Change-Effects Analysis for Evolving Software

Raul Santelices, Yiji Zhang, Haipeng Cai and Siyuan Jiang
University of Notre Dame, USA
Contact e-mail: rsanteli@nd.edu

Abstract

Software constantly changes during its life cycle. This phenomenon is particularly prominent in modern software, whose complexity keeps growing and changes rapidly in response to market pressures and user demands. At the same time, developers must assure the quality of this software in a timely manner. Therefore, it is of critical importance to provide developers with effective tools and techniques to analyze, test, and validate their software as it evolves.

While techniques for supporting software evolution abound, a conceptual foundation for understanding, analyzing, comparing, and developing new techniques is also necessary for the continuous growth of this field. A key challenge for many of these techniques is to accurately model and compute the effects of changes on the behavior of software systems. Such a model helps understand, compare, and further advance important activities such as change-impact analysis, regression testing, test-suite augmentation, and program merging.

To address this challenge, in this chapter, we describe our progress and those of other researchers in developing and studying a foundational approach called *change-effects analysis*. This kind of analysis computes, in an ideal setting (i.e, given unlimited resources), all the differences that a change in the code of a program can cause on every element (e.g., statement) of that program. These differences include not only *which* program elements are affected by a change, but also *how* exactly their *behavior* (i.e., execution frequency and states) is affected.

Naturally, computing all possible effects of changes is an undecidable problem. However, such a model for computing change effects can be the basis for practical approaches that approximate those effects. In this chapter, we describe various such approximations for change-effects analysis and their application to two key tasks in software evolution: change-impact analysis and change testing.

Keywords: software evolution; change-effects analysis; change-impact analysis; regression testing; test-suite augmentation; program merging.

1 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 might lead to undesired duplicate items. For another example, adding minimum-age constraints to a banking system might prevent children from monitoring their accounts even if only parents can perform sensitive operations. Hence, developers must understand the consequences and risks of modifying each part of a software system even before they can properly design and test their changes. Then, after modifying the software, developers must check the correctness of their changes by identifying and checking the actual effects of those changes on the behavior of the program. Developers can use program-analysis techniques [2] to those ends.

The class of techniques that searches for the effects of changes in software is called *change-impact analysis* [5, 7, 9, 13, 14, 23, 40, 44, 49, 50, 58, 61, 63]) (or, simply, *impact analysis*). Impact analyses identify the parts of the software that are potentially or actually affected by a change. We classify these techniques into two categories: predictive and descriptive. *Predictive* impact analyses (e.g., [5, 13, 14]) identify potential effects of changing a particular program location, such as a a statement or method, *before* changing it. This is particularly useful to assist developers before they decide to change that location, how to change it, and how to *propagate* that change [60] (i.e., which other changes to make down the line as a consequence). *Descriptive* impact analyses (e.g., [61, 63, 79]), in contrast, identify affected entities *after* changes are made by analyzing the differences in code and behavior observable between the unmodified and modified software.

After changes are made, to validate them, developers typically perform *regression testing*—re-testing a program after it is modified [8, 11, 25, 28, 29, 36, 38, 42, 51, 67, 70, 93]. 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., [16, 70]), (2) prioritizing test cases to find errors earlier (e.g., [68, 85]), and (3) reducing the size of the test suite according to some criterion (e.g., [27, 92]). 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.

A key challenge for many of these techniques is to accurately model and compute the effects of changes on the behavior of evolving software systems. Fundamental models help understand, compare, and further advance change-impact analysis, regression testing, and other software-evolution tasks, such as test-suite augmentation [57, 73, 78] and program merging [12, 46, 79].

In this chapter, we describe our progress and those of other researchers in developing and studying a foundational approach called *change-effects analysis* [72]. This kind of analysis computes, in an ideal setting (i.e, given unlimited resources), all the differences that a change in the code of a program can cause on every element (e.g., statement) of that program. These differences include not only *which* program elements are affected by a change, but also *how* exactly their *behavior* (i.e., execution frequency and states) is affected.

Naturally, computing all possible effects of changes is an undecidable problem. However, such a model for computing change effects can be the basis for practical approaches that approximate those effects. In this chapter, we describe various such approximations for change-effects analysis and their application to two key tasks in software evolution: change-impact analysis and change testing. Our conclusion is that the effects of changes on software behavior can be computed with enough precision to help testers analyze the consequences of changes and reuse and augment their test suites effectively.

The rest of this chapter is organized as follows. Section 2 provides necessary background concepts. Section 3 gives an overview of approaches by other researchers related to change-effects analysis, including briefs reviews of change-impact analysis and test-suite augmentation. Section 4 presents the foundations of change-effects analysis, including a procedure for finding all such effects in an ideal setting of infinite resources. Sections 6–8 describe practical approximations applied to test-suite augmentation for evolving software, change-impact analysis, and change interaction. Finally, Section 9 concludes and Section 10 discusses future directions.

2 Background

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) builds graph representations of all possible sequences of statements in program executions. Control- and data-dependence (Section 2.2) analyses identify the dependencies among statements that arise from control decisions and computations in the program. Slicing (Section 2.3) uses these dependencies to identify the subset of a program or execution that affects or is affected by a certain control decision or computation. Symbolic execution (Section 2.4) 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 Control Flow Analysis

In this chapter, we define a statement as 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 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 [3], 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

[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.

```
int Bin(int a, int b)
pre-condition: a,b ∈ [1,10]
01:  if (a <= 2) // C1 → if (a > 2)
02:      b = b + 1
03:  else
04:      b = b - 1
05:  // C2 → b = b * 2
06:  if (b > 2)
07:      return 1
08:  else
09:      return 0
```

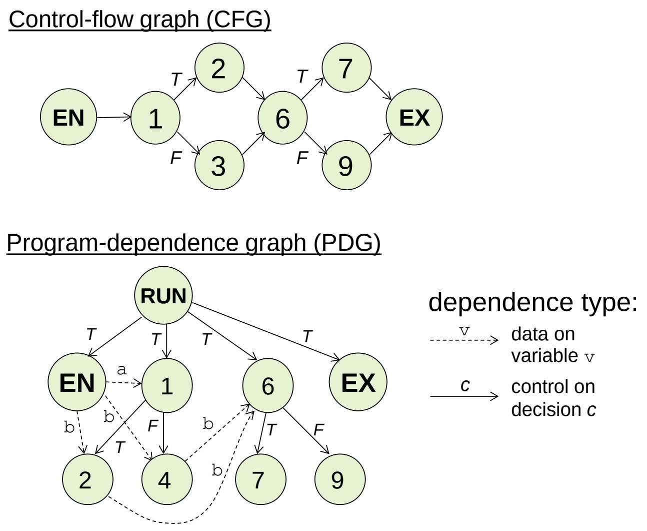

Figure 1: Example program `Bin` with two possible changes. To the right, its CFG and PDG.

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

To illustrate, consider the single-procedure program `Bin` in Figure 1 and its CFG graph in that figure on the top right. `Bin` 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 *EN* represents the entry to the procedure and node *EX* 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* [64] and *interprocedural* control-flow graphs (ICFG) [83]. In this chapter, 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 $E^P = (E_{cfgs} - E_{c,r}) \cup E_{inter}$, where

- E_{cfgs} is the union of all edge sets of the CFGs of all procedures of P,
- $E_{c,r}$ is the set of all edges connecting call nodes to return nodes, and
- E_{inter} is the set of all call edges (c, EN_p) and return edges (EX_p, r) 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 [83]. 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 (c, EN_p) is matched by its corresponding return edge (EX_p, r).

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.

2.2 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 [21].

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 (x, y) or (x, y, L) (the latter form if the decision matters), where y is control dependent on x for decision L. To illustrate, consider program `Bin` in Figure 1 and its CFG on the top right in that figure. In this program, statement 2 is control dependent on statement 1 for decision T, written as $(1, 2, T)$.

Dependencies created by accesses to program variables are called *data* dependencies [3].

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 (x, y) or (x, y, v), where statement y is data-dependent on statement x for variable v. For example, in Figure 1, statement 6 is data dependent on statement 2 for variable b, represented as $(2, 6, b)$.

Podgurski and Clarke defined *semantic* dependence [55] 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*) [21] 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 *RUN*. 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 *RUN* to n.

Figure 1 on the bottom right shows the PDG for the example program `Bin`, 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*) [83] for a program P is the PDG for P without the data-dependence edges.

2.3 Slicing

This section describes *slicing* from the perspective of the potential effects of changes on the rest of the program. Figure 2 on the left shows the example program `Arr` used in this section to illustrate slicing, including four proposed changes. Figure 2 on the right shows the PDG for `Arr` after those changes are made. This program inputs an integer `a` and an array of integers `b`, and initializes `v` to `a` at statement 2. In the loop of statements 3–8, the program iteratively determines the value of `j`—depending on the value of `b[i]`—and increments `v` by `j`. Finally, at statement 10, the program returns *true* if `v > 0` or *false* 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.

DEFINITION 11. The *static forward slice* [30, 31, 90] 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 `Arr` and its PDG in Figure 2. Table 1, in its second column, shows the static forward slices for three different nodes in `Arr`. For example, the static forward slice from node 2 is the set $\{2, 8, 10\}$. This slice is the same before and after the changes.

```
boolean Arr(int a, int[] b)
pre-condition: b.length ≥ 4
01:  int v, i, j
02:  v = a       // C1  → v = a - 1
03:  for (i = 1; i <= 4; ++i) {
04:     if (b[i] > 0)
05:        j = 4   // C2  → j = 5
06:     else
07:        j = 2
08:     v += j      }
09.  /* nothing */  // C3  → print j
10:  return v > 0  // C4  →  return v > -10
```

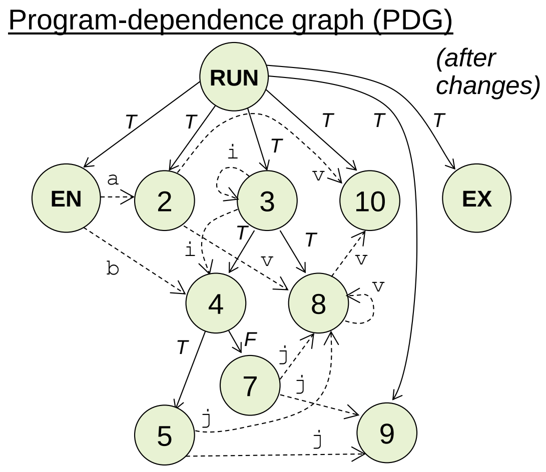

Figure 2: Example program `Arr` on the left with four possible changes and its PDG on the right.

Table 1: Example forward program slices from program `Arr` after applying changes in Figure 2.

slicing criterion	**static slice**	**input values**	**dynamic slice from first statement occurrence**
statement 2	{2, 8, 10}	`a=5, b=[1,-1,0,3]`	$\{2^1, 8^1, 8^2, 8^3, 8^4, 10^1\}$
statement 5	{5, 8, 9, 10}	`a=5, b=[1,-1,0,3]`	$\{5^1, 8^1, 8^2, 8^3, 8^4, 9^1, 10^1\}$
statement 10	{10}	`a=5, b=[1,-1,0,3]`	$\{10^1\}$

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* [10, 30, 31] from a statement s for an execution `Bin` 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 `Bin`.

In this chapter, 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 last column in Table 1 shows the dynamic forward slice for the first occurrence of the corresponding statement (first column) in the execution of the modified `Arr` on input {`a=5, b=[1,-1,0,3]`} (third column). (Note that, for the modified `Arr` and this input, the execution history of the *executable* statements—excluding 1 and 6—is $2^1,3^1,4^1,5^1,8^1,3^2,4^2,7^1,8^2,3^3,4^3,7^2,8^3,3^4,4^4,5^2,8^4,9^1,10^1$.) For example, the dynamic forward slice for statement occurrence 5^1 is the set $\{5^1,8^1,8^2,8^3,8^4,9^1,10^1\}$.

