz}_{10}$, the path family <4F> does not need to be partitioned any further—<4F> is also the condition for that definition to reach $\texttt{sz}_{10}$. Finally, SPD finds that $\texttt{m}_3$ (now occurring in both cases of the expression) is reached by the definition of `m` at statement 2 and by input $\texttt{m}_0$, whose respective conditions are <1T> and <1F>. After partitioning path families <4F> and <4T> with <1T> and <1F>, and replacing $\texttt{m}_3$ with the respective values, SPD produces the set of final path families (the two leftmost and two rightmost path families in Figure 9) that form the same result as Expression (1) above. In summary, for our example, SPD produces a minimal expression by exploring only four cases (final path families), in contrast with TRADSE, which explores all 12 paths. Furthermore, if we included the paths from all functions and also expanded 4F and 4T, the savings in the number of cases analyzed would be even greater because only the code in `hasKey` affects $\texttt{sz}_{15}$.

Algorithm DOSPD

Input: P: program to analyze
s, t: start and end statements in P
V: set of variables to evaluate at t
Abstract: boolean function on use-path pairs

Output: *PFC(s→t)*: path-family condition from s to t
V_{sym}: symbolic values of variables in V

(1) *PFC(s→t) = findEdgeLists(s, t)*
(2) V_{sym} = {<*u, PFC(s→t)*> | $u \in V \times \{t\}$}
(3) *workset* = V_{sym}
(4) **foreach** term $C \in$ *PFC(s→t)*
(5) *p = getPrefixToTerm(PFC(s→t), C)*
(6) **foreach** use $u \in C$
(7) *workset* $\cup =$ <*u, p*> // initially not expanded
(8) **endfor**
(9) **endfor**
(10) **while** *workset* $\neq \emptyset$
(11) pick and remove <*u, p*> from *workset*
(12) **if** *Abstract(<u, p>)*
(13) *linkPairs(<u, p>, <*$*$*, p>)*
(14) mark <*u, p*> expanded
(15) **continue** to (10)
(16) **endif**
(17) *D = getReachingDefinitions(u, p)*
(18) *I = getReachingInputs(u, p)*
(19) **foreach** assignment-path pair <*a,*p_a> $\in D \cup I$
(20) p' *= getDefClearPathFamily(*p_a*, var(u), p)*
(21) *linkPairs(<u, p>, <a,* p'*>)*
(22) **foreach** use $u_a \in$ *rhs(a)*
(23) **if** <u_a, p_a> not expanded: *workset* $\cup =$¡u_a, p_a>
(24) **endfor**
(25) **foreach** term $C \in p'$
(26) p_c *= getPrefixToTerm(*p'*, C)*
(27) **foreach** use $u_c \in C$
(28) **if** <u_c, p_c> not expanded: *workset* $\cup =$<u_c, p_c¿
(29) **endfor**
(30) **endfor**
(31) **endfor**
(32) mark <*u, p*> expanded
(33) **endwhile**
(34) **return** *PFC(s→t)*, V_{sym}

Figure 10: The algorithm for computing a Symbolic Program Decomposition (SPD) for all paths between two points.

4.2.6 The SPD Algorithm

In this section, we formally present algorithm DOSPD, listed in Figure 10, that performs symbolic program decomposition (SPD) for the set of all paths between two program points, using control and data dependencies. DOSPD inputs a program P, a starting point s in P, an ending point t in P, a (possibly empty) set V of variables to symbolically evaluate at t, and an abstraction function *Abstract*. DOSPD outputs the PFC to reach t from s and the symbolic values at t of the variables in V. DOSPD treats s as the entry point, so a variable x at entry s is the symbolic input x_0.

4.2.6.1 Initial Steps

At line 1, DOSPD invokes *findEdgeLists* (not shown) to initialize *PFC(s→t)* with the set E of ICDG-edge lists describing the initial path family. This description provides the *top-level terms* (i.e., the conditions given by the ICDG edges) for *PFC(s→t)*; those terms will be later expanded into full symbolic expressions. For example, the initial path family between EN and 12 in Figure 7, which we call $F_{EN \to 12}$, is <EN, 12, {<11T>}>. For $F_{EN \to 12}$, 11T is the only top-level term. At line 2, DOSPD creates the set V_{sym} containing a *pseudo-use* of each variable v in V at t (i.e., a pair (v, t) treated by SPD as a use of v at t, regardless of whether v is used at t or not). Each pseudo-use is paired in V_{sym} with the condition required to reach t: *PFC(s→t)*. For example, if V is {q}, then V_{sym} is assigned {<q$_{12}$,$F_{EN \to 12}$>}. At line 3, DOSPD initializes the working set *workset* of *use-path pairs* (i.e., pairs of uses and the PFCs to reach those uses from s) with all use-path pairs from V_{sym}. In lines 4–9, DOSPD adds to *workset* all uses of variables from the top-level terms of *PFC(s→t)* (e.g., sz$_{11}$), pairing each use u in those terms with the corresponding *prefix* of *PFC(s→t)* (e.g., <EN,11,{<>}>) computed by *getPrefixToTerm* (not shown), that ends at the term where u is located. For a general example, consider PFC <s,t,{<A,B,C>,<D,E>}>; the prefix of B in this path family is <s,t,{<A>}>, where the first list was trimmed and the second list was discarded.

4.2.6.2 Abstraction Decision

In lines 10–33, DOSPD proceeds to iteratively pick and remove from *workset* one use-path pair $<u,p>$ (line 11). The call to the user-provided *Abstract* at line 12 decides whether the pair $<u,p>$ is abstracted away. If *Abstract* returns true, DOSPD proceeds to lines 13–15, which *link* (assign) to $<u,p>$ the value $*$ and the same path family p, mark the pair as *expanded* (processed), and continue to the next iteration—skipping the regular processing of lines 17–32. In this case, no new elements are added to *workset*. For example, if *Abstract* decides to abstract $<\mathtt{sz}_{11},F_{EN\rightarrow 11}>$, then the value of $\mathtt{sz}_{11}$ in $F_{EN\rightarrow 11}$ is $*$ and this pair is not further processed.

4.2.6.3 Backward Expansion

At lines 17–31, DOSPD *backward-expands* u within p. Backward expansion of a use u in path family p first uses *getReachingDefinitions* and *getReachingInputs* (not shown) at lines 17–18 to find the data dependencies for u: all definitions and inputs for the variable of u that might reach the location of u through some definition-clear path within path family p. For example, use m_3 is reachable by definition m_2 and input m_0. In lines 19–31, for each such definition or input a and its associated path family p_a (which ends at a), DOSPD calls *getDefClearPathFamily* (line 20, not shown) to compute path family p' describing all paths from s that reach a and then continue to u within the constraining p and without redefining the variable in u. For example, if u is m_3, a is m_0, and p_a is $F_{EN\rightarrow EN}$, then p' is <EN,3,{<1F>}>. p' includes the aliasing conditions that specify that the variable of a is the same as the variable of u and that, for each potential re-definition k for a, the paths through k are either excluded or the variable at k differs from the variable at a. For example, for a definition of field $h.f$ that might reach a use $q.f$ within p, p' includes a special condition stating that the symbolic value of h in p_a equals q at the end of p'. The resulting pair $<a,p'>$ is then linked to $<u,p>$ at line 21 as a reaching definition or input for u in p—if p and p' are satisfied, then $u = a$.

4.2.6.4 Additional Uses

Identifying the definitions and inputs for u and their reaching conditions is, however, not sufficient for obtaining the symbolic value of u in p. All uses on the right-hand side (i.e., defining) expressions at definitions, as well as all uses in path family p', also have to be backward expanded until only input symbols are left. Thus, in lines 22–24, DOSPD adds to *workset* every use u_a on the right-hand side[8] of a, associating each u_a with path family p_a, which ends at u_a. For example, if a is the definition of sz at 3, then pair <m3,<EN,3,{<>}>> is added to *workset*. Meanwhile, lines 25–30 identify uses u_c in the terms of PFC p' (including aliasing conditions), pairing each use with the prefix PFC p_c of p' that ends at that use, and adds those pairs to *workset*. For example, if p' is <EN,7,{<4T,6T>}, then, for term 6T, pair <q_6,{<4T>}> is added to *workset*. Note that DOSPD only adds pairs to *workset* if they are not already backward expanded; this is how SPD reuses computations. Finally, the working pair is marked as expanded at line 32.

4.2.6.5 Symbolic Result

When the *while* loop ends, all uses in *PFC(s→t)* and V, as well as all uses transitively reachable through definitions and PFCs linked to those initial uses, have been expanded within their respective constraining path families. At this point, the linking process has produced an acyclic graph of use-path pairs that induces a case-like symbolic expression for all variables of interest in terms of the symbolic inputs. If needed, the explicit symbolic values for *PFC(s→t)* and V can be obtained by replacing all use-path pairs with their linked definitions and inputs and continuing through the use-path pairs on right-hand-side expressions and PFCs for those reaching definitions and inputs. However, the graph form of the result is, in general, more convenient than the full symbolic expression because it avoids the redundancy caused by multiple replacements of intermediate nodes. These nodes can be interpreted as special variables (e.g., <sz$_{11}$,$F_{EN\rightarrow 11}$>), keeping the size of

[8] The right-hand side for an input is empty.

the expression under control without losing any information.

4.2.6.6 *Complexity and Termination*

The worst-case complexity of SPD is exponential in the number of program points that affect *PFC(s→t)* or V because SPD does a path-sensitive traversal of the data- and control-dependence graphs. This complexity, however, is not greater than that of TRADSE because, unlike TRADSE, SPD does not traverse points or combinations of ICDG edges that do not affect the result. SPD also requires a dependence analysis of the program, but the cost of this analysis is only polynomial.

If the dependencies affecting *PFC(s→t)* or V are cyclic, DOSPD might not terminate. Thus, the abstraction function should be designed to identify such situations and abstract away uses to enforce termination. An alternative is to find closed-form expressions for the effects of cycles [28]. When this is not possible, however, DOSPD could be made to limit the number of iterations per loop—the typical approach used in the literature (e.g., [8, 24]).

4.3 *Implementation of* SPD

To support the evaluation of SPD, we implemented this technique as a tool called JSPD that we describe in this section. JSPD is based on DUA-FORENSICS [93], an interprocedural control- and data-dependence analysis tool that uses Soot [106] to analyze Java-bytecode programs. JSPD checks the feasibility of path families using the CVC3 SMT solver [11]. Also, to better analyze realistic Java programs that use libraries and contain loops, JSPD uses two mechanisms: (1) manually-encoded models of the effects of library calls to focus the analysis on the application code rather than the library code, and (2) user-provided limits for the length of control-dependence edge lists for path families and for the number of iterations per loop. By default, JSPD sets these limits to 10 and 2, respectively. The length limit might seem small, but long control-flow paths can exist within this limit and all such paths are analyzed by JSPD.

For abstraction, JSPD uses a generic strategy parameterized by a symbolic expression-tree depth limit e: JSPD replaces with $*$ all symbolic terms "below" depth e in the resulting symbolic expression trees. For example, when computing a PFC to a certain point, the uses in the conditions for reaching that point are located at depth 0, the PFCs and definitions for those uses are located at depth 1, and so on. The goal of this strategy is to abstract uses and PFCs that are "less important" than the uses and PFCs at depth e or less. Thus, the resulting symbolic expressions are over-approximated as multi-valued expressions (see Section 4.2.3).

In addition to SPD and this abstraction strategy, we implemented TRADSE using the same JSPD tool to provide a fair comparison during our experiments. To that end, we added a mode to JSPD in which all final path families are individual paths.

4.4 Study: Path-space Reduction

The goal of this study was to assess the gains in efficiency that SPD can achieve for multiple-path symbolic execution by measuring the reduction in the number of paths or path families that SPD (without abstractions) analyzes with respect to TRADSE for the same result.[9] First, we present the empirical setup, then, we analyze the results, and finally we discuss threats to the validity of this study.

4.4.1 Empirical Setup

Table 3 lists the subjects we used in our study. For each subject, the first column gives the name, the second column provides a short description, and the third column shows the size in non-comment non-blank lines of Java source code. Tcas, Tot_info, Schedule1, and Print_tokens1 are small programs from the Siemens suite that we translated from C to Java. NanoXML and XML-security are libraries used by many systems. We obtained these subjects from the SIR repository [36].

[9]For a study of the benefits of SPD and abstractions for computing change effects, see Section 5.4.2 in Chapter 5.

Table 3: Subjects for the study of SPD.

subject	description	lines of code
Tcas	air collision avoidance	131
Tot_info	information measure	283
Schedule1	priority scheduler	290
Print_tokens1	lexical analyzer	478
NanoXML	lean XML parser	3497
XML-security	signature and encryption	21613

For this experiment, we symbolically executed all paths without abstractions, bounded by the default length and iteration limits of JSPD, between pairs of statements in the subjects listed in Table 3. For each of Tcas, Tot_info, Schedule1, and Print_tokens1, we randomly selected 10 statements and computed the PFCs for reaching those statements from the entry of the program. For each of NanoXML and XML-security, we randomly chose 10 methods and 10 target statements (anywhere in the program) and computed the PFC from the entry of each method to the corresponding target statement. We considered only target statements for which JSPD found more than one path from the respective method.

To measure the cost of using SPD for computing each PFC, we counted the number of path families required by SPD and compared it with the number of control-flow paths required by TRADSE. For both techniques, we counted only the path families that CVC3 did not find to be infeasible. We also measured the cost in time incurred by SPD to construct the path families.

We ran our experiments on a dual-core 3 GHz machine with 2 GB of RAM running 32-bit Linux and Sun's Java virtual machine.

4.4.2 Results and Analysis

Table 4 shows, for each subject and PFC (first column for each subject), the number of control-flow paths and path families analyzed using TRADSE and SPD, respectively (second and third columns for each subject). The last two rows show, for each subject, the average number of paths per technique and the average ratio of TRADSE paths to SPD path

Table 4: Paths and path families explored to compute PFCs in the study. Path-family construction had a comparatively negligible cost.

# paths Tcas		
PFC	TRADSE	SPD
1	2,520	71
2	40	15
3	18	11
4	280	27
5	120	23
6	1,680	31
7	2,240	35
8	13,680	79
9	13,760	83
10	18	11
avg.	3,436	39

# paths Tot_info		
PFC	TRADSE	SPD
1	2	2
2	58,741,054	122
3	2	2
4	2	2
5	2	2
6	112	2
7	112	2
8	40,332,449	122
9	40,332,449	122
10	6,538,717	122
avg.	14,594,490	55

# paths Schedule1		
PFC	TRADSE	SPD
1	965,353	338
2	53,301	126
3	83,915,178	1,695
4	686,090,932	1,990
5	64	8
6	64	8
7	343,609,576	1,016
8	28,189,300	554
9	2,541,388	122
10	4,939,084	135
avg.	115,030,424	599

# paths Print_tokens1		
PFC	TRADSE	SPD
1	1,082,127,580	400
2	642,994,146	638
3	642,994,146	642
4	2	2
5	642,994,146	642
6	594,023,890	630
7	594,023,890	624
8	594,023,890	626
9	642,994,146	642
10	642,994,146	642
avg.	607,916,998	549

# paths NanoXML		
PFC	TRADSE	SPD
1	2,621	50
2	8,001	22
3	195	15
4	3,347	53
5	195	15
6	1,732	29
7	112	84
8	19,352	26
9	4,276	35
10	4,056	35
avg.	4,389	36

# paths XML-security		
PFC	TRADSE	SPD
1	23,039	43
2	490	57
3	201	14
4	145	21
5	9,997	691
6	115,781	118
7	15,882	31
8	29,737,820	275
9	463	52
10	19,813	100
avg.	2,992,363	140

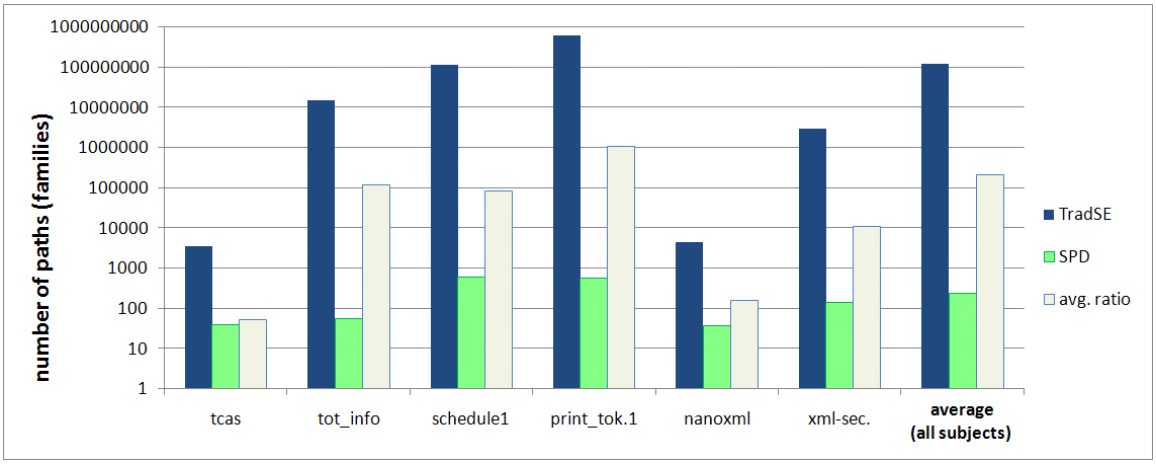


Figure 11: Average number of feasible paths or path families analyzed for TRADSE and SPD, and average ratios of these values.

families explored. For example, for PFC 1 in Tcas, TRADSE had to explore 2,520 paths, whereas SPD only explored 71 path families. On average for Tcas (second-to-last row), TRADSE explored 3,436 paths per PFC, whereas SPD covered the same paths by exploring only 39 path families. Figure 11 presents, in a logarithmic scale, the average number of paths or path families explored by TRADSE and SPD per subject and for all subjects. These numbers are shown by the first two bars—TRADSE and SPD—for each case. The third bar for each case, *avg. ratio*, shows the average of the ratios of paths explored by TRADSE to the path families explored by SPD. For example, the ratio of explored paths for PFC 1 in Tcas is 35 : 1 (not shown) and, for all PFCs in Tcas, the average ratio is 51 : 1 (see also Table 4). In Figure 11, the average of ratios for all 60 PFCs is 211,933 : 1.

The cost of constructing path families using SPD was usually a few seconds—negligible compared with the dozens or even hundreds of minutes it took to explore the paths and path families.

For most PFCs, the results show a large reduction in the number of path families explored by SPD to compute the PFCs with respect to the number of paths that must be explored by TRADSE. For many cases in Tot_info (e.g., PFC 2), Schedule1 (e.g., PFC 4), and Print_tokens1 (e.g., PFC 1), and for one case in XML-security (PFC 8), the reduction

is especially dramatic. Overall, for all subjects, there is an average reduction of orders of magnitude in exploration costs of using SPD with respect to TRADSE. For Tcas, the reduction is less than for the other subjects—"only" 51 times on average as the last row of Table 4 shows—which is explained by the small size of this subject and the absence of loops. For Tot_info, Schedule1, and Print_tokens1, which are only slightly larger than Tcas but more complex and contain loops, the average reduction is very large. The results for these three subjects suggest that considerable portions of their code contain series of independent computations that SPD does not analyze in combination but that lead to a dramatic explosion in the total number of control-flow paths. For NanoXML, and especially XML-security, which are real-world subjects, the average savings ratios are also considerable: 155 and 11,042 times, respectively. Note that, for XML-security, PFC 8 greatly influences this average; without that PFC, the average ratio is 253 : 1. These results confirm that important savings can also be achieved for larger and more complex subjects than the Siemens subjects. The variety of these results also reflects the diversity in the complexity of code across different programs and within each program. Also, recall that these results are constrained by the edge-list length and iteration limits discussed in Section 4.3; it is reasonable to expect that these differences keep growing at similar rates if those limits are increased.

The results of this study highlight the significant potential of SPD for shrinking path-exploration costs, which impacts a variety of applications. These results make it clear that exploiting the dependence structure of programs is crucial for making symbolic execution of multiple paths more scalable than current approaches. Note that SPD requires a dependence analysis to operate, which can be computed beforehand or on demand. However, dependencies often are already computed by many client analyses, and even if they are not, the cost of dependence analysis grows only polynomially with the size of the program, whereas path exploration is an exponential-cost problem that dominates the overall analysis cost.

4.4.3 Threats to Validity

The main internal threat to the validity of our results is the potential presence of errors in our JSPD tool, including our manual modeling of Java library methods. To minimize this threat, we tested JSPD on diverse examples and subjects, making extensive use of runtime checks and manually examining results.

The main external threat is the representativeness of our subject programs. To reduce this threat, we used subjects of different styles and purposes. Yet, PFCs on more subjects of different kinds need to be studied to generalize the conclusions of our studies.

4.5 Related Work

Symbolic execution was first introduced as a forward analysis of control-flow paths [27,63]. Backward substitution was later presented as an alternative method for performing symbolic execution and the term *global symbolic evaluation* was used to refer to the symbolic execution of all program paths [28]. Our SPD technique, in fact, performs global symbolic evaluation between two points in a program using backward substitution. SPD, however, analyzes paths in groups instead of individually, is driven by program dependencies instead of control-flow, and provides an over-approximation mechanism to reduce complexity. These features of SPD greatly improve the scalability of global symbolic evaluation.

Software model checking [9, 14, 107] exhaustively explores all sequences of states in programs using state abstractions (e.g., symbolic states) and refinement to reduce the search space with respect to the checked properties. Most software model checkers, however, work on the control-flow graph of the program, whereas SPD exploits a new kind of abstraction—path families—to explore paths in groups. Some techniques group paths as methods [8, 24], but methods are coarse abstractions—paths still must be analyzed individually after a method is expanded. SPD, in contrast, provides a fine-grained grouping of paths, based on control and data dependencies, which groups paths regardless of method

boundaries and is based strictly on the similarity of their effects on each separate component of the final result.

Path slicing [61] removes irrelevant portions of a counterexample path for effective refinement of state abstractions in software model checking. Interestingly, a sliced path corresponds to a path family described by the preserved portions of the original path. However, a sliced path is equivalent to only one of the "final" path families produced by SPD—the path families that are not partitioned any further. Also, unlike path slicing's bottom-up approach, SPD works in a top-down fashion by partitioning path families without requiring any initial individual control-flow path.

Bug-finding approaches (e.g., [24, 37, 43, 73]) generate *verification conditions* (VCs) that logically describe programs along with the properties to check. Efficient versions of bug-finding approaches that are based on the concept of weakest preconditions [35] control the size of the VC by summarizing the effects of code fragments [10, 43] or by performing a demand-driven backward analysis from the points of interest [24, 73]. These techniques, however, operate on control-flow paths instead of taking advantage of program dependencies and thus, unlike SPD, they may analyze irrelevant portions of the code. One bug-finding technique that does take dependencies into account is Miniatur [37], which slices the VC to remove the portions of the program that are irrelevant for the checked property. SPD, however, performs slicing at the path level, which permits multiple occurrences of the same use (i.e., a variable and a location) by distinguishing its context—the constraining path family. SPD only associates with each pair of use and context the reaching definitions and inputs for that context. Moreover, SPD can also over-approximate a use with $\star$. Another technique [102] also uses control dependencies to compute path conditions between two points, but this technique only partially expands the conditional terms as functions of the input, so in general it does not produce a complete symbolic-execution result.

Many representations of programs for compiler optimization have been proposed. The

Value Dependence Graph (VDG) [111], in particular, uses program dependencies to transform loop bodies and procedures in a way that is similar to SPD. Like Miniatur, however, the VDG is constrained to one occurrence of each value (variable use), whereas SPD works on use-path pairs to distinguish contexts, integrates aliasing conditions, and provides an over-approximation mechanism. On a related note, the *static single assignment* (SSA) form [33] or some variant of it is commonly used—implicitly or explicitly—by many of these techniques to distinguish different values of variables coming from different paths. SPD lazily determines at each use location, and for each path family, the value of the variable based on the reaching definitions and inputs within that path family. However, it remains to be seen whether using SSA's phi-functions within each path family provides any improvement for SPD.

Uninterpreted functions are used for operations or code segments whose semantics are unknown or not supported by solvers. They are often associated with abstract behaviors (e.g., [22]) to simplify verification. They are also used for symbolic execution when comparing two programs to skip the analysis of identical segments [32, 78, 100]. The value $\star$ in SPD can be seen as an uninterpreted function that replaces terms in symbolic expressions with a particular meaning: top. This is the same meaning assigned by some analyses (e.g., [14, 38]) to unsupported operations such as multiplication. SPD, however, uses $\star$ more generally to abstract away arbitrary subtrees of the resulting expression tree.

Researchers have combined symbolic analysis with data-flow analysis to achieve trade-offs between precision and efficiency (e.g., [15, 42, 104]). Most of these approaches introduce symbolic predicates to the lattice of data-flow facts to increase precision. SPD, however, starts with a fully path-sensitive analysis and only performs approximations based on a user-provided strategy to avoid the complexity of expanding certain subterms. In that sense, SPD can be seen as an instantiation of the framework of configurable program analysis [15].

Finally, a number of test-input generation techniques are based on symbolic execution,

including “dynamic” symbolic execution (DSE) [47, 99, 116] (also called “concolic” testing). We believe that our approach, based on path families, can improve these techniques, which use a smart exploration of individual control-flow paths. We also expect that modular analysis based on SPD will improve the control-flow based approach for compositional DSE [3].

CHAPTER V

ANALYSIS AND TESTING OF INDIVIDUAL CHANGES

This chapter presents and studies a principled and practical new technique for computing testing requirements for *individual* changes (i.e., regardless of other changes). This technique is rooted in the formal model and computational approach for effects of changes presented in Chapter 3. To begin, Section 5.1 addresses existing *coverage-based* techniques for testing changes (i.e., techniques based on covering program entities) and provides the first empirical evaluation of these techniques. Section 5.2 presents our new technique for testing changes, which we regard as *propagation-based* because of its roots in the PIE model (Section 2.2) and Chapter 3. Then, Sections 5.3 and 5.4 describe the implementation and evaluation, respectively, of this new technique. Finally, Section 5.5 discusses related work.