Table 2: Symbolic execution of program `Bin` from Figure 1

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

2.4 Symbolic Execution

Symbolic execution [17, 35] 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 `Bin` in Figure 1. 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 row represents a path condition and its corresponding symbolic state.

3 Related Approaches

Before formally presenting change-effects analysis, we give an overview of approaches that, without being defined in terms of change-effects analysis, have paved the way for this kind of analysis or implicitly use it. Section 3.1 discusses fault-propagation models that describe the conditions under which the effects of faults (or changes) propagate through code. Section 3.2 summarizes the concepts and techniques of change-impact analysis. Then, Section 3.3 discusses regression testing and test-suite augmentation, which also depend on the behavior of changes.

3.1 Fault Propagation Models

A number of researchers have worked on theoretical aspects of fault propagation for *fault-based* testing. Fault-based testing [19,26] 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* [19], which states that developers create programs that are syntactically close to a hypothetical correct version, and the *coupling effect* [19], 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 always known.

Morell [47] 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 [65] 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 highlight one of the main weaknesses of traditional testing criteria based on coverage of program entities (e.g., branches, data dependencies) [66], 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 [87], 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 quite 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 [88]. 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, however, Voas used this model to construct a dynamic (execution-based) 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 to identify the effects on the output of single faults [47, 66, 88] or multiple faults that do not interfere with each other [66]. For changes, however, developers need to determine the effects of changes on any part of the program and possibly do so for multiple changes simultaneously. To the best of our knowledge, no fundamental model exists, other than the one we present next in Section 4, to systematically compute the effects of one or more changes on each part of the program.

3.2 Change-Impact Analysis

Change-impact analysis (CIA) [13] finds all potentially-affected components that may require further changes or that may propagate change effects to other components. Static and dynamic program-analysis-based CIA techniques have been defined at various levels of granularity and precision, such as statements, methods, and classes [5, 13, 40, 54, 59, 63, 79, 94]. These techniques use control flow or dependencies but ignore the states under which changes create an impact. For comprehensive surveys of the literature on CIA, we refer the reader to References [18, 41, 43].

We classify CIA techniques into two categories: predictive and descriptive. *Predictive* impact analyses (e.g., [5, 13, 14]) identify potential effects of changing a particular program location, such as a a statement or method, *before* changing it. This is particularly useful to assist developers before they decide to change that location, how to change it, and how to *propagate* that change [60] (i.e., which other changes to make down the line as a consequence). *Descriptive* impact analyses (e.g., [61, 63, 79]), in contrast, identify affected entities *after* changes are made by analyzing the differences in code and behavior observable between the unmodified and modified software.

Change-impact analysis techniques provide various trade-offs between scalability and precision. The most scalable, coarse-grained impact analyses identify affected classes or methods (e.g., [5, 14, 63]), 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. The most fine-grained impact analyses use code dependencies (e.g., [10, 31]) to identify impacts at the statement level, but they are still 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.

Other approaches to CIA identify bug propensities and co-change trends using software repositories and identifier names [4, 14, 24, 56, 97]. Some of these approaches include non-code compo-

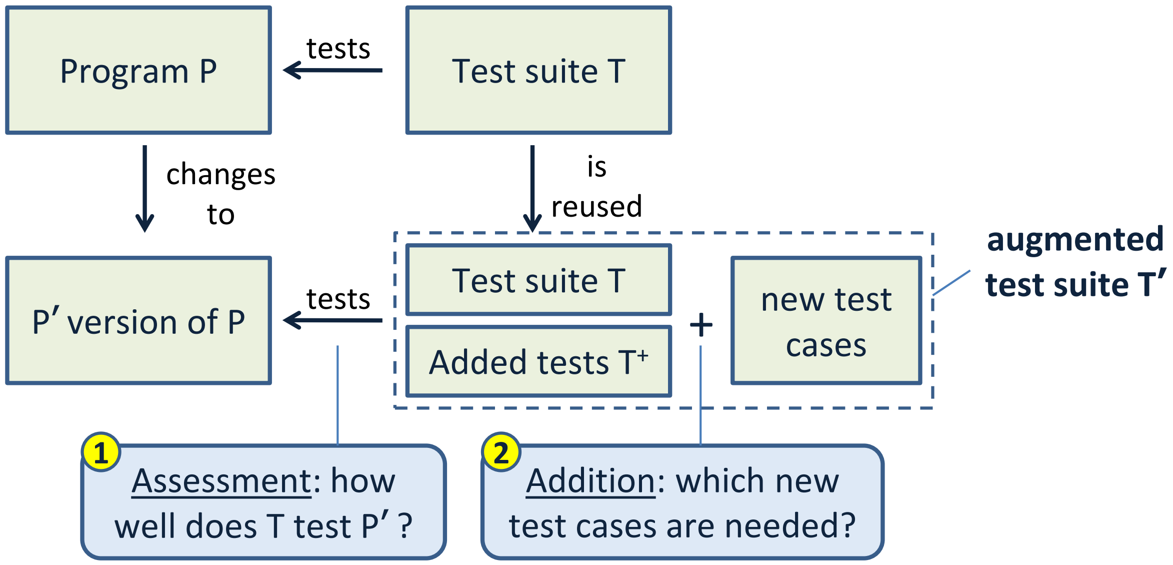

Figure 3: Test-suite augmentation process for evolving software.

nents [82, 91]. On one hand, these approaches give hints about the intents of developers that are hard to obtain via program analysis alone. On the other hand, they work at coarse granularities and still produce both false positives and false negatives.

To account for *state* effects at the statement-level, we developed *change-effects analysis* [73, 75, 77–79] for full precision and, for practical application, approximations based on propagation limits [73, 78] and effect quantification [15, 81]. These are the topics of the next sections. A few other authors have also addressed the state of change effects via summary-based symbolic execution [52, 53] and test-case generation heuristics [57, 86].

3.3 Regression Testing and Test-suite Augmentation

Regression testing is an important form of software testing that consists of re-testing software after it is modified [8, 11, 22, 25, 28, 29, 36, 38, 42, 48, 51, 67, 70, 93] to look for 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. Therefore, as a complement to regression testing, developers need to perform what is called *test suite augmentation* [6, 73], a process that assesses the existing test suite and adds new test cases as needed.

Figure 3 shows the two-step augmentation process in which a program P, tested by test suite T, is modified to obtain version P', for which an augmented test suite T' is obtained. In the first step of this process, developers first assess how well T and the test cases T^+ added during the modification of P test the effects of the changes on P'. Then, in the second step, the developers add any new test cases to $T \cup T^+$ needed to exercise the untested effects of the changes. The mechanism to determine which effects of changes are tested and which are not depend on the specific augmentation technique used and its underlying change-effects analysis.

Most research on regression testing in the literature has focused on *efficiency* aspects that reduce the cost of re-running the test suite (e.g., [16, 27, 70, 92]) and increase the probability of finding regression errors early in the process (e.g., [68, 85]). Research on efficiency includes the analysis of correlations between deviations in *program spectra* (i.e., profiling information) and regression faults (e.g., [29, 93]) to better identify test cases that need to be re-run. Regarding the *effectiveness* of regression testing and, especially, test-suite augmentation, the literature is smaller. Next, in Section 3.3.1, we discuss coverage-based testing techniques for whole programs that inspired early change-specific testing approaches. Then, Section 3.3.2 describes the research on such change-testing approaches published before our own work presented in later sections.

3.3.1 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 [8, 22, 48]. 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 [32, 33, 48] (e.g., statements, branches, paths) are covered, while other criteria require the coverage of all data-flow entities of some kind [8, 22, 39, 62] (e.g., definition-use pairs, chains of data dependencies). Frankl and Weyuker [22] provided a hierarchy of the main control- and data-flow testing criteria that shows the subsumption relationships among these criteria.

3.3.2 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 [11, 25, 69]. 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 [30, 31, 90]. 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 3.3.1). The effectiveness of these change-testing techniques, however, was not evaluated empirically until our work started in Reference [73].

4 Foundations of Change-Effects Analysis

This section presents precise foundations for the automatic analysis of the effects of changes. While being a contribution in itself for the field of software changes, these foundations lie at the core of the applications for change testing and analysis discussed in later sections. Section 4.1 formally defines the effects of a change and Section 4.2 shows how to compute these effects.

4.1 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 4.1.1 presents the example used to illustrate the model, Section 4.1.2 defines what we mean by a change in code, and Sections 4.1.3 and 4.1.4 present the model for individual and multiple changes, respectively.

4.1.1 Working Example

We will reuse our example program `Arr` from Figure 2 with the four changes shown as comments: $\texttt{C}_1$, $\texttt{C}_2$, $\texttt{C}_3$, and $\texttt{C}_4$. Applying these changes to `Arr` produces a modified version that we call `Arr'`. To the right of the program, Figure 2 shows the PDG for `Arr'`. Recall that `Arr` inputs an integer `a` and an array of integers `b`, and initializes `v` to `n` at line 2. In the loop of lines 3–8, the program iteratively determines the value of `v`—depending on the value of `b[j]`—and increments `v` by `j`. Line 8, which is empty in `Arr`, prints `j` in `Arr'`. Finally, at line 10, the program returns *true* if `v > 0` or *false* otherwise. In the modified program `Arr'`, change $\texttt{C}_1$ modifies the expression at line 2 to decrement `v` by 1 with respect to `Arr`, $\texttt{C}_2$ changes the value assigned to `j` at line 5, $\texttt{C}_3$ inserts `print j` at line 8, and $\texttt{C}_4$ modifies the expression evaluated and returned at line 10.

4.1.2 Definition of Code Change

To define the *effects* of a change, in this chapter, first we must define the notion of *change* in the executable code of a program.[2] 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 chapter, and without loss of generality, we assume that the area of the program where a code

[2]By *program*, we mean a syntactically-correct (compilable) sequence of declarations and executable 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[3] 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 $E{\to}V{\times}L_\varepsilon$ 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).

The single-entry property of code changes is needed for the definition of the effects of changes presented in Section 4.1.3. 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) all nodes in G and G', respectively.

To illustrate Definition 14, consider the code change in Figure 4. 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 $\texttt{C}_1$ in `Arr` (Figure 2). Both G and G' for $\texttt{C}_1$

[3]An ICFG is an interprocedural control-flow graph. See Definition 2 in Section 2.1.

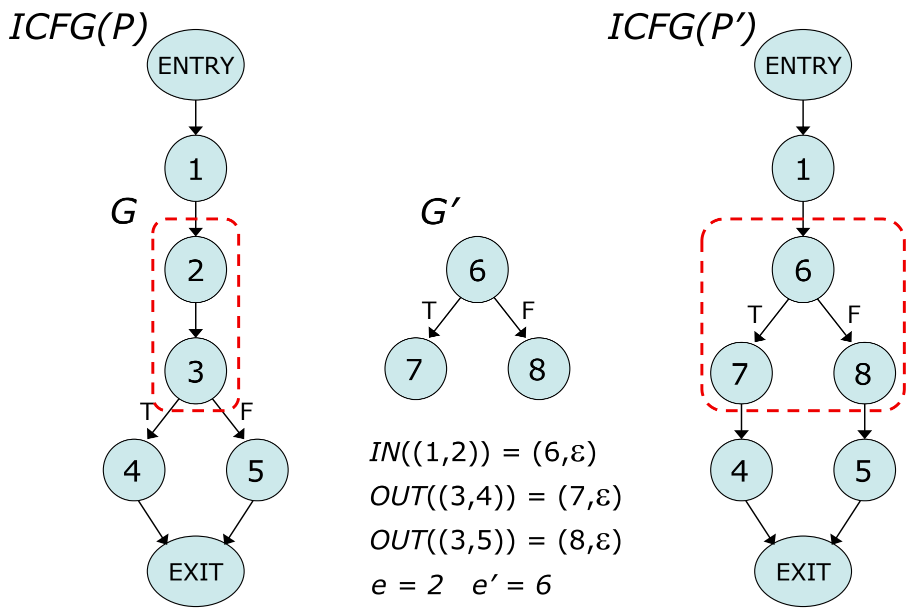

Figure 4: Example of a code change. On the left, the ICFG of program P and its sub-ICFG G (nodes 2 and 3). In the middle, sub-ICFG G', partial functions *IN* and *OUT*, and entry nodes e and e'. On the right, the ICFG of the modified program P'.

consist of node 2, whose corresponding statement is modified. Because node 2 is the only node in the change, both e and e' are node 2. In this particular example, *IN* maps edge (1,2) in G to node 2 in G' with no label and *OUT* maps (2,3) in G to node 2 in G' without label. For another example, $\texttt{C}_3$ in Figure 2 represents the insertion of node 9 before node 10 in `Arr`. In this example, because neither G nor G' can be empty, G can be defined as node 10—even though the corresponding statement is not modified—and G' can be specified by nodes 9 and 10 and edge (9,10). *IN* maps edge (3,10) in `Arr` to node 9 in `Arr′` to indicate that control flows from 3 to 9 instead of 10 after the change and *OUT* maps edge (10,*EX*) to node 10 to preserve that edge. All labels are empty. Node 10 in `Arr` and 9 in `Arr′` correspond to e and e' for this change, respectively.

An example of how Definition 14 handles code areas with multiple entry points is the change resulting from the union of $\texttt{C}_2$ and $\texttt{C}_3$ in `Arr` in Figure 2, which we denote by $\texttt{C}_{\{2,3\}}$. Both G and G' contain nodes 5 and 9 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 (4,5) and (3,10), violating the property that the target of those edges is a unique node e in G and e' in G'. Hence, G and G' must include nodes e and e' that dominate all other nodes in their respective graphs.[4] Therefore, change $\texttt{C}_{\{2,3\}}$ can be made consistent with Definition 14 by having both e and e' be node 3, G consist of nodes {3, 4, 5, 7, 8, 10} and edges {(3,4), (3,10), (4,5), (4,7), (5,8), (7,8)}, and G' consist of nodes {3, 4, 5, 7, 8, 9, 10} and edges {(3,4), (3,9), (4,5), (4,7), (5,8), (7,8), (9,10)}. For this change, *IN* maps edge (2,3) to node 3 with no label and *OUT* maps edge (10,*EX*) to node *EX* with no labels.

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

4.1.3 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).[5] Let P and P' be two versions[6] 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 `Arr'` (Figure 2) and input {`a=5, b=[1,-1,0,3]`}. Let pc denote the program counter after a statement execution. The first five elements in the augmented execution history of `Arr'` are:

$<2^1,\{v{=}4,pc{=}3\}>$, $<3^1,\{i{=}1,pc{=}4\}>$, $<4^1,\{pc{=}5\}>$, $<5^1,\{j{=}5,pc{=}8\}>$, and $<8^1,\{v{=}9,pc{=}3\}>$.

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.

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 2.2 for the definition of semantic dependence.)

For program `Arr'` from Figure 2, executed on input {`a=5, b=[1,-1,0,3]`}, the augmented execution history of change $\texttt{C}_2$ contains three occurrences of the change: 5^1, 5^2, and 5^3. The subset of the augmented execution history for occurrence 5^1 of $\texttt{C}_2$ in this example is:

$<5^1,\{j{=}5,pc{=}8\}>$, $<8^1,\{v{=}9,pc{=}3\}>$, $<8^2,\{v{=}11,pc{=}3\}>$, $<8^3,\{v{=}13,pc{=}3\}>$, $<8^4,\{v{=}18,pc{=}3\}>$, and $<10^1,\{ret{=}true,pc{=}EX\}>$

where *ret* represents the value to be returned. Note that the subsets for 5^2 and 5^3 overlap with the subset for 5^1 on all elements after 5^2 and 5^3, respectively, except for those two occurrences (5^2 and

[5]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.

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

5^3) and 9^1 which is in the augmented execution history for 5^3.

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[7] 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 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, no matter how far from the location of the change they are.

4.1.4 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_x affects another change C_y if the presence of C_x alters in any way the effects of C_y 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_x is *semantically dependent* on a change C_y in program P' if the effects of C_x on P' differ from the effects of that same change on $P'\backslash_{C_y}$ (i.e., P' without the changes in C_y).

In other words, C_x semantically depends on C_y if the presence or absence of C_y on the modified program P' determines which effects C_x has on P'. This definition of semantic change-dependence is similar to the semantic dependence among two statements as defined by Podgurski

[7]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.

and Clarke [55], 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_x between $P'\backslash_{C_y}$ 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_x is semantically dependent on a change C_y, then change C_y is also semantically dependent on change C_x.

The proof of this theorem is provided in Reference [72].

4.2 Computation of the Effects of Changes

Based on the formal model of change effects (Section 4.1) and the principles of the PIE model (Section 3.1), 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[8] listed in Figure 5. 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, for program `Arr` in Figure 2, the execution history of change $\texttt{C}_3$ in `Arr` is the history of line 10 (subgraph G for $\texttt{C}_3$). This history is not modified in `Arr'` and, therefore, it provides no clue on what program behaviors are affected. Only the inserted code for that change—graph G' for $\texttt{C}_3$—specifies that it is the return value 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.

4.2.1 Chain Conditions

The augmented execution history of a change (Definition 17 in Section 4.1.3), a key part of the definition of the effects of a change (Definitions 18 and 19 in Section 4.1.3), uses the 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).

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

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,[9] 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 4.1.3). 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 `Arr` of Figure 2 with and without change $\mathtt{C}_2$, there are four chains from node 5 to node 10: <5, ((5,8), (8,10))>, <5, ((5,8), (8,8), (8,10))>, <5, ((5,8), (8,8), (8,8), (8,10))>, and <5, ((5,8), (8,8), (8,8), (8,8), (8,10))>.

Lines 4–13 and 16–17 of procedure COMPUTEEFFECTS (Figure 5) 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 2.4) of the paths that cover it.[10] 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*.

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 4.2.2) for the chain. Finally, line 17 adds the chain to the work list to be further extended in later iterations of the *while* loop.

4.2.2 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 10 in program `Arr'` is connected to change $\mathtt{C}_1$ in node 2 via node 8 by a chain of data dependencies on variable `v`, but the value returned at node 10 is not affected by $\mathtt{C}_1$ unless the value of `v` that reaches that point is 0 with $\mathtt{C}_1$ and thus 1 without