5.1 Study of Coverage-based Testing of Individual Changes

A major problem with existing research on coverage-based testing of changes [17, 49, 87] (described in Section 2.3.2) is that no empirical evaluation of these coverage criteria has been performed. The only hints about the potential strength of these criteria come from their reliance on forward static slicing [57, 112], which can be considerably imprecise [18, 23]. The most closely related empirical studies available correspond to applications of change-impact analysis (e.g., [20, 21, 68, 76, 83]) that identify program entities potentially affected by a change at some location. However, change-impact analysis techniques usually identify entities of coarser granularity than statements, such as methods and classes. Therefore, empirical studies of the existing coverage-based strategies for testing changes are needed.

The goal of this section is to provide a preliminary empirical evaluation of the ability of

Table 5: Subjects for preliminary change-testing studies.

subject	description	lines of code	test cases	changes
Tcas	air traffic	131	1608	6
NanoXML-v1	XML parser	3497	214	7
NanoXML-v5	XML parser	4782	216	2

coverage-based approaches to reveal observably-different behaviors caused by changes—behaviors that need inspection by testers—and identify the necessities of further research.

5.1.1 Empirical Setup

To empirically evaluate coverage criteria representative of existing work, we extended the DUA-FORENSICS [93] dependence-analysis tool to monitor the coverage of affected branches and du-pairs in program executions after a change is covered. DUA-FORENSICS is implemented in Java and uses the Soot Framework [91, 106]. DUA-FORENSICS can determine the *dependence distance* from the change (i.e., the number of dependencies traversed from the change) at which each testing requirement is covered.

Table 5 lists the subject programs used in this study, their sizes in non-comment, non-blank lines of code, the size of the pool of test cases provided with the subject, and the number of changes studied per subject. The pools of test cases for these subjects are designed to be representative of the kinds of test cases that developers conceive in practice while providing enough test cases to construct different test suites satisfying the same criterion, for different coverage criteria. The changes were made by other researchers for their own studies.

The first subject, Tcas, is an air traffic collision-avoidance algorithm for avionics systems. This subject was adapted by Hutchins and colleagues for coverage-testing studies [60] and is representative of modules with straightforward logic found in many industrial domains. We used a version of Tcas translated to Java from the original version in C, and considered its first six changes. For a second subject, we used NanoXML, an XML parser available at the SIR repository [36, 40] that, in contrast to Tcas, represents

more complex, object-oriented software. We used all changes provided with version *v1* of NanoXML. For two of these changes, however, all test cases covering those changes reveal differences in the output. To compensate for these "uninteresting" changes, we included two changes from version *v5* of NanoXML (the version most different from *v1*). The first two changes in *v5* are located in unreachable code, as we confirmed manually, so we used the third and fourth changes provided with that version.

The existing coverage-based approaches define two types of entities affected by changes: branches and du-pairs. DUA-FORENSICS finds these entities using forward static slicing and monitors their coverage at runtime. For completeness, we also studied the coverage of the statements modified by the change, which the simplest way of testing a change. Also, to support a side-by-side comparison with the technique presented next in Section 5.2, we considered only those testing requirements (branches and du-pairs) covered within the same distance that the toolset for the technique of that section was able to reach. A more comprehensive study of coverage criteria is presented in Chapter 6 as part of the evaluation of the demand-driven alternative for the new technique.

In this study, for each change, we used the available pool of test cases to construct test suites for each criterion that maximize the coverage achievable for that criterion and that pool. To minimize randomness in the results for individual test suites, we created 100 different test suites for each coverage criterion studied. We constructed each test suite by picking from the pool one random test case at a time without replacement and adding that test case to the test suite if it increased the coverage of the test suite. To estimate the probability that using each criterion reveals a difference in the output, we computed the percentage of test suites for that criterion that revealed a difference in the output—our measure of success.

Table 6: Difference detection for test-suite augmentation criteria on Tcas.

change	distance	STMT	BR	DU	CHAIN	PROP
1	3	41.1%	41.1%	47.3%	47.3%	100%
2	6	15.2%	15.2%	15.2%	16.9%	54.1%
3	6	2%	3.2%	2%	18.4%	18.6%
4	6	10%	19.4%	10%	100%	100%
5	6	0.7%	0.7%	0.7%	3.5%	3.7%
6	5	2.1%	3.7%	2.1%	4.6%	100%

5.1.2 Results and Analysis

Tables 6 and 7 show the results of this study, using DUA-FORENSICS, for the changes in Tcas and NanoXML, respectively. In each table, column *change* identifies the change with a number and *distance* (or d) the dependence distance reached by the technique of Section 5.2. The change numbers reflect the order in which they are provided with each subject. For NanoXML, changes 1–7 are from version *v1* and changes 8 and 9 are from version *v5*. Distance d is specified only when simply covering the changed statements does not guarantee a difference in the output. Otherwise, *distance* is 0 and denoted by a dash (-).

Columns STMT, BR, and DU in these tables report the probability of revealing differences in the output using test suites satisfying the three coverage criteria: all changed statements (STMT), all affected branches within distance d of the change (BR), and all affected du-pairs within distance d of the change (DU).[1] When the BR and DU criteria are not applicable (i.e., when the change only affects control flow or data flow, but not both), they default to STMT, which corresponds to testing requirements at distance 0.

To illustrate, consider Change 1 in Tcas (Table 6). The table indicates that 41.1% of the test suites covering the requirements set by STMT or BR within distance 3 revealed a difference in the output, while 47.3% of the test suites covering DU at distance 3 or less revealed a difference—a slight increase over STMT and BR.

[1]The last two columns in these tables are used in Section 5.4.

Table 7: Difference detection for test-suite augmentation criteria on NanoXML.

change	distance	STMT	BR	DU	CHAIN	PROP
1	2	52.2%	100%	52.2%	100%	100%
2	3	67.2%	67.2%	67.2%	67.2%	67.2%
3	3	0%	0%	10.4%	10.4%	10.4%
4	3	13.4%	13.4%	48.7%	48.7%	100%
5	-	100%	100%	100%	100%	100%
6	-	100%	100%	100%	100%	100%
7	3	18.7%	18.7%	18.7%	18.7%	18.7%
8	4	35%	35%	35%	66%	66%
9	4	21.9%	21.9%	21.9%	42.2%	42.2%

The results show that, for most of these changes, coverage criteria do not provide confidence that differences in the output will be revealed. The probabilities in those cases are far less than 100%, even though, for every change, there is at least one test case in the pool that produces a difference. Therefore, the three coverage-based approaches are not capable of describing with enough detail the conditions that distinguish difference-revealing test cases from the rest.

The exceptions are changes 1, 5, and 6 in NanoXML (Table 7). Changes 5 and 6 are trivial in that, using this test pool, simply covering the modified statements guarantees that a difference is observed (hence, d is not specified). For Change 1, covering affected branches guarantees a difference, but not affected du-pairs or changed statements. In all, for most changes and criteria, the chances of observing a difference in the output are low—less than 50% for all changes except for the aforementioned exceptions and Change 2 in NanoXML.

This initial study suggests that existing coverage-based approaches for testing changes, at least under distance constraints, do not provide enough confidence that changes are exercised in ways that propagate their effects to the output. Thus, better approaches are needed.

5.2 Propagation-based Testing of Individual Changes

A careful analysis of coverage-based approaches and their preliminary evaluation from Section 5.1 suggests that their deficiencies stem from failing to exploit all the information

that a change provides. Coverage-based approaches require the coverage of the changed locations and potentially-affected entities but do not consider how the change affects the program state, how those entities are affected, and how the output is affected when those entities are affected. Approaches based only on the coverage of the change and affected entities only guarantee that the change is executed, as required by the PIE model adapted for changes (Section 2.2), but they do not provide sufficient guarantees that the state of the program is infected by the change and that the infection propagates to the output.

This section presents our new propagation-based technique for testing changes that explicitly addresses the infection and propagation requirements of the PIE model. This technique computes and monitors test-suite augmentation requirements (see Section 2.3) for individual changes based on a precise analysis of those changes. A preliminary version of this technique was presented in Reference [6] and then improved and evaluated in References [92] and [94].

5.2.1 Overview

The technique adapts the procedure COMPUTEEFFECTS of Figure 6 in Section 3.2, which computes the effects of a change on any point in the program, to make it applicable in practice. The technique works in two phases. Phase 1 identifies the *dependence chains* in the program (Definition 22) through which the effects of a change might propagate to the output. These chains are what we call the *chain* testing requirements in this technique—they should be covered as a necessary (albeit not sufficient) condition for exercising the many paths through which the effects of a change might propagate. Phase 2 uses *partial symbolic execution* (PSE) (Definition 23) to compute the *infection* and *propagation* conditions (i.e., the *state conditions* of Section 3.2.2) for the effects of the change along each dependence chain found in Phase 1. The conditions computed in Phase 2 for each chain are what we call the *state* testing requirements for those chains.

Computing all the requirements for covering a change, infecting the state, and propagating the infection to the output is impractical in general. Hence, the technique makes two simplifications. First, the technique omits the conditions for reaching the change from the entry of the program that COMPUTEEFFECTS obtains in Line 6 because an existing test suite created for high code coverage should already cover the change. Second, the technique sets a limit d on the *distance* from the change (i.e., the length of each chain from each node in the change) that the analysis is allowed to reach along each chain. This limit can be specified by the developer or a tool that uses this technique. The testing requirements obtained by this technique are, for two reasons, a safe approximation of the conditions for the execution, infection, and propagation of changes to the output. The first reason is that a test case that does not satisfy these requirements on a chain is guaranteed not to propagate an infection to the output through that chain. The second reason is that these requirements are a necessary but not sufficient condition for test cases to propagate an infection to the output through a chain. Thus, by requiring the propagation of an infection to distance d on each chain from the change, the technique aims at increasing, with respect to a coverage-based approach, the probability that the infection will keep propagating beyond that distance and eventually affect the output. The emphasis on propagation is the reason why we classify this technique as *propagation-based*.

Figure 12 provides an intuitive view of the way the technique works. The goal is to create requirements for the *difference-revealing* subset of all possible test cases, which are the test cases that produce an observably-different behavior in the original and modified programs. This subset is first approximated by test cases that just cover the change (i.e., criterion STMT in Section 5.1). From that point, the ideal subset is increasingly better approximated by subsets that satisfy the chain and state requirements for each distance because, as the distance increases, the test suites that satisfy those requirements restrict more and more the test cases that can possibly propagate an infection to the output. For each distance d, the requirements subsume BR and DU (Section 5.1) within that distance.

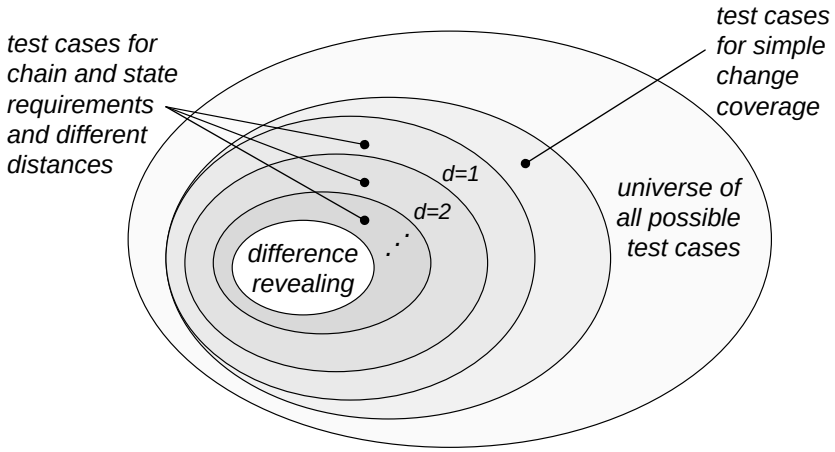


Figure 12: Intuitive view of proposed testing criteria for evolving software.

5.2.2 Technique

This section presents and illustrates in detail the propagation-based technique for analyzing and testing changes individually. Section 5.2.2.1 presents Phase 1, while Section 5.2.2.2 presents Phase 2.

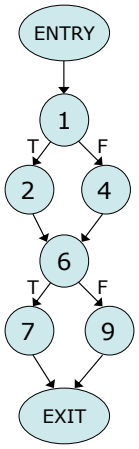
5.2.2.1 Phase 1: Chain Requirements

A simple strategy for testing a change is STMT—cover all changed statements. However, as shown in Section 5.1, this strategy often fails to guarantee that all possible effects of the change on different parts of the program are tested. Consider, for example, program E in Figure 13 and change `ch1`. Let E_1 be this modified version. This program has $10 \times 10 = 100$ test inputs. Examining the code in Figure 13 reveals that the change produces a different observable behavior in E_1 (i.e., a different output with respect to E) if and only if $y \in [2, 3]$, which corresponds to 20 inputs, or 20% of the input space. Hence, randomly picking an input that covers the change has a rather small chance of causing a difference.

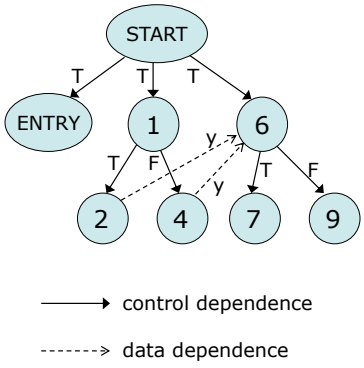
In many cases, an adequate test suite for the simple change-coverage strategy exercises only a small fraction of all dependence chains along which a change may propagate. Because the effects of a change can propagate forward along any of the chains starting from the change, an adequate testing strategy should require the coverage of all such chains.

```
program E(int x, y) // x,y ∈ [1,10]
1.  if (x <= 2) // ch1: if (x > 2)
2.     ++y
3.  else
4.     --y
5.  // ch2: y *= 2
6.  if (y > 2)
7.     print 1
8.  else
9.     print 0
```

(a)



(b)



(c)

Figure 13: Example program E with two changes (a), its CFG (b), and its PDG (c).

Table 8: Dependence chains in program E and change `ch1` from Figure 13.

distance	chains	inputs	differences
1	$q_{1,1} = <1,(1,2)>$	80	16
	$q_{1,2} = <1,(1,4)>$	20	4
2	$q_{2,1} = <1,(1,2),(2,6,y)>$	80	16
	$q_{2,1} = <1,(1,4),(4,6,y)>$	20	4
3	$q_{3,1} = <1,(1,2),(2,6,y),(6,7)>$	72	16
	$q_{3,2} = <1,(1,2),(2,6,y),(6,9)>$	8	0
	$q_{3,3} = <1,(1,4),(4,6,y),(6,7)>$	14	0
	$q_{3,4} = <1,(1,4),(4,6,y),(6,9)>$	6	4

Table 8 shows all dependence chains reaching distances 1, 2, and 3 from the change (Statement 1) in programs E and E_1—the chains of lengths 1, 2, and 3, respectively. A chain $q_{n,i}$ corresponds to the i^{th} chain of length n. For each chain, columns *inputs* and *differences* show the number of inputs that cover the chain and the number of covering inputs that also reveal a difference in the output.

Consider the requirement of propagating the effects of the change along all chains of length one (i.e., along direct dependences) from this change. Two control dependences must be tested: $(1,2)$ and $(1,4)$. Table 8 shows that, for $(1,2)$, there are 80 covering inputs, 16 of which reveal a difference. For $(1,4)$, there are 20 possible inputs, four of

which reveal a difference. Each chain has a probability of 20% of revealing a difference. A test suite T that covers all dependencies from the change in this example has thus a 36%[2] chance of revealing a difference. Therefore, covering all chains of length one increases the probability of difference detection from 20% to 36% with respect to simply covering the change.

Consider now dependence chains of length greater than one. The number of chains can grow exponentially with their length, but in practice it is possible to cover chains of a limited length. In the case of change `ch1`, there are four chains of length 3 reaching output statements 7 and 9. Two of these chains can reveal a difference, as Table 8 shows: $q_{3,1}$, which is covered by 72 inputs, revealing a difference in 16 cases (22.2%), and chain $q_{3,4}$, which is covered by six inputs, revealing a difference in four cases (66.7%). A test suite that covers all chains of length 3 would thus reveal a difference with probability 74.1%. A random test suite of size 4, in comparison, has a 59% chance of revealing a difference. This example shows that longer chains can improve the chances of revealing a different behavior after a change, compared to shorter chains or simple change coverage (i.e., chains of length 0) by requiring a well-distributed coverage of propagation paths from the change.

Figure 14 presents algorithm COMPUTEREQS, which computes the testing requirements for a set of changes. COMPUTEREQS is a practical variant of procedure COMPUTEEFFECTS (see Figure 6 in Section 3.2) for computing the effects of changes. Like COMPUTEEFFECTS, COMPUTEREQS inputs program P and change C. COMPUTEREQS, however, also inputs two parameters, d for the maximum distance (length) for the chains starting at the change and l for the maximum path length. These parameters are used to guarantee termination in a reasonable amount of time. Distance d is the maximum length of the dependence chains, as explained in Section 5.2.1. Limit l is discussed in Section 5.2.2.2.

[2]The probability of T revealing a difference is $1 - \prod_{t \in T}(1 - p_t)$, where p_t is the probability of test case t revealing a difference.

Algorithm COMPUTEREQS

Inputs: P: program; C: change; d: distance limit; l: path length limit
Output: *requirements*: map *program* → *dependence chain* → *conditions*

```
(1)   P' := applyChange (P, C)
(2)   foreach program ∈ { P, P' }
(3)     altProgram := P' if program is P ; P otherwise
(4)     chainWorklist := createEmptyChains (program, change)
(5)     foreach emptyChain ∈ chainWorklist
(6)       requirements[program][emptyChain] := ∅
(7)     endfor
        // incremental computation of conditions
(8)     while chainWorklist ≠ ∅
(9)       prefixChain := pop (chainWorklist)
(10)      foreach dep ∈ nextDependencies (program, prefixChain)
(11)        chain := append (prefixChain, dep)
            // chain requirements
(12)        depCond := dep if isRealizable (program, prefixChain, dep) ; false otherwise
(13)        chainCond := requirements[program][prefixChain] ∧ depCond
            // state requirements
(14)        (S, S_alt) := PSE_Dep_Limit (program, altProgram, dep, l)
(15)        stateCond := live(S) ≠ live(S_alt) ∨ (pc(S) ∧ ¬ pc(S_alt))
(16)        requirements[program][chain] := chainCond ∧ stateCond
(17)        if length (chain) < d
(18)          push (chainWorklist, chain)
(19)        endif
(20)      endfor
(21)    endwhile
(22)  endfor
(23)  return requirements
```

Figure 14: The algorithm for computing chain and state testing requirements.

The computation of chain requirements in COMPUTEREQS is identical to the computation of chain conditions in COMPUTEEFFECTS except for two differences. The first difference is that, at Line 6, the conditions for reaching the change are empty—these conditions are omitted for the reasons given in Section 5.2.1. The second difference is that, at Lines 17–19, the extended chain is added to the work list only if its length has not yet reached distance d.

In the rest of this dissertation, we use CHAIN to represent the strategy of satisfying chain requirements during testing.

5.2.2.2 *Phase 2: State Requirements*

Chain requirements can increase the probability of finding output differences, but they cannot guarantee that the effects of the change propagate to the end of each chain. For example, for program E and change `ch1` in Figure 13, Table 8 shows that only 16 out of 72 inputs (22%) that cover chain $q_{3,1}$ reveal a difference in the output. The reason is that the program state after covering chain $q_{3,1}$ and reaching Statement 7 in E might be the same as the state at Statement 7 in E_1 for the same input. Hence, to guarantee propagation of the infection to the end point of each chain—thus increasing the chances that the infection will keep propagating after that point and reach the output—algorithm COMPUTEREQS computes state requirements for each of the chains.

Like procedure COMPUTEEFFECTS, algorithm COMPUTEREQS performs partial symbolic execution (PSE) (Definition 23) on each dependence to obtain the state conditions for Cases 1 and 2 (Section 3.2.2) in which P and P' can differ: some live variable differs or the path conditions differ. These two state requirements for a chain reduce the space of test cases covering the chain to those that propagate an infection to the end of the chain, thus increasing the chances of propagating the infection to the output. However, unlike COMPUTEEFFECTS, COMPUTEREQS uses at Line 14 a modified version of PSE that takes into account the limit l provided as input.

In COMPUTEREQS, PSE stops analyzing a control-flow path that covers a dependence when this path reaches length l, thus avoiding the costly analysis of paths that are too long—perhaps infinitely long. When PSE stops analyzing a path, it stops adding terms to the conjunction that is the path condition for that path. In that case, the incomplete path condition might be satisfied even though, in reality, the path might not be covered because some condition after the l^{th} node in the path is not satisfied. Similarly, if a variable v is updated in a path after the limit l for that path is reached, the value v is considered *unknown*—it can have any value. In such a case, whether v differs in the original and modified programs cannot be determined (unless v does not exist in one of the programs).

To illustrate, consider again Figure 13. Using symbolic execution, the technique defines the state of E_1 at the change point as $\{x = x_0, y = y_0\}$. After following chain $q_{3,1}$, the symbolic state in E_1 is $\{x = x_0, y = y_0 + 1\}$, and the path condition for this chain is $x_0 > 2 \wedge y_0 > 1$. In the example, both x and y are dead at Statement 7, so the live states are the same in E_1 and E, that is, Case 1 for $q_{3,1}$ is not satisfied. However, the path condition in E for all paths to Statement 7 is $(x_0 \leq 2 \wedge y_0 > 1) \vee (x_0 > 2 \wedge y_0 > 3)$, so Case 2 for $q_{3,1}$ ($pc(S') \wedge \neg pc(S)$) is satisfied by $x_0 > 2$ and $y_0 \in [2, 3]$. Condition 2 reduces the number of test cases that cover the chain from 72 to 16, which are exactly those revealing a difference. Constraining the other difference-revealing chain, $q_{3,4}$, also yields a 100% detection.

In the rest of this dissertation, we use PROP to represent the strategy of satisfying CHAIN (chain requirements) and the state requirements for each such chain during testing.

5.3 Implementation of Propagation-based Technique

To support our evaluation of the new propagation-based testing requirements, we implemented COMPUTEREQS in Java as part of a tool called MATRIXRELOADED[3] that uses DUA-FORENSICS [93] and Soot [91,106] to analyze and instrument software in Java bytecode format. MATRIXRELOADED consists of two main parts: (1) a program analyzer that implements algorithm COMPUTEREQS (Figure 14) to identify test-suite augmentation requirements, and (2) a monitoring component that instruments the original and modified programs and monitors at runtime the coverage of the augmentation requirements.

MATRIXRELOADED reuses the extension of DUA-FORENSICS, described in Section 5.1.1, to identify chains of control and data dependences from any point in the program. The other key component of MATRIXRELOADED is a symbolic-execution engine that

[3]In Reference [6], we presented and used an earlier and partial implementation that we called MATRIX.

supports partial symbolic execution (PSE), which systematically explores all paths between a pair of arbitrary points in the program. The symbolic states computed by MATRIXRELOADED are expressed in terms of the symbolic values of live variables at the entry of the starting point. In the implementation for this study, the symbolic executor visits each statement at most twice and replaces the Java libraries with *models* (i.e., approximations) of their effects on the symbolic state. To improve the efficiency of the symbolic executor, the implementation leverages lazy-initialization [108].

5.4 Evaluation of Propagation-based Technique

In this section, we evaluate our propagation-based testing technique of Section 5.2 and compare it with the existing coverage-based approaches described in Section 5.1. To that end, we present two empirical studies in this section. The first study, in Section 5.4.1, uses TRADSE (traditional symbolic execution) to compute propagation-based requirements. The second study, in Section 5.4.2, evaluates the improvements achieved by replacing TRADSE with SPD (Chapter 4) for computing propagation-based testing requirements. Finally, in Section 5.4.3, we discuss the threats to the validity of these studies.

5.4.1 Study using Traditional Symbolic Execution

The goals of this first study are to assess the ability for detecting output differences of using the propagation-based testing requirements produced by algorithm COMPUTEREQS (Section 5.2) and to compare the abilities of these requirements and the requirements defined by existing coverage-based approaches for testing changes.

5.4.1.1 Empirical Setup

For this study, we used the same two subjects and the same test cases and changes utilized for the evaluation of existing change-testing techniques in Section 5.1. These subjects, listed in Table 5, correspond to software of different nature and complexity. Tcas is representative of modules with straightforward logic that can be found in avionics systems

and other industrial domains. NanoXML, in contrast, represents more complex, object-oriented software that relies on Java libraries. The pools of test cases for these subjects can be considered as representative of the test cases that testers create in practice because they are large and they cover—often with redundancy—the space of program behaviors comprehensively.

We ran our toolset on an Intel Core Duo 2 GHz machine with 1 GB RAM. Because our technique focuses on changes and their effects at a certain distance, the scalability of the approach in this machine or any other environment is affected by the complexity of the subject rather than its size.

Similar to the study of existing techniques in Section 5.1, for each change in this study, we created 100 different test suites for each of our propagation-based strategies. We studied not only PROP, but also CHAIN to help us assess the extent to which each of the two components of PROP—CHAIN and the state requirements for each chain in CHAIN—contribute to the effectiveness of the whole technique. To measure the ability of each strategy to produce observable differences that reveal the effects of a change to developers, we computed the percentage of the 100 test suites for that strategy that produced a different output in the original and modified programs. We compared the difference-detection abilities of these strategies with the results of the existing coverage-based strategies presented in Section 5.1: STMT (changed statements), BR (affected branches), and DU (affected du-pairs).

For CHAIN and PROP, we used the greatest distance d that MATRIXRELOADED could reach for PROP within the memory constraints of the machine used and the path-length limit l, which was set to 10 branches per path. On the performance side, the tool always took less than eight minutes to successfully analyze a pair of program versions and generate requirements. Thus, for the distances within the reach of this tool in this study and this path-length limit, the time required to generate the testing requirements was not an issue.

5.4.1.2 Results and Analysis

The last two columns in Tables 6 and 7 of Section 5.1 present, for Tcas and NanoXML, respectively, the percentage of the test suites satisfying the CHAIN and PROP requirements for each change that revealed a difference in the output. These two columns, which correspond to our propagation-based testing requirements, are separated from the columns for coverage-based criteria by a double line. Aside from the results presented in these tables, the average for all changes in Tcas of the maximum size of the test suites satisfying the PROP requirements was 5.8 test cases, whereas the average of that maximum size for the changes in NanoXML was 2.4 test cases.

For all changes in Tcas, Table 6 shows that the effectiveness of existing coverage-based criteria is low—between 0.7% and 47.3%, and below 16% in all but one case. In contrast, the propagation-based strategy PROP always showed a higher detection rate—in most cases, by a considerable amount. Some of the improvements are due to the chain requirements (CHAIN) alone, such as for Change 4. In other cases, such as for Changes 1, 2, and 6, the improvements observed are due to the addition of state requirements to chain requirements (i.e., PROP). However, there are cases in which, despite the improvements, detection rates remain low (e.g., Changes 3 and 5) after applying the new technique. For those cases, we hypothesize that the distances reached are not sufficient to provide acceptable confidence of propagation of the infection to the output and that greater distances, which require enhanced algorithmic and implementation support, are necessary. The study of the effect of SPD on PROP in Section 5.4.2 and the alternative approach of Chapter 6 provide important insights on the benefits of greater distances.

For NanoXML, our technique shows an improvement in detection with respect to simpler criteria for three out of the seven "interesting" changes listed in Table 7: Changes 4, 8, and 9. BR and DU, which are subsumed by CHAIN, sufficed for Changes 1 and 3 to improve on simple change coverage (i.e., STMT) although for Change 3 the detection rate

was still low. Meanwhile, STMT showed the same effectiveness as the other strategies—coverage-based and propagation-based—for the remaining Changes 2 and 7. To understand this last result, we manually inspected the code and found that, in both cases, (1) the infected states can reach an output, (2) the dependence chains from the changes to the output are longer than the maximum distance MATRIXRELOADED could analyze, and (3) the existing test cases that satisfy the new requirements do not always propagate the infected part of the state to the output. In other words, PROP pushed the infected state up to the longest distance reached by the tool, often achieving propagation to the end, but could not guarantee that the output itself would be infected.

The improvements achieved by propagation-based testing for NanoXML, within the short distances achieved, were not as dramatic as in the case of Tcas. However, there are important differences between the two subjects that help explain these results. First, NanoXML makes extensive use of Java strings and containers, and we found that the infected state is often confined to objects of these types. Because MATRIXRELOADED replaces library objects with approximate models, the tool cannot distinguish within these libraries some of the details of the real propagation conditions for the changes. Second, NanoXML uses heap objects and polymorphism, unlike Tcas, therefore imposing a considerably higher burden on symbolic execution due to lazy initialization. These complications make it harder for the dependence analysis and symbolic execution to operate on NanoXML than on Tcas, as reflected by the shorter distances reached on NanoXML. Nevertheless, the results for NanoXML, even at short distances, are promising; they suggest that this new approach can exercise more change-related behaviors in real object-oriented software, compared to existing change-testing criteria. The results for Tcas are also promising because they indicate that this technique can be quite effective for detecting differences in similar modules within larger software.

5.4.2 Study using SPD

The goal of this second study is to assess the benefits in difference-detection effectiveness of using SPD (Chapter 4) as a more efficient technique than TRADSE for multiple-path symbolic execution in propagation-based testing strategies. Despite the greater effectiveness of propagation-based strategies over coverage-based strategies observed in our first study, the limited distances achieved in that study motivate the use of a more cost-effective (i.e., more effective for the same amount of time) approach such as SPD.

5.4.2.1 Empirical Setup

For this study, we used the tool JSPD that implements SPD, described in Section 4.3, to replace the traditional symbolic execution component of MATRIXRELOADED. Thus, JSPD computed the state requirements for each chain identified by MATRIXRELOADED.

The subjects of this second study are eight changes: four in Tcas and four in release *v1* of NanoXML. These are the first four changes available for each of these subjects in the SIR repository. For Tcas and NanoXML, there are 1608 and 214 test cases, respectively. We ran this experiment on the same machine and environment of our path-space reduction study of Section 4.4. For each technique, change, and propagation-path length, we set a limit of one hour for running JSPD. In almost all cases, JSPD either finished within 30 minutes or had to be stopped after reaching the one-hour limit.

In this second study, we measured the relative effectiveness of using two instances of SPD with respect to TRADSE for computing test-suite augmentation conditions for different changes and for different lengths of the paths from these changes. The two instances of SPD are SPD_{prec} (i.e., precise SPD, without abstractions) and SPD_{abs} (i.e., SPD with abstraction level $e = 4$; see Section 4.3). Specifically, we measured two aspects of effectiveness. The first aspect is the confidence that the resulting conditions represent good testing requirements for each change, which we measured by the number of change-propagating

Table 9: Propagation paths covered for change testing.

change	technique	control-flow propagation paths covered					
		10s	20s	1m	5m	15m	30m
Tcas 1	TRADSE	8	115	374	1184	1184	1184
	SPD_{prec}	1184	1184	1184	1184	1184	1184
Tcas 2	TRADSE	939	1482	2807	3192	3192	3192
	SPD_{prec}	3536	4568	4568	4568	4568	4568
Tcas 3	TRADSE	3	4	7	62	290	570
	SPD_{prec}	4914	13325	13325	13325	13325	13325
Tcas 4	TRADSE	42	70	217	2323	6722	8048
	SPD_{prec}	8048	8048	8048	8048	8048	8048
NanoXML 1	TRADSE	8	15	38	176	521	581
	SPD_{prec}	8	220	1268	7558	23282	24985
NanoXML 2	TRADSE	5	329	835	835	835	835
	SPD_{prec}	6	3800	25263	71903	71903	71903
NanoXML 3	TRADSE	5	177	835	835	835	835
	SPD_{prec}	5	694	5276	71903	71903	71903
NanoXML 4	TRADSE	2	24	171	182	182	182
	SPD_{prec}	2	88	670	1144	1144	1144

control-flow paths covered by the analysis.[4] The second aspect is the number of output differences revealed by test suites satisfying those computed conditions.

To minimize the impact of randomness in our results, for each symbolic-execution technique and propagation-path length, we generated up to 1000 unique test suites—as many as possible unique test-suites from the available test pool—satisfying the respective requirements. For each set of test suites, we computed the average of the number of test-cases revealing a different output. We generated the test suites randomly from the pool of available test cases for each subject and change.

[4]The more of the program is included (even if no new differences are discovered), the more confidence the tester has that the analysis covers all potential effects of changes.

5.4.2.2 Results and Analysis

For every change, path length, and technique, JSPD took a different amount of time to complete. Thus, to compare the effectiveness of the techniques for the same points in time (i.e., cost), we constructed for each technique a curve fitting the data points for that technique. Thus, the effectiveness values presented in this section correspond to points interpolated from these curves.

Table 9 shows the cost-effectiveness of using SPD for achieving *confidence* (i.e., coverage of propagation paths) along time with respect to TRADSE. Changes are identified in the first column by the subject name and a number between 1 and 4. For each change, there is one row for SPD_{prec} and one for TRADSE;[5] each technique is named in the second column. The remaining columns show the confidence for six points in time—10 and 20 seconds, and 1, 5, 15, and 30 minutes—interpolated from the available data points for each technique.[6] For example, at 1 minute on Change 2 in NanoXML, SPD_{prec} has covered the equivalent of 25263 control-flow paths, while TRADSE has covered only 835 paths. For some changes, the number of paths levels off after a certain point in time because data points beyond that time are not available—the analysis for the chain length either ran out of memory or did not finish within one hour.

For the first minute, SPD_{prec} is considerably more effective than TRADSE. This trend continues until the time limit is reached Changes 2 and 3 in Tcas and for all changes in NanoXML. For Changes 1 and 4 in Tcas, TRADSE catches SPD_{prec} eventually, which is explained by the limited number of propagation paths from these changes in this subject. However, for all changes, SPD_{prec} covers either more paths than TRADSE or the same number of paths in less time. In all, we can conclude that, at least for this set of changes, SPD_{prec} is more cost-effective at providing confidence than TRADSE by ensuring that more

[5] SPD_{abs} does not analyze paths with precision, so we omitted it in this table.

[6] We omitted time points beyond 30 minutes because, as mentioned in Section 5.4.2.1, for all changes, JSPD terminated either before 30 minutes or after 60 minutes—our time limit.

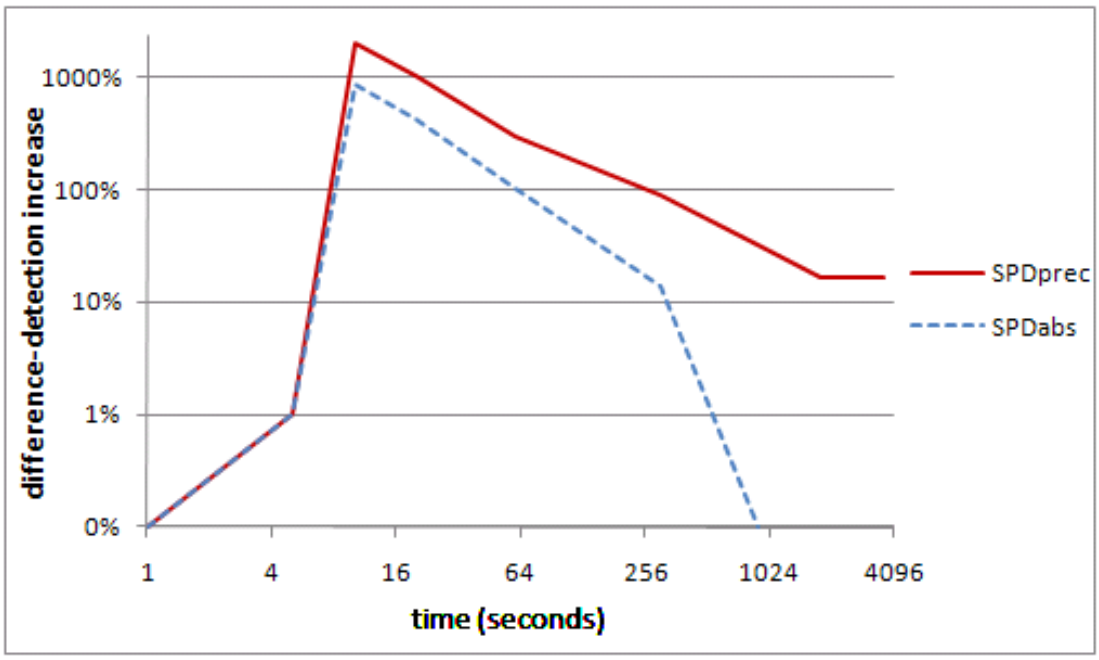


Figure 15: Average increase in difference-detection (in logarithmic scale) with respect to TRADSE versus analysis time, for all changes in Tcas.

paths that potentially propagate changes are analyzed in less time.

Figures 15 and 16 present a comparison of the average increase in *difference-detection* effectiveness (i.e., average number of differences detected by test suites satisfying the computed conditions) achieved by SPD_{prec} and SPD_{abs} with respect to TRADSE along time for the changes in Tcas and NanoXML, respectively. On each figure, the horizontal axis represents the analysis time in seconds (in logarithmic scale) and the vertical axis is the percentage of increase in effectiveness with respect to TRADSE. For Tcas (Figure 15), the vertical axis is also shown in logarithmic scale to facilitate visualization.

For Change 1 in Tcas and Change 3 in NanoXML (not shown individually in the graphs), all three techniques had the same effectiveness curves, suggesting that no useful additional conditions were found by SPD_{prec} or SPD_{abs} with respect to TRADSE—only more confidence was achieved. For all changes in each subject, the average increase in effectiveness observed in each figure was due to the remaining six changes studied.

For Tcas (Figure 15), both versions of SPD achieved a sharp increase in effectiveness

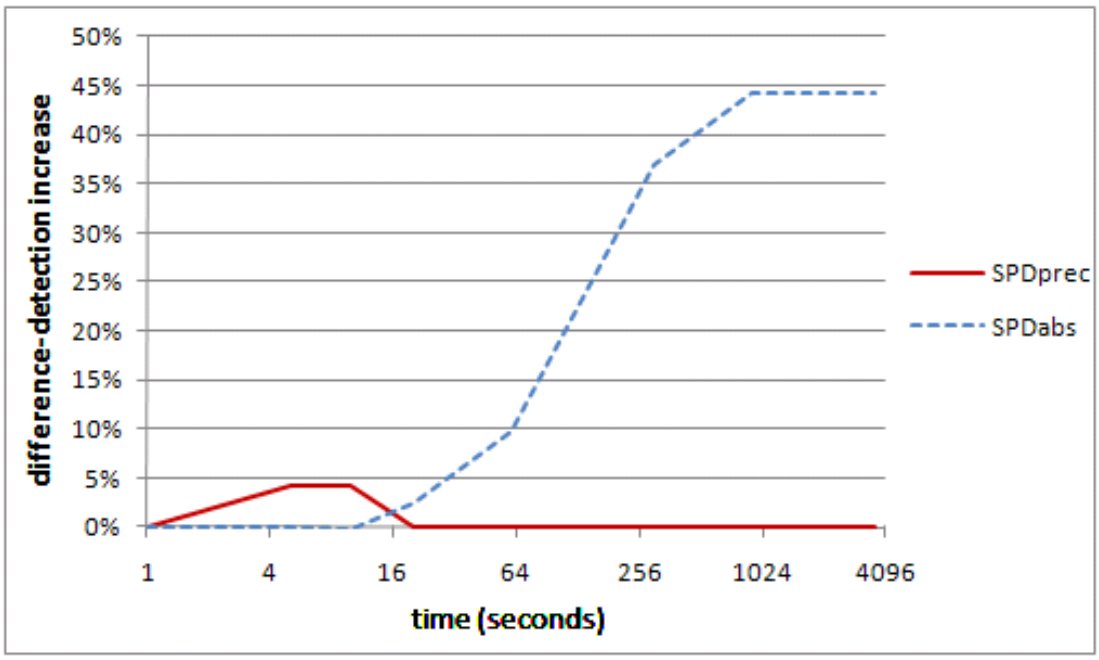


Figure 16: Average increase in difference-detection with respect to TRADSE versus analysis time, for all changes in NanoXML.

for the first 10 to 60 seconds, which indicates that TRADSE requires more effort to find the same conditions found by SPD for those times. In the long run, the advantage of SPD_{prec} over TRADSE in Tcas stabilized at 17%. For Change 3 in Tcas (the details are not shown), however, SPD_{abs} was less effective than the other techniques after 800 seconds, which explains the drop of the average for SPD_{abs} in Figure 15 at that point in time. Closely examining this change and the affected code in Tcas reveals that abstraction prevented SPD_{abs} from capturing important propagation conditions.

Meanwhile, for NanoXML (Figure 16), SPD_{abs} shows important gains in the long run. Considering the difficulty of finding propagation conditions beyond the vicinity of changes in the code—which is covered in the first few minutes by all of the techniques—this result is particularly encouraging.

In all, the results suggest that SPD without abstractions is the best choice for changes in straightforward code, such as that of Tcas, while SPD with abstractions—specifically, SPD_{abs} with $e = 4$—can capture important information beyond the distances reached by

precise analyses and obtain better results for object-oriented programs such as NanoXML.

5.4.3 Threats to Validity

The main internal threat to the validity of our two studies is the potential presence of implementation errors in the different components of MATRIXRELOADED, JSPD, and the underlying DUA-FORENSICS tool. Potential errors can also exist in our manually-created models of Java library methods. To minimize this threat, we tested and debugged our tools to the extent necessary to gain sufficient confidence in them for these studies. This effort included an extensive use of runtime checks and manual inspection of results.

The main external threat is the representativeness of our subjects—we evaluated two subjects and a limited number of changes for each of them. To reduce this threat, we chose two subjects of different coding styles and purposes, and changes on different parts of these subjects. Nevertheless, more subjects and changes need to be studied to fully assess the benefits of this technique using either TRADSE or SPD.

5.5 Related Work

An earlier approach that we developed for testing the effects of changes [6] is the work most closely related to the work in this chapter. Unlike that early work, however, we identified in the work of this chapter the multiple ways (dependence chains) in which a point in the program can be affected and we established strong links between our propagation-based technique and the foundations of change-effects analysis presented in Chapter 3. We also presented in this chapter, in contrast with that early work, a complete implementation of our propagation-based approach and more extensive studies, including the assessment of the benefits of applying the SPD technique of Chapter 4. These studies allowed us to empirically validate the qualities of our new technique for testing changes.

Several other techniques are related to the analysis of changes and test-suite augmentation presented in this chapter. These techniques can be classified into three categories. Techniques in the first category define general testing criteria for software in terms of

coverage of program entities, such as statements, branches, and definition-use pairs (e.g. [45, 67, 74]). These criteria can be adapted, as we did in our studies in this chapter, for modified software by considering only changed statements or affected data- or control-dependencies. Our empirical studies in this chapter showed that test suites satisfying our chain and state requirements have a greater likelihood of revealing different behavior than those that are based on data- or control-dependence only.

Techniques in the second category correspond to regression testing. The literature on regression testing is extensive, but most of it focuses on the efficiency of re-running an existing test suite. A few techniques, however, have been described (but not evaluated) for identifying and testing the effects of changes. For testing changes, Binkley [17], Rothermel and Harrold [87], and Gupta and colleagues [50] present techniques that use slicing [57, 112] to compute testing requirements in the form of individual data- and control-dependencies that are potentially affected by a change. Finding affected dependencies is one way of approximating the effects of a change in practice. However, as our studies in this chapter reveal, this approximation is insufficient for revealing the behavior of a change in comparison with our propagation-based approach of Section 5.2. The advantages of our approach can be attributed to the principled approximation of the effects of changes, as presented in Chapter 3, that includes all the ingredients (dependence chains and state differences) that define these effects and uses distance and path-length limits to make their computation practical.

Techniques in the third category correspond to change-impact analysis (e.g., [20, 21, 68, 76, 83]). Although the main goal of these techniques is to find which parts of the program might be affected by a change and require further inspection, these techniques, like the propagation-based approach in this chapter, compute approximations of the effects of changes to achieve their goal. These techniques, however, suffer the same limitations as the techniques in the second category because the elements they identify as affected constitute an incomplete view of how a change affects the behavior of the program. Furthermore,

most change-impact analysis techniques use coarse-grained entities such as methods to represent the potential effects of a change, making them even less precise.

CHAPTER VI

DEMAND-DRIVEN TESTING OF CHANGES

In this chapter, we analyze the inherent limits of computing propagation-based testing requirements as presented in Chapter 5 and we formulate an alternative, demand-driven approach for computing and monitoring these requirements. Section 6.1 discusses those limitations and Section 6.2 presents the demand-driven alternative that addresses those limitations to reach much greater distances and discusses the trade-offs of using this alternative. Then, Section 6.3 presents our implementation that we use for a comprehensive study of the new demand-driven approach in Section 6.4 and for case studies of the practicality of this approach in Section 6.5. Finally, Section 6.6 discusses related work.

6.1 Limitations of Propagation-based Testing Requirements

The distance limit d for the propagation-based strategy of Chapter 5 for testing changes addresses two problems: (1) the high cost of computing the testing requirements and (2) the need to present a manageable number of testing requirements to testers. Although the *standard* formulation[1] of the propagation-based requirements has its benefits—it obtains the set of all testing requirements within distance d in a way that can be used to automate the generation of satisfying inputs—this formulation still has two shortcomings:

1. Symbolic execution does not scale well, despite advances for multiple paths [94] (Chapter 4), compositionality [3], and other optimizations (e.g., [78]). Symbolic execution is generally intractable due to the path-explosion problem and, thus, the distances reachable by a propagation-based technique are often insufficient. Moreover, within the distance limits achieved in practice (Section 5.4), for many chains,

[1] We call *standard* the technique of Chapter 5 to distinguish it from the demand-driven alternative.

some paths that cover these chains cannot be fully analyzed because of their length. For such paths, the results are an approximation of the actual propagation conditions.

2. Often, a large fraction of the testing requirements identified by the static analysis of P and P' are either infeasible or too difficult to satisfy within a reasonable budget. Thus, many requirements do not get satisfied in practice, and the effort spent in computing them is wasted.

6.2 Demand-driven Propagation-based Testing Strategies

To circumvent the limitations listed in Section 6.1, we present a new approach for testing changes using CHAIN and PROP. Instead of computing all testing requirements beforehand (i.e., statically), the new approach monitors dependence-coverage and state differences during execution, identifying the satisfaction of testing requirements and extending each requirement—chain and state conditions—on demand only when necessary. This new approach preserves the precision of propagation-based testing requirements while avoiding the cost of symbolic execution and avoiding most of the effort incurred by performing a full static analysis to find requirements that, later, are not satisfied.

Naturally, using this approach comes at the expense of having the full set of testing requirements computed a priori. When all requirements are readily available, testers get a view of all untested affected behaviors, which can support more directly the (semi)automatic creation or modification of new test cases for unsatisfied requirements. The new approach, in contrast, provides the set of covered chains and state conditions along with the points in those chains from which areas of the program with unsatisfied requirements can be reached. However, the benefits of this new approach can very well justify this trade-off.

The new approach requires only static and dynamic dependence analysis and runtime state monitoring, in contrast with the technique of Section 5.2, which requires symbolic execution of all paths that can propagate a change and the enumeration of all dependence chains within a distance limit. For that reason, the new approach can significantly increase

the distances reachable for CHAIN and PROP, as the study in Section 6.4 shows. Also, this approach removes the burden from the tester of handling a large number of short-distance requirements and, instead, lets the tester focus on long-distance requirements that are actually satisfied.

The new approach for propagation-based test-suite augmentation is described by algorithm SATISFYREQSDEMAND in Figure 17. The inputs are the program P, the change C, a boolean function B that indicates whether the budget allows for more testing, and the existing test suite *TS* to be augmented. The first line creates the modified program P' by applying change C to P. Lines 2 and 3 find all control and data dependencies reachable via any sequence of dependencies that starts at some node in the sub-ICFGs of C (see definitions 13 and 14). Lines 4 and 5 instrument these two programs to determine at runtime the chains of these dependencies covered from the changed nodes and the portion of the program state that directly *affects* at runtime each of those dependencies. For a control dependence, the affecting state is simply the branching decision taken at the source node of the dependence. For a data dependence, the affecting state is the concrete value assigned to the variable at the definition.

The algorithm does not enumerate all chains statically. Instead, the algorithm instruments each dependence once such that the coverage of a dependence at runtime is reported only when the dependence starts at a changed node or when the execution of its source node is the target at runtime of another dependence dynamically linked to a changed node. The result of executing each of the instrumented programs is an acyclic directed graph where the root nodes (i.e., the nodes without predecessors) are occurrences of changed program statements, the remaining nodes represent occurrences of dependencies, and each edge (d_1, d_2) indicates that a changed statement or dependence d_1 was immediately followed by a dependence d_2. Thus, the covered dependence chains are the paths in this graph. Also, each node in this graph except for the root nodes is annotated with the affecting state for the dependence occurrence associated to that node.

Algorithm SATISFYREQSDEMAND

Inputs: P: program; C: change; B: budget function; *TS*: test suite
Output: *TS*: test suite; *Satisfied*: testing requirements

```
(1)   P' := applyChange (P, C)
(2)   deps := forwardDepChains (P, C)
(3)   deps' := forwardDepChains (P', C)
(4)   P_ins := instrument (P, deps)
(5)   P'_ins := instrument (P', deps')
(6)   Satisfied := createEmptyChains (P_ins, C) ∪ createEmptyChains (P'_ins, C)
(7)   totalEffort := 0
(8)   while B(totalEffort, Satisfied, TS) = true
(9)     (t, effortTest) := getNextTest (TS, P_ins, P'_ins, C, Satisfied)
(10)    (data, effortEx) := execute (P_ins, t)
(11)    (data', effortEx') := execute (P'_ins, t)
(12)    totalEffort := totalEffort + effortTest + effortEx + effortEx'
(13)    addTest := false
(14)    foreach prefixChain ∈ Satisfied
(15)      foreach dynDep ∈ extendingDeps(prefixChain, data, data')
(16)        chain := append (prefixChain, dynDep)
(17)        if chain ∉ Satisfied
(18)          Satisfied := Satisfied ∪ chain
(19)          addTest := true
(20)        endif
(21)        if propagates(chain, data, data')
(22)          markProp (chain, Satisfied)
(23)          addTest := true
(24)        endif
(25)      endfor
(26)    endfor
(27)    TS := TS ∪ (addTest? {t} : ∅)
(28)  endwhile
(29)  return TS, Satisfied
```

Figure 17: Algorithm for demand-driven, propagation-based testing of a change.

Line 6 in SATISFYREQSDEMAND initializes the set *Satisfied* of satisfied testing requirements to "empty" chains at the modified nodes in the instrumented programs—one chain $<n, ()>$ for each modified node n. Line 7 initializes the counter for the total testing effort spent in this algorithm.

The main loop of lines 8–28 computes the testing requirements satisfied by the original

test suite and by each new test case provided to the algorithm.[2] The loop only stops when function B signals that the budget has been exhausted by returning false, depending on the effort spent in testing so far, the testing requirements satisfied, or the size of the test suite. Line 9 obtains the next test case t and the effort involved in obtaining t. In the first $|TS|$ iterations of the loop, *getNextTest* returns each of the test cases in *TS*. In the remaining iterations, *getNextTest* obtains a new test case from the user or a test-case generator. Each new test case t is created using the knowledge of the programs, the change, and the requirements satisfied so far. The concrete mechanism that creates new test cases in *getNextTest* after the elements of *TS* have been processed is determined by the user—each new test case can be created manually, automatically, or semi-automatically.

Lines 10 and 11 execute the test case on the instrumented versions of the original and modified programs, respectively, and returns the runtime data—the annotated acyclic graphs of dependencies linked to the change—collected by the instrumentation and the effort spent executing the test case and collecting this data. Line 12 adds these runtime efforts and the test-case creation effort to the effort counter.