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

[10] 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 $\in \{ P, P' \}$
(3)     altProgram $:= P'$ if program is $P$ ; $P$ otherwise
(4)     chainWorklist := createEmptyChains (program, change)
(5)     foreach emptyChain $\in$ chainWorklist
(6)        effects[program][emptyChain] := reachingCondition (program, start(emptyChain) )
(7)     endfor
        // incremental computation of conditions
(8)     while chainWorklist $\neq \emptyset$
(9)        prefixChain := pop (chainWorklist)
(10)       foreach dep $\in$ nextDependencies (program, prefixChain)
(11)          chain := append (prefixChain, dep)
              // chain conditions
(12)          depCond := coverCondition (program, prefixChain, dep)
(13)          chainCond := effects[program][prefixChain] $\wedge$ depCond
              // state conditions
(14)          $(S, S_{alt})$ := PSE_Dep (program, altProgram, dep)
(15)          stateCond := $live(S) \neq live(S_{alt}) \vee (pc(S) \wedge \neg\, pc(S_{alt}))$
(16)          effects[program][chain] := chainCond $\wedge$ stateCond
(17)          push (chainWorklist, chain)
(18)       endfor
(19)    endwhile
(20) endfor
(21) return effects
```

Figure 5: Procedure that computes the effects of a change, which does not necessarily terminate.

C_1. 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 5. 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, 73] 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[10] 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[11] 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[12] 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 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*.

4.3 Comparison with Related Approaches

In References [6] and [73], we introduced partial symbolic execution (PSE) and symbolic state differencing as mechanisms for testing the effects of changes. In this section, we formalized the foundations for that work via a comprehensive formal model of the effects of changes that underlie and justify those mechanisms and by specifying in full how those effects can be obtained *programmatically* [72, 73, 75, 79].

[11]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.

[12]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.

The formal model of Section 4.1 is consistent with Morell's theory [47], its extension RELAY [65], and the more practical PIE model [88] described in Section 3.1. Unlike this theory and model, however, our model also provides a complete and 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. Our model also addresses multiple changes and their possibly-interacting effects. The definitions in our model lead directly to the formulation of a general *procedure* for obtaining the effects of changes. From this procedure, practical and effective approximations can be derived, as shown in the next sections.

For testing changes, the techniques discussed in Section 3.3 use slicing to obtain testing requirements corresponding to individual data- and control-dependencies affected by a change. Simple dependencies, however, without accounting for state differences caused by changes, 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. Individual dependencies do not fully reflect the different and complex ways in which dependencies interact to propagate infections.

Meanwhile, change-impact analysis techniques, such as those discussed in Section 3.2, 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.

5 Application: Test-Suite Augmentation Requirements

In recent work [73, 78], we performed the first empirical studies of the change-testing criteria proposed in earlier literature and discussed in Section 3.3.2. The results suggest a low capability to reveal observable behavioral differences caused by changes because these techniques fail 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 3.1), 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 describes the MATRIX technique for propagation-based testing of changes [6,73]. This technique is a practical, straightforward approximation of the formal ideal approach of Section 4 that explicitly addresses the infection and propagation requirements of the PIE model. MATRIX computes and monitors test-suite augmentation requirements for individual changes based on a precise analysis of those changes—unlike the simpler approaches of Section 3.3.2.

6 Application: Test-Suite Augmentation Requirements

In recent work [73, 78], we performed the first empirical studies of the change-testing criteria proposed in earlier literature and discussed in Section 3.3.2. The results suggest a low capability to reveal observable behavioral differences caused by changes because these techniques fail 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 3.1), 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 describes the MATRIX technique for propagation-based testing of changes [6,73]. This technique is a practical, straightforward approximation of the formal ideal approach of Section 4 that explicitly addresses the infection and propagation requirements of the PIE model. MATRIX computes and monitors test-suite augmentation requirements for individual changes based on a precise analysis of those changes—unlike the simpler approaches of Section 3.3.2.

6.1 Overview of MATRIX

The MATRIX technique adapts the procedure COMPUTEEFFECTS of Figure 5 in Section 4.2, 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 4.2.2) 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

Table 3: Dependence chains in program `Bin` and change C_1 from Figure 1.

	chains	**inputs**	**differences**
distance 1	$q_{1,1} = <1, (1, 2)>$	80	16
	$q_{1,2} = <1, (1, 4)>$	20	4
distance 2	$q_{2,1} = <1, (1, 2), (2, 6, y)>$	80	16
	$q_{2,1} = <1, (1, 4), (4, 6, y)>$	20	4
distance 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

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*.

The goal of the technique 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. 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 testing requirements subsume branch and data-dependence coverage criteria (Section 3.3.2) within that distance.

6.2 The MATRIX Technique

This section describes the propagation-based technique MATRIX for analyzing the testing requirements of single changes. Section 6.2.1 presents Phase 1, while Section 6.2.2 presents Phase 2.

6.2.1 Phase 1: Chain Requirements

A simple strategy for testing a change is STMT—cover all changed statements. However, as shown in studies [73], 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 `Bin` in Figure 1 and change C_1. Let $\texttt{Bin}_1$ be this modified version. This program has $10\times10=100$ test inputs. Examining the code in Figure 1 reveals that the change produces a different observable behavior in $\texttt{Bin}_1$ (i.e., a different output with respect to `Bin`) 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. Table 3 shows all dependence chains reaching distances 1, 2, and 3 from the change (Statement 1) in programs `Bin` and $\texttt{Bin}_1$—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 3 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%[13] 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 $\texttt{C}_1$, there are four chains of length 3 reaching output statements 7 and 9. Two of these chains can reveal a difference, as Table 3 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.

The computation of chain requirements for MATRIX 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 6.1. The second difference is that, at Line 17, for COMPUTEEFFECTS, the extended chain is added to the work list only if its length has not yet reached distance d. From now on, we use CHAIN to denote the strategy of satisfying chain requirements during testing.

6.2.2 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 `Bin` and change $\texttt{C}_1$ in Figure 1, Table 3 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 `Bin` might be the same as the state at Statement 7 in $\texttt{Bin}_1$ 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

[13]The probability that T reveals a difference is $1-\prod_{t\in T}(1-p_t)$. p_t is the probability that test t reveals a difference.

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 4.2.2) 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 a path-length 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 1. Using symbolic execution, the technique defines the state of `Bin`$_1$ at the change point as $\{x = x_0, y = y_0\}$. After following chain $q_{3,1}$, the symbolic state in `Bin`$_1$ 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 `Bin`$_1$ and `Bin`, that is, Case 1 for $q_{3,1}$ is not satisfied. However, the path condition in `Bin` 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.

From now on, we use PROP to denote the strategy of satisfying CHAIN (chain requirements) and the state requirements for each such chain during testing.

6.3 Summary of Empirical Results

To assess the potential of MATRIX beyond the simple example of Figure 1, we obtained empirical results for it in comparison with the coverage-based approaches of Section 3.3.2. We implemented MATRIX as an extension of DUA-FORENSICS [74, 80], our dependence-analysis and instrumentation framework. We studied the difference-detection abilities of the MATRIX strategies CHAIN and PROP alongside the coverage-based strategies STMT (changed statements), BR (affected branches), and DU (affected du-pairs) from Section 3.3.2. For CHAIN and PROP, we used the greatest distance d our tool could reach for PROP within a memory constraint of 1 GB and a path-length limit l of 10 branches.

The results are shown in Table 4. We studied 6 changes in Tcas, a small air control program, and 9 changes NanoXML, a lean XML parser. For each change and type of strategy, we generated 100 test suites and measured the probability that the strategy exposes at least one difference in the output. The table omits two changes in NanoXML for which all strategies trivially had 100%

Table 4: Difference-detection rates of test-suite augmentation requirements for various changes.

change	**max distance**	**STMT**	**BR**	**DU**	**CHAIN**	**PROP**
tcas_1	3	41.1%	41.1%	47.3%	47.3%	10.0%
tcas_2	6	15.2%	15.2%	15.2%	16.9%	54.1%
tcas_3	6	2.0%	3.2%	2.0%	18.4%	18.6%
tcas_4	6	1.0%	19.4%	1.0%	10.0%	10.0%
tcas_5	6	0.7%	0.7%	0.7%	3.5%	3.7%
tcas_6	5	2.1%	3.7%	2.1%	4.6%	10.0%
nanoxml_1	2	52.2%	10.0%	52.2%	10.0%	10.0%
nanoxml_2	3	67.2%	67.2%	67.2%	67.2%	67.2%
nanoxml_3	3	0.0%	0.0%	10.4%	10.4%	10.4%
nanoxml_4	3	13.4%	13.4%	48.7%	48.7%	10.0%
nanoxml_5	3	18.7%	18.7%	18.7%	18.7%	18.7%
nanoxml_6	4	35.0%	35.0%	35.0%	66.0%	66.0%
nanoxml_7	4	21.9%	21.9%	21.9%	42.2%	42.2%
average	4.2	21.5%	26.12%	25.49%	41.84%	60.07%

detection. For "non-trivial" changes, somewhat surprisingly, BR did slightly better on average than DU, while the two did better than STMT. The MATRIX strategies, especially PROP, did much better than the coverage-based strategies. This suggests that, while CHAIN is a viable improvement, PROP is the ideal choice with a more than 50% chance on average of finding observable effects of changes. For more details on this study, we refer the reader to Reference [73].

Because of the scalability issues of symbolic execution, which analyzes paths one by one, we developed a more scalable form of symbolic execution called SPD [75] which analyzes multiple paths simultaneously by exploiting the dependence structure of programs to obtain the same results as regular symbolic execution but more efficiently. For the changes in Tcas, using SPD, the effectiveness within our memory limits of PROP increased by 17% whereas, for NanoXML, a variant of SPD that constrains the size of symbolic expressions increased the the effectiveness of PROP by 44%. For full details on SPD and our second study, we refer the reader to Reference [75].

7 Application: Demand-driven Test-Suite Augmentation

Computing all change-testing requirements for a distance d, as in Section 6, gives developers an idea of how adequate a current test suite is and how much additional testing effort is needed. However, in practice, the number of test requirements grows explosively with distance d, which limits the distances to which change-effects propagations can be achieved (whether an effect keeps propagating beyond d is left to chance). Also, if only a small subset of these requirements are satisfied, the computation of the remaining requirements is wasted.

In this section, we present an alternative approach to achieve greater distances to test changes with CHAIN and PROP by identifying satisfied testing requirements on demand during testing instead of computing all testing requirements up-front. Naturally, this comes at the price of not knowing in advance the total number of requirements and not having up-front the propagation constraints that can help generate those test cases. Nevertheless, as we show in this section, a demand-driven alternative of MATRIX can be practical and effective.

7.1 Overview

To circumvent the limitations of the original version of MATRIX, the demand-driven approach for testing changes using the CHAIN and PROP strategies. This approach requires only static and dynamic dependence analysis and runtime state monitoring, in contrast with MATRIX, which requires symbolic execution of all paths that can propagate a change and the enumeration of all dependence chains within a distance limit. Instead of computing all test requirements beforehand, the approach identifies the unsatisfied test requirements that lie at the frontier of the test requirements already satisfied. Although the propagation conditions are no longer expressed symbolically in the approach, these conditions are still used as testing requirements—instead of solving the constraints, testers must ensure the propagation of an infection via a difference in state or a divergence in paths.

Given a test suite *TS*, a *frontier test requirement* for CHAIN is a dependence chain that is covered by *TS* up to, but excluding, its last dependence. For PROP, a frontier test requirement for *TS* is a dependence chain and its propagation conditions such that either the chain is a frontier test requirement for CHAIN or the propagation condition for the last dependence of the chain is not satisfied by *TS*. Consequently, the set of unsatisfied test requirements for test suite *TS* is the union of the frontier test requirements for *TS* and every other test requirement whose chain extends the chain of some frontier test requirement.

To illustrate, we use chain trees to depict the satisfied and frontier test requirements for changes. A *chain tree* is a tree whose root node represents a change and where each other node represents the pair of a dependence and its propagation condition. Each such pair is either part of a satisfied test requirement or the end of a frontier test requirement. A node at the end of a frontier test requirement is called *frontier node* and is a leaf in the chain tree. The root node is annotated with an identifier for the change and every other node is annotated with the corresponding dependence and propagation condition. (For CHAIN, the propagation condition is omitted.) Each edge (u, v) in a chain tree indicates that the dependence and propagation condition for node u (or the change, if u is the root) are followed by the dependence and propagation condition for node v in some test requirement. Therefore, the target PDG node of the dependence for u is the source PDG node of the dependence for v.

Consider, for example, program `Bin` and change `C`$_1$ from Figure 1, and testing strategy CHAIN. Before any test cases execute, the chain tree for `C`$_1$ in `Bin` consists of only one node—the frontier node for change `C`$_1$ in statement 1. Figure 6 shows this chain tree growing as three different test cases are added, one after another. The first tree results from adding test case t_1, which traverses chain ⟨(1,2), (2,5), (5,8)⟩ in `Bin`. The nodes belonging to this satisfied test requirement are shown as shaded solid circles. The frontier nodes, for dependencies (1,4) and (5,6), are shown as unshaded dashed circles. These dependencies, which succeed the change `C`$_1$ and dependence (2,5),

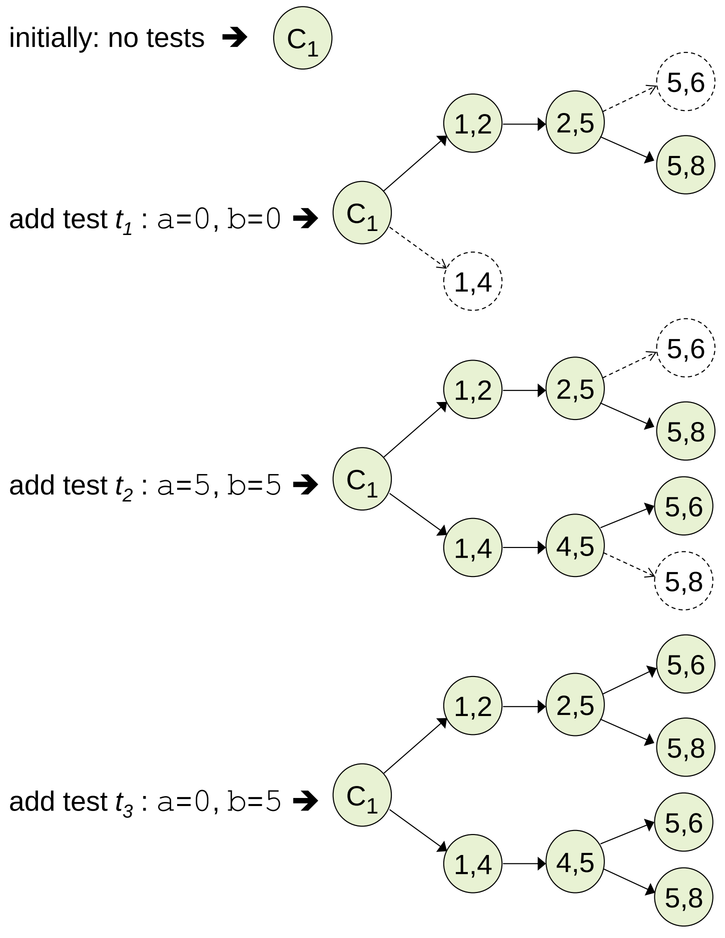

Figure 6: Chain trees for satisfied and frontier test requirements (shaded solid and unshaded dashed circles, respectively) for program `Bin` of Figure 1. In each step, a new test case expands the tree.

respectively, are identified by our approach as the next targets of testing—any chain covered next will have chain ⟨(1,4)⟩ or ⟨(1,2), (2,5), (5,6)⟩ as a prefix. In the middle and on the right, Figure 6 shows the results of adding two more test cases. Test case t_2 expands the chain tree via frontier node (1,4) by covering chain ⟨(1,4), (4,5), (5,6)⟩. Because this chain includes dependence (1,4), (1,4) is no longer a frontier node. Instead, node (5,8) becomes a frontier node that succeeds (4,5) in the tree because, after covering dependence (4,5), t_2 continues to its other successor—dependence (5,6). Then, test case t_3 covers dependence (5,6) after (1,2) and (2,5). Thus, node (5,6) is no longer a frontier node after t_3. Note that, if we used PROP instead of CHAIN, each node would also be annotated with the propagation condition for the corresponding dependence.

The demand-driven approach preserves the precision of propagation-based test requirements while avoiding the cost of symbolic execution and avoiding most of the effort incurred by MATRIX, which performs a full static analysis that can find a large number of test requirements, many of which are not satisfied later. Naturally, using this approach sacrifices the ability to obtain immediately the test requirements that describe the entire space of untested affected behaviors, which might better support the automatic generation of test cases. However, the benefits of this alternative approach justify this trade-off because it can significantly increase the distances reachable for CHAIN and PROP. Moreover, this demand-driven approach removes the burden from the tester of handling a large number of short-distance requirements and, instead, lets the tester focus on requirements that are more likely to be satisfied, possibly at greater distances.

7.2 Demand-Driven Approach to MATRIX

The demand-driven approach for propagation-based test-suite augmentation is described by algorithm SATISFYREQSDEMAND in Figure 7. 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 like MATRIX. 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.

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.[14] 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.

[14]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.

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)  $Satisfied := createEmptyChains\ (P_{ins}, C) \cup createEmptyChains\ (P'_{ins}, C)$
(7)  $totalEffort := 0$
(8)  while $B(totalEffort, Satisfied, TS) = \text{true}$
(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)    $addTest := \text{false}$
(14)    foreach $prefixChain \in Satisfied$
(15)       foreach $dynDep \in extendingDeps(prefixChain, data, data')$
(16)          $chain := append\ (prefixChain, dynDep)$
(17)          if $chain \notin Satisfied$
(18)             $Satisfied := Satisfied \cup chain$
(19)             $addTest := \text{true}$
(20)          endif
(21)          if $propagates(chain, data, data')$
(22)             $markProp\ (chain, Satisfied)$
(23)             $addTest := \text{true}$
(24)          endif
(25)       endfor
(26)    endfor
(27)    $TS := TS \cup (addTest?\ \{t\} : \emptyset)$
(28) endwhile
(29) return $TS$, $Satisfied$
```