The rest of the main loop determines what new testing requirements (CHAIN and PROP) are satisfied by t and whether t is added to the test suite. Line 13 initializes a flag to false to assume at first that t should not be added to *TS*. Then, the loop at lines 14–26 and its nested loop at lines 15–25 identify, for every chain in *Satisfied* (called "prefix" chain), all dependencies in t that extend that prefix chain.[3] For each such dependence, Line 16 creates the extended chain. Lines 17–20 add the extended chain to the set *Satisfied* if not already in that set (thus satisfying a new CHAIN requirement) and lines 21–24 mark that chain in *Satisfied* as *propagated*[4] if the differences in runtime data indicates so (thus satisfying a

[2]Test cases at this point need only include the scaffolding and input data to run the program. The creation of oracles for these test cases can be deferred until the test cases are accepted for addition to the test suite.

[3]A dependence d extends a chain c if the source node of d is the end node of c and d is covered immediately after the chain is covered.

[4]A chain c covered in an execution is marked as *propagated* if, for each dependence d in the chain, the affecting program state at the source node of d differs between *data* and *data'*.

new PROP requirement). In either case, the flag for adding t to the test suite is set because t satisfies at least one new chain or state requirement.

Finally, Line 27 adds t to the test suite *TS* if the flag is set, thus augmenting *TS*. Note that, for the first |*TS*| test cases, *TS* does not change because each t is already in *TS*. The purpose of the first |*TS*| iterations of the main loop is simply to populate *Satisfied* with all testing requirements satisfied by the existing test suite before new test cases are created and to determine whether those new test cases must be added to *TS*.

6.3 Implementation of Demand-driven Strategies

To support the evaluation of coverage-based and propagation-based strategies in a demand-driven fashion, we reused the DUA-FORENSICS system for dependence analysis and monitoring [93], which is based on the Soot Analysis Framework [91] and analyzes Java-bytecode programs. Strategies STMT, BR, and DU (defined in Section 5.1) are already implemented in DUA-FORENSICS. However, because there is no longer a unique distance limit in the demand-driven approach, we removed the restrictions to the distances from the change at which branches and du-pairs are reported covered (i.e., we used the full versions of BR and DU). Affected branches and du-pairs are reported as covered only if they are executed after the change has executed.

For the demand-driven CHAIN and PROP strategies described in Section 6.2, we extended the dynamic dependence analyzer provided by DUA-FORENSICS to identify not only which entities (i.e., statements and dependencies) are covered after a change but also which dependencies precede which other dependencies. This ability to track sequences of dependencies lets DUA-FORENSICS precisely monitor the dependence chains within each dynamic slice. In addition, for the PROP strategy, we extended DUA-FORENSICS to identify state modifications performed by each statement in a dependence chain.

6.4 Comprehensive Study of Demand-driven Strategies

The goal of this section is to provide a comprehensive study of the demand-driven approach for propagation-based testing of changes. More concretely, the goals of this study are (1) to evaluate in detail the cost-effectiveness of the demand-driven approach for detecting differences, which is made possible by the characteristics of this approach, (2) to use this approach to compare propagation-based strategies and coverage-based strategies to a greater extent than it was possible with the approach of Section 5.2, and (3) to determine in which situations the strategies work best. This study uses pools of existing test cases to represent the test inputs that testers would create during testing.

Section 6.4.1 describes the empirical setup for the study. Then, Section 6.4.2 presents and analyzes the results of this study. Finally, Section 6.4.3 addresses threats to the validity of our study and discusses mitigating factors for those threats.

6.4.1 Empirical Setup

To perform this study, we ran our tool (Section 6.3) on a machine with two quad-core Intel Core i7 CPUs and 12 GB RAM running Red Hat 6 Linux and the Oracle (formerly Sun) Hotspot 6 Java Virtual Machine.

Next, we present the subjects used in this study, our definitions of metrics of effectiveness, the concrete research questions for our study goals, and the method used to address those questions.

6.4.1.1 Subjects

For our study, we considered Java subjects for which a large number of test cases are available. Therefore, we selected a number of subjects from the Siemens suite [60] that we translated from C to Java.[5] These subjects are listed in Table 10 where the columns show, respectively, the name of the subject, a short description, the size in lines of code (LOC), the

[5]From the Siemens suite, we omitted Tcas to give priority to more complex subjects in this suite that can be analyzed efficiently using the demand-driven approach.

Table 10: Subjects, test cases, and changes for study of demand-driven testing of changes.

subject	description	LOC	test cases	changes
Tot_info	information measure	283	1052	8
Schedule1	priority scheduler	290	2650	8
Schedule2	priority scheduler	317	2710	8
Print_tokens	lexical analyzer	478	4130	9
Print_tokens2	lexical analyzer	410	4115	10
NanoXML v1	XML parser	3497	214	7
NanoXML v2	XML parser	4009	214	6
NanoXML v3	XML parser	4608	216	7
			total changes:	**62**

number of test cases available, and the number of changes studied. The pools of test cases provided with these subjects are very large and comprehensive enough to consider each test case selected from that pool as representative of the test cases that developers would create in practice. The Siemens subjects can be seen as representative of small programs as well as modules within larger programs.

We also studied three releases of NanoXML, an XML parser used in many applications and whose coding style and complexity make it representative of object-oriented programs. We obtained these releases of NanoXML from the SIR repository [36].

The changes listed in the last column of Table 10 were introduced by other researchers for their own studies. These changes were provided along with the test cases and subjects by the Siemens researchers [60] (the first five subjects) and by the maintainers of the SIR repository [36] (for NanoXML). In total, we studied 62 changes in eight subjects.

6.4.1.2 Metrics

In this study, we measured of the effectiveness of a test suite by its *difference-detection ability*—the number of test cases whose outputs in the original and modified versions of the program differ. The intuition behind this metric is that the more differences caused by the change the developer observes, the more information the developer has to either detect an error or gain confidence that the change is correct. Multiple differences are likely the

result of different aspects of the change because each test case satisfies a different set of requirements—requirements are designed to exercise different effects of the change.

Our measure of cost-effectiveness, or quality, of the test suite was the ratio of its difference-detection to the size of the test suite, which we call *ds-ratio*. The greater the ds-ratio, the better the strategy that created the test suite is at sampling and discriminating test cases that can cause differences. The maximum possible value of a ds-ratio is 1.0, which corresponds to the case in which all test cases in the test suite detect a difference.

To support our analysis of the factors affecting the effectiveness of each strategy, we defined the *detectability* of a change as the ds-ratio for STMT on that change. The detectability of a change is thus the estimated probability that a test case that covers the change also reveals a difference in the output. The lower the detectability of a change, the less likely it is that a test case reveals a difference for that change and, thus, the greater is the difficulty of testing that change successfully.

Also, to enable the classification of changes in our study according to the difficulty of their testing, we defined the *detectability limit* of a set of changes as the maximum detectability of all changes in the set.

6.4.1.3 Research Questions

To achieve the goals of our study, we identified four research questions:

> *RQ1*: How cost-effective are propagation-based strategies with respect to existing techniques for testing changes?
>
> *RQ2*: How much statistical confidence is there in the cost-effectiveness measured for each strategy?
>
> *RQ3*: How do these strategies perform on changes that are *difficult* to test? (i.e., changes for which a good strategy is most needed)
>
> *RQ4*: What are the computational costs of using propagation-based strategies?

6.4.1.4 Method

For this study, we designated STMT as our baseline strategy because we consider any strategy that is less effective than randomly exercising a change (i.e., creating tests that just cover the change without any other goal) to be useless. To evaluate each of the strategies BR, DU, CHAIN, and PROP, for each change in each subject, we used the following method:

1. For each test case in the pool available for that subject, perform on the original and modified programs (P and P') the analysis and monitoring of the testing requirements satisfied by that test case.

2. Using this information, construct 1000 different test suites (or as many such test suites as exist) by selecting from the pool at random, without replacement, one test case t at a time until all requirements that the pool can satisfy are satisfied. For CHAIN and PROP, use the demand-driven approach of Section 6.2 with a budget function B that terminates when all test cases in the pool have been executed.

3. Compute and record the average *difference-detection* (i.e., number of test cases that reveal an output difference), average size, and average ds-ratio of all test suites. These are the three kinds of data used in our empirical analysis.

For simplicity and accuracy, in the case of STMT, instead of performing this three-step process, we analytically computed the ds-ratio for each change by dividing the number of test cases in the pool that produce a different output between P and P' by the number of test cases in that same pool that cover the change. This value represents the minimum ds-ratio that any useful strategy should achieve for testing a change. Thus, the real benefit of using a strategy on a change corresponds to the *increase* in ds-ratio achieved by that strategy with respect to the ds-ratio for randomly testing that change.

By focusing our comparison of different strategies on the ds-ratios they achieve (or the differences in ds-ratios with respect to STMT) for each change, we can assess the relative

ability of the compared strategies for finding each new difference. Because, in practice, not all testing requirements can be satisfied within a reasonable budget (especially for changes), the key question for the testers is: "*Given a testing-effort budget B, how many differences can we detect within this budget by using a given strategy?*". The answer is the product of the ds-ratio for the strategy and the number of test cases that can be added to the test suite within budget B using that strategy.

6.4.2 Results and Analysis

For CHAIN and PROP, we experimented with all distances (values of d) that our implementation was capable of analyzing for each change, which in some cases reached dozens of dependencies from the change. However, to compute overall results per distance, we used as maximum value for d the shortest distance reached for all studied changes, which in our experiments was 10. Considering that the number of dependence chains can grow exponentially with the distance, this is a significant improvement over the original formulation of propagation-based strategies, which could reach distances of only about four dependencies in similar subjects [6, 92, 94].

In the rest of this section, we address our research questions.

6.4.2.1 RQ1: Cost-effectiveness of Strategies

Table 11 presents the average results for all strategies. In this table, the header row for each column except the first denotes the strategy, including all distances for CHAIN and PROP (sub-columns *d1* to *d10*, respectively) as well as a sub-column for the average results for all 10 distances for each of these strategies (sub-columns *avg*). The table shows three results for each strategy—one result per row—which are (1) the average number of differences for all test suites created for all changes using that strategy, (2) the average size of the test suites for all changes in that strategy, and (3) the average of the ds-ratios (differences-to-size ratios) for the test suites for all changes in that strategy. For example, for strategy DU, for all changes and all test suites created for each change using this strategy, the average

Table 11: Average number of differences, test-suite sizes, and ds-ratios per testing strategy for all changes.

metric	STMT	BR	DU	CHAIN										
				d1	d2	d3	d4	d5	d6	d7	d8	d9	d10	avg
diff.	0.32	1.67	4.57	0.45	0.51	0.71	1.04	1.53	2.24	3.20	4.53	6.34	8.41	2.90
size	1.00	7.90	40.1	1.26	2.05	3.06	4.88	10.65	17.82	24.24	34.04	39.56	51.21	18.88
ds-ratio	0.32	0.32	0.35	0.38	0.38	0.40	0.40	0.40	0.41	0.41	0.40	0.40	0.40	0.40

metric	PROP										
	d1	d2	d3	d4	d5	d6	d7	d8	d9	d10	avg
diff.	0.89	1.33	2.20	3.15	4.38	5.81	7.34	9.32	11.84	14.93	6.12
size	2.15	3.54	5.59	8.44	15.19	23.50	31.24	42.53	49.61	63.11	24.49
ds-ratio	0.48	0.50	0.52	0.53	0.52	0.52	0.50	0.50	0.49	0.48	0.50

number of differences detected in the output per test suite was 4.57, the average test-suite size was 40.1 test cases, and the average ds-ratio per test suite was 0.35. Note that, because 0.35 is the result of averaging the ds-ratios, it is not the same as the result of dividing the average number of differences per test suite by the average size of the test suites—the average of the ratios is not the same as the ratio of the averages. We regarded the ds-ratio as a unique quality of each test suite, and we reported its average value separately because this ratio represents the cost-effectiveness of a strategy.

The results in Table 11 for column STMT reflect that, indeed, every change in our study required only one test case to cover it (i.e., any test reaching a changed statement would cover all changed statements). This result is a direct consequence of the relatively small size of each change—a few lines of code, at most. The ds-ratios for STMT, which correspond to the detectability of the changes, ranged from 0.001 to 1.0, with an average of 0.32. In other words, for every 100 test cases, on average for all changes, 32 of them cause an output difference. Our results per individual change, however, indicate that there is a large diversity in this ratio. For instance, the standard deviation for detectability is 0.39. In fact, Table 13 (described later for RQ3) reveals that 14 changes have a detectability of 0.8 or more whereas nine changes have a detectability of 0.1 or less. Therefore, any interpretation

of these results must consider the large diversity in the behavior of these changes.

For BR, somewhat surprisingly, the average ds-ratio was indistinguishable from that for STMT, whereas for DU this ratio was only barely greater than for STMT and BR. Despite results in the literature for the Siemens subjects showing that the whole-program counterparts of BR and, especially, DU, perform better than random testing [60], these two strategies were barely useful in our study for the parts of the program affected by each change. This result can be explained by the relatively low coverage attained for BR and DU in these subjects, which ranged between 50–90% and was lower than for whole-program testing (as reported in Reference [60]), which often reached 100% for the same subjects. These lower coverage levels for BR and DU were expected because the difficulty of covering all affected branches and du-pairs after executing the change is usually greater than simply covering those branches and du-pairs without constraints.

It is worth noting, however, that for whole-program testing in the Siemens subjects, the effectiveness of branch and du-pair coverage is unequivocally better than random testing only at coverage levels of 93–100% [60], which are higher than the coverage levels we achieved for BR and DU using the same test pools. Thus, it remains to be seen whether a spike in cost-effectiveness can be observed when reaching a coverage near 100% for BR and DU (and, for that matter, for our propagation-based strategies). Achieving such a coverage, however, appears unrealistic because of the added constraint of having to execute the change before each testing requirement. Because of that constraint, we also expect that a larger proportion of the testing requirements is infeasible for change testing than for whole-program testing.

Another interesting result is the much greater number (on average) of differences detected and test-suite sizes for DU than for BR and STMT. These numbers indicate that DU is much more expensive to satisfy but also suggest that DU provides a greater effectiveness in total number of differences found. This effectiveness, however, is achieved at only a slightly greater ds-ratio (a measure of cost-effectiveness) than for BR and STMT. Thus, for

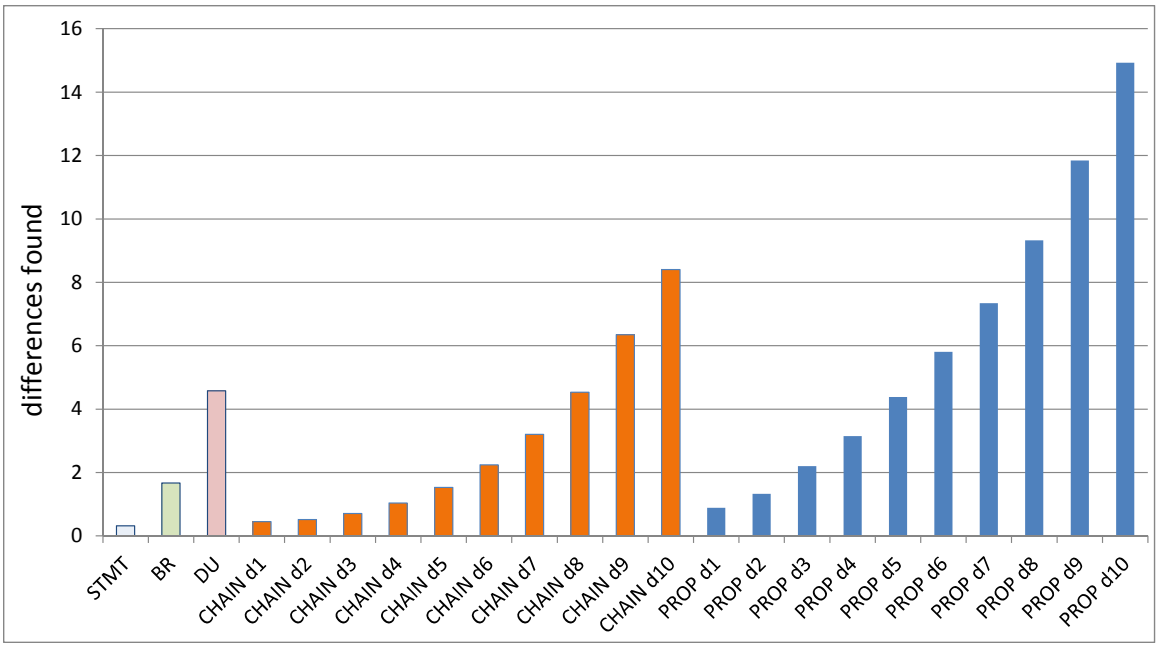


Figure 18: Average differences found by each strategy.

a testing budget that contemplates the creation of only a few test cases, DU is not a much better alternative than BR and STMT.

Table 11 shows, for our propagation-based strategies CHAIN and PROP at every distance and also on average for all distances, that the ds-ratios were better than for STMT, BR, and DU—much better in the case of PROP. CHAIN consistently achieved a ds-ratio of around .40 while finding an increasing number of differences as the distance increased. At distance 8, CHAIN already found almost as many differences on average as DU while requiring fewer tests than DU. At distances 9 and 10, CHAIN found more differences than DU. PROP, however, performed best of all strategies for all distances with a ds-ratio between .48–.53. The results also show that, on average, PROP detected more differences than DU starting at distance 6. Furthermore, for that same distance, PROP detects more than twice the number of differences than CHAIN detects. These results highlight the importance of our new approach for reaching longer distances in propagation-based strategies and exhibiting a greater effectiveness than coverage-based approaches.

Figures 18 and 19 provide a view, on average for all changes, of the effectiveness of

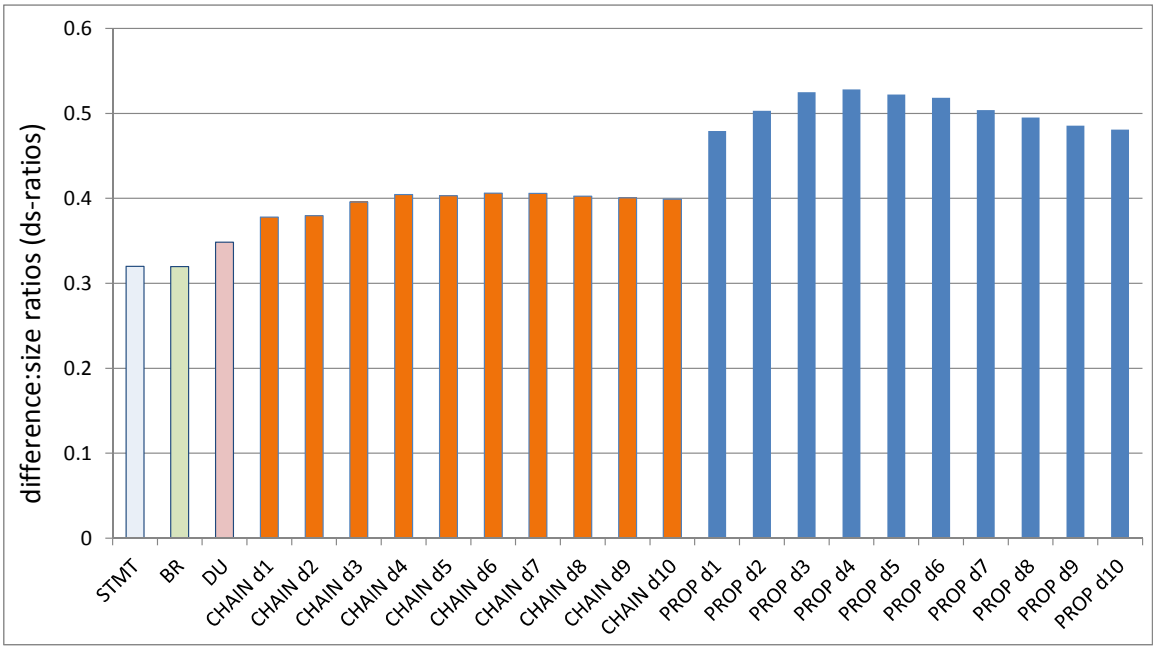


Figure 19: Average ratio of differences found to test-suite size (ds-ratio) per strategy.

our propagation-based strategies per distance and relative to the test-suite size (i.e., the ds-ratio), respectively, with respect to coverage-based strategies. The bar graph in Figure 18 shows the average number of differences found (vertical axis) for each strategy (horizontal axis) including all distances for CHAIN and PROP. The number of differences represents the effectiveness of a strategy, regardless of its cost in number of test cases. This figure provides a visual description of the magnitude of the differences in effectiveness among all the strategies that we studied. In particular, the graph highlights that BR is more effective (i.e., finds more differences) than STMT, DU is much more effective than BR and STMT—finding more than four differences on average—and that the propagation-based strategies, CHAIN and PROP, become much more effective as the distance increases and this effectiveness largely surpasses that of coverage-based strategies for distances greater than or equal to 9 and 6 for CHAIN and PROP, respectively.

The bar graph in Figure 19 provides a visual comparison of the *cost-effectiveness* (i.e. the effectiveness per each unit of cost, which is one test case) (vertical axis) of the strategies

studied (horizontal axis) as the average ds-ratio for all 62 changes. This graph shows not only that BR is not more cost-effective than STMT, but also that the improvement of DU over STMT is quite small. In contrast, CHAIN has a consistently greater cost-effectiveness on average than those other strategies, although the difference is not extraordinary, and that PROP is considerably better than all other strategies, including CHAIN, for all distances. An interesting aspect observable in this graph is that the cost-effectiveness of PROP spikes slightly at distance 4 before falling back at distances 9 and 10 to the levels achieved in the first two distances. This spike in cost-effectiveness, although not considerably higher than the average for all distances, suggests that the best distribution of the test cases created for any studied distance is not necessarily achieved at large distances such as 10. After all, despite finding more differences when the distance increases, it is not surprising that the ds-ratios for PROP and CHAIN stay relatively constant. The reason for this constancy in the ds-ratios is that the design goal of propagation-based strategies is to distribute well the test cases over the behavior space of the change for each distance and that this distribution should find a proportionally similar number of differences at different distances. We expect that the ds-ratios begin to approach 1.0 only when the distance limit gets close enough to the output statements, which was not the case for most changes in our studies. In general, however, it is unrealistic to expect that testers will achieve distances long enough to reach the output statements because those distances would be very large (much larger than 10 dependencies). As our results show, at least up to distance 10, the number of test cases needed grows considerably with the distance. Therefore, to reach the output, testers would have to create hundreds, if not thousands, of test cases for each change to guarantee a ds-ratio of 1.0.

In summary, for RQ1, we can conclude that, based on the changes and subjects used in this study:

- PROP produces, overall, considerably higher-quality test suites than the other strategies. In other words, PROP discriminates and distributes test cases over the behavior

Table 12: Statistical confidence of ds-ratio superiority of strategies versus each other.

strategy	CHAIN											PROP										
	d1	**d2**	**d3**	**d4**	**d5**	**d6**	**d7**	**d8**	**d9**	**d10**	**avg**	**d1**	**d2**	**d3**	**d4**	**d5**	**d6**	**d7**	**d8**	**d9**	**d10**	**avg**
vs STMT	×	×	†	†	†	‡	‡	†	†	†	†	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
vs BR	×	×	×	×	×	×	×	×	×	×	×	‡	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
vs DU	×	×	×	×	×	×	×	×	×	×	×	‡	✓	✓	✓	✓	✓	✓	‡	‡	‡	✓
vs CHAIN (average of all distances):												‡	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

space of changes better than the alternatives. PROP also finds more differences than DU and the other strategies at distances 6 and greater.

- CHAIN provides, overall, a less cost-effective but potentially simpler alternative of greater quality than coverage-based strategies.

- The longer distances achieved for propagation-based strategies using our new approach allow this kind of strategies to be more effective, in number of differences detected, than DU or BR. The flexibility that the tester has in choosing a distance, and even increasing it later, also give the tester more opportunities to achieve the desired results as well as operate within a testing budget.

- DU and BR are virtually indistinguishable from STMT in the average quality (ds-ratio) of the test suites it produces for the coverage levels achieved. Although DU finds more differences—it is more comprehensive in its testing requirements than BR—this benefit comes at the cost of a higher number of test cases with only a marginally better cost-effectiveness ratio.

6.4.2.2 RQ2: Statistical Analysis of Strategies

When addressing RQ1, we analyzed the average cost-effectiveness per strategy for all 62 changes. However, it is also necessary to quantify how reliable these results are so we can properly assess the potential validity of these results for other changes and subjects (of similar characteristics, at least).

Table 12 shows the confidence levels derived from paired t-tests [110] applied to the 62 changes in our study. A *paired t-test* is a statistical hypothesis test that compares means on two data sets, assumed to be independent and identically normally distributed, and determines whether they differ from each other in a significant way. We used these statistical tests to estimate the statistical significance of the improvements in ds-ratios of propagation-based strategies (columns) with respect to all other strategies (rows). To assess the level of significance for each cell in the table, we distinguished four cases for the level of confidence—a probability as a percentage—that the ds-ratios for the strategy in the column are greater than the ds-ratios for the strategy in the corresponding row. The four cases are:

1. Cross (×): the confidence is less than 95%, which is generally considered insufficient
2. Dagger (†): the confidence is between 95% and 99% (sufficient)
3. Double dagger (‡): the confidence is between 99% and 99.9% (strong)
4. Check mark (✓): the confidence is greater than 99.9% (very strong)

In Table 11 we already saw that BR is not better than STMT. DU, however, is slightly superior to BR and STMT, but this difference is far from being statistically significant according to our data. Therefore, we omit DU, BR, or STMT from the columns of Table 12 to simplify its presentation. For the first propagation-based strategy, CHAIN, Table 12 indicates that, on average for all 62 changes, this strategy has sufficient statistical superiority only over STMT and only for distance 3 or greater. For two distances—6 and 7—the confidence surpasses 99%. These results show that CHAIN can be considered a *useful* strategy (i.e., better than randomly testing a change) but do not demonstrate conclusively that this it is better than coverage-based strategies (BR and DU). Despite the lack of statistical significance for the improvement of CHAIN over BR and DU, the large number of changes that we used in this study suggests that CHAIN is indeed better than BR and DU. We expect

that an even larger study will confirm the superiority of CHAIN in a statistically significant way. Confirming the benefits of choosing CHAIN would be a positive development because CHAIN might be simpler to implement at a commercial scale than PROP.

Table 12 shows unequivocally that the observed superiority of PROP over coverage-based strategies and even CHAIN (the average for all distances for CHAIN) is statistical significant with a very high level of confidence—more than 99.9% for all distances of PROP except for distance 1. The significance of these results for PROP not only confirms what we observed in previous work [92,94] but also lets us quantify that superiority. On average for all distances, as Table 11 shows, PROP selects tests that have a 50% probability of revealing a difference, on average for changes like those we studied here. This probability is a clear improvement over random change-testing, coverage-based strategies, and even CHAIN. Therefore, from these results, we can be confident that PROP is a superior testing strategy for changes and that the strategy CHAIN that PROP subsumes is not enough. In other words, state-difference information, which has not been considered by researchers other than coarsely in one previous work [62], is a crucial component for causing the effects of changes to propagate and become observable.

In summary, for RQ2, we can conclude that, based on the changes and subjects used in this study:

- PROP has a strong, statistically significant superiority over all other strategies, including CHAIN, on the average quality (ds-ratio) of the test suites that it can create. The observed strength for all distances in PROP leads us to conclude that the state-difference requirements that distinguish PROP from CHAIN have a significant, positive impact that makes PROP the best testing strategy for changes that is available in the literature.

- CHAIN is a useful alternative to PROP for testing changes because, with sufficient statistical confidence, the average ds-ratio it achieves is better than for STMT. On

Table 13: Classification of changes by detectability limit.

detectability limit	**1.0**	**.8**	**.6**	**.4**	**.2**	**.1**	**.05**	**.03**	**.01**	**.005**	**.003**	**.001**
number of changes	62	48	46	43	39	30	22	18	9	7	4	1

average, CHAIN also performs better than DU and BR, although without a sufficiently strong confidence. Larger studies might be required to conclusively confirm this superiority as well.

- Although DU is slightly better than STMT and BR on average, there is no sufficient statistical evidence that this is in fact the case. Thus, we are not able to conclude that coverage-based strategies for testing changes are better than simply testing changes randomly. This result implies that, before our introduction of propagation-based strategies in the literature, the existing techniques for creating new tests for changes—a task also known as *test-suite augmentation* [92]—were not really useful.

6.4.2.3 RQ3: Effectiveness vs. Testing Difficulty

For RQ1 and RQ2, we discussed the average results of the strategies for all changes and analyzed them statistically, without distinguishing the difficulty of detecting differences (i.e., the detectability) for individual changes. However, the changes for which this difficulty is the greatest are those for which testers need a good testing strategy the most. Thus, for RQ3, we investigated how the detectability of each change influences the cost-effectiveness of all strategies. Table 13 shows the number of changes for different *detectability limits* (defined earlier in this section) in decreasing order of limits. For example, all 62 changes have a detectability limit 1.0 (i.e., detectability of at most 1.0, which is the maximum possible value), whereas 39 changes have a limit of 0.2 (i.e., detectability of at most 0.2).

The values in Table 13 reflect the diversity in the behaviors that changes in software can have. Although all the changes in our study correspond to fixes to observable failures and therefore produce a difference in at least one test (i.e., they do not represent refactoring

changes [44]), this diversity of values show that many changes that can be detected in the output are detectable only under specific conditions instead of every time they are executed.

The graph in Figure 20 illustrates how much more cost-effective PROP, CHAIN, BR, and DU are over STMT for the set of changes within each detectability limit. (For PROP and CHAIN, the graph presents average results for of all distances.) In this graph, the horizontal axis shows detectability limits in decreasing order and the vertical axis is the improvement in ds-ratio (cost-effectiveness) of a strategy over STMT—how much more cost-effective is a strategy than STMT. For detectability limit 1.0, which includes all changes, the cost-effectiveness increases are those directly derived from the results in Table 11 and Figure 19. For this limit, PROP is on average 0.58 times (0.58x) more cost-effective than STMT, CHAIN is 0.24x more cost-effective, DU is 0.9x more cost-effective, and BR is 0.00x more cost-effective (i.e., BR and STMT are equally cost-effective). As the detectability limit decreases, however, Figure 20 shows that the cost-effectiveness superiority over STMT of all coverage-based and propagation-based strategies grows substantially. For instance, at detectability limit 0.10 (which includes 30 changes) PROP is already more than five times more cost-effective than STMT, and for limits of 0.05 or less (corresponding to 22 changes), PROP is 8–51 times more cost-effective. Although the other three strategies also exhibit a considerable improvement over random testing, their growth is much slower than the growth of PROP. CHAIN is twice as effective as STMT (i.e., 1.03x) for a detectability of 0.8 or less, whereas BR and DU achieve the same improvement but only for a detectability lower than 0.2. CHAIN, however, is still no match for PROP, as the graph indicates. Thus, our results that distinguish detectability limits to isolate the hardest-to-test changes (i.e., changes whose detectability is quite low) strongly supports the superiority of PROP as a change-testing strategy.

Note, however, that the lower the detectability limit is, the fewer changes are considered and thus the fewer data points are used in our analysis, which can affect the statistical significance of our conclusions. For that reason, we used t-tests to compute the statistical

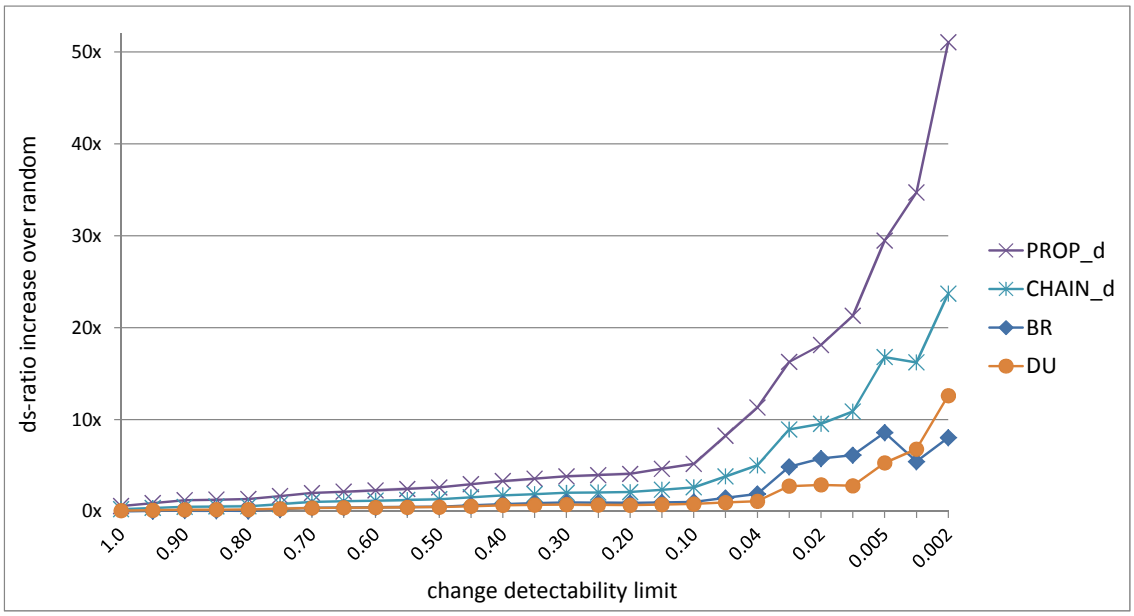


Figure 20: Cost-effectiveness (ds-ratio) increase over STMT of all strategies per detectability limit.

Table 14: Statistical confidence of advantage of PROP over the other strategies per detectability limit.

Detectability limit	**1.0**	**.8**	**.6**	**.4**	**.2**	**.1**	**.05**	**.03**	**.01**	**.005**	**.003**	**.001**
PROP vs STMT	✓	✓	✓	✓	✓	✓	✓	✓	†	×	×	×
PROP vs BR	✓	✓	✓	✓	†	†	×	×	×	×	×	×
PROP vs DU	✓	✓	✓	✓	‡	†	×	×	×	×	×	×
PROP vs CHAIN	✓	✓	✓	✓	✓	✓	✓	✓	×	×	×	×

significance (and, therefore, the confidence) of our results per detectability limit. Table 14 shows these confidence values for the observed superiority of PROP (on average for all distances) over the other strategies. The markers in this table are the same as in Table 12: × for no confidence, † for 95-99% confidence, ‡ for 99-99.9% confidence, and ✓ for 99.9% or higher confidence. The confidence levels that we obtained indicate that, for changes within a detectability limit of 0.01 or greater, there is enough confidence that PROP is superior to STMT. With respect to BR and DU, there is confidence only for detectability limits of 0.1 or greater, even though with respect to CHAIN the confidence is achieved for at least a limit of 0.03. This apparent counter-intuitive result is explained by the fact that

the ds-ratios BR and DU do not vary alongside PROP as closely as the ratios for CHAIN do, despite being inferior on average than those for CHAIN. The most likely reason for this effect is that CHAIN is a subset of PROP and therefore CHAIN has a much stronger statistical dependence on PROP than BR or DU do.

For detectability limits of 0.01 or less, there is little or no confidence that PROP is superior to the other strategies, despite the large differences reported in Figure 20 at those limits, simply because there are not enough data points—9 or less—to make any conclusive remark. Nevertheless, for detectability limits as low as 0.1, we can conclude that PROP is a superior strategy with a winning margin that grows as the limit decreases. For lower limits, we expect that larger studies on changes that are difficult to test will help determine whether the promising results that we obtained in this study for those low limits are also statistically significant.

In summary, for RQ3 on these changes and subjects, we can conclude that:

- PROP provides statistically-significant improvements in cost-effectiveness over the other strategies and these benefits are particularly large when the testing difficulty of the changes is high (i.e., their detectability is low). This conclusion is particularly important because low-detectability changes are those in greatest need for a good testing strategy. Despite the increase in size of our study, however, larger studies are needed to confirm statistically the particularly dramatic improvement of PROP observed on changes that are difficult to test.

- For many changes, it is easy to detect a difference during testing, but for about half the changes it is quite difficult to detect them (the detectability is 0.1 or less). Hence, STMT (i.e., simply covering the change once) is insufficient for testing changes. Moreover, the easily detectable changes are likely to be detected by an existing regression test suite that already covers them, so the real need for testers is a good strategy (e.g., PROP) for hard-to-detect changes. Coverage-based strategies show some

gains over STMT only for the hardest changes, but propagation-based strategies are the best—particularly PROP.

6.4.2.4 *RQ4: Cost of Propagation-based Strategies*

Computing propagation constraints in the studies of Chapter 5 often took one hour for small distances [92, 94]. In contrast, our new approach, which we studied thoroughly in this section, usually took only a few minutes. Meanwhile, the runtime overhead for the test cases was about 600%. This overhead was not as high as we expected, considering than the overhead of simply monitoring du-pairs can be more than 100% [93] even with efficient instrumentation and monitoring methods. Moreover, we expect that our tool can be optimized considerably to reduce this overhead.

6.4.3 Threats to Validity

The main internal threat to the validity of our conclusions is the potential presence of errors in our toolset that can distort the coverage and state differences that we measured for each test. To minimize this threat, we included diverse checks at different points in the toolset and manually inspected the results for several changes. Also, it is worth noting that DUA-FORENSICS has been under constant development and improvement for some years already [93], and has been used and tested in our previous work [92–94, 97, 98] as well as ongoing work [95].

Another internal threat to the validity of our study arises from problematic features of the Java language that are not well supported by the Soot Framework [91] that DUA-FORENSICS uses, such as reflection and exception flow. Reflection in Java causes transitions of control from one part of the program to another that, in general, cannot be captured by any analysis. Potential exception flows caused by all the exceptions that can potentially be thrown from each statement are excessive in number when identified by a safe analysis system such as Soot. Therefore, for our experiments, and for most uses of systems like Soot, we ignored such potential flows to allow our technique to operate in a reasonable

amount of time and memory. Because of this threat, our implementation of propagation-based strategies might have underestimated the set of propagations of change effects that took place when executing the modified programs in our study.

The main external threat to the validity of our results is the representativeness of our subjects and the tests and changes provided with them. We minimized this threat by using more subjects and changes than in previous studies—62 different changes in total—that allowed us to provide strong statistical assurances for our results so that our conclusions can be extrapolated with confidence to other subjects and changes of similar characteristics. To further generalize our conclusions, however, it is necessary to incorporate even more changes and subjects of different types and size and to reach greater levels of satisfaction of the testing requirements for all strategies.

6.5 Case Studies of Demand-driven Strategies

The goal of this section is to provide complementary evidence to the study of Section 6.4 about the practicality and scalability of propagation-based testing strategies. The study presented in that section uses pools of test cases provided with each subject as the way to represent the test cases that developers create in practice. However, to assess the practicality, usability, and scalability of the propagation-based techniques, it is also necessary to investigate whether these test cases can be created by developers to satisfy the testing requirements presented to them by those techniques, even for large subjects.

To achieve this goal, we present next two case studies of test-suite augmentation that we carried on different subjects and changes. First, Section 6.5.1 describes the empirical setup for the case studies. Then, Sections 6.5.2 and 6.5.3 present and analyze the results of these studies. Finally, Section 6.5.4 discusses threats to the validity of our studies.

Table 15: Subjects, test cases, and changes for case studies of change testing.

subject	description	LOC	test cases	change
Schedule1	process scheduler	290	2650	v1
Ant 1.8.2	software-build automation tool	131K	307	r663061

6.5.1 Empirical Setup

For the case studies in this section, we used the two subjects and changes listed in Table 15. Column *subject* names the subject, column *description* gives a brief description of its purpose, column *LOC* gives the size of the subject in lines of code, *test cases* indicates how many test cases are provided with the subject, and column *change* names the change studied for each subject. The name of a change is either the version in the Siemens suite [60] (i.e., *v1*) or the release in the corresponding Subversion [30] repository (i.e., *r663061*).

The first subject, Schedule1, belongs to the Siemens suite that we translated from C to Java. Schedule1 serves as a simple starting point for our manual study of our technique. The second subject, Ant 1.8.2 [4], is a popular *make*-like build tool written in Java and used primarily for Java projects. We chose this version of Ant because its size (greater than 100K lines of code) and complexity let us demonstrate the scalability and practicality of our technique. For Ant, we studied a change that we call *r663061* and that we obtained from the Subversion repository for this subject—revision 663061.

To perform the case studies, we added to our implementation of the demand-driven approach (Section 6.3) the ability to interact with the user. The tool starts by analyzing an existing test suite and collecting the testing requirements it satisfies. The tool identifies the points in the program where uncovered dependencies can start new chains or branch out of existing chains to satisfy new chain requirements. The report also includes dependencies in covered chains for which the state requirements are not satisfied. The tool presents to the user these chain and state requirements that lie at the vicinity of satisfied requirements. The requirements are ordered by dependence-distance from the change to let the user decide

how to prioritize those requirements.

Using this report, the user constructs a new test case intended to satisfy new requirements and feeds that test case to the tool. Then, the tool analyzes that test case by executing it and determining whether it covers new requirements or not. If the test case does satisfy new requirements, the tool adds the test case to the test suite. If not, the tool rejects the test case. In either case, the tool communicates the outcome to the user and presents again to the user the (possibly updated) report of unsatisfied testing requirements. The user then decides whether there are more testing requirements that need to be satisfied and, if so, whether the testing budget allows the next attempt.

For each subject and change listed in Table 15, we used DUA-FORENSICS and the pool of available test cases to create a starting test suite for the corresponding augmentation study. We did not use this pool to simulate the creation of new test cases like in previous studies. We created each starting test suite to maximize statement coverage. Such test suites are representative of industrial-strength test suites for software that is developed and tested professionally. We obtained each of these test suites by having our tool randomly pick one test case at a time from the pool, without replacement, and then determine whether the test case increased statement coverage. If it did, the test case was added to the test suite. The process stopped when no more test cases were left in the pool that could increase coverage.

For each change in our study, we created a new starting test suite using this method and we augmented the test suite using the method described by algorithm SATISFYREQSDEMAND in Figure 17 and Section 6.4.1. In our case studies, however, we created each new test case manually instead of selecting it from the pool. We augmented each starting test suite separately using each of the following testing strategies:

1. DU, the most cost-effective (albeit marginally) of the coverage-based strategies according to the study of Section 6.4

2. CHAIN, our propagation-based strategy that includes chains but not state conditions,

to help distinguish the separate contributions of chain and state requirements

3. PROP, our propagation-based strategy with both chain and state requirements

For each strategy, we set a time limit of 90 minutes for augmenting each test suite. This limit corresponds to the testing budget B in SATISFYREQSDEMAND. At the end of that period, we stopped and moved on to the next strategy or the next change. For each strategy and each change, within that period, we iteratively picked as our goal one unsatisfied testing requirement or set of requirements located close to each other. For each goal, we created a new test case to try to satisfy it. During this process, we marked every testing requirement we found that was unsatisfiable—whether it was part of our goal or not—to avoid targeting it again later and to reduce our estimated count of feasible requirements. If the entirety of the goal was found unsatisfiable, we moved on to the next goal.

For each test case that we created to try to satisfy the goal, we used our tool to determine whether new testing requirements for the strategy were satisfied. If so, we added the test case to the test suite, even if the newly satisfied requirements were not part of the goal (i.e., requirements satisfied "accidentally"). If not, we continued by inspecting the test case to understand why it failed to satisfy the goal, fixing the test case or creating a new one, and then repeating the process. We also recorded whether the new test case produced a different output before and after the change—our measure of success.

For the estimated effort, we took into account the number of minutes spent selecting each goal, checking whether the goal was satisfiable, creating each new test case, and running our tool (the last one being negligible in comparison to the manual activities). We did not count in this effort the time we spent fixing problems with our tool or improving it. We did not count either any interruptions or activities unrelated to the iterative creation of new test cases. We adjusted, however, for each new subject and change, the estimated effort for each augmentation after the first augmentation on that subject to account for the knowledge gained in the first augmentation, which helped perform the next augmentations more

quickly. To further minimize this issue, we picked a new strategy for the first augmentation for each new change. In all, our effort estimates were made to represent faithfully the time spent by a developer with similar experience and ability for each augmentation.

6.5.2 Results and Analysis: Schedule1

The change we studied for Schedule1 was *v1*, the first change provided with this subject, which corresponds to the fix of a bug in a function that finds a process in a list [60]. The bug lies in a termination condition of the loop that traverses the list, making the traversal continue past the end of the list under specific circumstances.

For our Java implementation of Schedule1, the statement coverage for the entire program that the test cases in the pool achieve is 97%. Despite the large number of test cases in the pool for this subject, however, less than 0.2% of them show a difference in the output for change *v1*. Thus, not surprisingly, the starting test suite that DUA-FORENSICS generated did not find any difference. This test suite contained six test cases.

Tables 16, 17, and 18 describe the augmentation processes we carried out for this change using DU, CHAIN, and PROP as testing strategies, respectively. Each row describes one step in the augmentation and is numbered by column *step*. For each step, the next columns describe various aspects of the step and results at the end of the step: the size of the resulting test suite, the number of differences found at that point, a short description of the action carried on in that step, the accumulated effort in minutes, and the percentage of testing requirements satisfied at that point. For space reasons in these tables, we used the abbreviation *req.* for *testing requirements*. For CHAIN and PROP (Tables 17 and 18), the last column is subdivided in sub-columns, one for each distance of interest for the augmentation, starting with the first distance that was not 100% satisfied from the beginning. For the testing requirements in DU, there is no distance distinction among them. For each step, the action was one of the following:

- *analyzed test suite*: determined which testing requirements were satisfied by the test

Table 16: Test-suite augmentation using DU for Schedule1, change *v1*.

step	test-suite size	differences	action performed	effort	req. satisfaction
1	6	0	analyzed test suite	5	77.11%
2	6	0	infeasible req.	35	80.40%
3	7	0	satisfied req.	50	81.82%
4	8	0	satisfied req.	70	82.10%
5	8	0	tried more req.	90	82.10%

suite

- *infeasible req.*: identified unsatisfiable (infeasible) requirements while trying to create the next test case
- *satisfied req.*: satisfied new requirements with a new test case and added the test case to the test suite
- *tried more req.*: tried to satisfy new requirements but run out of time

To illustrate, consider the row for step 2 in Table 17. The test suite in that step grew by one test case, from 6 test cases in step 1 to 7 test cases. The number of differences, however, stayed at 0—neither the initial test suite nor the new test case found a difference in the output. The action *satisfied req.* indicates that new testing requirements were satisfied in this step. The accumulated effort, because of this action, increased from 5 to 30 minutes (i.e., the step took 25 minutes). Lastly, the coverage of chains at distance 5 reached 100% due to this new test case, 98.2% for distance 6, and 59.0% for distance 7. In consequence, for the next step, an unsatisfied requirement at distance 6 was picked.

For all three augmentations, the action in the first step was always to analyze the initial test suite by running our tool to determine which testing requirements were already satisfied by the test suite and to identify the sets of unsatisfied testing requirements to target next, ordered by distance. For CHAIN and PROP, our goal was always to achieve 100% satisfaction at each distance starting from the change before moving on to the next distance. The reason for this prioritization was two-fold: (1) satisfying a testing requirement r at

Table 17: Test-suite augmentation using CHAIN for Schedule1, change *v1*.

step	test-suite size	differences	action performed	effort	req. satisfaction		
					d5	d6	d7
1	6	0	analyzed test suite	5	62.5%	43.9%	32.0%
2	7	0	satisfied req.	30	100.0%	98.2%	59.0%
3	8	0	satisfied req.	80	100.0%	100.0%	61.9%
4	8	0	tried more req.	90	100.0%	100.0%	61.9%

Table 18: Test-suite augmentation using PROP for Schedule1, change *v1*.

step	test-suite size	differences	action performed	effort	req. satisfaction	
					d2	d3
1	6	0	analyzed test suite	5	40.00%	25.00%
2	6	0	infeasible req.	15	66.67%	33.33%
3	7	1	satisfied req.	20	100.00%	50.00%
4	7	1	tried more req.	90	100.00%	50.00%

distance d is a requisite for satisfying requirements that extend r and (2) a testing requirement r at a distance d represents a potentially larger set of behaviors than individual testing requirements that extend r.

Whenever we discovered infeasible requirements while trying to create the next test case, we reported this event as a separate step to help understand the costs of dealing with such requirements and to examine how much the satisfaction of requirements, as a percentage, increased thanks to this discovery. For example, step 2 in Table 18 indicates that finding infeasible PROP testing requirements while trying to add the first new test took 10 minutes after the initial 5-minute analysis and that this discovery reduced the number of satisfiable requirements. As a consequence, the satisfaction as a percentage at distances 2 and 3 increased by 26.67% and 8.33%, respectively.

For this change and starting test suite in Schedule1, as tables 16 and 17 show, the augmentation using DU and CHAIN did not reveal any difference in the output within the 90-minutes budget, despite the addition of at least one new test case using each strategy. For PROP, in contrast, the augmentation succeeded at finding a difference, which happened

after only 20 minutes of work. The reason for the success of using PROP was that, unlike, DU and CHAIN, which focus on dependencies but ignore state conditions, our tool notified to us that at distance 2 there were covered chains that did not satisfy the state conditions necessary to guarantee a propagation of the effects of the change. DU and CHAIN, in contrast, would skip the dependencies at distance 2 because they were already covered and would focus instead on uncovered dependencies farther from the change.

In conclusion, for this change in Schedule1 and this starting test suite, PROP was essential for capturing and communicating to the tester the conditions that had to be satisfied for the effects of the change to propagate. DU and the partial version of our propagation-based approach, CHAIN, could not detect this need at distance 2 and caused the tester to waste effort covering dependencies or extending chains in other points of the program.

6.5.3 Results and Analysis: Ant

Our second case study emphasizes the practical ability of our change-testing technique to scale to and work in practice on applications of daily use. Thus, we chose a recent version of Ant [4], 1.8.2, whose size exceeds 100K lines of code (see Table 15). Ant is a popular, complex, and mature open-source application for automating the build process of software projects and is written in Java.

The change we studied was introduced by the developers of Ant in revision 663061 of the Subversion [30] repository for this software. Thus, we called this change *r663061*. We chose this change by first selecting the core class *Project* from package *org.apache.tools.ant*, which represents a project during a build. Then, we navigated the history of this class in the repository backwards in time by examining the changes committed for this class and stopped at the first change that could have an observable effect in the code of this class: revision 663061.

The location of the change is method *executeTargets(Vector names)* of class *Project*, which takes the list of the names of the build targets (groups of tasks, such as compiler

Table 19: Test-suite augmentation using DU for Ant 1.8.2, change *r663061*.

step	test-suite size	differences	action performed	effort	req. satisfaction
1	29	0	analyzed test suite	5	1.91%
2	29	0	infeasible req.	10	1.92%
3	30	0	satisfied req.	30	1.93%
4	30	0	infeasible req.	35	1.94%
5	31	0	satisfied req.	45	1.96%
6	31	0	infeasible req.	55	1.96%
7	32	0	satisfied req.	85	1.97%
8	32	0	tried more req.	90	1.97%

calls) specified by the user of Ant and executes each target. In this method, the change is code inserted just before the execution of the tasks to store as a string the names of the targets in a map that maintains the properties defined by the user.

Ant 1.8.2 is equipped with a pool of 307 unit-test classes (see Table 15), many of which contain more than one test method. Nevertheless, in this study, we regarded each class as a single test case. Thus, when we run our tool on each of these test cases, the tool reported for that test case the coverage of all test methods in it. The statement coverage our tool measured for the entire pool was 76%.

Interestingly, none of these test cases revealed a difference in the output for the change we studied, which implies that the developers forgot to augment or update their test suite when adding this change. In revision 663061 in the repository, no test case was created and no updates were made to existing test cases.