Figure 7: Algorithm for demand-driven, propagation-based testing of a change.

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.[15] 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*[16] if the differences in runtime data indicates so (thus satisfying a 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*.

7.3 Comprehensive Study

The demand-driven nature of this approach allowed for a more comprehensive study than for MATRIX. For this study, we used DUA-FORENSICS [80] again and we considered Java subjects for which a large number of test cases are available. Those pools of test cases allowed us to create up to 1000 test suites for each technique studied. Between a number of subjects from the Siemens suite [34] that we translated to Java and three versions of NanoXML [20], we studied a total of 62 changes. For complete details of this study, we refer the reader to References [77] and [78].

For each change, we computed and compared the *difference-detection ability* of each strategy—STMT, BR, DU, CHAIN, and PROP—to produce output differences. Such differences give to developers a view of the behavior of that change. Our metric of quality of the test suite is the ratio of the number of tests detecting differences per test suite to the size of the test suite. Thus, the best possible value of this metric is 1.0.

To support our analysis of the factors affecting the effectiveness of each strategy, we also defined the *detectability* of a change as the difference-detection 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 an output difference. 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.

Table 5 shows the average detection ratios for all strategies, up to distance 10 for CHAIN and PROP. For all changes, a distance of 10 or more was achieved for the latter two strategies. Therefore, distance 10 is the maximum distance for which we obtained results for all changes. In this study, DU was slightly more effective per test than BR, which had no significant improvement over STMT. CHAIN has a consistently greater cost-effectiveness on average than those three coverage-based strategies, although the difference is not great. PROP, however, did considerably better than all other strategies, including CHAIN, for all distances with strong statistical significance. Interestingly, PROP peaks at distance 4 and then decreases to some extent, suggesting that the best

[15]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.

[16]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'*.

Table 5: Average ratio of differences found to test-suite size (*ds-ratio*) per change-testing strategy.

STMT	**BR**	**DU**	**CHAIN**									
			d1	*d2*	*d3*	*d4*	*d5*	*d6*	*d7*	*d8*	*d9*	*d10*
.32	.32	.35	.38	.38	.40	.40	.40	.41	.41	.40	.40	.40

PROP									
d1	*d2*	*d3*	*d4*	*d5*	*d6*	*d7*	*d8*	*d9*	*d10*
.48	.50	.52	.53	.52	.52	.50	.50	.49	.48

Table 6: Difference-detection increase over STMT for all strategies per detectability limit.

detectability limit	**number of changes**	**BR**	**DU**	**CHAIN** (avg. all dist.)	**PROP** (avg. all dist.)
1.0	62	0.0x	0.1x	0.2x	0.6x
.8	48	0.4x	0.4x	1.0x	2.0x
.6	46	0.5x	0.5x	1.3x	2.6x
.4	43	1.0x	0.7x	2.0x	3.8x
.2	39	1.0x	0.8x	2.6x	5.2x
.1	30	1.9x	1.1x	5.0x	11.3x
.05	22	4.9x	2.8x	8.9x	16.3x
.03	18	6.1x	2.8x	10.9x	21.3x
.01	9	5.4x	6.8x	16.2x	34.7x
.005	7	8.0x	12.6x	23.7x	51.0x
.003	4	0.1x	1.7x	3.6x	74.0x
.001	1	0.0x	0.0x	0.0x	334.0x

distribution of test cases can be achieved early. Nevertheless, at distances 10 or greater, more differences in total can be found as the test suite sizes increase.

Another important consideration is the effectiveness of testing with respect to the *difficulty* of testing a change. Table 6 shows the breakdown of our results by the difficulty of testing subsets of all 62 changes. The table 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 0.2 or less).

The next columns in Table 6 indicate 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 results are the average for all distances between 1 and 10.) For example, for the subset of changes with detectability limit 0.6, the table indicates that PROP has a probability 2.6 times greater of finding a difference per test case than STMT. Remember that the detection probability for STMT for those changes is less than or equal to the value in the first column.

The data in this table indicates that, the harder-to-test changes are, the better all techniques do. However, CHAIN and, most of all, PROP, show an even grater superiority over coverage-based strategies for detecting differences on difficult changes, which are arguably the changes in greatest need for a good testing strategy.

In all, PROP produces considerably higher-quality test suites than the other strategies as it discriminates and distributes test cases over the behavior space of changes better than the alternatives. Meanwhile, CHAIN provides a less effective but perhaps simpler alternative to PROP. Moreover, the superiority of propagation-based techniques over coverage-based counterparts for hard-to-test changes is even greater. A Wilcoxon signed-rank test [89] indicates that the superiority of PROP over BR and DU with a p-value lower than 0.01.

8 Application: Quantitative Change-Impact Analysis

A major problem of program analyses is their imprecision, manifested in an excess of *false positives* (false alarms) or *false negatives* (missed targets) or both. Program slicing [90] is one such analysis that is popular but also quite imprecise [79]. Static slicing often reports too many potentially-relevant statements, whereas dynamic slicing—a variant that trades completeness for precision—can also produce many false positives [45, 79]. To increase the precision of static slicing by reducing the size of slices, researchers have proposed combining static and dynamic slicing (e.g., [37]) and pruning slices based on some criterion (e.g., [84, 95]). However, these methods can suffer from false positives and even false negatives. Other attempts to improve this precision include better alias and points-to analyses, which can reduce slice sizes to some extent.

To address the limitations of slicing, especially its static version, we introduce in this section *quantitative slicing* (*q-slicing*) [15,76,81], a general approach for assigning quantities to statements in a slice, such as scores or probabilities. These quantities denote the relevance of each statement in that slice to help users and tools focus their attention first on the parts of the slice that matter the most. For example, for change-impact analysis, q-slicing points users to the most affected areas of the program first so they can take major corrective actions early. Q-slicing can also help tools prioritize regression-testing targets by their likelihood of interaction with changes. These quantities are practical approximations of the state constraints that describe the effects of changes in the foundations presented in Section 4.

As with program analysis in general, we use two approaches to computing quantitative slices (q-slices): a static and a dynamic approach. We have developed one technique of each type. Each technique has its own merits and has the potential to complement the other. The first technique uses a probabilistic model of dependence propagation to analyze the static structure of the program [76]. The second technique applies sensitivity analysis to program executions to dynamically measure the degree to which statements in the program depend on modifications to other statements [15].

In this section, we emphasize the dynamic approach that combines sensitivity analysis and execution differencing and whose results are encouraging. Using our sensitivity-analysis tool, we quantified the forward static slices from various locations. The resulting scores isolated with great precision, among all potentially-impacted statements found by static slicing, the actual impacts observed at runtime after making real changes in those locations.

```
1: for (sign1 = +1; sign1 >= -1; sign1 -= 2) {
2:    c = (r1 - sign1 * r2) / d;
3:    if (c*c <= 1.0) {
4:       h = sqrt(d*d - pow(r1-sign1*r2),2)) / d;
5:       for (sign2=+1; sign2>=-1; sign2-=2) {
6:          nx = vx * c - sign2 * h * vy;
7:          ny = vy * c + sign2 * h * vx;
8:          print x1 + r1 * nx;
9:          print y1 + r1 * ny;
10:         print x2 + sign1 * r2 * nx;
11:         print y2 + sign1 * r2 * ny;   }}}
```

Figure 8: Excerpt from a program that computes the tangents among circles.

The benefits of quantifying slices as practical and effective approximations of the effects of changes are many and can further the adoption of program slicing in development tools. We already found that *early* change-impact analysis, in which users assess the potential impact of changing a location before designing the change, can become highly effective. Quantitative *forward* slicing can also support regression testing, testability analysis, and information-flow analysis. Meanwhile, quantitative *backward* slicing can improve debugging, comprehension, reverse engineering, and parallelization. Even dynamic slicing can benefit from q-slicing, as dynamic slices are mediocre approximations of true runtime dependencies [45, 79].

8.1 Example of Impact Quantification

Consider the code fragment in Figure 8 for finding tangents between circles. The forward static slice from `c` at line 2 contains lines 2–11, which suggests that they might be affected by `c`. However, `c` at line 2 strongly affects `c*c` at line 3 but less strongly affects the branching decision in that line—variations in `c` may or may not flip the branch taken. Therefore, because `c` may or may not affect this decision, the remaining lines are "less affected" than lines 2 and 3. Lines 6–11, however, also use the value of `c`, which makes them "more affected" than lines 4 and 5 (but still less affected than lines 2 and 3).

Q-slicing quantifies these differences among lines in the slice. A q-slicing technique can give, for example, a score of 1.0 to lines 2 and 3, 0.5 to lines 4 and 5, and an intermediate value 0.75 to the rest. The actual scores for these lines will depend on the specific q-slicing technique used to quantify the slice. We present two such techniques next.

8.2 Static Quantification

Static quantification of program slices can be achieved by analyzing the control- and data-flow structure of programs [76, 96]. The advantages of this approach are that no execution data is required and the results represent all behaviors of the program. The latter advantage is quite attractive for tasks like testing because, no matter how many test cases have been created, this approach points users to behaviors not tested yet.

Our static approach for q-slicing uses two key insights:

1. Some data dependencies are less likely to occur than others because the conditions to reach the target from the source of the dependence vary.

2. Data dependencies are more likely to propagate information than control dependencies, yet control dependencies should not be ignored either as in some existing work.

Using the first insight, we created a reachability and alias analysis of the control flow of the program that estimates the probability that the target of a dependence is reached from its source and that both points access the same memory location. Using the second insight, our approach performs another reachability analysis, this time on the dependence graph, that gives a lower but non-zero score to control dependencies.

We estimate the probability that a statement a affects a statement b by computing two components: (1) the probability that a sequence of dependencies from a to b occurs when the program executes and (2) the probability that information flows through that sequence. More details of this approach and early promising results are presented in [76, 96].

8.3 Dynamic Quantification

Given a representative test suite, we can quantify slices via differential execution analysis [79] and sensitivity analysis [71].