As in our first case study, we used our tool to randomly pick from the pool one test case at a time, without replacement, and added that test case to our initial test suite if the test case contributed any new statement coverage. The process stopped when no additional coverage could be achieved. The resulting initial test suite consisted of 29 test cases—many more than the six test cases for Schedule1. This is explained by the much greater size of Ant, which gives more opportunities to different test cases to cover unique parts of the program. Given this initial test suite, we proceeded to augment it separately using each of the DU,

Table 20: Test-suite augmentation using CHAIN for Ant 1.8.2, change *r663061*.

step	test-suite size	differences	action performed	effort	req. satisfaction		
					d3	d4	d5
1	29	0	analyzed test suite	5	25.0%	71.4%	43.5%
2	29	0	infeasible req.	20	100.0%	100.0%	47.9%
3	30	0	satisfied req.	40	100.0%	100.0%	81.1%
4	30	1	infeasible req.	45	100.0%	100.0%	82.0%
5	31	1	satisfied req.	60	100.0%	100.0%	83.8%
6	32	1	satisfied req.	70	100.0%	100.0%	85.6%
7	32	1	tried more req.	90	100.0%	100.0%	85.6%

Table 21: Test-suite augmentation using PROP for Ant 1.8.2, change *r663061*.

step	test-suite size	differences	action performed	effort	req. satisfaction		
					d3	d4	d5
1	29	0	analyzed test suite	5	25.0%	71.4%	42.5%
2	29	0	infeasible req.	20	100.0%	100.0%	46.7%
3	30	0	satisfied req.	40	100.0%	100.0%	79.9%
4	30	1	infeasible req.	45	100.0%	100.0%	80.8%
5	31	1	satisfied req.	60	100.0%	100.0%	81.4%
6	32	1	satisfied req.	70	100.0%	100.0%	82.6%
7	32	1	tried more req.	90	100.0%	100.0%	82.6%

CHAIN, and PROP strategies.

Tables 19, 20, and 21 present the augmentation process for each of these strategies following the same format described in the case study of Schedule1 in Section 6.5.2. The method used for obtaining these results was the same as in Section 6.5.2. In each table, column *step* enumerates the steps of the respective augmentation. For each step, the corresponding row contains the size of the test suite at the end of the step, the differences found so far, the action performed in that step, the effort spent so far (in minutes), and the percentage of the testing requirements satisfied. Each of the tables for CHAIN and PROP has three sub-columns within the last column representing the distances for which testing requirements were targeted at some step during augmentation. For these strategies, the satisfaction was 100% at distances 1 and 2, so the first distance shown is 3.

For each of the three strategies, the number of steps that we completed in 90 minutes was greater than for the corresponding augmentation in Section 6.5.2. The reason for this difference is that, despite the greater complexity of Ant, the pool of test cases provided with it is not as exhaustive as the pool for Schedule1 and, therefore, the initial test suite was able to cover shorter distances than for Schedule1. This is also the reason why the starting distance for CHAIN in this case study was much shorter than in the previous study.

The satisfaction levels for DU, as Table 19 shows, were quite small—all of them just below 2%. The reason for this phenomenon is not only that the test-case pool was smaller for Ant than for Schedule1, but also that Ant is a much bigger program and thus the number of du-pairs possibly affected by the change was much larger. Although the coverage levels for CHAIN and PROP shown in tables 20 and 21 were much greater than for DU, it is important to remember that the percentages in those tables are for distances of only 5 or less. For greater distances, the number of CHAIN and PROP testing requirements grows exponentially and much faster than the number of satisfied requirements.

In terms of effectiveness, we were again unable to detect any difference when augmenting the initial test suite with DU. Table 19 shows that, within the 90-minute budget, we were able to add three new test cases in addition to identifying three sets of infeasible du-pairs. For each of CHAIN and PROP, in contrast, we succeeded at finding a difference with the second test case that we constructed and it took 45 minutes for both strategies to find it. The reason why the augmentations for CHAIN and PROP were almost identical is that, for almost all covered chains, the state conditions were satisfied. Only for distance 5 the satisfaction was slightly lower for PROP than for CHAIN due to dependencies at the end of a few of these chains that failed to propagate a state difference. Thus, within the 90-minute budget, the testing requirements targeted in order of distance were almost the same for CHAIN and PROP, and therefore the effort for each step was the same.

In conclusion, similar to the first case study, the best coverage-based strategy, DU, was

unable to find an observable difference for the change, unlike the propagation-based strategies CHAIN and PROP, which found a difference half-way through the process. For this change, unlike the previous case study, both CHAIN and PROP were equally cost-effective. Therefore, this change in Ant is an example for which CHAIN alone is responsible for the effectiveness observed. Also, although we cannot generalize from only one change in a large subject and two changes in total, these two case studies highlight that propagation-based testing requirements can be satisfied as easily in practice as coverage-base requirements and that the difficulty of satisfying them does not necessarily depend on the size or complexity of the program. This conclusion is consistent with the fact that, because of the distance limits, propagation-based strategies are affected mostly by the type of the change and the surrounding code rather than the entire program.

6.5.4 Threats to Validity

The main internal threats to the validity of these case studies are (1) the possible existence of errors in our tool and in our use of the tool and (2) the representativeness of the author of this study as a tester using these test-suite augmentation strategies. The first threat was mitigated by the maturity of the underlying DUA-FORENSICS tool. The second threat was addressed by carefully designing the manual augmentation process to make it predictable and repeatable. Although the effort that different testers would spend augmenting these test suites may vary according to their expertise, we expect that these efforts will vary by the same proportion.

The main external threat to the validity of our conclusions is that we studied only two changes. Although each case was studied in detail, drawing any conclusion about the effectiveness of propagation-based strategies with manual (or automatic) test generation would require a considerably larger number of subjects and changes to make reliable predictions. Instead of asserting with certainty the effectiveness of our technique using this manual approach, the purpose of these studies was simpler: to show that it is possible to perform

propagation-based test-suite augmentation in practice by creating new test cases and that our technique and its implementation can scale to a program like Ant.

6.6 Related Work

The technique for computing propagation-based testing requirements presented in Chapter 5 and Reference [92] directly precedes the work in this chapter. This technique provides a complete view of all possible effects of a change and presents those effects as testing requirements in a way amenable for use in tools and constraint solvers. However, as discussed in Section 6.1, this technique is subject to the scalability problem of symbolic execution, even after accounting for the gains that SPD from Chapter 4) provides. Also, the number of testing requirement this technique generates can be quite large and hard to satisfy.

In contrast, the demand-driven alternative presented in this chapter circumvents the need for symbolic execution by placing the emphasis on the state differences observed in previous executions instead of all possible state differences identified statically. This new approach points to the tester the locations in the vicinity of the change or covered chains where unsatisfied testing requirements lie. As a result, the new technique lets the tester sacrifice the view of all possible effects of a change within a short distance in exchange for the ability to monitor the coverage of chains and state differences at greater distances.

The existing coverage-based testing techniques [17, 50, 87] described and studied in Chapter 5 constitute the next most-closely related work to the demand-driven, propagation-based approach of this chapter. The evaluation of those techniques in Chapter 5, although limited to the distances achieved by the static computation of propagation-based testing requirements for a side-to-side comparison, already showed the inadequacy of simple coverage-based testing of changes. The comprehensive study presented in this chapter not only confirms that inadequacy, but also puts it in full perspective with respect to the real potential of propagation-based testing demonstrated using the demand-driven approach.

In parallel to the work presented in this chapter, related techniques have been proposed

for test-suite augmentation that iteratively modify a test input to expose a difference in the output [80] or to increase program coverage after the program changes [117, 118]. These test-input generation techniques are mostly complementary to ours—they can be used to create test cases that satisfy the requirements computed or pointed to by our approaches. A distinctive aspect of our approaches with respect to those techniques, however, is that they cover the space of all effects of a change instead of focusing all resources on one behavior that might or might not affect the output. This broader treatment of the effects of changes lets our approaches distribute the testing effort over all affected behaviors of the program to provide a more complete picture of the change to the tester.

CHAPTER VII

ANALYSIS AND TESTING OF MULTIPLE CHANGES

The most general and typical scenario that developers face is when multiple changes are made to software. Analyzing changes in this context can be a much more challenging task. When multiple changes are made, computing the effects of each change individually might be insufficient to test and validate them. In such cases, developers must also consider the effects that each change has on other changes.

This chapter addresses the challenges posed by the presence of multiple changes for the analysis and testing of the effects of changes. Section 7.1 describes how the propagation-based testing requirements presented in Chapters 5 and 6 can be adapted for multiple-change scenarios. Then, Section 7.2 presents and evaluates a new technique that detects with unprecedented precision whether changes affect each other at runtime. Finally, Section 7.3 addresses other work that relates to this chapter.

7.1 Adapted Testing Requirements for Multiple Changes

The propagation-based technique of Section 5.2 computes testing requirements that represent the possible effects of a change within some limits. However, the *state* requirements for a change computed by this technique can be affected by the effects of other changes. In Section 7.1.1, we present a solution for dealing with such circumstances and, in Section 7.1.2, we present a case study that uses this solution [92].

7.1.1 Multiple-change Context for Individual Changes

Given a change C_1 and a state requirement r for C_1, the technique of Section 5.2 assumes that the set of variables V that appear as symbols in r's state constraints have the same values at the entry of C_1 in the original program P and the modified program P'. Hence, if

a test case executes another change C_2 before C_1, and C_2 causes one or more variables in V to have a different value at the entry of C_1 (i.e., C_2 *infects* the parts of the state used by r), then r is no longer applicable for that test case.

One solution for handling such cases is to regard the state constraints for r as "not satisfied" if any of the variables used in r is infected at the entry of the corresponding change. For example, consider program $E_{1,2}$, which is obtained by applying changes 1 and 2 to program E from Figure 13. The state requirement for chain $<5, (5, 6, y)>$ from change `ch2` in $E_{1,2}$ requires that the value of y at statement 6 be different in E and $E_{1,2}$. However, the value y_0 of y at the entry of change `ch2` will be different in E and $E_{1,2}$ because change `ch1` executes first and alters y_0. In that case, the conditions for the state requirement for a chain, when a variable that appears as a symbol in that requirement has been infected, might no longer represent the state-propagation conditions for that chain.

In this small example, we could obtain the symbolic value of y_0 at statement 5 in E and $E_{1,2}$ by analyzing change `ch1` at distance 2 and then replacing y_0 with the respective symbolic results in the state requirements for change `ch2`. However, in many cases, changes are located far away from each other, so the distances that can be reached in practice by COMPUTEREQS for one change are too short to obtain the symbolic value of an infected variable at the entry of another change. In such cases, for safety, when a variable in a state requirement is infected, that requirement should be considered invalid (i.e., unsatisfiable).

To solve this problem, we adapt our propagation-based techniques to use a conservative but practical approach to determine state-requirement validity: assume that the entire state of the program is infected after executing a change. This solution distinguishes each occurrence of a change in an execution, so that at runtime, after a change C executes once, the monitoring of the satisfaction of state requirements is disabled for all other changes, including the next execution of C if it occurs. When this monitoring is disabled, state requirements are not reported as satisfied even if their conditions are satisfied. Chain requirements, however, are always monitored and reported when satisfied.

To illustrate, consider program E Figure 13 and both changes. Change `ch2` can only execute after change `ch1`. Therefore, for all executions, only chain requirements are monitored for change `ch2`. However, if program E in Figure 13 is modified at statements 2 and 7, then the state requirements for the change at statement 7, which define the effect of this change on the output value, are invalidated only for executions in which the change at statement 2 causes a modification of the decision at statement 6. A modification of this decision is manifested as an infection of the program state for the change at statement 7 by executing this change in only one of the two versions of the program.

7.1.2 Case Study: Multiple-change Context for Individual Changes

This section presents a case study of interaction between two changes and an analysis of the consequences on the monitoring of requirements and the detection of differences of applying the multiple-change handling solution of Section 7.1.1 for these two changes.

An interesting interaction occurs between Changes 1 and 4 in NanoXML-v1. Change 1 is located in the DTD (Document Type Definition) parsing component, whereas Change 4 alters the final step of XML element attribute processing. Suppose that P_1 is the program version with Change 1 only, P_4 the version with Change 4 only, and $P_{1,4}$ the version with both changes. Out of 214 test cases available, 42 test cases show an output difference between the two programs. As expected, given that the difference detection using PROP on each of these changes alone was 100% (see the study in Section 5.4.1), all test suites created for $P_{1,4}$ using the multiple-change handling approach of Section 7.1.1 obtained also a 100% difference detection.

In $P_{1,4}$, Change 4 is executed by 164 test cases, out of which only seven test cases are difference-revealing. Interestingly, 135 test cases reach Change 4 before Change 1, but none of these test cases reveal a difference. Only seven test cases that cover Change 4 reveal a difference in the output, and in all seven cases Change 4 is infected by Change 1. Thus, state requirements in $P_{1,4}$ for Change 4 are never satisfied by difference-revealing

test cases, even though many other test cases reach this change without infection. In P_4, in contrast, Change 4 is covered by 26 difference-revealing test cases. Because Change 4 greatly benefits in P_4 from satisfying PROP over CHAIN (see Table 7), we would expect $P_{1,4}$ to exhibit a reduction in the ability of PROP to detect differences attributable to Change 4.

To measure the impact of Change 1 on the ability of Change 4 to cause output differences in $P_{1,4}$ when applying PROP with the multiple-change handling approach, we executed all 164 test cases that cover Change 4 in this version and measured the coverage of chain requirements for Change 4. From these 164 test cases, we generated 100 unique test suites satisfying the chain requirements for Change 4 on $P_{1,4}$, of which 23.7% revealed a difference in the output. In contrast, the difference detection in $P_{1,4}$ for simply covering Change 4 is 3.6%. Thus, considering that using PROP achieves 100% detection for Change 4 in P_4, there is a decrease of 76.3% in difference detection for Change 4 when Change 1 is also present, due to interference from Change 1. Yet, the multiple-change handling approach for adapting our PROP testing requirements still exhibits a greater difference detection over just covering Change 4 (23.7% vs. 3.6%).

7.2 Interactions among Multiple Changes

The solution presented in Section 7.1 adapts the testing requirements of individual changes to scenarios with multiple changes to account for the possibility that changes interact (affect each other). This solution makes the assumption that a change is affected by another change that executes first. However, the problem of determining whether changes interact or not—and what their combined effects are if they do—can be addressed with much greater precision. This section presents a precise new technique [97] for achieving that goal that is based on the same foundations (Chapter 3) on which the propagation-based analysis of individual changes is built (Chapters 5 and Chapter 6).

First, Sections 7.2.1, 7.2.2 and 7.2.3 motivate, justify, and present, respectively, the new technique called *Semantic Change-Interaction Detection* (SCHID). Then, Section 7.2.5

presents a study that shows the inaccuracy that existing techniques can exhibit for detecting change interactions and highlights the necessity of the SCHID technique and the formal model of Section 3.1 for the analysis of multiple changes.

7.2.1 Motivation for Change-interaction Detection

Developers often introduce multiple changes in software to cooperatively accomplish a goal, so the combined behaviors of such changes should be explicitly tested. Also, multiple changes that are introduced for independent purposes and are not supposed to interact might actually interfere. Thus, testers need to know exactly which changes interact to ensure that related changes are cooperating as intended and that unrelated changes do not interact unexpectedly, so they can assess and augment their regression test suites appropriately. In particular, the approach presented in Section 7.1 for adapting the testing requirements for individual changes when other changes interfere can greatly benefit from a precise interference-detection method that improves the conservative method explored in that section.

Existing techniques in the literature do not provide accurate information about interactions among changes. Such techniques either use static slicing [112] (a notoriously imprecise technique [18, 23]) to identify potential interferences among code changes when merging programs [19, 56] or rely on coarse-grained dynamic analysis [113]. For change-impact analysis, for example, the Chianti technique [83] explicitly addresses relationships among multiple changes but operates at a coarse-grained level, identifying only high-level syntactic dependencies among changes and relating changes covered by the same test cases. (The study in Section 7.2.5 provides evidence of the inaccuracy of existing techniques for identifying runtime interactions among code changes.)

Fortunately, the formal model of the effects of changes presented in Section 3.1 of this

Table 22: Forward slices for the changes in the example program `prog'` of Figure 4.

change	static slice	dynamic slice (first occurrence) input: `n=5, m=[1,-1,0,3]`
`ch1`	1, 6, 7	1^1,6^1,6^2,6^3,6^4,7^1
`ch2`	4, 6, 7, 8	4^1,6^1,6^2,6^3,6^4,7^1,8^1
`ch3`	7	7^1
`ch4`	8	8^1

dissertation provides important new definitions of how a change alters the behavior of programs. This model, which serves as the basis for the practical computation of propagation-based testing requirements (Chapters 5 and 6), can also be used to define a technique for determining with unprecedented precision whether multiple changes interact or not and thus overcome the limitations of existing techniques for detecting change interactions. The reasoning that leads to this new technique and the technique itself are presented next.

7.2.2 Towards Accurate Change-Interaction Detection

Forward slices are important ingredients for computing the effects of changes in practice. Table 22 lists the forward static slices (second column) for the four changes in program `prog'` of Figure 4 as well as the forward dynamic slices (third column) for the first occurrence of each of these changes for input {`n=5, m=[1,-1,0,3]`}. In this example, the forward slices for the changed nodes 1, 4, and 7 (i.e., `ch1`, `ch2`, and `ch3`) are the same in `prog'` and `prog` (also shown in Table 1). Table 22 includes the slices for node 8 in `prog'`, which is inserted by `ch4`).

A naive first attempt to determine which changes might interact in a modified program would be to identify, for a given change C, all other changes that are in the static forward slice from C. For example, in the modified program `prog'` from Figure 4, the slice from `ch1` contains only `ch3`. Using such a slice would lead us to identify only `ch3` as affected by `ch1`. However, variable `r`, modified at Node 1 by `ch1`, can also be affected by change `ch2` before `r` reaches Node 7, because nodes 6 and 7 are also in the forward slices of `ch2`. For

instance, for input {n=5, m=[1,-1,0,3]}, the value of r that reaches Node 7 in program prog′ is 18. If only ch1 were made to the original prog, however, the value of r at that point would be 16 instead. Therefore, the effects of ch1 actually interact with the effects of ch2.

Because the goal is to find the actual interactions among changes in modified software, we use the model of Section 3.1.4. Thus, we regard ch1 and ch2 in this example as *semantically dependent* on each other:[1] there exists an execution, specified by an input, of prog′ for which *removing* any of these changes (i.e., replacing the modified code with the original code for the change) alters the effects of the other change on that execution. In other words, the presence of either one of these changes alters the effects of the other change. A semantic dependence is symmetric: if the effects of one change alter the effects of another change, then both changes are mutually dependent and, thus, interact. In an execution of prog′ with input {n=5, m=[1,-1,0,3]}, changes ch1 and ch2 interact because both alter the value of r for that input. For input {n=5, m=[0,-1,0,0]}, however, change ch2 is not executed, so it does not interact with ch1. (In general, a change might execute but not interact with other changes.)

A second attempt to solve this problem is to use an approximation to change interaction by using the intersection of forward slices to identify potential interactions. If the static forward slices of two changes intersect, then these changes are *syntactically* dependent (i.e., a dependence found by static analysis). In prog′, for example, the static slices of changes ch1, ch2, and ch3 intersect at Node 7. Also, the dynamic forward slices of these changes for input {n=5, m=[0,-1,0,0]} intersect, implying runtime syntactic dependencies among them.

A technique for change-interaction detection, however, should identify actual—not just potential—change interactions. For example, for the execution of prog′ with a value of 1 for n, there is no *semantic dependence* (i.e., dependence on the actual effects) between ch1

[1] Semantic dependence between changes is a symmetric relationship, as proven in Section 3.1.4.

and `ch3`, even though, syntactically, their static and dynamic forward slices intersect at Node 7. The reason is that `ch1` does not affect the semantics of `ch3` when `n` is 1—Node 7, changed by `ch3`, will still print 1. Similarly, `ch2` interacts semantically with only `ch3` for executions in which the combined effect of the two changes determines whether Node 7 prints 0 or 1.

7.2.3 A Precise Change-interaction Detection Technique

This section presents *Semantic Change-Interaction Detection* (SCHID), a new technique for detecting runtime interactions among changes. The technique is based primarily on Definition 20 of semantic dependence among changes from Section 3.1.4. This definition of dependence among changes, however, is static (i.e., for all possible executions) and determining whether two changes are semantically dependent is an undecidable problem. However, for a finite execution, it is possible to determine whether a semantic dependence between two changes occurs. We call the occurrence of such a dependence an interaction among those changes.

DEFINITION 27. Changes C_1 and C_2 *interact* in the execution of modified program P' on input I if a semantic dependence between C_1 and C_2 is exercised in that execution. In other words, C_1 and C_2 interact if the semantic effect of C_1 on P' and $P'\backslash_{C_2}$ for input I differs or if the semantic effect of C_2 on P' and $P'\backslash_{C_1}$ for input I differs.

Unlike the static version of semantic dependence, it is possible to detect dynamic semantic dependence for terminating executions or finite portions of an execution. Algorithm FINDINTERACTIONS in Figure 21 determines which changes do interact (i.e., whether a semantic dependence is exercised). The algorithm inputs the original program P, a list S of non-empty *sets* of changes, and the program input I. The reason for using sets of changes instead of individual changes is to let the user group related changes and find the effects among arbitrary groups of changes.

For each pair of change sets in list S, FINDINTERACTIONS determines whether the

Algorithm FINDINTERACTIONS

Input: P: program; S: list of change sets; I: input
Output: R: set of pairs of change sets

```
(1)   R := ∅
(2)   P' := applyChanges (P, S)
(3)   for i = 1 to |S|−1
(4)     C_i := S[i]
(5)     for j = i + 1 to |S|
(6)       C_j := S[j]
(7)       if prune(P', I, C_i, C_j)
(8)         continue at (5)
(9)       endif
(10)      exHistP' := augmExecHist (P', I, C_i)
(11)      exHistP'\C_i := augmExecHist (P'\C_i, I, C_i)
(12)      effectOnP' := diff (exHistP', exHistP'\C_i)
(13)      exHistP'\C_j := augmExecHist (P'\C_j, I, C_i)
(14)      exHistP'\C_i∪C_j := augmExecHist (P'\C_i∪C_j, I, C_i)
(15)      effectOnP'\C_j := diff (exHistP', exHistP'\C_i)
(16)      if effectOnP' ≠ effectOnP'\C_j
(17)        R ∪:= <C_i, C_j>
(18)      endif
(19)    endfor
(20)  endfor
(21)  return R
```

Figure 21: Algorithm for accurately detecting runtime change interactions.

two sets in the pair interact (i.e., whether a semantic dependence between those sets is exercised according to Definition 27) during the execution of P' on I. To do that, the algorithm determines whether any change in one set interacts with any change in the other set by analyzing their static and dynamic forward slices, the coverage of those changes, and the differences in the augmented execution histories of those changes on different versions of the program. The algorithm then determines whether the effects of one change set alter the effects of the other change set, in which case these sets interact.

At line 1, the set R of interacting change sets is initialized to the empty set. At line 2, the algorithm constructs the modified program P' by applying all changes in S to the original program P. The loop at lines 3–20 and its nested loop at lines 5–19 analyze each pair of

change sets (lines 7–15) and, if the pair interacts, adds the pair to the result R (lines 16–18). To determine whether change sets C_i and C_j interact, the algorithm first invokes *prune* at line 7 to check some necessary but not sufficient conditions for the interaction of C_i and C_j; if any of these checks fails, the algorithm skips the analysis of lines 10–18 by continuing to line 5 for the next pair of change sets. The use of *prune* lets the algorithm eliminate some pairs of change sets through a simpler analysis, so that the more expensive subsequent analysis is applied to a smaller number of pairs.

The conditions checked by *prune* (not listed in Figure 21) are, in the order checked:

1. The static forward slices of some change in C_i and some change in C_j intersect on P', $P'\backslash_{C_i}$, $P'\backslash_{C_j}$, or $P'\backslash_{C_i \cup C_j}$.[2]

2. The execution of P, $P'\backslash_{C_i}$, $P'\backslash_{C_j}$, or $P'\backslash_{C_i \cup C_j}$ includes (covers) a change in C_i and a change in C_j.

3. The dynamic forward slices of a change in C_i and a change in C_j on the execution of P', $P'\backslash_{C_i}$, $P'\backslash_{C_j}$, or $P'\backslash_{C_i \cup C_j}$ intersect.

To illustrate, consider change sets {`ch3`} and {`ch4`} in program `prog'` (Figure 4). For this pair, Check 1 finds that the static slices of `ch3` and `ch4` do not intersect in `prog'`, as Table 22 shows, or any other program version. Thus, these changes cannot interact for any execution, so they need not be considered further. Next, consider an execution of `prog'` for an input in which all elements of `m` are negative. In this case, any pair in which one set is {`ch2`} passes Check 1 because the static slice of `ch2` intersects all other static slices (see Table 22), but Check 2 discards this pair because `ch2` is not covered by such an input. For input {`n=5`, `m=[3,-1,-1,-1]`}, however, the pair {`ch2`} and {`ch4`} passes the first two checks but fails Check 3 because the dynamic slices of `ch2` and `ch4` do not intersect for any program version. Finally, if all checks pass, FINDINTERACTIONS continues to line 10.

[2]For accuracy, the interprocedural static-slice intersection should account for calling contexts [19].

At lines 10 and 11, FINDINTERACTIONS invokes the auxiliary algorithm *augmExecHist* (not listed) to compute the augmented execution histories of the changes in C_i for the modified program P' and the version of P' without C_i, respectively. At line 12, the algorithm computes the differences between these augmented execution histories, which correspond to the semantic effect of C_i on P' (see Definition 18 in Section 3.1.3). Also, lines 13–15 compute the semantic effect of C_i on $P'\backslash_{C_j}$ (i.e., the modified version P' without C_j). Line 16 compares the two effects and line 17 adds the pair to R if those effects differ (i.e., if C_j altered the semantic effect of C_i on P'). *augmExecHist* and *diff* cache their results for later iterations to use.