8.3.1 Differential Execution Analysis

Differential execution analysis (DEA) is designed specifically for forward slicing from changes to identify the runtime *semantic dependencies* [55] of statements on changes. Semantic dependencies tell which statements are truly affected by which other statements or changes. Although finding semantic dependencies is an undecidable problem, DEA detects such dependencies on changes when they occur at runtime to under-approximate the set of semantic dependencies in the program. Therefore, DEA does not guarantee 100% recall of semantic dependencies but it achieves 100% precision. This is much better than what dynamic slicing normally achieves [45, 79].

DEA works by executing a program before and after the change, collecting the *augmented execution history* [79] of each execution, and then comparing both histories. The execution history of a program for an input is the sequence of statements executed for that input. The *augmented* execution history is the execution history annotated with the values read and written by each statement occurrence. The differences between two such histories reveal which statements had their occurrences or values altered by a change—the conditions for semantic dependence. A formal definition of DEA is given in [79].

8.3.2 Sensitivity Analysis

DEA can be used to quantify *static* forward slices when the change is known—for *post-change* impact analysis. To do this, a DEA-based q-slicing technique can execute the program repeatedly with and without the change for many inputs and find, for each statement, the frequency with which it is impacted by the change. If the inputs are sufficiently representative of the program's behavior, we can use these frequencies as the quantities for the statements in a q-slice.

More generally, however, the specifics of a change might not be known when a user asks for the impacts of modifying a statement or when the slicing task does not involve a change (e.g., debugging, information-flow analysis). For such situations, we created SENSA,[17] a sensitivity-analysis technique and tool for slice quantification and other applications. Sensitivity analysis is used in many fields to determine how modifications to some aspect of a system (e.g., an input) affect other aspects of that system (e.g., the outputs) [71].

We designed SENSA as a generic modifier of program states at given locations, such as changes or failing points. SENSA inputs a program P, a test suite T, and a statement c. For each test case t in T, SENSA executes t repeatedly, replaces each time the value(s) computed by c with a different value, and uses DEA to find which statements were affected by these modifications. With this information for all test cases in T, SENSA computes the sensitivity of each statement s in P to the behavior of c by measuring the frequency with which s is affected by c. These frequencies are the degree of dependence on statement c of all statements s in P, given T.

For a *forward* static slice from statement c in program P, SENSA uses T to quantify the dependence on c of the statements in that slice. For a *backward* static slice from s, SENSA can be used in a similar fashion to quantify the dependence of s on selected statements c from that slice.

SENSA is highly configurable. In addition to parameters such as the number of times to re-run each test case with a different modification (the default is 20), SENSA lets users choose among built-in modification strategies for picking new values for c at runtime. Furthermore, users can add their own strategies. SENSA ensures that each new value picked for c is unique, to maximize diversity while minimizing bias. Whenever a strategy runs out of possible values for a test case, SENSA stops and moves on to the next test case. SENSA offers three modification strategies from which the user can choose:

1. *Random*: A random value is picked within a specified range. By default, the range covers all elements of the value's type except for *char*, for which only readable characters are picked. For some reference types such as *String*, objects with random states are picked. For all other reference types, the strategy currently picks *null*.[18]

2. *Incremental*: A value is picked that diverges from the original value by increments of i (the default is 1.0). For example, for a value v, the strategy first picks $v + i$ and then picks $v - i$, $v + 2i$, $v - 2i$, etc. For common non-numeric types, the same idea is used. For example, for string *foo*, the strategy picks *fooo*, *fo*, *foof*, *oo*, etc.

3. *Observed*: First, the strategy collects all values that c computes when running T on P. Then, the strategy picks iteratively from this pool the new value at c. The goal is to ensure that values remain meaningful to the program.

[17] SENSA is available for download at http://nd.edu/~hcai/sensa/html

[18] We are developing a method to instantiate most types with random states.

Table 7: Results of change-impact prediction using SENSA for q-slicing

Subject	Average impact-prediction cost (% of ranking)				
	ideal	**slicing**	**SENSA-rnd**	**SENSA-inc**	**SENSA-obs**
Schedule1	37.1%	45.2%	38.4%	38.3%	38.4%
NanoXML	7.8%	24.6%	9.9%	10.7%	—
XML-security	5.7%	35.8%	12.0%	12.5%	—

8.3.3 Evaluation

We compared the q-slices produced by SENSA with context-insensitive static forward slices for predicting change impacts at various program locations. The scenario is *early change-impact analysis*, in which users query for the potential impacts of changing a location without necessarily knowing the details of the change yet. Our research questions were:

RQ1: How good is q-slicing at predicting impacts?

RQ2: How good is q-slicing for different usage budgets?

RQ3: How expensive is SENSA for q-slicing?

The first and second questions address the benefits of q-slicing overall and per slice-inspection effort. The third question targets the practicality of SENSA.

8.3.4 Setup

SENSA extends DUA-FORENSICS [74, 80], our analysis and monitoring tool for Java bytecode programs. For this study, we chose three Java subjects from the SIR repository [20] for which many test cases and changes (bug fixes) are provided. These subjects are Schedule1 (a small task scheduler), NanoXML (a lean XML parser), and XML-security (an encryption library of more than 20,000 lines of code). For each subject, we studied 7 changes, for a total of 21 changes.

8.3.5 Methodology

We first applied SENSA and static forward slicing to the locations of the changes to reproduce the scenario in which users query for the consequences that changing those locations would have. Then, for each q-slice given by SENSA, we ranked its statements from greatest to lowest score. For comparison, we also ranked the static slices using Weiser's approach [90] by visiting statements in breadth-first order from the change location and sorting them by increasing visit depth. For tied statements in a ranking, we used as their rank the average of their positions in that ranking.

To assess and compare the predictive power of the rankings given by q-slicing via SENSA and Weiser's slicing, we applied the changes, one at a time, to the corresponding location in its subject. Then, for each change, we used DEA on the unchanged and changed subject to find the statements actually impacted when running all test cases for that subject. Using these actual impacts, we calculated how closely each ranking predicted those impacts. For each ranking and each impacted statement found by DEA, we determined the percentage of the slice that would have to be traversed, in the ranking's order, to reach that statement. We call this percentage the *cost* of

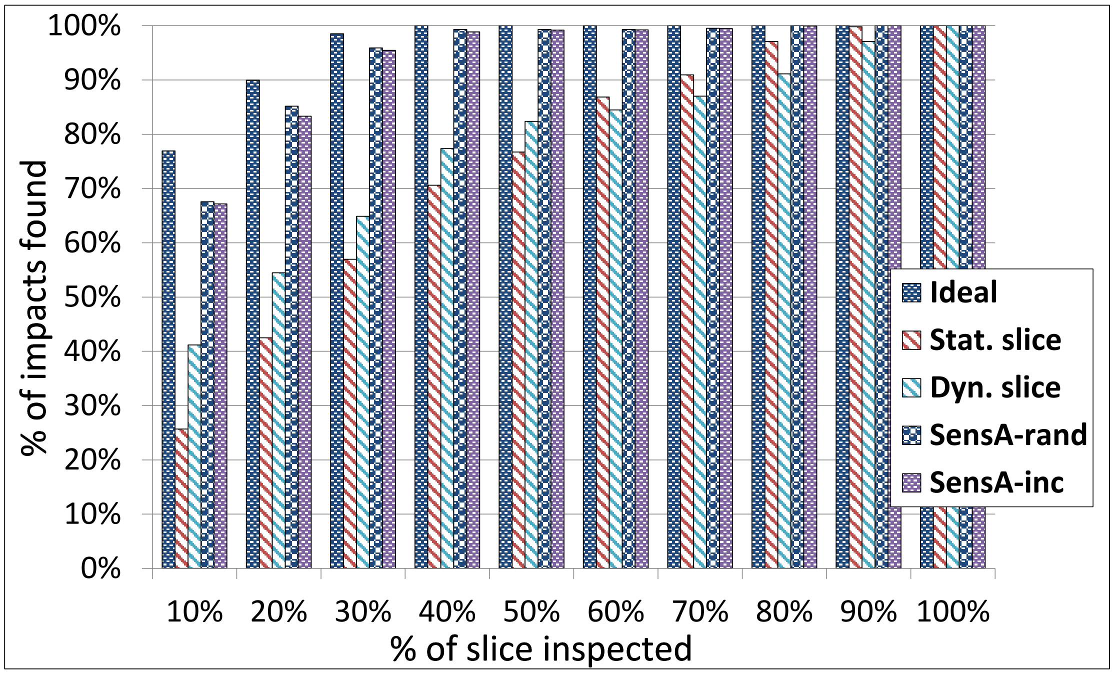

Figure 9: Prediction effectiveness of techniques per inspection effort for NanoXML.

finding an actually-impacted statement using that ranking. Then, we computed the average cost of finding all actual impacts for each ranking—the lower this cost is, the better the technique that produced that ranking is at predicting impacts.

Also, to assess how close to the best possible result each ranking was, we created the *ideal* ranking by placing all actually-impacted statements at the top of that ideal ranking.

8.3.6 Results and Analysis

Table 7 presents the average cost, for each subject and the seven changes in that subject, of five rankings: the *ideal* ranking, the *slicing* ranking (using Weiser's traversal), and the rankings for SENSA and its three strategies: *rnd* (Random), *inc* (Incremental), and *obs* (Observed). For some changes in NanoXML and XML-security, SENSA-*obs* was not applicable as only one value at the change was observed at runtime and, thus, SENSA could not find a different value to replace it. In consequence, we omitted the SENSA-*obs* results for those subjects.

RQ1: The *ideal* cost for Schedule1 in Table 7 reveals that, on average, more than a third of the statements in the slices were *actually* impacted, in contrast with the two other subjects, for which less than 10% of the statements in their slices were impacted. For Weiser's *slicing*, the prediction costs ranged between 25–45%, whereas SENSA was remarkably closer to the ideal predictions at 10–38%. All SENSA strategies exhibited similar costs when applicable. In all, SENSA for q-slicing seems much better overall than simple program slicing (Weiser's traversal) at predicting actual impacts.

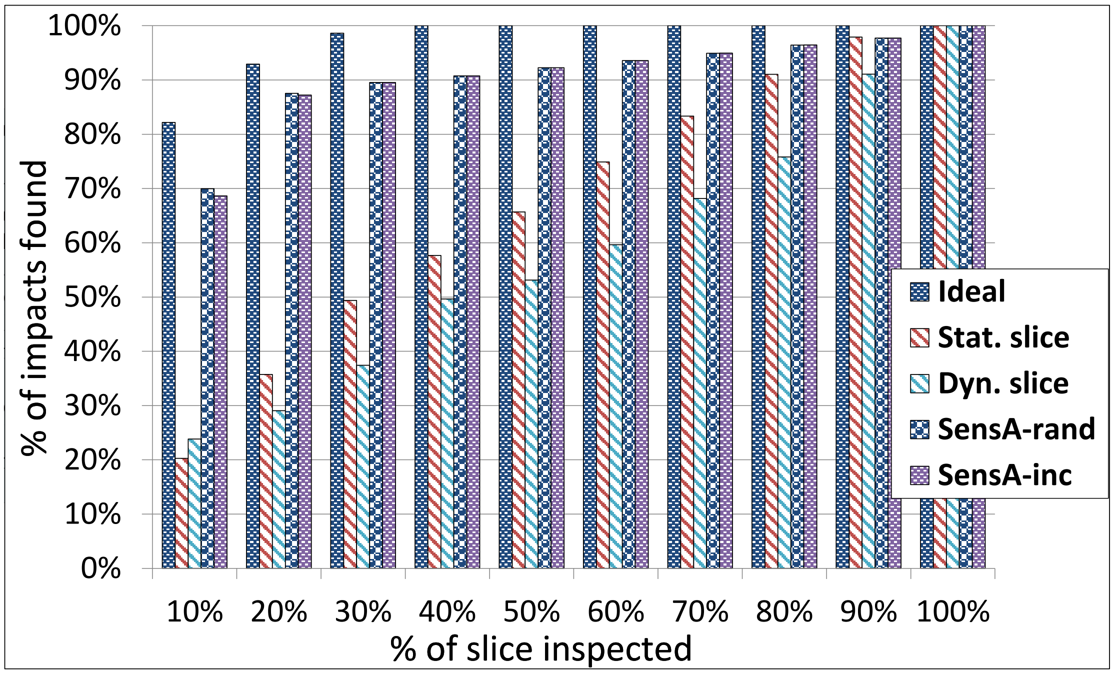

Figure 10: Prediction effectiveness of techniques per inspection effort for XML-security.