To illustrate lines 10–18, consider again Figure 4 (`prog'` is P') and single-change sets $\{$`ch1`$\}$ and $\{$`ch2`$\}$. For the execution of P' on input $\{$`n=5, m=[1,-1,0,3]`$\}$, the column labeled *dynamic slice* in Table 22 shows the part of the augmented execution history of the first occurrence of each change without state information. At lines 10–12, FINDINTERACTIONS determines that the augmented execution histories of $\{$`ch1`$\}$ on P' and $P'\backslash_{\{ch1\}}$ for that input have the same sequence of statement occurrences, but the value of `r` at each statement occurrence is 1 greater in $P'\backslash_{\{ch1\}}$ than in P'. This difference is the semantic effect of $\{$`ch1`$\}$ on P'. Also, lines 13–15 find that the semantic effect of $\{$`ch1`$\}$ on $P'\backslash_{\{ch2\}}$ corresponds to differences in the values of `r` in all occurrences of statement 6: at 6^1, 6^2, and 6^3, `r` is 1 less in $P'\backslash_{\{ch1,ch2\}}$ than in $P'\backslash_{\{ch2\}}$ and, at 6^4, `r` is 2 less. Line 16 finds that these effects differ, which implies that $\{$`ch1`$\}$ and $\{$`ch2`$\}$ alter the effects of each other, so line 17 adds this pair to the result.

For a given list of change sets, the worst-case space and time complexity of FINDINTERACTIONS is linear in the product of the size of the set of variables V in P and the length of the executions of the different variants of P on input I. This is the cost of finding the augmented execution histories and their differences. During execution, updating the dynamic forward slice of a change at a statement occurrence requires, at most, one check for each "live" definition (up to $|V|$) and each "live" conditional statement (always one).

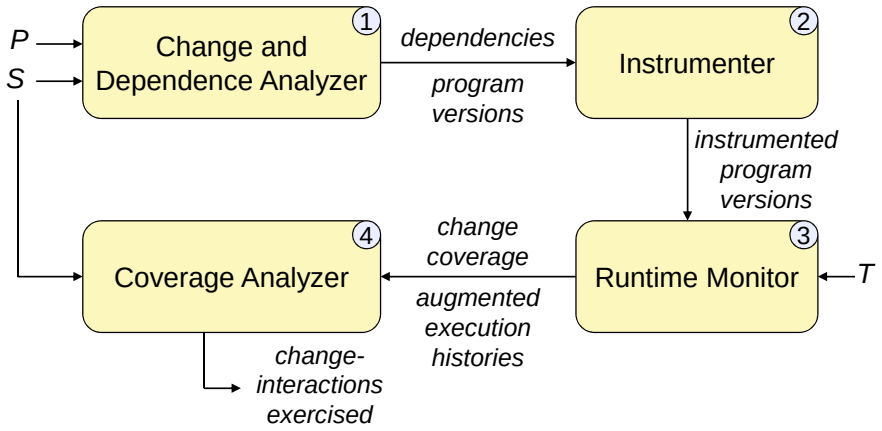


Figure 22: Toolset and process describing the implementation of SCHID.

When the dynamic slices of multiple occurrences of a change converge at a statement occurrence, only one dynamic slice needs to be tracked for that change after that point. Thus, the number of occurrences of a change does not impose an extra factor on the algorithm's complexity. An augmented execution history also includes the value of each modified variable at each point of a dynamic slice, but this is only a constant factor of the total cost. Finally, the cost of detecting dynamic data dependencies is almost constant if hashing of live variables is used, so the cost of FINDINTERACTIONS is, in practice, linear in the length of the execution.

7.2.4 Implementation of SCHID

To support the study of change-interaction detection techniques, we implemented SCHID using DUA-FORENSICS [93], the dependence analysis toolset described and used in all other studies in this dissertation.

Figure 22 shows the process that implements the technique as a data-flow diagram where boxes are processing components and edges represent the flow of elements between components. The components are numbered 1–4 according to their processing order:

1. The *Change and Dependence Analyzer* inputs program P and the collection of change sets S and performs three actions:

(a) applies the change sets in S to P to obtain program version P' and, for all C_i and C_j ($i < j$) in S, versions $P'\backslash_{C_i}$ and $P'\backslash_{C_i,C_j}$,

(b) identifies the direct and transitive data- and control-dependencies from the changed statements on each version, and

(c) discards as "non-interacting" all change-set pairs whose static forward slices do not intersect in the corresponding versions.

2. The *Instrumenter* takes the program versions and dependencies produced by Component 1 and instruments these versions to generate the change-coverage and execution-history information needed.

3. The *Runtime Monitor* runs test suite T on the instrumented program versions provided by Component 2, collects change-coverage and execution-history information from the executions, and outputs a coverage report for each test case and program version.

4. The *Coverage Analyzer* takes the change-coverage and execution-history reports from Component 3 and determines, for each test case, which change sets from S were covered, had their dynamic slices intersected, or interacted according to SCHID.

7.2.5 Study of Change-interaction Detection Techniques

This section presents our study of change-interaction detection whose goal was to assess and compare the accuracy of SCHID and other approaches for this task. For this study, we used the implementation described in Section 7.2.4 and compared SCHID with three alternative change-interaction detection techniques when applied to different sets of changes in Java software. The first two techniques are representative of existing change-analysis techniques: static-slice intersection (e.g., [19, 56]) and multiple-change coverage (e.g., [83, 113]). For the third technique, we used dynamic slicing [1, 64] in its finest-grained formulation to detect change interactions by intersecting dynamic forward slices

Table 23: Subjects used for the study of SCHID.

subject	description	LOC	test inputs	changes	change pairs
Tot_info	information measure	283	1052	4	6
Schedule1	priority scheduler	290	2650	4	6
NanoXML v1	XML parser	3497	214	4	6
NanoXML v5		4782	216	4	6

of changes. The static-slice approach determines which changes may interact in the modified program P', for any test case. The coverage approach reports potential interactions between pairs of changes covered by a test case on P'. The dynamic-slice approach reports potential interactions due to intersecting dynamic slices of changes for a test case on P'.

7.2.5.1 Empirical Setup

For this study, we used four subjects, including their respective test suites. The first two subjects, Tot_info and Schedule1, are part of the Siemens suite that we translated from C to Java. The remaining subjects are two versions of NanoXML available at the SIR repository [36, 40]. For each subject, we used the standard test suite and the first four seeded changes provided with the subject. Table 23 gives details about these subjects. In the table, for each subject, the second column gives a description, the third column gives the number of lines of code, the fourth column gives the number of test cases in the test suite, and the last two columns show, respectively, the number of changes considered and the corresponding number of change pairs for which potential interactions were analyzed. (A total of 24 pairs of changes were considered for interaction.)

To compare the accuracy of the interaction-detection techniques, we measured their precision—the fraction of reported interactions that are real (true positives)—and safety—the fraction of real interactions missed (false negatives). The three approaches that we compared with SCHID operate only on P'. Thus, when these approaches report no interaction between changes C_1 and C_2 on P', they may fail to recognize certain interactions

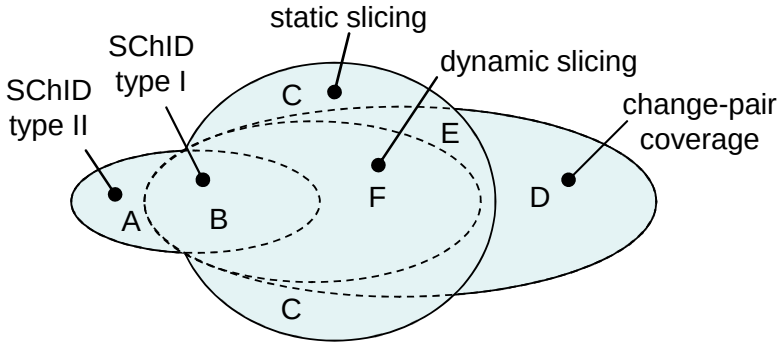


Figure 23: Relationships among techniques for change-interaction detection.

found by SCHID. Specifically, if C_1 has no effect on P' but has an effect on $P'\backslash_{C_2}$, then C_1 and C_2 interact, but the three approaches fail to identify this interaction—this is a false negative in their reports. This kind of interaction occurs, for example, when the effect of C_2 on P' prevents C_1 from executing on P'.

To facilitate the discussion, it is useful to classify the interactions found by SCHID into *Type I* for interactions where both changes have an effect on P' and *Type II* for all other interactions. The set diagram of Figure 23 shows the relationships among all techniques studied: Type-II interactions (A) are false negatives for all techniques but SCHID, while Type-I interactions (B) are the true positives for those techniques. The diagram also shows that the false positives reported by dynamic slicing (F) are also reported by static slicing and change coverage. These two techniques are necessary conditions for dynamic-slice intersection, but not sufficient (E), because they do not guarantee that after the changes execute their dynamic slices intersect. Also, the static slices of two changes might intersect, but the changes might not be covered (C). Similarly, the static slices of two changes might not intersect but the changes might be covered (D).

In the study, for each subject P, we analyzed the respective four changes as single-change sets. Thus, for P, we constructed versions P', $P'\backslash_{\{C_i\}}$ (for $i \in [1,4]$), and $P'\backslash_{\{C_i,C_j\}}$ (for $i, j \in [1,4]$ and $i<j$), and run the corresponding test suite on all these versions.

Table 24: Change-interaction detection results for Tot_info (1052 test inputs).

change pair	change-pair coverage		dynamic-slice intersection		SCHID	
	Type I	Type II	Type I	Type II	Type I	Type II
C_1, C_2	2	0	2	0	2	0
C_1, C_3	2	0	0	0	0	0
C_1, C_4	1	0	1	0	1	0
C_2, C_3	272	0	7	0	0	0
C_2, C_4	28	0	28	0	28	0
C_3, C_4	28	0	3	0	0	0
avg. false pos.	50.3	0.0	1.7	0.0		
avg. false neg.	0.0	0.0	0.0	0.0		

Table 25: Change-interaction detection results for Schedule1 (2650 test inputs).

change pair	change-pair coverage		dynamic-slice intersection		SCHID	
	Type I	Type II	Type I	Type II	Type I	Type II
C_1, C_2	852	0	640	0	1	0
C_1, C_3	773	0	752	0	0	0
C_1, C_4	773	0	752	0	0	1
C_2, C_3	710	0	710	0	21	4
C_2, C_4	710	0	710	0	323	4
C_3, C_4	866	0	866	0	431	389
avg. false pos.	651.3	0.0	609.0	0.0		
avg. false neg.	0.0	66.3	0.0	66.3		

7.2.5.2 Results and Analysis

Tables 24–27 show the results, for each subject and all change pairs in that subject, of all techniques except static-slice intersection. Static slices intersected on all modified subjects, so they are not shown in the table. In each table, the first column lists all pairs of changes, the second column shows the number of test cases that covered each pair, the third column shows the number of test cases for which the dynamic slices of both changes intersected, and the last column shows the number of test cases for which the changes interacted according to SCHID. The results for each technique are divided into Type-I and Type-II

Table 26: Change-interaction detection results for NanoXML *v1* (214 test inputs).

change pair	**change-pair coverage**		**dynamic-slice intersection**		**SCHID**	
	Type I	Type II	Type I	Type II	Type I	Type II
C_1, C_2	51	0	51	0	0	67
C_1, C_3	50	0	50	0	0	67
C_1, C_4	0	0	0	0	0	0
C_2, C_3	35	0	35	0	0	15
C_2, C_4	0	0	0	0	0	66
C_3, C_4	0	0	0	0	0	50
avg. false pos.	22.7	0.0	22.7	0.0		
avg. false neg.	0.0	44.2	0.0	44.2		

Table 27: Change-interaction detection results for NanoXML *v5* (216 test inputs).

change pair	**change-pair coverage**		**dynamic-slice intersection**		**SCHID**	
	Type I	Type II	Type I	Type II	Type I	Type II
C_1, C_2	0	0	0	0	0	0
C_1, C_3	0	0	0	0	0	0
C_1, C_4	0	0	0	0	0	0
C_2, C_3	0	0	0	0	0	0
C_2, C_4	77	0	0	0	0	0
C_3, C_4	0	0	0	0	0	0
avg. false pos.	12.8	0.0	12.8	0.0		
avg. false neg.	0.0	0.0	0.0	0.0		

interactions found by that technique. Also, in each table, two rows at the end show the average number, for the six change pairs, of false positives and false negatives for change-pair coverage and dynamic-slice intersection (SCHID has no false positives or false negatives). Recall that, unlike SCHID, these two techniques miss all Type-II interactions.

For example, for pair (C_1, C_2) in Schedule1 with all changes applied, column *change-pair coverage* shows that both changes were covered in 852 of the 2650 test cases. For that pair, column *dynamic-slice intersection* shows that in 640 of those test cases the dynamic slices of the two changes intersected. Also for that pair, column SCHID reports that

only one test case exercised a Type-I interaction, and no test case caused a Type-II interaction. Thus, change-pair coverage and dynamic-slice intersection caused 851 and 639 false positives, respectively, but no false negatives.

Although static slicing determined that all change pairs may interact, the Type-I results for SCHID show that, for many pairs, no test case in the test suite exercised Type-I interactions. For example, for Tot_info and its test suite, three change pairs did not cause Type-I interactions, and for NanoXML *v5*, no interaction was exercised by the test suite. Hence, static slicing is too imprecise for estimating Type-I interactions in these subjects, at least for the available test suites. Note, however, that the test suites for Tot_info and Schedule1 are quite large, but only a small fraction of those test cases cause actual interactions. Thus, it may be difficult to find additional test cases that cause the remaining interactions to occur, if possible at all.

The change-coverage approach performed better than static slicing by detecting that, for all change pairs, only a fraction of the test cases covered both changes. For example, for each pair in Schedule1, this approach found that no more than a third of all test cases covered the two changes on each pair. However, these numbers are still too imprecise to use to estimate Type-I interactions when compared with the actual results found by SCHID. In the best case, for this subject (i.e., C_3 and C_4), the number of Type-I interactions reported by change coverage was about twice the number of real interactions of this type; in the worst case (e.g., C_1 and C_3), all of the reported interactions were false positives.

Surprisingly, the method of intersecting dynamic slices was more precise than change-pair coverage in only a few cases. For example, for pairs (C_2,C_3) and (C_3,C_4) in Tot_info, this technique removed a large proportion of false positives reported by change coverage. Yet, dynamic-slice intersection was still too imprecise in identifying Type-I interactions as detected by SCHID. For example, for four pairs in NanoXML *v1* and *v5*, this approach detected 35–77 potential interactions, all of which were false positives.

Unlike the test suites for Tot_info and Schedule1, the test suites for both versions of

NanoXML did not exercise any Type-I interactions. There are three reasons that may explain this phenomenon: (1) the small sizes of Tot_info and Schedule1 make their changes "closer" (in a control- and data-dependence sense) to each other and, thus, make interactions more likely to occur, (2) the large sizes of the test suites for Tot_info and Schedule1 increase the likelihood of exercising interactions, and (3) some of the changes provided with NanoXML cause exceptions to be raised, preventing the execution from continuing normally and reaching other changes.

Type-II interactions also occurred occasionally, for a total of nine change pairs in Tot_info and NanoXML *v1*. Although in Tot_info the occurrence of Type-II interactions seems correlated with the occurrence of Type-I interactions, this was not the case in version *v1* of NanoXML, where many Type-II interactions but no Type-I interactions were exercised. The presence of only Type-II interactions in NanoXML *v1* is explained by the nature of the changes in this subject, which tend to prevent the execution from reaching other changes.

In conclusion, the results suggest that covering pairs of changes or having static or dynamic slices intersect is, often, too imprecise and also unsafe for establishing with any degree of confidence which interactions among changes actually occur. Hence, of all techniques evaluated, and at least for the subjects, changes, and test-suites considered in the study, only SCHID—a precise implementation of our formal model of change interactions—provides useful results.

7.2.5.3 Case Study in NanoXML

A possible explanation for the failure of the test suites in this study to exercise interactions for many pairs of changes is that the test suites provided with these subjects are inadequate for this task. To understand why so few test cases caused actual change interactions, we manually inspected NanoXML *v5* and its test suite of 216 test cases. The static-slice intersection approach in this study indicated that all six pairs of changes might interact, but no

test case exercised these pairs in practice. Thus, we carefully examined the changes and test suite for NanoXML *v5* and found that the pair (C_2, C_4) could actually interact. Then, we created a new test case that, as confirmed by using our toolset, causes a Type-I interaction of this pair. The failure of the original 216 test cases in making C_2 and C_4 interact is explained by the subtle conditions that make these particular changes interact.

For the remaining five change pairs in NanoXML *v5*, we found that they cannot interact: C_1 causes an error during DTD parsing that prevents any execution from reaching other changes, whereas C_3 modifies an unhandled exception object after which no other change can execute. For these reasons, C_1 and C_3 participate in behaviors that are mutually exclusive with each other and with C_2 and C_4, even though there are (infeasible) dependence paths between all of them found by static slicing.

7.2.5.4 Threats to Validity

The main internal threat to the validity of this study is the possibility of implementation errors in our change and dependence analysis, monitoring, and reporting. This threat is reduced by the maturity of DUA-FORENSICS, which has been in development for years.

The main external threat to this study is the limited variety, size, and nature of the subjects and changes used in the study. These subjects, however, have been used extensively for software-engineering studies; NanoXML is also a real-world program. Thus, although the conclusions for this study cannot be generalized to other types of subjects, the study highlights the interactions that can occur, the inaccuracy of existing approaches with respect to SCHID, and the inadequacy of existing test suites in exercising interactions.

7.3 Related Work

Although many techniques exist for regression testing and test-suite augmentation, most of these techniques address single changes only. The techniques by Rothermel and Harrold [87] and Binkley [17] identify the entities (e.g., statements, branches, and du-pairs)

affected by a change, but do not analyze relationships among changes. Our own change-effects analysis and testing techniques [92, 94, 96] presented in Chapters 5 and 6 also target individual changes. In contrast, in Section 7.1 of this chapter, we described how developers can address multiple changes by adapting the testing of individual changes when interactions with other changes may occur. This adaptation can work even with a very conservative approach, as shown in the case study in that section.

A more accurate computation of change interference, however, is desirable to ensure that state requirements are not invalidated unnecessarily. The SCHID technique presented in Section 7.2 addresses the problem of detecting change interference by building on the formal model of semantic dependence among changes from Section 3.1.4. Thus, the work presented in this chapter complements our own published techniques [92, 94, 96] by providing the precise change-interaction detection needed by them.

In the area of change-interference analysis for safe merging of program versions, there are static and dynamic techniques related to SCHID. On the static side, Horwitz and colleagues [19] [56] present techniques that use static slicing to determine which changes cannot interfere with each other and can thus be safely merged, although they do not present empirical studies. Like SCHID, their techniques work on code-level changes. However, our study in Section 7.2.5 shows that static-slice intersection can be too imprecise to be useful. On the dynamic side, Wloka and colleagues [113] present a technique that uses the outcomes of test cases in a test suite to decide which changes can be safely committed to a repository. This technique is limited in that it operates on coarse-grained changes (e.g., method modifications) and only identifies compilation dependencies among changes. SCHID, in contrast, can be used for program merging with accuracy thanks to a precise definition of semantic dependence to detect actual interactions among changes at runtime.

Another related area is change-impact analysis (e.g., [5, 83]), whose goal is to find which program elements are affected by changes. To the best of our knowledge, no technique in this area analyzes the impact that changes have on other changes. A technique

by Ren and colleagues [83], called Chianti, addresses multiple changes but only identifies structural (compilation) dependencies among coarse-grained changes. SCHID, in contrast, works on statement-level changes and identifies the precise semantic dependencies observed among changes.

Finally, a number of techniques address the problem of interaction testing. These techniques aim at effectively testing software that has an exponential number of configurations (e.g., [29]). Although interaction-testing techniques are loosely related to our work because they target the interaction of various software entities, such as parameters, features, and components, none of these techniques have been applied to changes.

CHAPTER VIII

CONCLUSION AND FUTURE WORK

Software is changed constantly throughout its life cycle. These changes to software pose serious challenges to developers and testers who must ensure that software remains correct after modifications by analyzing and testing those modifications. Existing research on change analysis and testing has focused either on program entities potentially affected by the changes—without any empirical validation—or on making regression testing more efficient. However, when software is changed, new behaviors are introduced and existing behaviors are modified in ways not anticipated when the original test suite was created. Therefore, automatic methods are needed that analyze the effects of changes effectively to support software-engineering tasks such as test-suite augmentation.

This dissertation addresses the challenge of making the analysis of the effects of changes effective and practical by presenting a set of principled and cost-conscious approaches and techniques. At a fundamental level, this dissertation provides theoretical foundations of the effects of changes by defining exactly what a change is, what the effects of a change are on the behavior of a program, and how changes affect each other. These foundations are the basis for a principled and accurate computational approach to identify these effects. At the fundamental level, this dissertation also identifies a costly core aspect of the computational approach—symbolic execution of multiple paths—and introduces a new technique that improves substantially the scalability of this form of symbolic execution.

At a practical level, this dissertation presents a number of techniques that provide usable approximations to the fundamental approach for computing the effects of changes. These techniques support the analysis and testing of changes by automatically identifying the set of potential effects of a change within distance limits, providing the means to test those

effects on demand, and detecting the interactions that changes exhibit at runtime. The potential effects identified by these approximations can serve as change-testing requirements, which we call "propagation-based" because they take into account the transfer conditions on the program state. The studies presented in this dissertation indicate that propagation-based testing requirements are significantly more effective for detecting behavioral differences than the testing approaches presented in the literature, which are based only on the coverage of affected program entities. These studies also show that propagation-based testing requirements can be particularly cost-effective by identifying and satisfying them on demand. In addition, two case studies in this dissertation illustrate how developers can satisfy propagation-based testing requirements in practice.

Another practical problem addressed in this dissertation is that developers often make multiple changes to software and thus the interplay among changes must be accounted for. For instance, developers need to know whether changes that have different purposes interfere with each other unexpectedly and whether changes that should work together are interacting as expected. Although the problem of determining whether changes affect each other is undecidable, this dissertation presents and studies a new technique, based on the same foundations of change-effects analysis, for precisely detecting whether changes interact at runtime. The studies indicate that this technique, despite its cost, has an unprecedented level of accuracy for detecting such interactions and that existing dynamic analyses, regarded as state-of-the-art, are actually quite insufficient for this task. Thus, new and better approximations are needed for change-interaction detection.

8.1 Merit

The research presented in this dissertation has several merits that enhance the understanding and technical development of the analysis of changes for researchers and practitioners. These merits are:

1. A foundational treatment of the nature of changes and their effects on the behavior

of programs that establishes, to an unprecedented level of detail, what these effects are and how they are obtained programmatically. These foundational concepts can serve as the basis for a variety of theoretical and practical studies and techniques in regression testing, software evolution, and other change-related disciplines.

2. A new class of precise analyses of the effects of changes that we call *change-effects analysis* and is based on the aforementioned foundations. This class of analyses enables the development of new and more cost-effective techniques for change-related tasks, including the techniques presented in this dissertation:

   (a) A new test-suite augmentation technique that provides developers with a set of *propagation-based* testing requirements for a modified program that represents precisely the space of all possibly affected behaviors, within some limits. The studies in this dissertation suggest that using these testing requirements is more effective than using the *coverage-based* techniques from the literature.

   (b) A demand-driven approach for this new test-suite augmentation technique that provides developers with on-demand information about the satisfaction of testing requirements to allow for greater distances to be reached. The studies in this dissertation of this approach confirm the superiority that developers can expect from using propagation-based requirements.

   (c) A new technique, SCHID, for accurately detecting whether changes interact at runtime, which lets developers determine whether it is safe, with respect to existing test cases, to merge changes that should not affect each other and whether changes interact as they should during testing. The study of this technique warns developers that analyses such as dynamic slicing are not precise for change interaction and that precise techniques such as SCHID are needed.

3. A new supporting approach for more scalable multiple-path symbolic execution that gives developers more information per unit of time about the effects of changes than

using traditional, path-by-path symbolic execution. This new analysis technique also shows great promise for significant improvements in other applications of symbolic execution, such as test-data generation and invariant discovery.

8.2 Future Work

The foundations of change-effects analysis presented in this dissertation have broader implications for the future of research on changes and their effects than just the techniques presented in this dissertation. These foundations can be investigated further as the basis for theoretical and practical work:

- At the theoretical level, the foundations can provide new insights on how changes propagate through software components, including not only code but also data (e.g., databases, files). New theories can be developed as an extension of these foundations that better suit changes in concurrent and data-driven software. (Our foundations currently support virtually all types of software, as long as it is possible to capture all sources of uncertainty as inputs.) Also, we envision the opportunity of using our foundations to define a framework for the specification of formal semantics of programming changes, much like existing formal semantics of programming languages.

- At the practical level, approximate models of the effects of changes can be created that, while not necessarily applicable directly, provide solid foundations for tasks such as change-impact analysis for which current research is inadequate in light of the results obtained in this dissertation. Specifically, our results highlight the importance of including state-difference information as a complement of coverage information. Thus, state-difference information needs to be researched further.

The techniques presented in this dissertation for computing testing requirements are by no means the only ways in which the effects of changes can be approximated. For example, various heuristics can be used to identify and prioritize clusters of the effects of changes to

focus the testing efforts on the areas that are most likely or most strongly impacted by those changes. In that line, we have already initiated research on a technique called *probabilistic forward slicing* [95] with promising initial results. The technique statically estimates the chances that any given point in the program is impacted by a change and the strength of that impact. Because the testing requirements that matter the most for testing are those not yet satisfied, it is important for developers to determine which not-yet-observed behaviors of a change are the most affected and likely to satisfy next during testing.

On the experimental side, it is necessary to investigate the ability of change-testing techniques to not only reveal the observable differences in the behavior of software caused by changes but also to detect any errors introduced by those changes. Finding differences caused by changes is already a difficult task, as shown in this dissertation, and therefore cost-effective techniques such as those developed in this work are needed to find as many differences as possible as a requisite for detecting change-related errors. However, more experiments and case studies conducted by developers other than the author should be conducted in the future to assess the error-detection abilities of these techniques.

Another important avenue of future research is the automation of test-data creation for satisfying propagation-based testing requirements for changes. Despite recent advances in test-data generation, the problem of *targeted* test-data generation (i.e., generation of inputs for reaching a particular point in the program) is far from solved. Moreover, the propagation-based testing of changes requires not only the execution of the change, but also the satisfaction of infection and propagation conditions. Thus, it is unclear whether test-data generation for change effects can be automated to a satisfactory level, although the current immaturity of this field suggests that much greater automation can be achieved. In all, creating test data for change testing is the natural next step in this line of research to improve upon the purely-manual approach used in our case studies. The large amount of developer involvement currently limits our ability to study change-testing techniques.