RQ2: Because slices can be large, if users cannot explore the entire slice, we expect that they will inspect each ranked slice from the top. The graphs of Figures 9 and 10 for NanoXML and XML-security—the two largest subjects—show, on average for all changes in the subject, the percentage of the actual impacts found (Y axis) for each percentage of the ranked slice inspected (X axis). For example, at 10% of the traversal for NanoXML, on average, the ideal ranking finds 77% of the actual impacts while the *slicing* predictions find only 25–41%. The SENSA predictions, in contrast, find 67–68% of those impacts. For XML-security, the trends and differences are similar. These results suggest that SENSA is not only better overall (RQ1) but is also especially good for the portions of the rankings that users and tools would inspect first.

RQ3: We run our experiments on an Intel Core i5 quad-core machine with 8GB RAM and 64-bit Linux. The average time taken by SENSA for all changes in our subjects ranged from 303 to 493 seconds. The runtime overhead of executing the entire test suites for the instrumented subjects ranged between 32–63%. We consider these costs acceptable. There is also plenty of room for optimizations in our implementation.

9 Conclusion

In this chapter, we addressed the challenge of making the analysis of the effects of changes well-founded to maximize its effectiveness through a principled and cost-conscious approach. At a fundamental level, this chapter 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. This foundational treatment of the nature of changes and their effects on the behavior of programs establishes, to an unprecedented level of detail, what these effects are and how they can be computed. It 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.

At the application level, we presented 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 and providing the means to test those effects on demand. The effects identified by these approximations can serve as change-testing requirements, which we call "propagation-based" because they include the effect-transfer conditions on the program state.

The results for these test-suite augmentation applications indicate that propagation-based testing requirements are significantly more effective for detecting behavioral differences than the testing approaches presented earlier 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.

The effects of changes can also be approximated as quantities denoting the degree of influence of changes on individual lines of code. The resulting technique, quantitative slicing, promises significant increases in the usefulness of program slices. Rather than pruning statements from slices, q-slicing grades statements according to their relevance in a slice. In this chapter, we discussed two approaches to q-slicing, including a novel dynamic technique for computing relevance scores. More techniques can be added under the umbrella of q-slicing. The possibilities are many for quantifying slices in better ways and for improving other applications, such as debugging. Q-slicing is an emerging, rich, and open field.

10 Future Directions

At the foundational level, our definitions in this chapter can lead to new ways to compute 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 code, 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.

The techniques presented in this chapter 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. 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 and, therefore, cost-effective techniques such as those we presented are needed to find as many differences as possible as a requisite for detecting change-related errors.

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.

Our immediate plan for q-slicing is to extend our studies to more subjects and changes of different types and sizes to further generalize our results and characterize the best and worst conditions for the use of q-slicing. We are also developing a visualization for quantified slices to improve our understanding of the approach, to enable user studies, and to support other researchers. Using this tool, we will study how developers take advantage in practice of quantified slices. Slightly farther in the future, we foresee adapting q-slicing to quantify important tasks other than change-impact analysis, such as debugging, comprehension, mutation and interaction testing, and information-flow analysis. More generally, because q-slicing's scores are abstractions of effects on states according to our foundations, these scores can be expanded within this framework to multi-dimensional values or data structures at first and reduced to discrete sets on demand for scalability.

References

[1] Hiralal Agrawal and Joseph R. Horgan. Dynamic Program Slicing. In *Proceedings of ACM Conference on Programming Language Design and Implementation*, pages 246–256, June 1990.

[2] Alfred V. Aho, Monica Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Prentice Hall, September 2006.

[3] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. A.-W., 1986.

[4] Giuliano Antoniol, Vincenzo Fabio Rollo, and Gabriele Venturi. Detecting Groups of Co-changing Files in CVS Repositories. In *Proceedings of International Workshop on Principles of Software Evolution*, pages 23–32, September 2005.

[5] Taweesup Apiwattanapong, Alessandro Orso, and Mary Jean Harrold. Efficient and Precise Dynamic Impact Analysis Using Execute-after Sequences. In *Proceedings of IEEE/ACM International Conference on Software Engineering*, pages 432–441, May 2005.

[6] Taweesup Apiwattanapong, Raul Santelices, Pavan Kumar Chittimalli, Alessandro Orso, and Mary Jean Harrold. Matrix: Maintenance-oriented testing requirement identifier and examiner. In *Proc. of Testing and Academic Industrial Conf. Practice and Research Techniques*, pages 137–146, August 2006.

[7] Robert S. Arnold and Shawn A. Bohner. Impact analysis - towards a framework for comparison. In *Proceedings of IEEE Conference on Software Maintenance*, pages 292–301, September 1993.

[8] Boris Beizer. *Software Testing Techniques*. International Thomson Computer Press; 2nd edition, June 1990.

[9] Keith H. Bennett and Václav T. Rajlich. Software maintenance and evolution: a roadmap. In *Proceedings of Conference on The Future of Software Engineering*, ICSE 2000, pages 73–87, May 2000.

[10] Dave Binkley, Sebastian Danicic, Tibor Gyimothy, Mark Harman, Akos Kiss, and Lahcen Ouarbya. Formalizing executable dynamic and forward slicing. In *Proceedings of Source Code Analysis and Manipulation, Fourth IEEE International Workshop*, pages 43–52, September 2004.

[11] David Binkley. Semantics guided regression test cost reduction. *IEEE Trans. on Softw. Eng.*, 23(8):498–516, August 1997.

[12] David Binkley, Susan Horwitz, and Thomas Reps. Program Integration for Languages with Procedure Calls. *ACM Transactions on Software Engineering and Methodology*, 4(1):3–35, 1995.

[13] Shawn A. Bohner and Robert S. Arnold. *An Introduction to Software Change Impact Analysis*. In Software Change Impact Analysis, Bohner & Arnold, Eds. IEEE Computer Society Press, pages 1–26, June 1996.

[14] Lionel C. Briand, Juergen Wuest, and Hakim Lounis. Using Coupling Measurement for Impact Analysis in Object-Oriented Systems. In *Proceedings of IEEE International Conference on Software Maintenance*, pages 475–482, August 1999.

[15] Haipeng Cai, Siyuan Jiang, Ying jie Zhang, Yiji Zhang, and Raul Santelices. SENSA: Sensitivity Analysis for Quantitative Change-impact Prediction. *Techical Report TR 2013-04*, CSE, U. of Notre Dame, March 2013. 11pp.

[16] Yih-Farn Chen, David S. Rosenblum, and Kiem-Phong Vo. Testtube: A System For Selective Regression Testing. In *Proceedings of IEEE/ACM International Conference on Software Engineering*, pages 211–220, May 1994.

[17] Lori A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Trans. on Softw. Eng.*, 2(3):215–222, September 1976.

[18] A. De Lucia, F. Fasano, and R. Oliveto. Traceability Management for Impact Analysis. In *Frontiers of Software Maintenance at ICSM*, pages 21–30, September 2008.

[19] Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. Hints on Test Data Selection: Help for the Practicing Programmer. *Computer*, 11(4):34–41, 1978.

[20] Hyunsook Do, Sebastian Elbaum, and Gregg Rothermel. Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact. *Empirical Software Engineering*, 10(4):405–435, 2005.

[21] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The Program Dependence Graph and Its Use in Optimization. *ACM Transactions on Program Languages and Systems*, 9(3):319-349, 1987.

[22] Phyllis G. Frankl and Elaine J. Weyuker. An Applicable Family of Data Flow Criteria. *IEEE Transactions on Software Engineering*, 14(10):1483-1498, October 1988.

[23] Keith Brian Gallagher and James R. Lyle. Using program slicing in software maintenance. *IEEE Transactions Software Engineering*, 17(8):751–761, 1991.

[24] Malcom Gethers, Bogdan Dit, Huzefa Kagdi, and Denys Poshyvanyk. Integrated Impact Analysis For Managing Software Changes. In *Proceedings of IEEE/ACM International Conference on Software Engineering*, pages 430–440, May 2012.

[25] Rajiv Gupta, Mary Jean Harrold, and Mary Lou Soffa. An Approach to Regression Testing Using Slicing. In *Proceedings of Conference on Software Maintenance*, pages 299–308, November 1992.

[26] R. G. Hamlet. Testing programs with the aid of a compiler. *IEEE Transactions Software Engineering*, 3(4):279–290, July 1977.

[27] M. J. Harrold, Rajiv Gupta, and Mary Lou Soffa. A methodology for controlling the size of a test suite. *ACM Transactions on Softw. Eng. and Methodology*, 2(3):270–285, July 1993.

[28] Mary Jean Harrold, James A. Jones, Tongyu Li, Donglin Liang, Alessandro Orso, Maikel Pennings, Saurabh Sinha, Stephan A. Spoon, and Ashish Gujarathi. Regression test selection for java software. In *Proc. of ACM Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2001)*, pages 312–326, Tampa Bay, FL, USA, november 2001.

[29] M.J. Harrold, G. Rothermel, K. Sayre, R. Wu, and L. Yi. An empirical investigation of the relationship between spectra differences and regression faults. *Journal of Software Testing, Verification and Reliability*, 10(3):171–194, September 2000.

[30] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 35–46, June 1988.

[31] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural Slicing Using Dependence Graphs. *ACM Transactions on Program Languages and Systems*, 12(1):26-60, January 1990.

[32] William E. Howden. *Tutorial: Software Testing and Validation Techniques*. IEEE Computer Society, 1978.

[33] J. C. Huang. An approach to program testing. *ACM Computing Surveys*, 7(3):113–128, 1975.

[34] Monica Hutchins, Herb Foster, Tarak Goradia, and Thomas Ostrand. Experiments of the Effectiveness of Dataflow- and Controlflow-based Test Adequacy Criteria. In *Proceedings of IEEE/ACM International Conference on Software Engineering*, pages 191–200, 1994.

[35] James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, July 1976.

[36] Bogdan Korel and Ali M. Al-Yami. Automated regression test generation. In *Proceedings of International Symposium on Software Testing and Analysis*, pages 143–152, March 1998.

[37] Jens Krinke. Effects of Context on Program Slicing. *Journal of Systems and Software*, 79(9):1249–1260, 2006.

[38] D. Kung, J. Gao, P. Hsia, Y. Toyashima, and C. Chen. Firewall regression testing and software maintenance of object-oriented systems. *Journal of Object-Oriented Programming*, 1994.

[39] J. W. Laski and B. Korel. A data flow oriented program testing strategy. *IEEE Trans. on Softw. Eng.*, 9(3):347–354, May 1983.

[40] James Law and Gregg Rothermel. Whole Program Path-Based Dynamic Impact Analysis. In *Proceedings of IEEE/ACM International Conference on Software Engineering*, pages 308–318, May 2003.

[41] Steffen Lehnert. A Review of Software Change Impact Analysis. Technical report, Technical Report, Ilmenau University of Technology, Department of Software Systems / Process Informatics, December 2011.

[42] Hareton K. N. Leung and Lee J. White. Insights Into Regression Testing. In *Proceedings of IEEE Conference on Software Maintenance*, pages 60–69, October 1989.

[43] Bixin Li, Xiaobing Sun, Hareton Leung, and Sai Zhang. A survey of code-based change impact analysis techniques. *Journal of Software Testing, Verification and Reliability*, April 2012.

[44] Li Li and A. Jefferson Offutt. Algorithmic analysis of the impact of changes to object-oriented software. In *Proceedings of IEEE International Conference on Software Maintenance*, pages 171–184, November 1996.

[45] Wassim Masri and Andy Podgurski. Measuring the Strength of Information Flows in Programs. *ACM Transactions on Software Engineering and Methodology*, 19(2):1–33, 2009.

[46] Tom Mens. A State-of-the-Art Survey on Software Merging. *IEEE Transactions on Software Engineering*, 28(5):449–462, 2002.

[47] Larry Morell. A Theory of Fault-based Testing. *IEEE Transactions on Softw. Eng.*, 16(8):844-857, August 1990.

[48] Glenford J. Myers. *The Art of Software Testing, 2nd Edition*. Wiley, June 2004.

[49] Alessandro Orso, Taweesup Apiwattanapong, and Mary Jean Harrold. Leveraging field data for impact analysis and regression testing. In *Proc. of 9th European Softw. Eng. Conf. and 10th ACM SIGSOFT Symp. on the Foundations of Softw. Eng.*, pages 128–137, Helsinki, Finland, september 2003.

[50] Alessandro Orso, Taweesup Apiwattanapong, James B. Law, Gregg Rothermel, and Mary Jean Harrold. An empirical comparison of dynamic impact analysis algorithms. In *Proc. of 26th IEEE and ACM SIGSOFT Int'l Conf. on Softw. Eng. (ICSE 2004)*, pages 491–500, Edinburgh, Scotland, may 2004.

[51] Alessandro Orso, Nanjuan Shi, and Mary Jean Harrold. Scaling regression testing to large software systems. In *Proc. of 12th ACM SIGSOFT Symp. on the Foundations of Softw. Eng. (FSE 2004)*, pages 241–252, November 2004.

[52] Suzette Person, Matthew B. Dwyer, Sebastian Elbaum, and Corina S. Păsăreanu. Differential symbolic execution. In *Proc. of Int'l Symp. on Foundations of Softw. Eng.*, pages 226–237, November 2008.

[53] Suzette Person, Guowei Yang, Neha Rungta, and Sarfraz Khurshid. Directed Incremental Symbolic Execution. In *Proceedings of ACM Conference on Programming Language Design and Implementation*, pages 504–515, June 2011.

[54] Maksym Petrenko and Václav Rajlich. Variable Granularity for Improving Precision of Impact Analysis. In *Proceedings of IEEE International Conference on Program Comprehension*, pages 10–19, May 2009.

[55] Andy Podgurski and Lori A. Clarke. A Formal Model of Program Dependences and Its Implications for Software Testing, Debugging, and Maintenance. *IEEE Transactions on Software Engineering*, 16(9):965–979, 1990.

[56] Denys Poshyvanyk, Andrian Marcus, Rudolf Ferenc, and Tibor Gyimóthy. Using Information Retrieval Based Coupling Measures For Impact Analysis. *Empirical Software Engineering*, 14(1):5–32, 2009.

[57] Dawei Qi, Abhik Roychoudhury, and Zhenkai Liang. Test Generation to Expose Changes in Evolving Programs. In *Proceedings of IEEE/ACM International Conference on Automated Software Engineering*, pages 397–406, September 2010.

[58] Václav Rajlich. A model for change propagation based on graph rewriting. In *Proceedings of IEEE International Conference on Software Maintenance*, pages 84–91, September 1997.

[59] Václav Rajlich. MSE: A Methodology for Software Evolution. *Journal of Software Maintenance: Research and Practice*, 9(2):103–124, March 1997.

[60] Václav Rajlich. *Software Engineering: The Current Practice*. Chapman and Hall/CRC, November 2011.

[61] Murali Krishna Ramanathan, Ananth Grama, and Suresh Jagannathan. Sieve: A Tool for Automatically Detecting Variations Across Program Versions. In *Proceedings of IEEE/ACM International Conference on Automated Software Engineering*, pages 241–252, September 2006.

[62] S. Rapps and E. J. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, 11(4):367–375, April 1985.

[63] Xiaoxia Ren, Fenil Shah, Frank Tip, Barbara G. Ryder, and Ophelia Chesley. Chianti: A Tool for Change Impact Analysis of Java Programs. In *Proceedings of ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications*, pages 432–448, October 2004.

[64] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of ACM SIGPLAN-SIGACT Symposium on Principles of programming languages*, pages 49–61, January 1995.

[65] D. Richardson and M. C. Thompson. The RELAY model of error detection and its application. In *Proc. of Workshop on Softw. Testing, Analysis and Verif.*, pages 223–230, July 1988.

[66] Debra J. Richardson and Margaret C. Thompson. An analysis of test data selection criteria using the RELAY model of fault detection. *IEEE Transactions Software Engineering*, 19(6):533–556, June 1993.

[67] G. Rothermel, M. J. Harrold, and J. Dedhia. Analyzing regression test selection techniques. 22(8):529–551, August 1996.

[68] G. Rothermel, R.H. Untch, C. Chu, and M.J. Harrold. Test Case Prioritization. *IEEE Transactions on Softw. Eng.*, 27(10):929–948, 2001.

[69] Gregg Rothermel and Mary Jean Harrold. Selecting tests and identifying test coverage requirements for modified software. In *Proceedings of ACM International Symposium on Software Testing and Analysis*, pages 169–184, August 1994.

[70] Gregg Rothermel and Mary Jean Harrold. A Safe, Efficient Regression Test Selection Technique. *ACM Transactions on Software Engineering and Methodology*, 6(2):173–210, 1997.

[71] Andrea Saltelli, K. Chan, and E. M. Scott. *Sensitivity Analysis*. John Wiley & Sons, March 2009.

[72] Raul Santelices. *Change-effects Analysis for Effective Testing and Validation of Evolving Software*. PhD thesis, Georgia Institute of Technology, Atlanta, Georgia, April 2012.

[73] Raul Santelices, Pavan Kumar Chittimalli, Taweesup Apiwattanapong, Alessandro Orso, and Mary Jean Harrold. Test-suite Augmentation for Evolving Software. In *Proceedings of IEEE/ACM International Conference on Automated Software Engineering*, pages 218–227, September 2008.

[74] Raul Santelices and Mary Jean Harrold. Efficiently Monitoring Data-flow Test Coverage. In *Proceedings of IEEE/ACM International Conference on Automated Software Engineering*, pages 343–352, November 2007.

[75] Raul Santelices and Mary Jean Harrold. Exploiting Program Dependencies for Scalable Multiple-path Symbolic Execution. In *Proceedings of ACM International Symposium on Software Testing and Analysis*, pages 195–206, July 2010.

[76] Raul Santelices and Mary Jean Harrold. Probabilistic slicing for predictive impact analysis. *Techical Report CERCS-10-10*, Georgia Tech, November 2010. 10pp.

[77] Raul Santelices and Mary Jean Harrold. Applying Aggressive Propagation-based Strategies for Testing Changes. In *Proceedings of IEEE International Conference on Software Testing, Verification and Validation*, pages 11–20, March 2011.

[78] Raul Santelices and Mary Jean Harrold. Demand-driven Propagation-based Strategies for Testing Changes. *Journal of Software Testing, Verification and Reliability*, 23(6):499–528, 2013.

[79] Raul Santelices, Mary Jean Harrold, and Alessandro Orso. Precisely detecting runtime change interactions for evolving software. In *Proceedings of IEEE International Conference on Software Testing, Verification and Validation*, pages 429–438, April 2010.

[80] Raul Santelices, Yiji Zhang, Haipeng Cai, and Siyuan Jiang. DUA-Forensics: A Fine-Grained Dependence Analysis and Instrumentation Framework Based on Soot. In *Proceeding of ACM SIGPLAN International Workshop on the State Of the Art in Java Program Analysis*, pages 13–18, June 2013.

[81] Raul Santelices, Yiji Zhang, Siyuan Jiang, Haipeng Cai, , and Ying jie Zhang. Quantitative Program Slicing: Separating Statements by Relevance. In *Proceedings of IEEE/ACM International Conference on Software Engineering – New Ideas and Emerging Results track*, pages 1269–1272, May 2013.

[82] Mark Sherriff and Laurie Williams. Empirical Software Change Impact Analysis using Singular Value Decomposition. In *Proceedings of IEEE International Conference on Software Testing, Verification and Validation*, pages 268–277, April 2008.

[83] Saurabh Sinha, Mary Jean Harrold, and Gregg Rothermel. Interprocedural Control Dependence. *ACM Transactions on Software Engineering and Methodology*, 10(2):209–254, 2001.

[84] Manu Sridharan, Stephen J. Fink, and Rastislav Bodik. Thin slicing. In *Proceedings of ACM Conference on Programming Language Design and Implementation*, pages 112–122, June 2007.

[85] Amitabh Srivastava and Jay Thiagarajan. Effectively Prioritizing Tests In Development Environment. In *Proceedings of ACM International Symposium on Software Testing and Analysis*, pages 97–106, July 2002.

[86] Kunal Taneja, Tao Xie, Nikolai Tillmann, and Jonathan de Halleux. eXpress: Guided Path Exploration for Efficient Regression Test Generation. In *Proceedings of ACM International Symposium on Software Testing and Analysis*, pages 1–11, July 2011.

[87] Margaret C. Thompson, Debra J. Richardson, and Lori A. Clarke. An information flow model of fault detection. In *Proceedings of International Symposium on Software Testing and Analysis*, pages 182–192, July 1993.

[88] Jeffrey M. Voas. PIE: A Dynamic Failure-based Technique. *IEEE Transactions on Software Engineering*, 18(8):717–727, August 1992.

[89] Ronald E. Walpole, Raymond H. Myers, Sharon L. Myers, and Keying E. Ye. *Probability and Statistics for Engineers and Scientists*. Prentice Hall, January 2011.

[90] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.

[91] Elaine J. Weyuker, Thomas J. Ostrand, and Robert M. Bell. Comparing The Effectiveness Of Several Modeling Methods For Fault Prediction. *Empirical Software Engineering*, 15(3):277–295, 2010.

[92] W. E. Wong, J. R. Horgan, S. London, and A. P. Mathur. Effect Of Test Set Minimization On Fault Detection Effectiveness. In *Proceedings of IEEE/ACM International Conference on Software Engineering*, pages 41–50, April 1995.

[93] Tao Xie and David Notkin. Checking Inside the Black Box: Regression Testing by Comparing Value Spectra. *IEEE Transactions on Software Engineering*, 31(10):869–883, 2005.

[94] Lingming Zhang, Miryung Kim, and Sarfraz Khurshid. Localizing Failure-Inducing Program Edits Based On Spectrum Information. In *Proceedings of IEEE International Conference on Software Maintenance*, pages 23–32, September 2011.

[95] Xiangyu Zhang, Neelam Gupta, and Rajiv Gupta. Pruning Dynamic Slices with Confidence. In *Proceedings of ACM Conference on Programming Language Design and Implementation*, pages 169–180, June 2006.

[96] Yiji Zhang and Raul Santelices. Predicting Data Dependences for Slice Inspection Prioritization. In *Proceedings of IEEE International Workshop on Program Debugging*, pages 177–182, November 2012.

[97] Thomas Zimmermann, Andreas Zeller, Peter Weissgerber, and Stephan Diehl. Mining version histories to guide software changes. *IEEE Transactions on Software Engineering*, 31(6):429–445, 2005.