One promising direction for improving test-data generation for change-testing and other goals is the use of SPD. SPD is not only capable of describing multiple related paths succinctly, but it can also use abstractions to further reduce the burden on the constraint solver and perhaps open the door for integrating other mechanisms, such as genetic algorithms, to satisfy abstract conditions. But test-data generation is not the only potential application of SPD. Other important applications of multiple-path symbolic execution—and thus of SPD—that could be investigated next include static specification mining, static invariant detection, verification, and statistical fault localization.

Finally, improvements can be made for the dynamic change-interaction detection technique presented in Chapter 7 to make its implementation practical for executions longer than those studied. Because the amount of dependence and state-difference information collected at runtime and the cost of analyzing that information can be very large, we should consider two lines of improvement: making the implementation more efficient and developing less accurate but more scalable approximations. Towards these goals, we are investigating static analyses that can considerably reduce the amount of instrumentation needed and keep under control the runtime overhead and the amount of information collected. Good candidates for achieving these goals are techniques that infer precisely or approximately the coverage of expensive-to-monitor entities using the coverage of cheaper-to-monitor entities, such as techniques we have already developed [93].

REFERENCES

[1] AGRAWAL, H. and HORGAN, J. R., "Dynamic program slicing," in *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 246–256, 1990.

[2] AHO, A. V., SETHI, R., and ULLMAN, J. D., *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.

[3] ANAND, S., GODEFROID, P., and TILLMANN, N., "Demand-driven compositional symbolic execution," in *Proceedings of International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 367–381, Mar. 2008.

[4] APACHE SOFTWARE FOUNDATION, THE. Apache Ant version 1.8.2. `http://ant.apache.org/`. Sept., 2010.

[5] APIWATTANAPONG, T., ORSO, A., and HARROLD, M. J., "Efficient and precise dynamic impact analysis using execute-after sequences," in *Proceedings of IEEE/ACM International Conference on Software Engineering*, May 2005.

[6] APIWATTANAPONG, T., SANTELICES, R., CHITTIMALLI, P. K., ORSO, A., and HARROLD, M. J., "Matrix: Maintenance-oriented testing requirement identifier and examiner," in *Proceedings of Testing and Academic Industrial Conference Practice and Research Techniques*, pp. 137–146, Aug. 2006.

[7] ARNOLD, R. S. and BOHNER, S. A., "Impact analysis - towards a framework for comparison," in *Proceedings of IEEE Conference on Software Maintenance*, pp. 292–301, Sept. 1993.

[8] BABIC, D. and HU, A. J., "Calysto: Scalable and precise extended static checking," in *Proceedings of IEEE/ACM International Conference on Software Engineering*, pp. 211–220, May 2008.

[9] BALL, T., MAJUMDAR, R., MILLSTEIN, T., and RAJAMANI, S. K., "Automatic predicate abstraction of C programs," in *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2001.

[10] BARNETT, M. and LEINO, K. R. M., "Weakest-precondition of unstructured programs," in *Proceedings of Workshop on Program Analysis for Software Tools and Engineering*, pp. 82–87, Sept. 2005.

[11] BARRETT, C. and TINELLI, C., "CVC3," in *Proceedings of International Conference on Computer Aided Verification*, vol. 4590 of *Lecture Notes in Computer Science*, pp. 298–302, Springer-Verlag, July 2007.

[12] BEIZER, B., *Software Testing Techniques*. International Thomson Computer Press; 2nd edition, June 1990.

[13] BENNETT, K. H. and RAJLICH, V. T., "Software maintenance and evolution: a roadmap," in *Proceedings of Conference on The Future of Software Engineering*, ICSE 2000, pp. 73–87, May 2000.

[14] BEYER, D., HENZINGER, T. A., JHALA, R., and MAJUMDAR, R., "The software model checker Blast: Applications to software engineering," *International Journal of Software Tools for Technology Transfer*, vol. 9, pp. 505–525, Oct. 2007.

[15] BEYER, D., HENZINGER, T. A., and THÉODULOZ, G., "Configurable software verification: Concretizing the convergence of model checking and program analysis," in *Proceedings of International Conference on Computer Aided Verification*, pp. 504–518, July 2007.

[16] BINKLEY, D., DANICIC, S., GYIMOTHY, T., HARMAN, M., KISS, A., and OUARBYA, L., "Formalizing executable dynamic and forward slicing," in *Proceedings of Source Code Analysis and Manipulation, Fourth IEEE International Workshop*, pp. 43–52, Sept. 2004.

[17] BINKLEY, D., "Semantics guided regression test cost reduction," *IEEE Transactions on Software Engineering*, 23(8):498–516, Aug. 1997.

[18] BINKLEY, D., GOLD, N., and HARMAN, M., "An empirical study of static program slice size," *ACM Transactions Software Engineering and Methodology*, vol. 16, Apr. 2007.

[19] BINKLEY, D., HORWITZ, S., and REPS, T., "Program integration for languages with procedure calls," *ACM Transactions on Software Engineering and Methodology*, vol. 4, pp. 3–35, Jan. 1995.

[20] BOHNER, S. A. and ARNOLD, R. S., *An introduction to software change impact analysis*. In Software Change Impact Analysis, Bohner & Arnold, Eds. IEEE Computer Society Press, pp. 1–26, June 1996.

[21] BRIAND, L. C., WUEST, J., and LOUNIS, H., "Using coupling measurement for impact analysis in object-oriented systems," in *Proceedings of IEEE International Conference on Software Maintenance*, pp. 475–482, Aug. 1999.

[22] BRYANT, R. E., LAHIRI, S. K., and SESHIA, S. A., "Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions," in *Proceedings of International Conference on Computer Aided Verification*, pp. 78–92, July 2002.

[23] CHAMBERS, C., MOCK, M., ATKINSON, D. C., and EGGERS, S. J., "Program slicing with dynamic points-to sets," *IEEE Transactions Software Engineering*, vol. 31, no. 8, pp. 657–678, 2005.

[24] CHANDRA, S., FINK, S. J., and SRIDHARAN, M., "Snugglebug: A powerful approach to weakest preconditions," in *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 363–374, June 2009.

[25] CHEN, Y. F., ROSENBLUM, D. S., and VO, K. P., "Testtube: A system for selective regression testing," in *Proceedings of IEEE/ACM International Conference on Software Engineering*, pp. 211–222, May 1994.

[26] CLARKE, E. M., GRUMBERG, O., JHA, S., LU, Y., and VEITH, H., "Counterexample-guided abstraction refinement for symbolic model checking," *Journal of the ACM*, vol. 50, no. 5, pp. 752–794, 2003.

[27] CLARKE, L. A., "A system to generate test data and symbolically execute programs," *IEEE Transactions on Software Engineering*, vol. 2, no. 3, pp. 215–222, 1976.

[28] CLARKE, L. A. and RICHARDSON, D. J., "Applications of symbolic evaluation.," *Journal of Systems and Software*, vol. 5, pp. 15–35, Feb. 1985.

[29] COHEN, M. B., GIBBONS, P. B., MUGRIDGE, W. B., and COLBOURN, C. J., "Constructing test suites for interaction testing," in *Proceedings of IEEE/ACM International Conference on Software Engineering*, pp. 38–48, May 2003.

[30] COLLINS-SUSSMAN, B., FITZPATRICK, B. W., and PILATO, C. M., *Version Control with Subversion*. The Apache Foundation and O'Reilly Media, Oct. 2011. `http://svnbook.org`.

[31] CSALLNER, C., TILLMANN, N., and SMARAGDAKIS, Y., "DySy: Dynamic symbolic execution for invariant inference," in *Proceedings of IEEE/ACM International Conference on Software Engineering*, pp. 281–290, May 2008.

[32] CURRIE, D., FENG, X., FUJITA, M., HU, A. J., KWAN, M., and RAJAN, S., "Embedded software verification using symbolic execution and uninterpreted functions," *International Journal of Parallel Programming*, vol. 34, no. 1, pp. 61–91, 2006.

[33] CYTRON, R., FERRANTE, J., ROSEN, B. K., WEGMAN, M. N., and ZADECK, F. K., "Efficiently computing static single assignment form and the control dependence graph," *ACM Transactions Programming Languages and Systems*, vol. 13, no. 4, pp. 451–490, 1991.

[34] DEMILLO, R. A., LIPTON, R. J., and SAYWARD, F. G., "Hints on test data selection: Help for the practicing programmer," *Computer*, vol. 11, pp. 34–41, Apr. 1978.

[35] DIJKSTRA, E. W., *A Discipline of Programming*. Prentice Hall, Englewood Cliffs, NJ, Oct. 1976.

[36] DO, H., ELBAUM, S., and ROTHERMEL, G., "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact," *Empirical Software Engineering*, vol. 10, pp. 405–435, Oct. 2005.

[37] DOLBY, J., VAZIRI, M., and TIP, F., "Finding bugs efficiently with a SAT solver," in *Proceedings of ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pp. 195–204, Sept. 2007.

[38] ENGLER, D. and DUNBAR, D., "Under-constrained execution: making automatic code destruction easy and scalable," in *Proceedings of International Symposium on Software Testing and Analysis*, pp. 1–4, July 2007.

[39] ERNST, M. D., COCKRELL, J., GRISWOLD, W. G., and NOTKIN, D., "Dynamically discovering likely program invariants to support program evolution," *IEEE Transactions on Software Engineering*, vol. 27, no. 2, pp. 99–123, 2001.

[40] ESQUARED LAB. Software-artifact Infrastructure Repository., Apr. 2012. `http://sir.unl.edu`. University of Nebraska, Lincoln.

[41] FERRANTE, J., OTTENSTEIN, K., and WARREN, J., "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems*, vol. 9, no. 3, pp. 319–349, 1987.

[42] FISCHER, J., JHALA, R., and MAJUMDAR, R., "Joining dataflow with predicates," in *Proceedings of ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pp. 227–236, Sept. 2005.

[43] FLANAGAN, C. and SAXE, J. B., "Avoiding exponential explosion: generating compact verification conditions," in *Proceedings of ACM Symposium on Principles of Programming Languages*, pp. 193–205, Jan. 2001.

[44] FOWLER, M., BECK, K., BRANT, J., OPDYKE, W., and ROBERTS, D., *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, July 1999.

[45] FRANKL, P. and WEYUKER, E. J., "An applicable family of data flow criteria," *IEEE Transactions on Software Engineering*, vol. 14, no. 10, pp. 1483–1498, 1988.

[46] GALLAGHER, K. B. and LYLE, J. R., "Using program slicing in software maintenance," *IEEE Transactions Software Engineering*, vol. 17, no. 8, pp. 751–761, 1991.

[47] GODEFROID, P., KLARLUND, N., and SEN, K., "DART: Directed automated random testing," in *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2005.

[48] GULAVANI, B. S., HENZINGER, T. A., KANNAN, Y., NORI, A. V., and RAJAMANI, S. K., "SYNERGY: A new algorithm for property checking," in *Proceedings of ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pp. 117–127, Nov. 2006.

[49] GUPTA, R., HARROLD, M. J., and SOFFA, M. L., "An approach to regression testing using slicing," in *Proceedings of IEEE Conference on Software Maintenance*, Nov. 1992.

[50] GUPTA, R., HARROLD, M., and SOFFA, M., "Program slicing-based regression testing techniques," *Journal of Software Testing, Verification, and Reliability*, 6(2):83–111, June 1996.

[51] HAMLET, R. G., "Testing programs with the aid of a compiler," *IEEE Transactions Software Engineering*, vol. 3, pp. 279–290, July 1977.

[52] HARROLD, M. J., GUPTA, R., and SOFFA, M. L., "A methodology for controlling the size of a test suite," *ACM Transactions on Software Engineering and Methodology*, 2(3):270–285, July 1993.

[53] HARROLD, M. J., JONES, J. A., LI, T., LIANG, D., ORSO, A., PENNINGS, M., SINHA, S., SPOON, S. A., and GUJARATHI, A., "Regression test selection for java software," in *Proceedings of ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pp. 312–326, Nov. 2001.

[54] HARROLD, M., ROTHERMEL, G., SAYRE, K., WU, R., and YI, L., "An empirical investigation of the relationship between spectra differences and regression faults," *Journal of Software Testing, Verification and Reliability*, vol. 10, pp. 171–194, Sept. 2000.

[55] HORWITZ, S., REPS, T., and BINKLEY, D., "Interprocedural slicing using dependence graphs," in *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 35–46, June 1988.

[56] HORWITZ, S., PRINS, J., and REPS, T., "Integrating non-interfering versions of programs," *ACM Transactions on Programming Languages and Systems*, vol. 11, pp. 345–387, July 1989.

[57] HORWITZ, S., REPS, T., and BINKLEY, D., "Interprocedural slicing using dependence graphs," *ACM Transactions on Programming Languages and Systems*, 12(1):26-60, Jan. 1990.

[58] HOWDEN, W. E., *Tutorial: Software Testing and Validation Techniques*. IEEE Computer Society, 1978.

[59] HUANG, J. C., "An approach to program testing," *ACM Computing Surveys*, vol. 7, no. 3, pp. 113–128, 1975.

[60] HUTCHINS, M., FOSTER, H., GORADIA, T., and OSTRAND, T., "Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria," in *Proceedings of IEEE/ACM International Conference on Software Engineering*, pp. 191–200, May 1994.

[61] JHALA, R. and MAJUMDAR, R., "Path slicing," in *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 38–47, June 2005.

[62] JIN, W., ORSO, A., and XIE, T., "Automated behavioral regression testing," in *Proceedings of IEEE International Conference on Software Testing, Verification and Validation*, pp. 137–146, Apr. 2010.

[63] KING, J. C., "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, pp. 385–394, July 1976.

[64] KOREL, B. and LASKI, J., "Dynamic program slicing," *Information Processing Letters*, vol. 29, no. 3, pp. 155–163, 1988.

[65] KOREL, B. and AL-YAMI, A. M., "Automated regression test generation," in *Proceedings of International Symposium on Software Testing and Analysis*, pp. 143–152, Mar. 1998.

[66] KUNG, D., GAO, J., HSIA, P., TOYASHIMA, Y., and CHEN, C., "Firewall regression testing and software maintenance of object-oriented systems," *Journal of Object-Oriented Programming*, 1994.

[67] LASKI, J. W. and KOREL, B., "A data flow oriented program testing strategy," *IEEE Transactions on Software Engineering*, vol. 9, no. 3, pp. 347–354, 1983.

[68] LAW, J. and ROTHERMEL, G., "Whole program path-based dynamic impact analysis," in *Proceedings of IEEE/ACM International Conference on Software Engineering*, pp. 308–318, May 2003.

[69] LEUNG, H. K. N. and WHITE, L. J., "Insights into regression testing," in *Proceedings IEEE of Conference on Software Maintenance*, pp. 60–69, Oct. 1989.

[70] LI, L. and OFFUTT, A. J., "Algorithmic analysis of the impact of changes to object-oriented software," in *Proceedings of IEEE International Conference on Software Maintenance*, pp. 171–184, Nov. 1996.

[71] MORELL, L., "A Theory of Fault-Based Testing," *IEEE Transactions on Software Engineering*, 16(8):844-857, Aug. 1990.

[72] MYERS, G. J., *The Art of Software Testing, 2nd Edition*. Wiley, June 2004.

[73] NANDA, M. G. and SINHA, S., "Accurate interprocedural null-dereference analysis for Java," in *Proceedings of IEEE/ACM International Conference on Software Engineering*, pp. 133–143, May 2009.

[74] NTAFOS, S. C., "On required element testing," *IEEE Transactions on Software Engineering*, 10(6):795–803, Nov. 1984.

[75] ORSO, A., APIWATTANAPONG, T., and HARROLD, M. J., "Leveraging field data for impact analysis and regression testing," in *Proceedings of ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pp. 128–137, Sept. 2003.

[76] ORSO, A., APIWATTANAPONG, T., LAW, J. B., ROTHERMEL, G., and HARROLD, M. J., "An empirical comparison of dynamic impact analysis algorithms," in *Proceedings of IEEE/ACM International Conference on Software Engineering*, pp. 491–500, May 2004.

[77] ORSO, A., SHI, N., and HARROLD, M. J., "Scaling regression testing to large software systems," in *Proceedings of ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pp. 241–252, Nov. 2004.

[78] PERSON, S., DWYER, M. B., ELBAUM, S., and PĂSĂREANU, C. S., "Differential symbolic execution," in *Proceedings of ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pp. 226–237, Nov. 2008.

[79] PODGURSKI, A. and CLARKE, L., "A formal model of program dependences and its implications for software testing, debugging, and maintenance," *IEEE Transactions on Software Engineering*, vol. 16, no. 9, pp. 965–979, 1990.

[80] QI, D., ROYCHOUDHURY, A., and LIANG, Z., "Test generation to expose changes in evolving programs," in *Proceedings of IEEE/ACM International Conference on Automated Software Engineering*, pp. 397–406, Sept. 2010.

[81] RAJLICH, V. T., "A model for change propagation based on graph rewriting," in *Proceedings of IEEE International Conference on Software Maintenance*, pp. 84–91, Sept. 1997.

[82] RAPPS, S. and WEYUKER, E. J., "Selecting software test data using data flow information," *IEEE Transactions on Software Engineering*, vol. 11, no. 4, pp. 367–375, 1985.

[83] REN, X., SHAH, F., TIP, F., RYDER, B. G., and CHESLEY, O., "Chianti: a tool for change impact analysis of java programs," in *Proceedings of ACM Conference on Object Oriented Programming Systems, Languages, and Applications*, pp. 432–448, Oct. 2004.

[84] REPS, T., HORWITZ, S., and SAGIV, M., "Precise interprocedural dataflow analysis via graph reachability," in *Proceedings of ACM Symposium on Principles of Programming languages*, pp. 49–61, 1995.

[85] RICHARDSON, D. and THOMPSON, M. C., "The RELAY model of error detection and its application," in *Proceedings of Workshop on Software Testing, Analysis and Verification*, pp. 223–230, July 1988.

[86] RICHARDSON, D. J. and THOMPSON, M. C., "An analysis of test data selection criteria using the RELAY model of fault detection," *IEEE Transactions Software Engineering*, vol. 19, pp. 533–556, June 1993.

[87] ROTHERMEL, G. and HARROLD, M. J., "Selecting tests and identifying test coverage requirements for modified software," in *Proceedings of International Symposium on Software Testing and Analysis*, pp. 169–184, Aug. 1994.

[88] ROTHERMEL, G. and HARROLD, M. J., "A safe, efficient regression test selection technique," *ACM Transactions on Software Engineering and Methodology*, vol. 6, pp. 173–210, Apr. 1997.

[89] ROTHERMEL, G., HARROLD, M. J., and DEDHIA, J., "Analyzing regression test selection techniques," vol. 22, pp. 529–551, Aug. 1996.

[90] ROTHERMEL, G., UNTCH, R., CHU, C., and HARROLD, M., "Test Case Prioritization," *IEEE Transactions on Software Engineering*, 27(10):929–948, Oct. 2001.

[91] SABLE RESEARCH GROUP. Soot Analysis Framework., Apr. 2012. `http://www.sable.mcgill.ca/soot`. McGill University.

[92] SANTELICES, R., CHITTIMALLI, P. K., APIWATTANAPONG, T., ORSO, A., and HARROLD, M. J., "Test-suite augmentation for evolving software," in *Proceedings of IEEE/ACM International Conference on Automated Software Engineering*, pp. 218–227, Sept. 2008.

[93] SANTELICES, R. and HARROLD, M. J., "Efficiently monitoring data-flow test coverage," in *Proceedings of IEEE/ACM International Conference on Automated Software Engineering*, pp. 343–352, Nov. 2007.

[94] SANTELICES, R. and HARROLD, M. J., "Exploiting program dependencies for scalable multiple-path symbolic execution," in *Proceedings of International Symposium on Software Testing and Analysis*, pp. 195–206, July 2010.

[95] SANTELICES, R. and HARROLD, M. J., "Probabilistic slicing for predictive impact analysis." *Technical Report GIT-CERCS-10-10, CERCS, Georgia Institute of Technology*. 10 pages. `http://hdl.handle.net/1853/36917`, Nov. 2010.

[96] SANTELICES, R. and HARROLD, M. J., "Applying aggressive propagation-based strategies for testing changes," in *Proceedings of Fourth IEEE International Conference on Software Testing, Verification and Validation*, pp. 11–20, Mar. 2011.

[97] SANTELICES, R., HARROLD, M. J., and ORSO, A., "Precisely detecting runtime change interactions for evolving software," in *Proceedings of IEEE International Conference on Software Testing, Verification and Validation*, pp. 429–438, Apr. 2010.

[98] SANTELICES, R., JONES, J. A., YU, Y., and HARROLD, M. J., "Lightweight fault localization using multiple coverage types," in *Proceedings of IEEE/ACM International Conference on Software Engineering*, pp. 56–66, May 2009.

[99] SEN, K., MARINOV, D., and AGHA, G., "CUTE: A concolic unit testing engine for C," in *Proceedings of ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pp. 263–272, Sept. 2005.

[100] SIEGEL, S. F., MIRONOVA, A., AVRUNIN, G. S., and CLARKE, L. A., "Combining symbolic execution with model checking to verify parallel numerical programs," *ACM Transactions Software Engineering and Methodology*, vol. 17, pp. 1–34, Apr. 2008.

[101] SINHA, S., HARROLD, M. J., and ROTHERMEL, G., "Interprocedural control dependence," *ACM Transactions Software Engineering and Methodology*, vol. 10, no. 2, pp. 209–254, 2001.

[102] SNELTING, G., ROBSCHINK, T., and KRINKE, J., "Efficient path conditions in dependence graphs for software safety analysis," *ACM Transactions Software Engineering and Methodology*, vol. 15, pp. 410–457, Oct. 2006.

[103] SRIVASTAVA, A. and THIAGARAJAN, J., "Effectively prioritizing tests in development environment," in *Proceedings of International Symposium on Software Testing and Analysis*, pp. 97–106, July 2002.

[104] TAGHDIRI, M., SEATER, R., and JACKSON, D., "Lightweight extraction of syntactic specifications," in *Proceedings of ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pp. 276–286, Nov. 2006.

[105] THOMPSON, M. C., RICHARDSON, D. J., and CLARKE, L. A., "An information flow model of fault detection," in *Proceedings of International Symposium on Software Testing and Analysis*, pp. 182–192, July 1993.

[106] VALLÉE-RAI, R., CO, P., GAGNON, E., HENDREN, L., LAM, P., and SUNDARESAN, V., "Soot - a java bytecode optimization framework," in *Proceedings of Conference of the Centre for Advanced Studies on Collaborative Research*, p. 13, IBM Press, 1999.

[107] VISSER, W., HAVELUND, K., BRAT, G., PARK, S., and LERDA, F., "Model checking programs," *Automated Software Engineering*, vol. 10, no. 2, pp. 203–232, 2003.

[108] VISSER, W., PĂSĂREANU, C. S., and KHURSHID, S., "Test input generation with java pathfinder," in *Proceedings of International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 97–107, Mar. 2004.

[109] VOAS, J., "PIE:A Dynamic Failure-Based Technique," *IEEE Transactions on Software Engineering*, 18(8):717–727, Aug. 1992.

[110] WALPOLE, R. E., MYERS, R. H., MYERS, S. L., and YE, K. E., *Probability and Statistics for Engineers and Scientists (9th Edition)*. Prentice Hall, Jan. 2011.

[111] WEISE, D., CREW, R. F., ERNST, M. D., and STEENSGAARD, B., "Value dependence graphs: Representation without taxation," in *Proceedings of ACM Symposium on Principles of Programming Languages*, pp. 297–310, Jan. 1994.

[112] WEISER, M., "Program slicing," *IEEE Transactions on Software Engineering*, vol. 10, no. 4, pp. 352–357, 1984.

[113] WLOKA, J., RYDER, B., TIP, F., and REN, X., "Safe-commit analysis to facilitate team software development," in *Proceedings of IEEE/ACM International Conference on Software Engineering*, pp. 507–517, May 2009.

[114] WONG, W. E., HORGAN, J. R., LONDON, S., and MATHUR, A. P., "Effect of test set minimization on fault detection effectiveness," in *Proceedings of IEEE/ACM International Conference on Software Engineering*, pp. 41–50, Apr. 1995.

[115] XIE, T. and NOTKIN, D., "Checking inside the black box: Regression testing by comparing value spectra," *IEEE Transactions Software Engineering*, vol. 31, pp. 869–883, Oct. 2005.

[116] XIE, T., TILLMANN, N., DE HALLEUX, P., and SCHULTE, W., "Fitness-guided path exploration in dynamic symbolic execution," in *Proceedings of IEEE/IFIP International Conference on Dependable Systems and Networks*, June 2009.

[117] XU, Z., KIM, Y., KIM, M., COHEN, M., and ROTHERMEL, G., "Directed test suite augmentation: techniques and tradeoffs," in *Proceedings Symposium Foundations Software Engineering*, pp. 257–266, Nov. 2010.

[118] XU, Z. and ROTHERMEL, G., "Directed test suite augmentation," in *Asia-Pacific Software Engineering Conference*, pp. 406–413, Dec. 2009.

VITA

Raúl Andrés Santelices Ahués was born May 22, 1975 in Santiago, Chile. He attended Colegio Antártica Chilena for primary and middle school and Colegio San Pedro Nolasco for high school. In 2000 he simultaneously received the Engineer in Computer Science and M.S. in Computer Science degrees from Pontificia Universidad Católica de Chile. Between 1997 and 2005, he worked as a software engineer and, later, as a chief software architect in the educational, mobile, and videogames industries. Between 2003 and 2005, he developed and taught a Software Architecture course at Pontificia Universidad Católica de Chile, after which he joined the Ph.D. program at the Georgia Institute of Technology under the advisement of Dr. Mary Jean Harrold. In August 2011, Raúl joined the academic faculty of the University of Notre Dame as an Assistant Professor in the Computer Science and Engineering department.